目　录

致谢

akka文档中文翻译

akka文档中文翻译

该文档是Akka官方文档的中文翻译。使用Gitbook制作。

本文大量参考了上海广谈信息技术有限公司的《Akka 2.0文档》。
广谈公益还有很多优秀的资源，如感兴趣请移步参考。

贡献力量

欢迎大家对该文档贡献力量，提出宝贵意见，修改并提交Pull Requests。

如果对scala和akka感兴趣，欢迎加入“无水scala群”：231809997，有各路scala高手帮你释疑。

对贡献的要求是保持术语和代码的一致

- 术语的一致：以《Akka 2.0文档》的术语为准，如果有更好的选择，欢迎提出issue
- 代码的一致：书的内容放在`版本号/语言/`目录下，如`2.3.6/scala/`，生成的html文件放在`版本号/语言/book/`目录下。如非必要，不要提交其他无关内容或生成的内容。

翻译进度与状态

2.3.6 scala

- Introduction
- 引言
  - Akka是什么?
  - 为什么使用Akka?
  - 入门
  - 必修的“Hello World”
  - 用例和部署场景
  - Akka使用实例
- 概述
  - 术语，概念
  - Actor系统
  - 什么是Actor?
  - 监管与监控
  - Actor引用，路径与地址
  - 位置透明性
  - Akka与Java内存模型
  - 消息发送语义
  - 配置
- Actors
  - Actors
  - 类型Actor
  - 容错
  - 调度器
  - 邮箱
  - 路由
  - 有限状态机(FSM)
  - 持久化

- [x] 测试Actor系统
- [x] Actor DSL
- [ ] Futures与Agents
  - [ ] Futures （翻译完成，待校验）
  - [ ] Agents （翻译完成，待校验）
- [ ] 网络
  - [ ] 集群规格
  - [ ] 集群用法
  - [ ] 远程 （翻译完成，待校验）
  - [ ] 序列化 （翻译完成，待校验）
  - [ ] I/O （翻译完成，待校验）
  - [ ] 使用TCP （翻译完成，待校验）
  - [ ] 使用UDP （翻译完成，待校验）
  - [ ] ZeroMQ （翻译完成，待校验）
  - [ ] Camel
- [ ] 实用工具
  - [ ] 事件总线 （翻译完成，待校验）
  - [ ] 日志 （翻译完成，待校验）
  - [ ] 调度器 （翻译完成，待校验）
  - [ ] Duration （翻译完成，待校验）
  - [ ] 线路断路器 （翻译完成，待校验）
  - [ ] Akka扩展 （翻译完成，待校验）
  - [ ] 微内核 （翻译完成，待校验）
- [ ] 如何使用：常用模式 （翻译完成，待校验）
  - [ ] 消息限流
  - [ ] 跨节点平衡工作负载
  - [ ] 工作拉取模式，来限流和分发工作，防止邮箱溢出
  - [ ] 顺序终止

- Akka AMQP代理
- Akka 2中的关闭模式
- 使用Akka做分布式（内存）图像处理
- 案例分析：一个使用Actor的自动更新缓存
- 使用蜘蛛模式发现actor系统的消息流向
- 周期信息调度
- 模板模式

- 实验模块
  - [x] 持久化
  - 多节点测试 （翻译完成，待校验）
  - Actors(使用Java的Lambda支持) not supported
  - FSM(使用Java的Lambda支持) not supported
  - 外部贡献 （翻译完成，待校验）
- Akka开发者信息
  - 构建Akka （翻译完成，待校验）
  - 多JVM测试 （翻译完成，待校验）
  - I/O层设计 （翻译完成，待校验）
  - 开发指南 （翻译完成，待校验）
  - 文档指南 not supported
  - 团队 not supported
- 工程信息
  - 迁移指南 not supported
  - 问题追踪 not supported
  - 许可证 not supported
  - 赞助商 not supported
  - 项目 not supported
- 附加信息
  - 常见问题 （翻译完成，待校验）

- 图书 not supported
- 其他语言绑定 not supported
- Akka与OSGi not supported
- 部分HTTP框架名单 not supported

Introduction

- AKKA 2.3.6 Scala 文档

> 无水*scala*群: `231809997`
> 有各路*scala*高手帮你释疑，语言风格像*scala*一样简洁，精确

AKKA 2.3.6 Scala 文档

该文档是Akka 2.3.6 Scala 文档的中文翻译（github）。

本文大量参考了上海广谈信息技术有限公司的《Akka 2.0文档》，广谈公益还有很多优秀的资源，感兴趣的请移步参考。

引言

引言

    - 通过IntelliJ IDEA使用Akka
    - 通过NetBeans使用Akka
    - 不要使用scala编译器的-optimize选项
    - 从源码编译
    - 需要帮助?
- 必修的“Hello World”
- 用例和部署场景
    - 我该如何使用和部署Akka?
- Akka使用实例
    - 这里是一些将Akka用于生产环境的领域

Akka是什么?

Akka是什么?

可扩展的实时事务处理

我们相信编写出正确的、具有容错性和可扩展性的并发程序太困难了。这多数是因为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，为构建可扩展的、有弹性的响应式并发应用提供了一个更好的平台—详见《响应式宣言》 。在容错性方面我们采用了“let it crash”（让它崩溃）模型，该模型已经在电信行业构建出“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是开源的，可以通过Apache 2许可获得。

可以从 http://akka.io/downloads/ 下载

请注意所有的代码示例都是可编译的，所以如果你想直接获得源代码，

可以查看github的”Akka Docs”子项目——java和scala

Akka实现了独特的混合模型

Actors

Actors为你提供:

- 对并发/并行程序的简单的、高级别的抽象。
- 异步、非阻塞、高性能的事件驱动编程模型。
- 非常轻量的事件驱动处理（1G内存可容纳数百万个actors）。

参阅 Actors (Scala) 和 Actors (Java)

容错性

- 使用“let-it-crash”语义的监控层次体系。
- 监控层次体系可以跨越多个JVM，从而提供真正的容错系统。
- 非常适合编写永不停机、自愈合的高容错系统。

参阅 容错性 (Scala) 和 容错性 (Java)

位置透明性

Akka的所有元素都为分布式环境而设计：所有actor只通过发送消息进行交互，所有操作都是异步的。

集群支持概览请参阅Java和Scala文档相关章节。

持久性

actor接收到的消息可以选择性的被持久化，并在actor启动或重启的时候重放。这使得actor能够恢复其状态，即使是在JVM崩溃或正在迁移到另外节点的情况下。

详情请参阅Java和Scala相关章节.

Scala 和 Java APIs

Akka同时提供 Scala API 和 Java API。

Akka的两种使用方式

以库的形式：在web应用中使用，放到 WEB-INF/lib 中或者作为一个普通的Jar包放进classpath。

以微内核的形式：可以将你的应用放进一个独立的内核。

参阅用例与部署场景了解细节。

商业支持

Typesafe提供Akka的商业许可，提供开发和产品支持，详见这里

为什么使用Akka?

为什么使用Akka?

Akka平台提供哪些有竞争力的特性？

Akka提供可扩展的实时事务处理。

Akka为以下目标提供了一致的运行时与编程模型：

- 垂直扩展（并发）
- 水平扩展（远程调用）
- 高容错

这个模型是唯一需要学习和掌握的，它具有高内聚和高一致的语义。

Akka是一种高度可扩展的软件，这不仅仅表现在性能方面，也表现在它所适用的应用的大小。Akka的核心—akka-actor是非常小的，可以方便地加入你的应用中，提供你所需要的异步无锁并行功能，不会有任何困扰。

你可以任意选择Akka的某些组件部分集成到你的应用中，也可以使用完整的包—Akka 微内核，它是一个独立的容器，可以直接部署你的Akka应用。随着CPU核数越来越多，即使你只使用一台主机，Akka也可作为一种性能卓越的选择。Akka还同时提供多种并发范型，允许用户选择正确的工具来完成工作。

什么场景下特别适合使用Akka？

我们看到Akka被成功运用在众多行业的众多企业中：

- 投资业到商业银行
- 零售业
- 社会媒体
- 仿真
- 游戏和博彩
- 汽车和交通系统
- 卫生保健
- 数据分析
  等等等等。任何需要高吞吐率和低延迟的系统都是使用Akka的候选。

Actor使你能够进行服务失败管理（监控），负载管理（缓和策略、超时和处理隔离），以及水平和垂直方向上的可扩展性（增加cpu核数和/或增加更多的机器）管理。

下面的链接中有一些Akka用户关于他们如何使用Akka的描述：
http://stackoverflow.com/questions/4493001/good-use-case-for-akka

所有以上这些都在这个使用Apache2许可的开源软件中。

入门

入门

准备工作

Akka要求你安装了 `Java 1.6`或更高版本。

入门指南和模板工程

最好的学习Akka的方法是下载“Typesafe Activator”并且尝试一下其中的Akka模板工程。

下载

下载Akka有几种方法。你可以通过下载Typesafe平台来下载Akka（如前所述）。你可以下载包含微内核的完整发布包（包含所有的模块）。或者也可以使用构建工具如Maven或SBT从Akka Maven仓库下载依赖。

模块

Akka的模块化做得非常好，它为不同的功能提供了不同的Jar包。

- akka-actor – 标准Actor，类型Actor，IO Actor等。
- akka-agent – Agent，与 Scala STM 集成
- akka-camel – Apache Camel 集成
- akka-cluster – 集群成员管理，弹性路由器。
- akka-kernel – Akka 微内核来运行简单应用服务器
- akka-osgi – 在OSGi容器中使用Akka的基本组件，包含akka-actor类
- akka-osgi-aries – Aries 的actor系统蓝图
- akka-remote.jar – 远程Actor
- akka-slf4j.jar – SLF4J日志(事件总线监听器)
- akka-testkit.jar – Actor系统的测试工具包
- akka-zeromq – ZeroMQ 集成

除了这些稳定的模块之外，还有一些虽然趋于稳定但仍然被标记为“实验”的模块。这并不是说他们的功能不符合预期，而主要的意思是他们的API还没有足够稳定到被认为已经固定了。你可以通过在我们的邮件

组里进行反馈，来加速试验模块发布的进程。

- akka-contrib – 一系列的Akka贡献，他们有可能被加入核心模块中，详情见外部贡献。

实际的jar包文件名会加上版本号，如akka-actor_2.10-2.3.6.jar（对其他模块也是类似）。

查看Akka模块之间的jar依赖的详情在依赖这一节中。

使用发布版

从http://akka.io/downloads 下载发布包并解压.

使用快照版

Akka的每日快照发布在 http://repo.akka.io/snapshots/, 版本号中包含 SNAPSHOT 和时间戳. 你可以选择一个快照版，可以决定何时升级到一个新的版本。Akka快照仓库也可以在 http://repo.typesafe.com/typesafe/snapshots/ 找到，此处还包含Akka模块依赖的其它仓库。

> 警告
>
> 不鼓励直接使用*Akka*快照版（*SNAPSHOT*）、每日构建版（*nightly*）和里程碑版（*milestone*），除非你知道自己在做什么。

微内核

Akka发布包包含微内核。要运行微内核，将你应用的jar包放到 `deploy` 目录下并运行 `bin` 目录下的脚本即可。

关于微内核的更多文档在 微内核(Scala) / 微内核(Java)。

使用构建工具

Akka可以与支持Maven仓库的构建工具一起使用。

Maven仓库

对Akka 2.1-M2 及以后的版本:

Maven Central

对以前的Akka 版本:

Akka Repo Typesafe Repo

通过Maven使用Akka

通过Maven使用Akka最简单的入门是检出“Typesafe Activator”中的模板工程“Akka Main in Java”。

由于Akka已经发布到Maven中心仓库了（自2.1-M2版本起），所以直接在POM文件中加入Akka依赖即可。例如。这是akka-actor的依赖:

```
1.     <dependency>
2.       <groupId>com.typesafe.akka</groupId>
3.       <artifactId>akka-actor_2.10</artifactId>
4.       <version>2.3.6</version>
5.     </dependency>
```

注意：对快照版本，SNAPSHOT和时间戳都在版本号中。

通过SBT使用Akka

通过SBT使用Akka最简单的入门是检出“Akka/SBT”模板工程。

通过SBT使用Akka的要点:

SBT安装指导 https://github.com/harrah/xsbt/wiki/Setup

`build.sbt` 文件：

```
1.  name := "My Project"
2.
3.  version := "1.0"
4.
5.  scalaVersion := "2.10.4"
6.
7.  resolvers += "Typesafe Repository" at
    "http://repo.typesafe.com/typesafe/releases/"
8.
9.  libraryDependencies +=
10.   "com.typesafe.akka" %% "akka-actor" % "2.3.6"
```

注意：以上的 `libraryDependencies` 设置需要SBT 0.12.x或更高的版本。如果你使用更老版本的SBT， `libraryDependencies` 需要这样设置：

```
1.  libraryDependencies +=
2.    "com.typesafe.akka" % "akka-actor_2.10" % "2.3.6"
```

Using Akka with Gradle

需要 Gradle 1.4 及以上的版本来使用 Scala 插件

```
1.  apply plugin: 'scala'
2.
3.  repositories {
4.    mavenCentral()
5.  }
6.
7.  dependencies {
8.    compile 'org.scala-lang:scala-library:2.10.4'
9.  }
10.
```

```
tasks.withType(ScalaCompile) {
  scalaCompileOptions.useAnt = false
}

dependencies {
  compile group: 'com.typesafe.akka', name: 'akka-actor_2.10', version: '2.3.6'
  compile group: 'org.scala-lang', name: 'scala-library', version: '2.10.4'
}
```

通过Eclipse使用Akka

建好SBT项目并使用 sbteclipse 来创建Eclipse项目。

通过IntelliJ IDEA使用Akka

建好SBT项目并使用 sbt-idea 来创建IntelliJ IDEA 项目。

通过NetBeans使用Akka

建好SBT项目并使用 nbsbt 来创建NetBeans项目。

你也应该使用nbscala提供的scala IDE支持

不要使用scala编译器的-optimize选项

> 警告
>
> *Akka*并没有在*scala*编译器的 `-optimize` 选项下编译和测试过。尝试过这种方式的用户发现了*Akka*的奇怪行为。

从源码编译

Akka使用Git并托管在 Github。

- Akka: 从 http://github.com/akka/akka 克隆 Akka 的资源库

具体请参考 构建Akka。

需要帮助?

如果有问题，你可以在 Akka Mailing List 获得帮助。

也可以寻求 商业支持。

感谢你成为Akka社区的一部分。

必修的“Hello World”

必修的“Hello World”

将著名的问候语——“Hello World”——打印在控制台中，这个艰巨任务的actor版在Typesafe Activator中一个名为“Akka Main in Scala”的教程中有介绍。

这个教程展示了一个通用启动类 `akka.Main` ，它只需要一个命令行参数：应用主actor的类名。这个main方法会创建actor需要的底层运行环境，启动主actor，并且做好在主actor结束的时候关闭整个系统的准备。

另外还有一个名为Hello Akka! 的 Typesafe Activator教程也是关于这一问题的，不过它更加深入的描述了Akka的基础。

用例和部署场景

用例和部署场景

我该如何使用和部署 Akka?

Akka 可以有几种使用方式:

- 作为一个库: 以普通jar包的形式放在classpath上，或放到web应用中的 `WEB-INF/lib` 位置
- 作为一个独立的应用程序，使用微内核(Scala) / 微内核(Java) ，自己使用一个main类来初始化Actor系统

将Akka作为一个库

当编写web应用的时候，你很可能要使用这种方式。通过添加更多的模块，可以有多种使用Akka库模式的方式。

将Akka用作单独的微内核

Akka 也可以作为独立微内核使用。参阅 微内核(Scala) / 微内核(Java) 获取更多信息.

Akka使用实例

Akka使用实例

我们看到Akka被成功运用在众多行业的众多大企业，从投资业到商业银行、从零售业到社会媒体、仿真、游戏和博彩、汽车和交通系统、医疗保健、数据分析等等等等。对任何需要高吞吐率和低延迟的系统，Akka都是优秀的候选。

这里是一个由生产用户撰写的关于Akka的使用实例的非常好的讨论

以下是Akka被部署到生产环境中的领域

事务处理 （在线游戏，金融/银行业，贸易，统计，赌博，社会媒体，电信）

垂直扩展，水平扩展，容错/高可用性

服务后端（任何行业，任何应用）

提供REST，SOAP，Cometd，WebSockets 等服务；作为消息总线/集成层；垂直扩展，水平扩展，容错/高可用性

并发/并行（任何应用）

运行正确，方便使用，只需要将jar包添加到现有的JVM项目中（可使用Scala，Java，Groovy或JRuby）

仿真

主/从，计算网格，MapReduce等等

批处理（任何行业）

Camel集成来连接批处理数据源，Actor分治批处理工作负载

通信Hub（电信，Web媒体，手机媒体）

垂直扩展，水平扩展，容错/高可用性

游戏与博彩（MOM，在线游戏，博彩）

垂直扩展，水平扩展，容错/高可用性

商业智能/数据挖掘/通用密集计算

垂直扩展，水平扩展，容错/高可用性

复杂事件流处理

垂直扩展，水平扩展，容错/高可用性

概述

概述

- 监管与监控
  - 监管的意思
  - 顶级监管者
  - 重启的含义
  - 生命周期监控的含义
  - 一对一策略 vs. 多对一策略
- Actor引用, 路径与地址
  - 什么是Actor引用？
  - 什么是Actor路径?
  - 如何获得Actor引用？
  - Actor引用和路径相等性
  - 重用Actor路径
  - 与远程部署之间的互操作
  - 路径中的地址部分用来做什么？
  - Actor路径的顶级作用域
- 位置透明性
  - 天生的分布式
  - 破坏透明性的方式
  - 远程调用如何使用？
  - Peer-to-Peer vs. Client-Server
  - 使用路由来进行垂直扩展的标记点
- Akka与Java内存模型
  - Java内存模型
  - Actors与Java内存模型
  - Future与Java内存模型
  - STM与Java内存模型
  - Actor与共享的可变状态
- 消息发送语义

    - 一般规则
    - JVM内（本地）消息发送规则
    - 更高层次的抽象
    - 死信
- 配置
    - 配置读取的地方
    - 当使用JarJar,, OneJar, Assembly或任何jar打包命令（jar-bundler）
    - 自定义application.conf
    - 包含文件
    - 配置日志
    - 谈一谈类加载器
    - 应用特定设置
    - 配置多个ActorSystem
    - 从自定义位置读取配置
    - Actor部署配置
    - 参考配置清单

术语，概念

术语，概念

这一章我们将尝试建立通用的术语，对Akka面向的并发、分布式系统等提供一个坚实的讨论基础。请注意，这里的很多术语都没有统一的定义。我们只是希望在Akka文档的范围内给出可用的定义。

并发 vs. 并行

并发和并行是相关的定义，有一些微小的不同。*并发* 指的是两个或多个任务都有进展，即使他们没有被同时执行。例如可以这样实现：划分出时间片，几个任务交叉执行，尽管时间片的执行是线性的。*并行* 则是指可以真正同时执行。

异步 vs. 同步

一个方法调用是 *同步* 的，当调用者不能继续处理，除非方法返回一个

值或抛出一个异常。另一方面，一个 异步 调用允许调用者在调用方法的有限步后能够继续执行，并且该方法的结束可以被额外的机制通知到（也许是一个注册的回调callback，一个Future或一个消息）。

一个同步的API也许会使用阻塞实现同步性，但也不是必须的。一个CPU极为密集的任务也会导致类似阻塞的行为。通常推荐使用非阻塞API，因为它们能确保系统继续处理。Actor本质上是异步的：一个Actor可以在发送消息后继续处理，而不需要等待消息确实被送达。

非阻塞 vs. 阻塞

如果一个线程的延迟会导致其它一些线程无限期的延迟，我们称之为阻塞。一个很好的例子是资源可以被线程通过互斥锁独占。如果这个线程无限期地占有这个资源（例如不小心进入死循环），其他等待这个资源的线程就无法处理了。相反地，非阻塞 意味着没有线程可以无限期的阻塞其他线程。

相比阻塞操作，我们推荐非阻塞的操作，因为很明显这样系统不会因为阻塞操作而不再继续处理。

死锁 vs. 饥饿 vs. 活锁

当多个参与者互相等待别人达到某个特殊的状态才能继续处理的时候，死锁 出现了。因为如果一些参与者不达到特定状态，所有的参与者都不能执行（就像《第二十二条军规》描述的那样），所有相关子系统都停顿了。死锁和阻塞息息相关，因为阻塞使得一个参与者线程可以无限期地推迟其他线程的处理。

在死锁中，没有参与者可以处理，然而相对的 饥饿 可能发生，当有些参与者可以不断地处理，而另一些可能不行。一个典型的场景是一个幼稚的调度算法—总是选择高优先级的任务。如果高优先级的任务数量一

直足够多，则低优先级的任务永远不会被完成。

活锁 和死锁类似，没有参与者可以处理。区别在于与进程进入等待其他进程处理的“冻结”状态不同，参与者不断地变换他们的状态。一个示例场景是两个参与者和两个特殊的资源。他们分别试图获取资源，并且检查是不是另一个参与者也需要这个资源。如果该资源被另一个参与者请求，则它们试图获取另一个资源。在一个很不幸的情况下，也许两个参与者会不停的在两个资源上“跳跃”，永远在谦让而不使用资源。

竞态条件

当一组事件的顺序假设可能被外部不确定因素影响，我们称之为 竞态条件。竞态条件经常在多个线程共享一个可变状态时出现，一个线程对这个状态的操作可能被交织从而导致意外的行为。尽管这是常见的情况，但是共享状态并不一定会导致竞态条件。例如一个客户端向服务器发送无序的包（例如UDP数据包） `P1` ， `P2` 。由于包可能经过不同的网络路由器传送，所以服务器可能先收到 `P2` ，后收到 `P1` 。如果消息中没有包含发送顺序的相关信息的话，服务器是不可能确定包是否是按照发送顺序接收的。根据包的内容这可能会导致竞态条件。

> 注意
>
> 对两个*actor*之间的消息发送，*Akka*唯一提供的保证是消息的发送顺序是被保留的。详见 *消息传送可靠性Message Delivery Reliability*

非阻塞担保（进展条件）

就像前几个章节描述的，阻塞是不受欢迎的，因为它有可能导致死锁并降低系统的吞吐量。在下面几节，我们将从不同深度讨论各种无阻塞特性。

无等待（Wait-freedom）

如果一个方法的调用可以保证在有限步骤内完成，则称该方法是 无等待 的。如果方法是 有界无等待 的，则方法的执行步数有一个确定的上界。

从这个定义可以得出无等待的方法永远不会阻塞，因此死锁是不可能发生的。此外，因为每个参与者都可以经过有限步后继续执行（当调用完成），所以无等待方法也不会出现饥饿的情况。

无锁（Lock-freedom）

无锁 是比 无等待 更弱的特性。在无锁调用的情况下，无限地经常有一些方法在有限步骤内完成。这个定义暗示着对无锁调用是不可能出现死锁的。另一方面，部分方法调用 在有限步骤内 结束，不足以保证所有调用最终完成。换句话说，无锁不足以保证不会出现饥饿。

无阻碍（Obstruction-freedom）

无阻碍 是这里讨论的最弱的无阻塞保证。对一个方法，当在某一个它独自执行的时间点（其他线程不在执行，例如都挂起了），之后它在有限步后能够结束，我们称之为 无阻碍。所有无锁的对象都是无阻碍的，但反之一般不成立。

乐观并发模型*OCC*（*Optimistic concurrency control* ） 的方法通常是无阻碍的。OCC的做法是，每一位参与者都试图在共享对象上执行操作，但是如果参与者检测到来自其他参与者的冲突，它回滚修改，并根据调度再次尝试。如果在某一个时间点，其中一个参与者，是唯一一个尝试修改的点，则其操作就会成功。

推荐文献

- The Art of Multiprocessor Programming, M. Herlihy and N Shavit, 2008. ISBN 978-0123705914

（注：中文译《多处理器编程的艺术》）

- Java Concurrency in Practice, B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes and D. Lea, 2006. ISBN 978-0321349606（注：中文译《Java并发编程实战》）

Actor系统

Actor系统

Actor是封装状态和行为的对象，他们唯一的通讯方式是交换消息—把消息存放在接收方的邮箱里。从某种意义上来说，actor是面向对象最严格的形式，不过最好把它们比作人：在使用actor来对解决方案建模时，把actor想象成一群人，把子任务分配给他们，将他们的功能整理成一个有组织的结构，考虑如何将失败逐级上传（好在我们并不需要真正跟人打交道，这样我们就不需要关心他们的情绪状态和道德问题）。这个结果就可以作为软件实现的思维框架。

> 注意
>
> 一个*Actor*系统是一个很重的结构，它会分配一到*N*个线程，所以对每一个逻辑应用创建一个就够了。

树形结构

象一个经济组织一样，actor自然形成树形结构。程序中负责某一功能的actor，可能需要把它的任务分拆成更小的、更易管理的部分。为此它启动子actor并监督它们。虽然后面章节会解释监督机制的细节，我们会集中在这一节里介绍其中的根本思想，唯一需要了解的前提是每个actor有且仅有一个监管者，就是创建它的那个actor。

Actor系统的精髓在于任务被拆开、委托，直到任务小到可以被完整地处理。这样做不仅清晰地划分出了任务本身的结构，而且最终的actor也能按照它们“应该处理什么类型的消息”，“如何处理正常流程”以及“如何应对失败流程”来进行推理。如果一个actor对某种状况无法进行处理，它会发送相应的失败消息给它的监管者请求帮助。这样的递归结构使得失败能够在正确的层次得到处理。

可以将这种思想与分层的设计方法进行比较。分层的设计方法最终很容易形成防护性编程，以防止任何失败被泄露出来；相比之下把问题交由正确的人处理会是比将所有的事情“藏在深处”更好的解决方案。

现在，设计这种系统的难度在于如何决定谁应该监管什么。这当然没有唯一的最佳方案，但是有一些指导原则可能会有帮助：

- 如果一个actor管理另一个actor所做的工作，如分配一个子任务，那么父actor应该监督子actor。因为父actor知道可能会出现哪些失败情况，以及如何处理它们。
- 如果一个actor携带着重要数据（即它的状态要尽可能地不被丢失），这个actor应该将任何可能出现危险的子任务分配给它所监管的子actor，并酌情处理子任务的失败。根据请求的性质，可能的话最好为每一个请求创建一个子actor，这样能简化收集回应的状态管理。这在Erlang中被称为“Error Kernel Pattern”。
- 如果actor A需要依赖actor B才能完成它的任务，A应该观测B的存活状态并对B的终止提醒消息进行响应。这与监管机制不同，因为观测方对监管机制没有影响；需要指出的是，仅仅是功能上的依赖并不足以用来决定是否在树形监管体系中添加子actor.

当然以上的规则都会有例外，但无论是遵循这些规则还是打破它们，都需要有足够的理由。

配置容器

actor系统是多个协作actor的组，它天生就是管理调度服务、配置、日志等共享设施的单元。使用不同配置的多个actor系统可以在同一个jvm中共存，Akka自身没有全局共享的状态。将这与actor系统之间的透明通讯（在同一节点上或者跨网络连接的多个节点）结合，可以看到actor系统本身可以被作为功能层次中的构建单元。

Actor最佳实践

- Actor应该被视为友好的同事：高效完成其工作，不会无必要地打扰其它人，也不会争抢资源。对应在编程中，其意思是以事件驱动的方式来处理事件并生成响应（或更多的请求）。Actor不应该因为某一个外部实体而阻塞（即占据一个线程又被动等待），这个外部实体可能是一个锁、一个网络socket等等；除非它是不可避免的，如下一节所述。
- 不要在actor之间传递可变对象。为了保证这一点，选择不可变消息。如果actor将他们的可变状态暴露给外界，打破了封装，你就回到了普通的Java并发领域并遭遇其所有缺点。
- Actor是行为和状态的容器，拥抱这一点意味着不要在消息中传递行为（例如在消息中使用scala闭包）。有一个风险是意外地在actor之间共享了可变状态，这种对actor模型规则的违反将破坏使用actor编程带来的所有良好体验。
- 顶级actor在错误内核最深处，所以尽量少创建它们并且选择真正的树形分层系统。这对故障处理有好处（同时考虑到配置的粒度和性能），同时也减少了对监管actor这个竞争单点的过度使用。

阻塞需要仔细的管理

在某些情况下，阻塞操作是不可避免的，即必须不定期地休眠一个线

程，等待外部事件唤醒。例如传统的关系型数据库的驱动程序或消息传递API，而深层的原因通常是出现幕后的（网络）I/O。面对这一点，你可能受到诱惑，只是将阻塞调用包装在 `Future` 中来替之工作，但这个策略太简单了：当应用的负载增加，你很可能会发现性能瓶颈，或者出现内存或线程耗尽的情况。

对“阻塞问题”的充分解决方案的清单是不会穷尽的，但肯定会有下面的建议：

- 在一个actor（或由一个路由器[Java, Scala]管理的一组actor内）内进行阻塞调用，并确保配置一个线程池，它要足够大或者专门用于这一目的。
- 在一个 `Future` 内进行阻塞调用，确保任意时间点内这种调用的数量都在一个上限内（无限制提交这类任务会耗尽你的内存或线程）。
- 在一个 `Future` 内进行阻塞调用，使用一个线程池，该线程池的线程数上限对应用程序运行的硬件是合适的。
- 奉献一个单独的线程来管理一组阻塞资源（如一个NIO选择器驱动多个频道），并在事件发生时把它们作为actor消息发送。

第一个建议对本质上是单线程的资源特别适合，如传统数据库句柄一次只能执行一个未完成的查询，并使用内部同步保证这一点。一个常见的模式是对N个actor创建一个router，每个actor包装一个数据库连接，并处理发送给这个router的查询。数目 `N` 必须被调整为最大吞吐量，这个数字取决于什么数据库管理系统部署在什么硬件上。

> 注意
>
> 配置线程池的任务最好代理给*Akka*来做，只要在 *`application.conf`* 中配置，并由 *`ActorSystem`* *[Java, Scala]* 实例化即可。

你不应该担心的事

一个actor系统管理它所配置使用的资源，运行它所包含的actor。在一个系统中可能有上百万个actor，不用担心，内存一定是够用的，因为每个actor实例仅占差不多300个字节。自然地，一个大系统中消息处理的具体顺序是不受应用开发者控制的，但这并不是有意为之。放松些，让Akka去做幕后的繁重事务吧。

什么是Actor?

什么是Actor?

上一节 Actor系统 解释了actor是应用创建中最小的单元，以及它们如何组成一个树形结构。本节单独来看看一个actor，解释在实现它时你会遇到的概念。更多细节请参阅 Actors (Scala)和 Actors (Java).

一个Actor是一个容器，它包含了状态，行为，一个邮箱，子Actor和一个监管策略。所有这些封装在一个Actor引用里。最终在Actor终止时，会有这些发生。

Actor引用" class="reference-link">Actor引用

如下所详细描述的，一个actor对象需要与外界隔离开才能从actor模型中获益。因此actor是以actor引用的形式展现给外界的，actor引用作为对象，可以被无限制地自由传递。内部和外部对象的这种划分使得所有想要的操作都能够透明：重启actor而不需要更新别处的引用，将实际actor对象放置到远程主机上，向另外一个应用程序发送消息。

但最重要的方面是从外界不可能到actor对象的内部获取其状态，除非这个actor非常不明智地将信息公布出去。

状态" class="reference-link">状态

Actor对象通常包含一些变量来反映其所处的可能状态。这可以是一个明确的状态机（例如使用 FSM 模块），或是一个计数器，一组监听器，待处理的请求，等等。这些数据使得actor有价值，并且必须将这些数据保护起来不被其它actor所破坏。好消息是在概念上每个Akka actor都有自己的轻量线程，它与系统其它部分是完全隔离的。这意味着你不需要使用锁来进行资源同步，可以直接编写你的actor代码，完全不必担心并发问题。

在幕后，Akka会在一组真实线程上运行Actor组，通常是很多actor共享一个线程，对某一个actor的调用可能会在不同的线程上得到处理。Akka保证这个实现细节不影响处理actor状态的单线程性。

由于内部状态对于actor的操作是至关重要的，所以状态不一致是致命的。因此当actor失败并被其监管者重新启动时，状态会被重新创建，就象第一次创建这个actor一样。这是为了实现系统的“自愈合”。

可选地，通过持久化收到的消息并在重启后重放它们，一个actor的状态可自动恢复到重启前的状态（详见持久化）

行为" class="reference-link">行为

每当一个消息被处理，它会与actor的当前行为进行匹配。行为是一个函数，它定义了在某个时间点处理当前消息所要采取的动作，例如如果客户已经授权，那么就对请求进行转发处理，否则拒绝。这个行为可能随着时间而改变，例如由于不同的客户在不同的时间获得授权，或是由于actor进入了“非服务”模式，之后又变回来。这些变化的实现，要么

是通过将它们编码入状态变量中并由行为逻辑读取，要么是函数本身在运行时被交换出来，见 `become` 和 `unbecome` 操作。但是actor对象在创建时所定义的初始行为是特殊的，因为actor重启时会恢复这个初始行为。

邮箱" class="reference-link">邮箱

Actor的目的是处理消息，这些消息是从其它actor（或者从actor系统外部）发送过来的。连接发送者与接收者的纽带是actor的邮箱：每个actor有且仅有一个邮箱，所有的发来的消息都在邮箱里排队。排队按照发送操作的时间顺序来进行，这意味着由于actor分布在不同的线程中，所以从不同的actor发来的消息在运行时没有一个固定的顺序。从另一个角度讲，从同一个actor发送到相同目标actor的多个消息，会按发送的顺序排队。

可以有不同的邮箱实现供选择，缺省的是FIFO：actor处理消息的顺序与消息入队列的顺序一致。这通常是一个好的选择，但是应用可能需要对某些消息进行优先处理。在这种情况下，可以使用优先邮箱来根据消息优先级将消息放在非队尾的某个指定位置，甚至可能是队列头。如果使用这样的队列，消息的处理顺序是由队列的算法决定的，而不是FIFO。

Akka与其它actor模型实现的一个重要区别在于：当前的行为总是必须处理下一个从队列中取出的消息，Akka不会扫描邮箱队列来获取下一个匹配的消息。无法处理某个消息通常被认为是失败情况，除非这个行为被重写。

子Actor" class="reference-link">子Actor

每个actor都是一个潜在的监管者：如果它创建了子actor来委托处理

子任务，它会自动地监管它们。子actor列表维护在actor的上下文中，actor可以访问它。对列表的更改是通过创建（`context.actorOf(...)`）或者停止（`context.stop(child)`）子actor来完成，并且这些更改会立刻生效。实际的创建和停止操作是在幕后以异步方式完成的，这样它们就不会“阻塞”其监管者。

监管策略" class="reference-link">监管策略

Actor的最后一部分是它用来处理其子actor错误状况的机制。错误处理是由Akka透明完成的，针对每个出现的失败，将应用监管与监控中所描述的一个策略。由于策略是actor系统组织结构的基础，所以一旦actor被创建，它就不能被修改。

考虑到对每个actor只有唯一的策略，这意味着：如果一个actor的子actor们应用了不同的策略，则这些子actor应该按照相同的策略来进行分组，并放在一个中间的监管者下，又一次转向了根据任务到子任务的划分来组织actor系统的结构的设计方法。

当Actor终止时" class="reference-link">当Actor终止时

当一个actor终止——即失败了且不能用重启来解决、停止它自己或者被它的监管者停止——它会释放其资源，将其邮箱中所有未处理的消息放进系统的“死信邮箱(dead letter mailbox)”，即将所有消息作为死信重定向到事件流中。而actor引用中的邮箱将会被一个系统邮箱所替代，将所有的新消息作为死信重定向到事件流中。 但是这些操作只是尽力而为，所以不能依赖它来实现“投递保证”。

不是简单地把消息扔掉的想法来源于我们的测试：我们在事件总线上注册了`TestEventListener`来接收死信，然后将每个收到的死信在日志中

生成一条警告—这对于更快地解析测试失败非常有帮助。可以想象这个特性也可以用于其它的目的。

监管与监控

监管与监控

这一节将简述监管背后的概念、原语及语义。要了解这些如何转换成真实代码，请参阅相关的Scala和Java API章节。

监管的意思" class="reference-link">监管的意思

在 Actor 系统 中说过，监管描述的是actor之间的依赖关系：监管者将任务委托给下属，并相应地对下属的失败状况进行响应。当一个下属出现了失败（即抛出一个异常），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。基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. 恢复下属，保持下属当前积累的内部状态
2. 重启下属，清除下属的内部状态
3. 永久地停止下属

4. 升级失败（沿监管树向上传递失败），由此失败自己

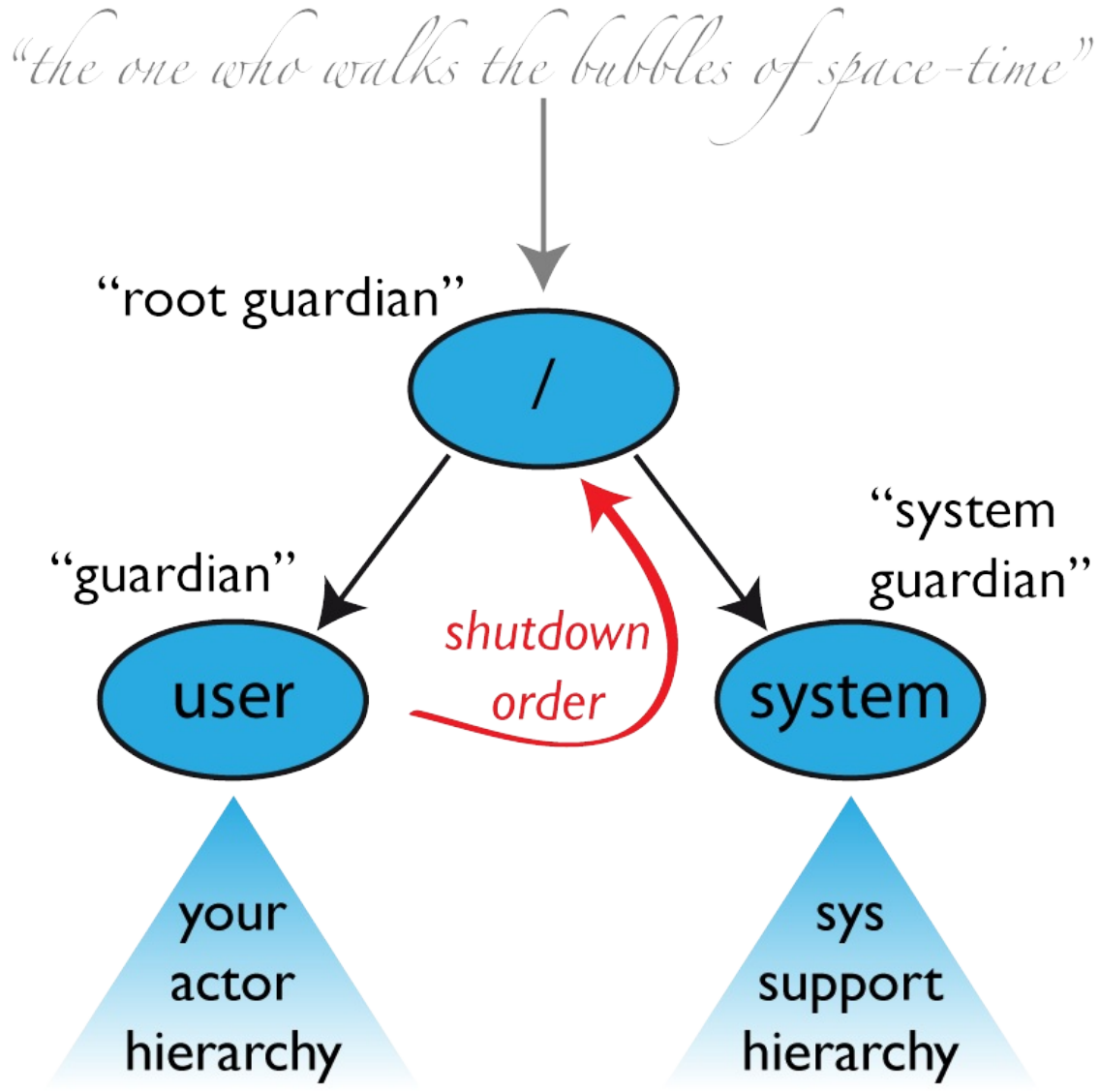
始终要把一个actor视为整个监管树形体系的一部分是很重要的，这解释了第4种选择存在的意义（因为一个监管者同时也是其上方监管者的下属），并且隐含在前3种选择中：恢复actor会恢复其所有下属，重启一个actor也必须重启其所有下属（不过需要看下面的详述获取更多细节），类似地终止一个actor会终止其所有下属。需要强调 `Actor` 类的 `preRestart` 钩子（hook）缺省行为是在重启前终止它的所有下属，但这个钩子可以被重写；对所有子actor的递归重启操作在这个钩子之后执行。

每个监管者都配置了一个函数，它将所有可能的失败原因（即异常）翻译成以上四种选择之一；注意，这个函数并不将失败actor的标识作为输入。我们很快会发现在有些结构中这种方式可能看起来不够灵活，例如会希望对不同的下属应用不同的策略。在这一点上我们一定要理解监管是为了组建一个递归的失败处理结构。如果你试图在某一个层次做太多事情，这个层次会变得复杂并难以理解，因此这时我们推荐的方法是增加一个监管层次。

Akka实现的是一种叫“父监管”的形式。Actor只能被其它的actor创建—顶部的actor由库来提供—每一个被创建的actor都由其父亲所监管。这种限制使得actor的监管结构隐式符合其树形层次，并提倡合理的设计方法。需要强调的是这也保证了actor不会成为孤儿或者拥有在系统外界的监管者（被外界意外捕获）。另外，这形成了对actor应用(或其子树)一种自然又干净的关闭过程。

> 警告
>
> 监管相关的父-子沟通，使用了特殊的系统消息及其固有的邮箱，从而和用户消息隔离开来。这意味着，监管相关的事件相对于普通的消息没有确定的顺序关系。在一般情况下，用户不能影响正常消息和失败通知的顺序。相关详细信息和示例，请参见讨论：消息排序。

顶级监管者



一个actor系统在其创建过程中至少要启动三个actor，如上图所示。有关actor路径及相关信息请参见Actor路径的顶级作用域。

`/user`: 守护Actor" class="reference-link"> `/user`: 守护Actor

这个名为 `"/user"` 的守护者，作为所有用户创建actor的父actor，可能是需要打交道最多的。使用 `system.actorOf()` 创建的actor都是其子actor。这意味着，当该守护者终止时，系统中所有的普通actor都将

被关闭。同时也意味着，该守护者的监管策略决定了普通顶级actor是如何被监督的。自Akka 2.1起就可以使用这个设定 `akka.actor.guardian-supervisor-strategy` ，以一个 `SupervisorStrategyConfigurator` 的完整类名进行配置。当这个守护者上升一个失败，根守护者的响应是终止该守护者，从而关闭整个actor系统。

`/system` : 系统守护者

这个特殊的守护者被引入，是为了实现正确的关闭顺序，即日志（logging）要保持可用直到所有普通actor终止，即使日志本身也是用actor实现的。其实现方法是：系统守护者观察user守护者，并在收到 `Terminated` 消息初始化其自己的关闭过程。顶级的系统actor被监管的策略是，对收到的除 `ActorInitializationException` 和 `ActorKilledException` 之外的所有 `Exception` 无限地执行重启，这也将终止其所有子actor。所有其他 `Throwable` 被上升，然后将导致整个actor系统的关闭。

`/` : 根守护者

根守护者所谓“顶级”actor的祖父，它监督所有在Actor路径的顶级作用域中定义的特殊actor，使用发现任何 `Exception` 就终止子actor的 `SupervisorStrategy.stoppingStrategy` 策略。其他所有Throwable都会被上升……但是上升给谁？所有的真实actor都有一个监管者，但是根守护者没有父actor，因为它就是整个树结构的根。因此这里使用一个虚拟的 `ActorRef` ，在发现问题后立即停掉其子actor，并在根守护者完全终止之后（所有子actor递归停止），立即把actor系统的 `isTerminated` 置为 `true` 。

重启的含义" class="reference-link">重启的含义

当actor在处理某条消息时失败时，失败的原因可以分成以下三类：

- 对收到的特定消息的系统错误（即程序错误）
- 处理消息时一些外部资源的（临时性）失败
- actor内部状态崩溃了

除非故障能被专门识别，否则所述的第三个原因不能被排除，从而引出内部状态需要被清除的结论。如果监管者确定它的其他子actor或本身不会受到崩溃的影响—例如使用了错误内核模式的能够自我恢复的应用—那么最好只重启这个孩子。具体实现是通过建立底层 `Actor` 类的新实例，并用新的 `ActorRef` 更换故障实例；能做到这一点是因为将actor都封装载了特殊的引用中。然后新actor恢复处理其邮箱，这意味着该启动在actor外部是不可见的，显著的异常是在发生失败期间的消息不会被重新处理。

重启过程中所发生事件的精确次序是：

1. actor被挂起（意味着它不会处理正常消息直到被恢复），并递归挂起其所有子actor
2. 调用旧实例的 `preRestart` hook（缺省实现是向所有子actor发送终止请求并调用 `postStop`）
3. 等待所有子actor终止（使用`context.stop()`）直到 `preRestart` 最终结束；这里所有的actor操作都是非阻塞的，最后被杀掉的子actor的终止通知会影响下一步的执行
4. 再次调用原来提供的工厂生成actor的新实例
5. 调用新实例的`postRestart`方法（其默认实现是调用`preStart`方法）
6. 对步骤3中没有被杀死的所有子actor发送重启请求；重启的actor会遵循相同的过程，从步骤2开始
7. 恢复这个actor

生命周期监控的含义

> 注意
>
> 生命周期监控在*Akka*中经常被引作 *DeathWatch*

与上面所描述的特殊父子关系相对的，每一个actor都可以监控其他任意actor。由于actor从创建到完全可用和重启都是除了监管者之外都不可见的，所以唯一可用于监视的状态变化是可用到失效的转变。监视因此被用于绑定两个actor，使监控者能对另一个actor的终止做出响应，而相应的，监督者是对失败做出响应。

生命周期监控是通过监管actor收到 `Terminated` 消息实现的，其默认行为是抛出一个 `DeathPactException` 。要开始监听 `Terminated` 消息，需要调用 `ActorContext.watch(targetActorRef)` 。要停止监听，需要调用 `ActorContext.unwatch(targetActorRef)` 。一个重要的特性是，消息将不考虑监控请求和目标终止发生的顺序，也就是说，即使在登记的时候目标已经死了，你仍然会得到消息。

如果一个监管者不能简单地重启其子actor，而必须终止它们，这时监控就特别有用，例如在actor初始化时发生错误。在这种情况下，它应该监控这些子actor并重新创建它们，或安排自己在稍后的时间重试。

另一个常见的应用情况是，一个actor需要在没有外部资源时失败，该资源也可能是它的子actor之一。如果第三方通过 `system.stop(child)` 或发送 `PoisonPill` 的方式终止子actor，其监管者很可能会受到影响。

一对一策略 vs. 多对一策略

Akka中有两种类型的监管策略： `OneForOneStrategy` 和 `AllForOneStrategy` 。两者都配置有从异常类型监管指令间的映射

（见上文），并限制了一个孩子被终止之前允许失败的次数。它们之间的区别在于，前者只将所获得的指令应用在发生故障的子actor上，而后者则是应用在所有孩子上。通常情况下，你应该使用 `OneForOneStrategy` ，这也是默认的策略。

`AllForOneStrategy` 适用的情况是，子actor之间有很紧密的依赖，以至于一个actor的失败会影响其他孩子，即他们是不可分开的。由于重启不清除邮箱，所以往往最好是失败时终止孩子并在监管者显式地重建它们（通过观察孩子们的生命周期）；否则你必须确保重启前入队的消息在重启后处理是没有问题的。

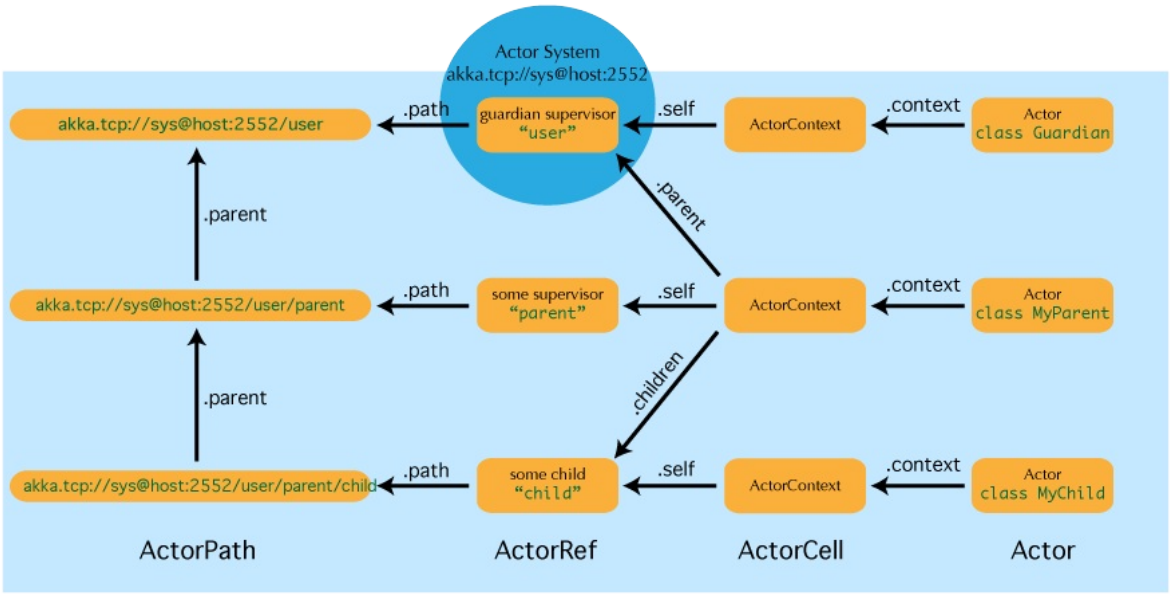
通常停止一个孩子（即对失败不再响应）不会自动终止多对一策略中其他的孩子；可以很容易地通过观察它们的生命周期来做到这点：如果 `Terminated` 的消息不能被监管者处理，它会抛出一个 `DeathPactException` ，并（这取决于其监管者）将重新启动，默认 `preRestart` 操作会终止所有的孩子。当然这也可以被显式地处理。

请注意，在多对一监管者下创建一个临时的actor会导致一个问题：临时actor的失败上升会使所有永久actor受到影响。如果这不是所期望的，安装一个中间监管者；这可以很容易地通过为工作者声明大小为1的路由器来完成，请参阅路由routing-scala或routing-java。

Actor引用, 路径与地址

Actor引用, 路径与地址

本章描述actor如何被确定，以及在一个可能是分布式的actor系统中如何定位。这与 Actor系统 的核心概念有关：固有的树形监管结构和跨多个网络节点的actor之间进行透明通讯。



上图展示了actor系统中最重要的实体关系，请继续阅读了解详情。

什么是Actor引用？" class="reference-link">什么是Actor引用？

Actor引用是 `ActorRef` 的子类，其最重要的目的是支持向它所代表的actor发送消息。每个actor通过 `self` 字段来访问自己的标准（本地）引用；在给其它actor发送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()` 方法来访问到当前消息的发送者的引用。

根据actor系统的配置，支持几种不同类型的actor引用：

- 纯本地actor引用，在配置为不使用网络功能的actor系统中使用。这些actor引用如果通过网络连接传给远程的JVM，将不能正常工作。
- 本地actor引用，在配置为使用远程功能的actor系统中使用，来代表同一个JVM的actor。为了能够在被发送到其它节点时仍然可达，这些引用包含了协议和远程地址信息。
- 本地actor引用的一个子类，用在路由器中（routers，即混入

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。
- 远程actor引用，代表可以通过远程通讯访问的actor，即向他们发送消息时会透明地对消息进行序列化，并发送到别的JVM。
- 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：
  - `PromiseActorRef` 表示一个`Promise`，其目的是通过一个actor返回的响应来完成。它是由 `akka.pattern.ask` 创建的。
  - `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被重新路由在此。
  - `EmptyLocalActorRef`是当查找一个不存在的本地actor路径时Akka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，并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消失之前被得到。
- 然后有一些内部实现，你应该永远不会用上：
  - 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”。
  - 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`。

什么是Actor路径?

由于actor是以一种严格的树形结构样式来创建的，所以沿着子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。这个序列可以被看做是文件系统中的文件路径，所以我们称之

为“路径”。就像在一些真正的文件系统中一样，也存在所谓的“符号链接”，即一个actor也许能通过不同的路径被访问到，除了原始路径外，其它的路径都涉及到对actor实际监管祖先链的某部分路径进行转换的方法。这些特性将在下面的内容中介绍。

一个actor路径包含一个锚点，来标识actor系统的，之后是各路径元素的连接，从根监护者到指定的actor；路径元素是路径经过的actor的名字，以”/“分隔。

Actor引用和路径之间有什么区别？

Actor引用标明了一个actor，其生命周期和actor的生命周期保持匹配；actor路径表示一个名称，其背后可能有也可能没有真实的actor，而且路径本身不具有生命周期，它永远不会失效。你可以创建一个actor路径，而无需创建一个actor，但你不能在创建actor引用时不创建相应的actor。

> 注意
>
> 这个定义并不适用于 `actorFor` ，这是为什么废弃 `actorFor` 而选择 `actorSelection` 的原因之一。

你可以创建一个actor，终止它，然后创建一个具有相同路径的新actor。新创建的实例是actor的一个新的化身。它并不是一样的actor。一个指向老的化身的actor引用不适用于新的化身。发送给老的actor引用的消息不会被传递到新的化身，即使它们拥有相同的路径。

Actor路径锚点

每一条actor路径都有一个地址组件，描述访问这个actor所需要的协议和位置，之后是从根到actor所经过的树节点上actor的名字。例如:

```
1. "akka://my-sys/user/service-a/worker1"                   // 纯本地
2. "akka.tcp://my-sys@host.example.com:5678/user/service-b" // 远程
```

在这里， `akka.tcp` 是Akka 2.2及以上版本默认的远程传输方式，其它的方式都是可以通过插件引入的。对使用UDP的远程主机可以使用 `akka.udp` 访问。对主机和端口部分的解析（即上例中的 `host.example.com:5678` ）决定于所使用的传输机制，但是必须遵循URI的结构标准。

Actor逻辑路径

顺着actor的父监管链一直到根的唯一路径被称为actor逻辑路径。这个路径与actor的创建祖先关系完全吻合，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。

Actor物理路径

Actor逻辑路径描述它在一个actor系统内部的功能位置，而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，即另一个actor系统中。在这种情况下，从根守护者穿过actor路径来找到该actor肯定需要访问网络，这是一个很昂贵的操作。因此，每一个actor同时还有一条物理路径，从actor对象实际所在的actor系统的根开始。与其它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。

物理路径的一个重要性质是它决不会跨多个actor系统或跨JVM虚拟机。这意味着如果一个actor有祖先被远程监管，则其逻辑路径（监管树）和物理路径（actor部署）可能会分叉。

如何获得Actor引用？

actor引用的获取方法分为两类：通过创建actor，或者通过查找

actor。后一种功能又分两种：通过具体的actor路径来创建actor引用，和查询actor逻辑树。

创建Actor

一个actor系统通常是在根守护者上使用 `ActorSystem.actorOf` 创建actor来启动，然后在创建出的actor中使用 `ActorContext.actorOf` 来展开actor树。这些方法返回的是指向新创建的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（通过 `ActorContext` 访问）。这些引用可以与消息一起被发送给别的actor，以便接收方直接回复。

通过具体的路径来查找actor

另外，可以使用 `ActorSystem.actorSelection` 来查找actor引用。“选择”可在已有actor与被选择的actor进行通讯的时候用到，在投递每条消息的时候都会用到查找。

为了获得一个绑定到指定actor生命周期的 `ActorRef` ，你需要发送一个消息，如内置的 `Identify` 信息，向指定的actor，所获得的 `sender()` 即为所求。

> 注意
>
> *actorFor* 因被 *actorSelection* 替代而废弃，因为 *actorFor* 对本地和远程的*actor*表现有所不同。对一个本地*actor*引用，被查找的*actor*需要在查找之前就存在，否则获得的引用是一个 *EmptyLocalActorRef* 。即使后来与实际路径相符的*actor*被创建，所获得引用仍然是这样。对于 *actorFor* 行为获得的远程*actor*引用则不同，每条消息的发送都会在远程系统中进行一次按路径的查找。

绝对路径 vs 相对路径

除了 `ActorSystem.actorSelection` 还有一个 `ActorContext.actorSelection` ，这是可以在任何一个actor实例中通过 `context.actorSelection` 访问的。它的actor查找与 `ActorSystem` 的

返回值非常类似，不同在于它的路径查找是从当前actor开始的，而不是从actor树的根开始。可以用 “..” 路径来访问父actor. 例如，你可以向一个指定兄弟发送消息:

```
1.  context.actorSelection("../brother") ! msg
```

当然绝对路径也可以在 context 中使用，即

```
1.  context.actorSelection("/user/serviceA") ! msg
```

也能正确运行。

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，对actor路径的匹配与Unix shell中支持的一样: 你可以将路径 (中的一部分) 用通配符 («*» 和 «?»)替换，来组成对0个或多个实际actor的选择。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。选择也可以用 `ActorSystem.actorSelection` 或 `ActorContext.actorSelection` 两种方式来获得，并且支持发送消息:

```
1.  context.actorSelection("../*") ! msg
```

会将msg发送给包括当前actor在内的所有兄弟。对于用 `actorFor` 获取的actor引用，为了进行消息的发送，会对监管树进行遍历。由于在消息到达其接收者的过程中，与查询条件匹配的actor集合可能会发生变化，要监视查询的实时变化是不可能的。如果要做这件事情，通过发送一个请求，收集所有的响应来解决不确定性，提取所有的发送方引用，然后监视所有被发现的具体actor。这种处理actor选择的方式也许会在未来的版本中进行改进。

总结： `actorOf` VS. `actorSelection` VS. `actorFor`

> *Note*
>
> 以上部分所描述的细节可以简要地总结和记忆成：
>
> - *`actorOf`* 永远都只会创建一个新的*actor*，这个新的*actor*是*actorOf*所调用上下文（可以是任意一个*actor*或*actor*系统本身）的直接子*actor*
> - *`actorSelection`* 只会在消息送达后查找已经存在的*actor*集合，即不会创建*actor*，也不会在创建选择集合时验证*actor*是否存在。
> - *`actorFor`*（废弃，已经被 *`actorSelection`* 取代） 永远都只是查找到一个已存在的*actor*，不会创建新的*actor*。

Actor引用和路径相等性

`ActorRef` 的相等性与 `ActorRef` 的目的匹配，即一个 `ActorRef` 对应一个目标actor化身。两个actor引用进行比较时，如果它们有相同的路径且指向同一个actor化身，则两者相等。指向一个已终止的actor的引用，与指向具有相同路径但却是另一个（重新创建）actor的引用是不相等的。需要注意的是，由于失败造导致的actor重启，仍意味着它是同一个actor化身，即重新启动对 `ActorRef` 消费者是不可见的。

由 `actorFor` 获得的远程actor引用不包括其身份的所有信息，因此，这种引用不能等于 `actorOf` ， `sender` 或 `context.self` 的引用。正因如此 `actorFor` 被 `actorSelection` 替换废弃。

如果你需要跟踪一个集合中的actor引用，并不关心具体的actor化身，你可以使用 `ActorPath` 为键（key），因为目标actor的标识符在比较actor路径时没有被用到。

重用Actor路径

当一个actor被终止，其引用将指向一个死信邮箱，DeathWatch将发布其最终的转变，并且一般地它也不会起死回生（因为actor的生命周期不允许这样）。虽然以后可能创建一个具有相同路径的actor—如果

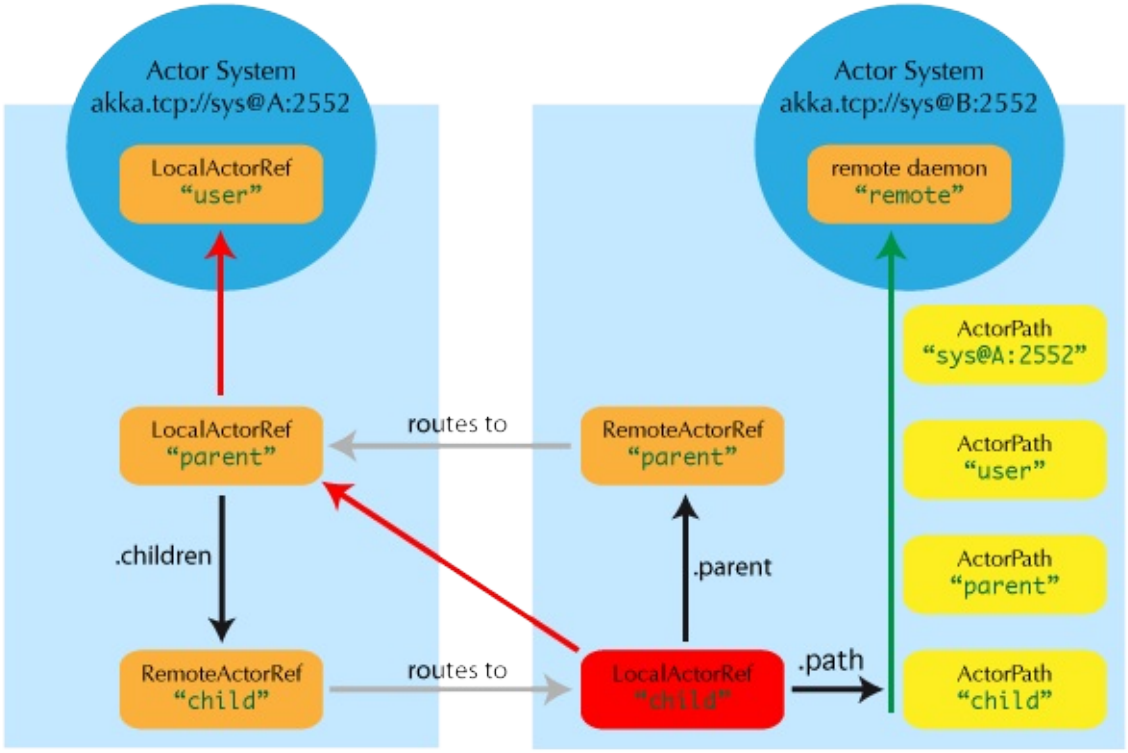
无法保留actor系统开始以来创建的所有可用actor，则无法保证其反向成立—但是这不是一个好的实践：通过 `acterFor` 获取的已经‘死亡’的远程actor引用突然再次开始工作，但没有这种过渡和任何其他事件之间顺序的任何保证，因此，该路径的新居民可能收到本意是送给其以前住户的消息。

在某些非常特殊的情况下这可能是正确的事情，但一定要限制这种处理只能由其监管者操作，因为它是唯一可以可靠地检测名称正确注销的actor，在注销之前的新创建子actor的操作将失败。

它在测试中可能也是必要的，当测试对象取决于某个特定路径被实例化的时候。在这种情况下，最好mock其监管者，这样它会将终止消息转发至测试过程正确的点，使后者能够等待登记名字的正确注销。

与远程部署之间的互操作

当一个actor创建一个子actor，actor系统的部署者会决定新的actor是在同一个jvm中还是在其它节点上。如果是后者，actor的创建会通过网络连接引到另一个jvm中进行，因而在另一个actor系统中。远程系统会将新的actor放在一个专为这种场景所保留的特殊路径下，新的actor的监管者将会是一个远程actor引用（代表触发它创建动作的actor）。这时， `context.parent` （监管者引用）和 `context.path.parent` （actor路径上的父actor）表示的actor是不同的。然而，在其监管者中查找这个actor的名称将会在远程节点上找到它，保持其逻辑结构，例如向另一个未确定(unresolved)的actor引用发送消息。



路径中的地址部分用来做什么？

在网络上传送actor引用时，是用它的路径来表示的。因此，它的路径必须包括能够用来向它所代表的actor发送消息的完整信息。这一点是通过将协议、主机名和端口编码在路径字符串的地址部分做到的。当actor系统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。

Actor路径的顶级作用域" class="reference-link">Actor路径的顶级作用域

在路径树的根上是根监管者，所有其他actor都可以从通过它找到；它的名字是 `"/"` 。在第二个层次上是以下这些：

- `"/user"` 是所有由用户创建的顶级actor的监管者；用`ActorSystem.actorOf`创建的actor在其下。
- `"/system"` 是所有由系统创建的顶级actor的监管者，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。
- `"/deadLetters"` 是死信actor，所有发往已经终止或不存在的actor的消息会被重定向到这里（以尽最大努力为基础：即使在本地JVM，消息也可能丢失）
- `"/temp"`是所有系统创建的短时actor的监管者，例如那些在`ActorRef.ask`的实现中用到的actor。
- `"/remote"` 是一个人造虚拟路径，用来存放所有其监管者是远程actor引用的actor。

需要为actor构建这样的名称空间源于一个核心的非常简单的设计目标：在树形结构中的一切都是一个actor，以及所有的actor都以相同方式工作。因此，你不仅可以查找你所创建的actor，你也可以查找系统守护者并发送消息（在这种情况下它会忠实地丢弃之）。这个强大的原则意味着不需要记住额外的怪异模式，它使整个系统更加统一和一致。

如果您想了解更多关于actor系统的顶层结构，参考顶级监管者。

位置透明性

位置透明性

上一节讲到了如何使用actor路径来实现位置透明性。这个特殊的功能需要额外的解释，因为“透明远程调用”在不同的上下文中（编程语言，平台，技术）有非常不同的用法。

天生的分布式

Akka中所有的东西都是被设计为在分布式环境下工作的：actor之间所有的互操作都是使用纯粹的消息传递机制，所有的操作都是异步的。付出这些努力是为了保证其所有功能，无论是在单一的JVM上还是在拥有很多机器的集群里都能同样有效。实现这一点的关键是从远程到本地进行优化，而不是从本地到远程进行一般化。参阅这篇 经典论文 来了解关于为什么第二种方式注定要失败的详细讨论。

破坏透明性的方式

由于设计分布式执行过程对可以做的事情添加了一些限制，Akka所满足的约束并不一定在使用akka的应用程序中也满足。最明显的一条是网络上发送的所有消息都必须是可序列化的。而不那么明显的是，这也包括在远程节点上创建actor时，用作actor工厂的闭包（即

在 `Props` 里）。

另一个结果是所有元素都需要知道所有交互是完全异步的，在一个计算机网络中这意味着一个消息可能需要好几分钟才能到达接收方（跟配置有关）。还意味着消息丢失的概率比在单一的jvm中（接近0，但仍不能完全保证！）高得多。

远程调用如何使用？

我们把透明性的想法限制在“Akka中几乎没有为远程调用层设计的API”：而完全由配置来驱动。你只需要按照之前的章节概括的那些原则来编写你的应用，然后在配置文件里指定远程部署的actor子树。这样你的应用可以不用通过修改代码来实现扩展。API中唯一允许编程来影响远程部署的部分是 `Props` 包含的一个属性，这个属性可能被设为一个特定的 `Deploy` 实例；这与在配置文件中进行部署设置具有相同的效果（如果两者都有，那么配置文件优先）。

Peer-to-Peer vs. Client-Server

Akka Remoting是以对等网络方式进行actor系统连接的通信模块，它是Akka集群的基础。远程处理的设计取决于两个（相关）的设计决策：

- 涉及系统之间的通信是对称的：如果系统A可连接到系统B，那么B系统必须也能够独立连接到系统A。
- 通信系统中的角色按照连接的模式是对称的：不存在系统只接受连接，也没有系统只发起连接。

这些决定的结果是，它不可能按照预定义的角色安全地创建纯粹的“客户端-服务器”设置（违反假设2 ），并使用网络地址转换（NAT）或负载平衡器（违反假设1）。

对于“客户端-服务器”的设置，最好是使用HTTP或Akka I/O 。

使用路由来进行垂直扩展的标记点

一个actor系统除了可以在集群中的不同节点上运行不同的部分，还可以通过并行增加actor子树的方法来垂直扩展到多个cpu核上（设想例如搜索引擎并行处理不同的检索词）。新增出来的子树可以使用不同的方法来进行路由，例如循环（round-robin）。要达到这种效果，开发者只需要声明一个“withRouter”的actor，这样系统会创建一个路由actor取而代之，该路由actor会按照期望类型和配置的数目生成子actor，并使用所配置的方式来对这些子actor进行路由。一旦声明了这样的路由，它的配置可以自由地被配置文件里的配置进行重写，包括把它与其(某些)子actor的远程部署进行混合。具体可参阅 路由器(Scala) 和 路由器(Java)。

Akka与Java内存模型

Akka与Java内存模型

使用包含Scala和Akka在内的Typesafe平台的主要好处是它简化了并发软件的编写过程。本文将讨论Typesafe平台，尤其是Akka是如何在并发应用中访问共享内存的。

Java内存模型

在Java 5之前，Java内存模型（JMM）定义是有问题的。当多个线程访问共享内存时很可能得到各种奇怪的结果，例如：

- 一个线程看不到其它线程所写入的值：可见性问题
- 由于指令没有按期望的顺序执行，一个线程观察到其它线程的 ‘不可能’ 行为：指令重排序问题

随着Java 5中JSR 133的实现，很多这种问题都被解决了。 JMM是一组基于 “发生在先” 关系的规则， 限制了一个内存访问行为何时必须在另一个内存访问行为之前发生，以及反过来，它们何时能够不按顺序发生。这些规则的两个例子包括：

- 监视器锁规则： 对一个锁的释放先于所有后续对同一个锁的获取

- **volatile**变量规则：对一个volatile变量的写操作先于所有对同一volatile变量的后续读操作

虽然JMM看起来很复杂，但是其规范试图在易用性和编写高性能、可扩展的并发数据结构的能力之间寻找一个平衡。

Actors与Java内存模型

使用Akka中的Actor实现，有两种方法让多个线程对共享的内存进行操作：

- 如果一条消息被（例如，从另一个actor）发送到一个actor，大多数情况下消息是不可变的，但是如果这条消息不是一个正确创建的不可变对象，如果没有 “发生先于” 规则， 有可能接收方会看到部分初始化的数据，甚至可能看到无中生有的数据（long/double）。
- 如果一个actor在处理某条消息时改变了自己的内部状态，而之后又在处理其它消息时又访问了这个状态。一条很重要的需要了解的规则是，在使用actor模型时你无法保证，同一个线程会在处理不同的消息时使用同一个actor。

为了避免actor中的可见性和重排序问题，Akka保证以下两条 “发生在先” 规则：

- **actor**发送规则 ： 一条消息的发送动作先于目标actor对同一条消息的接收。
- **actor**后续处理规则：对同一个actor，一条消息的处理先于下一条消息处理

> 注意
>
> 通俗地说，这意味着当这个*actor*处理下一个消息的时候，对*actor*的内部字段的改变是可见的。因此，在你的*actor*中的域不需要是*volitale*或是同等可见性的。

这两条规则都只应用于同一个actor实例，对不同的actor则无效。

Future与Java内存模型

一个Future的完成 “先于” 任何注册到它的回调函数的执行。

我们建议不要在回调中捕捉（close over）非final的值 （Java中称final，Scala中称val)， 如果你一定要捕捉非final的域，则它们必须被标记为*volatile*来让它的当前值对回调代码可见。

如果你捕捉一个引用，你还必须保证它所指代的实例是线程安全的。我们强烈建议远离使用锁的对象，因为它们会引入性能问题，甚至最坏可能造成死锁。这些是使用synchronized的风险。

STM与Java内存模型

Akka中的软件事务性内存 （STM） 也提供了一条 “发生在先” 规则：

- 事务性引用规则： 对一个事务性引用，在提交过程中一次成功的写操作，先于所有对同一事务性引用的后续读操作发生。

这条规则非常象JMM中的“volatile 变量”规则。目前Akka STM只支持延迟写，所以对共享内存的实际写操作会被延迟到事务提交之时。在事务中发生的写操作会被存放在一个本地缓冲区内 （事务的写操作集），并且对其它事务是不可见的。这就是为什么脏读是不可能的。

这些规则在Akka中的实现会随时间而变化，精确的细节甚至可能依赖于所使用的配置。但是它们是建立在其它JMM规则之上的，如监视器锁规则、volatile变量规则。 这意味着Akka用户不需要操心为了提供“发生先于”关系而增加同步，因为这是Akka的工作。这样你可以腾出手来处理业务逻辑，让Akka框架来保证这些规则的满足。

Actor与共享的可变状态" class="reference-link">Actor与共享的可变状态

因为Akka运行在JVM上，所以还有一些其它的规则需要遵守。

- 捕捉Actor内部状态并暴露给其它线程

```
1.
2.  class MyActor extends Actor {
3.    var state = ...
4.    def receive = {
5.      case _ =>
6.        // 错误的做法
7.
8.        // 非常错误，共享可变状态，
9.        // 会让应用莫名其妙地崩溃
10.       Future { state = NewState }
11.       anotherActor ? message onSuccess { r => state = r }
12.
13.       // 非常错误，共享可变状态 bug
14.       // "发送者"是一个可变变量，随每个消息改变
15.       Future { expensiveCalculation(sender) }
16.
17.       //正确的做法
18.
19.       // 非常安全， "self" 被闭包捕捉是安全的
20.       // 并且它是一个Actor引用，是线程安全的
21.       Future { expensiveCalculation() } onComplete { f => self !
    f.value.get }
22.
23.       // 非常安全，我们捕捉了一个固定值
24.       // 并且它是一个Actor引用，是线程安全的
25.       val currentSender = sender
26.       Future { expensiveCalculation(currentSender) }
27.   }
28. }
```

- 消息应当是不可变的， 这是为了避开共享可变状态的陷阱。

消息发送语义

消息发送语义

Akka帮助您在多核的单机上（“向上扩展”或纵向扩展）或分布式计算机网络中（“向外扩展”或横向扩展）构建可靠的应用程序。这里关键的抽象是，你的代码单元—actor—之间所有的交互都是通过消息传递

完成，这也是为什么“消息是如何在actor之间传递”的准确语义应该拥有自己的章节。

为了给出下面讨论的一些背景，考虑一个跨越多个网络主机的应用。首先通信的基本机制是相同的，无论是发送到一个在本地JVM中的actor，还是一个远程actor，不过当然在投递延迟上会有可观察到的差异（也可能决定于网络带宽和消息大小）和可靠性。对远程消息发送，显然会有更多步骤，从而意味着更多出错的可能。另一方面，本地消息发送只会传递一个本地JVM中消息的引用，所以没有对发送的底层对象上做任何限制，而远程传输将对消息的大小进行限制。

如果你在编写actor时，认为每一次消息交互都可能是远程的，这是安全但悲观的赌注。这意味着，只依赖那些始终被保证的特性（下面将详细讨论这些特性）。这样做当然会在actor的实现中带来一些额外的开销。如果你愿意牺牲完全的位置透明性—例如有一组密切合作的actor—你可以总是将它们放在同一个JVM中，并享受更加严格的消息传递保证。这种折衷的细节在下面会进一步讨论。

作为一个补充，我们将为如何在内建机制上构建更强的可靠性，给出一些指导意见。本章以讨论“死信办公室”的角色作为结束。

一般规则" class="reference-link">一般规则

这些是消息发送的规则（即 `tell` 或 `!` 方法，这也是 `ask` 的底层实现方式）：

- 至多一次投递，即不保证投递
- 对每个“发送者-接收者”对，有消息排序

第一条规则是典型的，并在其他actor框架中有出现，而第二个则是Akka独有的。

讨论：“最多一次”是什么意思？

当涉及到描述传递机制的语义时，有三种基本类型：

- 至多一次投递的意思是对该机制下的每条消息，会被投递0或1次；更随意的说法就是，它意味着消息可能会丢失。
- 至少一次投递的意思对该机制下的每条消息，有可能为投递进行多次尝试，以使得至少有一个成功；更随意的说法就是，消息可能重复，但不会丢失。
- 恰好一次投递的意思对该机制下的每条消息，接收者会正好得到一次投递；消息既不能丢，也不会重复。

第一种是最廉价的—性能最高，实现开销最少—因为它可以用打后不管（fire-and-forget）的方式完成，不需要在发送端或传输机制中保留状态。第二种方式要求重试来对抗传输丢失，这意味着需要在发送端保持状态，并在接收端使用确认机制。第三种是最昂贵的—并因此表现最差—因为除了需要第二种方式的机制以外，还需要在接收端保持状态，以过滤重复的投递。

讨论：为什么没有投递保证？

这个问题的核心在于这个担保究竟具体是什么意思：

1. 消息被发送在网络上？
2. 消息被其他主机接收？
3. 消息放到目标actor的邮箱中？
4. 消息开始被目标actor处理？
5. 消息被目标actor成功处理？

以上每个担保都具有不同的挑战和成本，并且很明显，存在一些情况导致任何消息传递框架都将无法遵守这些担保；想象例如可配置邮箱类型，以及一个有界的邮箱将如何与第3点互动，甚或第5点的“成功”由

什么决定。

在上面的话中包含着同样的推理——没人需要可靠地消息机制。对发送者来说，确定交互是否成功的唯一有意义的方式是收到业务层级的确认消息，这不是Akka可以做到的（我们不会写一个“按我的意思来做”的框架，也没有人会想让我们这样做）。

Akka拥抱了分布式计算，将消息传递的不可靠性明确化，因此它不会尝试说谎和实现一个有问题的抽象。这是一个已经在Erlang中大获成功的模型，它要求用户围绕它进行设计。你可以在“Erlang 文档”中读到更多介绍（第10.9和10.10节），Akka紧密地沿用了这种方法。

对这个问题的另一个角度是，通过只提供基本保障，那些无需更强可靠性的用例也就不要支付额外的实施成本；总是可以在基本的基础上添加更强的可靠性，但不可能相反移除可靠性，来获得更高的性能。

讨论：消息顺序" class="reference-link">讨论：消息顺序

该规则更具体的讲是，对于给定的一对**actor**，从第一个**actor**直接发送到第二个**actor**的消息不会被乱序接收。直接这个词强调通过 `tell` 操作符直接发给最终的目的地，而没有使用中介者或其他信息传播特性时（除非另有说明）。

该担保说明如下：

Actor `A1` 发送消息 `M1`, `M2`, `M3` 到 `A2`

Actor `A3` 发送消息 `M4`, `M5`, `M6` 到 `A2`

意味着：

1. 如果 `M1` 被投递，则它必须在 `M2` 和 `M3` 前被投递
2. 如果 `M2` 被投递，则它必须在 `M3` 前被投递
3. 如果 `M4` 被投递，则它必须在 `M5` 和 `M6` 前被投递

4. 如果 `M5` 被投递，则它必须在 `M6` 前被投递
5. `A2` 可以交织地看到 `A1` 和 `A3` 的消息
6. 因为没有投递保证，以上任意消息都有可能被丢弃，即没有到达 `A2`

> 注意
>
> 需要注意的是*Akka*保证消息被排队到收件人邮箱的顺序是很重要的。如果邮箱的实现不遵循*FIFO*的顺序（例如，一个 `PriorityMailbox` ），则*actor*的处理顺序可能偏离排队顺序。

请注意，这条规则不具有传递性：

> *Actor* `A` 发送消息 `M1` 给 *actor* `C`
>
> *Actor* `A` 然后发送消息 `M2` 给 *actor* `B`
>
> *Actor* `B` 转发消息 `M2` 给 *actor* `C`
>
> *Actor* `C` 可以以任何顺序接收 `M1` 和 `M2`

因果型顺序传递性意味着 `M2` 永远不会再 `M1` 之前到达actor `C` （尽管它们中的任何一个都可能会丢失）。这种顺序性无法保证，因为消息具有不同的传递延迟，例如当 `A` ， `B` 和 `C` 位于不同的网络主机时，详见下文。

> 注意
>
> *Actor*的创建被视为是从父节点发送到孩子的消息，和上面的讨论具有相同的语义。以消息可以被重排序的方式发送一个消息到一个*actor*，会导致消息丢失，因为创建的消息也许没有发送导致*actor*还不存在。一个消息可能过早到来的例子是，创建一个远程部署的*actor R1*，将其引用发送给另一个远程*actor R2*，并且让*R2*发送消息给*R1*。一个明确定义排序的例子是一个父节点创建一个*actor*，并立即向它发送消息。

失败消息的传达

请注意，上面只讨论了actor之间的用户消息的顺序保证。一个actor的孩子的失败是通过特殊的系统消息传达的，与普通用户发送的消息没有顺序关系。特别是：

> 子 *actor* `C` 发送 `M` 给其父节点 `P`
>
> 子 *actor* 失败并发送失败消息 `F`
>
> 父 *actor* `P` 可能以 `M`, `F` 或 `F`, `M` 的顺序收到两个事件

这样做的原因是，内部系统消息有其自己的邮箱，因此用户和系统信息的排队的顺序不能保证其出队的时间顺序。

JVM内（本地）消息发送规则" class="reference-link">JVM内（本地）消息发送规则

对本节介绍的内容要小心使用！

不建议依托本节所介绍的更强可靠性，因为它会绑定您的应用程序只能进行本地部署：为了适应在机器集群上运行，应用程序可能需要不同的设计（而不是仅仅是对actor采用一些本地消息交换模式）。我们的信条是“一次设计，按照任何你希望的方式部署”，要实现这一点，你应该只依靠一般规则。

本地消息发送的可靠性

Akka测试套件依赖于本地上下文没有消息丢失（以及对远程部署的非错误条件测试也成立），也就是说，我们实际中确实是以最大努力来保证我们测试的稳定性。然而，就像一个方法调用可能在JVM上失败一样，本地的 `tell` 操作也可能会因为同样的原因而失败：

- `StackOverflowError`
- `OutOfMemoryError`
- 其他`VirtualMachineError`

此外，本地传输可以以Akka特定的方式失败：

- 如果邮箱不接收消息（例如已满的BoundedMailbox）
- 如果接收的actor在处理消息时失败，或actor已终止

第一个显然是配置的问题，不过第二个是值得一些思考的：如果处理的时候有异常，则消息的发送者不会得到反馈，而是将通知发送给其父监管者了。对外部观察者来说，这和丢失这个消息没有区别。

本地消息发送顺序

假设使用严格的先进先出邮箱，则前面提到的消息非传递的排序担保，在一定条件下可以被消除。你会注意到，这些是很微妙的，甚至未来的性能优化有可能将本节的所有内容变为无效。反标志的一些可能如下：

- 在收到顶层actor的第一个回应之前，有一个锁用于保护内部的临时队列，并且该锁是非公平的；言下之意是，在actor的构造过程中从不同的发送者发来的入队请求（这里只是比喻，细节会更为复杂），也许会因低级别的线程调度导致重新排序。由于完全公平锁在JVM上并不存在，这是不可修复的。
- 路由器（更准确地说是路由ActorRef）的构造过程也是使用相同的机制，因此对使用路由部署的actor也存在同样的问题。
- 如上所述，在入队过程中任何涉及锁的地方都会有此问题，这也适用于自定义邮箱。

这份清单经过精心编制，但其他有问题的场景仍然可能会逃过我们的分析。

本地消息排序和网络消息排序如何关联

正如上一段所解释的，本地消息发送在一定条件下服从传递因果顺序。如果远程信息传输也遵从这个排序规则，这将转化为跨越单个网络链接的传递因果顺序，也就是说，如果正好只有两个网络主机参与。涉及多个环节则无法作此保证，如上面提到的位于三个不同节点的三个actor。

目前的远程传输不支持此排序规则（这同样是由于锁的唤醒顺序不满足

FIFO，此时是指连接建立的序列）。

从一个投机观点来看，未来有可能支持这种排序的保证，通过用actor完全重写远程传输层来实现；同时我们正在研究提供如UDP或SCTP的底层传输协议，这将带来更高的吞吐或更低的延迟，不过将再次删除此保证，这将意味着在不同的实现之间进行选择就是在顺序担保和性能之间进行折中。

更高层次的抽象

基于Akka的核心中小而一致的工具集，Akka也在其上提供了强大的，更高层次的抽象。

消息模式

上面讨论的实现可靠投递的问题，一个直截了当的答案是使用明确的ACK-RETRY协议。其最简单的形式需要

- 一种方法来识别个体信息，并将它与确认进行关联
- 一个重试机制，如果没有及时确认，将重新发送消息
- 一种接收方用来检测和丢弃重复消息的方法

第三步是必要的，因为确认消息本质上也是不能确保到达的。
一个企业级确认的ACK-RETRY协议，在Akka Persistence模块中以至少一次投递的方式支持了。至少一次投递的消息可以通过跟踪标识符的方式进行重复的检测。实现第三步的另一种方式是在业务逻辑中实现消息处理的幂等性（译者注：即每次消息处理的结果都是一样的）。

实现所有三个要求的另一个例子在可靠的代理模式中展示了（现在被至少一次投递所取代）。

事件源

事件源（和分片）使得大型网站能扩展到支持数以十亿计的用户，并且其想法很简单：当一个组件（想象为actor）处理一个命令，它会生成表示该命令效果的一组事件的列表。这些事件除了被应用到该组件的状态之外，也被存储。这个方案的好处是，事件永远只会被附加存储上，没有什么是可变的；这使得完美的复制和扩展这一事件流的消费者群体得到支持（即其他组件也可以消费这个事件流，只需要复制组件的状态到一个新的空间中，并且对变化进行反应即可）。如果组件的状态丢失——由于某台机器的故障，或者是被淘汰出缓存——它仍然可以很容易地通过重播事件流（通常使用快照来加快处理）进行重建。`event-sourcing`由Akka Persistence支持。

具有明确确认功能的邮箱

通过实现自定义邮箱类型，有可能在接收actor结束时重试消息处理，以处理临时故障。这种模式一般在本地通信上下文非常有用，否则投递担保不足以满足应用程序的需求。

请注意，”JVM内（本地）消息发送规则“中的警告仍然有效。

实现这种模式的示例展示在邮箱确认中。

死信

不能被投递（并且可以被确定没有投递成功）的消息，会被投递到一个名为`/deadLetters`的人造actor。该投递以尽力而为为基础；它甚至可以在本地JVM中失败（例如actor终止时）。在不可靠的网络传输丢失的消息将会被丢弃，而不会作为死信处理。

死信应该被用来做什么？

这个组件的主要用途是调试，特别是如果一个actor的发送始终没有送达的时候（通常查看死信，你会发现发件者或接收者在某个环节上设置

错了）。为了能更好地用于该目的，最好的实践是，尽可能避免发送消息到deadLetters，即使用一个合适的死信日志记录器（详见下文）来运行应用程序，并时不时地清理日志输出。这个实践—和其他所有实践类似—需要按照常识构建明智的应用程序：有可能避免发送消息给一个已终止的actor，给发送者代码增加的复杂性，多于调试输出带来的清晰度。

死信服务与其他所有消息投递一样，对于投递保证遵循相同的规则的，因此它不能被用来实现投递保证。

怎样接收死信？

一个actor可以在事件流中订阅类 `akka.actor.DeadLetter` ，请参阅事件流（java）或事件流（scala）来了解如何做到这一点。那么订阅的actor会从该点起收到（本地）系统中发布的所有死信。死信不会在网络上传播，如果你想在一个地方收集他们，你将不得不在每个网络节点上使用一个actor订阅，并手动转发。同时注意，在该节点上生成的死信，能够确定一个发送操作失败，这对于远程发送来说，可以是本地系统（如果不能建立网络连接）也可以是远程系统（如果你要发送的目标actor在该时间点不存在的话）。

（通常）不用担心死信

每当一个actor不是通过自身决定终止的，则存在这样的可能—它发送到自身的一些消息丢失了。这在复杂关闭场景下是很容易发生的，而这些场景通常也是良性的：看到一个 `akka.dispatch.Terminate` 消息被丢弃意味着，两个中断请求被发送，当然只有一个可以成功。同样道理，在停止一个子树的actor时，如果父节点在终止的时候仍然观察着子节点，则你可能会看到从孩子发出的 `akka.actor.Terminated` 消息会转变为死信。

配置

配置

你开始使用Akka时可以不需要定义任何配置，因为Akka提供了合理的

默认值。不过为了适应特定的运行环境，你可能需要修改设置来更改默认行为。可以修改的典型设置的例子：

- 日志级别和后端记录器
- 启用远程
- 消息序列化
- 路由器定义
- 调度的调整

Akka使用Typesafe配置库，它同样也是你自己的应用或库的不错选择，不管你用不用Akka。这个库由Java实现，没有外部的依赖；本文后面将只对这个库有一些归纳，你应该查看其文档参考具体的使用（尤其是`ConfigFactory`）。

> 警告
>
> 如果你在*Scala REPL*的*2.9.x*系列版本中使用*Akka*，并且你不提供自己的*ClassLoader*给*ActorSystem*，则需要以*"-Yrepl-sync"*选项启动*REPL*来解决上下文*ClassLoader*的缺陷。

配置读取的地方

Akka的所有配置都保存在 `ActorSystem` 的实例中，或者换一种说法，从外界来看， `ActorSystem` 是配置信息的唯一消费者。在构造一个actor系统时，你可以选择传进一个 `Config` 对象，如果不传则等效于传入 `ConfigFactory.load()` （通过正确的类加载器）。粗略的讲，这意味着默认会解析classpath根目录下所有的 `application.conf` ， `application.json` 和 `application.properties` 文件——请参考前面提到的文档以获取细节。然后actor系统会合并classpath根目录下的所有 `reference.conf` 行成后备配置，也就是说，它在内部使用

```
1.
```

```
    appConfig.withFallback(ConfigFactory.defaultReference(classLoader))
```

其哲学是代码永远不包含缺省值，相反是依赖于随库提供的 `reference.conf` 中的配置。

系统属性中覆盖的配置具有最高优先级，参见HOCON 规范（靠近末尾的位置）。此外值得注意的是，应用程序配置—缺省为 `application` —可以使用 `config.resource` 属性重写 （还有更多，请参阅配置文档）。

> 注意
>
> 如果你正在编写一个*Akka* 应用，将你的配置保存类路径的根目录下的 *`application.conf`* 文件中。如果你正在编写一个基于*Akka*的库，将其配置保存在*JAR*包根目录下的 *`reference.conf`* 文件中。

当使用JarJar,, OneJar, Assembly或任何jar打包命令（jar-bundler）

> 警告
>
> *Akka*的配置方法重度依赖于这个理念—每一模块/*jar*都有它自己的 *`reference.conf`* 文件，所有这些都将会被配置发现并加载。不幸的是，这也意味着如果你放置/合并多个*jar*到相同的 *jar*中，你也需要合并所有的 *`reference.conf`* 文件。否则所有的默认设置将会丢失，*Akka*将无法工作。

如果你使用 Maven 打包应用程序，你还可以使用Apache Maven Shade Plugin中对资源转换（Resource Transformers）的支持，来将所有构建类路径中的 `reference.conf` 合并为一个文件。

插件配置可能如下所示：

```
1.      <plugin>
2.       <groupId>org.apache.maven.plugins</groupId>
3.       <artifactId>maven-shade-plugin</artifactId>
```

```
      <version>1.5</version>
      <executions>
       <execution>
        <phase>package</phase>
        <goals>
         <goal>shade</goal>
        </goals>
        <configuration>
         <shadedArtifactAttached>true</shadedArtifactAttached>
         <shadedClassifierName>allinone</shadedClassifierName>
         <artifactSet>
          <includes>
           <include>*:*</include>
          </includes>
         </artifactSet>
         <transformers>
           <transformer
implementation="org.apache.maven.plugins.shade.resource.AppendingTran
            <resource>reference.conf</resource>
           </transformer>
           <transformer
implementation="org.apache.maven.plugins.shade.resource.ManifestResou
            <manifestEntries>
             <Main-Class>akka.Main</Main-Class>
            </manifestEntries>
           </transformer>
         </transformers>
        </configuration>
       </execution>
      </executions>
     </plugin>
```

自定义application.conf

一个自定义的 `application.conf` 可能看起来像这样:

```
  # In this file you can override any option defined in the reference files.
  # Copy in parts of the reference files and modify as you please.

  akka {

    # Loggers to register at boot time (akka.event.Logging$DefaultLogger logs
    # to STDOUT)
    loggers = ["akka.event.slf4j.Slf4jLogger"]

    # Log level used by the configured loggers (see "loggers") as soon
    # as they have been started; before that, see "stdout-loglevel"
    # Options: OFF, ERROR, WARNING, INFO, DEBUG
    loglevel = "DEBUG"

    # Log level for the very basic logger activated during ActorSystem startup.
    # This logger prints the log messages to stdout (System.out).
    # Options: OFF, ERROR, WARNING, INFO, DEBUG
    stdout-loglevel = "DEBUG"

    actor {
      provider = "akka.cluster.ClusterActorRefProvider"

      default-dispatcher {
        # Throughput for default Dispatcher, set to 1 for as fair as possible
        throughput = 10
      }
    }

    remote {
      # The port clients should connect to. Default is 2552.
      netty.tcp.port = 4711
    }
  }
```

包含文件" class="reference-link">包含文件

有时包含另一个配置文件内容的能力是非常有用的，例如假设你有一个 `application.conf` 包含所有环境独立设置，然后使用特定环境的设置覆写。

用 `-Dconfig.resource=/dev.conf` 制定系统属性，将会加载 `dev.conf` 文件，并包含 `application.conf`

dev.conf:

```
1.  include "application"
2.
3.  akka {
4.    loglevel = "DEBUG"
5.  }
```

更高级的包括和替换机制的解释在HOCON规范中。

配置日志

如果系统或配置属性 `akka.log-config-on-start` 被设置为 `on`，则在actor系统启动的时候，就完成了INFO级别的日志设置。当你不能确定使用何种配置时，这很有用。

如果有疑问，你也可以在创建actor系统之前或之后，很容易很方便地检查配置对象：

```
1.  Welcome to Scala version @scalaVersion@ (Java HotSpot(TM) 64-Bit
    Server VM, Java 1.6.0_27).
2.  Type in expressions to have them evaluated.
3.  Type :help for more information.
4.
5.  scala> import com.typesafe.config._
```

```
6.    import com.typesafe.config._
7.
8.    scala> ConfigFactory.parseString("a.b=12")
9.    res0: com.typesafe.config.Config = Config(SimpleConfigObject({"a"
    : {"b" : 12}}))
10.
11.   scala> res0.root.render
12.   res1: java.lang.String =
13.   {
14.       # String: 1
15.       "a" : {
16.           # String: 1
17.           "b" : 12
18.       }
19.   }
```

展示结果中，每个项目前会有评论展示这个配置的起源（对应的文件和行数），并展示已存在的评论，如配置参考中的。actor系统合并参考并解析后形成的设置，可以这样显示：

```
1.    final ActorSystem system = ActorSystem.create();
2.    System.out.println(system.settings());
3.    // this is a shortcut for
    system.settings().config().root().render()
```

谈一谈类加载器

在配置文件的几个地方，可以通过制定类的全名来让Akka实例化该类。这是通过Java反射完成的，相应地用到了一个 `ClassLoader` 。在具有挑战性的环境中，如应用容器和OSGi绑定中，选择正确的类加载器并不总是一件简单的事情，Akka的现行做法是每个 `ActorSystem` 实现存储当前线程的上下文类加载器（如果可用，否则就使用他自己的加载器 `this.getClass.getClassLoader` ），并使用它为所有的反射访问服务。这意味着Akka放在引导类路径（boot class path）下，会从

奇怪的地方产生 `NullPointerException` ： 这里就是不支持。

应用特定设置

配置也可用于应用程序特定的设置。一个好的实践是将这些设置放在一个扩展中，像下面的章节所描述的:

- Scala API: 应用特定设置
- Java API: 应用特定设置

配置多个ActorSystem

如果你有一个以上的 `ActorSystem` （或你正在写一个库，有可能有一个独立于应用的 `ActorSystem` ） 你可能想要为每个系统进行单独配置。

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并在配置树中区分actor系统:

```
1.  myapp1 {
2.    akka.loglevel = "WARNING"
3.    my.own.setting = 43
4.  }
5.  myapp2 {
6.    akka.loglevel = "ERROR"
7.    app2.setting = "appname"
8.  }
9.  my.own.setting = 42
10.  my.other.setting = "hello"
```

```
1.  val config = ConfigFactory.load()
2.  val app1 = ActorSystem("MyApp1",
    config.getConfig("myapp1").withFallback(config))
3.  val app2 = ActorSystem("MyApp2",
4.
    config.getConfig("myapp2").withOnlyPath("akka").withFallback(config))
```

这两个例子演示了“提升子树”技巧的不同变种： 第一种情况下，actor系统获得的配置是

```
    akka.loglevel = "WARNING"
    my.own.setting = 43
    my.other.setting = "hello"
    // plus myapp1 and myapp2 subtrees
```

而在第二种情况下，只有 “akka” 子树被提升了，结果如下：

```
    akka.loglevel = "ERROR"
    my.own.setting = 42
    my.other.setting = "hello"
    // plus myapp1 and myapp2 subtrees
```

> 注意
>
> 这个配置文件库非常强大，这里不可能解释其所有的功能。特别是如何在配置文件中包含其它的配置文件（在 *包含文件Including files* 中有一个简单的例子）以及通过路径替换来复制部分配置树。

你也可以在初始化 `ActorSystem` 时，通过代码的形式，使用其它方法来指定和解析配置信息。

```
import akka.actor.ActorSystem
import com.typesafe.config.ConfigFactory
    val customConf = ConfigFactory.parseString("""
      akka.actor.deployment {
        /my-service {
          router = round-robin-pool
          nr-of-instances = 3
        }
      }
      """)
    // ConfigFactory.load sandwiches customConfig between default reference
```

```
12.     // config and default overrides, and then resolves it.
13.     val system = ActorSystem("MySystem",
    ConfigFactory.load(customConf))
```

从自定义位置读取配置

你可以使用代码或系统属性，来替换或补充 `application.conf` 。

如果你使用的方法是 `ConfigFactory.load()` （Akka默认方式），你可以通过定义 `-Dconfig.resource=whatever` 、 `-Dconfig.file=whatever` 或 `-Dconfig.url=whatever` 替换 `application.conf` 。

在 `-Dconfig.resource` 和相关选项指定的替换配置文件中，如果你还想使用 `application.{conf,json,properties}` ，可以使用 `include "application"` 。在 `include "application"` 之前指定的设置会被包含进来的文件内容重写，同理所包含文件的内容也会被之后的内容重写。

在代码中，有很多自定义选项。

`ConfigFactory.load()` 有几个重载；这些重载允许你指定夹在 系统属性（重写）和默认值（来自 `reference.conf` ）之间的配置，并替换通常的 `application.{conf,json,properties}` 和 `-Dconfig.file` 相关选项。

`ConfigFactory.load()` 最简单的变体需要资源基本名称（ `application` 之外的）；如 `myname.conf` 、 `myname.json` 和 `myname.properties` 而不是 `application.{conf,json,properties}` 。

最灵活的变体是以一个 `Config` 对象为参数，你可以使用 `ConfigFactory` 中的任何方法加载。例如，你可以在代码中使用 `ConfigFactory.parseString()` 处理一个配置字符串，或者你可以使用 `ConfigFactory.parseMap()` 创建一个映射，或者也可以加载一个文

件。

你也可以将自定义的配置与通常的配置组合起来，像这样：

```
// make a Config with just your special setting
Config myConfig =
  ConfigFactory.parseString("something=somethingElse");
// load the normal config stack (system props,
// then application.conf, then reference.conf)
Config regularConfig =
  ConfigFactory.load();
// override regular stack with myConfig
Config combined =
  myConfig.withFallback(regularConfig);
// put the result in between the overrides
// (system props) and defaults again
Config complete =
  ConfigFactory.load(combined);
// create ActorSystem
ActorSystem system =
  ActorSystem.create("myname", complete);
```

使用 `Config` 对象时，请牢记这个蛋糕有三”层”：

- `ConfigFactory.defaultOverrides()`（系统属性）
- 应用设置
- `ConfigFactory.defaultReference()`（reference.conf）

正常的目标是要自定义中间一层，不管其他两个。

- `ConfigFactory.load()` 加载整个堆栈
- `ConfigFactory.load()` 的重载允许你指定一个不同的中间层
- `ConfigFactory.parse()` 变体加载单个文件或资源

要叠加两层，可使用 `override.withFallback(fallback)`；请努力保持系统属性（`defaultOverrides()`）在顶部，`reference.conf`

（ `defaultReference()` ）在底部。

要记住，通常你只需要在 `application.conf` 添加一个 `include` 语句，而不是编写代码。在 `application.conf` 顶部引入的将被 `application.conf` 其余部分覆盖，而那些在底部的设置将覆盖以前的内容。

Actor部署配置

可以在配置的 `akka.actor.deployment` 节中定义特定actor的部署设置。在部署部分有可能定义这些事物——调度器、邮箱、路由器设置和远程部署。在相应主题的章节中详细介绍了配置的这些特性。一个例子，可以如下所示：

```
akka.actor.deployment {

  # '/user/actorA/actorB' is a remote deployed actor
  /actorA/actorB {
    remote = "akka.tcp://sampleActorSystem@127.0.0.1:2553"
  }

  # all direct children of '/user/actorC' have a dedicated dispatcher
  "/actorC/*" {
    dispatcher = my-dispatcher
  }

  # '/user/actorD/actorE' has a special priority mailbox
  /actorD/actorE {
    mailbox = prio-mailbox
  }

  # '/user/actorF/actorG/actorH' is a random pool
  /actorF/actorG/actorH {
    router = random-pool
```

```
    nr-of-instances = 5
  }
}

my-dispatcher {
  fork-join-executor.parallelism-min = 10
  fork-join-executor.parallelism-max = 10
}
prio-mailbox {
  mailbox-type = "a.b.MyPrioMailbox"
}
```

一个指定actor部署部分的设置是通过其相对 `/user` 的路径来标识的。

你可以使用星号作为通配符匹配actor的路径部分，所以你可以指定: `/*/sampleActor` 将匹配该树形结构中那个级别上的所有 `sampleActor` 。你也能把通配符放在最后来匹配某一级别的所有actor: `/someParent/*` 。非通配符匹配总是有更高的优先级，所以: `/foo/bar` 比 `/foo/*` 更具体，并且只有最高优先的匹配才会被使用。请注意它不能用于部分匹配，像这样: `/foo*/bar` 、 `/f*o/bar` 等。

参考配置清单

每个Akka模块都有保存默认值的“reference”配置文件。

akka-actor

```
#####################################
# Akka Actor Reference Config File #
#####################################

# This is the reference config file that contains all the default
settings.
```

```
6.  # Make your edits/overrides in your application.conf.
7.
8.  akka {
9.    # Akka version, checked against the runtime version of Akka.
10.   version = "2.3.6"
11.
12.   # Home directory of Akka, modules in the deploy directory will be
    loaded
13.   home = ""
14.
15.   # Loggers to register at boot time
    (akka.event.Logging$DefaultLogger logs
16.   # to STDOUT)
17.   loggers = ["akka.event.Logging$DefaultLogger"]
18.
19.   # Loggers are created and registered synchronously during
    ActorSystem
20.   # start-up, and since they are actors, this timeout is used to
    bound the
21.   # waiting time
22.   logger-startup-timeout = 5s
23.
24.   # Log level used by the configured loggers (see "loggers") as
    soon
25.   # as they have been started; before that, see "stdout-loglevel"
26.   # Options: OFF, ERROR, WARNING, INFO, DEBUG
27.   loglevel = "INFO"
28.
29.   # Log level for the very basic logger activated during
    ActorSystem startup.
30.   # This logger prints the log messages to stdout (System.out).
31.   # Options: OFF, ERROR, WARNING, INFO, DEBUG
32.   stdout-loglevel = "WARNING"
33.
34.   # Log the complete configuration at INFO level when the actor
    system is started.
35.   # This is useful when you are uncertain of what configuration is
    used.
```

```
36.   log-config-on-start = off
37.
38.   # Log at info level when messages are sent to dead letters.
39.   # Possible values:
40.   # on: all dead letters are logged
41.   # off: no logging of dead letters
42.   # n: positive integer, number of dead letters that will be logged
43.   log-dead-letters = 10
44.
45.   # Possibility to turn off logging of dead letters while the actor
      system
46.   # is shutting down. Logging is only done when enabled by 'log-
      dead-letters'
47.   # setting.
48.   log-dead-letters-during-shutdown = on
49.
50.   # List FQCN of extensions which shall be loaded at actor system
      startup.
51.   # Should be on the format: 'extensions = ["foo", "bar"]' etc.
52.   # See the Akka Documentation for more info about Extensions
53.   extensions = []
54.
55.   # Toggles whether threads created by this ActorSystem should be
      daemons or not
56.   daemonic = off
57.
58.   # JVM shutdown, System.exit(-1), in case of a fatal error,
59.   # such as OutOfMemoryError
60.   jvm-exit-on-fatal-error = on
61.
62.   actor {
63.
64.     # FQCN of the ActorRefProvider to be used; the below is the
      built-in default,
65.     # another one is akka.remote.RemoteActorRefProvider in the
      akka-remote bundle.
66.     provider = "akka.actor.LocalActorRefProvider"
67.
```

```
68.     # The guardian "/user" will use this class to obtain its
    supervisorStrategy.
69.     # It needs to be a subclass of
    akka.actor.SupervisorStrategyConfigurator.
70.     # In addition to the default there is
    akka.actor.StoppingSupervisorStrategy.
71.     guardian-supervisor-strategy =
    "akka.actor.DefaultSupervisorStrategy"
72.
73.     # Timeout for ActorSystem.actorOf
74.     creation-timeout = 20s
75.
76.     # Frequency with which stopping actors are prodded in case they
    had to be
77.     # removed from their parents
78.     reaper-interval = 5s
79.
80.     # Serializes and deserializes (non-primitive) messages to
    ensure immutability,
81.     # this is only intended for testing.
82.     serialize-messages = off
83.
84.     # Serializes and deserializes creators (in Props) to ensure
    that they can be
85.     # sent over the network, this is only intended for testing.
    Purely local deployments
86.     # as marked with deploy.scope == LocalScope are exempt from
    verification.
87.     serialize-creators = off
88.
89.     # Timeout for send operations to top-level actors which are in
    the process
90.     # of being started. This is only relevant if using a bounded
    mailbox or the
91.     # CallingThreadDispatcher for a top-level actor.
92.     unstarted-push-timeout = 10s
93.
94.     typed {
```

```
95.       # Default timeout for typed actor methods with non-void
    return type
96.       timeout = 5s
97.     }
98.
99.     # Mapping between ´deployment.router' short names to fully
    qualified class names
100.    router.type-mapping {
101.      from-code = "akka.routing.NoRouter"
102.      round-robin-pool = "akka.routing.RoundRobinPool"
103.      round-robin-group = "akka.routing.RoundRobinGroup"
104.      random-pool = "akka.routing.RandomPool"
105.      random-group = "akka.routing.RandomGroup"
106.      balancing-pool = "akka.routing.BalancingPool"
107.      smallest-mailbox-pool = "akka.routing.SmallestMailboxPool"
108.      broadcast-pool = "akka.routing.BroadcastPool"
109.      broadcast-group = "akka.routing.BroadcastGroup"
110.      scatter-gather-pool =
    "akka.routing.ScatterGatherFirstCompletedPool"
111.      scatter-gather-group =
    "akka.routing.ScatterGatherFirstCompletedGroup"
112.      tail-chopping-pool = "akka.routing.TailChoppingPool"
113.      tail-chopping-group = "akka.routing.TailChoppingGroup"
114.      consistent-hashing-pool =
    "akka.routing.ConsistentHashingPool"
115.      consistent-hashing-group =
    "akka.routing.ConsistentHashingGroup"
116.    }
117.
118.    deployment {
119.
120.      # deployment id pattern - on the format: /parent/child etc.
121.      default {
122.
123.        # The id of the dispatcher to use for this actor.
124.        # If undefined or empty the dispatcher specified in code
125.        # (Props.withDispatcher) is used, or default-dispatcher if
    not
```

```
126.         # specified at all.
127.         dispatcher = ""
128.
129.         # The id of the mailbox to use for this actor.
130.         # If undefined or empty the default mailbox of the
     configured dispatcher
131.         # is used or if there is no mailbox configuration the
     mailbox specified
132.         # in code (Props.withMailbox) is used.
133.         # If there is a mailbox defined in the configured
     dispatcher then that
134.         # overrides this setting.
135.         mailbox = ""
136.
137.         # routing (load-balance) scheme to use
138.         # - available: "from-code", "round-robin", "random",
     "smallest-mailbox",
139.         #              "scatter-gather", "broadcast"
140.         # - or:        Fully qualified class name of the router
     class.
141.         #              The class must extend
     akka.routing.CustomRouterConfig and
142.         #              have a public constructor with
     com.typesafe.config.Config
143.         #              and optional akka.actor.DynamicAccess
     parameter.
144.         # - default is "from-code";
145.         # Whether or not an actor is transformed to a Router is
     decided in code
146.         # only (Props.withRouter). The type of router can be
     overridden in the
147.         # configuration; specifying "from-code" means that the
     values specified
148.         # in the code shall be used.
149.         # In case of routing, the actors to be routed to can be
     specified
150.         # in several ways:
151.         # - nr-of-instances: will create that many children
```

```
152.         # - routees.paths: will route messages to these paths using
     ActorSelection,
153.         #   i.e. will not create children
154.         # - resizer: dynamically resizable number of routees as
     specified in
155.         #   resizer below
156.         router = "from-code"
157.
158.         # number of children to create in case of a router;
159.         # this setting is ignored if routees.paths is given
160.         nr-of-instances = 1
161.
162.         # within is the timeout used for routers containing future
     calls
163.         within = 5 seconds
164.
165.         # number of virtual nodes per node for consistent-hashing
     router
166.         virtual-nodes-factor = 10
167.
168.         tail-chopping-router {
169.           # interval is duration between sending message to next
     routee
170.           interval = 10 milliseconds
171.         }
172.
173.         routees {
174.           # Alternatively to giving nr-of-instances you can specify
     the full
175.           # paths of those actors which should be routed to. This
     setting takes
176.           # precedence over nr-of-instances
177.           paths = []
178.         }
179.
180.         # To use a dedicated dispatcher for the routees of the pool
     you can
181.         # define the dispatcher configuration inline with the
```

```
property name
182.          # 'pool-dispatcher' in the deployment section of the
router.
183.          # For example:
184.          # pool-dispatcher {
185.          #   fork-join-executor.parallelism-min = 5
186.          #   fork-join-executor.parallelism-max = 5
187.          # }
188.
189.          # Routers with dynamically resizable number of routees;
this feature is
190.          # enabled by including (parts of) this section in the
deployment
191.          resizer {
192.
193.            enabled = off
194.
195.            # The fewest number of routees the router should ever
have.
196.            lower-bound = 1
197.
198.            # The most number of routees the router should ever have.
199.            # Must be greater than or equal to lower-bound.
200.            upper-bound = 10
201.
202.            # Threshold used to evaluate if a routee is considered to
be busy
203.            # (under pressure). Implementation depends on this value
(default is 1).
204.            # 0:   number of routees currently processing a message.
205.            # 1:   number of routees currently processing a message
has
206.            #      some messages in mailbox.
207.            # > 1: number of routees with at least the configured
pressure-threshold
208.            #      messages in their mailbox. Note that estimating
mailbox size of
209.            #      default UnboundedMailbox is O(N) operation.
```

```
210.           pressure-threshold = 1
211.
212.           # Percentage to increase capacity whenever all routees
     are busy.
213.           # For example, 0.2 would increase 20% (rounded up), i.e.
     if current
214.           # capacity is 6 it will request an increase of 2 more
     routees.
215.           rampup-rate = 0.2
216.
217.           # Minimum fraction of busy routees before backing off.
218.           # For example, if this is 0.3, then we'll remove some
     routees only when
219.           # less than 30% of routees are busy, i.e. if current
     capacity is 10 and
220.           # 3 are busy then the capacity is unchanged, but if 2 or
     less are busy
221.           # the capacity is decreased.
222.           # Use 0.0 or negative to avoid removal of routees.
223.           backoff-threshold = 0.3
224.
225.           # Fraction of routees to be removed when the resizer
     reaches the
226.           # backoffThreshold.
227.           # For example, 0.1 would decrease 10% (rounded up), i.e.
     if current
228.           # capacity is 9 it will request an decrease of 1 routee.
229.           backoff-rate = 0.1
230.
231.           # Number of messages between resize operation.
232.           # Use 1 to resize before each message.
233.           messages-per-resize = 10
234.         }
235.       }
236.     }
237.
238.     default-dispatcher {
239.       # Must be one of the following
```

```
240.       # Dispatcher, PinnedDispatcher, or a FQCN to a class
     inheriting
241.       # MessageDispatcherConfigurator with a public constructor
     with
242.       # both com.typesafe.config.Config parameter and
243.       # akka.dispatch.DispatcherPrerequisites parameters.
244.       # PinnedDispatcher must be used together with
     executor=thread-pool-executor.
245.       type = "Dispatcher"
246.
247.       # Which kind of ExecutorService to use for this dispatcher
248.       # Valid options:
249.       #  - "default-executor" requires a "default-executor" section
250.       #  - "fork-join-executor" requires a "fork-join-executor"
     section
251.       #  - "thread-pool-executor" requires a "thread-pool-executor"
     section
252.       #  - A FQCN of a class extending ExecutorServiceConfigurator
253.       executor = "default-executor"
254.
255.       # This will be used if you have set "executor = "default-
     executor"".
256.       # If an ActorSystem is created with a given ExecutionContext,
     this
257.       # ExecutionContext will be used as the default executor for
     all
258.       # dispatchers in the ActorSystem configured with
259.       # executor = "default-executor". Note that "default-executor"
260.       # is the default value for executor, and therefore used if
     not
261.       # specified otherwise. If no ExecutionContext is given,
262.       # the executor configured in "fallback" will be used.
263.       default-executor {
264.         fallback = "fork-join-executor"
265.       }
266.
267.       # This will be used if you have set "executor = "fork-join-
     executor""
```

```
268.       fork-join-executor {
269.         # Min number of threads to cap factor-based parallelism
     number to
270.         parallelism-min = 8
271.
272.         # The parallelism factor is used to determine thread pool
     size using the
273.         # following formula: ceil(available processors * factor).
     Resulting size
274.         # is then bounded by the parallelism-min and parallelism-
     max values.
275.         parallelism-factor = 3.0
276.
277.         # Max number of threads to cap factor-based parallelism
     number to
278.         parallelism-max = 64
279.       }
280.
281.       # This will be used if you have set "executor = "thread-pool-
     executor""
282.       thread-pool-executor {
283.         # Keep alive time for threads
284.         keep-alive-time = 60s
285.
286.         # Min number of threads to cap factor-based core number to
287.         core-pool-size-min = 8
288.
289.         # The core pool size factor is used to determine thread
     pool core size
290.         # using the following formula: ceil(available processors *
     factor).
291.         # Resulting size is then bounded by the core-pool-size-min
     and
292.         # core-pool-size-max values.
293.         core-pool-size-factor = 3.0
294.
295.         # Max number of threads to cap factor-based number to
296.         core-pool-size-max = 64
```

```
297.
298.         # Minimum number of threads to cap factor-based max number
     to
299.         # (if using a bounded task queue)
300.         max-pool-size-min = 8
301.
302.         # Max no of threads (if using a bounded task queue) is
     determined by
303.         # calculating: ceil(available processors * factor)
304.         max-pool-size-factor  = 3.0
305.
306.         # Max number of threads to cap factor-based max number to
307.         # (if using a  bounded task queue)
308.         max-pool-size-max = 64
309.
310.         # Specifies the bounded capacity of the task queue (< 1 ==
     unbounded)
311.         task-queue-size = -1
312.
313.         # Specifies which type of task queue will be used, can be
     "array" or
314.         # "linked" (default)
315.         task-queue-type = "linked"
316.
317.         # Allow core threads to time out
318.         allow-core-timeout = on
319.       }
320.
321.       # How long time the dispatcher will wait for new actors until
     it shuts down
322.       shutdown-timeout = 1s
323.
324.       # Throughput defines the number of messages that are
     processed in a batch
325.       # before the thread is returned to the pool. Set to 1 for as
     fair as possible.
326.       throughput = 5
327.
```

```
328.       # Throughput deadline for Dispatcher, set to 0 or negative
    for no deadline
329.       throughput-deadline-time = 0ms
330.
331.       # For BalancingDispatcher: If the balancing dispatcher should
    attempt to
332.       # schedule idle actors using the same dispatcher when a
    message comes in,
333.       # and the dispatchers ExecutorService is not fully busy
    already.
334.       attempt-teamwork = on
335.
336.       # If this dispatcher requires a specific type of mailbox,
    specify the
337.       # fully-qualified class name here; the actually created
    mailbox will
338.       # be a subtype of this type. The empty string signifies no
    requirement.
339.       mailbox-requirement = ""
340.     }
341.
342.     default-mailbox {
343.       # FQCN of the MailboxType. The Class of the FQCN must have a
    public
344.       # constructor with
345.       # (akka.actor.ActorSystem.Settings,
    com.typesafe.config.Config) parameters.
346.       mailbox-type = "akka.dispatch.UnboundedMailbox"
347.
348.       # If the mailbox is bounded then it uses this setting to
    determine its
349.       # capacity. The provided value must be positive.
350.       # NOTICE:
351.       # Up to version 2.1 the mailbox type was determined based on
    this setting;
352.       # this is no longer the case, the type must explicitly be a
    bounded mailbox.
353.       mailbox-capacity = 1000
```

```
354.
355.       # If the mailbox is bounded then this is the timeout for
     enqueueing
356.       # in case the mailbox is full. Negative values signify
     infinite
357.       # timeout, which should be avoided as it bears the risk of
     dead-lock.
358.       mailbox-push-timeout-time = 10s
359.
360.       # For Actor with Stash: The default capacity of the stash.
361.       # If negative (or zero) then an unbounded stash is used
     (default)
362.       # If positive then a bounded stash is used and the capacity
     is set using
363.       # the property
364.       stash-capacity = -1
365.     }
366.
367.     mailbox {
368.       # Mapping between message queue semantics and mailbox
     configurations.
369.       # Used by akka.dispatch.RequiresMessageQueue[T] to enforce
     different
370.       # mailbox types on actors.
371.       # If your Actor implements RequiresMessageQueue[T], then when
     you create
372.       # an instance of that actor its mailbox type will be decided
     by looking
373.       # up a mailbox configuration via T in this mapping
374.       requirements {
375.         "akka.dispatch.UnboundedMessageQueueSemantics" =
376.           akka.actor.mailbox.unbounded-queue-based
377.         "akka.dispatch.BoundedMessageQueueSemantics" =
378.           akka.actor.mailbox.bounded-queue-based
379.         "akka.dispatch.DequeBasedMessageQueueSemantics" =
380.           akka.actor.mailbox.unbounded-deque-based
381.         "akka.dispatch.UnboundedDequeBasedMessageQueueSemantics" =
382.           akka.actor.mailbox.unbounded-deque-based
```

```
383.         "akka.dispatch.BoundedDequeBasedMessageQueueSemantics" =
384.           akka.actor.mailbox.bounded-deque-based
385.         "akka.dispatch.MultipleConsumerSemantics" =
386.           akka.actor.mailbox.unbounded-queue-based
387.       }
388.
389.       unbounded-queue-based {
390.         # FQCN of the MailboxType, The Class of the FQCN must have
    a public
391.         # constructor with (akka.actor.ActorSystem.Settings,
392.         # com.typesafe.config.Config) parameters.
393.         mailbox-type = "akka.dispatch.UnboundedMailbox"
394.       }
395.
396.       bounded-queue-based {
397.         # FQCN of the MailboxType, The Class of the FQCN must have
    a public
398.         # constructor with (akka.actor.ActorSystem.Settings,
399.         # com.typesafe.config.Config) parameters.
400.         mailbox-type = "akka.dispatch.BoundedMailbox"
401.       }
402.
403.       unbounded-deque-based {
404.         # FQCN of the MailboxType, The Class of the FQCN must have
    a public
405.         # constructor with (akka.actor.ActorSystem.Settings,
406.         # com.typesafe.config.Config) parameters.
407.         mailbox-type = "akka.dispatch.UnboundedDequeBasedMailbox"
408.       }
409.
410.       bounded-deque-based {
411.         # FQCN of the MailboxType, The Class of the FQCN must have
    a public
412.         # constructor with (akka.actor.ActorSystem.Settings,
413.         # com.typesafe.config.Config) parameters.
414.         mailbox-type = "akka.dispatch.BoundedDequeBasedMailbox"
415.       }
416.     }
```

```
417.
418.     debug {
419.       # enable function of Actor.loggable(), which is to log any
     received message
420.       # at DEBUG level, see the “Testing Actor Systems” section of
     the Akka
421.       # Documentation at http://akka.io/docs
422.       receive = off
423.
424.       # enable DEBUG logging of all AutoReceiveMessages (Kill,
     PoisonPill et.c.)
425.       autoreceive = off
426.
427.       # enable DEBUG logging of actor lifecycle changes
428.       lifecycle = off
429.
430.       # enable DEBUG logging of all LoggingFSMs for events,
     transitions and timers
431.       fsm = off
432.
433.       # enable DEBUG logging of subscription changes on the
     eventStream
434.       event-stream = off
435.
436.       # enable DEBUG logging of unhandled messages
437.       unhandled = off
438.
439.       # enable WARN logging of misconfigured routers
440.       router-misconfiguration = off
441.     }
442.
443.     # Entries for pluggable serializers and their bindings.
444.     serializers {
445.       java = "akka.serialization.JavaSerializer"
446.       bytes = "akka.serialization.ByteArraySerializer"
447.     }
448.
449.     # Class to Serializer binding. You only need to specify the
```

```
name of an
450.     # interface or abstract base class of the messages. In case of
ambiguity it
451.     # is using the most specific configured class, or giving a
warning and
452.     # choosing the “first” one.
453.     #
454.     # To disable one of the default serializers, assign its class
to "none", like
455.     # "java.io.Serializable" = none
456.     serialization-bindings {
457.       "[B" = bytes
458.       "java.io.Serializable" = java
459.     }
460.
461.     # Configuration items which are used by the
akka.actor.ActorDSL._ methods
462.     dsl {
463.       # Maximum queue size of the actor created by newInbox(); this
protects
464.       # against faulty programs which use select() and consistently
miss messages
465.       inbox-size = 1000
466.
467.       # Default timeout to assume for operations like Inbox.receive
et al
468.       default-timeout = 5s
469.     }
470.   }
471.
472.   # Used to set the behavior of the scheduler.
473.   # Changing the default values may change the system behavior
drastically so make
474.   # sure you know what you're doing! See the Scheduler section of
the Akka
475.   # Documentation for more details.
476.   scheduler {
477.     # The LightArrayRevolverScheduler is used as the default
```

```
    scheduler in the
478.     # system. It does not execute the scheduled tasks on exact
    time, but on every
479.     # tick, it will run everything that is (over)due. You can
    increase or decrease
480.     # the accuracy of the execution timing by specifying smaller or
    larger tick
481.     # duration. If you are scheduling a lot of tasks you should
    consider increasing
482.     # the ticks per wheel.
483.     # Note that it might take up to 1 tick to stop the Timer, so
    setting the
484.     # tick-duration to a high value will make shutting down the
    actor system
485.     # take longer.
486.     tick-duration = 10ms
487.
488.     # The timer uses a circular wheel of buckets to store the timer
    tasks.
489.     # This should be set such that the majority of scheduled
    timeouts (for high
490.     # scheduling frequency) will be shorter than one rotation of
    the wheel
491.     # (ticks-per-wheel * ticks-duration)
492.     # THIS MUST BE A POWER OF TWO!
493.     ticks-per-wheel = 512
494.
495.     # This setting selects the timer implementation which shall be
    loaded at
496.     # system start-up.
497.     # The class given here must implement the akka.actor.Scheduler
    interface
498.     # and offer a public constructor which takes three arguments:
499.     #  1) com.typesafe.config.Config
500.     #  2) akka.event.LoggingAdapter
501.     #  3) java.util.concurrent.ThreadFactory
502.     implementation = akka.actor.LightArrayRevolverScheduler
503.
```

```
504.     # When shutting down the scheduler, there will typically be a
     thread which
505.     # needs to be stopped, and this timeout determines how long to
     wait for
506.     # that to happen. In case of timeout the shutdown of the actor
     system will
507.     # proceed without running possibly still enqueued tasks.
508.     shutdown-timeout = 5s
509.   }
510.
511.   io {
512.
513.     # By default the select loops run on dedicated threads, hence
     using a
514.     # PinnedDispatcher
515.     pinned-dispatcher {
516.       type = "PinnedDispatcher"
517.       executor = "thread-pool-executor"
518.       thread-pool-executor.allow-core-pool-timeout = off
519.     }
520.
521.     tcp {
522.
523.       # The number of selectors to stripe the served channels over;
     each of
524.       # these will use one select loop on the selector-dispatcher.
525.       nr-of-selectors = 1
526.
527.       # Maximum number of open channels supported by this TCP
     module; there is
528.       # no intrinsic general limit, this setting is meant to enable
     DoS
529.       # protection by limiting the number of concurrently connected
     clients.
530.       # Also note that this is a "soft" limit; in certain cases the
     implementation
531.       # will accept a few connections more or a few less than the
     number configured
```

```
532.        # here. Must be an integer > 0 or "unlimited".
533.        max-channels = 256000
534.
535.        # When trying to assign a new connection to a selector and
    the chosen
536.        # selector is at full capacity, retry selector choosing and
    assignment
537.        # this many times before giving up
538.        selector-association-retries = 10
539.
540.        # The maximum number of connection that are accepted in one
    go,
541.        # higher numbers decrease latency, lower numbers increase
    fairness on
542.        # the worker-dispatcher
543.        batch-accept-limit = 10
544.
545.        # The number of bytes per direct buffer in the pool used to
    read or write
546.        # network data from the kernel.
547.        direct-buffer-size = 128 KiB
548.
549.        # The maximal number of direct buffers kept in the direct
    buffer pool for
550.        # reuse.
551.        direct-buffer-pool-limit = 1000
552.
553.        # The duration a connection actor waits for a `Register`
    message from
554.        # its commander before aborting the connection.
555.        register-timeout = 5s
556.
557.        # The maximum number of bytes delivered by a `Received`
    message. Before
558.        # more data is read from the network the connection actor
    will try to
559.        # do other work.
560.        max-received-message-size = unlimited
```

```
561.
562.       # Enable fine grained logging of what goes on inside the
     implementation.
563.       # Be aware that this may log more than once per message sent
     to the actors
564.       # of the tcp implementation.
565.       trace-logging = off
566.
567.       # Fully qualified config path which holds the dispatcher
     configuration
568.       # to be used for running the select() calls in the selectors
569.       selector-dispatcher = "akka.io.pinned-dispatcher"
570.
571.       # Fully qualified config path which holds the dispatcher
     configuration
572.       # for the read/write worker actors
573.       worker-dispatcher = "akka.actor.default-dispatcher"
574.
575.       # Fully qualified config path which holds the dispatcher
     configuration
576.       # for the selector management actors
577.       management-dispatcher = "akka.actor.default-dispatcher"
578.
579.       # Fully qualified config path which holds the dispatcher
     configuration
580.       # on which file IO tasks are scheduled
581.       file-io-dispatcher = "akka.actor.default-dispatcher"
582.
583.       # The maximum number of bytes (or "unlimited") to transfer in
     one batch
584.       # when using `WriteFile` command which uses
     `FileChannel.transferTo` to
585.       # pipe files to a TCP socket. On some OS like Linux
     `FileChannel.transferTo`
586.       # may block for a long time when network IO is faster than
     file IO.
587.       # Decreasing the value may improve fairness while increasing
     may improve
```

```
588.        # throughput.
589.        file-io-transferTo-limit = 512 KiB
590.
591.        # The number of times to retry the `finishConnect` call after
    being notified about
592.        # OP_CONNECT. Retries are needed if the OP_CONNECT
    notification doesn't imply that
593.        # `finishConnect` will succeed, which is the case on Android.
594.        finish-connect-retries = 5
595.      }
596.
597.      udp {
598.
599.        # The number of selectors to stripe the served channels over;
    each of
600.        # these will use one select loop on the selector-dispatcher.
601.        nr-of-selectors = 1
602.
603.        # Maximum number of open channels supported by this UDP
    module Generally
604.        # UDP does not require a large number of channels, therefore
    it is
605.        # recommended to keep this setting low.
606.        max-channels = 4096
607.
608.        # The select loop can be used in two modes:
609.        # - setting "infinite" will select without a timeout, hogging
    a thread
610.        # - setting a positive timeout will do a bounded select call,
611.        #   enabling sharing of a single thread between multiple
    selectors
612.        #   (in this case you will have to use a different
    configuration for the
613.        #   selector-dispatcher, e.g. using "type=Dispatcher" with
    size 1)
614.        # - setting it to zero means polling, i.e. calling
    selectNow()
615.        select-timeout = infinite
```

```
616.
617.       # When trying to assign a new connection to a selector and
     the chosen
618.       # selector is at full capacity, retry selector choosing and
     assignment
619.       # this many times before giving up
620.       selector-association-retries = 10
621.
622.       # The maximum number of datagrams that are read in one go,
623.       # higher numbers decrease latency, lower numbers increase
     fairness on
624.       # the worker-dispatcher
625.       receive-throughput = 3
626.
627.       # The number of bytes per direct buffer in the pool used to
     read or write
628.       # network data from the kernel.
629.       direct-buffer-size = 128 KiB
630.
631.       # The maximal number of direct buffers kept in the direct
     buffer pool for
632.       # reuse.
633.       direct-buffer-pool-limit = 1000
634.
635.       # The maximum number of bytes delivered by a `Received`
     message. Before
636.       # more data is read from the network the connection actor
     will try to
637.       # do other work.
638.       received-message-size-limit = unlimited
639.
640.       # Enable fine grained logging of what goes on inside the
     implementation.
641.       # Be aware that this may log more than once per message sent
     to the actors
642.       # of the tcp implementation.
643.       trace-logging = off
644.
```

```
645.        # Fully qualified config path which holds the dispatcher
    configuration
646.        # to be used for running the select() calls in the selectors
647.        selector-dispatcher = "akka.io.pinned-dispatcher"
648.
649.        # Fully qualified config path which holds the dispatcher
    configuration
650.        # for the read/write worker actors
651.        worker-dispatcher = "akka.actor.default-dispatcher"
652.
653.        # Fully qualified config path which holds the dispatcher
    configuration
654.        # for the selector management actors
655.        management-dispatcher = "akka.actor.default-dispatcher"
656.      }
657.
658.      udp-connected {
659.
660.        # The number of selectors to stripe the served channels over;
    each of
661.        # these will use one select loop on the selector-dispatcher.
662.        nr-of-selectors = 1
663.
664.        # Maximum number of open channels supported by this UDP
    module Generally
665.        # UDP does not require a large number of channels, therefore
    it is
666.        # recommended to keep this setting low.
667.        max-channels = 4096
668.
669.        # The select loop can be used in two modes:
670.        # - setting "infinite" will select without a timeout, hogging
    a thread
671.        # - setting a positive timeout will do a bounded select call,
672.        #   enabling sharing of a single thread between multiple
    selectors
673.        #   (in this case you will have to use a different
    configuration for the
```

```
674.        #   selector-dispatcher, e.g. using "type=Dispatcher" with
    size 1)
675.        # - setting it to zero means polling, i.e. calling
    selectNow()
676.        select-timeout = infinite
677.
678.        # When trying to assign a new connection to a selector and
    the chosen
679.        # selector is at full capacity, retry selector choosing and
    assignment
680.        # this many times before giving up
681.        selector-association-retries = 10
682.
683.        # The maximum number of datagrams that are read in one go,
684.        # higher numbers decrease latency, lower numbers increase
    fairness on
685.        # the worker-dispatcher
686.        receive-throughput = 3
687.
688.        # The number of bytes per direct buffer in the pool used to
    read or write
689.        # network data from the kernel.
690.        direct-buffer-size = 128 KiB
691.
692.        # The maximal number of direct buffers kept in the direct
    buffer pool for
693.        # reuse.
694.        direct-buffer-pool-limit = 1000
695.
696.        # The maximum number of bytes delivered by a `Received`
    message. Before
697.        # more data is read from the network the connection actor
    will try to
698.        # do other work.
699.        received-message-size-limit = unlimited
700.
701.        # Enable fine grained logging of what goes on inside the
    implementation.
```

```
702.       # Be aware that this may log more than once per message sent
     to the actors
703.       # of the tcp implementation.
704.       trace-logging = off
705.
706.       # Fully qualified config path which holds the dispatcher
     configuration
707.       # to be used for running the select() calls in the selectors
708.       selector-dispatcher = "akka.io.pinned-dispatcher"
709.
710.       # Fully qualified config path which holds the dispatcher
     configuration
711.       # for the read/write worker actors
712.       worker-dispatcher = "akka.actor.default-dispatcher"
713.
714.       # Fully qualified config path which holds the dispatcher
     configuration
715.       # for the selector management actors
716.       management-dispatcher = "akka.actor.default-dispatcher"
717.     }
718.
719.   }
720.
721.
722. }
```

akka-agent

```
1.  ####################################
2.  # Akka Agent Reference Config File #
3.  ####################################
4.
5.  # This is the reference config file that contains all the default
    settings.
6.  # Make your edits/overrides in your application.conf.
7.
8.  akka {
9.    agent {
```

```
10.
11.     # The dispatcher used for agent-send-off actor
12.     send-off-dispatcher {
13.       executor = thread-pool-executor
14.       type = PinnedDispatcher
15.     }
16.
17.     # The dispatcher used for agent-alter-off actor
18.     alter-off-dispatcher {
19.       executor = thread-pool-executor
20.       type = PinnedDispatcher
21.     }
22.   }
23. }
```

akka-camel

```
1.  ####################################
2.  # Akka Camel Reference Config File #
3.  ####################################
4.
5.  # This is the reference config file that contains all the default
    settings.
6.  # Make your edits/overrides in your application.conf.
7.
8.  akka {
9.    camel {
10.     # FQCN of the ContextProvider to be used to create or locate a
    CamelContext
11.     # it must implement akka.camel.ContextProvider and have a no-
    arg constructor
12.     # the built-in default create a fresh DefaultCamelContext
13.     context-provider = akka.camel.DefaultContextProvider
14.
15.     # Whether JMX should be enabled or disabled for the Camel
    Context
16.     jmx = off
17.     # enable/disable streaming cache on the Camel Context
```

```
    streamingCache = on
    consumer {
      # Configured setting which determines whether one-way communications
      # between an endpoint and this consumer actor
      # should be auto-acknowledged or application-acknowledged.
      # This flag has only effect when exchange is in-only.
      auto-ack = on

      # When endpoint is out-capable (can produce responses) reply-timeout is the
      # maximum time the endpoint can take to send the response before the message
      # exchange fails. This setting is used for out-capable, in-only,
      # manually acknowledged communication.
      reply-timeout = 1m

      # The duration of time to await activation of an endpoint.
      activation-timeout = 10s
    }

    #Scheme to FQCN mappings for CamelMessage body conversions
    conversions {
      "file" = "java.io.InputStream"
    }
  }
}
```

akka-cluster

```
##########################################
# Akka Cluster Reference Config File #
##########################################

# This is the reference config file that contains all the default settings.
# Make your edits/overrides in your application.conf.
```

```
7.
8.  akka {
9.
10.   cluster {
11.     # Initial contact points of the cluster.
12.     # The nodes to join automatically at startup.
13.     # Comma separated full URIs defined by a string on the form of
14.     # "akka.tcp://system@hostname:port"
15.     # Leave as empty if the node is supposed to be joined manually.
16.     seed-nodes = []
17.
18.     # how long to wait for one of the seed nodes to reply to
    initial join request
19.     seed-node-timeout = 5s
20.
21.     # If a join request fails it will be retried after this period.
22.     # Disable join retry by specifying "off".
23.     retry-unsuccessful-join-after = 10s
24.
25.     # Should the 'leader' in the cluster be allowed to
    automatically mark
26.     # unreachable nodes as DOWN after a configured time of
    unreachability?
27.     # Using auto-down implies that two separate clusters will
    automatically be
28.     # formed in case of network partition.
29.     # Disable with "off" or specify a duration to enable auto-down.
30.     auto-down-unreachable-after = off
31.
32.     # deprecated in 2.3, use 'auto-down-unreachable-after' instead
33.     auto-down = off
34.
35.     # The roles of this member. List of strings, e.g. roles = ["A",
    "B"].
36.     # The roles are part of the membership information and can be
    used by
37.     # routers or other services to distribute work to certain
    member types,
```

```
38.     # e.g. front-end and back-end nodes.
39.     roles = []
40.
41.     role {
42.       # Minimum required number of members of a certain role before
    the leader
43.       # changes member status of 'Joining' members to 'Up'.
    Typically used together
44.       # with 'Cluster.registerOnMemberUp' to defer some action,
    such as starting
45.       # actors, until the cluster has reached a certain size.
46.       # E.g. to require 2 nodes with role 'frontend' and 3 nodes
    with role 'backend':
47.       #   frontend.min-nr-of-members = 2
48.       #   backend.min-nr-of-members = 3
49.       #<role-name>.min-nr-of-members = 1
50.     }
51.
52.     # Minimum required number of members before the leader changes
    member status
53.     # of 'Joining' members to 'Up'. Typically used together with
54.     # 'Cluster.registerOnMemberUp' to defer some action, such as
    starting actors,
55.     # until the cluster has reached a certain size.
56.     min-nr-of-members = 1
57.
58.     # Enable/disable info level logging of cluster events
59.     log-info = on
60.
61.     # Enable or disable JMX MBeans for management of the cluster
62.     jmx.enabled = on
63.
64.     # how long should the node wait before starting the periodic
    tasks
65.     # maintenance tasks?
66.     periodic-tasks-initial-delay = 1s
67.
68.     # how often should the node send out gossip information?
```

```
69.     gossip-interval = 1s
70.
71.     # discard incoming gossip messages if not handled within this
    duration
72.     gossip-time-to-live = 2s
73.
74.     # how often should the leader perform maintenance tasks?
75.     leader-actions-interval = 1s
76.
77.     # how often should the node move nodes, marked as unreachable
    by the failure
78.     # detector, out of the membership ring?
79.     unreachable-nodes-reaper-interval = 1s
80.
81.     # How often the current internal stats should be published.
82.     # A value of 0s can be used to always publish the stats, when
    it happens.
83.     # Disable with "off".
84.     publish-stats-interval = off
85.
86.     # The id of the dispatcher to use for cluster actors. If not
    specified
87.     # default dispatcher is used.
88.     # If specified you need to define the settings of the actual
    dispatcher.
89.     use-dispatcher = ""
90.
91.     # Gossip to random node with newer or older state information,
    if any with
92.     # this probability. Otherwise Gossip to any random live node.
93.     # Probability value is between 0.0 and 1.0. 0.0 means never,
    1.0 means always.
94.     gossip-different-view-probability = 0.8
95.
96.     # Reduced the above probability when the number of nodes in the
    cluster
97.     # greater than this value.
98.     reduce-gossip-different-view-probability = 400
```

```
99.
100.     # Settings for the Phi accrual failure detector
(http://ddg.jaist.ac.jp/pub/HDY+04.pdf
101.     # [Hayashibara et al]) used by the cluster subsystem to detect
unreachable
102.     # members.
103.     failure-detector {
104.
105.       # FQCN of the failure detector implementation.
106.       # It must implement akka.remote.FailureDetector and have
107.       # a public constructor with a com.typesafe.config.Config and
108.       # akka.actor.EventStream parameter.
109.       implementation-class =
"akka.remote.PhiAccrualFailureDetector"
110.
111.       # How often keep-alive heartbeat messages should be sent to
each connection.
112.       heartbeat-interval = 1 s
113.
114.       # Defines the failure detector threshold.
115.       # A low threshold is prone to generate many wrong suspicions
but ensures
116.       # a quick detection in the event of a real crash. Conversely,
a high
117.       # threshold generates fewer mistakes but needs more time to
detect
118.       # actual crashes.
119.       threshold = 8.0
120.
121.       # Number of the samples of inter-heartbeat arrival times to
adaptively
122.       # calculate the failure timeout for connections.
123.       max-sample-size = 1000
124.
125.       # Minimum standard deviation to use for the normal
distribution in
126.       # AccrualFailureDetector. Too low standard deviation might
result in
```

```
127.       # too much sensitivity for sudden, but normal, deviations in
     heartbeat
128.       # inter arrival times.
129.       min-std-deviation = 100 ms
130.
131.       # Number of potentially lost/delayed heartbeats that will be
132.       # accepted before considering it to be an anomaly.
133.       # This margin is important to be able to survive sudden,
     occasional,
134.       # pauses in heartbeat arrivals, due to for example garbage
     collect or
135.       # network drop.
136.       acceptable-heartbeat-pause = 3 s
137.
138.       # Number of member nodes that each member will send heartbeat
     messages to,
139.       # i.e. each node will be monitored by this number of other
     nodes.
140.       monitored-by-nr-of-members = 5
141.
142.       # After the heartbeat request has been sent the first failure
     detection
143.       # will start after this period, even though no heartbeat
     mesage has
144.       # been received.
145.       expected-response-after = 5 s
146.
147.     }
148.
149.     metrics {
150.       # Enable or disable metrics collector for load-balancing
     nodes.
151.       enabled = on
152.
153.       # FQCN of the metrics collector implementation.
154.       # It must implement akka.cluster.MetricsCollector and
155.       # have public constructor with akka.actor.ActorSystem
     parameter.
```

```
156.       # The default SigarMetricsCollector uses JMX and Hyperic
    SIGAR, if SIGAR
157.       # is on the classpath, otherwise only JMX.
158.       collector-class = "akka.cluster.SigarMetricsCollector"
159.
160.       # How often metrics are sampled on a node.
161.       # Shorter interval will collect the metrics more often.
162.       collect-interval = 3s
163.
164.       # How often a node publishes metrics information.
165.       gossip-interval = 3s
166.
167.       # How quickly the exponential weighting of past data is
    decayed compared to
168.       # new data. Set lower to increase the bias toward newer
    values.
169.       # The relevance of each data sample is halved for every
    passing half-life
170.       # duration, i.e. after 4 times the half-life, a data sample's
    relevance is
171.       # reduced to 6% of its original relevance. The initial
    relevance of a data
172.       # sample is given by 1 - 0.5 ^ (collect-interval / half-
    life).
173.       # See
    http://en.wikipedia.org/wiki/Moving_average#Exponential_moving_averag
174.       moving-average-half-life = 12s
175.     }
176.
177.     # If the tick-duration of the default scheduler is longer than
    the
178.     # tick-duration configured here a dedicated scheduler will be
    used for
179.     # periodic tasks of the cluster, otherwise the default
    scheduler is used.
180.     # See akka.scheduler settings for more details.
181.     scheduler {
182.       tick-duration = 33ms
```

```
183.         ticks-per-wheel = 512
184.       }
185.
186.     }
187.
188.     # Default configuration for routers
189.     actor.deployment.default {
190.       # MetricsSelector to use
191.       # - available: "mix", "heap", "cpu", "load"
192.       # - or: Fully qualified class name of the MetricsSelector
    class.
193.       #       The class must extend
    akka.cluster.routing.MetricsSelector
194.       #       and have a public constructor with
    com.typesafe.config.Config
195.       #       parameter.
196.       # - default is "mix"
197.       metrics-selector = mix
198.     }
199.     actor.deployment.default.cluster {
200.       # enable cluster aware router that deploys to nodes in the
    cluster
201.       enabled = off
202.
203.       # Maximum number of routees that will be deployed on each
    cluster
204.       # member node.
205.       # Note that nr-of-instances defines total number of routees,
    but
206.       # number of routees per node will not be exceeded, i.e. if you
207.       # define nr-of-instances = 50 and max-nr-of-instances-per-node
    = 2
208.       # it will deploy 2 routees per new member in the cluster, up to
209.       # 25 members.
210.       max-nr-of-instances-per-node = 1
211.
212.       # Defines if routees are allowed to be located on the same node
    as
```

```
213.     # the head router actor, or only on remote nodes.
214.     # Useful for master-worker scenario where all routees are
     remote.
215.     allow-local-routees = on
216.
217.     # Deprecated in 2.3, use routees.paths instead
218.     routees-path = ""
219.
220.     # Use members with specified role, or all members if undefined
     or empty.
221.     use-role = ""
222.
223.   }
224.
225.   # Protobuf serializer for cluster messages
226.   actor {
227.     serializers {
228.       akka-cluster =
     "akka.cluster.protobuf.ClusterMessageSerializer"
229.     }
230.
231.     serialization-bindings {
232.       "akka.cluster.ClusterMessage" = akka-cluster
233.     }
234.
235.     router.type-mapping {
236.       adaptive-pool =
     "akka.cluster.routing.AdaptiveLoadBalancingPool"
237.       adaptive-group =
     "akka.cluster.routing.AdaptiveLoadBalancingGroup"
238.     }
239.   }
240.
241. }
```

akka-multi-node-testkit

```
1.  ##############################################
2.  # Akka Remote Testing Reference Config File #
3.  ##############################################
4.
5.  # This is the reference config file that contains all the default
    settings.
6.  # Make your edits/overrides in your application.conf.
7.
8.  akka {
9.    testconductor {
10.
11.     # Timeout for joining a barrier: this is the maximum time any
    participants
12.     # waits for everybody else to join a named barrier.
13.     barrier-timeout = 30s
14.
15.     # Timeout for interrogation of TestConductor’s Controller actor
16.     query-timeout = 5s
17.
18.     # Threshold for packet size in time unit above which the
    failure injector will
19.     # split the packet and deliver in smaller portions; do not give
    value smaller
20.     # than HashedWheelTimer resolution (would not make sense)
21.     packet-split-threshold = 100ms
22.
23.     # amount of time for the ClientFSM to wait for the connection
    to the conductor
24.     # to be successful
25.     connect-timeout = 20s
26.
27.     # Number of connect attempts to be made to the conductor
    controller
28.     client-reconnects = 10
29.
30.     # minimum time interval which is to be inserted between
    reconnect attempts
31.     reconnect-backoff = 1s
```

```
32.
33.     netty {
34.       # (I&O) Used to configure the number of I/O worker threads on
    server sockets
35.       server-socket-worker-pool {
36.         # Min number of threads to cap factor-based number to
37.         pool-size-min = 1
38.
39.         # The pool size factor is used to determine thread pool
    size
40.         # using the following formula: ceil(available processors *
    factor).
41.         # Resulting size is then bounded by the pool-size-min and
42.         # pool-size-max values.
43.         pool-size-factor = 1.0
44.
45.         # Max number of threads to cap factor-based number to
46.         pool-size-max = 2
47.       }
48.
49.       # (I&O) Used to configure the number of I/O worker threads on
    client sockets
50.       client-socket-worker-pool {
51.         # Min number of threads to cap factor-based number to
52.         pool-size-min = 1
53.
54.         # The pool size factor is used to determine thread pool
    size
55.         # using the following formula: ceil(available processors *
    factor).
56.         # Resulting size is then bounded by the pool-size-min and
57.         # pool-size-max values.
58.         pool-size-factor = 1.0
59.
60.         # Max number of threads to cap factor-based number to
61.         pool-size-max = 2
62.       }
63.     }
```

```
  }
}
```

akka-persistence

```
##############################################
# Akka Persistence Reference Config File #
##############################################



akka {

  # Protobuf serialization for persistent messages
  actor {

    serializers {

      akka-persistence-snapshot = "akka.persistence.serialization.SnapshotSerializer"
      akka-persistence-message = "akka.persistence.serialization.MessageSerializer"
    }

    serialization-bindings {

      "akka.persistence.serialization.Snapshot" = akka-persistence-snapshot
      "akka.persistence.serialization.Message" = akka-persistence-message
    }
  }

  persistence {

    journal {

      # Maximum size of a persistent message batch written to the
```

```
    journal.
30.       max-message-batch-size = 200
31.
32.       # Maximum size of a confirmation batch written to the
    journal.
33.       max-confirmation-batch-size = 10000
34.
35.       # Maximum size of a deletion batch written to the journal.
36.       max-deletion-batch-size = 10000
37.
38.       # Path to the journal plugin to be used
39.       plugin = "akka.persistence.journal.leveldb"
40.
41.       # In-memory journal plugin.
42.       inmem {
43.
44.         # Class name of the plugin.
45.         class = "akka.persistence.journal.inmem.InmemJournal"
46.
47.         # Dispatcher for the plugin actor.
48.         plugin-dispatcher = "akka.actor.default-dispatcher"
49.       }
50.
51.       # LevelDB journal plugin.
52.       leveldb {
53.
54.         # Class name of the plugin.
55.         class = "akka.persistence.journal.leveldb.LeveldbJournal"
56.
57.         # Dispatcher for the plugin actor.
58.         plugin-dispatcher = "akka.persistence.dispatchers.default-
    plugin-dispatcher"
59.
60.         # Dispatcher for message replay.
61.         replay-dispatcher = "akka.persistence.dispatchers.default-
    replay-dispatcher"
62.
63.         # Storage location of LevelDB files.
```

```
64.         dir = "journal"
65.
66.         # Use fsync on write
67.         fsync = on
68.
69.         # Verify checksum on read.
70.         checksum = off
71.
72.         # Native LevelDB (via JNI) or LevelDB Java port
73.         native = on
74.       }
75.
76.       # Shared LevelDB journal plugin (for testing only).
77.       leveldb-shared {
78.
79.         # Class name of the plugin.
80.         class =
    "akka.persistence.journal.leveldb.SharedLeveldbJournal"
81.
82.         # Dispatcher for the plugin actor.
83.         plugin-dispatcher = "akka.actor.default-dispatcher"
84.
85.         # timeout for async journal operations
86.         timeout = 10s
87.
88.         store {
89.
90.           # Dispatcher for shared store actor.
91.           store-dispatcher = "akka.persistence.dispatchers.default-
    plugin-dispatcher"
92.
93.           # Dispatcher for message replay.
94.           replay-dispatcher =
    "akka.persistence.dispatchers.default-plugin-dispatcher"
95.
96.           # Storage location of LevelDB files.
97.           dir = "journal"
98.
```

```
99.           # Use fsync on write
100.          fsync = on
101.
102.          # Verify checksum on read.
103.          checksum = off
104.
105.          # Native LevelDB (via JNI) or LevelDB Java port
106.          native = on
107.        }
108.      }
109.    }
110.
111.    snapshot-store {
112.
113.      # Path to the snapshot store plugin to be used
114.      plugin = "akka.persistence.snapshot-store.local"
115.
116.      # Local filesystem snapshot store plugin.
117.      local {
118.
119.        # Class name of the plugin.
120.        class =
    "akka.persistence.snapshot.local.LocalSnapshotStore"
121.
122.        # Dispatcher for the plugin actor.
123.        plugin-dispatcher = "akka.persistence.dispatchers.default-
    plugin-dispatcher"
124.
125.        # Dispatcher for streaming snapshot IO.
126.        stream-dispatcher = "akka.persistence.dispatchers.default-
    stream-dispatcher"
127.
128.        # Storage location of snapshot files.
129.        dir = "snapshots"
130.      }
131.    }
132.
133.    view {
```

```
134.
135.       # Automated incremental view update.
136.       auto-update = on
137.
138.       # Interval between incremental updates
139.       auto-update-interval = 5s
140.
141.       # Maximum number of messages to replay per incremental view
    update. Set to
142.       # -1 for no upper limit.
143.       auto-update-replay-max = -1
144.     }
145.
146.     at-least-once-delivery {
147.       # Interval between redelivery attempts
148.       redeliver-interval = 5s
149.
150.       # After this number of delivery attempts a
    `ReliableRedelivery.UnconfirmedWarning`
151.       # message will be sent to the actor.
152.       warn-after-number-of-unconfirmed-attempts = 5
153.
154.       # Maximum number of unconfirmed messages that an actor with
    AtLeastOnceDelivery is
155.       # allowed to hold in memory.
156.       max-unconfirmed-messages = 100000
157.     }
158.
159.     dispatchers {
160.       default-plugin-dispatcher {
161.         type = PinnedDispatcher
162.         executor = "thread-pool-executor"
163.       }
164.       default-replay-dispatcher {
165.         type = Dispatcher
166.         executor = "fork-join-executor"
167.         fork-join-executor {
168.           parallelism-min = 2
```

```
169.           parallelism-max = 8
170.         }
171.       }
172.       default-stream-dispatcher {
173.         type = Dispatcher
174.         executor = "fork-join-executor"
175.         fork-join-executor {
176.           parallelism-min = 2
177.           parallelism-max = 8
178.         }
179.       }
180.     }
181.   }
182. }
```

akka-remote

```
1.  ######################################
2.  # Akka Remote Reference Config File #
3.  ######################################
4.
5.  # This is the reference config file that contains all the default
    settings.
6.  # Make your edits/overrides in your application.conf.
7.
8.  # comments about akka.actor settings left out where they are
    already in akka-
9.  # actor.jar, because otherwise they would be repeated in config
    rendering.
10.
11. akka {
12.
13.   actor {
14.
15.     serializers {
16.       akka-containers =
    "akka.remote.serialization.MessageContainerSerializer"
17.       proto = "akka.remote.serialization.ProtobufSerializer"
```

```
18.       daemon-create =
    "akka.remote.serialization.DaemonMsgCreateSerializer"
19.     }
20.
21.
22.     serialization-bindings {
23.       # Since com.google.protobuf.Message does not extend
    Serializable but
24.       # GeneratedMessage does, need to use the more specific one
    here in order
25.       # to avoid ambiguity
26.       "akka.actor.ActorSelectionMessage" = akka-containers
27.       "com.google.protobuf.GeneratedMessage" = proto
28.       "akka.remote.DaemonMsgCreate" = daemon-create
29.     }
30.
31.     deployment {
32.
33.       default {
34.
35.         # if this is set to a valid remote address, the named actor
    will be
36.         # deployed at that node e.g. "akka.tcp://sys@host:port"
37.         remote = ""
38.
39.         target {
40.
41.           # A list of hostnames and ports for instantiating the
    children of a
42.           # router
43.           #   The format should be on "akka.tcp://sys@host:port",
    where:
44.           #    - sys is the remote actor system name
45.           #    - hostname can be either hostname or IP address the
    remote actor
46.           #      should connect to
47.           #    - port should be the port for the remote server on
    the other node
```

```
48.           # The number of actor instances to be spawned is still
    taken from the
49.           # nr-of-instances setting as for local routers; the
    instances will be
50.           # distributed round-robin among the given nodes.
51.           nodes = []
52.
53.         }
54.       }
55.     }
56.   }
57.
58.   remote {
59.
60.     ### General settings
61.
62.     # Timeout after which the startup of the remoting subsystem is
    considered
63.     # to be failed. Increase this value if your transport drivers
    (see the
64.     # enabled-transports section) need longer time to be loaded.
65.     startup-timeout = 10 s
66.
67.     # Timout after which the graceful shutdown of the remoting
    subsystem is
68.     # considered to be failed. After the timeout the remoting
    system is
69.     # forcefully shut down. Increase this value if your transport
    drivers
70.     # (see the enabled-transports section) need longer time to stop
    properly.
71.     shutdown-timeout = 10 s
72.
73.     # Before shutting down the drivers, the remoting subsystem
    attempts to flush
74.     # all pending writes. This setting controls the maximum time
    the remoting is
75.     # willing to wait before moving on to shut down the drivers.
```

```
76.     flush-wait-on-shutdown = 2 s
77.
78.     # Reuse inbound connections for outbound messages
79.     use-passive-connections = on
80.
81.     # Controls the backoff interval after a refused write is
    reattempted.
82.     # (Transports may refuse writes if their internal buffer is
    full)
83.     backoff-interval = 5 ms
84.
85.     # Acknowledgment timeout of management commands sent to the
    transport stack.
86.     command-ack-timeout = 30 s
87.
88.     # If set to a nonempty string remoting will use the given
    dispatcher for
89.     # its internal actors otherwise the default dispatcher is used.
    Please note
90.     # that since remoting can load arbitrary 3rd party drivers (see
91.     # "enabled-transport" and "adapters" entries) it is not
    guaranteed that
92.     # every module will respect this setting.
93.     use-dispatcher = "akka.remote.default-remote-dispatcher"
94.
95.     ### Security settings
96.
97.     # Enable untrusted mode for full security of server managed
    actors, prevents
98.     # system messages to be send by clients, e.g. messages like
    'Create',
99.     # 'Suspend', 'Resume', 'Terminate', 'Supervise', 'Link' etc.
100.    untrusted-mode = off
101.
102.    # When 'untrusted-mode=on' inbound actor selections are by
    default discarded.
103.    # Actors with paths defined in this white list are granted
    permission to receive actor
```

```
104.     # selections messages.
105.     # E.g. trusted-selection-paths = ["/user/receptionist",
     "/user/namingService"]
106.     trusted-selection-paths = []
107.
108.     # Should the remote server require that its peers share the
     same
109.     # secure-cookie (defined in the 'remote' section)? Secure
     cookies are passed
110.     # between during the initial handshake. Connections are refused
     if the initial
111.     # message contains a mismatching cookie or the cookie is
     missing.
112.     require-cookie = off
113.
114.     # Generate your own with the script availbale in
115.     # '$AKKA_HOME/scripts/generate_config_with_secure_cookie.sh' or
     using
116.     # 'akka.util.Crypt.generateSecureCookie'
117.     secure-cookie = ""
118.
119.     ### Logging
120.
121.     # If this is "on", Akka will log all inbound messages at DEBUG
     level,
122.     # if off then they are not logged
123.     log-received-messages = off
124.
125.     # If this is "on", Akka will log all outbound messages at DEBUG
     level,
126.     # if off then they are not logged
127.     log-sent-messages = off
128.
129.     # Sets the log granularity level at which Akka logs remoting
     events. This setting
130.     # can take the values OFF, ERROR, WARNING, INFO, DEBUG, or ON.
     For compatibility
131.     # reasons the setting "on" will default to "debug" level.
```

```
    Please note that the effective
132.     # logging level is still determined by the global logging level
    of the actor system:
133.     # for example debug level remoting events will be only logged
    if the system
134.     # is running with debug level logging.
135.     # Failures to deserialize received messages also fall under
    this flag.
136.     log-remote-lifecycle-events = on
137.
138.     # Logging of message types with payload size in bytes larger
    than
139.     # this value. Maximum detected size per message type is logged
    once,
140.     # with an increase threshold of 10%.
141.     # By default this feature is turned off. Activate it by setting
    the property to
142.     # a value in bytes, such as 1000b. Note that for all messages
    larger than this
143.     # limit there will be extra performance and scalability cost.
144.     log-frame-size-exceeding = off
145.
146.     # Log warning if the number of messages in the backoff buffer
    in the endpoint
147.     # writer exceeds this limit. It can be disabled by setting the
    value to off.
148.     log-buffer-size-exceeding = 50000
149.
150.     ### Failure detection and recovery
151.
152.     # Settings for the failure detector to monitor connections.
153.     # For TCP it is not important to have fast failure detection,
    since
154.     # most connection failures are captured by TCP itself.
155.     transport-failure-detector {
156.
157.       # FQCN of the failure detector implementation.
158.       # It must implement akka.remote.FailureDetector and have
```

```
159.       # a public constructor with a com.typesafe.config.Config and
160.       # akka.actor.EventStream parameter.
161.       implementation-class = "akka.remote.DeadlineFailureDetector"
162.
163.       # How often keep-alive heartbeat messages should be sent to
    each connection.
164.       heartbeat-interval = 4 s
165.
166.       # Number of potentially lost/delayed heartbeats that will be
167.       # accepted before considering it to be an anomaly.
168.       # A margin to the `heartbeat-interval` is important to be
    able to survive sudden,
169.       # occasional, pauses in heartbeat arrivals, due to for
    example garbage collect or
170.       # network drop.
171.       acceptable-heartbeat-pause = 20 s
172.     }
173.
174.     # Settings for the Phi accrual failure detector
    (http://ddg.jaist.ac.jp/pub/HDY+04.pdf
175.     # [Hayashibara et al]) used for remote death watch.
176.     watch-failure-detector {
177.
178.       # FQCN of the failure detector implementation.
179.       # It must implement akka.remote.FailureDetector and have
180.       # a public constructor with a com.typesafe.config.Config and
181.       # akka.actor.EventStream parameter.
182.       implementation-class =
    "akka.remote.PhiAccrualFailureDetector"
183.
184.       # How often keep-alive heartbeat messages should be sent to
    each connection.
185.       heartbeat-interval = 1 s
186.
187.       # Defines the failure detector threshold.
188.       # A low threshold is prone to generate many wrong suspicions
    but ensures
189.       # a quick detection in the event of a real crash. Conversely,
```

```
a high
190.        # threshold generates fewer mistakes but needs more time to
detect
191.        # actual crashes.
192.        threshold = 10.0
193.
194.        # Number of the samples of inter-heartbeat arrival times to
adaptively
195.        # calculate the failure timeout for connections.
196.        max-sample-size = 200
197.
198.        # Minimum standard deviation to use for the normal
distribution in
199.        # AccrualFailureDetector. Too low standard deviation might
result in
200.        # too much sensitivity for sudden, but normal, deviations in
heartbeat
201.        # inter arrival times.
202.        min-std-deviation = 100 ms
203.
204.        # Number of potentially lost/delayed heartbeats that will be
205.        # accepted before considering it to be an anomaly.
206.        # This margin is important to be able to survive sudden,
occasional,
207.        # pauses in heartbeat arrivals, due to for example garbage
collect or
208.        # network drop.
209.        acceptable-heartbeat-pause = 10 s
210.
211.
212.        # How often to check for nodes marked as unreachable by the
failure
213.        # detector
214.        unreachable-nodes-reaper-interval = 1s
215.
216.        # After the heartbeat request has been sent the first failure
detection
217.        # will start after this period, even though no heartbeat
```

```
mesage has
      # been received.
      expected-response-after = 3 s

    }

    # After failed to establish an outbound connection, the
remoting will mark the
    # address as failed. This configuration option controls how
much time should
    # be elapsed before reattempting a new connection. While the
address is
    # gated, all messages sent to the address are delivered to
dead-letters.
    # Since this setting limits the rate of reconnects setting it
to a
    # very short interval (i.e. less than a second) may result in a
storm of
    # reconnect attempts.
    retry-gate-closed-for = 5 s

    # After catastrophic communication failures that result in the
loss of system
    # messages or after the remote DeathWatch triggers the remote
system gets
    # quarantined to prevent inconsistent behavior.
    # This setting controls how long the Quarantine marker will be
kept around
    # before being removed to avoid long-term memory leaks.
    # WARNING: DO NOT change this to a small value to re-enable
communication with
    # quarantined nodes. Such feature is not supported and any
behavior between
    # the affected systems after lifting the quarantine is
undefined.
    prune-quarantine-marker-after = 5 d

    # This setting defines the maximum number of unacknowledged
```

```
system messages
243.     # allowed for a remote system. If this limit is reached the
remote system is
244.     # declared to be dead and its UID marked as tainted.
245.     system-message-buffer-size = 1000
246.
247.     # This setting defines the maximum idle time after an
individual
248.     # acknowledgement for system messages is sent. System message
delivery
249.     # is guaranteed by explicit acknowledgement messages. These
acks are
250.     # piggybacked on ordinary traffic messages. If no traffic is
detected
251.     # during the time period configured here, the remoting will
send out
252.     # an individual ack.
253.     system-message-ack-piggyback-timeout = 0.3 s
254.
255.     # This setting defines the time after internal management
signals
256.     # between actors (used for DeathWatch and supervision) that
have not been
257.     # explicitly acknowledged or negatively acknowledged are
resent.
258.     # Messages that were negatively acknowledged are always
immediately
259.     # resent.
260.     resend-interval = 2 s
261.
262.     # WARNING: this setting should not be not changed unless all of
its consequences
263.     # are properly understood which assumes experience with
remoting internals
264.     # or expert advice.
265.     # This setting defines the time after redelivery attempts of
internal management
266.     # signals are stopped to a remote system that has been not
```

```
    confirmed to be alive by
267.     # this system before.
268.     initial-system-message-delivery-timeout = 3 m
269.
270.     ### Transports and adapters
271.
272.     # List of the transport drivers that will be loaded by the
    remoting.
273.     # A list of fully qualified config paths must be provided where
274.     # the given configuration path contains a transport-class key
275.     # pointing to an implementation class of the Transport
    interface.
276.     # If multiple transports are provided, the address of the first
277.     # one will be used as a default address.
278.     enabled-transports = ["akka.remote.netty.tcp"]
279.
280.     # Transport drivers can be augmented with adapters by adding
    their
281.     # name to the applied-adapters setting in the configuration of
    a
282.     # transport. The available adapters should be configured in
    this
283.     # section by providing a name, and the fully qualified name of
284.     # their corresponding implementation. The class given here
285.     # must implement
    akka.akka.remote.transport.TransportAdapterProvider
286.     # and have public constructor without parameters.
287.     adapters {
288.       gremlin = "akka.remote.transport.FailureInjectorProvider"
289.       trttl = "akka.remote.transport.ThrottlerProvider"
290.     }
291.
292.     ### Default configuration for the Netty based transport drivers
293.
294.     netty.tcp {
295.       # The class given here must implement the
    akka.remote.transport.Transport
296.       # interface and offer a public constructor which takes two
```

```
    arguments:
297.        #  1) akka.actor.ExtendedActorSystem
298.        #  2) com.typesafe.config.Config
299.        transport-class =
    "akka.remote.transport.netty.NettyTransport"
300.
301.        # Transport drivers can be augmented with adapters by adding
    their
302.        # name to the applied-adapters list. The last adapter in the
303.        # list is the adapter immediately above the driver, while
304.        # the first one is the top of the stack below the standard
305.        # Akka protocol
306.        applied-adapters = []
307.
308.        transport-protocol = tcp
309.
310.        # The default remote server port clients should connect to.
311.        # Default is 2552 (AKKA), use 0 if you want a random
    available port
312.        # This port needs to be unique for each actor system on the
    same machine.
313.        port = 2552
314.
315.        # The hostname or ip to bind the remoting to,
316.        # InetAddress.getLocalHost.getHostAddress is used if empty
317.        hostname = ""
318.
319.        # Enables SSL support on this transport
320.        enable-ssl = false
321.
322.        # Sets the connectTimeoutMillis of all outbound connections,
323.        # i.e. how long a connect may take until it is timed out
324.        connection-timeout = 15 s
325.
326.        # If set to "<id.of.dispatcher>" then the specified
    dispatcher
327.        # will be used to accept inbound connections, and perform IO.
    If "" then
```

```
328.        # dedicated threads will be used.
329.        # Please note that the Netty driver only uses this
    configuration and does
330.        # not read the "akka.remote.use-dispatcher" entry. Instead it
    has to be
331.        # configured manually to point to the same dispatcher if
    needed.
332.        use-dispatcher-for-io = ""
333.
334.        # Sets the high water mark for the in and outbound sockets,
335.        # set to 0b for platform default
336.        write-buffer-high-water-mark = 0b
337.
338.        # Sets the low water mark for the in and outbound sockets,
339.        # set to 0b for platform default
340.        write-buffer-low-water-mark = 0b
341.
342.        # Sets the send buffer size of the Sockets,
343.        # set to 0b for platform default
344.        send-buffer-size = 256000b
345.
346.        # Sets the receive buffer size of the Sockets,
347.        # set to 0b for platform default
348.        receive-buffer-size = 256000b
349.
350.        # Maximum message size the transport will accept, but at
    least
351.        # 32000 bytes.
352.        # Please note that UDP does not support arbitrary large
    datagrams,
353.        # so this setting has to be chosen carefully when using UDP.
354.        # Both send-buffer-size and receive-buffer-size settings has
    to
355.        # be adjusted to be able to buffer messages of maximum size.
356.        maximum-frame-size = 128000b
357.
358.        # Sets the size of the connection backlog
359.        backlog = 4096
```

```
360.
361.       # Enables the TCP_NODELAY flag, i.e. disables Nagle’s
     algorithm
362.       tcp-nodelay = on
363.
364.       # Enables TCP Keepalive, subject to the O/S kernel’s
     configuration
365.       tcp-keepalive = on
366.
367.       # Enables SO_REUSEADDR, which determines when an ActorSystem
     can open
368.       # the specified listen port (the meaning differs between *nix
     and Windows)
369.       # Valid values are "on", "off" and "off-for-windows"
370.       # due to the following Windows bug:
     http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4476378
371.       # "off-for-windows" of course means that it's "on" for all
     other platforms
372.       tcp-reuse-addr = off-for-windows
373.
374.       # Used to configure the number of I/O worker threads on
     server sockets
375.       server-socket-worker-pool {
376.         # Min number of threads to cap factor-based number to
377.         pool-size-min = 2
378.
379.         # The pool size factor is used to determine thread pool
     size
380.         # using the following formula: ceil(available processors *
     factor).
381.         # Resulting size is then bounded by the pool-size-min and
382.         # pool-size-max values.
383.         pool-size-factor = 1.0
384.
385.         # Max number of threads to cap factor-based number to
386.         pool-size-max = 2
387.       }
388.
```

```
389.       # Used to configure the number of I/O worker threads on
     client sockets
390.       client-socket-worker-pool {
391.         # Min number of threads to cap factor-based number to
392.         pool-size-min = 2
393.
394.         # The pool size factor is used to determine thread pool
     size
395.         # using the following formula: ceil(available processors *
     factor).
396.         # Resulting size is then bounded by the pool-size-min and
397.         # pool-size-max values.
398.         pool-size-factor = 1.0
399.
400.         # Max number of threads to cap factor-based number to
401.         pool-size-max = 2
402.       }
403.
404.
405.     }
406.
407.     netty.udp = ${akka.remote.netty.tcp}
408.     netty.udp {
409.       transport-protocol = udp
410.     }
411.
412.     netty.ssl = ${akka.remote.netty.tcp}
413.     netty.ssl = {
414.       # Enable SSL/TLS encryption.
415.       # This must be enabled on both the client and server to work.
416.       enable-ssl = true
417.
418.       security {
419.         # This is the Java Key Store used by the server connection
420.         key-store = "keystore"
421.
422.         # This password is used for decrypting the key store
423.         key-store-password = "changeme"
```

```
424.
425.         # This password is used for decrypting the key
426.         key-password = "changeme"
427.
428.         # This is the Java Key Store used by the client connection
429.         trust-store = "truststore"
430.
431.         # This password is used for decrypting the trust store
432.         trust-store-password = "changeme"
433.
434.         # Protocol to use for SSL encryption, choose from:
435.         # Java 6 & 7:
436.         #   'SSLv3', 'TLSv1'
437.         # Java 7:
438.         #   'TLSv1.1', 'TLSv1.2'
439.         protocol = "TLSv1"
440.
441.         # Example: ["TLS_RSA_WITH_AES_128_CBC_SHA",
    "TLS_RSA_WITH_AES_256_CBC_SHA"]
442.         # You need to install the JCE Unlimited Strength
    Jurisdiction Policy
443.         # Files to use AES 256.
444.         # More info here:
445.         #
    http://docs.oracle.com/javase/7/docs/technotes/guides/security/SunPro
446.         enabled-algorithms = ["TLS_RSA_WITH_AES_128_CBC_SHA"]
447.
448.         # There are three options, in increasing order of security:
449.         # "" or SecureRandom => (default)
450.         # "SHA1PRNG" => Can be slow because of blocking issues on
    Linux
451.         # "AES128CounterSecureRNG" => fastest startup and based on
    AES encryption
452.         # algorithm
453.         # "AES256CounterSecureRNG"
454.         # The following use one of 3 possible seed sources,
    depending on
455.         # availability: /dev/random, random.org and SecureRandom
```

```
    (provided by Java)
456.          # "AES128CounterInetRNG"
457.          # "AES256CounterInetRNG" (Install JCE Unlimited Strength
    Jurisdiction
458.          # Policy Files first)
459.          # Setting a value here may require you to supply the
    appropriate cipher
460.          # suite (see enabled-algorithms section above)
461.          random-number-generator = ""
462.        }
463.      }
464.
465.      ### Default configuration for the failure injector transport
    adapter
466.
467.      gremlin {
468.        # Enable debug logging of the failure injector transport
    adapter
469.        debug = off
470.      }
471.
472.      ### Default dispatcher for the remoting subsystem
473.
474.      default-remote-dispatcher {
475.        type = Dispatcher
476.        executor = "fork-join-executor"
477.        fork-join-executor {
478.          # Min number of threads to cap factor-based parallelism
    number to
479.          parallelism-min = 2
480.          parallelism-max = 2
481.        }
482.      }
483.
484.      backoff-remote-dispatcher {
485.        type = Dispatcher
486.        executor = "fork-join-executor"
487.        fork-join-executor {
```

```
488.          # Min number of threads to cap factor-based parallelism
      number to
489.          parallelism-min = 2
490.          parallelism-max = 2
491.        }
492.      }
493.
494.
495.    }
496.
497.  }
```

akka-testkit

```
 1.  ######################################
 2.  # Akka Testkit Reference Config File #
 3.  ######################################
 4.
 5.  # This is the reference config file that contains all the default
      settings.
 6.  # Make your edits/overrides in your application.conf.
 7.
 8.  akka {
 9.    test {
10.      # factor by which to scale timeouts during tests, e.g. to
      account for shared
11.      # build system load
12.      timefactor =  1.0
13.
14.      # duration of EventFilter.intercept waits after the block is
      finished until
15.      # all required messages are received
16.      filter-leeway = 3s
17.
18.      # duration to wait in expectMsg and friends outside of within()
      block
19.      # by default
```

```
    single-expect-default = 3s

    # The timeout that is added as an implicit by DefaultTimeout trait
    default-timeout = 5s

    calling-thread-dispatcher {
      type = akka.testkit.CallingThreadDispatcherConfigurator
    }
  }
}
```

akka-zeromq

```
#####################################
# Akka ZeroMQ Reference Config File #
#####################################

# This is the reference config file that contains all the default settings.
# Make your edits/overrides in your application.conf.

akka {

  zeromq {

    # The default timeout for a poll on the actual zeromq socket.
    poll-timeout = 100ms

    # Timeout for creating a new socket
    new-socket-timeout = 5s

    socket-dispatcher {
      # A zeromq socket needs to be pinned to the thread that created it.
      # Changing this value results in weird errors and race conditions within
      # zeromq
```

```
22.         executor = thread-pool-executor
23.         type = "PinnedDispatcher"
24.         thread-pool-executor.allow-core-timeout = off
25.       }
26.     }
27.   }
```

Actors

Actors

Actors

Actors

Actor模型为编写并发和分布式系统提供了一种更高的抽象级别。它将开发人员从显式地处理锁和线程管理的工作中解脱出来，使编写并发和并行系统更加容易。Actor模型是在1973年Carl Hewitt的论文中定义的，不过直到被Erlang语言采用后才变得流行起来，一个成功案例是爱立信使用Erlang非常成功地创建了高并发的可靠的电信系统。

Akka Actor的API与Scala Actor类似，它们都借鉴了Erlang的一些语法。

创建Actor

> 注意
>
> 由于*Akka*采用强制性的父子监管，每一个*actor*都被监管着，并且（可能）是别的*actor*的监管者；我们建议你熟悉一下*Actor系统* 和 监管与监控，阅读 *Actor引用，路径与地址*也有帮助。

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（使用标准的Scala模式匹配），以及消息如何被处理。

如下例：

```
1.  import akka.actor.Actor
2.  import akka.actor.Props
3.  import akka.event.Logging
4.
5.  class MyActor extends Actor {
6.    val log = Logging(context.system, this)
7.    def receive = {
8.      case "test" => log.info("received test")
9.      case _      => log.info("received unknown message")
10.   }
11. }
```

请注意Akka Actor的 `receive` 消息循环是完全的，这与Erlang和Scala的Actor行为不同。这意味着你需要提供一个模式匹配，来处理这个actor所能够接受的所有消息的规则，如果你希望处理未知的消息，你需要象上例一样提供一个缺省的case分支。否则会有一个 `akka.actor.UnhandledMessage(message, sender, recipient)` 被发布到Actor系统（ `ActorSystem` ）的事件流（ `EventStream` ）中。

进一步注意到上面定义行为的返回类型是 `Unit` ；如果该actor应该回应收到的消息，则必须明确写出，如下文所述。

`receive` 方法的结果是一个偏函数对象，它存储在actor中作为其”最初的行为”，对actor构造完成后改变行为的进一步信息，请参阅Become/Unbecome。

Props" class="reference-link">Props

`Props` 是一个用来在创建actor时指定选项的配置类，可以把它看作是不可变的，因此在创建包含相关部署信息的actor时（例如使用哪一个调度器(dispatcher)，详见下文），是可以自由共享的。以下是如何创建 `Props` 实例的示例.

```
1. import akka.actor.Props
2.
3. val props1 = Props[MyActor]
4. val props2 = Props(new ActorWithArgs("arg")) // careful, see below
5. val props3 = Props(classOf[ActorWithArgs], "arg")
```

第二个变量 `props2` 演示了创建actor时，怎样将构造函数参数传进去，但它只应在actor之外使用，详见下文。

最后一行展示了一种可能性，它在不关注上下文的情况下传递构造函数参数。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` 。

危险的变量

```
1. // 不建议在另一个actor内使用:
2. // 因为它鼓励封闭作用域
3. val props7 = Props(new MyActor)
```

该方法不建议在另一个actor内使用，因为它鼓励封闭作用域，从而导致不可序列化的 `Props` 和可能的竞态条件（破坏了actor封装）。我

们将提供一个基于宏的解决方案，在将来的版本中没有头痛地支持类似的语法，届时该变量将被恰当的废弃。另一方面在actor的伴生对象下，在一个 `Props` 工厂中使用该变量是完全没问题的，如下面”推荐做法”所述。

这些方法有两个用例：将构造函数的参数传递给该actor—由新推出的 `Props.apply(clazz, args)` 方法解决了，如上面或下面的推荐做法所示—和作为匿名类创建”临时性”的actor。后者应通过命名这些actor，而非匿名来解决（如果他们未在顶级 `object` 中声明，则封闭的实例的 `this` 引用需要作为第一个参数传入）。

> 警告
>
> 在另一个*actor*中声明一个*actor*是非常危险的，会打破*actor*的封装。永远不要将一个*actor*的 *`this`* 引用传进 *`Props`* ！

推荐做法

在每一个 `Actor` 的伴生对象中提供工厂方法是一个好主意，这有助于保持创建合适的 `Props` ，尽可能接近actor的定义。这也避免了使用 `Props.apply(...)` 方法将采用一个“按名”（by-name）参数的缺陷，因为伴生对象的给定代码块中将不会保留包含作用域的引用：

```
1.  object DemoActor {
2.    /**
3.     * Create Props for an actor of this type.
4.     * @param magciNumber The magic number to be passed to this
    actor’s constructor.
5.     * @return a Props for creating this actor, which can then be
    further configured
6.     *         (e.g. calling `.withDispatcher()` on it)
7.     */
8.    def props(magicNumber: Int): Props = Props(new
    DemoActor(magicNumber))
9.  }
10.
```

```
11.  class DemoActor(magicNumber: Int) extends Actor {
12.    def receive = {
13.      case x: Int => sender() ! (x + magicNumber)
14.    }
15.  }
16.
17.  class SomeOtherActor extends Actor {
18.    // Props(new DemoActor(42)) would not be safe
19.    context.actorOf(DemoActor.props(42), "demo")
20.    // ...
21.  }
```

使用Props创建Actor

Actor可以通过将 `Props` 实例传入 `actorOf` 工厂方法来创建， `ActorSystem` 和 `ActorContext` 中都有该方法。

```
1.  import akka.actor.ActorSystem
2.
3.  // ActorSystem is a heavy object: create only one per application
4.  val system = ActorSystem("mySystem")
5.  val myActor = system.actorOf(Props[MyActor], "myactor2")
```

使用 `ActorSystem` 将创建顶级actor，由actor系统提供的守护actor监管；如果使用的是actor的上下文，则创建一个该actor的子actor。

```
1.  class FirstActor extends Actor {
2.    val child = context.actorOf(Props[MyActor], name = "myChild")
3.    // plus some behavior ...
4.  }
```

推荐创建一个树形结构，包含子actor、孙子等等，使之符合应用的逻辑错误处理结构，见Actor系统。

对 `actorOf` 的调用返回一个 `ActorRef` 实例。它是actor实例的句柄，

并且是与之进行交互的唯一方法。 `ActorRef` 是不可变的，与其代表的actor有一一对应关系。 `ActorRef` 也是可序列化，并能通过网络传输。这意味着你可以将其序列化，通过网线发送，并在远程主机上使用它，而且虽然跨网络，它将仍然代表在原始节点上相同的actor。

名称参数是可选的，不过你应该良好命名你的actor，因为名称将被用于日志打印和标识actor。名称不能为空，且不能以 `$` 开头，不过它可以包含URL编码字符（如空格用 `%20` 表示）。如果给定的名称已经被同一个父亲下的另一个子actor使用，则会抛出一个 `InvalidActorNameException` 。

actor在创建时，会自动异步启动。

依赖注入

如果你的actor有带参数的构造函数，则这些参数也需要成为 `Props` 的一部分，如上文所述。但有些情况下必须使用工厂方法，例如，当实际构造函数的参数由依赖注入框架决定。

```
1.  import akka.actor.IndirectActorProducer
2.
3.  class DependencyInjector(applicationContext: AnyRef, beanName:
    String)
4.    extends IndirectActorProducer {
5.
6.    override def actorClass = classOf[Actor]
7.    override def produce =
8.      // obtain fresh Actor instance from DI framework ...
9.  }
10.
11. val actorRef = system.actorOf(
12.   Props(classOf[DependencyInjector], applicationContext, "hello"),
13.   "helloBean")
```

> 警告

> 你有时可能倾向于提供一个 `IndirectActorProducer` 始终返回相同的实例，例如通过使用 `lazy val` 。这是不受支持的，因为它违背了*actor*重启的意义。
>
> 当使用依赖注入框架时，*actor bean*必须不是单例作用域。

关于依赖注入极其Akka集成的更多内容情参见“在Akka中使用依赖注入”指南和Typesafe Activator的“Akka Java Spring”教程。

收件箱

当在actor外编写与actor交互的代码时， `ask` 模式可以作为一个解决方案（见下文），但有两件事它不能做：接收多个答复（例如将 `ActorRef` 订阅到一个通知服务）和查看其他actor的生命周期。为达到这些目的可以使用 `Inbox` 类：

```
1. implicit val i = inbox()
2. echo ! "hello"
3. i.receive() should be("hello")
```

有一个从收件箱到actor引用的一个隐式转换，意味着在这个例子中，发送者引用将成为收件箱隐藏的actor。这就允许了收件箱收到回应，如最后一行所示。监控一个actor也相当简单：

```
1. val target = // some actor
2. val i = inbox()
3. i watch target
```

Actor API

`Actor` trait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` ，其缺省实现是向actor系统的事件流中发布一

条 `akka.actor.UnhandledMessage(message, sender, recipient)` （将配置项 `akka.actor.debug.unhandled` 设置为 `on` 来将它们转换为实际的调试消息）。

另外，它还包括：

- `self` 引用代表本actor的 `ActorRef`
- `sender` 引用代表最近收到消息的发送actor，通常用于下面将讲到的消息回应中
- `supervisorStrategy` 用户可重写它来定义对子actor的监管策略

  该策略通常在actor内声明，这样决定函数就可以访问actor的内部状态：因为失败通知作为消息发送给监管者，并像普通消息一样被处理（尽管不是正常行为），所有的值和actor变量都是可用的，以及 `sender` 引用 （报告失败的将是直接子actor；如果原始失败发生在遥远的后裔，它仍然是一次向上报告一层）。
- `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` ）
  - actor所属的系统
  - 父监管者
  - 所监管的子actor
  - 生命周期监控
  - hotswap行为栈，见Become/Unbecome

你可以import `context` 的成员来避免总加上 `context.` 前缀

```
1.  class FirstActor extends Actor {
2.    import context._
3.    val myActor = actorOf(Props[MyActor], name = "myactor")
4.    def receive = {
```

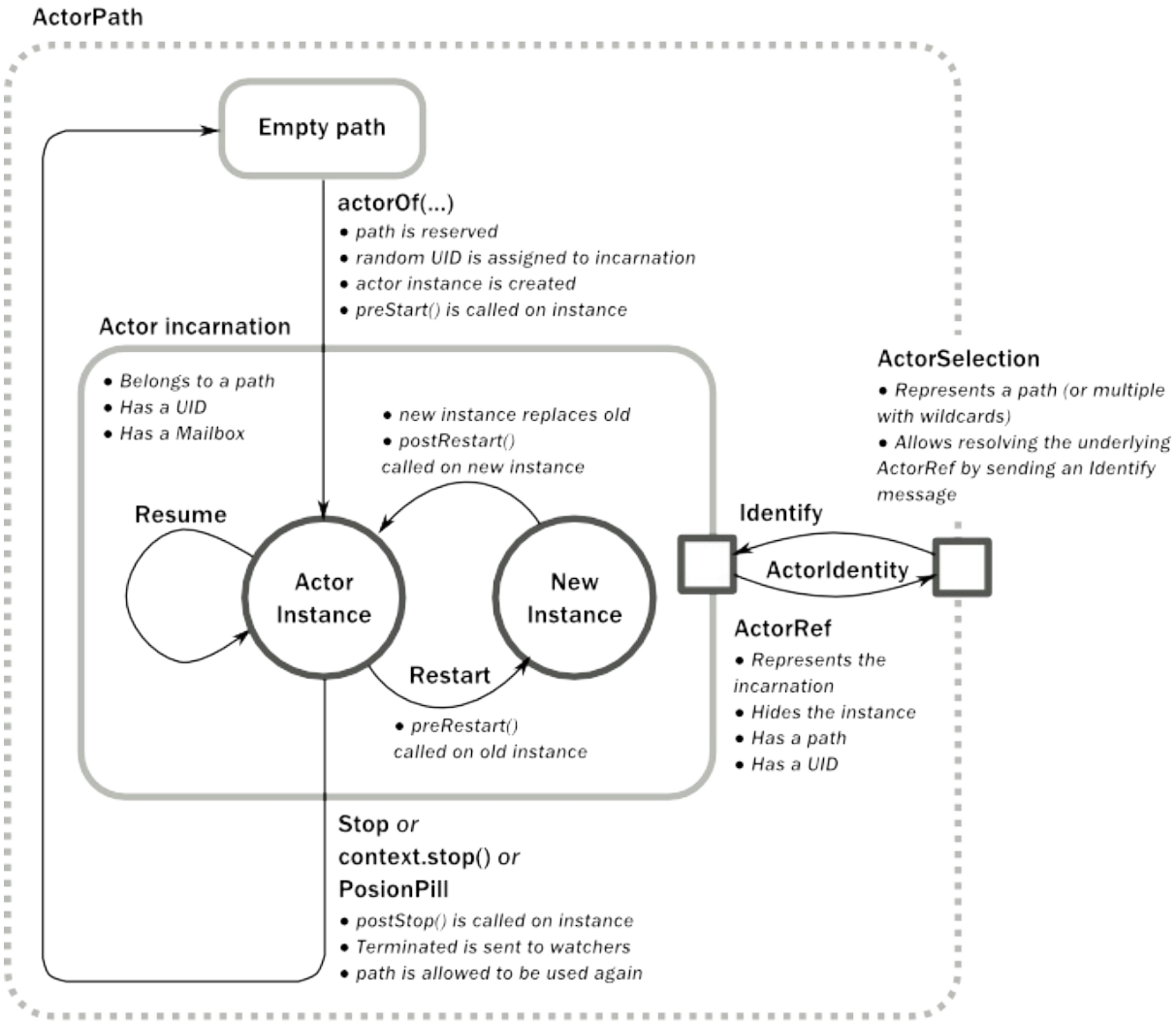
```
    case x => myActor ! x
  }
}
```

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(): Unit = ()

def postStop(): Unit = ()

def preRestart(reason: Throwable, message: Option[Any]): Unit = {
  context.children foreach { child ⇒
    context.unwatch(child)
    context.stop(child)
  }
  postStop()
}

def postRestart(reason: Throwable): Unit = {
  preStart()
}
```

以上代码所示的是 `Actor` trait的缺省实现。

Actor生命周期



actor系统中的路径代表一个"地方"，这里可能会被活着的actor占据。最初（除了系统初始化actor）路径都是空的。在调用 `actorOf()` 时它将为指定路径分配根据传入 `Props` 创建的一个actor化身。actor化身是由路径和一个*UID*标识的。重新启动只会替换有 `Props` 定义的 `Actor` 实例，但不会替换化身，因此UID保持不变。

当actor停止时，其化身的生命周期结束。在这一时间点上相关的生命周期事件被调用，监视该actor的actor都会获得终止通知。当化身停止后，路径可以重复使用，通过 `actorOf()` 创建一个actor。在这种情

况下，除了UID不同外，新化身与老化身是相同的。

`ActorRef` 始终表示化身（路径和UID）而不只是一个给定的路径。因此如果actor停止，并且创建一个新的具有相同名称的actor，则指向老化身的 `ActorRef` 将不会指向新的化身。

相对地， `ActorSelection` 指向路径（或多个路径，如果使用了通配符），且完全不关注有没有化身占据它。因此 `ActorSelection` 不能被监视。获取某路径下的当前化身 `ActorRef` 是可能的，只要向该 `ActorSelection` 发送 `Identify` ，如果收到 `ActorIdentity` 回应，则正确的引用就包含其中（详见通过Actor Selection确定Actor）。也可以使用 `ActorSelection` 的 `resolveOne` 方法，它会返回一个包含匹配 `ActorRef` 的 `Future` 。

使用DeathWatch进行生命周期监控

为了在其它actor终止时（即永久停止，而不是临时的失败和重启)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（见停止 Actor）。这个服务是由actor系统的 `DeathWatch` 组件提供的。

注册一个监视器很简单：

```
1.  import akka.actor.{ Actor, Props, Terminated }
2.
3.  class WatchActor extends Actor {
4.    val child = context.actorOf(Props.empty, "child")
5.    context.watch(child) // <-- this is the only call needed for
    registration
6.    var lastSender = system.deadLetters
7.
8.    def receive = {
9.      case "kill" =>
10.       context.stop(child); lastSender = sender()
```

```
11.       case Terminated(`child`) => lastSender ! "finished"
12.     }
13.   }
```

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。特别地，即使在注册时，被观察的actor已经终止了，监视actor仍然会受到一个 `Terminated` 消息。

多次注册并不表示会有多个消息产生，也不保证有且只有一个这样的消息被接收到：如果被监控的actor已经生成了消息并且已经进入了队列，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的监控注册操作会立刻导致 `Terminated` 消息的产生。

可以使用 `context.unwatch(target)` 来停止对另一个actor生存状态的监控。即使 `Terminated` 已经加入邮箱，该操作仍有效；一旦调用 `unwatch` ，则被观察的actor的 `Terminated` 消息就都不会再被处理。

启动Hook

actor启动后，它的 `preStart` 方法会被立即执行。

```
1.  override def preStart() {
2.    // registering with other actors
3.    someService ! Register(self)
4.  }
```

在actor第一次创建时，将调用此方法。在重新启动期间，它被 `postRestart` 的默认实现调用，这意味着通过重写该方法，你可以选择是仅仅在初始化该actor时调用一次，还是为每次重新启动都调用。actor构造函数中的初始化代码将在每个actor实例创建的时候被调用，这也发生在每次重启时。

重启Hook

所有的actor都是被监管的，即与另一个使用某种失败处理策略的actor绑定在一起。如果在处理一个消息的时候抛出了异常，Actor将被重启（详见监管与监控）。这个重启过程包括上面提到的Hook：

1. 要被重启的actor被通知是通过调用 `preRestart` ，包含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处理而发生的，则所携带的消息为 `None` ，例如，当一个监管者没有处理某个异常继而被其监管者重启时，或者因其兄弟节点的失败导致的重启。如果消息可用，则消息的发送者通常也可用（即通过调用 `sender` ）。

   这个方法是用来完成清理、准备移交给新actor实例等操作的最佳位置。其缺省实现是终止所有子actor并调用 `postStop` 。

2. 最初调用`actorOf`的工厂将被用来创建新的实例。
3. 新的actor的`postRestart`方法被调用时，将携带着导致重启的异常信息。默认实现中， `preStart`被调用时，就像一个正常的启动一样。

actor的重启只会替换掉原来的actor对象；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。触发异常的消息不会被重新接收。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。

> 警告
>
> 要知道失败通知与用户消息的相关顺序不是决定性的。尤其是，在失败以前收到的最后一条消息被处理之前，父节点可能已经重启其子节点了。详细信息请参见“讨论：消息顺序”。

终止 Hook

一个Actor终止后，其 `postStop` hook将被调用，它可以用来，例如

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，即之后发给该actor的消息将被重定向到 `ActorSystem` 的 `deadLetters` 中。

通过Actor Selection定位Actor

如Actor引用，路径与地址中所述，每个actor都拥有一个唯一的逻辑路径，此路径是由从actor系统的根开始的父子链构成；它还拥有一个物理路径，如果监管链包含有远程监管者，此路径可能会与逻辑路径不同。这些路径用来在系统中查找actor，例如，当收到一个远程消息时查找收件者，但是它们更直接的用处在于：actor可以通过指定绝对或相对路径（逻辑的或物理的）来查找其它的actor，并随结果获取一个 `ActorSelection` ：

```
1. // will look up this absolute path
2. context.actorSelection("/user/serviceA/aggregator")
3. // will look up sibling beneath same supervisor
4. context.actorSelection("../joe")
```

其中指定的路径被解析为一个 `java.net.URI` ，它以 `/` 分隔成路径段。如果路径以 `/` 开始，表示一个绝对路径，且从根监管者（ `"/user"` 的父亲）开始查找；否则是从当前actor开始。如果某一个路径段为 `..` ，会找到当前所遍历到的actor的上一级，否则则会向下一级寻找具有该名字的子actor。 必须注意的是actor路径中的 `..` 总是表示逻辑结构，即其监管者。

一个actor selection的路径元素中可能包含通配符，从而允许向匹配模式的集合广播该条消息：

```
1. // will look all children to serviceB with names starting with
   worker
2. context.actorSelection("/user/serviceB/worker*")
```

```
// will look up all siblings beneath same supervisor
context.actorSelection("../*")
```

消息可以通过 `ActorSelection` 发送，并且在投递每条消息时 `ActorSelection` 的路径都会被查找。如果selection不匹配任何actor，则消息将被丢弃。

要获得 `ActorSelection` 的 `ActorRef` ，你需要发送一条消息到selection，然后使用答复消息的 `sender()` 引用即可。有一个内置的 `Identify` 消息，所有actor会理解它并自动返回一个包含 `ActorRef` 的 `ActorIdentity` 消息。此消息被遍历到的actor特殊处理为，如果一个具体的名称查找失败（即一个不含通配符的路径没有对应的活动actor），则会生成一个否定结果。请注意这并不意味着应答消息有到达保证，它仍然是一个普通的消息。

```
import akka.actor.{ Actor, Props, Identify, ActorIdentity,
Terminated }

class Follower extends Actor {
  val identifyId = 1
  context.actorSelection("/user/another") ! Identify(identifyId)

  def receive = {
    case ActorIdentity(`identifyId`, Some(ref)) =>
      context.watch(ref)
      context.become(active(ref))
    case ActorIdentity(`identifyId`, None) => context.stop(self)

  }

  def active(another: ActorRef): Actor.Receive = {
    case Terminated(`another`) => context.stop(self)
  }
}
```

你也可以通过 `ActorSelection` 的 `resolveOne` 方法获取 `ActorSelection` 的一个 `ActorRef` 。如果存在这样的actor，它将返回一个包含匹配的 `ActorRef` 的 `Future` 。如果没有这样的actor存在或识别没有在指定的时间内完成，它将以失败告终——`akka.actor.ActorNotFound` 。

如果开启了远程调用，则远程actor地址也可以被查找：

```
1.  context.actorSelection("akka.tcp://app@otherhost:1234/user/serviceB")
```

一个关于actor查找的示例见远程查找。

> 注意
>
> *actorFor* 因被 *actorSelection* 替代而废弃，因为 *actorFor* 对本地和远程的*actor*表现有所不同。对一个本地*actor*引用，被查找的*actor*需要在查找之前就存在，否则获得的引用是一个 *EmptyLocalActorRef* 。即使后来与实际路径相符的*actor*被创建，所获得引用仍然是这样。对于 *actorFor* 行为获得的远程*actor*引用则不同，每条消息的发送都会在远程系统中进行一次按路径的查找。

消息与不可变性

重要：消息可以是任何类型的对象，但必须是不可变的。（目前）Scala还无法强制不可变性，所以这一点必须作为约定。String、Int、Boolean这些原始类型总是不可变的。 除了它们以外，推荐的做法是使用Scala case class，它们是不可变的（如果你不专门暴露状态的话），并与接收侧的模式匹配配合得非常好。

以下是一个例子：

```
1.  // define the case class
2.  case class Register(user: User)
3.
4.  // create a new case class message
5.  val message = Register(user)
```

发送消息

向actor发送消息需使用下列方法之一。

- `!`意思是“fire-and-forget”，即异步发送一个消息并立即返回。也称为`tell`。
- `?`异步发送一条消息并返回一个`Future`代表一个可能的回应。也称为`ask`。

对每一个消息发送者，分别有消息顺序保证。

> 注意
>
> 使用 *`ask`* 有一些性能内涵，因为需要跟踪超时，需要有桥梁将 *`Promise`* 转为 *`ActorRef`* ，并且需要在远程情况下可访问。所以为了性能应该总选择 *`tell`* ，除非只能选择 *`ask`* 。

Tell: Fire-forget

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。

```
1.  actorRef ! message
```

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(): ActorRef`成员一起发送。目的actor可以使用它来向源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。

Ask: Send-And-Receive-Future

`ask`模式既包含actor也包含future，所以它是一种使用模式，而不是`ActorRef`的方法：

```
import akka.pattern.{ ask, pipe }
import system.dispatcher // The ExecutionContext that will be used
case class Result(x: Int, s: String, d: Double)
case object Request

implicit val timeout = Timeout(5 seconds) // needed for `?` below

val f: Future[Result] =
  for {
    x <- ask(actorA, Request).mapTo[Int] // call pattern directly
    s <- (actorB ask Request).mapTo[String] // call by implicit conversion
    d <- (actorC ? Request).mapTo[Double] // call by symbolic name
  } yield Result(x, s, d)

f pipeTo actorD // .. or ..
pipe(f) to actorD
```

上面的例子展示了将 `ask` 与future上的 `pipeTo` 模式一起使用，因为这是一种非常常用的组合。 请注意上面所有的调用都是完全非阻塞和异步的： `ask` 产生 `Future` ，三个 `Future` 通过for-语法组合成一个新的future，然后用 `pipeTo` 在future上安装一个 `onComplete` -处理器来完成将收集到的 `Result` 发送到另一个actor的动作。

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` 操作包括创建一个内部actor来处理回应，必须为这个内部actor指定一个超时期限，并且过期销毁该内部actor以防止内存泄露。

> 警告
>
> 如果要以异常来完成*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
  }
```

如果actor没有完成future，它会在超时时限到来时过期，以 `AskTimeoutException` 结束。超时的时限是按下面的代码来设置的：

1. 显式指定超时：

```
import akka.util.duration._
import akka.pattern.ask
val future = myActor.ask("hello")(5 seconds)
```

1. 提供类型为 `akka.util.Timeout` 的隐式参数，例如，

```
import scala.concurrent.duration._
import akka.util.Timeout
import akka.pattern.ask
implicit val timeout = Timeout(5 seconds)
val future = myActor ? "hello"
```

参阅 Futures (Scala) 了解更多关于等待和查询future的信息。

`Future` 的 `onComplete` 、 `onSuccess` 或 `onFailure` 方法可以用来注册一个回调，以便在Future完成时得到通知。从而提供一种避免阻塞的方法。

> 警告
>
> 在使用*future*回调如 *`onComplete`* 、 *`onSuccess`* 和 *`onFailure`* 时，在*actor*内部你要小心避免捕捉该*actor*的引用，即不要在回调中调用该*actor*的方法或访问其可变状态。这会破坏*actor*的封装，会引用同步*bug*和竞态条件，因为回调会与此*actor*一同被并发调度。不幸的是目前还没有一种编译时的方法能够探测到这种非法访问。参阅*：* *Actor与共享可变状态*

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“中间人”，但最初的发送者地址/引用将保持不变。当实现类似路由器、负载均衡器、复制器等功能的actor时会很有用。

```
1.  myActor.forward(message)
```

接收消息

Actor必须实现 `receive` 方法来接收消息：

```
1.  protected def receive: PartialFunction[Any, Unit]
```

这个方法应返回一个 `PartialFunction` ，例如一个“match/case”子句，消息可以与其中的不同分支进行scala模式匹配。如下例：

```
1.  import akka.actor.Actor
2.  import akka.actor.Props
3.  import akka.event.Logging
4.
5.  class MyActor extends Actor {
6.    val log = Logging(context.system, this)
7.    def receive = {
8.      case "test" => log.info("received test")
9.      case _      => log.info("received unknown message")
10.   }
11. }
```

回应消息

如果你需要一个用来发送回应消息的目标，可以使用 `sender()` ，它返回一个Actor引用。你可以用 `sender() ! replyMsg` 向这个引用发送回应消息。你也可以将这个ActorRef保存起来，将来再作回应或传给其

它actor。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 “死信”actor引用.

```
  case request =>
    val result = process(request)
    sender() ! result       // will have dead-letter actor as default
```

接收超时

`ActorContext` 的 `setReceiveTimeout` 定义一个不活动时间，在这个时间到达后后，将触发一个 `ReceiveTimeout` 消息的发送。当指定超时时，接收函数应该能够处理 `akka.actor.ReceiveTimeout` 消息。最低支持的超时是 1 毫秒。

请注意接收超时引发的 `ReceiveTimeout` 消息可能在另一条消息后加入队列；因此不能保证收到的接收超时必须与设置的空闲时间长度一致。

一旦进行了设置，接收超时将一直有效（即继续在空闲期后重发）。通过传入 `Duration.Undefined` 关掉此功能。

```
import akka.actor.ReceiveTimeout
import scala.concurrent.duration._
class MyActor extends Actor {
  // To set an initial delay
  context.setReceiveTimeout(30 milliseconds)
  def receive = {
    case "Hello" =>
      // To set in a response to a message
      context.setReceiveTimeout(100 milliseconds)
    case ReceiveTimeout =>
      // To turn it off
      context.setReceiveTimeout(Duration.Undefined)
      throw new RuntimeException("Receive timed out")
  }
```

```
15. }
```

终止Actor

通过调用 `ActorRefFactory` （即 `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。实际的终止操作是异步执行的，即 `stop` 可能在actor被终止之前返回。

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，但是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 `deadLetters` 中，但是这取决于邮箱的实现。

actor的终止分两步： 第一步actor将挂起对邮箱的处理，并向所有子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，最后终止自己（调用 `postStop` ，清空邮箱，向DeathWatch发布 `Terminated` ，通知其监管者）。这个过程保证actor系统中的子树以一种有序的方式终止，将终止命令传播到叶子结点并收集它们回送的确认消息给被终止的监管者。如果其中某个actor没有响应（即由于处理消息用了太长时间以至于没有收到终止命令），整个过程将会被阻塞。

在 `ActorSystem.shutdown()` 被调用时，系统根监管actor会被终止，以上的过程将保证整个系统的正确终止。

`postStop()` hook 是在actor被完全终止以后调用的。这是为了清理资源：

```
1. override def postStop() {
2.   // clean up some resources ...
3. }
```

> 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 `watch()` 正在终止的*actor*，并在 `Terminated` 最终到达后作为回应创建它的替代者。

PoisonPill

你也可以向actor发送 `akka.actor.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。

优雅地终止

如果你需要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` ：

```
1.  import akka.pattern.gracefulStop
2.  import scala.concurrent.Await
3.
4.  try {
5.    val stopped: Future[Boolean] = gracefulStop(actorRef, 5 seconds,
    Manager.Shutdown)
6.    Await.result(stopped, 6 seconds)
7.    // the actor has been stopped
8.  } catch {
9.    // the actor wasn't stopped within 5 seconds
10.   case e: akka.pattern.AskTimeoutException =>
11. }
```

```
1.  object Manager {
2.    case object Shutdown
3.  }
4.
5.  class Manager extends Actor {
6.    import Manager._
7.    val worker = context.watch(context.actorOf(Props[Cruncher],
    "worker"))
```

```
 8.
 9.    def receive = {
10.      case "job" => worker ! "crunch"
11.      case Shutdown =>
12.        worker ! PoisonPill
13.        context become shuttingDown
14.    }
15.
16.    def shuttingDown: Receive = {
17.      case "job" => sender() ! "service unavailable, shutting down"
18.      case Terminated(`worker`) =>
19.        context stop self
20.    }
21.  }
```

当 `gracefulStop()` 成功返回时，actor的 `postStop()` hook将会被执行：在 `postStop()` 结束和 `gracefulStop()` 返回之间存在happens-before边界。

在上面的示例中自定义的 `Manager.Shutdown` 消息是发送到目标actor来启动actor的终止过程。你可以使用 `PoisonPill` ，但之后在停止目标actor之前，你与其他actor的互动的机会有限。在 `postStop` 中，可以处理简单的清理任务。

> 警告
>
> 请记住，*actor*停止和其名称被注销是彼此异步发生的独立事件。因此，在 *`gracefulStop()`* 返回后。你会发现其名称仍可能在使用中。为了保证正确注销，只在你控制的监管者内，并且只在响应 *`Terminated`* 消息时重用名称，即不是用于顶级*actor*。

Become/Unbecome

升级

Akka支持在运行时对Actor消息循环（即其实现）进行实时替换：在actor中调用 `context.become` 方法。 `become` 要求一

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

> 警告
>
> 请注意*actor*被其监管者重启后将恢复其最初的行为。

使用 `become` 替换Actor的行为：

```
1.  class HotSwapActor extends Actor {
2.    import context._
3.    def angry: Receive = {
4.      case "foo" => sender() ! "I am already angry?"
5.      case "bar" => become(happy)
6.    }
7.
8.    def happy: Receive = {
9.      case "bar" => sender() ! "I am already happy :-)"
10.     case "foo" => become(angry)
11.   }
12.
13.   def receive = {
14.     case "foo" => become(angry)
15.     case "bar" => become(happy)
16.   }
17. }
```

`become` 方法的变种还有很多其它的用处，例如实现一个有限状态机（FSM，例子见Dining Hakkers）。它将取代当前的行为 （即行为堆栈的顶部），这意味着你不需要使用 `unbecome` ，相反下一个行为总是被显式安装。

其他使用 `become` 的方式不是替换，而是添加到行为堆栈的顶部。在这种情况下必须小心，以确保长远而言，“pop”操作（即 `unbecome` ）与“push”操作相当，否则会导致内存泄漏 （这就是为什么这种行为不是默认的）。

```
case object Swap
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() // resets the latest 'become' (just for fun)
      }, discardOld = false) // push on top instead of replace
  }
}

object SwapperApp extends App {
  val system = ActorSystem("SwapperSystem")
  val swap = system.actorOf(Props[Swapper], name = "swapper")
  swap ! Swap // logs Hi
  swap ! Swap // logs Ho
  swap ! Swap // logs Hi
  swap ! Swap // logs Ho
  swap ! Swap // logs Hi
  swap ! Swap // logs Ho
}
```

对Scala Actors 嵌套接收消息进行编码而不会造成意外的内存泄露

参阅解嵌套接收消息示例。

贮藏(Stash)

`Stash` 特质使actor可以暂时贮藏消息，来跳过当前行为不能或不应该处理的消息。在actor的消息处理程序改变时，即调用 `context.become` 或 `context.unbecome` 前，所有贮藏的消息可以

是“unstashed”，从而前置到actor的邮箱中。这种方式下，贮藏消息可以按照其原始接收顺序被处理。

> 注意
>
> `Stash` 特质继承自标记特质 `RequiresMessageQueue[DequeBasedMessageQueueSemantics]` ，它要求系统自动为*actor*选择一个基于*deque*的邮箱实现。如果你想更好地控制该邮箱，参阅文档邮箱。

这里是 `Stash` 的一个实际例子：

```
1.  import akka.actor.Stash
2.  class ActorWithProtocol extends Actor with Stash {
3.    def receive = {
4.      case "open" =>
5.        unstashAll()
6.        context.become({
7.          case "write" => // do writing...
8.          case "close" =>
9.            unstashAll()
10.           context.unbecome()
11.         case msg => stash()
12.       }, discardOld = false) // stack on top instead of replacing
13.     case msg => stash()
14.   }
15. }
```

调用 `stash()` 将当前消息（actor最后接收到的消息）添加到actor的贮藏处。通常在其它case语句不能处理该消息时，在默认处理中调用来把消息贮藏起来。两次贮藏同一个消息是非法的；这样做会抛出 `IllegalStateException` 。贮藏操作也可能是有界的，在这种情况下调用 `stash()` 可能会导致超出容量，结果抛出 `StashOverflowException` 。可以使用邮箱配置中的 `stash-capacity` （一个 `Int` ）配置贮存能力。

调用 `unstashAll()` 将把贮藏的消息转移到actor的邮箱中，直到邮箱满（请注意消息从贮藏处被前置到邮箱中的）。万一有界的邮箱溢出，

则抛出 `MessageQueueAppendFailedException` 。贮藏箱被保证在调用 `unstashAll()` 后被清空。

贮藏箱由 `scala.collection.immutable.Vector` 支持。其结果是，即使有很大数目的消息被贮藏也不会对性能有大的影响。

> 警告
>
> *Stash* 特质必须在 *preRestart* 回调被任何一个特质/类重写之前，被混入到 *Actor* （的一个子类）中。这意味着如果 *MyActor* 重写 *preRestart* ，则不可能这样写— *Actor with MyActor with Stash* 。

请注意贮藏箱是actor的瞬时状态，不同于邮箱。因此，它应该像actor中其他具有相同属性的状态一样被管理。 `Stash` 的 `preRestart` 实现将调用 `unstashAll()` ，这通常也是期望的行为。

> 注意
>
> 如果你想强制你的*actor*只能在无界贮藏箱下工作，则你应该换用 *UnboundedStash* 特质。

杀死actor

你可以发送 `Kill` 消息来杀死actor。这将导致actor抛出 `ActorKilledException` ，触发失败。该actor将暂停操作，其主管也将会被问及如何处理这一失败，这可能意味着恢复actor、 重新启动或完全终止它。更多的信息，请参阅监管的意思。

像这样使用 `Kill` ：

```
1.  // kill the 'victim' actor
2.  victim ! Kill
```

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（即从邮箱中取出并交给当前行为后）发生了异常，这个消息将被丢失。必须明白它不会被放回到邮箱中。所以如果你希望重试对消息的处理，你需要自己抓住异常然后在异常处理流程中重试。请确保限制重试的次数，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）。另一种可能性请参见PeekMailbox 模式。

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，仍然是相同的邮箱在那里。邮箱中的所有消息不会丢失。

actor会怎样

如果actor代码抛出了异常，actor会被暂停并启动监管过程（参见监管与监控）。根据监管者的策略，actor可以被恢复（好像什么也没有发生过）、重启（消灭其内部状态并从零开始）或终止。

使用PartialFunction链来扩展actor

有时在一些actor中分享共同的行为，或通过若干小的函数构成一个actor的行为是很有用的。这由于actor的 `receive` 方法返回一个 `Actor.Receive` （ `PartialFunction[Any,Unit]` 的类型别名）而使之成为可能，多个偏函数可以使用 `PartialFunction#orElse` 链接在一起。你可以根据需要链接尽可能多的功能，但是你要牢记”第一个匹配”获胜——这在组合可以处理同一类型的消息的功能时会很重要。

例如，假设你有一组actor是生产者 `Producers` 或消费者 `Consumers` ，然而有时候需要actor分享这两种行为。这可以很容易实现而无需重复代码，通过提取行为的特质和并将actor的 `receive` 实

现为这些偏函数的组合。

```
1.  trait ProducerBehavior {
2.    this: Actor =>
3.
4.    val producerBehavior: Receive = {
5.      case GiveMeThings =>
6.        sender() ! Give("thing")
7.    }
8.  }
9.
10. trait ConsumerBehavior {
11.   this: Actor with ActorLogging =>
12.
13.   val consumerBehavior: Receive = {
14.     case ref: ActorRef =>
15.       ref ! GiveMeThings
16.
17.     case Give(thing) =>
18.       log.info("Got a thing! It's {}", thing)
19.   }
20. }
21.
22. class Producer extends Actor with ProducerBehavior {
23.   def receive = producerBehavior
24. }
25.
26. class Consumer extends Actor with ActorLogging with
    ConsumerBehavior {
27.   def receive = consumerBehavior
28. }
29.
30. class ProducerConsumer extends Actor with ActorLogging
31.   with ProducerBehavior with ConsumerBehavior {
32.
33.   def receive = producerBehavior orElse consumerBehavior
34. }
35.
```

```
36.  // protocol
37.  case object GiveMeThings
38.  case class Give(thing: Any)
```

不同于继承，相同的模式可以通过组合实现—可以简单地通过委托的偏函数组合成 `receive` 方法。

初始化模式

actor丰富的生命周期钩子（hook）提供一个有用的工具包，可用于实现各种初始化模式。在 `ActorRef` 的生命中，actor可能会经历多次重启，老的实例被替换为新的实例，除观察者以外是觉察不到的，只能看到一个 `ActorRef` 。

一个人可能把新实例看做是"化身"。初始化对actor每个化身都是必要的，但有时你需要初始化只在第一个实例创建时，即 `ActorRef` 创建时发生。以下各节提供了满足不同的初始化需求的模式。

通过构造函数初始化

使用构造函数初始化有各种好处。首先，使得用 `val` 字段来存储在actor实例的生命周期内不变的状态成为可能，使actor的实现更加健壮。对actor的每个化身都会调用一次构造函数，因此，actor内部总是可以假定正确地完成初始化。这也是这种方法的缺点，例如当想要避免在重启时重新初始化内部状态的情况下。例如，跨重启保留子actor经常很有用。下面提供了该情况的一种模式。

通过preStart初始化

actor的 `preStart()` 方法只在第一个实例的初始化时调用一次，即 `ActorRef` 创建时。在重新启动后， `preStart()` 是由 `postRestart()` 调用，因此如果重写， `preStart()` 对每个化身都会被调用。然而，重写 `postRestart()` 可以禁用此行为，并确保只有一个

对 `preStart()` 的调用。

这种模式的一个有用用法是禁止在重启期间为子actor创建新 `ActorRefs` 。这可以通过重写 `preRestart()` 实现：

```
override def preStart(): Unit = {
  // Initialize children here
}

// Overriding postRestart to disable the call to preStart()
// after restarts
override def postRestart(reason: Throwable): Unit = ()

// The default implementation of preRestart() stops all the children
// of the actor. To opt-out from stopping the children, we
// have to override preRestart()
override def preRestart(reason: Throwable, message: Option[Any]): Unit = {
  // Keep the call to postStop(), but no stopping of children
  postStop()
}
```

请注意，子actor仍会重新启动，但不会创建新的 `ActorRef` 。可以以递归方式为子actor应用相同的原则，确保其 `preStart()` 方法只在创建引用时被调用一次。

有关更多信息，请参见重启的含义。

通过消息传递初始化

有些情况下不可能在构造函数中传入actor初始化所需要的所有信息，例如存在循环依赖关系。在这种情况下actor应该监听初始化消息，并使用 `become()` 或一个有限状态机状态转换来编码actor的初始化和未初始化状态。

```
1.  var initializeMe: Option[String] = None
2.
3.  override def receive = {
4.    case "init" =>
5.      initializeMe = Some("Up and running")
6.      context.become(initialized, discardOld = true)
7.
8.  }
9.
10. def initialized: Receive = {
11.   case "U OK?" => initializeMe foreach { sender() ! _ }
12. }
```

如果actor在初始化之前可能收到消息，一个有用的 `Stash` 工具可以用来存储消息直到初始化完成，并在actor完成初始化后回放这些消息。

> 警告
>
> 此模式应小心使用，并且仅当上述模式都不适用时才应用。其潜在的问题之一是当发送到远程的*actor*时，消息可能会丢失。此外，发布一个处于未初始化状态的 *`ActorRef`* 可能会导致竞态条件，即在初始化完成前它接收到一个用户消息。

有类型Actor

有类型Actor

有类型Actor是Active Objects 模式的一种实现。Smalltalk诞生之时，就已经缺省地将方法调用从同步操作换为异步派发。

有类型Actor由两 “部分” 组成， 一个公开的接口和一个实现， 如果你有“企业级”Java的开发经验， 则应该非常熟悉。 对普通actor来说，你拥有一个外部API（公开接口的实例）来将方法调用异步地委托

给其实现的私有实例。

有类型Actor相对于普通Actor的优势在于有类型Actor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，即你不能使用 `become/unbecome` 。

有类型Actor是使用JDK Proxies实现的，JDK Proxies提供了非常简单的api来拦截方法调用。

> 注意
>
> 和普通*Akka actor*一样，有类型*actor*一次也只处理一个消息。

何时使用有类型actor

有类型actor是桥接actor系统（”内部”）和非actor代码 （”外部”）的良好方式，因为它们允许你在外部编写普通OO式代码。把它们看做大门：其实用性在于私有领域和公共接口之间，而你不想你的房子内部有太多的门，不是吗？更长的讨论请参见这篇博客。

更多的背景：TypedActors可以很容易被滥用作RPC，它们都是一个抽象概念，众所周知是有缺陷的。因此当我们容易和正确的编写高度可扩展的并行软件时，TypedActors并非首选。他们有自己的定位，必要时才使用它们。

工具箱

在创建第一个有类型Actor之前，我们先了解一下我们手上可供使用的工具，它位于 `akka.actor.TypedActor` 中。

```
1. import akka.actor.TypedActor
2.
3. //返回有类型actor扩展
4. val extension = TypedActor(system) //system是一个Actor系统实例
```

```
5.
6.  //判断一个引用是否是有类型actor代理
7.  TypedActor(system).isTypedActor(someReference)
8.
9.  //返回一个外部有类型actor代理所代表的Akka actor
10. TypedActor(system).getActorRefFor(someReference)
11.
12. //返回当前的ActorContext,
13. // 此方法仅在一个TypedActor 实现的方法中有效
14. val c: ActorContext = TypedActor.context
15.
16. //返回当前有类型actor的外部代理,
17. // 此方法仅在一个TypedActor 实现的方法中有效
18. val s: Squarer = TypedActor.self[Squarer]
19.
20. //返回一个有类型Actor扩展的上下文实例
21. //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
22. TypedActor(TypedActor.context)
```

> 警告
>
> 就象不应该暴露*Akka actor*的 `this` 一样，不要暴露有类型*Actor*的 `this` ，你应该传递其外部代理引用，它可以在你的有类型*Actor*中用 `TypedActor.self` 获得，这是你的外部标识，就象 `ActorRef` 是*Akka actor*的外部标识一样。

创建有类型Actor

要创建有类型Actor，需要一个或多个接口，和一个实现。

我们的示例接口：

```
1.  trait Squarer {
2.    def squareDontCare(i: Int): Unit //fire-forget
3.
4.    def square(i: Int): Future[Int] //non-blocking send-request-reply
5.
6.    def squareNowPlease(i: Int): Option[Int] //blocking send-request-
    reply
```

```
7.
8.    def squareNow(i: Int): Int //blocking send-request-reply
9.
10.   @throws(classOf[Exception]) //declare it or you will get an
    UndeclaredThrowableException
11.   def squareTry(i: Int): Int //blocking send-request-reply with
    possible exception
12. }
```

好，现在我们有了一些可以调用的方法，但我们需要在SquarerImpl中实现。

```
1.  class SquarerImpl(val name: String) extends Squarer {
2.
3.    def this() = this("default")
4.    def squareDontCare(i: Int): Unit = i * i //Nobody cares :(
5.
6.    def square(i: Int): Future[Int] = Future.successful(i * i)
7.
8.    def squareNowPlease(i: Int): Option[Int] = Some(i * i)
9.
10.   def squareNow(i: Int): Int = i * i
11.
12.   def squareTry(i: Int): Int = throw new Exception("Catch me!")
13. }
```

太好了，我们现在有了接口，也有了对这个接口的实现，我们还知道如何从他们来创建一个有类型actor，现在我们来看看如何调用这些方法。

创建我们的Squarer的有类型actor实例的最简单方法是：

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

第一个类型是代理的类型，第二个类型是实现的类型。如果要调用某特

定的构造方法要这样做：

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

由于你提供了一个 Props，你可以指定使用哪个派发器，缺省的超时时间等。

方法派发语义

方法返回：

- `Unit` 会以 `fire-and-forget` 语义进行派发，与 `ActorRef.tell` 完全一致。
- `akka.dispatch.Future[_]` 会以 `send-request-reply` 语义进行派发，与 `ActorRef.ask` 完全一致。
- `scala.Option[_]` 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 `scala.None`，否则返回包含结果的 `scala.Some[_]`。在这个调用中发生的异常将被重新抛出。
- 任何其它类型的值将以 `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。

消息与不可变性

虽然Akka不能强制要求你传给有类型Actor方法的参数类型是不可变的，我们强烈建议只传递不可变参数。

单向消息发送

```
1.  mySquarer.squareDontCare(10)
```

就是这么简单！方法会在另一个线程中异步地调用。

请求-响应消息发送

```
1.  val oSquare = mySquarer.squareNowPlease(10) //Option[Int]
```

如果需要，这会阻塞到有类型actor的Props中设置的超时时限。如果超时，会返回 `None` 。

```
1.  val iSquare = mySquarer.squareNow(10) //Int
```

如果需要，这会阻塞到有类型actor的Props中设置的超时时限。如果超时，会抛出 `java.util.concurrent.TimeoutException` 。

请求-以future作为响应的消息发送

```
1.  val fSquare = mySquarer.square(10) //A Future[Int]
```

这个调用是异步的，返回的Future可以用作异步组合。

终止有类型Actor

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

```
1.  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。

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

这将会在有类型actor完成所有入队的调用后异步地终止它。

有类型Actor监管树

你可以通过传入一个 `ActorContext` 来获得有类型Actor上下文，所以你可以对它调用 `typedActorOf(..)` 来创建有类型子actor。

```
1. //Inside your Typed Actor
2. val childSquarer: Squarer =
3.
   TypedActor(TypedActor.context).typedActorOf(TypedProps[SquarerImpl]
   ())
4. //Use "childSquarer" as a Squarer
```

通过将 `ActorContext` 作为参数传给 `TypedActor.get(…)` ，也可以为普通的Akka actor创建有类型子actor。

监管策略

通过让你的有类型Actor的具体实现类实现 `TypedActor.Supervisor` 方法，你可以定义用来监管子actor的策略，就像监管与监控 和容错(Scala)所描述的。

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进有类型actor的整个生命周期。

接收任意消息

如果你的有类型actor的实现类扩展了 `akka.actor.TypedActor.Receiver` ，所有非方法调用 `MethodCall` 的消息会被传给 `onReceive` 方法.

这使你能够对DeathWatch的 `Terminated` 消息及其它类型的消息进行处理，例如，与无类型actor进行交互的场合。

代理

你可以使用带TypedProps和ActorRef参数的 `typedActorOf` 来将指定的Actor引用代理成一个有类型Actor。这在你需要与远程主机上的有类型Actor通信时会有用， 只要将 `ActorRef` 传递给 `typedActorOf` 即可。

> 注意
>
> 目标*Actor*引用需要能处理 *`MethodCall`* 消息.

查找与远程处理

因为 `TypedActor` 底层还是 `Akka Actors` ，你可以使用 `typedActorOf` 来代理可能在远程节点上的 `ActorRefs` 。

```
1. val typedActor: Foo with Bar =
2.   TypedActor(system).
3.     typedActorOf(
4.       TypedProps[FooBar],
5.       actorRefToRemoteActor)
6. //Use "typedActor" as a FooBar
```

功能扩充

以下是使用traits来为你的有类型actor混入行为的示例:

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

trait Bar {
  def doBar(str: String): Future[String] =
    Future.successful(str.toUpperCase)
}

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)
```

有类型路由器模式

有时你想要传播多个actor之间的消息。在Akka中实现这一目标的最简单方法是使用一个路由器，可以实现特定的路由逻辑，例如最小邮箱 `smallest-mailbox` 或一致性哈希 `consistent-hashing` 等。

路由器不能直接提供给有类型actor，但可以很容易的利用非类型化的路由器，并在其使用一个有类型代理即可。为了展示，让我们创建有类型actor并分配它们一些随机 `id` ，所以我们知道事实上，路由器已向消息发送给不同的actor：

```
trait HasName {
  def name(): String
}

class Named extends HasName {
```

```
  import scala.util.Random
  private val id = Random.nextInt(1024)

  def name(): String = "name-" + id
}
```

为了在此类actor的几个实例中轮询访问（round robin），你可以简单地创建一个普通的非类型化路由器，然后像下面的示例所示把它包装为一个 `TypedActor` 。这之所以能够正确工作，是因为有类型actor与普通actor使用相同的机制通讯，其方法调用最终都被转换为 `MethodCall` 消息的发送。

```
def namedActor(): HasName =
TypedActor(system).typedActorOf(TypedProps[Named]())

// prepare routees
val routees: List[HasName] = List.fill(5) { namedActor() }
val routeePaths = routees map { r =>
  TypedActor(system).getActorRefFor(r).path.toStringWithoutAddress
}

// prepare untyped router
val router: ActorRef =
system.actorOf(RoundRobinGroup(routeePaths).props())

// prepare typed proxy, forwarding MethodCall messages to `router`
val typedRouter: HasName =
  TypedActor(system).typedActorOf(TypedProps[Named](), actorRef =
router)

println("actor was: " + typedRouter.name()) // name-184
println("actor was: " + typedRouter.name()) // name-753
println("actor was: " + typedRouter.name()) // name-320
println("actor was: " + typedRouter.name()) // name-164
```

容错

容错

如Actor系统中所述，每一个actor都是其子actor的监管者，而且每一个actor会定义一个处理错误的监管策略。这个策略制定以后就不能修改，因为它被集成为actor系统结构所必须的一部分。

错误处理实践

首先我们来看一个例子，演示处理数据库错误的一种方法，数据库错误是真实应用中的典型错误类型。当然在实际的应用中这要依赖于当数据库发生错误时能做些什么， 在这个例子中，我们使用尽量重新连接的方法。

阅读以下源码。其中的注释解释了错误处理的各个片段以及为什么要加上它们。我们还强烈建议运行这个例子，因为根据输出日志理解运行时发生的事情会比较容易。

- 容错示例图片
- 容错示例完整源代码

创建一个监管策略

以下章节更加深入地解释了错误处理机制和可选的方法。

为了演示我们假设有这样的策略：

```
1.  import akka.actor.OneForOneStrategy
2.  import akka.actor.SupervisorStrategy._
3.  import scala.concurrent.duration._
4.
5.  override val supervisorStrategy =
6.    OneForOneStrategy(maxNrOfRetries = 10, withinTimeRange = 1
    minute) {
7.      case _: ArithmeticException      => Resume
8.      case _: NullPointerException     => Restart
9.      case _: IllegalArgumentException => Stop
10.     case _: Exception                => Escalate
11.   }
```

我选择了一些非常著名的异常类型来演示监管与监控中描述的错误处理方式。首先，它是一对一策略，意思是每一个子actor会被单独处理（多对一的策略与之相似，唯一的区别在于任何决策都应用于监管者的所有子actor，而不仅仅是出错的那一个）。这里我们对重启的频率作了限制，即每分钟最多进行10次重启；所有这样的设置都可以被空着，也就是说，相应的限制并不被采用，留下了设置重启频率的绝对上限或让重启无限进行的可能性。如果超出上限则子actor将被终止。

构成主体的match语句的类型是 `Decider` ，它是一个 `PartialFunction[Throwable, Directive]` 。该部分将把子actor的失败类型映射到相应的指令上。

> 注意
>
> 如果在监管*actor*内部声明策略（而不是在伴生对象中），其决策者就能够以线程安全的方式访问*actor*的所有内部状态，包括获取对当前失败的子*actor*引用（作为失败消息的发送者）。

缺省的监管机制

如果定义的监管机制没有覆盖抛出的异常，将使用 `Escalate` 上溯机制。

如果某个actor没有定义监管机制，下列异常将被缺省地处理为：

- `ActorInitializationException` 将终止出错的子actor
- `ActorKilledException` 将终止出错的子actor
- `Exception` 将重启出错的子actor
- 其它的 `Throwable` 将被上溯传给父actor

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。

你可以将自己的策略与默认策略结合：

```
1.  import akka.actor.OneForOneStrategy
2.  import akka.actor.SupervisorStrategy._
3.  import scala.concurrent.duration._
4.
5.  override val supervisorStrategy =
6.    OneForOneStrategy(maxNrOfRetries = 10, withinTimeRange = 1
    minute) {
7.      case _: ArithmeticException => Resume
8.      case t =>
9.        super.supervisorStrategy.decider.applyOrElse(t, (_: Any) =>
    Escalate)
10.   }
```

停止监管策略

Erlang方式的策略是当actor失败时，终止它们，然后当DeathWatch通知子actor消失时，采取正确的措施。这一策略还提供了预打包为 `SupervisorStrategy.stoppingStrategy` 及伴随

的 `StoppingSupervisorStrategy` 配置，当你想要使用 `"/user"` 监管者时可以使用它。

actor失败的日志记录

默认情况下 `SupervisorStrategy` 会日志记录失败，除非他们被上溯升级。升级的失败应该被树形结构中更高级的监管者处理，即可能在那里记录日志。

当实例化时，你可以通过将 `loggingEnabled` 设置为 `false` 来取消 `SupervisorStrategy` 的默认日志记录。自定义日志记录可以在 `Decider` 内完成。请注意当 `SupervisorStrategy` 是在监管actor内声明的时候，可以通过 `sender` 获取当前失败的子actor引用。

你也可以通过重写 `logFailure` 方法在你自己的 `SupervisorStrategy` 实现定制日志记录。

监督顶级actor

顶级actor是指那些使用 `system.actorOf()` 创建的，而它们将是User监管Actor的子actor。这里没有使用特殊规则，监管者只是应用了已配置的策略。

测试应用

以下部分展示了实际中不同的指令的效果，为此我们需要创建一个测试环境。首先我们需要一个合适的监管者：

```
1.  import akka.actor.Actor
2.
3.  class Supervisor extends Actor {
4.    import akka.actor.OneForOneStrategy
5.    import akka.actor.SupervisorStrategy._
6.    import scala.concurrent.duration._
```

```
7.
8.     override val supervisorStrategy =
9.       OneForOneStrategy(maxNrOfRetries = 10, withinTimeRange = 1
   minute) {
10.        case _: ArithmeticException      => Resume
11.        case _: NullPointerException     => Restart
12.        case _: IllegalArgumentException => Stop
13.        case _: Exception                => Escalate
14.      }
15.
16.    def receive = {
17.      case p: Props => sender() ! context.actorOf(p)
18.    }
19.  }
```

该监管者将被用来创建一个可以做试验的子actor:

```
1.  import akka.actor.Actor
2.
3.  class Child extends Actor {
4.    var state = 0
5.    def receive = {
6.      case ex: Exception => throw ex
7.      case x: Int        => state = x
8.      case "get"         => sender() ! state
9.    }
10. }
```

这个测试可以用测试Actor系统(Scala)中的工具来进行简化，比如 `AkkaSpec` 是 `TestKit with WordSpec with MustMatchers` 的一个方便的混合

```
1.  import akka.testkit.{ AkkaSpec, ImplicitSender, EventFilter }
2.  import akka.actor.{ ActorRef, Props, Terminated }
3.
4.  class FaultHandlingDocSpec extends AkkaSpec with ImplicitSender {
5.
```

```
6.   "A supervisor" must {
7.
8.     "apply the chosen strategy for its child" in {
9.       // code here
10.    }
11.  }
12. }
```

现在我们来创建actor：

```
1. val supervisor = system.actorOf(Props[Supervisor], "supervisor")
2.
3. supervisor ! Props[Child]
4. val child = expectMsgType[ActorRef] // 从 TestKit 的 testActor 中获取回应
```

第一个测试是为了演示 `Resume` 指令，我们试着将actor设为非初始状态然后让它出错：

```
1. child ! 42 // 将状态设为 42
2. child ! "get"
3. expectMsg(42)
4.
5. child ! new ArithmeticException // crash it
6. child ! "get"
7. expectMsg(42)
```

可以看到错误处理指令完后仍能得到42的值。现在如果我们将错误换成更严重的 `NullPointerException` ，情况就不同了：

```
1. child ! new NullPointerException // crash it harder
2. child ! "get"
3. expectMsg(0)
```

而最后当致命的 `IllegalArgumentException` 发生时子actor将被其监管

者终止：

```
1. watch(child) // have testActor watch “child”
2. child ! new IllegalArgumentException // break it
3. expectMsgPF() { case Terminated(`child`) => () }
```

到目前为止监管者完全没有被子actor的错误所影响，因为指令集确实处理了这些错误。而对于 `Exception` ，就不是这么回事了，监管者会将失败上溯传递。

```
1. supervisor ! Props[Child] // create new child
2. val child2 = expectMsgType[ActorRef]
3.
4. watch(child2)
5. child2 ! "get" // verify it is alive
6. expectMsg(0)
7.
8. child2 ! new Exception("CRASH") // escalate failure
9. expectMsgPF() {
10.   case t @ Terminated(`child2`) if t.existenceConfirmed => ()
11. }
```

监管者自己是被 `ActorSystem` 的顶级actor所监管的，顶级actor的缺省策略是对所有的 `Exception` 情况（注意 `ActorInitializationException` 和 `ActorKilledException` 例外）进行重启．由于缺省的重启指令会杀死所有的子actor，所以我们知道（期望）可怜的子actor最终无法从这个失败中幸免。

如果这不是我们希望的行为（这取决于实际情况），我们需要使用一个不同的监管者来覆盖这个行为。

```
1. class Supervisor2 extends Actor {
2.   import akka.actor.OneForOneStrategy
3.   import akka.actor.SupervisorStrategy._
```

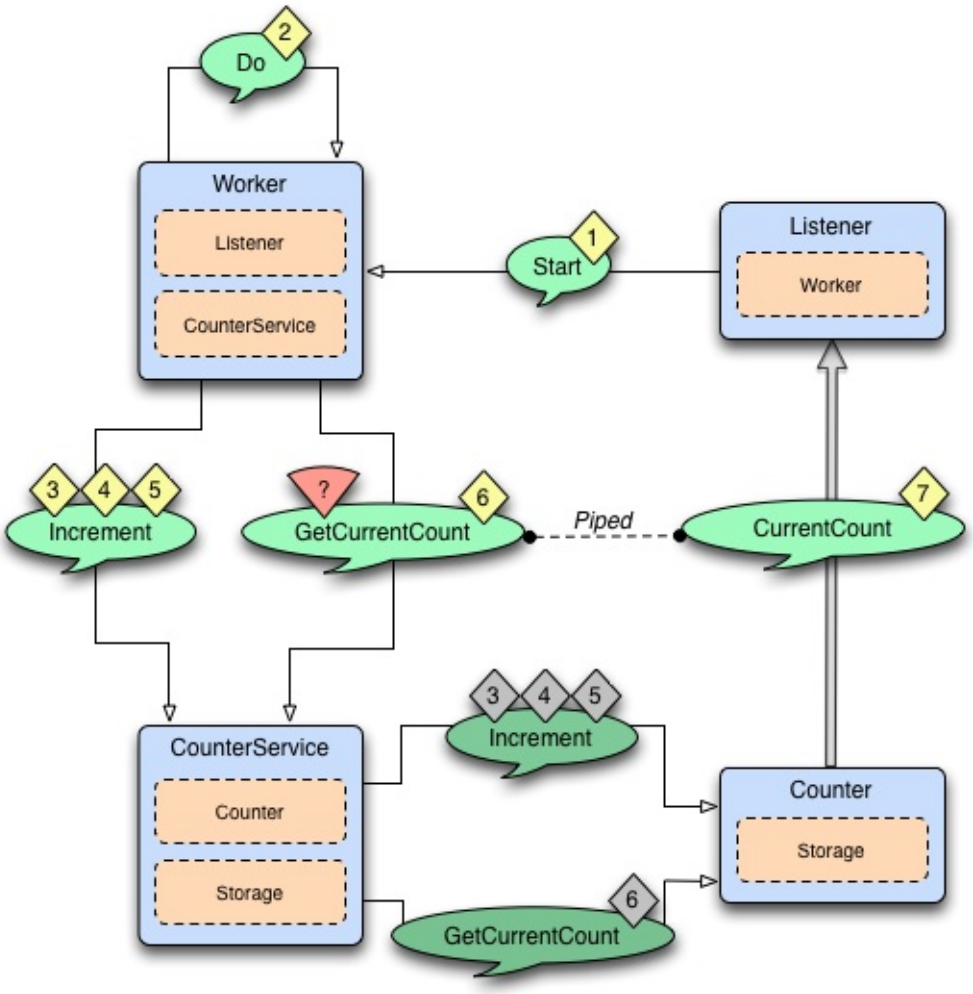
```
  import scala.concurrent.duration._

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

  def receive = {
    case p: Props => sender() ! context.actorOf(p)
  }
  // override default to kill all children during restart
  override def preRestart(cause: Throwable, msg: Option[Any]) {}
}
```

在这个父actor之下，子actor在上溯的重启中得以幸免，在如下这个最后的测试中：

```
val supervisor2 = system.actorOf(Props[Supervisor2], "supervisor2")

supervisor2 ! Props[Child]
val child3 = expectMsgType[ActorRef]

child3 ! 23
child3 ! "get"
expectMsg(23)

child3 ! new Exception("CRASH")
child3 ! "get"
expectMsg(0)
```

容错示例

容错示例

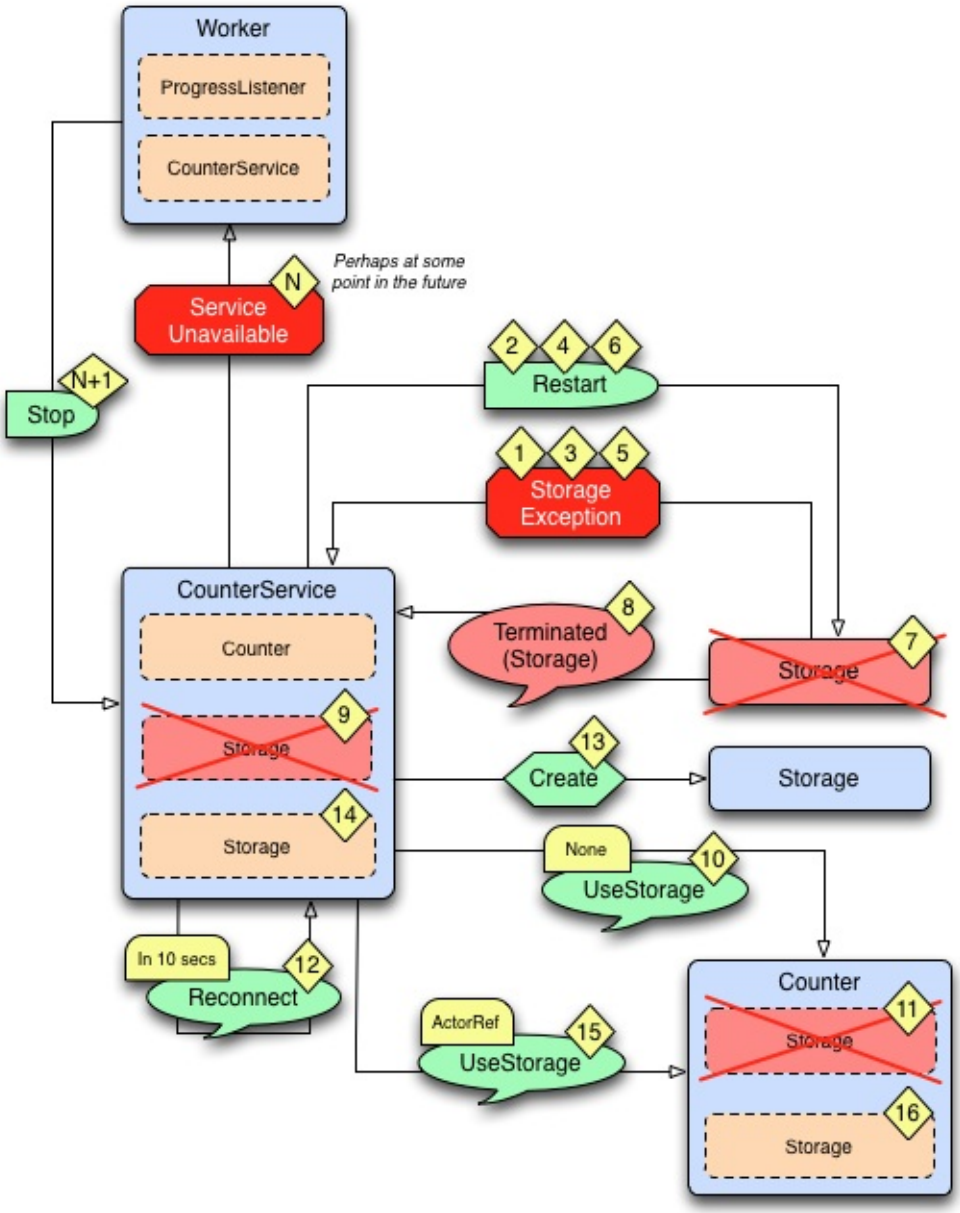


以上图示了正常的消息流。

正常流程：

步	描述

骤	
1	过程 `Listener` 启动工作。
2	`Worker` 通过定期向自己发送 `Do` 消息来计划工作
3, 4, 5	接收到 `Do` 时 `Worker` 通知 `CounterService` 更新计数器值，三次。 `Increment` 消息被转发给 `Counter` ，它会更新自己的计数器变量并将当前值发给 `Storage` 。
6, 7	`Worker` 向 `CounterService` 请求计数器的当前值并将结果回送给 `Listener` 。



以上图示了当发生数据库失败时的过程

失败流程：

步骤	描述
1	`Storage` 抛出 `StorageException`。
2	`CounterService` 是 `Storage` 的监管者，`StorageException` 被抛出时它将重启 `Storage`。
3, 4, 5, 6	`Storage` 仍旧失败，又被重启.
7	在5秒内三次失败和重启后 `Storage` 被它的监管者，即 `CounterService` 终止。
8	`CounterService` 同时监视着 `Storage` 并在 `Storage` 被终止时收到 `Terminated` 消息…
9, 10, 11	告诉 `Counter` 当前没有可用的 `Storage`。
12	`CounterService` 计划一个 `Reconnect` 消息发给自己。
13, 14	收到 `Reconnect` 消息后它创建一个新的 `Storage`…
15, 16	并通知 `Counter` 使用新的 `Storage`。

容错示例完整源代码" class="reference-link">容错示例完整源代码

```
1.  import akka.actor._
2.  import akka.actor.SupervisorStrategy._
3.  import scala.concurrent.duration._
4.  import akka.util.Timeout
5.  import akka.event.LoggingReceive
6.  import akka.pattern.{ ask, pipe }
7.  import com.typesafe.config.ConfigFactory
8.
9.  /**
10.  * Runs the sample
11.  */
12. object FaultHandlingDocSample extends App {
13.   import Worker._
14.
15.   val config = ConfigFactory.parseString("""
16.     akka.loglevel = "DEBUG"
17.     akka.actor.debug {
18.       receive = on
```

```
19.       lifecycle = on
20.     }
21.     """)
22.
23.   val system = ActorSystem("FaultToleranceSample", config)
24.   val worker = system.actorOf(Props[Worker], name = "worker")
25.   val listener = system.actorOf(Props[Listener], name = "listener")
26.   // start the work and listen on progress
27.   // note that the listener is used as sender of the tell,
28.   // i.e. it will receive replies from the worker
29.   worker.tell(Start, sender = listener)
30. }
31.
32. /**
33.  * Listens on progress from the worker and shuts down the system
    when enough
34.  * work has been done.
35.  */
36. class Listener extends Actor with ActorLogging {
37.   import Worker._
38.   // If we don't get any progress within 15 seconds then the
    service is unavailable
39.   context.setReceiveTimeout(15 seconds)
40.
41.   def receive = {
42.     case Progress(percent) =>
43.       log.info("Current progress: {} %", percent)
44.       if (percent >= 100.0) {
45.         log.info("That's all, shutting down")
46.         context.system.shutdown()
47.       }
48.
49.     case ReceiveTimeout =>
50.       // No progress within 15 seconds, ServiceUnavailable
51.       log.error("Shutting down due to unavailable service")
52.       context.system.shutdown()
53.   }
54. }
```

```
55.
56.  object Worker {
57.    case object Start
58.    case object Do
59.    case class Progress(percent: Double)
60.  }
61.
62.  /**
63.   * Worker performs some work when it receives the `Start` message.
64.   * It will continuously notify the sender of the `Start` message
65.   * of current ``Progress``. The `Worker` supervise the
     `CounterService`.
66.   */
67.  class Worker extends Actor with ActorLogging {
68.    import Worker._
69.    import CounterService._
70.    implicit val askTimeout = Timeout(5 seconds)
71.
72.    // Stop the CounterService child if it throws ServiceUnavailable
73.    override val supervisorStrategy = OneForOneStrategy() {
74.      case _: CounterService.ServiceUnavailable => Stop
75.    }
76.
77.    // The sender of the initial Start message will continuously be
     notified
78.    // about progress
79.    var progressListener: Option[ActorRef] = None
80.    val counterService = context.actorOf(Props[CounterService], name
     = "counter")
81.    val totalCount = 51
82.    import context.dispatcher // Use this Actors' Dispatcher as
     ExecutionContext
83.
84.    def receive = LoggingReceive {
85.      case Start if progressListener.isEmpty =>
86.        progressListener = Some(sender)
87.        context.system.scheduler.schedule(Duration.Zero, 1 second,
     self, Do)
```

```
88.
89.     case Do =>
90.       counterService ! Increment(1)
91.       counterService ! Increment(1)
92.       counterService ! Increment(1)
93.
94.       // Send current progress to the initial sender
95.       counterService ? GetCurrentCount map {
96.         case CurrentCount(_, count) => Progress(100.0 * count /
    totalCount)
97.       } pipeTo progressListener.get
98.   }
99. }
100.
101. object CounterService {
102.   case class Increment(n: Int)
103.   case object GetCurrentCount
104.   case class CurrentCount(key: String, count: Long)
105.   class ServiceUnavailable(msg: String) extends
    RuntimeException(msg)
106.
107.   private case object Reconnect
108. }
109.
110. /**
111.  * Adds the value received in `Increment` message to a persistent
112.  * counter. Replies with `CurrentCount` when it is asked for
    `CurrentCount`.
113.  * `CounterService` supervise `Storage` and `Counter`.
114.  */
115. class CounterService extends Actor {
116.   import CounterService._
117.   import Counter._
118.   import Storage._
119.
120.   // Restart the storage child when StorageException is thrown.
121.   // After 3 restarts within 5 seconds it will be stopped.
122.   override val supervisorStrategy =
```

```
    OneForOneStrategy(maxNrOfRetries = 3,
123.     withinTimeRange = 5 seconds) {
124.       case _: Storage.StorageException => Restart
125.     }
126.
127.   val key = self.path.name
128.   var storage: Option[ActorRef] = None
129.   var counter: Option[ActorRef] = None
130.   var backlog = IndexedSeq.empty[(ActorRef, Any)]
131.   val MaxBacklog = 10000
132.
133.   import context.dispatcher // Use this Actors' Dispatcher as
    ExecutionContext
134.
135.   override def preStart() {
136.     initStorage()
137.   }
138.
139.   /**
140.    * The child storage is restarted in case of failure, but after 3
    restarts,
141.    * and still failing it will be stopped. Better to back-off than
    continuously
142.    * failing. When it has been stopped we will schedule a Reconnect
    after a delay.
143.    * Watch the child so we receive Terminated message when it has
    been terminated.
144.    */
145.   def initStorage() {
146.     storage = Some(context.watch(context.actorOf(Props[Storage],
    name = "storage")))
147.     // Tell the counter, if any, to use the new storage
148.     counter foreach { _ ! UseStorage(storage) }
149.     // We need the initial value to be able to operate
150.     storage.get ! Get(key)
151.   }
152.
153.   def receive = LoggingReceive {
```

```
154.
155.     case Entry(k, v) if k == key && counter == None =>
156.       // Reply from Storage of the initial value, now we can create
     the Counter
157.       val c = context.actorOf(Props(classOf[Counter], key, v))
158.       counter = Some(c)
159.       // Tell the counter to use current storage
160.       c ! UseStorage(storage)
161.       // and send the buffered backlog to the counter
162.       for ((replyTo, msg) <- backlog) c.tell(msg, sender = replyTo)
163.       backlog = IndexedSeq.empty
164.
165.     case msg @ Increment(n)    => forwardOrPlaceInBacklog(msg)
166.
167.     case msg @ GetCurrentCount => forwardOrPlaceInBacklog(msg)
168.
169.     case Terminated(actorRef) if Some(actorRef) == storage =>
170.       // After 3 restarts the storage child is stopped.
171.       // We receive Terminated because we watch the child, see
     initStorage.
172.       storage = None
173.       // Tell the counter that there is no storage for the moment
174.       counter foreach { _ ! UseStorage(None) }
175.       // Try to re-establish storage after while
176.       context.system.scheduler.scheduleOnce(10 seconds, self,
     Reconnect)
177.
178.     case Reconnect =>
179.       // Re-establish storage after the scheduled delay
180.       initStorage()
181.   }
182.
183.   def forwardOrPlaceInBacklog(msg: Any) {
184.     // We need the initial value from storage before we can start
     delegate to
185.     // the counter. Before that we place the messages in a backlog,
     to be sent
186.     // to the counter when it is initialized.
```

```
187.      counter match {
188.        case Some(c) => c forward msg
189.        case None =>
190.          if (backlog.size >= MaxBacklog)
191.            throw new ServiceUnavailable(
192.              "CounterService not available, lack of initial value")
193.          backlog :+= (sender() -> msg)
194.      }
195.    }
196.
197.  }
198.
199.  object Counter {
200.    case class UseStorage(storage: Option[ActorRef])
201.  }
202.
203.  /**
204.   * The in memory count variable that will send current
205.   * value to the `Storage`, if there is any storage
206.   * available at the moment.
207.   */
208.  class Counter(key: String, initialValue: Long) extends Actor {
209.    import Counter._
210.    import CounterService._
211.    import Storage._
212.
213.    var count = initialValue
214.    var storage: Option[ActorRef] = None
215.
216.    def receive = LoggingReceive {
217.      case UseStorage(s) =>
218.        storage = s
219.        storeCount()
220.
221.      case Increment(n) =>
222.        count += n
223.        storeCount()
224.
```

```
225.      case GetCurrentCount =>
226.        sender() ! CurrentCount(key, count)
227.
228.    }
229.
230.    def storeCount() {
231.      // Delegate dangerous work, to protect our valuable state.
232.      // We can continue without storage.
233.      storage foreach { _ ! Store(Entry(key, count)) }
234.    }
235.
236.  }
237.
238.  object Storage {
239.    case class Store(entry: Entry)
240.    case class Get(key: String)
241.    case class Entry(key: String, value: Long)
242.    class StorageException(msg: String) extends RuntimeException(msg)
243.  }
244.
245.  /**
246.   * Saves key/value pairs to persistent storage when receiving
      `Store` message.
247.   * Replies with current value when receiving `Get` message.
248.   * Will throw StorageException if the underlying data store is out
      of order.
249.   */
250.  class Storage extends Actor {
251.    import Storage._
252.
253.    val db = DummyDB
254.
255.    def receive = LoggingReceive {
256.      case Store(Entry(key, count)) => db.save(key, count)
257.      case Get(key)                 => sender() ! Entry(key,
      db.load(key).getOrElse(0L))
258.    }
259.  }
```

```
260.
261.  object DummyDB {
262.    import Storage.StorageException
263.    private var db = Map[String, Long]()
264.
265.    @throws(classOf[StorageException])
266.    def save(key: String, value: Long): Unit = synchronized {
267.      if (11 <= value && value <= 14)
268.        throw new StorageException("Simulated store failure " +
      value)
269.      db += (key -> value)
270.    }
271.
272.    @throws(classOf[StorageException])
273.    def load(key: String): Option[Long] = synchronized {
274.      db.get(key)
275.    }
276.  }
```

调度器

调度器

Akka `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` 实现也同时也是一个 `ExecutionContext` ，这意味着它们可以用来执行任何代码，例如 Future(Scala)。

缺省派发器

在没有为Actor作配置的情况下，每一个 `ActorSystem` 将有一个缺省的派发器。该缺省派发器可以被配置，默认是使用指定的 `default-executor` 的一个 `Dispatcher` 。如果一个ActorSystem是使用传入的ExecutionContext创建的，则该ExecutionContext将被用作所有派发器的默认执行器（“executor”）。如果没有给定ExecutionContext，则会回退使用 `akka.actor.default-dispatcher.default-executor.fallback` 指定的执行器。缺省情况下是使用“fork-join-executor”，它在大多数情况下拥有非常好的性能。

查找一个派发器

派发器实现了 `ExecutionContext` 接口，因此可以用来运行 `Future` 调用

等。

```
// for use with Futures, Scheduler, etc.
implicit val executionContext = system.dispatchers.lookup("my-dispatcher")
```

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，首先是配置派发器：

```
my-dispatcher {
  # Dispatcher 是基于事件的派发器的名称
  type = Dispatcher
  # 使用何种ExecutionService
  executor = "fork-join-executor"
  # 配置 fork join 池
  fork-join-executor {
    # 容纳基于因子的并行数量的线程数下限
    parallelism-min = 2
    # 并行数（线程）... ceil(可用CPU数*因子)
    parallelism-factor = 2.0
    # 容纳基于因子的并行数量的线程数上限
    parallelism-max = 10
  }
  # Throughput 定义了线程切换到下一个actor之前处理的消息数上限
  # 设置成1表示尽可能公平。
  throughput = 100
}
```

以下是另一个使用“thread-pool-executor”的例子：

```
my-thread-pool-dispatcher {
  # Dispatcher是基于事件的派发器的名称
  type = Dispatcher
  # 使用何种 ExecutionService
```

```
  executor = "thread-pool-executor"
  # 配置线程池
  thread-pool-executor {
    # 容纳基于因子的内核数的线程数下限
    core-pool-size-min = 2
    # 内核线程数 .. ceil(可用CPU数＊倍数)
    core-pool-size-factor = 2.0
    # 容纳基于倍数的并行数量的线程数上限
    core-pool-size-max = 10
  }
  # Throughput 定义了线程切换到下一个actor之前处理的消息数上限
  # 设置成1表示尽可能公平.
  throughput = 100
}
```

更多选项，请参阅配置的缺省派发器(default-dispatcher)一节。

然后可以像往常一样创建actor并在部署配置中定义调度器。

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

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

部署配置的替代方法是在代码中定义调度器。如果你在部署配置中定义 `dispatcher` ，则实际使用的将是此值，而不是以编程方式提供的参数。

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

> 注意
>
> 你在 `withDispatcher` 中指定的调度器，和在部署文件指定的 `dispatcher` 设置其实是配置中的一个路径. 所以在这个例子中它位于配置的顶层，但你可以例如把它放在下面的层次，用“.”来代表子层次，象这样： `"foo.bar.my-dispatcher"`

派发器的类型

一共有4种类型的消息派发器：

- Dispatcher
    - 这是基于事件的调度器，将一组actor绑定到线程池。如果未指定派发器，则它将被用作默认调度器。
    - 可共享性：无限制
    - 邮箱：任意，为每一个Actor创建一个
    - 使用场景：缺省派发器，Bulkheading
    - 底层驱动：`java.util.concurrent.ExecutorService` 通过”executor”指定，可使用”fork-join-executor”、”thread-pool-executor”或一个 `akka.dispatcher.ExecutorServiceConfigurator` 的限定
- PinnedDispatcher
    - 这个调度器为每一个使用它的actor分配一个独立的线程；即每个actor会有其独有的只有一个线程的线程池。
    - 可共享性：无
    - 邮箱：任意，为每个Actor创建一个
    - 使用场景：Bulkheading
    - 底层驱动：任意 `akka.dispatch.ThreadPoolExecutorConfigurator` 缺省为一个“thread-pool-executor”
- BalancingDispatcher

    - 这是基于事件的调度器，将尝试从繁忙的actor重新分配工作到空闲的actor。
    - 所有actor共享单个邮箱，并从中获取他们的消息。
    - 这里假定所有使用此调度器的actor都可以处理发送到其中一个actor的所有的消息；即actor属于actor池，并且对客户端来说没有保证来决定哪个actor实例实际上处理某个特定的消息。
    - 可共享性：仅对同一类型的Actor共享
    - 邮箱：任意，为所有的Actor创建一个
    - 使用场景：Work-sharing
    - 底层驱动：`java.util.concurrent.ExecutorService`
      通过”executor”指定，可使用 “fork-join-executor”, “thread-pool-executor”
      或`akka.dispatcher.ExecutorServiceConfigurator`的限定
    - 请注意不能将`BalancingDispatcher`用作一个路由器调度程序。（但是你可以把它用作**Routees**）

- CallingThreadDispatcher

    - 该调度器只在当前线程上调用。该调度器不会创建任何新的线程，但它可以被相同的actor从不同的线程同时使用。更多信息和限制，请参阅CallingThreadDispatcher。
    - 可共享性：无限制
    - 邮箱：任意，每Actor每线程创建一个（需要时）
    - 使用场景：测试
    - 底层使用：调用的线程 (duh)

更多 dispatcher 配置的例子

配置一个`PinnedDispatcher`：

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

然后使用它：

```
val myActor =
  context.actorOf(Props[MyActor].withDispatcher("my-pinned-dispatcher"), "myactor2")
```

注意 `thread-pool-executor` 配置按上述 `my-thread-pool-dispatcher` 的调度程序例子并不适用。这是因为每个actor使用 `PinnedDispatcher` 时，会有其自己的线程池，并且该池将只有一个线程。

注意没有保证随时间推移，相同的线程会被使用，由于核心池超时被 `PinnedDispatcher` 用于在空闲actor的情况下降低资源使用率。总是使用相同的线程需要在 `PinnedDispatcher` 的配置中添加 `thread-pool-executor.allow-core-timeout=off` 。

邮箱

邮箱

一个Akka `Mailbox`保存发往某个Actor的消息。通常每个Actor都拥有自己的邮箱，但也有例外，例如使用`BalancingPool`的所有路由子(routee)共享同一个邮箱实例。

邮箱选择

为actor指定一个消息队列类型

为某个特定类型的actor指定一个特定类型的消息队列是有可能的，只要通过actor扩展`RequiresMessageQueue`参数化特质即可。下面是一个示例：

```
1.  import akka.dispatch.RequiresMessageQueue
2.  import akka.dispatch.BoundedMessageQueueSemantics
3.
```

```
class MyBoundedActor extends MyActor
  with RequiresMessageQueue[BoundedMessageQueueSemantics]
```

`RequiresMessageQueue`特质的类型参数需要映射到配置中的邮箱，像这样：

```
bounded-mailbox {
  mailbox-type = "akka.dispatch.BoundedMailbox"
  mailbox-capacity = 1000
  mailbox-push-timeout-time = 10s
}

akka.actor.mailbox.requirements {
  "akka.dispatch.BoundedMessageQueueSemantics" = bounded-mailbox
}
```

现在每当你创建一个类型为`MyBoundedActor`的actor时，它会尝试得到一个有界的邮箱。如果actor在部署中具有一个不同的邮箱配置—直接地，或是通过一个指定的邮箱类型的调度器—那么它将覆盖此映射。

> 注意
>
> 为*actor*创建的邮箱队列类型，会和特质中要求的类型进行检查，如果队列没有实现要求的类型，则*actor*创建会失败。

为调度器指定一个消息队列类型

调度器也可能对actor使用的邮箱类型进行限制。一个例子是BalancingDispatcher，它要求消息队列对多个并发的消费者是线程安全的。该约束是在配置的调度器一节中像下面这样设定：

```
  my-dispatcher {
    mailbox-requirement = org.example.MyInterface
  }
```

给定的约束是要求指定的类或者接口，必须是消息队列的实现的超类。

如果出现冲突—例如如果actor要求的邮箱类型不能满足要求—则actor创建会失败。

如何选择邮箱类型

当一个actor创建时， `ActorRefProvider` 首先确定将执行它的调度器。然后，邮箱确定如下：

1. 如果actor的部署配置节包含 `mailbox` 键，则其描述邮箱类型将被使用。
2. 如果actor的 `Props` 包含邮箱选择—即它调用了 `withMailbox` —则其描述邮箱类型将被使用。
3. 如果调度器的配置节包含 `mailbox-type` 键，则该节内容将用于配置邮箱类型。
4. 如果该actor需要邮箱类型，如上文所述，然后该约束的映射将用于确定使用的邮箱类型；如果不能满足调度器的约束—如果有的话—将继续替换尝试。
5. 如果调度器需要一个邮箱类型，如上文所述，则该约束的映射将被用来确定要使用的邮箱类型。
6. 将使用默认邮箱 `akka.actor.default-mailbox` 。

默认邮箱

当未如上所述，指定使用邮箱时，将使用默认邮箱。默认情况它是无界的邮箱，由 `java.util.concurrent.ConcurrentLinkedQueue` 实现。

`SingleConsumerOnlyUnboundedMailbox` 是一个更有效率的邮箱，而且它也可以用作默认邮箱，但它不能与 BalancingDispatcher一起使用。

`SingleConsumerOnlyUnboundedMailbox` 作为默认邮箱的配置：

```
1.  akka.actor.default-mailbox {
2.    mailbox-type =
```

```
    "akka.dispatch.SingleConsumerOnlyUnboundedMailbox"
3.   }
```

哪项配置会传递给邮箱类型

每个邮箱类型由扩展 `MailboxType` 并带两个构造函数参数的类实现：一个 `ActorSystem.Settings` 对象和一个 `Config` 对象。后者通过actor系统配置的指定配置节被计算出来，重写其指定邮箱类型的配置路径的 `id` 键，并添加一个到默认邮箱配置节的回退。

内置实现

Akka自带有一些内置的邮箱实现：

- UnboundedMailbox
  - 默认邮箱
  - 底层是一个`java.util.concurrent.ConcurrentLinkedQueue`
  - 阻塞：否
  - 有界：否
  - 配置名称：”unbounded” 或 “akka.dispatch.UnboundedMailbox”
- SingleConsumerOnlyUnboundedMailbox
  - 底层是一个非常高效的多生产者单消费者队列，不能被用于BalancingDispatcher
  - 阻塞：否
  - 有界：否
  - 配置名称：”akka.dispatch.SingleConsumerOnlyUnboundedMailbox”

- BoundedMailbox
  - 底层是一个`java.util.concurrent.LinkedBlockingQueue`
  - 阻塞：是
  - 有界：是
  - 配置名称：”bounded” 或 “akka.dispatch.BoundedMailbox”
- UnboundedPriorityMailbox
  - 底层是一个`java.util.concurrent.PriorityBlockingQueue`
  - 阻塞：是
  - 有界：否
  - 配置名称：”akka.dispatch.UnboundedPriorityMailbox”
- BoundedPriorityMailbox
  - 底层是一个 `java.util.PriorityBlockingQueue`包装为`akka.util.BoundedBlockingQueue`
  - 阻塞：是
  - 有界：是
  - 配置名称：”akka.dispatch.BoundedPriorityMailbox”

邮箱配置示例

如何创建一个PriorityMailbox:

```
1.  import akka.dispatch.PriorityGenerator
2.  import akka.dispatch.UnboundedPriorityMailbox
3.  import com.typesafe.config.Config
4.
```

```
// We inherit, in this case, from UnboundedPriorityMailbox
// and seed it with the priority generator
class MyPrioMailbox(settings: ActorSystem.Settings, config: Config)
  extends UnboundedPriorityMailbox(
    // Create a new PriorityGenerator, lower prio means more important
    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
    })
```

并添加到配置文件：

```
prio-dispatcher {
  mailbox-type = "docs.dispatcher.DispatcherDocSpec$MyPrioMailbox"
  //Other dispatcher configuration goes here
}
```

以下示例演示如何使用它：

```
// We create a new Actor that just prints out what it processes
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"))

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

也可以像这样直接配置邮箱类型：

```
prio-mailbox {
  mailbox-type = "docs.dispatcher.DispatcherDocSpec$MyPrioMailbox"
  //Other mailbox configuration goes here
}

akka.actor.deployment {
  /priomailboxactor {
    mailbox = prio-mailbox
  }
}
```

然后可以在部署环境像这样使用它：

```
import akka.actor.Props
val myActor = context.actorOf(Props[MyActor], "priomailboxactor")
```

或像这样在代码中：

```
import akka.actor.Props
val myActor = context.actorOf(Props[MyActor].withMailbox("prio-mailbox"))
```

创建自己的邮箱类型

例子比文字更有说服力：

```
import akka.actor.ActorRef
import akka.actor.ActorSystem
import akka.dispatch.Envelope
import akka.dispatch.MailboxType
import akka.dispatch.MessageQueue
import akka.dispatch.ProducesMessageQueue
import com.typesafe.config.Config
import java.util.concurrent.ConcurrentLinkedQueue
import scala.Option

// Marker trait used for mailbox requirements mapping
trait MyUnboundedMessageQueueSemantics

object MyUnboundedMailbox {
  // This is the MessageQueue implementation
  class MyMessageQueue extends MessageQueue
    with MyUnboundedMessageQueueSemantics {

    private final val queue = new ConcurrentLinkedQueue[Envelope]()

    // these should be implemented; queue used as example
    def enqueue(receiver: ActorRef, handle: Envelope): Unit =
      queue.offer(handle)
    def dequeue(): Envelope = queue.poll()
```

```
25.     def numberOfMessages: Int = queue.size
26.     def hasMessages: Boolean = !queue.isEmpty
27.     def cleanUp(owner: ActorRef, deadLetters: MessageQueue) {
28.       while (hasMessages) {
29.         deadLetters.enqueue(owner, dequeue())
30.       }
31.     }
32.   }
33. }
34.
35. // This is the Mailbox implementation
36. class MyUnboundedMailbox extends MailboxType
37.   with ProducesMessageQueue[MyUnboundedMailbox.MyMessageQueue] {
38.
39.   import MyUnboundedMailbox._
40.
41.   // This constructor signature must exist, it will be called by
    Akka
42.   def this(settings: ActorSystem.Settings, config: Config) = {
43.     // put your initialization code here
44.     this()
45.   }
46.
47.   // The create method is called to create the MessageQueue
48.   final override def create(owner: Option[ActorRef],
49.                             system: Option[ActorSystem]):
    MessageQueue =
50.     new MyMessageQueue()
51. }
```

然后在派发器配置中，或邮箱配置中以你定义的邮箱类型的全称作为“mailbox-type”的值。

> 注意
>
> 一定要定义一个以 `akka.actor.ActorSystem.Settings` 和 `com.typesafe.config.Config` 为参数的构造函数，该构造函数将以反射的方式被调用来创建你的邮箱类型。第二个传入的配置参数是配置文件中使用这个邮箱类型的派发器或邮箱的描述；该邮箱类型会为每一个使用它的派发

> 器或邮箱创建一个实例。

此外可以作为派发器的约束这样使用邮箱：

```
custom-dispatcher {
  mailbox-requirement =
  "docs.dispatcher.MyUnboundedJMessageQueueSemantics"
}

akka.actor.mailbox.requirements {
  "docs.dispatcher.MyUnboundedJMessageQueueSemantics" =
  custom-dispatcher-mailbox
}

custom-dispatcher-mailbox {
  mailbox-type = "docs.dispatcher.MyUnboundedJMailbox"
}
```

或通过在你的actor类上定义约束，像这样：

```
class MySpecialActor extends Actor
  with RequiresMessageQueue[MyUnboundedMessageQueueSemantics] {
  // ...
}
```

`system.actorOf` 的特殊语义

为了使 `system.actorOf` 同步和非阻塞，并且返回类型保持为 `ActorRef` （并且保持返回的ref是完全函数式的语义），这种情况下会进行特殊处理。在幕后，一种空心的actor引用会被构造，然后发送到系统的监管者，该监管者实际创建actor和它的上下文，并把它们传到引用内部。直到这些发生以后，发送到该 `ActorRef` 的消息会被本地排队，而只有当填入真实信息后，它们才会被传入真实的邮箱，因此，

```
1. val props: Props = ...
2. // this actor uses MyCustomMailbox, which is assumed to be a
   singleton
3. system.actorOf(props.withDispatcher("myCustomMailbox")) ! "bang"
4. assert(MyCustomMailbox.instance.getLastEnqueuedMessage == "bang")
```

可能会失败；你将不得不留出一些时间来传递，然后重新尝试检查 `TestKit.awaitCond` 。

路由

- 自定义路由actor
- 配置调度器" level="3">配置调度器

路由

消息可以通过路由器发送，以便有效地将它们路由到目的actor，称为其*routee*。一个 `Router` 可以在actor内部或外部使用，并且你可以自己管理routee或使用有配置功能的自我包含的路由actor。

根据你应用程序的需求，可以使用不同的路由策略。Akka附带了几个有用的路由策略，开箱即用。但是正如将在这一章中看到地，你也可以创建自己的路由。

一个简单的路由器

下面的示例阐释如何使用 `Router` 和在actor内管理routee。

```
1.  import akka.routing.ActorRefRoutee
2.  import akka.routing.Router
3.  import akka.routing.RoundRobinRoutingLogic
4.
5.  class Master extends Actor {
6.    var router = {
7.      val routees = Vector.fill(5) {
8.        val r = context.actorOf(Props[Worker])
9.        context watch r
10.       ActorRefRoutee(r)
11.     }
12.     Router(RoundRobinRoutingLogic(), routees)
13.   }
14.
15.   def receive = {
16.     case w: Work =>
17.       router.route(w, sender())
18.     case Terminated(a) =>
```

```
19.         router = router.removeRoutee(a)
20.         val r = context.actorOf(Props[Worker])
21.         context watch r
22.         router = router.addRoutee(r)
23.       }
24.     }
```

我们创建一个 `Router` ，并指定当路由消息到routee时，它应该使用 `RoundRobinRoutingLogic` 。

Akka自带的路由逻辑如下：

- `akka.routing.RoundRobinRoutingLogic`
- `akka.routing.RandomRoutingLogic`
- `akka.routing.SmallestMailboxRoutingLogic`
- `akka.routing.BroadcastRoutingLogic`
- `akka.routing.ScatterGatherFirstCompletedRoutingLogic`
- `akka.routing.TailChoppingRoutingLogic`
- `akka.routing.ConsistentHashingRoutingLogic`

我们像在 `ActorRefRoutee` 包装下创建普通子actor一样创建routee。我们监控routee从而能够在它们被终止的情况下取代他们。

通过路由器发送消息是用 `route` 方法完成的，像上面例子中的 `Work` 消息一样。

`Router` 是不可变的，而 `RoutingLogic` 是线程安全的；意味着他们也可以在actor外部使用。

> 注意
>
> 一般情况下，任何发送到路由器的消息将被向前发送到它的*routee*，但有一个例外。特别地广播消息将发送到路由器下所有的*routee*

一个路由actor

一个路由器也可以被创建为一个自包含的actor，来管理routee，载入路由逻辑和其他配置设置。

这种类型的路由actor有两种不同的模式：

- 池——路由器创建routee作为子actor，并在该子actor终止时将它从路由器中移除。
- 群组——routee actor在路由器外部创建，而路由器将通过使用actor选择将消息发送到指定路径，而不监控其终止。

路由actor可以通过在配置中或以编程方式被定义。虽然路由actor可以在配置文件中定义，但仍然必须以编程方式创建，即你不能只通过外部配置创建路由器。如果你在配置文件中定义路由actor，则实际将使用这些设置，而不是以编程方式提供的参数。

你可以通过路由actor向routee 发送消息，就像向普通actor发消息一样，即通过其 `ActorRef` 。路由actor转发消息给routee时不会更改原始发件人。当routee答复路由消息时，回复将发送到原始发件人，而不是路由actor。

> 注意
>
> 一般地，任何发送到路由器的消息将被向前发送到它的*routee*，但也有几个例外。这些记录在下面特殊处理消息一节中。

池

下面的代码和配置片段展示了如何创建一个将消息转发给五个 `Worker` routee的轮循(`round-robin`)路由器。Routees 将被创建为路由器的子actor。

```
1.  akka.actor.deployment {
2.    /parent/router1 {
3.      router = round-robin-pool
4.      nr-of-instances = 5
```

```
  }
}
```

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

这里是相同的例子，但路由配置是以编程方式而不是从配置获取。

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

远程部署的Routee

除了能够创建本地actor作为routee，你可以指示路由器来部署其子actor到一系列远程主机上。Routee将以轮循方式被部署。若要远程部署routee，将路由配置包在一个 `RemoteRouterConfig` 下，附加要部署到的节点的远程地址。远程部署要求 `akka-remote` 模块被包含在类路径中。

```
import akka.actor.{ Address, AddressFromURIString }
import akka.remote.routing.RemoteRouterConfig
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]))
```

发送者

默认情况下，当一个routee发送一条消息，它将隐式地设置自身为发送者。

```
sender() ! x // replies will go to this actor
```

然而，通常为routee将路由器设置为发送者更有用。例如，你可能想要将路由器设置为发件人，如果你想要隐藏在路由器后面routee的细节。下面的代码片段演示如何设置父路由器作为发送者。

```
1. sender().tell("reply", context.parent) // replies will go back to
   parent
2. sender().!("reply")(context.parent) // alternative syntax (beware
   of the parens!)
```

监管

由池路由器创建的routee将成为路由器的孩子。因此，路由器也是子actor的监管者。

可以用该池的 `supervisorStrategy` 属性配置路由器actor的监管策略。如果没有提供配置，路由器的默认策略是"总是上溯"。这意味着错误都会向上传递给路由器的监管者进行处理。路由器的监管者将决定对任何错误该做什么。

请注意路由器的监管者将把错误视为路由器本身的错误。因此一个指令，用于停止或重新启动将导致路由器本身以停止或重新启动。此路由器，相应地，将导致它的孩子停止并重新启动。

应该提到的是路由器重新启动行为已被重写，以便重新启动时，仍重新创建这些孩子，并会在池中保留相同数量的actor。

这意味着如果你还没有指定路由器或其父节点中的 `supervisorStrategy` ，routee的失败会上升到路由器，并将在默认情况下重新启动路由器，从而将重新启动所有的routee（它使用上升，并在重新启动过程中不停止routee）。原因是为了使类似在子actor定义中添加 `.withRouter` 这样的默认行为，不会更改应用于子actor的监管策略。你可以在定义路由器时指定策略来避免低效。

设置策略是很容易完成的：

```
1.  val escalator = OneForOneStrategy() {
2.    case e ⇒ testActor ! e; SupervisorStrategy.Escalate
3.  }
4.  val router = system.actorOf(RoundRobinPool(1, supervisorStrategy =
    escalator).props(
5.    routeeProps = Props[TestActor]))
```

> 注意
>
> 如果路由器池的子*actor*终止，池路由器不会自动产生一个新的*actor*。在池路由器所有子*actor*都终止的事件中，路由器将终止本身，除非它是一个动态的路由器，例如使用了大小调整。

群组

有时候，相比于由路由actor创建其routee，我们更希望单独创建routee，并提供路由器供其使用。你可以通过将 routee路径传递给路由器的配置来实现。消息将通过 `ActorSelection` 发送到这些路径。

下面的示例演示如何通过提供三个routee actor的路径字符串来创建一个路由器。

```
1.  akka.actor.deployment {
2.    /parent/router3 {
3.      router = round-robin-group
4.      routees.paths = ["/user/workers/w1", "/user/workers/w2",
    "/user/workers/w3"]
5.    }
6.  }
```

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

这里是相同的例子，但路由配置是以编程方式而不是从配置获取。

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

Routee actor将在路由器外部被创建:

```
1.  system.actorOf(Props[Workers], "workers")
```

```
1.  class Workers extends Actor {
2.    context.actorOf(Props[Worker], name = "w1")
3.    context.actorOf(Props[Worker], name = "w2")
4.    context.actorOf(Props[Worker], name = "w3")
5.    // ...
```

在路径中可能包含为actor在远程主机上运行的协议和地址信息。远程部署要求 `akka-remote` 模块被包含在类路径中。

路由器使用

在本节中，我们将描述如何创建不同类型的路由actor。

本节中的路由actor将由名为 `parent` 的顶级actor创建。请注意在配置中的部署路径以 `/parent/` 开头，并紧接着路由actor的名字。

```
1.  system.actorOf(Props[Parent], "parent")
```

RoundRobinPool 和 RoundRobinGroup

对其routee使用轮循机制(round-robin)轮询。

在配置中定义的RoundRobinPool:

```
1.  akka.actor.deployment {
2.    /parent/router1 {
3.      router = round-robin-pool
4.      nr-of-instances = 5
```

```
  }
}
```

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

在代码中定义的RoundRobinPool:

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

在配置中定义的RoundRobinGroup:

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

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

在代码中定义的RoundRobinGroup:

```
val paths = List("/user/workers/w1", "/user/workers/w2",
"/user/workers/w3")
val router4: ActorRef =
  context.actorOf(RoundRobinGroup(paths).props(), "router4")
```

RandomPool 和 RandomGroup

该路由器类型会对每一条消息随机选择其routee。

在配置中定义的RandomPool:

```
akka.actor.deployment {
  /parent/router5 {
    router = random-pool
    nr-of-instances = 5
  }
}
```

```
val router5: ActorRef =
  context.actorOf(FromConfig.props(Props[Worker]), "router5")
```

在代码中定义的RandomPool:

```
val router6: ActorRef =
  context.actorOf(RandomPool(5).props(Props[Worker]), "router6")
```

在配置中定义的RandomGroup:

```
akka.actor.deployment {
  /parent/router7 {
    router = random-group
    routees.paths = ["/user/workers/w1", "/user/workers/w2",
"/user/workers/w3"]
  }
}
```

```
val router7: ActorRef =
  context.actorOf(FromConfig.props(), "router7")
```

在代码中定义的RandomGroup:

```
val paths = List("/user/workers/w1", "/user/workers/w2",
"/user/workers/w3")
val router8: ActorRef =
```

```
3.    context.actorOf(RandomGroup(paths).props(), "router8")
```

BalancingPool

将尝试重新从繁忙routee分配任务到空闲routee的路由器。所有routee都共享同一个邮箱。

在配置中定义的BalancingPool:

```
1.  akka.actor.deployment {
2.    /parent/router9 {
3.      router = balancing-pool
4.      nr-of-instances = 5
5.    }
6.  }
```

```
1.  val router9: ActorRef =
2.    context.actorOf(FromConfig.props(Props[Worker]), "router9")
```

在代码中定义的BalancingPool:

```
1.  val router10: ActorRef =
2.    context.actorOf(BalancingPool(5).props(Props[Worker]),
    "router10")
```

平衡调度器的额外配置，被池使用，可以通过路由部署配置的 `pool-dispatcher` 节进行设定。

```
1.  akka.actor.deployment {
2.    /parent/router9b {
3.      router = balancing-pool
4.      nr-of-instances = 5
5.      pool-dispatcher {
6.        attempt-teamwork = off
7.      }
8.    }
```

```
9. }
```

对BalancingPool没有群组变体。

SmallestMailboxPool

试图向邮箱中有最少消息的非暂停子routee发送消息的路由器。按此顺序进行选择：

- 挑选有空邮箱的空闲routee（即没有处理消息）
- 选择任一空邮箱routee
- 选择邮箱中有最少挂起消息的routee
- 选择任一远程routee，远程actor考虑优先级最低，因为其邮箱大小未知

在配置中定义的SmallestMailboxPool:

```
1. akka.actor.deployment {
2.   /parent/router11 {
3.     router = smallest-mailbox-pool
4.     nr-of-instances = 5
5.   }
6. }
```

```
1. val router11: ActorRef =
2.   context.actorOf(FromConfig.props(Props[Worker]), "router11")
```

在代码中定义的SmallestMailboxPool:

```
1. val router12: ActorRef =
2.   context.actorOf(SmallestMailboxPool(5).props(Props[Worker]),
   "router12")
```

SmallestMailboxPool没有群组变体，因为邮箱大小和actor的内

部调度状态从routee的路径看实际上是不可用的。

BroadcastPool 和 BroadcastGroup

广播的路由器将接收到的消息转发到它所有的routee。

在配置中定义的BroadcastPool:

```
akka.actor.deployment {
  /parent/router13 {
    router = broadcast-pool
    nr-of-instances = 5
  }
}
```

```
val router13: ActorRef =
  context.actorOf(FromConfig.props(Props[Worker]), "router13")
```

在代码中定义的BroadcastPool:

```
val router14: ActorRef =
  context.actorOf(BroadcastPool(5).props(Props[Worker]),
"router14")
```

在配置中定义的BroadcastGroup:

```
akka.actor.deployment {
  /parent/router15 {
    router = broadcast-group
    routees.paths = ["/user/workers/w1", "/user/workers/w2",
"/user/workers/w3"]
  }
}
```

```
val router15: ActorRef =
  context.actorOf(FromConfig.props(), "router15")
```

在代码中定义的BroadcastGroup:

```
1. val paths = List("/user/workers/w1", "/user/workers/w2",
   "/user/workers/w3")
2. val router16: ActorRef =
3.   context.actorOf(BroadcastGroup(paths).props(), "router16")
```

> 注意
>
> *Broadcast*路由器总是向其*routee*广播每一条消息。如果你不想播出每条消息，则你可以使用非广播路由器并使用所需的广播消息。

ScatterGatherFirstCompletedPool 和 ScatterGatherFirstCompletedGroup

ScatterGatherFirstCompletedRouter将会把消息发送到它所有的routee。然后它等待直到收到第一个答复。该结果将发送回原始发送者。其他的答复将被丢弃。

在配置的时间内，它期待至少一个答复，否则它将回复一个包含 `akka.pattern.AskTimeoutException` 的 `akka.actor.Status.Failure` 。

在配置中定义的ScatterGatherFirstCompletedPool:

```
1. akka.actor.deployment {
2.   /parent/router17 {
3.     router = scatter-gather-pool
4.     nr-of-instances = 5
5.     within = 10 seconds
6.   }
7. }
```

```
1. val router17: ActorRef =
2.   context.actorOf(FromConfig.props(Props[Worker]), "router17")
```

在代码中定义的ScatterGatherFirstCompletedPool:

```
1. val router18: ActorRef =
2.   context.actorOf(ScatterGatherFirstCompletedPool(5, within =
   10.seconds).
3.     props(Props[Worker]), "router18")
```

在配置中定义的ScatterGatherFirstCompletedGroup:

```
1. akka.actor.deployment {
2.   /parent/router19 {
3.     router = scatter-gather-group
4.     routees.paths = ["/user/workers/w1", "/user/workers/w2",
   "/user/workers/w3"]
5.     within = 10 seconds
6.   }
7. }
```

```
1. val router19: ActorRef =
2.   context.actorOf(FromConfig.props(), "router19")
```

在代码中定义的ScatterGatherFirstCompletedGroup:

```
1. val paths = List("/user/workers/w1", "/user/workers/w2",
   "/user/workers/w3")
2. val router20: ActorRef =
3.   context.actorOf(ScatterGatherFirstCompletedGroup(paths,
4.     within = 10.seconds).props(), "router20")
```

TailChoppingPool 和 TailChoppingGroup

TailChoppingRouter 将首先发送消息到一个随机挑取的routee，短暂的延迟后发给第二个routee（从剩余的routee中随机挑选），以此类推。它等待第一个答复，并将它转回给原始发送者。其他答复将被丢弃。

此路由器的目标是通过查询到多个routee来减少延迟，假设其他的

actor之一仍可能比第一个actor更快响应。

Peter Bailis很好地在一篇博客文章中描述了这个优化： 做冗余的工作，以加快分布式查询。

在配置中定义的TailChoppingPool:

```
akka.actor.deployment {
  /parent/router21 {
    router = tail-chopping-pool
    nr-of-instances = 5
    within = 10 seconds
    tail-chopping-router.interval = 20 milliseconds
  }
}
```

```
val router21: ActorRef =
  context.actorOf(FromConfig.props(Props[Worker]), "router21")
```

在代码中定义的TailChoppingPool:

```
val router22: ActorRef =
  context.actorOf(TailChoppingPool(5, within = 10.seconds, interval = 20.millis).
    props(Props[Worker]), "router22")
```

在配置中定义的TailChoppingGroup:

```
akka.actor.deployment {
  /parent/router23 {
    router = tail-chopping-group
    routees.paths = ["/user/workers/w1", "/user/workers/w2", "/user/workers/w3"]
    within = 10 seconds
    tail-chopping-router.interval = 20 milliseconds
  }
```

```
8. }
```

```
1. val router23: ActorRef =
2.   context.actorOf(FromConfig.props(), "router23")
```

在代码中定义的TailChoppingGroup:

```
1. val paths = List("/user/workers/w1", "/user/workers/w2",
   "/user/workers/w3")
2. val router24: ActorRef =
3.   context.actorOf(TailChoppingGroup(paths,
4.     within = 10.seconds, interval = 20.millis).props(), "router24")
```

ConsistentHashingPool 和 ConsistentHashingGroup

ConsistentHashingPool基于已发送的消息使用一致性哈希(consistent hashing)选择routee。这篇文章给出了如何实现一致性哈希非常好的见解。

有三种方式来定义使用哪些数据作为一致的散列键。

- 你可以定义路由的`hashMapping`，将传入的消息映射到它们一致散列键。这使决策对发送者透明。
- 这些消息可能会实现`akka.routing.ConsistentHashingRouter.ConsistentHashable`。键是消息的一部分，并很方便地与消息定义一起定义。
- 消息可以被包装在一个`akka.routing.ConsistentHashingRouter.ConsistentHashableEnvelope`中，来定义哪些数据可以用来做一致性哈希。发送者知道要使用的键。

这些定义一致性哈希键的方法，可以同时对一个路由器在一起使用。`hashMapping`被第一个尝试。

代码示例：

```
import akka.actor.Actor
import akka.routing.ConsistentHashingRouter.ConsistentHashable

class Cache extends Actor {
  var cache = Map.empty[String, String]

  def receive = {
    case Entry(key, value) => cache += (key -> value)
    case Get(key)          => sender() ! cache.get(key)
    case Evict(key)        => cache -= key
  }
}

case class Evict(key: String)

case class Get(key: String) extends ConsistentHashable {
  override def consistentHashKey: Any = key
}

case class Entry(key: String, value: String)
```

```
import akka.actor.Props
import akka.routing.ConsistentHashingPool
import akka.routing.ConsistentHashingRouter.ConsistentHashMapping
import
akka.routing.ConsistentHashingRouter.ConsistentHashableEnvelope

def hashMapping: ConsistentHashMapping = {
  case Evict(key) => key
}

val cache: ActorRef =
  context.actorOf(ConsistentHashingPool(10, hashMapping =
hashMapping).
    props(Props[Cache]), name = "cache")
```

```
13.
14. cache ! ConsistentHashableEnvelope(
15.   message = Entry("hello", "HELLO"), hashKey = "hello")
16. cache ! ConsistentHashableEnvelope(
17.   message = Entry("hi", "HI"), hashKey = "hi")
18.
19. cache ! Get("hello")
20. expectMsg(Some("HELLO"))
21.
22. cache ! Get("hi")
23. expectMsg(Some("HI"))
24.
25. cache ! Evict("hi")
26. cache ! Get("hi")
27. expectMsg(None)
```

在上面的例子中可以看到 `Get` 消息自己实现了 `ConsistentHashable`，而 `Entry` 消息包裹在一个 `ConsistentHashableEnvelope` 中。 `Evict` 消息由 `hashMapping` 偏函数处理。

在配置中定义的ConsistentHashingPool:

```
1. akka.actor.deployment {
2.   /parent/router25 {
3.     router = consistent-hashing-pool
4.     nr-of-instances = 5
5.     virtual-nodes-factor = 10
6.   }
7. }
```

```
1. val router25: ActorRef =
2.   context.actorOf(FromConfig.props(Props[Worker]), "router25")
```

在代码中定义的ConsistentHashingPool:

```
1. val router26: ActorRef =
```

```
2.    context.actorOf(ConsistentHashingPool(5).props(Props[Worker]),
3.      "router26")
```

在配置中定义的ConsistentHashingGroup:

```
1.  akka.actor.deployment {
2.    /parent/router27 {
3.      router = consistent-hashing-group
4.      routees.paths = ["/user/workers/w1", "/user/workers/w2",
    "/user/workers/w3"]
5.      virtual-nodes-factor = 10
6.    }
7.  }
```

```
1.  val router27: ActorRef =
2.    context.actorOf(FromConfig.props(), "router27")
```

在代码中定义的ConsistentHashingGroup:

```
1.  val paths = List("/user/workers/w1", "/user/workers/w2",
    "/user/workers/w3")
2.  val router28: ActorRef =
3.    context.actorOf(ConsistentHashingGroup(paths).props(),
    "router28")
```

`virtual-nodes-factor`(虚拟节点因子)是每个routee的虚拟节点数，用来在一致性哈希节点环中使用，使分布更加均匀。

特殊处理的消息

发送到路由actor的大多数消息将根据路由器的路由逻辑进行转发。然而，有几种具有特殊行为的消息类型。

请注意这些特别的消息，除了 `Broadcast` 消息之外，只被自我包含的路由actor处理，而不是简单的路由器中描述的 `akka.routing.Router` 组

件。

广播消息

`Broadcast` 消息可用于向路由器的所有routee发送一条消息。当路由器接收一个 `Broadcast` 消息时，它将把消息的有效载荷发给所有的routee，不管该路由器通常如何路由其消息。

下面的示例演示了如何使用 `Broadcast` 消息向路由器下的每个routee发送一个非常重要的信息。

```
1.  import akka.routing.Broadcast
2.  router ! Broadcast("Watch out for Davy Jones' locker")
```

在此示例中路由器接收的广播消息，提取其有效载荷（ `"Watch out for Davy Jones' locker"` ），然后发送到有效载荷路由器的所有routee。它是由每个 routee actor来处理接收到的有效负载消息。

PoisonPill消息

PoisonPill消息对所有actor，包括路由actor，都有特殊的处理。当任何一个actor收到 `PoisonPill` 消息时，该actor将被停止。有关详细信息请参见PoisonPill。

```
1.  import akka.actor.PoisonPill
2.  router ! PoisonPill
```

路由器通常将消息传递给routee，但对于 `PoisonPill` 消息，需要认识到的很重要一点是它只被路由器处理。发送到路由器的 `PoisonPill` 消息将不会发送到routee。

然而，发送到路由器的 `PoisonPill` 消息可能仍会影响其routee，因为路由器停止时它也会停止其子actor。停止子actor是普通的actor行

为。路由器将会停止其作为子actor创建的各个routee。每个孩子将处理其当前的消息，然后停止。这可能会导致一些消息未被处理。停止actor的详细信息，请参阅文档。

如果你希望停止路由器及其routee，但你希望routee在停止前先处理目前在其邮箱中的所有消息，则你不应该发送 `PoisonPill` 消息。相反，你应该将 `PoisonPill` 包装在一个 `Broadcast` ，以便每个routee都能收到 `PoisonPill` 消息。请注意这将停止所有的routee，即使routee不是路由器的孩子，也就是即使是通过编程方式提供给router的routee。

```
1. import akka.actor.PoisonPill
2. import akka.routing.Broadcast
3. router ! Broadcast(PoisonPill)
```

如上代码所示，每个routee将收到一个 `PoisonPill` 消息。每个routee会继续如常处理其邮件，最终处理 `PoisonPill` 。这将导致routee停止。所有routee停止后，路由器将自动停止自己，除非它是一个动态的路由器，例如尺寸调整器。

> 注意
>
> *Brendan W McAdams*的优秀博客文章*"分布化Akka工作负载—以及完成后的关闭"*更详细地讨论了如何使用 *`PoisonPill`* 消息来关闭路由器和*routee*。

Kill消息

`Kill` 消息是另一种需要特殊处理的消息类型。请参阅"如何杀掉一个actor"来获取actor如何处理 `Kill` 消息的一般信息。

当 `Kill` 消息被发送到路由器，路由器将内部处理该消息，并且不会将它发送到其routee。路由器将抛出 `ActorKilledException` 并失败。然后它将被恢复、 重新启动或终止，取决于它如何被监督。

路由器的子routee亦将暂停，并将受应用在路由器上的监管指令影响。对不是路由器的孩子的Routee，即那些在路由器外部被创建的，将不受影响。

```
1. import akka.actor.Kill
2. router ! Kill
```

相比于 `PoisonPill` 消息，杀死一个路由器，间接杀死其子（即那些routee），和直接杀死routee（其中有些未必是其孩子）之间是有明显区别的。要直接杀死routee，路由器应发送包裹着 `Kill` 消息的 `Broadcast` 消息。

```
1. import akka.actor.Kill
2. import akka.routing.Broadcast
3. router ! Broadcast(Kill)
```

Managagement消息

- 发送`akka.routing.GetRoutees`到一个路由actor，使其回送一个包含当前使用routee的`akka.routing.Routees`消息。
- 发送`akka.routing.AddRoutee`到一个路由actor会将那个routee添加到其routee集合中。
- 发送`akka.routing.RemoveRoutee`到一个路由actor将从其routee集合删除该routee。
- 发送`akka.routing.AdjustPoolSize`到一个池路由actor将从其routee集合中添加或删除该数目的routee。

这些管理消息可能晚于其他消息处理，所以如果你发送 `AddRoutee` 后立即发送普通消息，并不能保证当普通消息被路由时，routee已被更改。如果你需要知道更改何时生效，你可以发送 `AddRoutee` 紧跟着 `GetRoutees` ，当你收到 `Routees` 答复，你就知道前面的变化已被应

用。

动态改变大小的池

大多数池可以使用固定数量的routee或有一个调整策略来动态调整routee数。

在配置中定义的包含resizer的池：

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

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

更多选项在配置的 `akka.actor.deployment.default.resizer` 部分中有描述。

在代码中定义的包含resizer的池：

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

需要指出如果你在配置文件中定义了 `router` ，那么这个值将比在代码

中传入的参数有更高的优先级。

> 注意
>
> 改变大小的行为是通过向*actor*池发送消息来触发的，但它不是完全同步的；而是向 `RouterActor` 的*“head”*发送消息来执行修改。所以在别的*actor*忙碌时，你不能假设改变大小的操作会立即创建新的工作*actor*，因为消息会被发到忙碌*actor*的邮箱中排队。要解决这个问题，配置*actor*池使用一个平衡的派发器，更多信息见*Configuring Dispatchers*。

Akka中的路由是如何设计的

从表面看，路由器就像普通的actor，但是它们实际实现是不同的。路由器在收消息和快速发消息给routee被设计的极度优化。

一个普通的actor可以用来路由消息，但是actor的单线程处理会成为一个瓶颈。路由器可以通过优化原有消息处理pipeline来支持多线程，从而达到更高的吞吐量。这里是通过直接嵌入路由逻辑到其 `ActorRef` ，而不是在路由actor本身。发送到路由器 `ActorRef` 的消息可以直接被路由到routee，从而完全跳过单线程的路由actor。

当然，这个改进的成本是路由代码的内部构造相比于使用普通actor构造来说复杂许多。幸运的是所有这种复杂性对于路由API消费者来说是不可见的。然而，这却是你在实现自己的路由器时需要意识到的。

自定义路由actor

如果觉得Akka自带的路由actor都不合用，你也可以创建自己的路由actor。要创建自己的路由，你需要满足本节中所列出的条件。

在创建你自己的路由器之前，你应该考虑一个拥有类似路由器行为的普通actor是否能完成一个成熟路由器的功能。正如上文解释，路由器相比于普通actor主要好处是他们拥有更高的性能。但相比普通actor他们的代码也更为复杂。因此，如果在你的应用程序中较低的最大吞吐量

是可以接受的，则不妨继续使用传统的actor。不过这一节假定你想要获得最大性能，并因而演示如何创建你自己的路由器。

在此示例中创建的路由器将把每个消息复制到几个目的地。

首先从路由逻辑开始：

```
1.  import scala.collection.immutable
2.  import scala.concurrent.forkjoin.ThreadLocalRandom
3.  import akka.routing.RoundRobinRoutingLogic
4.  import akka.routing.RoutingLogic
5.  import akka.routing.Routee
6.  import akka.routing.SeveralRoutees
7.
8.  class RedundancyRoutingLogic(nbrCopies: Int) extends RoutingLogic {
9.    val roundRobin = RoundRobinRoutingLogic()
10.   def select(message: Any, routees: immutable.IndexedSeq[Routee]):
    Routee = {
11.     val targets = (1 to nbrCopies).map(_ =>
    roundRobin.select(message, routees))
12.     SeveralRoutees(targets)
13.   }
14. }
```

在这个例子中 `select` 将被每个消息调用来使用轮询来挑选几个目的地，通过重用现有的 `RoundRobinRoutingLogic` 并将结果包装在一个 `SeveralRoutees` 实例中。 `SeveralRoutees` 将会把消息发送给所有提供的routee。

路由逻辑的实现必须是线程安全的，因为它可能在actor外被使用。

路由逻辑的一个单元测试：

```
1.  case class TestRoutee(n: Int) extends Routee {
2.    override def send(message: Any, sender: ActorRef): Unit = ()
3.  }
```

```
4.
5.    val logic = new RedundancyRoutingLogic(nbrCopies = 3)
6.
7.    val routees = for (n <- 1 to 7) yield TestRoutee(n)
8.
9.    val r1 = logic.select("msg", routees)
10.   r1.asInstanceOf[SeveralRoutees].routees should be(
11.     Vector(TestRoutee(1), TestRoutee(2), TestRoutee(3)))
12.
13.   val r2 = logic.select("msg", routees)
14.   r2.asInstanceOf[SeveralRoutees].routees should be(
15.     Vector(TestRoutee(4), TestRoutee(5), TestRoutee(6)))
16.
17.   val r3 = logic.select("msg", routees)
18.   r3.asInstanceOf[SeveralRoutees].routees should be(
19.     Vector(TestRoutee(7), TestRoutee(1), TestRoutee(2)))
```

你可以停在这儿，通过 `akka.routing.Router` 使用 `RedundancyRoutingLogic` ，如一个简单路由器中所述。

让我们继续，并使之成为一个自包含的、可配置的路由器actor。

创建一个类来扩展 `Pool` ， `Group` 或 `CustomRouterConfig` 。该类是一个路由逻辑的工厂，并持有路由器的配置。在这里，我们把它变成一个 `Group` 。

```
1.  import akka.dispatch.Dispatchers
2.  import akka.routing.Group
3.  import akka.routing.Router
4.  import akka.japi.Util.immutableSeq
5.  import com.typesafe.config.Config
6.
7.  case class RedundancyGroup(override val paths:
    immutable.Iterable[String], nbrCopies: Int) extends Group {
8.
9.    def this(config: Config) = this(
10.     paths = immutableSeq(config.getStringList("routees.paths")),
```

```
11.     nbrCopies = config.getInt("nbr-copies"))
12.
13.   override def createRouter(system: ActorSystem): Router =
14.     new Router(new RedundancyRoutingLogic(nbrCopies))
15.
16.   override val routerDispatcher: String =
    Dispatchers.DefaultDispatcherId
17. }
```

这样就可以像Akka提供的路由actor完全一样使用。

```
1. for (n <- 1 to 10) system.actorOf(Props[Storage], "s" + n)
2.
3. val paths = for (n <- 1 to 10) yield ("/user/s" + n)
4. val redundancy1: ActorRef =
5.   system.actorOf(RedundancyGroup(paths, nbrCopies = 3).props(),
6.     name = "redundancy1")
7. redundancy1 ! "important"
```

请注意我们在 `RedundancyGroup` 添加一个以 `Config` 为参数的构造函数。这样一来，我们就可能在配置中定义。

```
1. akka.actor.deployment {
2.   /redundancy2 {
3.     router = "docs.routing.RedundancyGroup"
4.     routees.paths = ["/user/s1", "/user/s2", "/user/s3"]
5.     nbr-copies = 5
6.   }
7. }
```

请注意 `router` 属性中的类的全名。路由器类必须继承 `akka.routing.RouterConfig` （ `Pool` ， `Group` 或 `CustomRouterConfig` ），并且有一个以 `com.typesafe.config.Config` 为参数的构造函数。配置的部署部分将被传递给构造函数。

```
val redundancy2: ActorRef = system.actorOf(FromConfig.props(),
  name = "redundancy2")
redundancy2 ! "very important"
```

配置调度器" class="reference-link">配置调度器

创建子actor池的调度器将取自 `Props` ，如调度器中所述。

为了可以很容易地定义池routee的调度器，你可以在配置的部署一节中定义内联调度器。

```
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
    }
  }
}
```

这是启用一个池专用调度器，你唯一需要做的。

> 注意
>
> 如果你使用*actor*群，并路由到它们的路径，然后他们将仍然使用配置在其 *Props* 的相同的调度器，不可能在*actor*创建后改变*actor*的调度器。

“头”路由器不能总是在相同的调度器上运行，因为它不处理同一类型的消息，因此这个特殊的actor没有使用配置的调度器，但相反，使用 `RouterConfig` 的 `routerDispatcher` ，它是actor系统默认调度器默认的。所有标准路由器允许它们在构造函数或工厂方法中设置此属性，自定义路由器必须以适当的方式实现该方法。

```
1.  val router: ActorRef = system.actorOf(
2.    // “head” router actor will run on "router-dispatcher" dispatcher
3.    // Worker routees will run on "pool-dispatcher" dispatcher
4.    RandomPool(5, routerDispatcher = "router-
    dispatcher").props(Props[Worker]),
5.    name = "poolWithDispatcher")
```

> 注意
>
> 不允许配置 `routerDispatcher` 为 `akka.dispatch.BalancingDispatcherConfigurator` ，因为用于特殊路由*actor*的消息不能被任意其他*actor*处理。

有限状态机(FSM)

有限状态机(FSM)

概述

FSM（有限状态机）可以mixin到akka Actor中，其概念在Erlang设计原则中有最好的描述。

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，则我们应执行操作*A*，然后将状态转换为*S'*。

一个简单的例子

为了演示 `FSM` trait的大部分功能，考虑一个actor，它接收到一组突然爆发的消息而将其送入邮箱队列，然后在消息爆发期过后或收到flush请求时再对消息进行发送。

首先，假设以下所有代码都使用这些import语句:

```
1. import akka.actor.{ Actor, ActorRef, FSM }
2. import scala.concurrent.duration._
```

我们的 “Buncher” actor的契约是接收或产生以下消息:

```
1. // received events
2. case class SetTarget(ref: ActorRef)
3. case class Queue(obj: Any)
4. case object Flush
5.
6. // sent events
7. case class Batch(obj: immutable.Seq[Any])
```

`SetTarget` 用来启动，为 `Batches` 设置发送目标； `Queue` 会添加到内部队列而 `Flush` 标志着消息爆发的结束。

```
1. // states
2. sealed trait State
3. case object Idle extends State
4. case object Active extends State
```

```
5.
6.  sealed trait Data
7.  case object Uninitialized extends Data
8.  case class Todo(target: ActorRef, queue: immutable.Seq[Any])
    extends Data
```

这个actor可以处于两种状态：队列中没有消息（即 `Idle` ）或有消息（即 `Active` ）。只要一直有消息进来并且没有flush请求，它就停留在active状态。这个actor的内部状态数据是由批消息的发送目标actor引用和实际的消息队列组成.

现在让我们看看我们的FSM actor的框架：

```
1.  class Buncher extends Actor with FSM[State, Data] {
2.
3.    startWith(Idle, Uninitialized)
4.
5.    when(Idle) {
6.      case Event(SetTarget(ref), Uninitialized) =>
7.        stay using Todo(ref, Vector.empty)
8.    }
9.
10.   // transition elided ...
11.
12.   when(Active, stateTimeout = 1 second) {
13.     case Event(Flush | StateTimeout, t: Todo) =>
14.       goto(Idle) using t.copy(queue = Vector.empty)
15.   }
16.
17.   // unhandled elided ...
18.
19.   initialize()
20. }
```

基本策略就是声明actor类，混入 `FSM` trait，并将可能的状态和数据设定为类型参数。在actor的内部使用一个DSL来声明状态机：

- `startsWith`定义初始状态和初始数据
- 然后对每一个状态有一个`when(<state>) { ... }`声明待处理的事件（可以是多个，传入的`PartialFunction`将用`orElse`进行连接）
- 最后使用`initialize`来启动它，这会执行到初始状态的转换并启动定时器（如果需要的话）。

在这个例子中，我们从`Idle`和`Uninitialized`状态开始，这两种状态下只处理`SetTarget()`消息；`stay`准备结束这个事件的处理而不离开当前状态，而`using`使得FSM将其内部状态（这时为`Uninitialized`）替换为一个新的包含目标actor引用的`Todo()`对象。`Active`状态声明了一个状态超时，意思是如果1秒内没有收到消息，将生成一个`FSM.StateTimeout`消息。在本例中这与收到`Flush`指令消息具有相同的效果，即转回`Idle`状态并将内部队列重置为空vector。但消息是如何进入队列的？由于在两种状态下都要做这件事，我们利用了任何`when()`块未处理的消息被发送到`whenUnhandled()`块这个事实：

```
1.  whenUnhandled {
2.    // common code for both states
3.    case Event(Queue(obj), t @ Todo(_, v)) =>
4.      goto(Active) using t.copy(queue = v :+ obj)
5.
6.    case Event(e, s) =>
7.      log.warning("received unhandled request {} in state {}/{}", e,
    stateName, s)
8.      stay
9.  }
```

这里第一个case是将`Queue()`请求加入内部队列中并进入`Active`状态（当然显然地，如果已经在`Active`状态则停留），前提是收到`Queue()`时FSM数据不是`Uninitialized`。否则—及其它所有未命中的情况—第二个case记录一个警告到日志并保持内部状态。

最后剩下的只有 `Batches` 实际上是如何发送到目标的，这里我们使用 `onTransition` 机制：你可以声明多个这样的块，在状态切换发生时（即只有当状态真正改变时）所有的块都将被尝试来作匹配。

```
onTransition {
  case Active -> Idle =>
    stateData match {
      case Todo(ref, queue) => ref ! Batch(queue)
    }
}
```

状态转换回调是一个偏函数，它以一对状态作为输入的—当前状态和下一个状态。FSM trait为此提供了一个方便的箭头形式的提取器，非常贴心地提醒你所匹配到的状态转换的方向。在状态转换过程中，如示例所示旧状态数据可以通过 `stateData` 获得，新状态数据可以通过 `nextStateData` 获得。

要确认这个buncher真实可用，可以很简单地利用测试Actor系统(Scala)中的工具写一个测试，它方便地将ScalaTest trait融入 `AkkaSpec` 中：

```
import akka.actor.Props
import scala.collection.immutable

class FSMDocSpec extends MyFavoriteTestFrameWorkPlusAkkaTestKit {

  // fsm code elided ...

  "simple finite state machine" must {

    "demonstrate NullFunction" in {
      class A extends Actor with FSM[Int, Null] {
        val SomeState = 0
        when(SomeState)(FSM.NullFunction)
      }
```

```
15.     }
16.
17.     "batch correctly" in {
18.       val buncher = system.actorOf(Props(classOf[Buncher], this))
19.       buncher ! SetTarget(testActor)
20.       buncher ! Queue(42)
21.       buncher ! Queue(43)
22.       expectMsg(Batch(immutable.Seq(42, 43)))
23.       buncher ! Queue(44)
24.       buncher ! Flush
25.       buncher ! Queue(45)
26.       expectMsg(Batch(immutable.Seq(44)))
27.       expectMsg(Batch(immutable.Seq(45)))
28.     }
29.
30.     "not batch if uninitialized" in {
31.       val buncher = system.actorOf(Props(classOf[Buncher], this))
32.       buncher ! Queue(42)
33.       expectNoMsg
34.     }
35.   }
36. }
```

参考

FSM Trait 及 FSM Object

`FSM` trait只能被混入到 `Actor` 子类中。这里选择了 使用self类型的写法而不是继承 `Actor` ，这样是为了标明事实上创建的是一个actor。//TODO这里代码有误？

```
1. class Buncher extends Actor with FSM[State, Data] {
2.
3.   // fsm body ...
4.
5.   initialize()
6. }
```

> 注意
>
> *FSM*特质定义 `receive` 方法处理内部消息，并将其它一切通过*FSM*逻辑传递（根据当前状态）。当重写 `receive` 方法，请牢记例如超时状态处理取决于实际通过*FSM*逻辑传递的消息。

`FSM` trait 有两个类型参数 ：

1. 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
2. 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；如果你坚持这种模式，不向*FSM*类中加入可变量成员，你就可以充分享受在一些周知的位置改变所有内部状态的好处.

定义状态

状态的定义是通过一次或多次调用

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

方法。

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；特别是它不能为可变变量。满足这些条件的最简单的就是 case objects。

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。初始化转换时显式指定一个超时可以用来覆盖这个缺省行为，更多信息见发起状态转换。在操作执行的过程中可以通过使用 `setStateTimeout(state, duration)` 方法来修改任何状态的超时时间。这使得运行时配置（例如通过外部消息）成为可能。

参数 `stateFunction` 是一个 `PartialFunction[Event, State]` ，可以很方便地用偏函数的语法来指定，见下例：

```
1.  when(Idle) {
2.    case Event(SetTarget(ref), Uninitialized) =>
3.      stay using Todo(ref, Vector.empty)
4.  }
5.
6.  when(Active, stateTimeout = 1 second) {
7.    case Event(Flush | StateTimeout, t: Todo) =>
8.      goto(Idle) using t.copy(queue = Vector.empty)
9.  }
```

该 `Event(msg: Any, data: D)` case类将FSM持有的数据类型参数化，从而可以方便地进行模式匹配。

> 警告
>
> 它要求你为所有可能的*FSM*状态定义处理程序，否则尝试切换到未声明的状态时会有失败。

推荐的实践做法是，将状态声明为对象，继承自一个sealed特质，然后验证是否对每一个状态都有 `when` 子句。如果你想要一些状态“不被处理”（详见下文），它仍然需要像这样声明：

```
1.  when(SomeState)(FSM.NullFunction)
```

定义初始状态

每个FSM都需要一个起点，用以下代码声明

```
1.  startWith(state, data[, timeout])
```

其中可选的超时参数将覆盖所有为期望的初始状态所指定的值。如果你想要取消缺省的超时，使用 `Duration.Inf` 。

未处理事件

如果一个状态未能处理一个收到的事件，日志中将记录一条警告。这种情况下如果你想做点其它的事，你可以使用 `whenUnhandled(stateFunction)` 来指定：

```
1. whenUnhandled {
2.   case Event(x: X, data) =>
3.     log.info("Received unhandled event: " + x)
4.     stay
5.   case Event(msg, _) =>
6.     log.warning("Received unknown event: " + msg)
7.     goto(Error)
8. }
```

在此处理程序中可以使用 `stateName` 方法查询FSM的状态。

重要：这个处理器不会入栈叠加，这意味着 `whenUnhandled` 的每一次调用都会覆盖先前指定的处理程序。

发起状态转换" class="reference-link">发起状态转换

任何 `stateFunction` 的结果都必须是下一个状态的定义，除非是终止FSM，这种情况在从内部终止中介绍。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。结果对象可以通过下面列出的修饰器作进一步的限制：

- `forMax(duration)`

  这个修饰器为新状态指定了一个状态超时。这意味着将启动一个定时器，它过期时将向FSM发送一个 `StateTimeout` 消息。其间接收到任何其它消息，定时器都将被取消；你可以确定的事实是 `StateTimeout` 消息不会在任何一个中间消息之后被处理。

这个修饰器也可以用于覆盖任何对目标状态指定的缺省超时。如果要取消缺省超时，可以使用 `Duration.Inf` 。

- `using(data)`

  这个修饰器用给定的新数据替换旧的状态数据。如果你遵循上面的建议，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完的消息发送一个应答，不同的是它不会改变状态转换。

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：

```
1. when(SomeState) {
2.   case Event(msg, _) =>
3.     goto(Processing) using (newData) forMax (5 seconds) replying
   (WillDo)
4. }
```

事实上这里所有的括号都不是必须的，但它们在视觉上将修饰器和它们的参数区分开，因而使代码对于他人来说有更好的可读性。

> 注意
>
> 请注意 *`return`* 语句不可以用于 *`when`* 或类似的代码块中；这是*Scala*的限制。要么使用 *`if () ... else ...`* 重构你的代码，要么将它改写到一个方法定义中。

监控状态转换

在概念上，转换发生在“两个状态之间”，也就是在你放在事件处理代码块执行的任何操作之后；这是显然的，因为只有在事件处理逻辑返回了值以后，才能确定下一个状态。相对于设置内部状态变量，你不需要担心操作顺序的细节，因为FSM actor中的所有代码都是在一个线程中运行的。

内部监控

到目前为止，FSM DSL都围绕着状态和事件。另外一种视角是将其描述成一系列的状态转换。是通过这个方法实现的

```
1.  onTransition(handler)
```

它将操作与状态转换联系起来，而不是联系状态与事件。这个处理器是一个偏函数，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。

```
1.  onTransition {
2.    case Idle -> Active => setTimer("timeout", Tick, 1 second, true)
3.    case Active -> _    => cancelTimer("timeout")
4.    case x -> Idle      => log.info("entering Idle from " + x)
5.  }
```

提取器 `->` 用来解开状态对，并以清晰的形式表达了状态转换的方向。与通常的模式匹配一样，可以用 `_` 来表示不关心的内容；或者你可以将不关心的状态绑定到一个无约束的变量，例如像上一个例子那样供记日志使用。

也可以向 `onTransition` 传递一个以两个状态为参数的函数，此时你的状态转换处理逻辑是定义成方法的：

```
1.  onTransition(handler _)
2.
3.  def handler(from: StateType, to: StateType) {
4.    // handle it here ...
5.  }
```

用这个方法注册的处理器是堆栈迭加的，这样你可以将 `onTransition` 块和 `when` 块分散定义以适应设计的需要。但需要注意的是，所有的处理器对每一次状态转换都会被调用，而不只是最先匹

配的那个。这是有意设计的，使得你可以将某一部分状态转换处理放在某一个地方，而不用担心先前的定义会屏蔽后面的逻辑；当然这些操作还是按定义的顺序执行的。

> 注意
>
> 这种内部监控可以用于通过状态转换来构建你的*FSM*，这样在添加新的目标状态时，不会忘记例如在离开某个状态时，取消定时器这种操作。

外部监控

可以通过发送一个 `SubscribeTransitionCallBack(actorRef)` 消息注册外部actor，来接收状态转换的通知。这个被命名的actor将立即收到 `CurrentState(self, stateName)` 消息，并在之后每次进入新状态时收到 `Transition(actorRef, oldState, newState)` 消息。可以通过向FSM actor发送 `UnsubscribeTransitionCallBack(actorRef)` 来注销外部监控actor。

停止一个监听器，而不注销，将不会从注册列表中移除它；需要在停止监听器前使用 `UnsubscribeTransitionCallback` 。

转换状态

给 `when()` 传递的偏函数参数，可以使用Scala充分的函数式编程工具来转换。为了保留类型推断，还有一个辅助函数，它可以在通用处理逻辑中使用，并被应用到不同子句：

```
1. when(SomeState)(transform {
2.   case Event(bytes: ByteString, read) => stay using (read +
   bytes.length)
3. } using {
4.   case s @ FSM.State(state, read, timeout, stopReason, replies) if
   read > 1000 =>
5.     goto(Processing)
6. })
```

不用说此方法的参数也可能被存储，被多次使用，例如在几个不同的 `when()` 代码块中应用相同的转换：

```
1.  val processingTrigger: PartialFunction[State, State] = {
2.    case s @ FSM.State(state, read, timeout, stopReason, replies) if
    read > 1000 =>
3.      goto(Processing)
4.  }
5.
6.  when(SomeState)(transform {
7.    case Event(bytes: ByteString, read) => stay using (read +
    bytes.length)
8.  } using processingTrigger)
```

定时器

除了状态超时，FSM还管理以 `String` 类型名称为标识的定时器。你可以用下面代码设置定时器

```
1.  setTimer(name, msg, interval, repeat)
```

其中 `msg` 是经过 `interval` 时间以后发送的消息对象。 如果 `repeat` 设成 `true` ，定时器将以 `interval` 参数指定的时间段重复规划。添加新的计时器之前，具有相同名称的任何现有计时器将被自动取消。

可以用下面代码取消定时器

```
1.  cancelTimer(name)
```

取消操作确保立即执行，这意味着在这个调用之后，定时器已经规划的消息将不会执行，即使定时器已经发起并入队该消息。任何定时器的状态可以用下面的代码进行查询

```
1. isTimerActive(name)
```

这些具名定时器是对状态超时的补充，因为它们不受中间收到的其它消息的影响。

从内部终止" class="reference-link">从内部终止

可以像下面这样指定结果状态来终止FSM

```
1. stop([reason[, data]])
```

其中的 `reason` 必须是 `Normal` （默认值）、 `Shutdown` 或 `Failure(reason)` 之一，可以提供第二个参数来改变状态数据，在终止处理器中可以使用该数据。

> 注意
>
> 必须注意 *`stop`* 并不会中止当前的操作并立即停止*FSM*。 *`stop`* 操作必须像状态转换一样从事件处理器中返回（但要注意 *`return`* 语句不能用在 *`when`* 代码块中）。

```
1. when(Error) {
2.   case Event("stop", _) =>
3.     // do cleanup ...
4.     stop()
5. }
```

你可以用 `onTermination(handler)` 来指定当FSM停止时要运行的代码。其中的handler是以一个 `StopEvent(reason, stateName, stateData)` 为参数的偏函数：

```
1. onTermination {
2.   case StopEvent(FSM.Normal, state, data)         => // ...
3.   case StopEvent(FSM.Shutdown, state, data)       => // ...
4.   case StopEvent(FSM.Failure(cause), state, data) => // ...
5. }
```

对于使用 `whenUnhandled` 的场合，这个处理器不会堆栈迭加，所以每次 `onTermination` 调用都会替换先前指定的处理器。

从外部终止

当与FSM关联的 `ActorRef` 被 `stop` 方法停止后，它的 `postStop` hook 将被执行。在 `FSM` 特质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。

> 警告
>
> 如果你重写 *`postStop`* 并而希望你的 *`onTermination`* 处理器被调用，不要忘记调用 *`super.postStop`* 。

测试和调试有限状态机

在开发和调试过程中，FSM和其它actor一样需要照顾。测试有限状态机以及下文中介绍了一些专门的工具。

事件跟踪

配置文件中的 `akka.actor.debug.fsm` 打开用 `LoggingFSM` 实例完成的事件跟踪日志：

```
1. import akka.actor.LoggingFSM
2. class MyFSM extends Actor with LoggingFSM[StateType, Data] {
3.   // body elided ...
4. }
```

这个FSM将以DEBUG级别记录日志：

- 所有处理完的事件，包括 `StateTimeout` 和计划的定时器消息
- 所有具名定时器的设置和取消
- 所有的状态转换

生命周期变化及特殊消息可以如Actor中所述进行日志记录。

滚动事件日志

`LoggingFSM` 特质为FSM添加了一个新的特性：一个滚动的事件日志，它可以在debugging中使用（跟踪为什么FSM会进入某个失败的状态）或其它的新用法：

```
1.  import akka.actor.LoggingFSM
2.  class MyFSM extends Actor with LoggingFSM[StateType, Data] {
3.    override def logDepth = 12
4.    onTermination {
5.      case StopEvent(FSM.Failure(_), state, data) =>
6.        val lastEvents = getLog.mkString("\n\t")
7.        log.warning("Failure in state " + state + " with data " +
    data + "\n" +
8.          "Events leading up to this point:\n\t" + lastEvents)
9.    }
10.   // ...
11. }
```

`logDepth` 缺省值为0，意思是关闭事件日志。

> 警告
>
> 日志缓冲区是在*actor*创建时分配的，这也是为什么*logDepth*的配置使用了虚方法调用。如果你想用一个 *val* 对其进行覆盖，必须保证它的初始化在 *LoggingFSM* 的初始化之前完成，而且在缓冲区分配完成后不要修改 *logDepth* 返回的值。

事件日志的内容可以用 `getLog` 方法获取，它返回一个 `IndexedSeq[LogEntry]` ，其中最老的条目下标为0。

示例

在Typesafe Activator的模板工程Akka FSM in Scala中，可以找到一个比Actor's `become/unbecome` 更大的FSM示例。

测试Actor系统

测试Actor系统

对于任何软件开发，自动化测试都是开发过程中一个重要组成部分。actor 模型对于代码单元如何划分，它们之间如何交互提供了一种新的视角，这对如何编写测试也造成了影响。

Akka 有一个专门的模块—— `akka-testkit` 来支持不同层次上的测试，测试很明显有两个类别：

- 测试独立的、不包括actor模型的代码，即没有多线程的内容；这意味着给事件发生的次序给定，有完全确定的行为，没有任何并发考虑，这在下文中称为 单元测试（**Unit Testing**）。
- 测试（多个）包装过的actor，包括多线程调度；这意味着事件的次序没有确定性，但由于使用了actor模型而不需要考虑并发，这在下文中被称为 集成测试（**Integration Testing**）。

当然这两个类型有着不同的测试粒度，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。是否把并发考虑为测试的一部分，是其中重要的区别。我们提供的工具将在下面的章节中详细介绍。

> 注意
>
> 请确保在依赖项中加入 *`akka-testkit`* 模块。

用`TestActorRef`做同步单元测试

`Actor`类中的业务逻辑测试可以分为两部分：首先，每个原子操作必须独立工作，然后输入的事件序列必须被正确处理，即使事件的次序存在一些可能的变化。前者是单线程单元测试的主要使用场景，而后者可以在集成测试中进行确认。

通常，`ActorRef`将实际的`Actor`实例与外界隔离开，唯一的通信通道是actor的邮箱。这个限制是单元测试的障碍，所以我们引入了`TestActorRef`。这个特殊类型的引用是专门为测试而设计的，它允许以两种方式访问actor：通过获取实际actor实例的引用，或者通过调用或查询actor的行为（`receive`）。下文中对每一种方式都有专门的章节介绍。

获取一个`Actor`的引用

能够访问到实际的`Actor`对象使得所有的传统单元测试技术都可以用于测试其中的方法。可以像这样获取一个引用：

```
1. import akka.testkit.TestActorRef
2.
3. val actorRef = TestActorRef[MyActor]
4. val actor = actorRef.underlyingActor
```

由于`TestActorRef`是actor类型的高阶类型（泛型），因此它返回的底层actor具有正确的静态类型。这样以后你就可以象平常一样将任何单元测试工具用于actor上了。

测试有限状态机" class="reference-link">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，并且额外地允许访问其内部状态：

```
import akka.testkit.TestFSMRef
import akka.actor.FSM
import scala.concurrent.duration._

val fsm = TestFSMRef(new TestFsmActor)

val mustBeTypedProperly: TestActorRef[TestFsmActor] = fsm

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)

assert(fsm.isTimerActive("test") == false)
fsm.setTimer("test", 12, 10 millis, true)
assert(fsm.isTimerActive("test") == true)
fsm.cancelTimer("test")
assert(fsm.isTimerActive("test") == false)
```

由于Scala类型推断的限制，只有一个如上所示的工厂方法，所以你很可能需要写像 `TestFSMRef(new MyFSM)` 这样的代码，而不是想象中的类似 `ActorRef` 的 `TestFSMRef[MyFSM]` 。上例所示的所有方法都直接访问FSM的状态，不作任何同步；这在使用 `CallingThreadDispatcher` 且没有其它线程参与的情况下是合适的，但如果你实际上需要处理定时器事件可能会导致意外的情形，因为它们是在 `Scheduler` 线程中执行的。

测试Actor的行为

当消息派发器调用actor的逻辑来处理一条消息时，它实际上是对当前注册到actor的行为调用了 `apply` 。行为的初始值是代码中声明

的 `receive` 方法的返回值，但可以通过对外部消息的响应调用 `become` 和 `unbecome` 来改变这个行为。所有这些特性使得进行actor的行为测试不太容易。因此 `TestActorRef` 提供了一种不同的操作方式来对 `Actor` 的测试进行补充：它支持所有正常的 `ActorRef` 中的操作。发往actor的消息在当前线程中同步处理，应答象往常一样回送。这个技巧来自下面所介绍的 `CallingThreadDispatcher` ；这个派发器被隐式地用于所有实例化为 `TestActorRef` 的actor。

```
1.  import akka.testkit.TestActorRef
2.  import scala.concurrent.duration._
3.  import scala.concurrent.Await
4.  import akka.pattern.ask
5.
6.  val actorRef = TestActorRef(new MyActor)
7.  // hypothetical message stimulating a '42' answer
8.  val future = actorRef ? Say42
9.  val Success(result: Int) = future.value.get
10.  result should be(42)
```

由于 `TestActorRef` 是 `LocalActorRef` 的子类，只是附加了一些特殊功能，所以像监管和重启这样的功能也能正常工作，但要注意：只要所有相关的actor都使用 `CallingThreadDispatcher` ，那么所有的执行过程都是严格同步的。一旦你增加了一些元素，其中包括比较复杂的定时任务，你就离开了单元测试的范畴，因为你必须要重新将异步性纳入考虑范围（在大多数情况下问题在于要等待希望的结果有机会发生）。

另一个在单线程测试中被覆盖的特殊点是 `receiveTimeout` ，由于包含了它会产生异步的 `ReceiveTimeout` 消息队列，因此与同步约定矛盾。

> 警告
>
> 综上所述： *`TestActorRef`* 重写了两个成员：它设置派发器为 *`CallingThreadDispatcher.global`* ，设置 *`receiveTimeout`* 为*None*。

介于两者之间方法：期望异常

如果你想要测试actor的行为，包括热替换，但是不涉及消息派发器，也不希望 `TestActorRef` 吞掉所有抛出的异常，那么为你准备了另一种模式：只要使用 `TestActorRef` 的 `receive` 方法 ，这将会把消息转发给内部的 actor：

```
1. import akka.testkit.TestActorRef
2.
3. val actorRef = TestActorRef(new Actor {
4.   def receive = {
5.     case "hello" => throw new IllegalArgumentException("boom")
6.   }
7. })
8. intercept[IllegalArgumentException] { actorRef.receive("hello") }
```

使用场景

当然你可以根据自己的测试需求来混合使用 `TestActorRef` 的不同用法：

- 一个常见的使用场景是在发送测试消息之前设置actor进入某个特定的内部状态
- 另一个场景是发送了测试消息之后验证内部状态转换的正确性

放心大胆地对各种可能性进行实验，如果你发现了有用的模式，快让Akka论坛知道它！常用操作甚至可能放进优美的DSL中，谁知道呢。

用`TestKit`进行异步集成测试

当基本确定你的actor的业务逻辑是正确的之后，下一步就是确认它在目标环境中也能正确工作（如果actor个体都足够简单，可能是因为他们使用了 `FSM` 模块，这也可以放到第一步）。关于环境的定义当然很大程度上由手头的问题和你打算测试的程度来决定，从功能/集成测

试到完整的系统测试。最简单的步骤包括测试过程（提供所期望的触发条件）、要测试的actor和接收应答的actor。大一些的系统将被测的actor替换成一组actor网络，将触发条件应用于不同的切入点，并整理将会从不同的输出位置发送的结果，但基本的原则保持不变，就是测试由一个单独的流程来驱动。

`TestKit` 类包含一组工具来简化这些常用的工作.

```
1.  import akka.actor.ActorSystem
2.  import akka.actor.Actor
3.  import akka.actor.Props
4.  import akka.testkit.{ TestActors, TestKit, ImplicitSender }
5.  import org.scalatest.WordSpecLike
6.  import org.scalatest.Matchers
7.  import org.scalatest.BeforeAndAfterAll
8.
9.  class MySpec(_system: ActorSystem) extends TestKit(_system) with
    ImplicitSender
10.   with WordSpecLike with Matchers with BeforeAndAfterAll {
11.
12.   def this() = this(ActorSystem("MySpec"))
13.
14.   override def afterAll {
15.     TestKit.shutdownActorSystem(system)
16.   }
17.
18.   "An Echo actor" must {
19.
20.     "send back messages unchanged" in {
21.       val echo = system.actorOf(TestActors.echoActorProps)
22.       echo ! "hello world"
23.       expectMsg("hello world")
24.     }
25.
26.   }
27. }
```

`TestKit` 中有一个名为 `testActor` 的actor作为将要被不同的 `expectMsg...` 断言检查的消息的入口，下面会详细介绍这些断言。 当混入了 `ImplicitSender` trait后，这个测试actor会在整个测试过程中，被隐式地用作消息的发送者引用。 `testActor` 也可以像平常一样被发送给其它的actor，通常是订阅成为通知监听器。有一套检查方法，例如 接收所有匹配某些条件的消息，接收一系列固定的消息序列或类，在某段时间内收不到消息，等。

作为参数传递到 `TestKit` 构造函数的 `ActorSystem` 可以通过 `system` 成员来访问。记得在测试完成后关闭actor系统（即使是在测试失败的情况下）以保证所有的actor—包括测试actor—被停止.

内置断言

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`

  给定的消息对象必须在指定的时间内到达；该消息对象将被返回。

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

  在给定的时间段内，必须有一条消息到达，必须为这类消息定义了偏函数；将收到的消息应用于偏函数并返回其结果。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`

  在分配的时间片内必须接收到 `Class` 类型的对象实例；并返回该对象。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)`

  在分配的时间片内必须收到指定类型（擦除后)的对象实例；并返回该对象。这个方法基本上与`expectMsgClass(implicitly[ClassTag[T]].runtimeClass)`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多个对象引用中至少一个相等(用 `==` 进行比较)；并返回该对象。

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

  在指定的时间内必须收到一个对象，它必须指定的多个`Class`中某一个类对象的实例；并返回该对象。

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

  在指定时间内必须收到与指定的对象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之相等(用`==`进行比较）。返回收到对象的完整序列。

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

  在指定时间内必须收到与指定的 `Class` 数组相等数量的对象，对数组中的每一个类，必须至少有一个对象的类与之相等(用`==`进行比较）（这 不是 子类兼容的类型检查)。返回收到对象的完整序列。

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

  在指定时间内必须收到与指定的 `Class` 数组相等数量的对象，对数组中的每个类必须至少有一个对象是这个类的实例。返回收到对象的完整序列。

- `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，而且偏函数匹配收到的消息并返回 `false` ，则一直接收消息。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供debug使用。

除了消息接收断言，还有一些方法来对消息流提供帮助：

- `receiveOne(d: Duration): AnyRef`

  尝试等待给定的时间间隔来接收一个消息，如果失败则返回 `null` 。如果给定的 `Duration` 是0，则调用是非阻塞的（轮询模式）。

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

  只要满足以下条件就收集消息：

  - 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

返回收集到的所有消息。时间上限缺省值是最深层的 `within` 块中的剩余时间，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。

- `awaitAssert(a: => Any, max: Duration, interval: Duration)`

  在每个时间间隔验证一遍给定的断言函数，直到它不抛出异常，或 `max` 持续时间用完。如果在超时已到则抛出最后一个异常。间隔默认为 `100 ms` 和最大值默认为最深层的 `within` 块中的剩余时间。
  TODO Akka文档中的重复

- `ignoreMsg(pf: PartialFunction[AnyRef, Boolean])`
  `ignoreNoMsg`

  内部的 `testActor` 包含一个偏函数用来忽略消息：它只会将与偏函数不匹配或使函数返回 `false` 的消息放进队列。 这个函数可以用上面的方法进行设置和重设；每一次调用都会覆盖之前的函数，而不会迭加。

  这个功能在你想忽略一般的消息而只对指定的一些消息感兴趣时（例如测试日志系统时）比较有用。

期望日志消息

由于集成测试不允许进入参与测试的actor内部处理流程，无法直接确

认预料中的异常。因此为了做这件事，只能使用日志系统：将普通的事件处理器替换成 `TestEventListener` 并使用一个 `EventFilter` ，从而可以对日志消息，包括由于异常产生的日志，做断言：

```
1.  import akka.testkit.EventFilter
2.  import com.typesafe.config.ConfigFactory
3.
4.  implicit val system = ActorSystem("testsystem",
    ConfigFactory.parseString("""
5.    akka.loggers = ["akka.testkit.TestEventListener"]
6.    """))
7.  try {
8.    val actor = system.actorOf(Props.empty)
9.    EventFilter[ActorKilledException](occurrences = 1) intercept {
10.     actor ! Kill
11.   }
12. } finally {
13.   shutdown(system)
14. }
```

如果出现的次数是具体的—如上所示—然后 `intercept` 将阻塞直到收到匹配消息达到这一数字，或 `akka.test.filter-leeway` 配置的超时时间用完了（在传入的代码块返回后，时间开始计数）。超时时测试失败。

> 注意
>
> 一定要在你的 *`application.conf`* 中替换默认的日志记录器为 *`TestEventListener`* ，以启用此功能：
>
> ```
> 1. akka.loggers = [akka.testkit.TestEventListener]
> ```

对定时进行断言

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。因此所有的进行检查的方法 都接收一个时间上限，不论是正面还是负面的结果都应该在这个时

间之前获得。时间下限需要在这个检测方法之外进行检查，因而有一个新的工具来管理时间期限：

```
1.     within([min, ]max) {
2.       ...
3.     }
```

传给 `within` 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` 缺省值为0。将 `max` 参数与块的启动时间相加得到的时间期限在所有检查方法块内部都可以隐式获得，如果你没有指定 `max` 值，它会从最深层的 `within` 块继承这个值.

应注意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。

```
1.  import akka.actor.Props
2.  import scala.concurrent.duration._
3.
4.  val worker = system.actorOf(Props[Worker])
5.  within(200 millis) {
6.    worker ! "some work"
7.    expectMsg("some result")
8.    expectNoMsg // will block for the rest of the 200ms
9.    Thread.sleep(300) // will NOT make this block fail
10. }
```

> 注意
>
> 所有的时间都以 `System.nanoTime` 为单位，即它们描述的是墙上时间，而非CPU时间.

Ray Roestenburg 写了一篇关于使用 TestKit 的好文。完整的示例也可以在这里找到.

考虑很慢的测试系统

你在跑得飞快的笔记本上使用的超时设置在高负载的Jenkins（或类似的）服务器上通常都会导致虚假的测试失败。 为了考虑这种情况，所有的时间上限都在内部乘以一个系数，这个系数来自配置文件中的 `akka.test.timefactor` ，缺省值为 1。

你也可以用 `akka.testkit` 包对象中的隐式转换来将同样的系数来作用于其它的时限，为 `Duration` 添加扩展函数。

```
1. import scala.concurrent.duration._
2. import akka.testkit._
3. 10.milliseconds.dilated
```

用隐式的ActorRef解决冲突

如果你希望在基于TestKit的测试中，消息发送者为 `testActor` ，只需要在你的测试代码混入 `ImplicitSender` 。

```
1. class MySpec(_system: ActorSystem) extends TestKit(_system) with
   ImplicitSender
2.   with WordSpecLike with Matchers with BeforeAndAfterAll {
```

使用多个探针 Actor

如果待测的actor会发送多个消息到不同的目标，在使用 `TestKit` 时可能会难以分辨到达 `testActor` 的消息流。另一种方法是用它来创建简单的探针actor，将它们插入到消息流中。为了让这种方法更加强大和方便，我们提供了一个具体实现，称为 `TestProbe` 。它的功能可以用下面的小例子说明：

```
1. import scala.concurrent.duration._
2. import akka.actor._
3. import scala.concurrent.Future
4.
```

```
5.  class MyDoubleEcho extends Actor {
6.    var dest1: ActorRef = _
7.    var dest2: ActorRef = _
8.    def receive = {
9.      case (d1: ActorRef, d2: ActorRef) =>
10.       dest1 = d1
11.       dest2 = d2
12.     case x =>
13.       dest1 ! x
14.       dest2 ! x
15.   }
16. }
```

```
1.  val probe1 = TestProbe()
2.  val probe2 = TestProbe()
3.  val actor = system.actorOf(Props[MyDoubleEcho])
4.  actor ! ((probe1.ref, probe2.ref))
5.  actor ! "hello"
6.  probe1.expectMsg(500 millis, "hello")
7.  probe2.expectMsg(500 millis, "hello")
```

这里我们用 `MyDoubleEcho` 来模拟一个待测系统，它会将输入镜像为两个输出。关联两个测试探针来进行（最简单）行为的确认。还有一个例子是两个相互协作的 actor A，B， A 发送消息给 B。为了确认这个消息流，可以插入一个 `TestProbe` 作为A的目标，使用转发功能或下文中的自动导向功能在测试上下文中包含一个真实的B。

还可以为探针配备自定义的断言来使测试代码更简洁清晰：

```
1.  ase class Update(id: Int, value: String)
2.
3.  val probe = new TestProbe(system) {
4.    def expectUpdate(x: Int) = {
5.      expectMsgPF() {
6.        case Update(id, _) if id == x => true
7.      }
```

```
8.       sender() ! "ACK"
9.     }
10.  }
```

这里你拥有完全的灵活性，可以将 `TestKit` 提供的工具与你自己的检测代码混合和匹配，并为它取一个有意义的名字。在实际开发中你的代码很可能比上面的示例要复杂；要充分利用工具的力量！

> 警告
>
> 任何从 *`TestProbe`* 发送到另一个运行在 *`CallingThreadDispatcher`* 上的*actor*的消息，如果另一个*actor*也可能会向此探测器发送消息，则存在死锁的可能性。 *`TestProbe.watch`* 和 *`TestProbe.unwatch`* 的实现也将向监控者发送消息，这意味着尝试监视它是危险的，例如从 *`TestProbe`* 监视 *`TestActorRef`* 。

从探针观察其它actor

`TestProbe` 可以将自己注册为任意其他actor的DeathWatch:

```
1.  val probe = TestProbe()
2.  probe watch target
3.  target ! PoisonPill
4.  probe.expectTerminated(target)
```

对探针收到的消息进行应答

探针在可能的条件下，会记录通讯通道以便进行应答:

```
1.  val probe = TestProbe()
2.  val future = probe.ref ? "hello"
3.  probe.expectMsg(0 millis, "hello") // TestActor runs on
    CallingThreadDispatcher
4.  probe.reply("world")
5.  assert(future.isCompleted && future.value ==
    Some(Success("world")))
```

对探针收到的消息进行转发

假定一个象征性的actor网络中某目标 actor `dest` 从 actor `source` 收到一条消息。如果你安排消息使其先发往 `TestProbe` `probe`，你可以在保持网络功能的同时对消息流的容量和时限进行断言：

```
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(classOf[Source], probe.ref))
val dest = system.actorOf(Props[Destination])
source ! "start"
probe.expectMsg("work")
probe.forward(dest)
```

目标 `dest` actor 将收到同样的消息，就象没有插入探针一样。

自动导向

将收到的消息放进队列以便以后处理，这种方法不错，但要保持测试运行并对其运行过程进行跟踪，你也可以为参与测试的探针（事实上是任何 `TestKit` ）安装一个 `AutoPilot` （自动导向）。自动导向在消息进入检查队列之前启动。以下代码可以用来转发消息，例如 `A --> Probe --> B` ，只要满足一定的协议。

```
val probe = TestProbe()
```

```
2.  probe.setAutoPilot(new TestActor.AutoPilot {
3.    def run(sender: ActorRef, msg: Any): Option[TestActor.AutoPilot]
    =
4.      msg match {
5.        case "stop" ⇒ None
6.        case x      ⇒ testActor.tell(x, sender); Some(this)
7.      }
8.  })
```

`run` 方法必须返回 `auto-pilot` 供下一条消息使用，它可以是 `KeepRunning` 来保存当前值，或者是 `NoAutoPilot` 来终止自动导向。

小心定时器断言

在使用测试探针时， `within` 块的行为可能会不那么直观：你需要记住 上文 所描述的nicely scoped期限仅对每一个探针的局部作用域有效。因此，探针不会响应别的探针的期限，也不响应包含它的 `TestKit` 实例的期限：

```
1.  val probe = TestProbe()
2.  within(1 second) {
3.    probe.expectMsg("hello")
4.  }
```

这里 `expectMsg` 调用将使用默认超时。

CallingThreadDispatcher

如上文所述， `CallingThreadDispatcher` 在单元测试中非常重要，但最初它出现是为了在出错的时候能够生成连续的stacktrace。由于这个特殊的派发器一般地将任何消息直接运行在当前线程中，所以只要所有的actor都是在这个派发器上运行，消息处理的完整历史信息在调用堆栈上就有记录。

如何使用它

只要象平常一样设置派发器:

```
1. import akka.testkit.CallingThreadDispatcher
2. val ref =
   system.actorOf(Props[MyActor].withDispatcher(CallingThreadDispatcher.
```

它是如何运作的

在被调用时， `CallingThreadDispatcher` 会检查接收消息的actor是否已经在当前线程中了。这种情况的最简单的例子是actor向自己发送消息。这时，不能马上对它进行处理，因为这违背了actor模型，于是这个消息被放进队列，直到actor的当前消息被处理完毕；因此，新消息会在调用的线程上被处理，只是在actor完成其先前的工作之后。在别的情况下，消息会在当前线程中立即得到处理。通过这个派发器规划的Future也会立即执行。

这种工作方式使 `CallingThreadDispatcher` 象一个为永远不会因为外部事件而阻塞的actor所设计的通用派发器。

在有多个线程的情况下，有可能同时存在两个使用这个派发器的actor在不同线程中收到消息。此时，它们会立即在自己的线程中被执行，并竞争actor锁，竞争失败的那个必须等待。 这样我们保持了actor模型，但由于使用了受限的调度我们损失了一些并发性。从这个意义上说，它等同于使用传统的基于互斥的并发。

另一个困难是正确地处理挂起和继续：当actor被挂起时，后续的消息将被放进一个thread-local的队列中（和正常情况下使用的队列是同一个）。但是对 `resume` 的调用，是由一个特定的线程执行的，系统中所有其它的线程很可能不会运行这个特定的actor，这会导致thread-local队列无法被它们的本地线程清空。于是，调用 `resume` 的线程

会从所有线程收集所有当前在队列中的消息到自己的队列中，然后处理它们。

局限性

> 警告
>
> 在 `CallingThreadDispatcher` 被用作顶级*actor*，但没有通过 `TestActorRef` 的情况下，则存在一个时间窗，在此期间*actor*会等待*user* 守护者*actor*的构造。在此期间发送到这个*actor*的消息会被加入队列，然后在守护者的线程，而不是调用者的线程上执行。要避免此问题，请使用 `TestActorRef` 。

如果一个actor发送完消息后由于某种原因（通常是被发完消息后的调用actor所影响）阻塞了，此时若使用这个派发器，显然将导致死锁。这在使用基于 `CountDownLatch` 同步actor测试中是很常见的情景：

```
1.     val latch = new CountDownLatch(1)
2.     actor ! startWorkAfter(latch)   // actor will call latch.await()
   before proceeding
3.     doSomeSetupStuff()
4.     latch.countDown()
```

这个例子将无限挂起，消息处理到达第二行而永远到不了第四行，而只有在第四行才能在一个普通的派发器上取消它的阻塞。

所以要记住 `CallingThreadDispatcher` 并不是普通派发器的通用替代品。而另一方面在它上面运行你的actor网络测试会非常有用，因为如果它在机率特别高的条件下都能不死锁，那么在生产环境中也不会。

> 警告
>
> 上面这句话很遗憾并不是一个有力的保证，因为你的代码运行在不同的派发器上时可能直接或间接地改变它的行为。 如果你想要寻找帮助你*debug*死锁的工具， `CallingThreadDispatcher` 在有些错误场合下可能会有用，但要记住它既可能给出错误的正面结果也可能给出错误的负面结果。

线程中断

如果 `CallingThreadDispatcher` 看到当前线程在消息处理返回时已设置其 `isInterrupted()` 标志，它将在完成所有其处理（即，如上文所述的需要处理的所有消息，都会在这种情况发生之前处理）后抛出 `InterruptedException` 异常。正如 `tell` 由于其契约不能抛出异常，该异常将然后被捕获并记录日志，并且在该线程中断状态将再次设置。

如果在消息处理过程中抛出 `InterruptedException` 异常，则它将被 `CallingThreadDispatcher` 的消息处理循环捕获，然后将设置该线程 `interrupted` 标志，并且处理将继续正常进行。

> 注意
>
> 这两个段落的总结是，如果当前线程在 *`CallingThreadDispatcher`* 下工作时中断，则将导致当消息发送返回时 *`isInterrupted`* 标志被置为 *`true`* ，并不抛出 *`InterruptedException`* 异常。

好处

综上所述，以下是 `CallingThreadDispatcher` 能够提供的特性：

- 确定地执行单线程测试，同时保持几乎所有的actor语义
- 在异常stacktrace中记录从失败点开始的完整的消息处理历史
- 排除某些类型的死锁场景

跟踪Actor调用" class="reference-link">跟踪Actor调用

到目前为止所有的测试工具都针对系统的行为构造断言。如果测试失败，通常是由你来查找原因，进行修改并进行下一轮验证测试。这个过程既有debugger支持，又有日志支持，这里Akka工具箱提供以下选项：

- 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*的消息调用记录日志

这是通过在配置文件里添加设置项来打开 — 即 `akka.actor.debug.receive` — 它使得 `loggable` 语句被应用在actor的 `receive` 函数上：

```
1. import akka.event.LoggingReceive
2. def receive = LoggingReceive {
3.   case msg => // Do something ...
4. }
5. def otherState: Receive = LoggingReceive.withLabel("other") {
6.   case msg => // Do something else ...
7. }
```

如果在配置文件中没有上面给出的配置，这个方法将直接移交给给定的未被修改的 `Receive` 函数，也就是说如果不打开，就没有运行时开销。

这个日志功能是与指定的局部标记绑定的，因为将其一致地应用于所有的actor一般不是你所需要的，如果被用于事件主线日志监听器，它还可能导致无限循环。

- 对特殊的消息记录日志

Actor会自动处理某些特殊消息，例如 `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，你可以用以下配置片段打开actor活动的完整日志：

```
1.  akka {
2.    loglevel = DEBUG
3.    actor {
4.      debug {
5.        receive = on
6.        autoreceive = on
7.        lifecycle = on
8.      }
9.    }
10. }
```

不同的测试框架

Akka的测试套件是使用ScalaTest，并参照文档中的示例编写的。然而，TestKit和其工具包并不依赖于这一框架，实际上你可以使用最适合你开发风格的测试框架。

本节包含使用其他框架时的已知陷阱，这不是详尽的，并不意味着需要认证或使用特殊支持。

当你需要它是一个特质

如果由于某种原因是， `TestKit` 是一个具体的类，而不是特质，因此测试工具无法继承它，还有 `TestKitBase` ：

```
1.  import akka.testkit.TestKitBase
2.
3.  class MyTest extends TestKitBase {
4.    implicit lazy val system = ActorSystem()
5.
```

```
6.    // put your test code here ...
7.
8.    shutdown(system)
9.  }
```

`implicit lazy val system` 必须这样声明（你当然可以根据需要将参数传递到actor系统工厂中）因为特质 `TestKitBase` 在其构造过程中需要system。

> 警告
>
> 不提倡使用这个特质，由于在未来二进制向后兼容性可能会成为问题，所以使用它须自行承担风险。

Specs2

一些 `Specs2` 用户贡献了如何变通解决可能出现的一些冲突的示例：

- 在TestKit中混入 `org.specs2.mutable.Specification` 将导致涉及 `end` 方法的名称冲突（它在TestKit中是私有变量，而在Specification中是抽象方法）；如果第一次混入测试工具包，代码可以编译，但可能在运行时失败。变通方法—实际上也有利于第三点— 是将 `org.specs2.specification.Scope` 和TestKit一起应用。
- Specification特质提供了一个 `Duration` DSL 的部分，使用了与 `scala.concurrent.duration.Duration` 具有相同名称的方法，如果引入 `scala.concurrent.duration._` 会导致含糊不清的隐式值。有两个变通：
    - 要么使用Specification的Duration变体，提供到Akka Duration的隐式转换。这种转换不由Akka提供，因为这将意味着我们的 JAR 文件将依赖于 Specs2，为实现这样一个小特性不合理。

    - 或混入`org.specs2.time.NoTimeConversions`到Specification中。

- Specifications默认是并发执行的，因此执行写入测试或者`sequential`关键字时需要谨慎一些。

配置

TestKit模块的几个配置属性，请参阅配置参考。

TestKit实例

TestKit实例(Scala)

这是Ray Roestenburg 在 他的博客 中的示例代码，作了改动以兼容 Akka 2.x。

```
1.  import scala.util.Random
2.
3.  import org.scalatest.BeforeAndAfterAll
4.  import org.scalatest.WordSpecLike
5.  import org.scalatest.Matchers
6.
7.  import com.typesafe.config.ConfigFactory
8.
9.  import akka.actor.Actor
10. import akka.actor.ActorRef
11. import akka.actor.ActorSystem
12. import akka.actor.Props
13. import akka.testkit.{ TestActors, DefaultTimeout, ImplicitSender,
    TestKit }
14. import scala.concurrent.duration._
15. import scala.collection.immutable
16.
17. /**
18.  * a Test to show some TestKit examples
19.  */
20. class TestKitUsageSpec
21.   extends TestKit(ActorSystem("TestKitUsageSpec",
22.     ConfigFactory.parseString(TestKitUsageSpec.config)))
23.   with DefaultTimeout with ImplicitSender
24.   with WordSpecLike with Matchers with BeforeAndAfterAll {
25.   import TestKitUsageSpec._
26.
```

```
27.   val echoRef = system.actorOf(TestActors.echoActorProps)
28.   val forwardRef = system.actorOf(Props(classOf[ForwardingActor],
    testActor))
29.   val filterRef = system.actorOf(Props(classOf[FilteringActor],
    testActor))
30.   val randomHead = Random.nextInt(6)
31.   val randomTail = Random.nextInt(10)
32.   val headList = immutable.Seq().padTo(randomHead, "0")
33.   val tailList = immutable.Seq().padTo(randomTail, "1")
34.   val seqRef =
35.     system.actorOf(Props(classOf[SequencingActor], testActor,
    headList, tailList))
36.
37.   override def afterAll {
38.     shutdown()
39.   }
40.
41.   "An EchoActor" should {
42.     "Respond with the same message it receives" in {
43.       within(500 millis) {
44.         echoRef ! "test"
45.         expectMsg("test")
46.       }
47.     }
48.   }
49.   "A ForwardingActor" should {
50.     "Forward a message it receives" in {
51.       within(500 millis) {
52.         forwardRef ! "test"
53.         expectMsg("test")
54.       }
55.     }
56.   }
57.   "A FilteringActor" should {
58.     "Filter all messages, except expected messagetypes it receives"
    in {
59.       var messages = Seq[String]()
60.       within(500 millis) {
```

```
61.         filterRef ! "test"
62.         expectMsg("test")
63.         filterRef ! 1
64.         expectNoMsg
65.         filterRef ! "some"
66.         filterRef ! "more"
67.         filterRef ! 1
68.         filterRef ! "text"
69.         filterRef ! 1
70.
71.         receiveWhile(500 millis) {
72.           case msg: String => messages = msg +: messages
73.         }
74.       }
75.       messages.length should be(3)
76.       messages.reverse should be(Seq("some", "more", "text"))
77.     }
78.   }
79.   "A SequencingActor" should {
80.     "receive an interesting message at some point " in {
81.       within(500 millis) {
82.         ignoreMsg {
83.           case msg: String => msg != "something"
84.         }
85.         seqRef ! "something"
86.         expectMsg("something")
87.         ignoreMsg {
88.           case msg: String => msg == "1"
89.         }
90.         expectNoMsg
91.         ignoreNoMsg
92.       }
93.     }
94.   }
95. }
96.
97. object TestKitUsageSpec {
98.   // Define your test specific configuration here
```

```
 99.   val config = """
100.     akka {
101.       loglevel = "WARNING"
102.     }
103.     """
104.
105.   /**
106.    * An Actor that forwards every message to a next Actor
107.    */
108.   class ForwardingActor(next: ActorRef) extends Actor {
109.     def receive = {
110.       case msg => next ! msg
111.     }
112.   }
113.
114.   /**
115.    * An Actor that only forwards certain messages to a next Actor
116.    */
117.   class FilteringActor(next: ActorRef) extends Actor {
118.     def receive = {
119.       case msg: String => next ! msg
120.       case _           => None
121.     }
122.   }
123.
124.   /**
125.    * An actor that sends a sequence of messages with a random head
    list, an
126.    * interesting value and a random tail list. The idea is that you
    would
127.    * like to test that the interesting value is received and that
    you cant
128.    * be bothered with the rest
129.    */
130.   class SequencingActor(next: ActorRef, head:
    immutable.Seq[String],
131.                         tail: immutable.Seq[String]) extends Actor
    {
```

```
132.      def receive = {
133.        case msg => {
134.          head foreach { next ! _ }
135.          next ! msg
136.          tail foreach { next ! _ }
137.        }
138.      }
139.    }
140.  }
```

Actor DSL

Actor DSL

The Actor DSL

简单的actor——例如一次性worker甚或至在REPL中尝试的事物——可以更简洁地使用 `Act` 特质创建。支持的基础设施通过以下导入绑定：

```
1. import akka.actor.ActorDSL._
2. import akka.actor.ActorSystem
3. 
4. implicit val system = ActorSystem("demo")
```

在这一节的所有代码示例都假定有此导入。下面的所有示例中隐式actor系统都作为 `ActorRefFactory` 提供服务。要定义一个简单的actor，以下代码就足够了：

```
1. val a = actor(new Act {
2.   become {
3.     case "hello" ⇒ sender() ! "hi"
4.   }
5. })
```

在这里， `actor` 方法，根据其调用的上下文，取代了 `system.actorOf` 或 `context.actorOf` ：它需要一个隐式的 `ActorRefFactory` ，其中在

actor内可以通过以下方式获取 `implicit val context: ActorContext` 。在actor外，你不得不要声明隐式的 `ActorSystem` ，或者你可以显式地提供工厂（具体参见下文）。

两种发起 `context.become` （更换或添加新的行为）的可能方式是分别提供的，从而支持不凌乱的嵌套接收标记语法：

```
1.  val a = actor(new Act {
2.    become { // this will replace the initial (empty) behavior
3.      case "info" ⇒ sender() ! "A"
4.      case "switch" ⇒
5.        becomeStacked { // this will stack upon the "A" behavior
6.          case "info"   ⇒ sender() ! "B"
7.          case "switch" ⇒ unbecome() // return to the "A" behavior
8.        }
9.      case "lobotomize" ⇒ unbecome() // OH NOES: Actor.emptyBehavior
10.   }
11. })
```

请注意，调用 `unbecome` 比 `becomeStacked` 次数多将导致原始行为被安装，对 `Act` 特质来说是空行为（外部 `become` 只是在构造过程中替换它）。

生命周期管理

生命周期挂钩也可以暴露为 DSL 元素使用 （见启动Hook和终止Hook），在那里如下所示的调用方法可以取代各自挂钩的内容：

```
1.  val a = actor(new Act {
2.    whenStarting { testActor ! "started" }
3.    whenStopping { testActor ! "stopped" }
4.  })
```

如果actor的逻辑生命周期匹配重新启动周期（即 `whenStopping` 在重新启动之前执行，并且 `whenStarting` 在重启之后执行），上面的代

码就足够了。如果这不是所期望的，可以使用下面的两个挂钩（请参阅重启Hook）：

```
1. val a = actor(new Act {
2.   become {
3.     case "die" ⇒ throw new Exception
4.   }
5.   whenFailing { case m @ (cause, msg) ⇒ testActor ! m }
6.   whenRestarted { cause ⇒ testActor ! cause }
7. })
```

另外还可以创建嵌套actors，即孙子actor，像这样：

```
1. // here we pass in the ActorRefFactory explicitly as an example
2. val a = actor(system, "fred")(new Act {
3.   val b = actor("barney")(new Act {
4.     whenStarting { context.parent ! ("hello from " + self.path) }
5.   })
6.   become {
7.     case x ⇒ testActor ! x
8.   }
9. })
```

> 注意
>
> 在某些情况下必须显式传递 *ActorRefFactory* 给 *actor()* 方法（当编译器告诉你出现了模糊蕴涵*(implicits)*时，你会发现的）。

孙子actor会被子actor监管；此外这类关系的监管策略也可以使用DSL 元素进行配置（监管指令是 `Act` 特质的一部分）：

```
1. superviseWith(OneForOneStrategy() {
2.   case e: Exception if e.getMessage == "hello" ⇒ Stop
3.   case _: Exception                             ⇒ Resume
4. })
```

有 `Stash` 的actor

最后并且很重要的一点是：还有一点像魔法一样方便的内置，可以检测静态给定的actor子类型的运行时类，是不是通过 `Stash` 特质扩展 `RequiresMessageQueue` 特质（这是一个复杂表述，即 `new Act with Stash` 不能工作，因为它的运行时类型被擦除为 `Act` 的匿名子类型）。目的是为自动根据 `Stash` 的需要使用适当的基于deque的邮箱类型。如果你想要使用这个魔法，只需扩展 `ActWithStash`：

```
1. val a = actor(new ActWithStash {
2.   become {
3.     case 1 ⇒ stash()
4.     case 2 ⇒
5.       testActor ! 2; unstashAll(); becomeStacked {
6.         case 1 ⇒ testActor ! 1; unbecome()
7.       }
8.   }
9. })
```

Futures与Agents

Futures与Agents

Futures

Futures

> 注：本节未经校验，如有问题欢迎提*issue*

简介

在 Akka 中，一个Future是用来获取某个并发操作结果的数据结构。这个结果可以以同步（阻塞）或异步（非阻塞）的方式访问。

执行上下文

为了运行回调和操作，Futures 需要有一个 `ExecutionContext` ，它

与 `java.util.concurrent.Executor` 很相像. 如果你在作用域内有一个 `ActorSystem`，它会把自己的派发器用作 `ExecutionContext`，或者你也可以用 `ExecutionContext` 伴生对象提供的工厂方法来将 `Executors` 和 `ExecutorServices` 进行包装，或者甚至创建自己的实例。

```
1.  import scala.concurrent.{ ExecutionContext, Promise }
2.
3.  implicit val ec =
    ExecutionContext.fromExecutorService(yourExecutorServiceGoesHere)
4.
5.  // Do stuff with your brand new shiny ExecutionContext
6.  val f = Promise.successful("foo")
7.
8.  // Then shut your ExecutionContext down at some
9.  // appropriate place in your program/application
10. ec.shutdown()
```

在Actor中

每个actor都被配置为在 `MessageDispatcher` 上运行，且该调度器又被用作为 `ExecutionContext`。如果被actor调用的Future的性质匹配或兼容与那个actor的活动（例如，全CPU绑定，也没有延迟要求），那么它可能是最容易重用派发器，只需要通过导入 `context.dispatcher` 来运行Futures。

```
1.  class A extends Actor {
2.    import context.dispatcher
3.    val f = Future("hello")
4.    def receive = {
5.      // receive omitted ...
6.    }
7.  }
```

用于 Actor

通常有两种方法来从一个 `Actor` 获取回应：第一种是发送一个消息（ `actor ! msg` ，这种方法只在发送者是一个 `Actor` 时有效），第二种是通过一个 `Future` 。

使用 `Actor` 的 `?` 方法来发送消息会返回一个 `Future` . 要等待并获取结果的最简单方法是：

```
1. import scala.concurrent.Await
2. import akka.pattern.ask
3. import akka.util.Timeout
4. import scala.concurrent.duration._
5.
6. implicit val timeout = Timeout(5 seconds)
7. val future = actor ? msg // enabled by the “ask” import
8. val result = Await.result(future,
   timeout.duration).asInstanceOf[String]
```

这会导致当前线程阻塞，并等待 `Actor` 通过它的应答来 ‘完成’ `Future` 。但是阻塞会导致性能问题，所以是不推荐的. 导致阻塞的操作位于 `Await.result` 和 `Await.ready` 中，这样就方便定位阻塞的位置. 对阻塞方式的替代方法会在本文档中进一步讨论。还要注意 `Actor` 返回的 `Future` 的类型 是 `Future[Any]` ，这是因为 `Actor` 是动态的. 这也是为什么上例中使用了 `asInstanceOf` 。在使用非阻塞方式时，最好使用 `mapTo` 方法来将 `Future` 转换到期望的类型：

```
1. import scala.concurrent.Future
2. import akka.pattern.ask
3.
4. val future: Future[String] = ask(actor, msg).mapTo[String]
```

如果转换成功， `mapTo` 方法会返回一个包含结果的新的 `Future` ，如果不成功，则返回 `ClassCastException` . 对 `Exception` 的处理将在本文档进一步讨论。

要把 `Future` 的结果发送给一个 `Actor` ，你可以使用 `pipe` 构建：

```
1. import akka.pattern.pipe
2. future pipeTo actor
```

直接使用

Akka中的一个常见用例是在不需要额外使用 `Actor` 工具的情况下并发地执行计算. 如果你发现你只是为了并行地执行一个计算而创建了一堆 `Actor` ，下面是一种更好（也更快）的方法：

```
1. import scala.concurrent.Await
2. import scala.concurrent.Future
3. import scala.concurrent.duration._
4.
5. val future = Future {
6.   "Hello" + "World"
7. }
8. future foreach println
```

在上面的代码中，被传递给 `Future` 的代码块会被缺省的 `Dispatcher` 执行，代码块的返回结果会被用来完成 `Future` (在这个例子中，结果是一个字符串：“HelloWorld”). 与从 `Actor` 返回的 `Future` 不同，这个 `Future` 拥有合适的类型，我们还避免了管理 `Actor` 的开销。

你还可以用 `Future` 伴生对象创建一个已经完成的 `Future` ，它可以是成功的：

```
1. val future = Future.successful("Yay!")
```

或是失败的:

```
1. val otherFuture = Future.failed[String](new
   IllegalArgumentException("Bang!"))
```

也可以创建一个空的 `Promise`，以后填充它，并包含一个相应的 `Future`:

```
1. val promise = Promise[String]()
2. val theFuture = promise.future
3. promise.success("hello")
```

函数式 Future

Scala 的 `Future` 有一些 monadic 方法，与Scala集合所使用的方法非常相似. 这使你可以构造出可以传递结果的 ‘管道’ 或 ‘数据流’ 。

Future 是 Monad

让 `Future` 以函数式风格工作的第一个方法是 `map` . 它需要一个 `Function` 来对 `Future` 的结果进行处理，返回一个新的结果。 `map` 方法的返回值是包含新结果的另一个 `Future` :

```
1. val f1 = Future {
2.   "Hello" + "World"
3. }
4. val f2 = f1 map { x =>
5.   x.length
6. }
7. f2 foreach println
```

这个例子中我们在 `Future` 内部连接两个字符串。我们没有等待这个 `Future` 结束，而是使用 `map` 方法来将计算字符串长度的函数应用于它. 现在我们有了第二个 `Future` ，它的最终结果是一个 `Int` . 当先前的 `Future` 完成时，它会应用我们的函数并用其结果来完成第二个 `Future` 。最终我们得到的结果是 10. 先前的 `Future` 仍然持有字符串“HelloWorld”，而不受 `map` 的影响。

如果我们只是修改一个 `Future` ， `map` 方法就够用了。但如果有2个以上 `Future` 时， `map` 无法将他们组合到一起：

```
1.  val f1 = Future {
2.    "Hello" + "World"
3.  }
4.  val f2 = Future.successful(3)
5.  val f3 = f1 map { x =>
6.    f2 map { y =>
7.      x.length * y
8.    }
9.  }
10. f3 foreach println
```

`f3` 的类型是 `Future[Future[Int]]` 而不是我们所期望的 `Future[Int]` . 这时我们需要使用 `flatMap` 方法：

```
1.  val f1 = Future {
2.    "Hello" + "World"
3.  }
4.  val f2 = Future.successful(3)
5.  val f3 = f1 flatMap { x =>
6.    f2 map { y =>
7.      x.length * y
8.    }
9.  }
10. f3 foreach println
```

使用嵌套的 `map` 或 `flatmap` 组合子来组合 `Future`，有时会变得非常复杂和难以阅读，这时使用 Scala 的 ‘for comprehensions’ 一般会生成可读性更好的代码。见下一部分的示例。

如果你需要进行条件筛选外延，可以使用 `filter`：

```
1.  val future1 = Future.successful(4)
2.  val future2 = future1.filter(_ % 2 == 0)
3.
4.  future2 foreach println
5.
6.  val failedFilter = future1.filter(_ % 2 == 1).recover {
7.    // When filter fails, it will have a
    java.util.NoSuchElementException
8.    case m: NoSuchElementException => 0
9.  }
10.
11. failedFilter foreach println
```

For Comprehensions

由于 `Future` 拥有 `map`， `filter` 和 `flatMap` 方法，它可以方便地用于 ‘for comprehension’：

```
1.  val f = for {
2.    a <- Future(10 / 2) // 10 / 2 = 5
3.    b <- Future(a + 1) //  5 + 1 = 6
4.    c <- Future(a - 1) //  5 - 1 = 4
5.    if c > 3 // Future.filter
6.  } yield b * c //  6 * 4 = 24
7.
8.  // Note that the execution of futures a, b, and c
9.  // are not done in parallel.
10.
11. f foreach println
```

这样写代码的时候需要记住的是：虽然看上去上例的部分代码可以并发地运行，for comprehension的每一步实际是顺序执行的。每一步是在单独的线程中运行的，但是相较于将所有的计算在一个单独的 `Future` 中运行并没有太大好处。只有先创建 `Future` ，然后对其进行组合的情况下才能真正得到好处。

组合 Futures

上例中的for comprehension 是对 `Future` 进行组合的例子. 这种方法的常见用例是将多个 `Actor` 的回应组合成一个单独的计算而不用调用 `Await.result` 或 `Await.ready` 来阻塞地获得每一个结果. 先看看使用 `Await.result` 的例子：

```
1.  val f1 = ask(actor1, msg1)
2.  val f2 = ask(actor2, msg2)
3.
4.  val a = Await.result(f1, 3 seconds).asInstanceOf[Int]
5.  val b = Await.result(f2, 3 seconds).asInstanceOf[Int]
6.
7.  val f3 = ask(actor3, (a + b))
8.
9.  val result = Await.result(f3, 3 seconds).asInstanceOf[Int]
```

> 警告
>
> *`Await.result`* 和 *`Await.ready`* 为必须阻塞的特殊情况提供的，一个很好的经验法则是仅在你知道为什么你必须阻塞的情况下使用它们。对于所有其他情况，使用如下所述的异步组合。

这里我们等待前2个 `Actor` 的结果然后将其发送给第三个 `Actor` . 我们调用了3次 `Await.result` ，导致我们的程序在获得最终结果前阻塞了3次。现在跟下例比较：

```
1.  val f1 = ask(actor1, msg1)
2.  val f2 = ask(actor2, msg2)
```

```
3.
4.  val f3 = for {
5.    a <- f1.mapTo[Int]
6.    b <- f2.mapTo[Int]
7.    c <- ask(actor3, (a + b)).mapTo[Int]
8.  } yield c
9.
10. f3 foreach println
```

这里我们有两个actor各自处理自己的一条消息。一旦这2个结果可用了（注意我们并没有阻塞地等待这些结果！），它们会被加起来发送给第三个 `Actor` ，这第三个actor回应一个字符串，我们把它赋值给‘result’。

上面的方法对已知给定Actor数量的时候就足够了，但是当Actor数量较大时就显得比较笨重。 `sequence` 和 `traverse` 两个辅助方法可以帮助处理更复杂的情况。这两个方法都是用来将 `T[Future[A]]` 转换为 `Future[T[A]]` （其中 `T` 是 `Traversable` 子类）. 例如:

```
1.  // oddActor returns odd numbers sequentially from 1 as a
    List[Future[Int]]
2.  val listOfFutures = List.fill(100)(akka.pattern.ask(oddActor,
    GetNext).mapTo[Int])
3.
4.  // now we have a Future[List[Int]]
5.  val futureList = Future.sequence(listOfFutures)
6.
7.  // Find the sum of the odd numbers
8.  val oddSum = futureList.map(_.sum)
9.  oddSum foreach println
```

现在来解释一下， `Future.sequence` 将输入的 `List[Future[Int]]` 转换为 `Future[List[Int]]` . 这样我们就可以将 `map` 直接作用于 `List[Int]` ，从而得到 `List` 的总和。

`traverse` 方法与 `sequence` 类似，但它以 `T[A]` 和 `A => Future[B]` 函数为参数返回一个 `Future[T[B]]` ，这里的 `T` 同样也是 `Traversable` 的子类. 例如，用 `traverse` 来计算前100个奇数的和：

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

其结果与这个例子是一样的：

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

但是用 `traverse` 也许会快一些，因为它不用创建一个 `List[Future[Int]]` 的临时变量。

然后我们有一个方法 `fold` ，它的参数包括一个初始值 ，一个 `Future` 序列和一个作用于初始值和 `Future` 类型返回与初始值相同类型的函数，它将这个函数异步地应用于future序列的所有元素，它的执行将在最后一个Future完成之后开始。

```
1. // Create a sequence of Futures
2. val futures = for (i <- 1 to 1000) yield Future(i * 2)
3. val futureSum = Future.fold(futures)(0)(_ + _)
4. futureSum foreach println
```

就是这么简单！

如果传给 `fold` 的序列是空的，它将返回初始值，在上例中，这个值是0. 有时你没有一个初始值，而使用序列中第一个已完成的 `Future` 的值作为初始值，你可以使用 `reduce` ，它的用法是这样的：

```
// Create a sequence of Futures
val futures = for (i <- 1 to 1000) yield Future(i * 2)
val futureSum = Future.reduce(futures)(_ + _)
futureSum foreach println
```

与 `fold` 一样，它是在最后一个 `Future` 完成后异步执行的，你也可以对这个过程进行并行化：将future分成子序列分别进行reduce，然后对reduce的结果再次reduce。

回调

有时你只想要监听 `Future` 的完成事件，对其进行响应，不是创建新的 `Future`，而仅仅是产生副作用. Scala为这种情况准备了 `onComplete`， `onSuccess` 和 `onFailure`，其中后两者是第一项的特例。

```
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" =>
  //OHNOES! We are in deep trouble, do something!
  case e: Exception =>
  //Do something else
}
```

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

定义次序

由于回调的执行是无序的，而且可能是并发执行的，当你需要操作有序的时候代码行为往往很怪异。但有一个解决办法是使用 `andThen` 。它会为指定的回调创建一个新的 `Future` ，这个 `Future` 与原先的 `Future` 拥有相同的结果，这样就可以像下例一样定义次序：

```
1. val result = Future { loadPage(url) } andThen {
2.   case Failure(exception) => log(exception)
3. } andThen {
4.   case _ => watchSomeTV()
5. }
6. result foreach println
```

辅助方法

`Future````fallbackTo` 将两个 `Futures` 合并成一个新的 `Future` ，如果第一个 `Future` 失败了，它将持有第二个 `Future` 的成功值。

```
1. val future4 = future1 fallbackTo future2 fallbackTo future3
2. future4 foreach println
```

你也可以使用 `zip` 操作将两个 `Futures` 组合成一个新的持有二者成功结果的tuple元组的 `Future`

```
1. val future3 = future1 zip future2 map { case (a, b) => a + " " + b
   }
2. future3 foreach println
```

异常

由于 `Future` 的结果是与程序的其它部分并发生成的，因此异常需要作特殊的处理。不管是 `Actor` 或是派发器正在完成此 `Future` ，如果抛

出了 `Exception`， `Future` 将持有这个异常而不是一个有效的值. 如果 `Future` 持有 `Exception`，调用 `Await.result` 将导致此异常被再次抛出从而得到正确的处理。

通过返回一个不同的结果来处理 `Exception` 也是可能的. 这是使用 `recover` 方法实现的. 例如:

```
1.  val future = akka.pattern.ask(actor, msg1) recover {
2.    case e: ArithmeticException => 0
3.  }
4.  future foreach println
```

在这个例子中，如果actor回应了包含 `ArithmeticException` 的 `akka.actor.Status.Failure`，我们的 `Future` 将持有 `0` 作为结果. `recover` 方法与标准的 `try/catch` 块非常相似，可以用这种方式处理多种 `Exception`， 如果其中有没有提到的 `Exception`，这种异常将以“好像没有定义 `recover` 一样”的的方式来处理。

你也可以使用 `recoverWith` 方法，它和 `recover` 的关系就象 `flatMap` 与 `map` 的关系，用法如下:

```
1.  val future = akka.pattern.ask(actor, msg1) recoverWith {
2.    case e: ArithmeticException => Future.successful(0)
3.    case foo: IllegalArgumentException =>
4.      Future.failed[Int](new IllegalStateException("All br0ken!"))
5.  }
6.  future foreach println
```

After

`akka.pattern.after` 使得在给定超时后完成一个 `Future`，获取其值或异常比昂的很容易。

```
1. // TODO after is unfortunately shadowed by ScalaTest, fix as part
   of #3759
2. // import akka.pattern.after
3.
4. val delayed = akka.pattern.after(200 millis, using =
   system.scheduler)(Future.failed(
5.   new IllegalStateException("OHNOES")))
6. val future = Future { Thread.sleep(1000); "foo" }
7. val result = Future firstCompletedOf Seq(future, delayed)
```

Agents

Agents

> 注：本节未经校验，如有问题欢迎提*issue*

Akka中的Agent是受 Clojure agent启发的.

Agent 提供对独立位置的异步修改. Agent在其生命周期中绑定到一个单独的存储位置，对这个存储位置的数据的修改（到一个新的状态）仅允许作为一个操作的结果发生。 对其进行修改的操作是函数，该函数被异步地应用于Agent的状态，其返回值成为Agent的新状态。Agent的状态应该是不可变的。

虽然对Agent的修改是异步的，但是其状态总是可以随时被任何线程来获得(通过 `get` 或 `apply`)而不需要发送消息。

Agent是响应式的（reactive）. 对所有agent的更新操作在一个 `ExecutionContext` 的不同线程中并发执行。在每一个时刻，每一个Agent最多只有一个 `send` 被执行. 从某个线程派发到agent上的操作的执行次序与其发送的次序一致，但有可能与从其它线程源派发来的操作交织在一起。

> 注意
>
> *Agent*对创建它们的节点是本地的。这意味着你一般不应包括它们在消息中，因为可能会被传递到远程*actor*或作为远程*actor*的构造函数参数；那些远程*Actor*不能读取或更新*Agent*。

创建Agent

创建Agent时，调用 `Agent(value)` ，传入它的初始值并提供一个隐式的 `ExecutionContext` 供其使用，在这些例子中我们将使用默认的全局量，不过你的方法可能不同(YMMV : Your Method May Vary):

```
1. import scala.concurrent.ExecutionContext.Implicits.global
2. import akka.agent.Agent
3. val agent = Agent(5)
```

读取 Agent 的值

Agent可以用括号调用来去引用（你可以获取一个Agent的值），像这样:

```
1. val result = agent()
```

或者使用 get 方法:

```
1. val result = agent.get
```

读取Agent的当前值不包括任何消息传递，并立即执行。所以说虽然Agent的更新的异步的，对它的状态的读取却是同步的。

更新 Agent (send & alter)

更新Agent有两种方法：send一个函数来转换当前的值，或直接send一个新值。Agent会自动异步地应用新的值或函数。更新是以一种“发射后不管”的方式完成的，唯一的保证是它会被应用。 至于什么时候应

用则没有保证，但是从同一个线程发到Agent的操作将被顺序应用。你通过调用 `send` 函数来应用一个值或函数。

```
// send a value, enqueues this change
// of the value of the Agent
agent send 7

// send a function, enqueues this change
// to the value of the Agent
agent send (_ + 1)
agent send (_ * 2)
```

你也可以在一个独立的线程中派发一个函数来改变其内部状态。这样将不使用响应式线程池，并可以被用于长时间运行或阻塞的操作。 相应的方法是 `sendOff` 。 派发器不管使用 `sendOff` 还是 `send` 都会顺序执行。

```
// the ExecutionContext you want to run the function on
implicit val ec = someExecutionContext()
// sendOff a function
agent sendOff longRunningOrBlockingFunction
```

所有的 `send` 都有一个对应的 `alter` 方法来返回一个 `Future` 。参考Futures来获取 `Future` 的更多信息。

```
// alter a value
val f1: Future[Int] = agent alter 7

// alter a function
val f2: Future[Int] = agent alter (_ + 1)
val f3: Future[Int] = agent alter (_ * 2)
```

```
// the ExecutionContext you want to run the function on
implicit val ec = someExecutionContext()
// alterOff a function
```

```
4. val f4: Future[Int] = agent alterOff longRunningOrBlockingFunction
```

等待Agent的返回值

也可以获得一个Agent值的 `Future` ，将在所有当前排队的更新请求都完成以后完成：

```
1. val future = agent.future
```

参考Futures来获取 `Future` 的更多信息。

Monadic 用法

Agent 也支持 monadic 操作, 这样你就可以用for-comprehensions对操作进行组合. 在 monadic 用法中, 旧的Agent不会变化，而是创建新的Agent。 所以老的值（Agents）仍像原来一样可用。这就是所谓的‘持久’.

monadic 用法示例:

```
1. import scala.concurrent.ExecutionContext.Implicits.global
2. val agent1 = Agent(3)
3. val agent2 = Agent(5)
4.
5. // uses foreach
6. for (value <- agent1)
7.   println(value)
8.
9. // uses map
10. val agent3 = for (value <- agent1) yield value + 1
11.
12. // or using map directly
13. val agent4 = agent1 map (_ + 1)
14.
15. // uses flatMap
```

```
16.  val agent5 = for {
17.    value1 <- agent1
18.    value2 <- agent2
19.  } yield value1 + value2
```

配置

有一些配置属性是针对Agent模块的，请参阅参考配置。

废弃的事务性Agent

Agent参与封闭 STM 事务是 2.3 废弃的功能。

如果Agent在一个封闭的事务中使用，然后它将参与该事务。如果你在一个事务内发送到Agent，然后对该Agent的派发将暂停直到该事务被提交，如果事务中止则丢弃该派发。下面是一个示例：

```
1.  import scala.concurrent.ExecutionContext.Implicits.global
2.  import akka.agent.Agent
3.  import scala.concurrent.duration._
4.  import scala.concurrent.stm._
5.
6.  def transfer(from: Agent[Int], to: Agent[Int], amount: Int):
    Boolean = {
7.    atomic { txn =>
8.      if (from.get < amount) false
9.      else {
10.       from send (_ - amount)
11.       to send (_ + amount)
12.       true
13.     }
14.   }
15. }
16.
17. val from = Agent(100)
18. val to = Agent(20)
```

```
19.  val ok = transfer(from, to, 50)
20.
21.  val fromValue = from.future // -> 50
22.  val toValue = to.future // -> 70
```

网络

网络

集群规格

集群规格

集群用法

集群用法

远程

远程

> 注: 本节未经校验, 如有问题欢迎提*issue*

要了解关于Akka的远程调用能力的简介请参阅位置透明性.

> 注意

> 正如那一章所解释的，*Akka remoting*是按照端到端（*peer-to-peer*）对等通信的方式设计的，并在建立客户端-服务器（*client-server*）模式时受到限制。特别是*Akka Remoting*除其他外，不能与网络地址转换（*Network Address Translation*）和负载均衡器（*Load Balancer*）一起工作。

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. 情确认你的项目中包括以下依赖:

```
1. "com.typesafe.akka" %% "akka-remote" % "2.3.6"
```

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:

```
1. akka {
2.   actor {
3.     provider = "akka.remote.RemoteActorRefProvider"
4.   }
5.   remote {
6.     enabled-transports = ["akka.remote.netty.tcp"]
7.     netty.tcp {
8.       hostname = "127.0.0.1"
9.       port = 2552
10.    }
11.  }
12. }
```

从上例中可以看到你开始时需要加入4个东西:

- 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; 这个主机名与传给远程系统的内容完全一样，用来标识这个系统，并为后续根据需要连接回这个系统时使用, 所以要把它设置成一个可到达的IP

地址或一个可以正确解析的域名来保证网络可访问性.

- 增加端口号—actor 系统监听的端口号，`0`表示让它自动选择

> 注意
>
> 端口号对相同机器上的*actor*系统必须是唯一的，即使*actor*系统具有不同的名称。这是因为每个*actor*系统有其自身网络子系统，来监听连接并处理消息，以免与其他*actor*系统干扰。

上例只是演示了要进行远程调用所需要添加的最小属性。所有的设置在远程调用配置一节中描述。

远程交互的类型

Akka 远程调用有两种方式:

- 查找 ： 使用`actorSelection(path)`在远程主机上查找一个actor
- 创建 ： 使用`actorOf(Props(...), actorName)`在远程主机上创建一个actor

下面章节将对这两种方法进行详细介绍.

查找远程 Actors

`actorSelection(path)`会获得远程结点上一个Actor的`ActorSelection`， 例如:

```
1.  val selection =
2.
    context.actorSelection("akka.tcp://actorSystemName@10.0.0.1:2552/user
```

可以看到以下模式被用来在远程结点上查找一个actor:

```
1.  akka.<protocol>://<actor system>@<hostname>:<port>/<actor path>
```

一旦得到了actor的selection，你就可以像与本地actor通讯一样与它进行通迅， 例如:

```
1.  selection ! "Pretty awesome feature"
```

要获得 `ActorSelection` 的 `ActorRef` 你需要发送一条消息到selection，然后使用actor答复中的 `sender` 引用。有一个内置的 `Identify` 消息所有Actor都会理解并自动回复一个包含 `ActorRef` 的 `ActorIdentity` 消息。这也可以通过 `ActorSelection` 的 `resolveOne` 方法实现，它返回一个包含匹配 `ActorRef` 的 `Future` 。

> 注意
>
> 要了解更多*actor*地址和路径的组成、使用的详细信息， 请参考 *Actor 引用，路径和地址*.

创建远程 Actor

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):

```
1.  akka {
2.    actor {
3.      deployment {
4.        /sampleActor {
5.          remote = "akka.tcp://sampleActorSystem@127.0.0.1:2553"
6.        }
7.      }
8.    }
9.  }
```

这个配置告知Akka当一个路径为 `/sampleActor` 的actor被创建时，即使用 `system.actorOf(Props(...), "sampleActor")` 时，要进行响应． 指定的actor不会被直接实例化， 而是远程actor系统的daemon会被要求

创建这个actor，本例中的远程actor系统是 `sampleActorSystem@127.0.0.1:2553` .

一旦配置了以上属性你就可以在代码中进行如下操作：

```
1. val actor = system.actorOf(Props[SampleActor], "sampleActor")
2. actor ! "Pretty slick"
```

actor类 `SampleActor` 必须在运行时可用，即，actor系统的classloader中必须有一个包含这个类的JAR包.

> 注意
>
> 当创建*actor*传递构造函数参数时，为了确保 *`Props`* 的序列化特性，不要是内部类作为工厂：它将天生地捕获其封闭对象的引用，而在大多数情况下对象的引用是不可序列化的。最好在*actor*类的伴生对象中创建工厂方法。
>
> 通过设置配置项目 *`akka.actor.serialize-creators=on`* ，所有*Props*的序列化都可以被测试。只有其 *`deploy`* 具有 *`LocalScope`* 的*Props*会被免除这一检查。
>
> 注意
>
> 你可以使用星号作为通配符匹配*actor*路径，因此你可以指定： *`/*/sampleActor`* ，并匹配该树形结构中那一级别上的所有 *`sampleActor`* 。你也能在最后一个位置使用通配符来匹配某一级别的所有*actor*： *`/someParent/*`* 。非通配符匹配相比之下总是有更高的优先级，所以： *`/foo/bar`* 被认为比 *`/foo/*`* 更具体，并且只有优先级最高的匹配才会被使用。请注意它不能用于部分匹配，像这样： *`/foo*/bar`* ， *`/f*o/bar`* 等。

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.

加入这些import：

```
1. import akka.actor.{ Props, Deploy, Address, AddressFromURIString }
2. import akka.remote.RemoteScope
```

和一个像这样的远程地址：

```
1.  val one = AddressFromURIString("akka.tcp://sys@host:1234")
2.  val two = Address("akka.tcp", "sys", "host", 1234) // this gives
    the same
```

你可以像这样建议系统在此远程结点上创建一个子actor：

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

生命周期和故障恢复模式

每个远程系统的链接可以在上面所示的四个状态之一。对一个给定 `Address` 的远程系统，在任何通信发生之前，其链接状态为 `Idle` 。第一次，一条消息试图发送到远程系统，或一个呼入连接被接受，则链接状态变为 `Active` ，表明两个系统有消息来发送或接收，并且目前没有发生失败。当通信发生故障和两个系统之间失去连接时，链接变为 `Gated` 。

在这个状态下，系统不会尝试连接到远程主机，并将丢弃所有出站消息。链接处于 `Gated` 状态的时间由设置 `akka.remote.retry-gate-closed-for` 控制： 这个时间过去后链接状态会重新变为 `Idle` 。 `Gate` 从某种意义上是单方面的，在 `Gate` 状态下的任何时候，一个入站连接被成功接受，它将自动转为 `Active` 并且通信会立即恢复。

面对因为参与系统的状态不一致导致的无法恢复的通信失败，远程系统变为 `Quarantined` 。与 `Gate` 不同，隔离是永久性的，并一直持续到其中一个系统重新启动。重新启动后通讯可以再度恢复，并且链接可以重新变为 `Active` 。

监视远程actor

监视一个远程actor与监视一个本地actor没有不同，如使用DeathWatch进行生命周期监控中所述。

> 警告
>
> 警告： 监视通过 *actorFor* 获取的 *ActorRef* 在失去连接时不会触发 *Terminated* 消息。 *actorFor* 是被 *actorSelection* 取代的废弃方法。应监视通过 *Identify* 和 *ActorIdentity* 获得的 *ActorRef* ，如通过*Actor Selection*定位*Actor*所描述的。

失效检测器

在底层，远程death watch使用心跳消息和一个失效检测器来对网络故障和JVM崩溃生成 `Terminated` 消息，并对被监视的actor优雅地终止。

心跳到达的时间是由Phi自增失效检测器的一个实现解释的。

对故障的怀疑级别由名为*phi*的值给定。Phi失效检测器的基本思想是在某个规模上描述*phi*值，来动态地调整以反映当前的网络状况。

*phi*值是这样计算的：

```
1.  phi = -log10(1 - F(timeSinceLastHeartbeat))
```

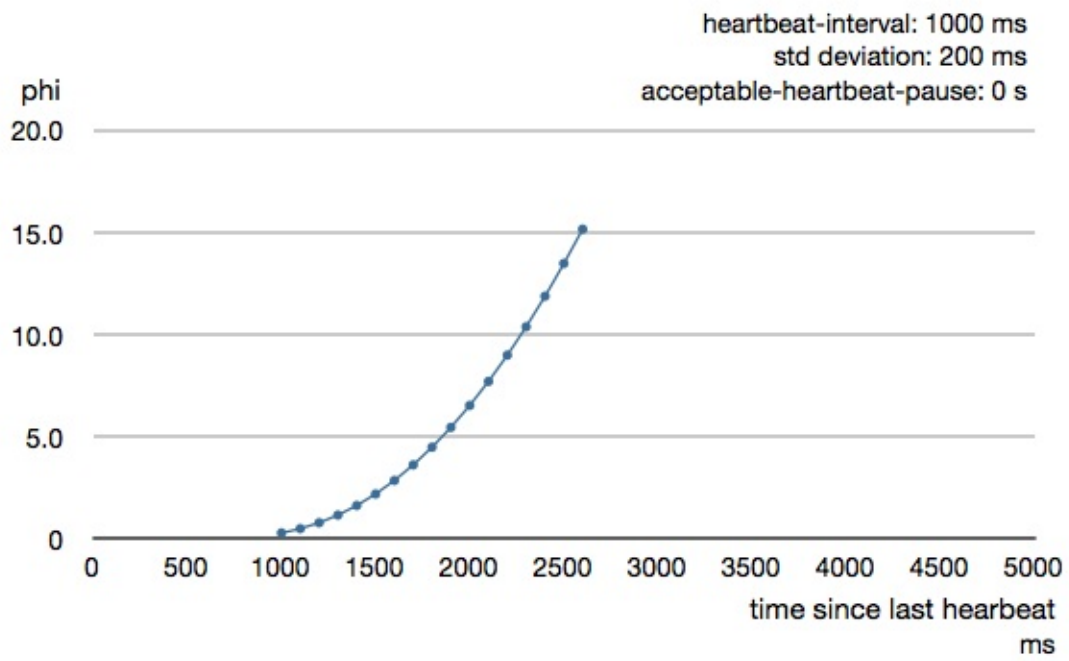
其中F是正态分布曲线的平均值和标准偏差的估计，从历史的心跳间隔到达次数的累积分布函数。

在远程配置中你可以调整 `akka.remote.watch-failure-detector.threshold` 来定义什么样的*phi*值被认为是一个失败。

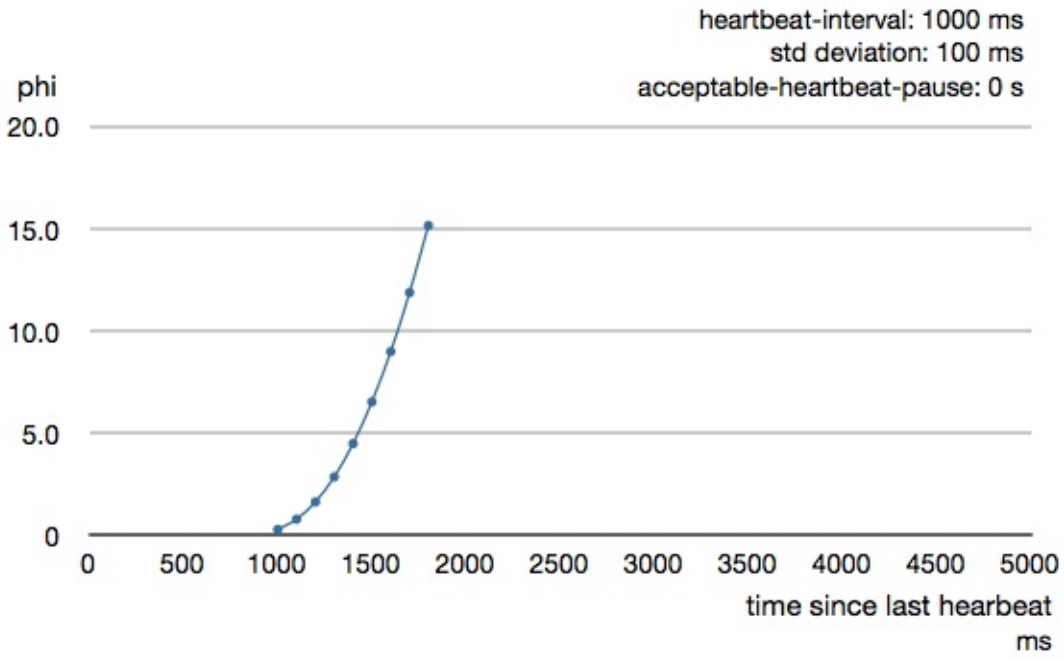
一个低的 `threshold` 容易产生许多假阳性反应，但可以确保一个真正崩溃发生时能快速检测到。相反，一个高 `threshold` 会生成更少的错误，但需要更多的时间来检测真正的崩溃。默认的 `threshold` 是10，它适

合大多数情况。但是在云环境中，如Amazon EC2，该值可增至12来匹配有时会发生在这类平台上的网络问题。

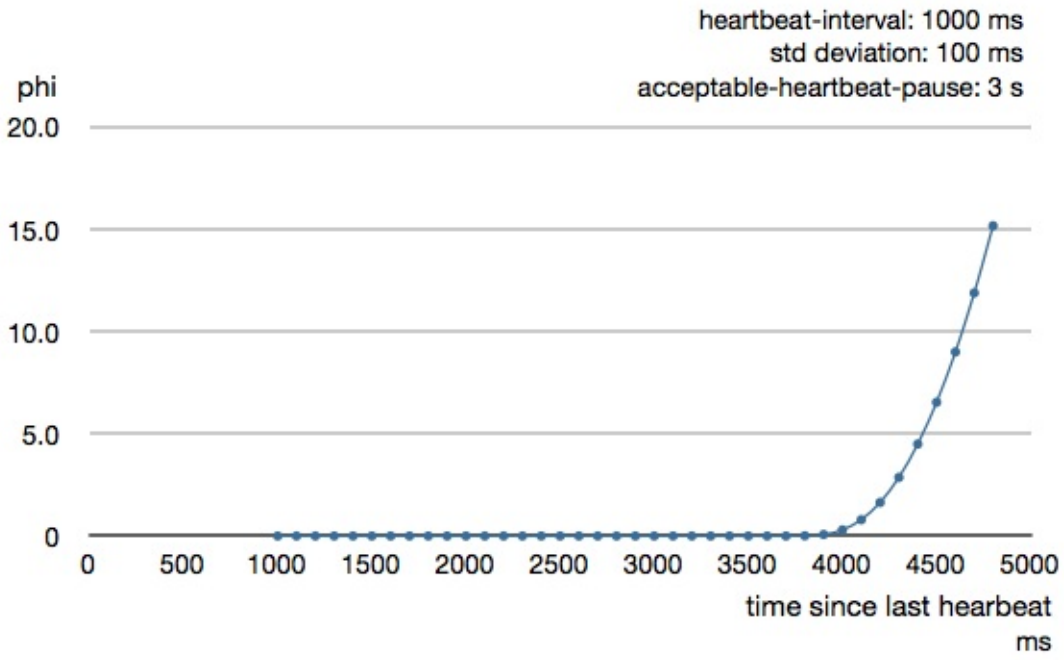
下面的图表说明了*phi*随着距离上次心跳的时间的增加是如何增加的。



Phi是通过历史讲个到达次数的平均值和标准偏差计算的。前面的图表是标准偏差为200 ms的例子。如果心跳到达偏差更小，则曲线会变得更陡峭，即有可能更快地确定故障。标准偏差为 100 毫秒的曲线看起来像这样。



为了能够适应突然的异常，如垃圾收集导致的暂停和瞬态网络故障，失效检测器配置了一个便捷—— `akka.remote.watch-failure-detector.acceptable-heartbeat-pause` 。你可能想要根据以来的环境调整远程配置。当 `acceptable-heartbeat-pause` 被设置为3秒时，曲线看上去像这样。



序列化

对actor使用远程调用时，你必须保证这些actor所使用的 `props` 和 `messages` 是可序列化的. 如果不能保证会导致系统产生意料之外的行为.

更多信息请参阅序列化(Scala)。

有远程目标的路由actor

将远程调用与路由(Scala)进行组合绝对是可行的.

远程部署routees池可以这样被配置:

```
1.  akka.actor.deployment {
2.    /parent/remotePool {
3.      router = round-robin-pool
4.      nr-of-instances = 10
5.      target.nodes = ["akka.tcp://app@10.0.0.2:2552",
    "akka://app@10.0.0.3:2552"]
```

```
6.    }
7.  }
```

此配置设置将克隆10个定义在 `remotePool` 的 `Props` 的actor，并将其均匀地分布在两个给定的目标节点上部署。

一个远程actor group可以这样配置：

```
1.  akka.actor.deployment {
2.    /parent/remoteGroup {
3.      router = round-robin-group
4.      routees.paths = [
5.        "akka.tcp://app@10.0.0.1:2552/user/workers/w1",
6.        "akka.tcp://app@10.0.0.2:2552/user/workers/w1",
7.        "akka.tcp://app@10.0.0.3:2552/user/workers/w1"]
8.    }
9.  }
```

此配置设置将想定义的远程actor路径发送消息。它要求你在远程节点相匹配的路径上创建目标actor。这不是由路由器做的。

远程处理示例

Typesafe Activator。名为Akka Remote Samples with Scala的教程一并演示了远程部署和查找远程actor。

可插拔的传输支持

Akka可以为远程系统配置使用不同的传输协议进行通信。此功能的核心部件是 `akka.remote.Transport` SPI。传输实现必须扩展这一特质。可以通过设置 `akka.remote.enabled-transports` 配置键，使其指向一个或多个包含驱动程序说明的配置节，来载入传输。

设置基于SSL驱动程序的Netty作为默认值的示例：

```
akka {
  remote {
    enabled-transports = [akka.remote.netty.ssl]

    netty.ssl.security {
      key-store = "mykeystore"
      trust-store = "mytruststore"
      key-store-password = "changeme"
      key-password = "changeme"
      trust-store-password = "changeme"
      protocol = "TLSv1"
      random-number-generator = "AES128CounterSecureRNG"
      enabled-algorithms = [TLS_RSA_WITH_AES_128_CBC_SHA]
    }
  }
}
```

一个设置自定义传输实现的示例：

```
akka {
  remote {
    applied-transports = ["akka.remote.mytransport"]

    mytransport {
      # The transport-class configuration entry is required, and
      # it must contain the fully qualified name of the transport
      # implementation
      transport-class = "my.package.MyTransport"

      # It is possible to decorate Transports with additional
services.
      # Adapters should be registered in the "adapters" sections to
      # be able to apply them to transports
      applied-adapters = []

      # Driver specific configuration options has to be in the same
      # section:
```

```
18.         some-config = foo
19.         another-config = bar
20.     }
```

远程事件

可以监听Akka远程调用中发生的事件，也可以订阅/取消订阅这些事情，你只需要在 `ActorSystem.eventStream` 中为下面所列出类型的事件注册监听器.

> 注意
>
> 若要订阅任意远程事件，订阅 *`RemotingLifecycleEvent`* 。若要订阅只涉及链接的生命周期的事件，请订阅 *`akka.remote.AssociationEvent`* 。
>
> 注意
>
> 使用"链接"而不是"连接"一词，反映了远程处理子系统可能使用无连接传输，但链接类似于运输层连接，来维持点到点之间的*Akka*协议。

默认情况下注册的事件监听器，会记录所有下面描述的事件。此默认值被选为帮助建立一个系统，但一旦完成了这一阶段的项目，一般会选择关掉此日志记录。

> 注意
>
> 设置 *`application.conf`* 中的 *`akka.remote.log-remote-lifecycle-events = off`* 来关闭日志记录。

要在链接结束(“disconnected”)时收到通知，监听 `DisassociatedEvent` ， 这个事件持有链接的方向（传入或传出）和参与方的地址.

要在链接成功建立(“connected”)时收到通知，监听 `AssociatedEvent` ， 这个事件持有链接的方向（传入或传出）和参与方的地址.

要拦截与链接直接相关的错误，监听 `AssociationErrorEvent` ， 这个事件持有链接的方向（传入或传出）、参与方的地址和 `Throwable` 原因.

要在远程子系统准备好接受链接时收到通知，监听 `RemotingListenEvent` ，这个事件持有远程监听的地址.

要在远程子系统被关闭时收到通知，监听 `RemotingShutdownEvent` .

要拦截与远程相关的广泛错误，监听包含 `Throwable` 原因的 `RemotingErrorEvent` .

远程安全

Akka提供了几种方式来加强远程节点（客户端/服务器）之间的安全:

- 不受信任的模式
- 安全 Cookie 握手

不受信任的模式

一旦Actor系统可以远程连接到另一个系统，它原则上可以向包含在该远程系统内的任何一个actor发送任何可能的消息。一个例子是可能会给系统守护者发送 `PoisonPill` ，关闭该系统。这并非总是符合期望，它可以通过下列设置禁用:

```
1.  akka.remote.untrusted-mode = on
```

对设置了此标志的系统，这禁用了系统消息发送（actor生命周期命令，DeathWatch，等等）和任何继承自 `PossiblyHarmful` 的消息。客户端应该发送他们，尽管它们会被丢弃和记录日志（在DEBUG调试级别以减少拒绝服务攻击的可能性）。 `PossiblyHarmful` 涉及的预定义的消息，像 `PoisonPill` 和 `Kill` ，但它也可以被添加到用户定义的消息

作为标记特质。

通过actor selection发送的消息在不受信任模式下默认是丢弃的，但接收actor selection消息的权限可授予特定的actor，像这样在配置中定义：

```
1.  akka.remote.trusted-selection-paths = ["/user/receptionist",
    "/user/namingService"]
```

实际的消息仍然必须不能是 `PossiblyHarmful` 类型。

总之，配置为不受信任模式的系统通过远程处理层传入的以下操作将被忽略：

- 远程部署 （这也意味着没有远程监控）
- 远程DeathWatch
- `system.stop()`， `PoisonPill`， `Kill`
- 发送任何继承自 `PossiblyHarmful` 标记接口的消息，包括 `Terminated`
- 通过actor selection发送的消息，除非目标定义在 `trusted-selection-paths` 中。

> 注意
>
> 启用不受信任模式并不会取消客户端能够自由选择其消息发送目标的能力，这意味着不按上述规则禁止的消息可以发送给远程系统中的任何一个*actor*。对一个面向客户的系统，仅仅包含一组定义良好的入口点*actor*，然后将请求转发（可能在执行验证后）到另一个包含实际工作者*actor*的*actor*系统是一个好的实践。如果两个服务器端系统之间消息传递使用本地 *ActorRef* （他们安全地在同一个*JVM*上的两个*actor*系统之间交换），你可以通过标记他们为 *PossiblyHarmful* 来限制此接口上的消息，从而使客户端不能伪造。

安全 Cookie 握手

Akka远程处理还允许你指定一个安全cookie，它将被交换并确保在客户端和服务器之间的连接握手中是相同的。如果他们不相同，则客户端将被拒绝连接到服务器。

安全cookie可以是任何类型的字符串。但推荐使用此脚本生成一个加密安全cookie——
$ `$AKKA_HOME/scripts/generate_config_with_secure_cookie.sh` 或者从代码中使用 `akka.util.Crypt.generateSecureCookie()` 工具方法。

你必须确保连接的客户端和服务器都有相同的一个安全 cookie，并同时打开了 `require-cookie` 选项。

下面是一个示例配置：

```
1.  akka.remote {
2.    secure-cookie = "090A030E0F0A05010900000A0C0E0C0B03050D05"
3.    require-cookie = on
4.  }
```

SSL

SSL可以用作远程运输，通过添加 `akka.remote.netty.ssl` 到 `enabled-transport` 配置节。请参阅远程配置以节中的设置说明。

SSL支持是用Java安全套接字扩展实现的，请参阅官方的Java安全套接字扩展文档和相关的资源进行故障排除。

> 注意
>
> 当在*Linux*上使用*SHA1PRNG*时，推荐指定 `-Djava.security.egd=file:/dev/./urandom` 作为*JVM*参数形式指定以防止阻塞。它并不安全因为它重用了种子。使用*'/dev/./urandom'*，而不使用*'/dev/urandom'*是行不通的，见*Bug ID: 6202721*。

远程配置" class="reference-link">远程配置

有很多与Akka远程处理相关的配置属性。可以在参考配置中获取详细信息。(译者注：中文翻译附在本节后面，摘自Akka 2.0的翻译)

> 注意

> 以编程方式设置如监听*IP*和端口号的属性，最好是通过类似以下内容的方式：

```
ConfigFactory.parseString("akka.remote.netty.tcp.hostname=\"1.2.3.
      .withFallback(ConfigFactory.load());
```

远程配置信息：

```
#####################################
# Akka 远程调用参考配置文件 #
#####################################

# 本参考配置文件包含所有的缺省配置.
# 在你自己的 application.conf 可对其进行编辑/重写.

# 关于akka-actor.jar 中已有的akka.actor设置的注释被去掉了, 不然会发生重复设置.

akka {

  actor {

    serializers {
      proto = "akka.serialization.ProtobufSerializer"
    }


    serialization-bindings {
      # 由 com.google.protobuf.Message 没有继承 Serializable 但 GeneratedMessage
      # 有, 这里必须使用更明确的类来避免歧义
      "com.google.protobuf.GeneratedMessage" = proto
    }

    deployment {

      default {

        # 如果设置为一个可用的远程地址, 这个有名称的actor会被部署到那个结点
```

```
30.         # e.g. "akka://sys@host:port"
31.         remote = ""
32.
33.         target {
34.
35.           # 一个主机名和端口列表，用来创建一个非直接路由actor的子actor
36.           #   格式应为 "akka://sys@host:port"，其中：
37.           #     - sys 是远程actor系统的名称
38.           #     - hostname 可以是主机名或远程主机应连接到的IP地址
39.           #     - port 应为其它结点上的远程服务的端口
40.           # 象本地路由actor一样，新生成的actor实例的数量仍从
41.           # nr-of-instances 配置中获取；新的实例在给定的结点中将以
42.           # round-robin 的方式分布
43.           nodes = []
44.
45.         }
46.       }
47.     }
48.   }
49.
50.   remote {
51.
52.     # 使用 akka.remote.RemoteTransport 的哪个实现
53.     # 缺省是基于TCP，Netty上的远程传输层
54.     transport = "akka.remote.netty.NettyRemoteTransport"
55.
56.     # 打开为服务器管理的actor的完全的安全性打开不信任模式， 允许不受信任的
57.     # 客户端建立连接.
58.     untrusted-mode = off
59.
60.     # 集群操作的 ACK 超时，例如检查 actor 等.
61.     remote-daemon-ack-timeout = 30s
62.
63.     # 如果这个值是 "on"， Akka 会以DEBUG级别记录所有接收到的消息到日志， 如果
    是 off 则不会被记录
64.     log-received-messages = off
65.
66.     # 如果这个值是 "on"， Akka 会以DEBUG级别记录所有发送的消息到日志，如果是
```

```
    off 则不会被记录
67.     log-sent-messages = off
68.
69.     # 每一个属性被标记为 (I) 或 (O) 或 (I&O), I 代表 “输入” O 代表 “输
    出” 连接.
70.     # NettyRemoteTransport 启动的服务器总是允许输入的连接, 当发送到某个尚未
    连接的目标时总是会启动活跃的客户端连接
71.     # ; 如果配置指定, 它可以重用输入的连接来发送应答, 这被称为被动客户端连接
72.     #  (i.e. 从服务器到客户端).
73.     netty {
74.
75.       # (O) 在延迟变长/溢出的情况下要等待多久 (阻塞发送方)
76.       # 才取消发送
77.       # 0 表示 "不取消", 任何正数表示最长的阻塞时间.
78.       backoff-timeout = 0ms
79.
80.       # (I&O) 用
    '$AKKA_HOME/scripts/generate_config_with_secure_cookie.sh' 创建自己
    cookie
81.       # 或使用 'akka.util.Crypt.generateSecureCookie'
82.       secure-cookie = ""
83.
84.       # (I) 远程服务器是否要求连接对方也共享同样的 secure-cookie
85.       # (在 'remote' 部分定义)?
86.       require-cookie = off
87.
88.       # (I) 重用输入连接来发送消息
89.       use-passive-connections = on
90.
91.       # (I) 远程调用所绑定的主机名或ip,
92.       # 不设则使用InetAddress.getLocalHost.getHostAddress
93.       hostname = ""
94.
95.       # (I) 客户端应连接到的缺省远程服务器端口.
96.       # 缺省值为 2552 (AKKA), 0 表示随机选择一个可用端口
97.       port = 2552
98.
99.       # (O) 创建输出连接时绑定到的本地网络接口地址 (IP 地址)
```

```
100.        # 设置为 "" 或 "auto" 表示自动选择本地地址.
101.        outbound-local-address = "auto"
102.
103.        # (I&O) 如果你希望发送内容较大的消息则设置这个参数
104.        message-frame-size = 1 MiB
105.
106.        # (O) 超时间隔
107.        connection-timeout = 120s
108.
109.        # (I) 储备连接的大小
110.        backlog = 4096
111.
112.        # (I) 核心线程空闲时保持存活的时间长度, 以 akka.time-unit 为单位
113.        execution-pool-keepalive = 60s
114.
115.        # (I) 远程执行单元的核心池的大小
116.        execution-pool-size = 4
117.
118.        # (I) channel 大小的上限, 0 表示关闭
119.        max-channel-memory-size = 0b
120.
121.        # (I) 所有channel总大小的上限, 0 表示关闭
122.        max-total-memory-size = 0b
123.
124.        # (O) 活跃客户端重连的间隔
125.        reconnect-delay = 5s
126.
127.        # (O) 读非活跃时间 (最小单位为秒)
128.        # 经过这么长时间后, 活跃客户端将被关闭;
129.        # 当有新的通信请求时将被重新建立.
130.        # 0表示关闭这个功能
131.        read-timeout = 0s
132.
133.        # (O) 写非活跃时间 (最小单位为秒)
134.        # 经过这么长时间后将发送心跳.
135.        # 0表示关闭这个功能
136.        write-timeout = 10s
137.
```

```
138.         # (O) 读和写的非活跃时间 (最小单位为秒)
139.         # 经过这么长时间后活跃客户端连接将被关闭;
140.         # 当有新的通信请求时将被重新建立
141.         # 0表示关闭这个功能
142.         all-timeout = 0s
143.
144.         # (O) 客户端应进行重连的最大时间窗口
145.         reconnection-time-window = 600s
146.       }
147.
148.       # 系统 actor "network-event-sender" 所使用的派发器
149.       network-event-sender-dispatcher {
150.         executor = thread-pool-executor
151.         type = PinnedDispatcher
152.       }
153.     }
154.   }
```


序列化

序列化

> 注：本节未经校验，如有问题欢迎提*issue*

Akka 提供了内置的支持序列化的扩展，你可以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。

用法

配置

为了让 Akka 知道对什么任务使用哪个 `Serializer`，你需要编辑你的 配置文件，在 “akka.actor.serializers”一节将名称绑定为 `akka.serialization.Serializer` 的实现，像这样：

```
akka {
  actor {
    serializers {
      java = "akka.serialization.JavaSerializer"
      proto = "akka.remote.serialization.ProtobufSerializer"
      myown = "docs.serialization.MyOwnSerializer"
    }
  }
}
```

在将名称与 `Serializer` 的不同实现绑定后，你需要指定哪些类的序列化使用哪种 `Serializer`， 这部分配置写在“akka.actor.serialization-bindings”一节中:

```
akka {
  actor {
    serializers {
      java = "akka.serialization.JavaSerializer"
      proto = "akka.remote.serialization.ProtobufSerializer"
      myown = "docs.serialization.MyOwnSerializer"
    }

    serialization-bindings {
      "java.lang.String" = java
      "docs.serialization.Customer" = java
      "com.google.protobuf.Message" = proto
      "docs.serialization.MyOwnSerializable" = myown
      "java.lang.Boolean" = myown
    }
  }
}
```

你只需要指定消息的接口或抽象基类. 当消息实现了配置中多个类时，为避免歧义， 将使用最具体的类， 即对其它所有配置指定类，它都是子类的那一个. 如果这个条件不满足， 例如因为同时配

置 `java.io.Serializable` 和 `MyOwnSerializable`，而彼此都不是对方的子类型，将生成一个警告。

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 消息将总是使用 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”:

```
1. akka.actor.serialization-bindings {
2.   "java.io.Serializable" = none
3. }
```

确认

如果你希望确认你的消息是可以被序列化的，你可以打开这个配置项:

```
1. akka {
2.   actor {
3.     serialize-messages = on
4.   }
5. }
```

> 警告
>
> 我们推荐只在运行测试代码的时候才打开这个选项. 在其它的场景打开它完全没有道理.

如果你希望确认你的 Props 可以被序列化，你可以打开这个配置项:

```
1. akka {
2.   actor {
3.     serialize-creators = on
4.   }
```

```
5. }
```

> 警告
>
> 我们推荐只在运行测试代码的时候才打开这个选项. 在其它的场景打开它完全没有道理.

通过代码

如果你希望通过代码使用 Akka Serialization来进行序列化/反序列化, 以下是一些例子:

```
1. import akka.actor.{ ActorRef, ActorSystem }
2. import akka.serialization._
3. import com.typesafe.config.ConfigFactory
4.
5.     val system = ActorSystem("example")
6.
7.     // Get the Serialization Extension
8.     val serialization = SerializationExtension(system)
9.
10.    // Have something to serialize
11.    val original = "woohoo"
12.
13.    // Find the Serializer for it
14.    val serializer = serialization.findSerializerFor(original)
15.
16.    // Turn it into bytes
17.    val bytes = serializer.toBinary(original)
18.
19.    // Turn it back into an object
20.    val back = serializer.fromBinary(bytes, manifest = None)
21.
22.    // Voilá!
23.    back should be(original)
```

更多信息请见 `akka.serialization._` 的 `ScalaDoc`

自定义

如果你希望创建自己的 `Serializer`，应该已经看到配置例子中的 `docs.serialization.MyOwnSerializer` 了吧?

创建新的 Serializer

首先你需要为你的 `Serializer` 写一个类定义，像这样:

```
1.  import akka.actor.{ ActorRef, ActorSystem }
2.  import akka.serialization._
3.  import com.typesafe.config.ConfigFactory
4.
5.  class MyOwnSerializer extends Serializer {
6.
7.    // This is whether "fromBinary" requires a "clazz" or not
8.    def includeManifest: Boolean = false
9.
10.   // Pick a unique identifier for your Serializer,
11.   // you've got a couple of billions to choose from,
12.   // 0 - 16 is reserved by Akka itself
13.   def identifier = 1234567
14.
15.   // "toBinary" serializes the given object to an Array of Bytes
16.   def toBinary(obj: AnyRef): Array[Byte] = {
17.     // Put the code that serializes the object here
18.     // ... ...
19.   }
20.
21.   // "fromBinary" deserializes the given array,
22.   // using the type hint (if any, see "includeManifest" above)
23.   // into the optionally provided classLoader.
24.   def fromBinary(bytes: Array[Byte],
25.                  clazz: Option[Class[_]]): AnyRef = {
26.     // Put your code that deserializes here
27.     // ... ...
28.   }
29. }
```

然后你只需要做填空，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.

Actor引用的序列化

所有的 ActorRef 都是用 JavaSerializer进行序列化的, 但如果你写了自己的serializer， 你可能想知道如何正确对它们进行序列化和反序列化。在一般情况下，要使用的本地地址取决于作为序列化信息收件人的远程地址的类型。像这样使用 `Serialization.serializedActorPath(actorRef)` :

```
1.  import akka.actor.{ ActorRef, ActorSystem }
2.  import akka.serialization._
3.  import com.typesafe.config.ConfigFactory
4.
5.      // Serialize
6.      // (beneath toBinary)
7.      val identifier: String =
    Serialization.serializedActorPath(theActorRef)
8.
9.      // Then just serialize the identifier however you like
10.
11.     // Deserialize
12.     // (beneath fromBinary)
13.     val deserializedActorRef =
    extendedSystem.provider.resolveActorRef(identifier)
14.     // Then just use the ActorRef
```

这是假定序列化发生在通过远程传输发送消息的上下文中。不过有一些序列化有其他用途，例如在actor应用程序之外存储actor引用（数据库等）。在这种情况下，需要牢记的重要一点是，actor路径的地址部分决定actor如何被通信连通。存储一个本地actor路径可能是正确的选择，如果回取发生在相同的逻辑上下文，但当在不同的网络主机上反序列化时它还不够： 为此，它将需要包括系统的远程传输地址。一个

actor系统并不局限于只有一个远程运输，使得这个问题变得更有意思。当向 `remoteAddr` 发送消息时，要找出恰当的地址，你可以像下面这样使用 `ActorRefProvider.getExternalAddressFor(remoteAddr)` ：

```
1.  object ExternalAddress extends ExtensionKey[ExternalAddressExt]
2.
3.  class ExternalAddressExt(system: ExtendedActorSystem) extends
    Extension {
4.    def addressFor(remoteAddr: Address): Address =
5.      system.provider.getExternalAddressFor(remoteAddr) getOrElse
6.        (throw new UnsupportedOperationException("cannot send to " +
    remoteAddr))
7.  }
8.
9.  def serializeTo(ref: ActorRef, remote: Address): String =
10.
    ref.path.toSerializationFormatWithAddress(ExternalAddress(extendedSys
11.     addressFor(remote))
```

> 注意
>
> 如果地址并没有 *host* 和 *port* 组件时 *ActorPath.toSerializationFormatWithAddress* 不同于 *toString* ，即它只为本地地址插入地址信息。
>
> *toSerializationFormatWithAddress* 还为*actor*添加了唯一 *id*，当*actor*终止，然后又按照相同名称重新创建时，该*id*将改变。将消息发送到指向那个老*actor*的引用，将不会传递给新的*actor*。如果你不想要这种行为，例如在长期存储引用的情况下，你可以使用不包括的唯一*id*的 *toStringWithAddress* 。

这就要求你至少知道哪种类型的地址将被反序列化生成的actor引用的系统支持；如果你手头没有具体的地址，你可以创建一个虚拟使用正确协议的地址 `Address(protocol, "", "", 0)` （假定所用的实际传输与Akka的 `RemoteActorRefProvider` 是相兼容的）。

也有一个用来支持集群的默认远程地址（并且典型系统只有这一个）；

你可以像这样获取它：

```
1. object ExternalAddress extends ExtensionKey[ExternalAddressExt]
2.
3. class ExternalAddressExt(system: ExtendedActorSystem) extends
   Extension {
4.   def addressForAkka: Address = system.provider.getDefaultAddress
5. }
6.
7. def serializeAkkaDefault(ref: ActorRef): String =
8.
   ref.path.toSerializationFormatWithAddress(ExternalAddress(theActorSys
9.     addressForAkka)
```

Actor的深度序列化

做内部actor状态深度序列化，推荐的方法是使用Akka持久化。

关于 Java 序列化

如果在做Java序列化时没有使用 `JavaSerializer` ， 你必须保证在动态变量 `JavaSerializer.currentSystem` 中提供一个有效的 `ExtendedActorSystem` . 它用在读取 `ActorRef` 表示时将其字符串表示转换成实际的引用. 动态变量 `DynamicVariable` 是一个 thread-local变量，所以在反序列化任何可能包含actor引用的数据时要保证这个变量有值.

外部 Akka Serializers

Akka-protostuff by Roman Levenstein

Akka-quickser by Roman Levenstein

Akka-kryo by Roman Levenstein

I/O

I/O

> 注：本节未经校验，如有问题欢迎提*issue*

介绍

`akka.io` 包是由Akka和spray.io团队协作开发的。它的设计结合了 `spray-io` 模块的经验，并共同进行了改进，使其适应基于actor服务的更加普遍的消费需求。

该 I/O 实现的指导设计目标是要达到极端的可扩展性，要毫不妥协地提供一个API正确匹配底层传输机制，并且是完全的事件驱动、无阻塞和异步。该API命中注定是网络协议实现和构建更高抽象的坚实基础；它不是为终端用户提供的全套的高级别的NIO包装。

术语，概念

I/O API完全是基于actor的，意味着所有的操作实现都是通过消息传递而不是直接方法调用。每个 I/O 驱动程序 （TCP、 UDP） 有一个特殊的actor，被称为一个管理器，用作 API 的入口点。I/O 被

分成几个驱动程序。用于某个特定驱动程序的管理器是通过 `IO` 入口点获取的。例如下面的代码查找 TCP 管理器，并返回其 `ActorRef` ：

```
1. import akka.io.{ IO, Tcp }
2. import context.system // implicitly used by IO(Tcp)
3.
4. val manager = IO(Tcp)
```

管理器接收 I/O 命令消息并实例化工作actor作为回应。工作actor将自身返回给 API 用户作为发送该命令的答复。例如给TCP 管理器发送 `Connect` 命令后，管理器创建了代表 TCP 连接的actor。当该actor通过发送一个 `Connected` 消息宣布自身后，所有与给定 TCP 连接相关的操作都可以通过发送消息到连接actor来调用。

DeathWatch和资源管理

I/O 工作actor接收命令，并且也发出事件。它们通常需要一个用户端对应的actor监听这些事件 （此类事件可以是入站的连接，传入的字节或写操作确认）。这些工作actor观察它们的对应监听。如果监听器停止工作，则工作actor将自动释放它所拥有的任何资源。这种设计使得该 API 更能抵抗资源泄漏。

多亏I/O API是完全基于actor设计的，相反的方向也可以工作：一个负责处理连接的用户actor可以观察连接actor，如果它意外终止也将收到通知。

写模型(Ack, Nack)

I/O设备有一个最大吞吐量来限制写操作的频率和大小。当一个应用程序试图推相比设备处理能力更多的数据时，驱动程序不得不缓冲字节，直到设备能够继续写他们。缓冲可以处理短暂的密集写入—但没有缓冲区是无限的。这时需要”流控制”来避免设备缓冲区不足的问题。

Akka支持两种类型的流量控制：

- 基于*Ack*，当写操作成功的时候，驱动程序通知写者。
- 基于*Nack*，当写操作失败时，驱动程序会通知写者。

每一种模型在Akka I/O的 TCP 和 UDP 实现中都可用。

单独的写操作可以通过在写入消息（TCP中的 `Write` 和UDP中的 `Send` ）中提供一个 ack 对象来确认。写操作完成时工作者将发送 ack 对象给写actor。这可以用于实现基于*Ack*的流量控制；只有当老数据被确认时才发送新数据。

如果写入（或任何其他命令）失败，驱动程序会发送具有该命令一个特殊消息（UDP 和 TCP中是 `CommandFailed` ）来通知actor。此消息也会通知写者一个失败，作为那个写的一个nack。请注意，在基于nack的流控制设置中，写者必须准备到，失败的写操作可能不是最近写操作的事实。例如，对于 `W1` 的写入失败通知可能在后来的写命令 `W2` 和 `W3` 被发送之后到达。如果写者想要重发送任何nack消息，它可能需要保留一个挂起消息的缓冲区。

> 警告
>
> 一个确认的写并不意味着确认送达或存储的；收到一个写*ack*只是表明*I/O* 驱动程序成功处理了写操作。这里描述的 *Ack/Nack* 协议是一种流量控制手段而不是错误处理。换句话说，数据仍然可能会丢失，即使每一个写操作都被确认。

ByteString" class="reference-link">ByteString

为了保持隔离，actor应该只通过不可变对象沟通。 `ByteString` 是bytes的不可的容器。它被用在Akka I/O系统中，作为在jvm上处理IO的传统字节容器，如 `Array[Byte]` 和 `ByteBuffer` 的一种高效的、不可变的替代者。

ByteString 是一个绳状)数据结构，它不可变且提供了高效地连接和

切片操作（完美的 I/O）. 当两个 `ByteString` 被连接在一起时，是将两者都保存在结果 `ByteString` 内部而不是将它们复制到新的 `Array` 中. 像 `drop` 和 `take` 这种操作返回的 `ByteString` 仍引用之前的 `Array`，只是改变了外部可见的offset和length。我们花了很大力气保证内部的 `Array` 不能被修改. 每次当不安全的 `Array` 被用于创建新的 `ByteString` 时，会创建一个防御性拷贝。如果你需要一个 `ByteString` 为其内容占用尽可能少的内存，使用 `compact` 方法来获取一个 `CompactByteString` 实例。如果 `ByteString` 表示只是原始数组中的一片，这将导致复制在片中的所有字节。

`ByteString` 从 `IndexedSeq` 继承了所有方法，它也有一些新的方法。更多信息请参考 `akka.util.ByteString` 类和其伴生对象的 ScalaDoc。

`ByteString` 还带有它自己优化类—— `ByteStringBuilder` 和 `ByteIterator`，在普通生成器和迭代器之外提供额外的功能。

与 java.io 的兼容性

`ByteStringBuilder` 可以通过 `asOutputStream` 方法包装为一个 `java.io.OutputStream`。同样， `ByteIterator` 可以通过 `asInputStream` 包装为一个 `java.io.InputStream`。使用这些， `akka.io` 应用程序可以集成基于 `java.io` 流的遗留代码。

深入体系结构

关于内部体系结构设计有关的详细信息请参阅I/O 层设计。

使用TCP

使用TCP

> 注：本节未经校验，如有问题欢迎提*issue*

贯穿本节的代码片段假定以下imports:

```
1. import akka.actor.{ Actor, ActorRef, Props }
2. import akka.io.{ IO, Tcp }
3. import akka.util.ByteString
4. import java.net.InetSocketAddress
```

所有的Akka I/O API 都通过管理器对象来访问。当使用I/O API时，第一步是获得适当的管理器对象的引用。下面的代码演示如何获取 `Tcp` 管理器的引用。

```
1. import akka.io.{ IO, Tcp }
2. import context.system // implicitly used by IO(Tcp)
3.
4. val manager = IO(Tcp)
```

管理器是一个actor，来处理底层低级别 I/O 资源(selectors, channels)并为特定任务，如监听传入的连接，来实例化工作者。

连接

```
1.  object Client {
2.    def props(remote: InetSocketAddress, replies: ActorRef) =
3.      Props(classOf[Client], remote, replies)
4.  }
5.
6.  class Client(remote: InetSocketAddress, listener: ActorRef) extends
    Actor {
7.
8.    import Tcp._
9.    import context.system
10.
11.   IO(Tcp) ! Connect(remote)
12.
13.   def receive = {
14.     case CommandFailed(_: Connect) =>
15.       listener ! "connect failed"
16.       context stop self
17.
18.     case c @ Connected(remote, local) =>
19.       listener ! c
20.       val connection = sender()
21.       connection ! Register(self)
22.       context become {
23.         case data: ByteString =>
24.           connection ! Write(data)
25.         case CommandFailed(w: Write) =>
26.           // O/S buffer was full
27.           listener ! "write failed"
28.         case Received(data) =>
29.           listener ! data
30.         case "close" =>
31.           connection ! Close
```

```
32.         case _: ConnectionClosed =>
33.           listener ! "connection closed"
34.           context stop self
35.       }
36.   }
37. }
```

连接到一个远程地址的第一步是将 `Connect` 消息发送到 TCP 管理器；除了上面所示的简单形式外，还可以指定要绑定的本地 `InetSocketAddress` 和要应用的套接字选项列表。

> 注意
>
> *SO_NODELAY (Windows下是TCP_NODELAY)* 套接字选项在*Akka*中默认为 *true*，独立于操作系统的默认设置。此设置禁用*Nagle*算法，大大提高对于大多数应用程序的延迟。此设置可以通过在 *Connect* 消息的套接字选项的列表中加入 *SO.TcpNoDelay(false)* 来重写。

TCP 管理器然后将要么回复 `CommandFailed` ，要么产生一个内部的actor代表新的连接。这位新actor然后将向 `Connect` 消息的原始发送人发送一个 `Connected` 消息。

为了激活新的连接， `Register` 消息必须发送到连接actor，通知它谁将从套接字接收数据。此步骤完成之前不能使用连接，而且还有一个内部超时，在此之后如果没有收到 `Register` 消息，连接actor将关闭自身。

连接actor观察注册的处理程序，并在处理者终止时关闭连接，从而清理与该连接相关的所有内部资源。

上面的示例中actor使用 `become` 来从无连接状态切换到已连接状态，演示那种状态下被观察到的命令和事件。关于 `CommandFailed` 参见下文的讨论——读写限流。 `ConnectionClosed` 是一个特质，它标志着不同的连接关闭事件。最后一行中以同样的方式处理所有的连接关闭事件。有可能监听更多的细粒度的连接关闭事件，请参阅下面的关闭连接。

接受连接

```
1.  class Server extends Actor {
2.
3.    import Tcp._
4.    import context.system
5.
6.    IO(Tcp) ! Bind(self, new InetSocketAddress("localhost", 0))
7.
8.    def receive = {
9.      case b @ Bound(localAddress) =>
10.       // do some logging or setup ...
11.
12.     case CommandFailed(_: Bind) => context stop self
13.
14.     case c @ Connected(remote, local) =>
15.       val handler = context.actorOf(Props[SimplisticHandler])
16.       val connection = sender()
17.       connection ! Register(handler)
18.   }
19.
20. }
```

要创建一个 TCP 服务器并监听入站连接，需要发送一个 `Bind` 命令到 TCP 管理器。这将指示 TCP 管理器来侦听一个特定 `InetSocketAddress` 的 TCP 连接；可将端口指定为 0来绑定一个随机端口。

发送 `Bind` 消息的actor将收到 `Bound` 消息，表示服务器已准备好接收传入的连接 ；此消息还包含实际绑定到套接字的 `InetSocketAddress` （即解析号的 IP 地址和正确的端口号）。

从这个点开始，处理连接的过程与传出连接是相同的。该示例演示了，当发送 `Register` 消息时，来自某个连接的读取可以委派给另一位被指定处理程序的actor。写操作可以从系统中任何一个actor发送到连接

actor （即发送 `Connected` 消息的actor）。简单化的处理程序定义如下：

```
1. class SimplisticHandler extends Actor {
2.   import Tcp._
3.   def receive = {
4.     case Received(data) => sender() ! Write(data)
5.     case PeerClosed     => context stop self
6.   }
7. }
```

在发送时考虑到失败可能性的更完整示例，请参阅下面的读写限流。

对传出连接的唯一区别是内部actor管理监听端口 —— `Bound` 消息的发件人 —— 观察在 `Bind` 消息中被指定为 `Connected` 消息接收者的那个actor。那个actor终止时监听端口将会封闭，与它相关的所有资源将都被释放 ；在这一点上，现有连接不会被终止。

关闭连接

可以通过发送 `Close` ， `ConfirmedClose` 或 `Abort` 之一的命令到连接actor关闭连接。

通过发送 `FIN` 消息， `Close` 将关闭该连接，但没有等待远端的确认。挂起的写操作将被执行(flush)。如果关闭成功， 监听器将收到 `Closed` 通知。

通过发送 `FIN` 消息， `ConfirmedClose` 将关闭发送方的连接，但仍会收到数据直到远端也关闭连接。挂起的写操作将被执行(flush)。如果关闭成功， 监听器将收到 `ConfirmedClosed` 通知。

通过发送 `RST` 消息到远端， `Abort` 将立即终止连接。挂起的写操作将不会被执行(flush)。如果关闭成功， 监听器将收到 `Aborted` 通知。

如果该连接被远端关闭，监听器将收到 `PeerClosed` 通知。默认情况下，该连接然后也会被本端自动关闭。要支持半闭连接，请将 `Register` 消息的 `keepOpenOnPeerClosed` 成员设置为 `true` ，在此情况下连接将保持打开状态，直到它接收到一个上文介绍的关闭命令。

每当一个错误发生并迫使该连接关闭时， `ErrorClosed` 将发送到监听器。

所有关闭通知都是 `ConnectionClosed` 的子类型，所以不需要细粒度事件的监听器可以以相同的方式处理所有的关闭事件。

写入一个连接

一旦连接建立，数据可以从任何actor通过 `Tcp.WriteCommand` 的形式发送给连接。 `Tcp.WriteCommand` 是一个抽象类，有三个具体的实现：

- `Tcp.Write`

  最简单的 `WriteCommand` 实现，它包装了一个 `ByteString` 实例和一个”ack”事件。一个 `ByteString` （如在这一节解释的）建模了一个或多个不可变的内存数据块，最大可达2GB（2^31个字节）。

- `Tcp.WriteFile`

  如果你想要从一个文件发送”原始”数据，你可以高效地使用 `Tcp.WriteFile` 命令。它将允许你指定磁盘上一块（连续的）字节在连接中发送，而无需首先将其加载到JVM内存中。 `Tcp.WriteFile` 可以”hold”大于2 GB的数据和并根据需要提供”ack”事件。

- `Tcp.CompoundWrite`

有时你可能想要组（或交错）若干 `Tcp.Write` 和/或 `Tcp.WriteFile` 命令到一个原子写命令，一次写入连接。 `Tcp.CompoundWrite` 允许你做到这一点，并提供三个优点：

i. 如下节中所述，TCP连接actor一次只能处理一个单一写命令。通过组合几个写到一个 `CompoundWrite`，你可以让他们通过连接发送，并拥有最低开销，也无需像”一勺勺喂”一样通过基于*ACK*的消息协议发送给连接actor。

ii. 因为 `WriteCommand` 是原子的，你可以确定没有其他actor可以在你的写序列中”注入”其他写操作，如果你将它们组合为一个单一的 `CompoundWrite`。在多个actor写入相同连接的情况下，这可以是一个重要的特性，通过别的方式很难实现。

iii. `CompoundWrite` 的”子写操作”是普通的 `Write` 或 `WriteFile` 命令，其本身可以请求”ack”事件。这些ACK会在相应的”子写操作”被完成时尽快发送。这允许你将附加多个ACK到 `Write` 或 `WriteFile`（通过组合一个要求ACK的空写）或让连接actor通过在任意点发送中间ACK确认 `CompoundWrite` 的传输进度。

读写限流" class="reference-link">读写限流

TCP 连接actor的基本模型是它有没有内部缓冲 （即它一次只能处理一个写操作，意味着它可以缓冲一次写入，直到写入全部被传递到操作系统内核）。写入和读取的拥塞情况需要在用户级别上处理。

高压写有三种操作模式

- 基于*ACK*：每个 `Write` 命令装载一个任意的对象，并且如果该对象不是 `Tcp.NoAck`，则它将在所有包含数据成功写入到套接字后返回给 `Write` 的发送人。如果收到此确认之前没有启动其他写操作，则

由于缓冲区重用没有失败可以发生。

- 基于*NACK*：在前一个写操作完成之前，每一个新到达的写都将被回复一个包含失败写操作的`CommandFailed`消息。仅仅依靠这种机制要求实现协议容忍跳过写操作（例如如果每个写入操作自身是一个有效的消息，并且不要求所有都送达）。通过在连接激活阶段的`Register`消息中设置`useResumeWriting`标志为 `false`来启用此模式。
- 基于*NACK*与写挂起： 这种模式非常类似于基于NACK的模式，但一旦一次写操作失败，则没有进一步写操作会成功直到接收到`ResumeWriting`消息。一旦上一次接受的写入操作完成，将由一个`WritingResumed`消息回答此消息。如果驱动连接的actor实现缓冲，并在等到`WritingResumed`信号后重发NACK标记的消息，则每个消息确切地只传递一次给网络套接字。

这些高压写模型在下面的示例中进行了完整演示 （除了相当专业化的第二个）。完整的相关资源可以在github上获得。

高压读有两种操作模式

- 推送读： 在此模式下连接actor一旦收到`Received`事件，就向注册的读actor发送传入的数据。每当读actor想要通知到远程TCP 端点压力过大时，它可以发送 `SuspendReading`消息到连接actor表示它想要暂停接收新数据。没有收到`Received`事件直到`ResumeReading`被发送，表示接收actor又准备好了。
- 拉取读： 发送一个`Received`事件后连接actor会自动挂起从套接字接受数据，直到读actor发送一个`ResumeReading`信息，作为信号表示它准备好要处理更多的输入数据。因此新的数据是通过发送`ResumeReading`消息从连接”拉”到的。

> 注意

> 显而易见，所有这些流控制方案只能在一对写者/读者和一个连接*actor*之间工作；一旦多个*actor*发送写命令到一个单独的连接，可以会出现不一致的结果。

基于ACK的高压写

为下面示例能恰当的工作，很重要的一点是当远端关闭其写入时，要配置连接保持半打开状态： 这允许 `EchoHandler` 在连接完全关闭之前将所有未提交数据写到客户端。它是使用连接激活的标志启用的 （观察 `Register` 消息）：

```
1.  case Connected(remote, local) =>
2.    log.info("received connection from {}", remote)
3.    val handler = context.actorOf(Props(handlerClass, sender(),
    remote))
4.    sender() ! Register(handler, keepOpenOnPeerClosed = true)
```

有了这样的让我们深入看下处理程序：

```
1.    // storage omitted ...
2.  class SimpleEchoHandler(connection: ActorRef, remote:
    InetSocketAddress)
3.    extends Actor with ActorLogging {
4.
5.    import Tcp._
6.
7.    // sign death pact: this actor terminates when connection breaks
8.    context watch connection
9.
10.   case object Ack extends Event
11.
12.   def receive = {
13.     case Received(data) =>
14.       buffer(data)
15.       connection ! Write(data, Ack)
16.
17.       context.become({
```

```
18.           case Received(data) => buffer(data)
19.           case Ack            => acknowledge()
20.           case PeerClosed     => closing = true
21.         }, discardOld = false)
22.
23.       case PeerClosed => context stop self
24.     }
25.
26.     // storage omitted ...
27.   }
```

原理很简单： 当写如一大块数据后，总是在发送下一个块之前等待 `Ack` 发送回来。在等待期间，我们切换行为来缓冲新传入的数据。所使用的辅助函数有点长但并不复杂：

```
1.  private def buffer(data: ByteString): Unit = {
2.    storage :+= data
3.    stored += data.size
4.
5.    if (stored > maxStored) {
6.      log.warning(s"drop connection to [$remote] (buffer overrun)")
7.      context stop self
8.
9.    } else if (stored > highWatermark) {
10.     log.debug(s"suspending reading")
11.     connection ! SuspendReading
12.     suspended = true
13.   }
14. }
15.
16. private def acknowledge(): Unit = {
17.   require(storage.nonEmpty, "storage was empty")
18.
19.   val size = storage(0).size
20.   stored -= size
21.   transferred += size
22.
```

```
23.    storage = storage drop 1
24.
25.    if (suspended && stored < lowWatermark) {
26.      log.debug("resuming reading")
27.      connection ! ResumeReading
28.      suspended = false
29.    }
30.
31.    if (storage.isEmpty) {
32.      if (closing) context stop self
33.      else context.unbecome()
34.    } else connection ! Write(storage(0), Ack)
35.  }
```

最有趣的部分也许是最后一行： 一个 `Ack` 从缓冲区中删除最旧的数据块，如果这是最后一个块则我们要么关闭连接 （如果同行已经关闭了它的一半）要么返回到空闲的行为；不然我们就发送下一个缓冲块，并等待下一个 `Ack` 。

高压也可以传播，穿过读侧回到连接另一端的写侧，通过将 `SuspendReading` 命令发送到连接actor。这将导致不再从套接字读取任何数据 （尽管这会延迟发生，因为它需要一些时间，直到连接actor处理这个命令，因此适当的头部缓冲区应该出现），反过来这将导致我们这一端的操作系统内核缓冲区填满，然后 TCP 窗口机制将停止远端写，填满其写入缓冲区，直到最后另一端的写者无法推如任何数据到套接字中了。这是端到端高压如何在跨 TCP 连接实现的。

带有挂起的基于NACK高压写

```
1.  class EchoHandler(connection: ActorRef, remote: InetSocketAddress)
2.    extends Actor with ActorLogging {
3.
4.    import Tcp._
5.
```

```
  case class Ack(offset: Int) extends Event

  // sign death pact: this actor terminates when connection breaks
  context watch connection

  // start out in optimistic write-through mode
  def receive = writing

  def writing: Receive = {
    case Received(data) =>
      connection ! Write(data, Ack(currentOffset))
      buffer(data)

    case Ack(ack) =>
      acknowledge(ack)

    case CommandFailed(Write(_, Ack(ack))) =>
      connection ! ResumeWriting
      context become buffering(ack)

    case PeerClosed =>
      if (storage.isEmpty) context stop self
      else context become closing
  }

  // buffering ...

  // closing ...

  override def postStop(): Unit = {
    log.info(s"transferred $transferred bytes from/to [$remote]")
  }

  // storage omitted ...
}
  // storage omitted ...
```

这里的原则是保持写，直到接收到一个 `CommandFailed` ，仅使用确认来

修剪重发缓冲区。当收到了此类故障时，过渡到一个不同的处理状态，并处理重发队列中的所有数据：

```
1.  def buffering(nack: Int): Receive = {
2.    var toAck = 10
3.    var peerClosed = false
4.
5.    {
6.      case Received(data)         => buffer(data)
7.      case WritingResumed         => writeFirst()
8.      case PeerClosed             => peerClosed = true
9.      case Ack(ack) if ack < nack => acknowledge(ack)
10.     case Ack(ack) =>
11.       acknowledge(ack)
12.       if (storage.nonEmpty) {
13.         if (toAck > 0) {
14.           // stay in ACK-based mode for a while
15.           writeFirst()
16.           toAck -= 1
17.         } else {
18.           // then return to NACK-based again
19.           writeAll()
20.           context become (if (peerClosed) closing else writing)
21.         }
22.       } else if (peerClosed) context stop self
23.       else context become writing
24.   }
25. }
```

应指出的是当前缓冲的所有写操作，在进入这种状态时也都已被发送到连接actor，这意味着 `ResumeWriting` 消息是在那些写操作之后入队的，导致接收所有突出的 `CommandFailed` 消息 （在这种状态被忽略）在收到 `WritingResumed` 信号之前。后一种消息只在内部排队的写操作完全完成时，由连接actor发送，意味着一个后续的写操作不会失败。这被 `EchoHandler` 用来为头十个写操作切换到基于ACK的方式，在一次失

败后和恢复到乐观完全写行为之前。

```
def closing: Receive = {
  case CommandFailed(_: Write) =>
    connection ! ResumeWriting
    context.become({

      case WritingResumed =>
        writeAll()
        context.unbecome()

      case ack: Int => acknowledge(ack)

    }, discardOld = false)

  case Ack(ack) =>
    acknowledge(ack)
    if (storage.isEmpty) context stop self
}
```

仍在发送所有数据时关闭连接，相比基于ACK的办法涉及更多内容：这个想法是，总是发送所有未完成的消息和确认所有成功的写操作，并且如果发生故障则切换行为等待 `WritingResumed` 事件并重新开始。

辅助函数非常类似于基于 ACK 的例子：

```
private def buffer(data: ByteString): Unit = {
  storage :+= data
  stored += data.size

  if (stored > maxStored) {
    log.warning(s"drop connection to [$remote] (buffer overrun)")
    context stop self

  } else if (stored > highWatermark) {
    log.debug(s"suspending reading at $currentOffset")
    connection ! SuspendReading
```

```
12.       suspended = true
13.     }
14.   }
15.
16.   private def acknowledge(ack: Int): Unit = {
17.     require(ack == storageOffset, s"received ack $ack at
    $storageOffset")
18.     require(storage.nonEmpty, s"storage was empty at ack $ack")
19.
20.     val size = storage(0).size
21.     stored -= size
22.     transferred += size
23.
24.     storageOffset += 1
25.     storage = storage drop 1
26.
27.     if (suspended && stored < lowWatermark) {
28.       log.debug("resuming reading")
29.       connection ! ResumeReading
30.       suspended = false
31.     }
32.   }
```

带有拉取模式高压写

当使用拉取读时，来自套接字的数据在可用时会被尽快发送到actor。在前面的Echo服务器示例中的情况下，这意味着我们需要维护一个传入数据缓冲区来保持它，因为写入的速度可能会低于新数据到达的速度。

在拉取模式下这个缓冲区可以完全消除，如下面的代码段所示：

```
1.  override def preStart: Unit = connection ! ResumeReading
2.
3.  def receive = {
4.    case Received(data) => connection ! Write(data, Ack)
```

```
5.    case Ack            => connection ! ResumeReading
6.  }
```

这里的想法是，直到以前写操作已被连接actor完全确认才恢复读。每个拉取模式连接actor都是从挂起状态开始。要启动数据流我们发送一个 `ResumeReading` 到 `preStart` 方法告诉连接actor我们准备好接收第一个块数据了。因为我们只在先前的数据块已经被完全写入的情况下才恢复读，所以就没有必要维持一个缓冲区。

要对一个出站连接启用拉取读，`Connect` 的 `pullMode` 参数应被设置为 `true`：

```
1.  IO(Tcp) ! Connect(listenAddress, pullMode = true)
```

入站连接的拉取读模式

前一节演示了如何为出站连接启用拉取读模式，但也可能创建一个监听器actor具有这种读模式，通过设置 `Bind` 命令的 `pullMode` 参数设置为 `true`：

```
1.  IO(Tcp) ! Bind(self, new InetSocketAddress("localhost", 0),
    pullMode = true)
```

此设置的影响之一是此监听器actor所接受的所有连接将都使用拉取读模式。

此设置的另一个影响是除了所有的入站连接被设置为拉取读模式，接受连接也变为基于拉取的了。这意味着在处理一个（或多个）`Connected` 事件后监听器actor一定要通过发送它一条 `ResumeAccepting` 消息来恢复。

拉取模式中的侦听器actor以挂起方式开始来接受连接，在绑定成功后必须发送 `ResumeAccepting` 命令到监听actor：

```
case Bound(localAddress) =>
  // Accept connections one by one
  sender ! ResumeAccepting(batchSize = 1)
  context.become(listening(sender))
```

在处理传入的连接后，我们需要再一次恢复接收：

```
def listening(listener: ActorRef): Receive = {
  case Connected(remote, local) =>
    val handler = context.actorOf(Props(classOf[PullEcho], sender))
    sender ! Register(handler, keepOpenOnPeerClosed = true)
    listener ! ResumeAccepting(batchSize = 1)
}
```

`ResumeAccepting` 接受一个 `batchSize` 参数，指定在需要下一个 `ResumeAccepting` 消息恢复处理新的连接之前，接受多少新的连接。

使用UDP

使用UDP

> 注：本节未经校验，如有问题欢迎提*issue*

UDP 无连接的数据报协议，在JDK 级别上提供两种不同的通信方式：

- 套接字可以自由地发送数据报到任何目的地，并从任何来源接收数据报
- 套接字被限定只和一个特定的远程套接字地址通信

低级API中区分是——令人困惑地——通过是否在套接字上调用 `connect` 方法（甚至当 `connect` 被调用，协议仍然是无连接的）。UDP 使用的这两种形式是使用不同的 IO 扩展提供的，如下所述。

无连接UDP

简单发送

```
1.  class SimpleSender(remote: InetSocketAddress) extends Actor {
2.    import context.system
3.    IO(Udp) ! Udp.SimpleSender
4.
5.    def receive = {
6.      case Udp.SimpleSenderReady =>
7.        context.become(ready(sender()))
8.    }
```

```
9.
10.   def ready(send: ActorRef): Receive = {
11.     case msg: String =>
12.       send ! Udp.Send(ByteString(msg), remote)
13.   }
14. }
```

UDP 使用最简单的形式是只发送数据报，而不需要得到回复。为此目的一个”简单的发送者”工具如上所示。UDP 扩展使用 `SimpleSender` 消息查询 ，由 `SimpleSenderReady` 通知回答。此消息的发送者是新创建的发送者actor，从此时起可以用于将数据报发送到任意的目的地 ；在此示例中，它将只发送任何收到的utf-8编码的 `String` 到一个预定义的远程地址。

> 注意
>
> 简单的发送者不会关闭本身，因为它无法知道什么时候完成工作了。当你想要关闭该发送者的短暂绑定的端口时，你需要发送一个 *`PoisonPill`* 给它。

绑定（和发送）

```
1. class Listener(nextActor: ActorRef) extends Actor {
2.   import context.system
3.   IO(Udp) ! Udp.Bind(self, new InetSocketAddress("localhost", 0))
4.
5.   def receive = {
6.     case Udp.Bound(local) =>
7.       context.become(ready(sender()))
8.   }
9.
10.   def ready(socket: ActorRef): Receive = {
11.     case Udp.Received(data, remote) =>
12.       val processed = // parse data etc., e.g. using PipelineStage
13.       socket ! Udp.Send(data, remote) // example server echoes back
14.       nextActor ! processed
15.     case Udp.Unbind  => socket ! Udp.Unbind
16.     case Udp.Unbound => context.stop(self)
```

```
17.   }
18. }
```

如果你想要实现一个 UDP 服务器侦听套接字来接收传入数据报，则你需要使用 `Bind` 命令，如上所示。指定的本地地址可能会有一个为零的端口，此时操作系统会自动选择一个自由端口并将它分配给新的套接字。通过检查 `Bound` 消息可以发现实际上绑定的是哪个端口。

`Bound` 消息的发送者是管理新套接字的那个actor。发送数据报是通过使用 `Send` 消息类型实现的，并且套接字可以通过发送一个 `Unbind` 命令来关闭，这种情况下套接字actor会回复一个 `Unbound` 通知。

接收的数据报发送到被 `Bind` 消息指定的actor， `Bound` 消息将被发送给 `Bind` 的发送者。

连接的 UDP

通过基于连接的 UDP API 提供的服务，类似于我们前面所述的bind-and-send服务，但主要区别是连接只是能够将发送到它连接到de `remoteAddress` ，并将只能从该地址接收数据报。

```
1.  class Connected(remote: InetSocketAddress) extends Actor {
2.    import context.system
3.    IO(UdpConnected) ! UdpConnected.Connect(self, remote)
4.
5.    def receive = {
6.      case UdpConnected.Connected =>
7.        context.become(ready(sender()))
8.    }
9.
10.   def ready(connection: ActorRef): Receive = {
11.     case UdpConnected.Received(data) =>
12.       // process data, send it on, etc.
```

```
13.       case msg: String =>
14.         connection ! UdpConnected.Send(ByteString(msg))
15.       case d @ UdpConnected.Disconnect => connection ! d
16.       case UdpConnected.Disconnected   => context.stop(self)
17.     }
18.   }
```

因此在这里的例子看起来非常类似于前一个示例，最大的区别是 `Send` 和 `Received` 消息中没有远程地址信息。

> 注意
>
> 还有相比无连接的，使用基于连接的 *UDP API*有小的性能好处。如果系统启用了一个安全管理器，每个无连接的消息发送要经过一个安全检查，而在基于连接的 *UDP* 的情况下，安全检查连接后被缓存，因此写操作不会遭受额外的性能惩罚。

ZeroMQ

ZeroMQ

> 注： 直接从*Akka 2.0*文档中复制，没有仔细检查

Akka提供一个 ZeroMQ 模块对 ZeroMQ 连接进行抽象从而允许 Akka actor之间在ZeroMQ连接之上进行消息交互。这些消息可以是专有格式或者使用Protobuf来定义。socket actor缺省就具有容错性，当你调用newSocket方法创建新的Socket时它会恰当地对socket进行重新初始化。

ZeroMQ 在多线程方面有较大的强制性，所以配置选项 `akka.zeromq.socket-dispatcher` 一定要设置成 PinnedDispatcher，这是因为实际的ZeroMQ socket只能在创建它的线程中访问。

Akka的ZeroMQ模块是按照JZMQ中的API编写的，JZMQ使用JNI与本地ZeroMQ库通信，但Akka ZeroMQ模块没有用JZMQ，用的是ZeroMQ的Scala绑定，与本地ZeroMQ库通信是通过JNA完成的. 换句话说，这个模块所需要的本地库只有本地 ZeroMQ 库. 使用scala库的好处是你不需要编译和管理本地的信赖，当然这会有一些性能上的损失。Scala绑定与JNI绑定是兼容的，如果你的确需要获得最好的性能，直

接用后者替换前者即可。

> 注意
>
> `zeromq-scala-bindings` 当前所用的版本只兼容 *zeromq 2* ; *zeromq 3* 不受支持。

连接

ZeroMQ 支持多种连接模式, 每一种用来满足不同的需求。 目前这个模块支持发布－订阅型连接和基于Router－Dealer的连接. 为了发起连接或接收连接，必须创建一个socket. Socket都是用akka.zeromq.ZeroMQExtension创建的, 例如:

```
1. import akka.zeromq.ZeroMQExtension
2. val pubSocket = ZeroMQExtension(system).newSocket(SocketType.Pub,
3.   Bind("tcp://127.0.0.1:21231"))
```

或者通过导入 akka.zeromq._ 包来获得隐式的newSocket方法.

```
1. import akka.zeromq._
2. val pubSocket2 = system.newSocket(SocketType.Pub,
   Bind("tcp://127.0.0.1:1234"))
```

上例将在本机的1234端口创建一个ZeroMQ发布者socket。.

类似的你可以创建一个带有监听器的订阅socket, 从发布者订阅所有的消息:

```
1. import akka.zeromq._
2. val listener = system.actorOf(Props(new Actor {
3.   def receive: Receive = {
4.     case Connecting    ⇒ //...
5.     case m: ZMQMessage ⇒ //...
6.     case _             ⇒ //...
7.   }
8. }))
```

```
9.  val subSocket = system.newSocket(SocketType.Sub,
    Listener(listener), Connect("tcp://127.0.0.1:1234"), SubscribeAll)
```

下面的章节将描述所支持的连接方式以及它们在Akka环境中的用法。但是，如果要详细了解各种连接方式，请参阅 ZeroMQ – The Guide.

发布-订阅型连接

在一个发布-订阅(pub-sub)型连接中， 一个发布者可以有多个订阅者. 每个订阅者订阅一个或多个主题 ， 发布者向主题发布消息. 订阅者也可以订阅所有的主题. 在 Akka 环境中， 如果要向并不直接与某actor打交道的actor发布消息，需要使用pub-sub 连接.

在使用 zeromq pub/sub 时你必须知道它需要多播 - 检查你的云环境 - 才能正确工作，同时对事件主题的过滤发生在客户端， 因此所有的事件都会广播到所有的订阅者.

actor订阅主题的过程如下:

```
1.  val subTopicSocket = system.newSocket(SocketType.Sub,
    Listener(listener), Connect("tcp://127.0.0.1:1234"),
    Subscribe("foo.bar"))
```

订阅时使用前缀匹配，所以它订阅了所有以 foo.bar 开头的主题. 注意，如果没有提供主题名称或使用了 SubscribeAll ， actor 将订阅所有的主题.

要取消订阅:

```
1.  subTopicSocket ! Unsubscribe("foo.bar")
```

要向主题发布消息你必须使用两个 Frames，第一个Frame是主题.

```
pubSocket ! ZMQMessage(Seq(Frame("foo.bar"), Frame(payload)))
```

Pub-Sub实战

下例演示了一个有两个订阅者的发布者.

发布者监视当前的堆使用量和系统负载并周期性在”health.heap”主题上发布 Heap 事件， 在”health.load”主题上发布 Load 事件.

```
import akka.zeromq._
import akka.actor.Actor
import akka.actor.Props
import akka.actor.ActorLogging
import akka.serialization.SerializationExtension
import java.lang.management.ManagementFactory

case object Tick
case class Heap(timestamp: Long, used: Long, max: Long)
case class Load(timestamp: Long, loadAverage: Double)

class HealthProbe extends Actor {

  val pubSocket = context.system.newSocket(SocketType.Pub,
Bind("tcp://127.0.0.1:1235"))
  val memory = ManagementFactory.getMemoryMXBean
  val os = ManagementFactory.getOperatingSystemMXBean
  val ser = SerializationExtension(context.system)

  override def preStart() {
    context.system.scheduler.schedule(1 second, 1 second, self,
Tick)
  }

  override def postRestart(reason: Throwable) {
    // 还要调用preStart, 仅调度一次
  }
```

```
26.
27.   def receive: Receive = {
28.     case Tick ⇒
29.       val currentHeap = memory.getHeapMemoryUsage
30.       val timestamp = System.currentTimeMillis
31.
32.       // 用 akka SerializationExtension 转换成字节
33.       val heapPayload = ser.serialize(Heap(timestamp,
    currentHeap.getUsed, currentHeap.getMax)).fold(throw _, identity)
34.       // 第一Frame是话题，第二Frame是消息
35.       pubSocket ! ZMQMessage(Seq(Frame("health.heap"),
    Frame(heapPayload)))
36.
37.       // 用 akka SerializationExtension 转换成字节
38.       val loadPayload = ser.serialize(Load(timestamp,
    os.getSystemLoadAverage)).fold(throw _, identity)
39.       // 第一Frame是主题，第二Frame是消息
40.       pubSocket ! ZMQMessage(Seq(Frame("health.load"),
    Frame(loadPayload)))
41.   }
42. }
43.
44.   system.actorOf(Props[HealthProbe], name = "health")
```

我们添加一个订阅者来记录日志. 它订阅所有以 “health”开头的主题, i.e. 包括 Heap 和 Load 事件.

```
1.  class Logger extends Actor with ActorLogging {
2.
3.    context.system.newSocket(SocketType.Sub, Listener(self),
    Connect("tcp://127.0.0.1:1235"), Subscribe("health"))
4.    val ser = SerializationExtension(context.system)
5.    val timestampFormat = new SimpleDateFormat("HH:mm:ss.SSS")
6.
7.    def receive = {
8.      // 第一Frame是主题，第二Frame是消息
9.      case m: ZMQMessage if m.firstFrameAsString == "health.heap" ⇒
```

```
10.       ser.deserialize(m.payload(1), classOf[Heap]) match {
11.         case Right(Heap(timestamp, used, max)) ⇒
12.           log.info("Used heap {} bytes, at {}", used,
    timestampFormat.format(new Date(timestamp)))
13.         case Left(e) ⇒ throw e
14.       }
15.
16.     case m: ZMQMessage if m.firstFrameAsString == "health.load" ⇒
17.       ser.deserialize(m.payload(1), classOf[Load]) match {
18.         case Right(Load(timestamp, loadAverage)) ⇒
19.           log.info("Load average {}, at {}", loadAverage,
    timestampFormat.format(new Date(timestamp)))
20.         case Left(e) ⇒ throw e
21.       }
22.   }
23. }
24.
25.   system.actorOf(Props[Logger], name = "logger")
```

另一个订阅者了解堆内存用量，如果堆使用过量则提出警告. 它仅订阅 Heap 事件.

```
1.  class HeapAlerter extends Actor with ActorLogging {
2.
3.    context.system.newSocket(SocketType.Sub, Listener(self),
    Connect("tcp://127.0.0.1:1235"), Subscribe("health.heap"))
4.    val ser = SerializationExtension(context.system)
5.    var count = 0
6.
7.    def receive = {
8.      // 第一Frame是主题, 第二Frame是消息
9.      case m: ZMQMessage if m.firstFrameAsString == "health.heap" ⇒
10.       ser.deserialize(m.payload(1), classOf[Heap]) match {
11.         case Right(Heap(timestamp, used, max)) ⇒
12.           if ((used.toDouble / max) > 0.9) count += 1
13.           else count = 0
14.           if (count > 10) log.warning("Need more memory, using {}
```

```
    %", (100.0 * used / max))
15.         case Left(e) ⇒ throw e
16.       }
17.   }
18. }
19.
20.   system.actorOf(Props[HeapAlerter], name = "alerter")
```

Router-Dealer Connection

虽然 Pub/Sub 是很好的连接方式但zeromq的真正优势在于它象一个用于可靠消息通信的 “乐高玩具” . 而由于有如此多的集成方式，它的多语言支持是极好的. 当你使用ZeroMQ来集成多个系统时你可能需要创建自己的ZeroMQ机制。这时就轮到router 和 dealer socket类型发挥作用了. 使用这些socket类型你可以创建自己的使用TCP/IP的可靠pub sub broker，并实现发布方的事件过滤.

要创建配置了高水位的Router socket:

```
1. val highWatermarkSocket = system.newSocket(
2.   SocketType.Router,
3.   Listener(listener),
4.   Bind("tcp://127.0.0.1:1234"),
5.   HighWatermark(50000))
```

akka-zeromq 模块支持大部分zeromq socket的配置选项.

推－拉型连接

Akka ZeroMQ 模块支持 推-拉 型连接.

创建 Push 连接:

```
1. def newPushSocket(socketParameters: Array[SocketOption]): ActorRef
```

创建 Pull 型连接:

```
1.  def newPullSocket(socketParameters: Array[SocketOption]): ActorRef
```

很快将提供有更多的文档和示例.

请求－响应型连接

Akka ZeroMQ 模块支持 请求-响应 型连接.

创建 响应 连接:

```
1.  def newReqSocket(socketParameters: Array[SocketOption]): ActorRef
```

创建 请求 连接:

```
1.  def newRepSocket(socketParameters: Array[SocketOption]): ActorRef
```

很快将提供更多的文档和示例.

Camel

Camel

实用工具

实用工具

事件总线

事件总线

> 注：本节未经校验，如有问题欢迎提*issue*

最初设想是为了提供一种向多个actor群发消息的方法，之后 `EventBus` 被一般化为一组实现一个简单接口的可组合的特质：

```
1.  /**
2.   * Attempts to register the subscriber to the specified Classifier
3.   * @return true if successful and false if not (because it was
    already
4.   *   subscribed to that Classifier, or otherwise)
5.   */
6.  def subscribe(subscriber: Subscriber, to: Classifier): Boolean
7.
8.  /**
9.   * Attempts to deregister the subscriber from the specified
    Classifier
10.  * @return true if successful and false if not (because it wasn't
    subscribed
```

```
11.   *   to that Classifier, or otherwise)
12.   */
13.  def unsubscribe(subscriber: Subscriber, from: Classifier): Boolean
14.
15.  /**
16.   * Attempts to deregister the subscriber from all Classifiers it
     may be subscribed to
17.   */
18.  def unsubscribe(subscriber: Subscriber): Unit
19.
20.  /**
21.   * Publishes the specified Event to this bus
22.   */
23.  def publish(event: Event): Unit
```

> 注意
>
> 请注意 *EventBus* 不会保留发布消息的发件人。如果你需要原始发件人的引用必须在消息内部提供。

这个机制在Akka的多个地方用到，例如事件流. 具体实现可以使用下面列出的特定构建工具块。

一个事件总线必须定义以下三种抽象类型：

- `Event` 所有发布到该总线上的事件的类型
- `Subscriber` 允许注册到该总线上的订阅者的类型
- `Classifier` 定义用来派发消息时选择订阅者的分类器

下面的 `trait` 在这些类型中仍然是泛化的，但它们必须在任何具体的实现中被定义。

类别（Classifiers）

这里提到的类别是Akka发布包的一部分，如果没找到合适的就实现一个自己的类别，并不困难，到 [github]

（http://github.com/akka/akka/tree/v2.3.6/akka-actor/src/main/scala/akka/event/EventBus.scala） 了解已有类别的实现。

查找分类法

最简单的分类法是给每个事件提取一个随机的类别，并为每一种类别维护一组订阅者。这可以用在收音机上选台来类比。

`LookupClassification` trait 仍然是泛化的，抽象了如何比较订阅者，以及具体分类的方法。

需要实现的方法如下：

```
1.  import akka.event.EventBus
2.  import akka.event.LookupClassification
3.
4.  case class MsgEnvelope(topic: String, payload: Any)
5.
6.  /**
7.   * Publishes the payload of the MsgEnvelope when the topic of the
8.   * MsgEnvelope equals the String specified when subscribing.
9.   */
10. class LookupBusImpl extends EventBus with LookupClassification {
11.   type Event = MsgEnvelope
12.   type Classifier = String
13.   type Subscriber = ActorRef
14.
15.   // is used for extracting the classifier from the incoming events
16.   override protected def classify(event: Event): Classifier =
    event.topic
17.
18.   // will be invoked for each event for all subscribers which
    registered themselves
19.   // for the event's classifier
20.   override protected def publish(event: Event, subscriber:
    Subscriber): Unit = {
```

```
21.     subscriber ! event.payload
22.   }
23.
24.   // must define a full order over the subscribers, expressed as
    expected from
25.   // `java.lang.Comparable.compare`
26.   override protected def compareSubscribers(a: Subscriber, b:
    Subscriber): Int =
27.     a.compareTo(b)
28.
29.   // determines the initial size of the index data structure
30.   // used internally (i.e. the expected number of different
    classifiers)
31.   override protected def mapSize: Int = 128
32.
33. }
```

对该实现的测试看上去像这样：

```
1. val lookupBus = new LookupBusImpl
2. lookupBus.subscribe(testActor, "greetings")
3. lookupBus.publish(MsgEnvelope("time", System.currentTimeMillis()))
4. lookupBus.publish(MsgEnvelope("greetings", "hello"))
5. expectMsg("hello")
```

这种分类法在对某个特定事件没有任何订阅者时是高效的。

子频道分类法

如果类别构成一个树形结构而且订阅可以不仅针对叶子结点，这种分类法可能是最合适的。 这可以类比成在安风格划分过的（多个）收音机频道中进行调台。 开发这种分类法是为了用在分类器正好是事件的JVM类，并且订阅者可能对订阅某个特定类的所有子类感兴趣的场合，但是它可以用在任何树形类别体系。

需要实现的方法如下：

```
import akka.util.Subclassification

class StartsWithSubclassification extends Subclassification[String]
{
  override def isEqual(x: String, y: String): Boolean =
    x == y

  override def isSubclass(x: String, y: String): Boolean =
    x.startsWith(y)
}

import akka.event.SubchannelClassification

/**
 * Publishes the payload of the MsgEnvelope when the topic of the
 * MsgEnvelope starts with the String specified when subscribing.
 */
class SubchannelBusImpl extends EventBus with
SubchannelClassification {
  type Event = MsgEnvelope
  type Classifier = String
  type Subscriber = ActorRef

  // Subclassification is an object providing `isEqual` and
`isSubclass`
  // to be consumed by the other methods of this classifier
  override protected val subclassification:
Subclassification[Classifier] =
    new StartsWithSubclassification

  // is used for extracting the classifier from the incoming events
  override protected def classify(event: Event): Classifier =
event.topic

  // will be invoked for each event for all subscribers which
registered
  // themselves for the event's classifier
  override protected def publish(event: Event, subscriber:
```

```
    Subscriber): Unit = {
33.     subscriber ! event.payload
34.   }
35. }
```

对该实现的测试看上去像这样：

```
1. val subchannelBus = new SubchannelBusImpl
2. subchannelBus.subscribe(testActor, "abc")
3. subchannelBus.publish(MsgEnvelope("xyzabc", "x"))
4. subchannelBus.publish(MsgEnvelope("bcdef", "b"))
5. subchannelBus.publish(MsgEnvelope("abc", "c"))
6. expectMsg("c")
7. subchannelBus.publish(MsgEnvelope("abcdef", "d"))
8. expectMsg("d")
```

这种分类法在某事件没有任何订阅者时也很高效，但它使用一个保守锁来对共内部的类别缓存进行同步，所以不适合订阅关系以很高的频率变化的场合（记住通过发送第一个消息来“打开”一个类别，也将需要重新检查所有之前的订阅）。

扫描分类法

上一种分类法是为严格的树形多类别订阅设计的，而这个分类法是用在覆盖事件空间中可能互相重叠的非树形结构类别上的。 这可以比喻为有地理限制的（比如老旧的收音机信号传播方式）的（可能有多个）收音机频道之间进行调台。

需要实现的方法如下：

```
1. import akka.event.ScanningClassification
2.
3. /**
4.  * Publishes String messages with length less than or equal to the
    length
```

```
 * specified when subscribing.
 */
class ScanningBusImpl extends EventBus with ScanningClassification
{
  type Event = String
  type Classifier = Int
  type Subscriber = ActorRef

  // is needed for determining matching classifiers and storing
them in an
  // ordered collection
  override protected def compareClassifiers(a: Classifier, b:
Classifier): Int =
    if (a < b) -1 else if (a == b) 0 else 1

  // is needed for storing subscribers in an ordered collection
  override protected def compareSubscribers(a: Subscriber, b:
Subscriber): Int =
    a.compareTo(b)

  // determines whether a given classifier shall match a given
event; it is invoked
  // for each subscription for all received events, hence the name
of the classifier
  override protected def matches(classifier: Classifier, event:
Event): Boolean =
    event.length <= classifier

  // will be invoked for each event for all subscribers which
registered themselves
  // for a classifier matching this event
  override protected def publish(event: Event, subscriber:
Subscriber): Unit = {
    subscriber ! event
  }
}
```

对该实现的测试看上去像这样：

```
1. val scanningBus = new ScanningBusImpl
2. scanningBus.subscribe(testActor, 3)
3. scanningBus.publish("xyzabc")
4. scanningBus.publish("ab")
5. expectMsg("ab")
6. scanningBus.publish("abc")
7. expectMsg("abc")
```

这种分类法耗费的时间总是与订阅关系的数量成正比，而与实际匹配的数量无关。

Actor 分类法

这种分类法原来是专门为实现 DeathWatch开发的: 订阅者和类别都是 `ActorRef` 类型.

需要实现的方法如下:

```
1. import akka.event.ActorEventBus
2. import akka.event.ActorClassification
3. import akka.event.ActorClassifier
4.
5. case class Notification(ref: ActorRef, id: Int)
6.
7. class ActorBusImpl extends ActorEventBus with ActorClassifier with
   ActorClassification {
8.   type Event = Notification
9.
10.  // is used for extracting the classifier from the incoming events
11.  override protected def classify(event: Event): ActorRef =
   event.ref
12.
13.  // determines the initial size of the index data structure
14.  // used internally (i.e. the expected number of different
   classifiers)
15.  override protected def mapSize: Int = 128
16. }
```

对该实现的测试看上去像这样：

```
1.  val observer1 = TestProbe().ref
2.  val observer2 = TestProbe().ref
3.  val probe1 = TestProbe()
4.  val probe2 = TestProbe()
5.  val subscriber1 = probe1.ref
6.  val subscriber2 = probe2.ref
7.  val actorBus = new ActorBusImpl
8.  actorBus.subscribe(subscriber1, observer1)
9.  actorBus.subscribe(subscriber2, observer1)
10. actorBus.subscribe(subscriber2, observer2)
11. actorBus.publish(Notification(observer1, 100))
12. probe1.expectMsg(Notification(observer1, 100))
13. probe2.expectMsg(Notification(observer1, 100))
14. actorBus.publish(Notification(observer2, 101))
15. probe2.expectMsg(Notification(observer2, 101))
16. probe1.expectNoMsg(500.millis)
```

这种分类器对事件类型仍然是泛化的，它在所有的场合下都是高效的。

事件流" class="reference-link">事件流

事件流是每个actor系统的主事件总线：它用来携带 日志消息 和 死信，并且也可以被用户代码使用来达到其它目的。它使用 子频道分类法 使得可以向一组相关的频道进行注册（就象 `RemoteLifeCycleMessage` 所用的那样）. 以下例子演示了一个简单的订阅是如何工作的：

```
1.  import akka.actor.{ Actor, DeadLetter, Props }
2.
3.  class Listener extends Actor {
4.    def receive = {
5.      case d: DeadLetter => println(d)
6.    }
```

```
7.  }
8.  val listener = system.actorOf(Props(classOf[Listener], this))
9.  system.eventStream.subscribe(listener, classOf[DeadLetter])
```

缺省的处理器

actor系统在启动时会创建一些actor，并为其在事件流上订阅日志消息：这些是缺省的处理器，可以配置在例如 `application.conf` ：

```
1.  akka {
2.    loggers = ["akka.event.Logging$DefaultLogger"]
3.  }
```

这里以全路径类名列出的处理器将订阅所有配置的日志级别以上的日志事件类，并且当日志级别在运行时被修改时这些订阅关系也会同步修改：

```
1.  system.eventStream.setLogLevel(Logging.DebugLevel)
```

这意味着一个低于日志级别的日志事件，事实上根本不会被派发（除非专门为相应的事件类进行了手工订阅）

死信" class="reference-link">死信

正如 终止actors所述， 当actor终止后其邮箱队列中的剩余消息及后续被发送的消息都将被发送到死信邮箱， 它缺省情况下会发布打包在 `DeadLetter` 中的消息. 这种打包动作会保留被重定向消息的原始发送者、接收者以及消息内容。

其它用处

事件流一直存在并可用，你可以向它发布你自己的事件 （它接受 `AnyRef` ） 并对相应的JVM类添加订阅监听器。

日志

日志

> 注：本节未经校验，如有问题欢迎提*issue*

Logging在Akka中不是依赖于一个特定的日志记录后端的。默认日志消息要打印到 STDOUT，但你可以使用插件如SLF4J logger或自己的logger。日志记录是以异步方式执行的，以确保日志记录具有最小的性能影响。Logging一般意味着IO和锁，如果同步执行会减慢你的代码操作。

如何记录日志" class="reference-link">如何记录日志

创建一个 `LoggingAdapter` 并使用它的 `error` , `warning` ,

`info`, 或 `debug` 方法, 如下例所示:

```
1.  import akka.event.Logging
2.
3.  class MyActor extends Actor {
4.    val log = Logging(context.system, this)
5.    override def preStart() = {
6.      log.debug("Starting")
7.    }
8.    override def preRestart(reason: Throwable, message: Option[Any])
    {
9.      log.error(reason, "Restarting due to [{}] when processing
    [{}]",
10.       reason.getMessage, message.getOrElse(""))
11.   }
12.   def receive = {
13.     case "test" => log.info("Received test")
14.     case x      => log.warning("Received unknown message: {}", x)
15.   }
16. }
```

方便起见你可以向actor中混入 `log` 成员, 而不是像上例一下定义它.

```
1.  class MyActor extends Actor with akka.actor.ActorLogging {
2.    ...
3.  }
```

`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:

- 如果它是Actor或ActorRef, 则使用它的路径
- 如果是String, 就使用它自己
- 如果是类, 则使用其simpleName的近似
- 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 `{}`，如果该日志级别被打开，则占位符会被替换。如果给出的参数数量多过 占位符的数量，则在日志语句的结尾将被追加一个警告（即在同一级别的同一行）。你可以只传入一个Java数组来为多个占位符分别提供数据：

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

日志源的 Java `Class` 也会被包含在生成的 `LogEvent` 中. 如果是简单的字符串，它会被替换成一个 “标志” 类 `akka.event.DummyClassForStringSources` 以便对这种情况作特殊处理，例如在SLF4j事件监听器中会使用字符串而不是类名来查找要用到的日志记录器实例。

死信的日志记录

默认情况下发送到死信的消息会在info级别记录日志。死信的存在并不一定表示有问题，但它可能是，因此他们默认情况下被记录。在几个消息之后此日志记录被关闭，以避免日志泛滥。你可以完全禁用此记录或调整记录多少死信。在系统关闭期间，你很可能看到死信，因为在actor邮箱中的待执行消息会发送到死信。你还可以禁用死信在关闭过程中的日志记录。

```
1.  akka {
2.    log-dead-letters = 10
3.    log-dead-letters-during-shutdown = on
4.  }
```

若要进一步自定义日志记录或为死信采取其他行动，你可以订阅事件流。

辅助日志选项

Akka 为非常底层的debug提供了一组配置选项，这些主要是为Akka的开发者所用，而非普通用户。

你一定要将日志级别设为 DEBUG 来使用这些选项：

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

如果你想知道Akka装载了哪些配置设置，下面这个配置选项非常有用：

```
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
}
```

如果你希望记录用户级消息的细节，则使用 `akka.event.LoggingReceive` 包装你的actor行为，并打开 `receive` 选项：

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

如果你希望记录Actor处理的所有自动接收的消息的细节：

```
akka {
  actor {
```

```
    debug {
      # 为所有的 AutoReceiveMessages(Kill, PoisonPill 之类) 打开DEBUG
日志
      autoreceive = on
    }
  }
}
```

如果你希望记录Actor的所有生命周期变化 (重启，死亡等) 的细节:

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

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

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

如果你希望监控对 ActorSystem.eventStream 的订阅/取消订阅:

```
akka {
  actor {
```

```
    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

如果你希望以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的，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
  }
}
```

如果你希望以DEBUG级别查看所有接收到的远程消息：(这些日志是被传输层接收时所记录的，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
  }
}
```

如果你希望以INFO级别查看的消息类型与大于指定限制的负载大小(字节)：

```
    akka {
      remote {
        # Logging of message types with payload size in bytes
```

```
    larger than
4.          # this value. Maximum detected size per message type is
    logged once,
5.          # with an increase threshold of 10%.
6.          # By default this feature is turned off. Activate it by
    setting the property to
7.          # a value in bytes, such as 1000b. Note that for all
    messages larger than this
8.          # limit there will be extra performance and scalability
    cost.
9.          log-frame-size-exceeding = 1000b
10.       }
11.     }
```

同时参阅 TestKit 的日志选项: 跟踪Actor调用.

将日志源转换为字符串和类

在运行时将源对象转换成要插入 `LogEvent` 的源字符串和类的规则是使用隐式参数的方式实现的， 因此是完全可配置的: 只需要创建你自己的 `LogSource[T]` 实例并将它放在创建logger的作用域中即可。

```
1.  import akka.event.LogSource
2.  import akka.actor.ActorSystem
3.
4.  object MyType {
5.    implicit val logSource: LogSource[AnyRef] = new LogSource[AnyRef]
    {
6.      def genString(o: AnyRef): String = o.getClass.getName
7.      override def getClazz(o: AnyRef): Class[_] = o.getClass
8.    }
9.  }
10.
11. class MyType(system: ActorSystem) {
12.   import MyType._
13.   import akka.event.Logging
14.
```

```
15.    val log = Logging(system, this)
16.  }
```

这个例子创建了一个日志源来模拟Java logger的传统用法，该日志源使用原对象的类名作为日志类别。 在这里加入对 getClazz 的重写只是为了作说明，因为它精确地包含了缺省行为。

> 注意
>
> 你也可以先创建字符串然后将它作为日志源传入，但要知道这时放入 *LogEvent* 中的 *Class[_]* 将是 *akka.event.DummyClassForStringSources* .
>
> *SLF4J* 事件监听器对这种情况会特殊处理 *(*使用实际的字符串来查找*logger*实例而不是类名*)*, 你在实现自己的日志适配器时可能也会这么做。

关闭日志记录

要关闭日志记录，你可以配置日志级别为 `OFF` 像这样。

```
1.  akka {
2.    stdout-loglevel = "OFF"
3.    loglevel = "OFF"
4.  }
```

`stdout-loglevel` 将仅在系统启动和关闭时起作用，并且也设置为 `OFF` ，以确保在系统启动或关闭过程中没有日志记录。

事件处理器

日志记录是通过一个事件总线异步地完成的。日志事件都由一个actor事件处理程序来处理，它将按照它们被发送的相同顺序接收日志事件。

你可以配置的系统启动时创建哪一个事件处理程序来监听日志记录事件。这是使用配置中的 `loggers` 元素。你还可以在这里定义日志级别。

```
1.  akka {
2.    # 在启动时注册的事件处理器 (akka.event.Logging$DefaultLogger 记录日志到
```

```
    标准输出)
3.    loggers = ["akka.event.Logging$DefaultLogger"]
4.    # Options: OFF, ERROR, WARNING, INFO, DEBUG
5.    loglevel = "DEBUG"
6.  }
```

缺省会注册一个事件处理器，它将日志记录到标准输出. 在生产系统中不建议使用它。 在 ‘akka-slf4j’ 模块中还有一个 SLF4J logger。

创建监听器的示例:

```
1.  import akka.event.Logging.InitializeLogger
2.  import akka.event.Logging.LoggerInitialized
3.  import akka.event.Logging.Error
4.  import akka.event.Logging.Warning
5.  import akka.event.Logging.Info
6.  import akka.event.Logging.Debug
7.
8.  class MyEventListener extends Actor {
9.    def receive = {
10.     case InitializeLogger(_)                          => sender() !
    LoggerInitialized
11.     case Error(cause, logSource, logClass, message) => // ...
12.     case Warning(logSource, logClass, message)      => // ...
13.     case Info(logSource, logClass, message)         => // ...
14.     case Debug(logSource, logClass, message)        => // ...
15.   }
16. }
```

在启动和关闭时记录日志到stdout

当actor系统启动和关闭配置的 `loggers` 不会被使用。相反日志消息会打印到标准输出stdout（System.out）。对这个stdout logger的默认的日志级别是 `WARNING` ，并可以通过设置 `akka.stdout-`

`loglevel=OFF` 使其完全沉默。

SLF4J" class="reference-link">SLF4J

Akka 为 SL4FJ提供了一个logger. 它在 ‘akka-slf4j.jar’ 模块中. 它的唯一依赖是 slf4j-api 包. 在运行时，你还需要一个 SLF4J 后端, 我们推荐 Logback:

```
1.  lazy val logback = "ch.qos.logback" % "logback-classic" % "1.0.13"
```

你需要打开配置中的’loggers’来启用 Slf4jLogger。 你还可以在这里定义事件总线的日志级别。更细粒度的日志级别可以在SLF4j后端的配置（例如: logback.xml）中定义。

```
1.  akka {
2.    loggers = ["akka.event.slf4j.Slf4jLogger"]
3.    loglevel = "DEBUG"
4.  }
```

一个隐含的迷惑是，时间戳在事件处理程序中标记，而不是实际记录时。

SLF4J 为每个日志事件选择日志记录器是基于创建 `LoggingAdapter` 时的日志源的 `Class[_]` ，除非日志源是一个字符串，那么就使用这个串 (即 `LoggerFactory.getLogger(c: Class[_]` 用于前者而 `LoggerFactory.getLogger(s: String)` 用于后者).

> 注意
>
> 如果创建*LoggingAdapter*时向工厂方法提供了一个 *ActorSystem* ，那么该*Actor*系统的名字将被加在 *String* 日志源的后面。 如果不希望这样，像下面这样传一个 *LoggingBus* :

```
1.  val log = Logging(system.eventStream, "my.nice.string")
```

用MDC记录线程日志与Akka源

因为日志的记录是异步的，完成日志记录的线程被保存在 Mapped Diagnostic Context (MDC) 的 `sourceThread` 属性里. 在 Logback 的模式布局配置中线程名可以通过 %X{sourceThread} 指定:

```
1. <appender name="STDOUT"
   class="ch.qos.logback.core.ConsoleAppender">
2.   <encoder>
3.     <pattern>%date{ISO8601} %-5level %logger{36} %X{sourceThread} -
   %msg%n</pattern>
4.   </encoder>
5. </appender>
```

> 注意
>
> 最好在应用程序的非*Akka*的部分也使用*MDC*的 `sourceThread` ，使得这个值在日志里保持一致。

Akka 的另一个有用的工具是创建logger实例时会捕捉 actor 的地址，这意味着可以访问到actor实例的全部身份信息来将日志信息与其它信息进行关联（例如：路由的成员）。 这个信息保存在MDC的 `akkaSource` 属性中:

```
1. <appender name="STDOUT"
   class="ch.qos.logback.core.ConsoleAppender">
2.   <encoder>
3.     <pattern>%date{ISO8601} %-5level %logger{36} %X{akkaSource} -
   %msg%n</pattern>
4.   </encoder>
5. </appender>
```

要了解这个属性所包括的内容（也适用于非actor）的细节见 如何记录日志.

MDC中日志输出更准确的时间戳

Akka的日志记录是异步的，这意味着日志条目的时间戳取的是调用底层的记录器实现的时间，这一开始是令人惊讶的。如果你想要更准确地输出时间戳，使用 MDC 属性 `akkaTimestamp` ：

```
1. <appender name="STDOUT"
   class="ch.qos.logback.core.ConsoleAppender">
2.   <encoder>
3.     <pattern>%X{akkaTimestamp} %-5level %logger{36} %X{akkaSource}
   - %msg%n</pattern>
4.   </encoder>
5. </appender>
```

由应用程序定义的 MDC 值

Slf4j一个很有用的功能是MDC，Akka有一种方法让应用程序指定自定义值，你只需要湖区特殊
的 `LoggingAdapter` —— `DiagnosticLoggingAdapter` 。为了得到它，你将使用工厂接收actor作为logSource:

```
1. // Within your Actor
2. val log: DiagnosticLoggingAdapter = Logging(this);
```

一旦你有了记录器，你只需要在记录日志之前先添加自定义的值。这种方式下，这些值将在日志追加前放在 SLF4J MDC中，并在追加后移除。

> 注
>
> 清理 （清除） 应该在*actor*之后处理，否则，如果它不设置新的*map*, 下一条消息海将使用相同的 *mdc* 值。使用 `log.clearMDC()` 。

```
1. val mdc = Map("requestId" -> 1234, "visitorId" -> 5678)
2. log.mdc(mdc)
3.
```

```
// Log something
log.info("Starting new request")

log.clearMDC()
```

方便起见你可以在actor中混入 `log` 成员，而不是如上定义。这一特质还允许你重写 `def mdc(msg: Any): MDC` 用于为当前消息指定 MDC值，并允许你忘记清理，因为它已经为你做过了。

```
import Logging.MDC

case class Req(work: String, visitorId: Int)

class MdcActorMixin extends Actor with
akka.actor.DiagnosticActorLogging {
  var reqId = 0

  override def mdc(currentMessage: Any): MDC = {
    reqId += 1
    val always = Map("requestId" -> reqId)
    val perMessage = currentMessage match {
      case r: Req => Map("visitorId" -> r.visitorId)
      case _      => Map()
    }
    always ++ perMessage
  }

  def receive: Receive = {
    case r: Req => {
      log.info(s"Starting new request: ${r.work}")
    }
  }
}
```

现在，这些值将可用在MDC中，因此你可以在布局模式中使用它们：

```
<appender name="STDOUT"
```

```
    class="ch.qos.logback.core.ConsoleAppender">
2.    <encoder>
3.      <pattern>
4.        %-5level %logger{36} [req: %X{requestId}, visitor:
    %X{visitorId}] - %msg%n
5.      </pattern>
6.    </encoder>
7.  </appender>
```

调度器

调度器

> 注：本节未经校验，如有问题欢迎提*issue*

有时需要设定将来发生的事情，这时该怎么办？ `ActorSystem` 就能搞定一切！ 在那儿你能找到 `scheduler` 方法，它返回一个 `akka.actor.Scheduler` 实例， 这个实例在每个Actor系统里是唯一的，用来在内部指定一段时间后发生的行为。

请注意定时任务是使用 ActorSystem 的 MessageDispatcher 执行的.

你可以计划向actor发送消息或执行任务（函数或Runnable). 你会得到一个 `Cancellable` 类型的返回值，你可以调用 `cancel` 来取消定时操作的执行。

> 警告
>
> *Akka*中使用的 *`Scheduler`* 的默认实现是基于根据一个固定的时间表清空的工作桶。它不是在确切的时间执行任务，而是在每个时间刻度，它将运行 （结束） 到期的一切。可以通过 *`akka.scheduler.tick-duration`* 配置属性修改默认 *`Scheduler`* 的时间精度。

示例

```
1.  import akka.actor.Actor
```

```
import akka.actor.Props
import scala.concurrent.duration._

    //Use the system's dispatcher as ExecutionContext
    import system.dispatcher

    //Schedules to send the "foo"-message to the testActor after 50ms
    system.scheduler.scheduleOnce(50 milliseconds, testActor, "foo")
```

```
//Schedules a function to be executed (send a message to the testActor) after 50ms
system.scheduler.scheduleOnce(50 milliseconds) {
  testActor ! System.currentTimeMillis
}
```

```
val Tick = "tick"
class TickActor extends Actor {
  def receive = {
    case Tick => //Do something
  }
}
val tickActor = system.actorOf(Props(classOf[TickActor], this))
//Use system's dispatcher as ExecutionContext
import system.dispatcher

//This will schedule to send the Tick-message
//to the tickActor after 0ms repeating every 50ms
val cancellable =
  system.scheduler.schedule(0 milliseconds,
    50 milliseconds,
    tickActor,
    Tick)

//This cancels further Ticks to be sent
cancellable.cancel()
```

> 警告
>
> 如果你计划函数或*Runnable*实例时应该多加小心，不要关闭*(*闭合包含*)*不稳定的引用。在实践中这意味着在*Actor*实例中，不要在闭包中使用 `this` ，不直接访问 `sender()` 并且不要直接调用*actor*实例的方法。如果你需要安排一个调用，则安排一个发往 `self` 的消息（包含必需的参数），然后在收到消息时再调用方法。

来自 `akka.actor.ActorSystem`

```
/**
 * Light-weight scheduler for running asynchronous tasks after some deadline
 * in the future. Not terribly precise but cheap.
 */
def scheduler: Scheduler
```

Scheduler 接口

实际的调度程序实现是在 `ActorSystem` 启动时反射加载的，这意味着通过使用 `akka.scheduler.implementation` 配置属性它可能提供一个的不同实现。引用的类必须实现以下接口：

```
/**
 * An Akka scheduler service. This one needs one special behavior: if
 * Closeable, it MUST execute all outstanding tasks upon .close() in order
 * to properly shutdown all dispatchers.
 *
 * Furthermore, this timer service MUST throw IllegalStateException if it
 * cannot schedule a task. Once scheduled, the task MUST be executed. If
 * executed upon close(), the task may execute before its timeout.
 *
 * Scheduler implementation are loaded reflectively at ActorSystem start-up
```

```
11.   * with the following constructor arguments:
12.   *  1) the system's com.typesafe.config.Config (from
    system.settings.config)
13.   *  2) a akka.event.LoggingAdapter
14.   *  3) a java.util.concurrent.ThreadFactory
15.   */
16. trait Scheduler {
17.   /**
18.    * Schedules a message to be sent repeatedly with an initial
    delay and
19.    * frequency. E.g. if you would like a message to be sent
    immediately and
20.    * thereafter every 500ms you would set delay=Duration.Zero and
21.    * interval=Duration(500, TimeUnit.MILLISECONDS)
22.    *
23.    * Java & Scala API
24.    */
25.   final def schedule(
26.     initialDelay: FiniteDuration,
27.     interval: FiniteDuration,
28.     receiver: ActorRef,
29.     message: Any)(implicit executor: ExecutionContext,
30.                   sender: ActorRef = Actor.noSender): Cancellable =
31.     schedule(initialDelay, interval, new Runnable {
32.       def run = {
33.         receiver ! message
34.         if (receiver.isTerminated)
35.           throw new SchedulerException("timer active for terminated
    actor")
36.       }
37.     })
38.
39.   /**
40.    * Schedules a function to be run repeatedly with an initial
    delay and a
41.    * frequency. E.g. if you would like the function to be run after
    2 seconds
42.    * and thereafter every 100ms you would set delay = Duration(2,
```

```
    TimeUnit.SECONDS)
43.     * and interval = Duration(100, TimeUnit.MILLISECONDS)
44.     *
45.     * Scala API
46.     */
47.   final def schedule(
48.     initialDelay: FiniteDuration,
49.     interval: FiniteDuration)(f: ⇒ Unit)(
50.       implicit executor: ExecutionContext): Cancellable =
51.     schedule(initialDelay, interval, new Runnable { override def
    run = f })
52.
53.   /**
54.     * Schedules a function to be run repeatedly with an initial
    delay and
55.     * a frequency. E.g. if you would like the function to be run
    after 2
56.     * seconds and thereafter every 100ms you would set delay =
    Duration(2,
57.     * TimeUnit.SECONDS) and interval = Duration(100,
    TimeUnit.MILLISECONDS)
58.     *
59.     * Java API
60.     */
61.   def schedule(
62.     initialDelay: FiniteDuration,
63.     interval: FiniteDuration,
64.     runnable: Runnable)(implicit executor: ExecutionContext):
    Cancellable
65.
66.   /**
67.     * Schedules a message to be sent once with a delay, i.e. a time
    period that has
68.     * to pass before the message is sent.
69.     *
70.     * Java & Scala API
71.     */
72.   final def scheduleOnce(
```

```
73.     delay: FiniteDuration,
74.     receiver: ActorRef,
75.     message: Any)(implicit executor: ExecutionContext,
76.                   sender: ActorRef = Actor.noSender): Cancellable =
77.     scheduleOnce(delay, new Runnable {
78.       override def run = receiver ! message
79.     })
80.
81.   /**
82.    * Schedules a function to be run once with a delay, i.e. a time
    period that has
83.    * to pass before the function is run.
84.    *
85.    * Scala API
86.    */
87.   final def scheduleOnce(delay: FiniteDuration)(f: ⇒ Unit)(
88.     implicit executor: ExecutionContext): Cancellable =
89.     scheduleOnce(delay, new Runnable { override def run = f })
90.
91.   /**
92.    * Schedules a Runnable to be run once with a delay, i.e. a time
    period that
93.    * has to pass before the runnable is executed.
94.    *
95.    * Java & Scala API
96.    */
97.   def scheduleOnce(
98.     delay: FiniteDuration,
99.     runnable: Runnable)(implicit executor: ExecutionContext):
    Cancellable
100.
101.   /**
102.    * The maximum supported task frequency of this scheduler, i.e.
    the inverse
103.    * of the minimum time interval between executions of a recurring
    task, in Hz.
104.    */
105.   def maxFrequency: Double
```

```
106.
107.  }
```

Cancellable 接口

它使你可以 取消 计划执行的任务。

警告
它不会中止已经启动的任务的执行。

调度的任务会返回一个 `Cancellable` （或抛出 `IllegalStateException` 时，如果在调度器关闭后尝试使用）。这允许你取消原定执行的东西。

> 警告
>
> 这不会中止已经开始执行的任务。检查 *`cancel`* 的返回值以检测已计划的任务是被取消，还是（最终）被运行。

```
1.  /**
2.   * Signifies something that can be cancelled
3.   * There is no strict guarantee that the implementation is thread-
    safe,
4.   * but it should be good practice to make it so.
5.   */
6.  trait Cancellable {
7.    /**
8.     * Cancels this Cancellable and returns true if that was
    successful.
9.     * If this cancellable was (concurrently) cancelled already, then
    this method
10.    * will return false although isCancelled will return true.
11.    *
12.    * Java & Scala API
13.    */
14.   def cancel(): Boolean
15.
```

```
16.   /**
17.    * Returns true if and only if this Cancellable has been
    successfully cancelled
18.    *
19.    * Java & Scala API
20.    */
21.   def isCancelled: Boolean
22. }
```

Duration

Duration

> 注：本节未经校验，如有问题欢迎提*issue*

Duration在Akka库中被广泛使用，这代表一个特殊的数据类型——`scala.concurrent.duration.Duration`。这个类型的值可以表示无限(`Duration.Inf`，`Duration.MinusInf`)或有限的时间段，或是`Duration.Undefined`。

有限与无限

试图将无限的duration转换成一个具体的时间单位，如秒，将引发异常，在编译时有不同类型可用于区分两者：

- `FiniteDuration`保证是有限的，调用`toNanos`和相关方法是安全的
- `Duration`可以是有限或无限的，因此这种类型只用与有限性并不重要的场合；这是`FiniteDuration`的超类。

Scala

在Scala，时间段可以通过一个迷你DSL来创建，并支持所有期望的算术操作：

```
1.  import scala.concurrent.duration._
```

```
2.
3.  val fivesec = 5.seconds
4.  val threemillis = 3.millis
5.  val diff = fivesec - threemillis
6.  assert(diff < fivesec)
7.  val fourmillis = threemillis * 4 / 3 // you cannot write it the
    other way around
8.  val n = threemillis / (1 millisecond)
```

> 注意
>
> 如果表达式划定了明显的边界（例如在括号里或在参数列表里），你可以省略“.”，但是如果时间单位是一行代码的最后一个词，建议你加上它，否则行末分号推断可能会出错，这取决于下一行是如何开始的。
>
> *Java*
>
> *Java*提供的语法糖比较少，所以你必须用方法调用来拼出要进行的操作：

```
1.  import scala.concurrent.duration.Duration;
2.  import scala.concurrent.duration.Deadline;
```

```
1.  final Duration fivesec = Duration.create(5, "seconds");
2.  final Duration threemillis = Duration.create("3 millis");
3.  final Duration diff = fivesec.minus(threemillis);
4.  assert diff.lt(fivesec);
5.  assert Duration.Zero().lt(Duration.Inf());
```

Deadline

Duration 有一个兄弟类，名为 `Deadline`，表示一个持有绝对的时间点的类，并且支持通过计算当前时间到deadline之间的差距来生成Duration。当你想要保持一个总体的期限，而无需记录—自己关注使用时间时，这非常有用：

```
1.  val deadline = 10.seconds.fromNow
2.  // do something
```

```
3.  val rest = deadline.timeLeft
```

在Java中使用duration来创建dealine:

```
1.  final Deadline deadline = Duration.create(10, "seconds").fromNow();
2.  final Duration rest = deadline.timeLeft();
```

线路断路器

线路断路器

> 注：本节未经校验，如有问题欢迎提*issue*

为什么使用它们？

线路断路器用于提供稳定性并防止在分布式系统中的级联故障。它们应该结合在远程系统之间的接口使用明智的超时，以防止单个组件的故障拖垮所有组件。

作为一个例子，我们有一个web 应用程序与远程的第三方web服务进行交互。假如第三方已用完了他们的容量，他们的数据库也在高荷载作用下熔化。假设数据库在这种情况下失败，第三方 web 服务用了很长的时间来返回一个错误。这进一步使调用在长一段时间后失败。回到我们的 web 应用程序，用户已注意到其表单提交似乎比需要的使用了更长的时间。当然用户知道要做的是点击刷新按钮，将更多的请求添加到其已在运行的请求中。这最终导致 web 应用程序由于资源枯竭而失败。

这将会影响所有用户，甚至是那些没有使用依赖于此第三方 web 服务的功能的。

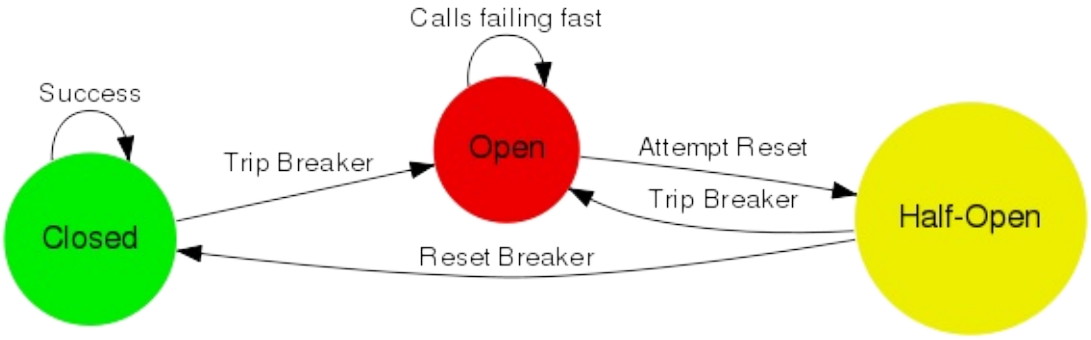
为 web 服务的调用引入断路器会导致请求开始快速失败，让用户知道有什么地方不对劲，并且他们不需要刷新他们的请求。这还局限失效行为只影响到那些正在使用此第三方功能依赖的用户，因为没有资源枯竭，其他用户不再受影响。电路断路器还可以允许聪明的开发者来标记使用不可用功能的网站部分，或也许在断路器处于打开状态时，显示出一些适当的缓存内容。

Akka库提供名为 `akka.pattern.CircuitBreaker` 的断路器实现，具有如下所述的行为。

他们是做什么工作的？

- 在正常操作期间，断路器处于`Closed`状态：
  - 异常或超过配置的 `callTimeout` 的调用增加一个失败计数
  - 成功重置失败计数为零
  - 当失败计数达到 `maxFailures` 时，断路器跳入`Open`状态
- 在`Open`状态：
  - 所有调用通过 `CircuitBreakerOpenException` 快速失败
  - 经过配置的 `resetTimeout`，断路器进入`Half-Open`状态
- 在`Half-Open`状态：
  - 第一个调用被允许尝试，而不通过快速失败
  - 其他调用就像在`Open`状态一样快速失败
  - 如果第一次调用成功，断路器是重置回`Closed`状态
  - 如果第一次调用失败，断路器再次跳入`Open`状态并经历另一个完整的`resetTimeout`
- 状态转换监听器：
  - 可以为每个状态条目通过 `onOpen`、 `onClose` 和

`onHalfOpen` 提供回调
- 这些都在提供的 `ExecutionContext` 中执行。



例子

初始化

下面是如何配置一个 `CircuitBreaker`：

- 最多失败5次
- 调用超时时间为 `10` 秒
- 重置超时时间为 `1` 分钟

Scala

```
1.  import scala.concurrent.duration._
2.  import akka.pattern.CircuitBreaker
3.  import akka.pattern.pipe
4.  import akka.actor.Actor
5.  import akka.actor.ActorLogging
6.  import scala.concurrent.Future
7.  import akka.event.Logging
8.
9.  class DangerousActor extends Actor with ActorLogging {
10.   import context.dispatcher
11.
12.   val breaker =
13.     new CircuitBreaker(context.system.scheduler,
```

```
14.       maxFailures = 5,
15.       callTimeout = 10.seconds,
16.       resetTimeout = 1.minute).onOpen(notifyMeOnOpen())
17.
18.   def notifyMeOnOpen(): Unit =
19.     log.warning("My CircuitBreaker is now open, and will not close
    for one minute")
```

Java

```
1.  import akka.actor.UntypedActor;
2.  import scala.concurrent.Future;
3.  import akka.event.LoggingAdapter;
4.  import scala.concurrent.duration.Duration;
5.  import akka.pattern.CircuitBreaker;
6.  import akka.event.Logging;
7.
8.  import static akka.pattern.Patterns.pipe;
9.  import static akka.dispatch.Futures.future;
10.
11. import java.util.concurrent.Callable;
12.
13. public class DangerousJavaActor extends UntypedActor {
14.
15.   private final CircuitBreaker breaker;
16.   private final LoggingAdapter log =
    Logging.getLogger(getContext().system(), this);
17.
18.   public DangerousJavaActor() {
19.     this.breaker = new CircuitBreaker(
20.       getContext().dispatcher(), getContext().system().scheduler(),
21.       5, Duration.create(10, "s"), Duration.create(1, "m"))
22.       .onOpen(new Runnable() {
23.         public void run() {
24.           notifyMeOnOpen();
25.         }
26.       });
27.   }
```

```
28.
29.   public void notifyMeOnOpen() {
30.     log.warning("My CircuitBreaker is now open, and will not close
    for one minute");
31.   }
```

调用保护

下面是如何将 `CircuitBreaker` 用于保护一个异步调用，以及一个同步调用：

Scala

```
1.  def dangerousCall: String = "This really isn't that dangerous of a
    call after all"
2.
3.  def receive = {
4.    case "is my middle name" =>
5.      breaker.withCircuitBreaker(Future(dangerousCall)) pipeTo
    sender()
6.    case "block for me" =>
7.      sender() ! breaker.withSyncCircuitBreaker(dangerousCall)
8.  }
```

Java

```
1.  public String dangerousCall() {
2.    return "This really isn't that dangerous of a call after all";
3.  }
4.
5.  @Override
6.  public void onReceive(Object message) {
7.    if (message instanceof String) {
8.      String m = (String) message;
9.      if ("is my middle name".equals(m)) {
10.       pipe(breaker.callWithCircuitBreaker(
11.         new Callable<Future<String>>() {
12.           public Future<String> call() throws Exception {
```

```
13.             return future(
14.                     new Callable<String>() {
15.                         public String call() {
16.                           return dangerousCall();
17.                         }
18.                       }, getContext().dispatcher());
19.           }
20.         }), getContext().dispatcher()).to(getSender());
21.     }
22.     if ("block for me".equals(m)) {
23.       getSender().tell(breaker
24.         .callWithSyncCircuitBreaker(
25.           new Callable<String>() {
26.             @Override
27.             public String call() throws Exception {
28.               return dangerousCall();
29.             }
30.           }), getSelf());
31.     }
32.   }
33. }
```

> 注意
>
> 使用 `CircuitBreaker` 伴生对象的 `apply` 或 `create` 方法将返回在调用者的线程中执行回调的 `CircuitBreaker` 。如果异步 `Future` 不必要的时候，这可以是很有用的，例如仅调用同步的*API*。

Akka扩展

Akka扩展

> 注：本节未经校验，如有问题欢迎提*issue*

如果想要为Akka添加特性，有一个非常优美而且强大的工具，称为 Akka 扩展。它由两部分组成： `Extension` 和 `ExtensionId` .

Extensions 在每个 `ActorSystem` 中只会加载一次，并被Akka所管理。 你可以选择按需加载你的Extension或是在 `ActorSystem` 创建时通过Akka配置来加载。 关于这些细节，见下文 “从配置中加载” 的部分.

> 警告
>
> 由于扩展是*hook*到*Akka*自身的，所以扩展的实现者需要保证自己扩展的线程安全性。

构建一个扩展

现在我们来创建一个扩展示例，它的功能是对某件事发生的次数进行统计。

首先定义 `Extension` 的功能：

```
1.  import akka.actor.Extension
2.
3.  class CountExtensionImpl extends Extension {
```

```
  //Since this Extension is a shared instance
  // per ActorSystem we need to be threadsafe
  private val counter = new AtomicLong(0)

  //This is the operation this Extension provides
  def increment() = counter.incrementAndGet()
}
```

然后需要为扩展指定一个 `ExtensionId`，这样我们可以获取它的实例.

```
import akka.actor.ActorSystem
import akka.actor.ExtensionId
import akka.actor.ExtensionIdProvider
import akka.actor.ExtendedActorSystem

object CountExtension
  extends ExtensionId[CountExtensionImpl]
  with ExtensionIdProvider {
  //The lookup method is required by ExtensionIdProvider,
  // so we return ourselves here, this allows us
  // to configure our extension to be loaded when
  // the ActorSystem starts up
  override def lookup = CountExtension

  //This method will be called by Akka
  // to instantiate our Extension
  override def createExtension(system: ExtendedActorSystem) = new
CountExtensionImpl

  /**
   * Java API: retrieve the Count extension for the given system.
   */
  override def get(system: ActorSystem): CountExtensionImpl =
super.get(system)
}
```

好了！然后我们就可以使用它了：

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

或者在Akka Actor中使用:

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

你也可以将扩展藏在 trait 里:

```
trait Counting { self: Actor =>
  def increment() = CountExtension(context.system).increment()
}
class MyCounterActor extends Actor with Counting {
  def receive = {
    case someMessage => increment()
  }
}
```

这样就搞定了!

从配置中加载

为了能够从Akka配置中加载扩展，你必须在为 `ActorSystem` 提供的配置文件中的 `akka.extensions` 部分加上 `ExtensionId` 或 `ExtensionIdProvider` 实现类的完整路径。

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

实用性

充分发挥你的想象力，天空才是极限！ 顺便提一下，你知道 Akka 的 `Typed Actor` ， `Serialization` 和其它一些特性都是以Akka扩展的形式实现的吗？

应用特定设置

可以用 `Configuration` 来指定应用特有的设置。将这些设置放在一个扩展里是一个好习惯。

配置示例：

```
myapp {
  db {
    uri = "mongodb://example1.com:27017,example2.com:27017"
  }
  circuit-breaker {
    timeout = 30 seconds
  }
}
```

`Extension` 的实现：

```
import akka.actor.ActorSystem
import akka.actor.Extension
import akka.actor.ExtensionId
import akka.actor.ExtensionIdProvider
import akka.actor.ExtendedActorSystem
import scala.concurrent.duration.Duration
import com.typesafe.config.Config
import java.util.concurrent.TimeUnit

class SettingsImpl(config: Config) extends Extension {
  val DbUri: String = config.getString("myapp.db.uri")
  val CircuitBreakerTimeout: Duration =
```

```
13.     Duration(config.getMilliseconds("myapp.circuit-
    breaker.timeout"),
14.       TimeUnit.MILLISECONDS)
15. }
16. object Settings extends ExtensionId[SettingsImpl] with
    ExtensionIdProvider {
17.
18.   override def lookup = Settings
19.
20.   override def createExtension(system: ExtendedActorSystem) =
21.     new SettingsImpl(system.settings.config)
22.
23.   /**
24.    * Java API: retrieve the Settings extension for the given
    system.
25.    */
26.   override def get(system: ActorSystem): SettingsImpl =
    super.get(system)
27. }
```

使用它：

```
1. class MyActor extends Actor {
2.   val settings = Settings(context.system)
3.   val connection = connect(settings.DbUri,
    settings.CircuitBreakerTimeout)
```

微内核

微内核

> 注：本节未经校验，如有问题欢迎提*issue*

Akka微内核的目的是提供一个捆绑机制，以便将 Akka 应用程序作为一个单一有效载荷分发，而不需要在 Java 应用程序服务器中运行，或手动创建一个启动脚本。

Akka 微内核包含在Akka下载中.

要通过微内核运行应用，你需要创建一个 Bootable 类来处理应用的启动和关闭。下面例子中有介绍。

将你的应用jar包放在 `deploy` 目录下，并把依赖放入 `lib` 目录下以便自动装载并放在类路径中。

要启动内核使用 `bin` 目录下的脚本，将应用启动类传进来.

以下是akka下载包中的一个应用与微内核一起运行的例子. 它可以用以下的命令运行（在unix系统上）:
启动脚本添加 `config` 目录作为类路径第一个元素，其次是 `lib/*` 。它以 `akka.kernel.Main` 为主类运行 java 并提供引导类作为参数。

（基于 unix 的系统）示例命令:

```
1.  bin/akka sample.kernel.hello.HelloKernel
```

使用 `Ctrl-C` 来中断并退出微内核.

在Windows机器上你可以使用 `bin/akka.bat` 脚本.

以下是Hello Kernel 示例 (参考 `HelloKernel` 创建一个 Bootable 类):

```
1.  package sample.kernel.hello
2.
3.  import akka.actor.{ Actor, ActorSystem, Props }
4.  import akka.kernel.Bootable
5.
6.  case object Start
7.
8.  class HelloActor extends Actor {
9.    val worldActor = context.actorOf(Props[WorldActor])
10.
11.   def receive = {
12.     case Start => worldActor ! "Hello"
13.     case message: String =>
14.       println("Received message '%s'" format message)
15.   }
16. }
17.
18. class WorldActor extends Actor {
19.   def receive = {
20.     case message: String => sender() ! (message.toUpperCase + "
    world!")
21.   }
22. }
23.
24. class HelloKernel extends Bootable {
25.   val system = ActorSystem("hellokernel")
26.
27.   def startup = {
28.     system.actorOf(Props[HelloActor]) ! Start
29.   }
30.
31.   def shutdown = {
32.     system.shutdown()
```

```
33.     }
34. }
```

如何使用：常用模式

如何使用：常用模式

> 注：本节未经校验，如有问题欢迎提*issue*

本节列出了一些常见的actor模式，它们已被发现是有用的、优雅的或有启发意义的。所有主题都是受欢迎的，列出的示例主题包括消息路由策略，监督模式，重启处理等。作为一个特殊的奖励，这一节补充标记了贡献者的名字，如果每个Akka使用者在他或她的代码中发现重复出现的模式，并为所有人分享它的好处是多么美好的事情啊。在适和的情

况下也可能加入 `akka.pattern` 包来创建一个类OTP库（OTP-like library）。

限制消息" class="reference-link">限制消息

贡献者: Kaspar Fischer

“一个消息节流器以确保消息不会以太高的速率发送。”

该模式详见在Akka2中做消息限流。

跨节点平衡负载" class="reference-link">跨节点平衡负载

贡献者: Derek Wyatt

“很多时候，人们需要BalancingDispatcher的功能中包含在不同节点上拥有独立邮箱的Actor工作的规则。在这篇文章我们将探索实现这样一个概念。”

该模式详见Akka2跨节点负载均衡。

工作拉取模式来限流和分发工作，并防止邮箱溢出" class="reference-link">工作拉取模式来限流和分发工作，并防止邮箱溢出

贡献者: Michael Pollmeier

“如果创建工作的速度实际上比执行它块，这种模式可以确保你的邮箱不会溢出 —— 当邮箱最终变得太满时这会导致内存溢出错误。它也让你围绕你的群集分配工作，动态地扩展规模，并且是完全无阻塞的。这是‘负载均衡模式’的一个特例。

该模式详见工作拉取模式来限流和分发工作，并防止邮箱溢出。

有序终止" class="reference-link">有序终止

贡献者: Derek Wyatt

“当一个actor停止时，它的子actor以未知顺序停止。子actor终止是异步的，因而是不确定的。

如果actor的孩子有顺序依赖关系，则你可能需要确保这些子actor以特定顺序关闭，从而使其postStop() 方法按正确顺序调用。”

该模式详见一种Akka2终止器。

Akka AMQP 代理" class="reference-link">Akka AMQP 代理

贡献者: Fabrice Drouin

““AMQP 代理”是将AMQP与Akka结合进行跨计算节点网络工作分发的简单方法。你将仍然编写“本地”代码，进行很少的配置，并将最终拥有一个分布式的、弹性的、容错的网格，其计算节点可以几乎以任何语言编写。”

该模式详见Akka AMQP 代理。

Akka2 关闭模式" class="reference-link">Akka2 关闭模式

贡献者: Derek Wyatt

“当一切都结束时你如何告诉Akka关闭ActorSystem？原来竟然没有这样一个神奇的标志，没有配置设置，没有可以注册特殊回调的地方，

也没有杰出的关机童话仙子用她的荣光在那完美的一刻恩典你的应用程序。她就是很普通刻薄。

在这篇文章中，我们将讨论为什么是这种情况，并为你提供一个简单的选项”在正确的时间”关闭，以及一个并不是-那么-简单-的选项来达到同样的目的。”

该模式详见Akka2 关闭模式。

Akka分布式（内存中）图处理" class="reference-link">Akka分布式（内存中）图处理

贡献者: Adelbert Chang

“图在数学和计算机科学（以及其他领域）中一直是一个有趣的研究结构，而且在社交网络如Facebook和Twitter中变得更加有趣，其底层网络结构可以很好地由图来描述”。

该模式详见Akka分布式（内存中）图处理。

案例研究: 使用actor自动更新缓存" class="reference-link">案例研究: 使用actor自动更新缓存

贡献者: Eric Pederson

“我们最近需要在一个缓慢的后端系统前构建一个高速缓存系统，并符合下列要求:

后端系统中的数据是不断更新的，所以需要每隔N分钟更新缓存。对后端系统的直接请求需要被限流。我们建立的缓存系统使用了Akka

actor和 Scala 中函数作为头等对象的支持。”

该模式详见案例研究：使用actor自动更新缓存。

使用蜘蛛模式在actor系统中发现消息流" class="reference-link">使用蜘蛛模式在actor系统中发现消息流

贡献者：Raymond Roestenburg

“构建actor系统是有趣的，可是调试它们可能很困难，你大多情况下需要在多个机器中浏览大量日志文件，来了解到底发生了什么。我敢肯定你在啃日志时会想，“嘿，这个消息跑哪了？”，“为什么这个消息引起这种效果”或“为什么这个actor永远得不到消息？”

这是蜘蛛模式的进来。”

该模式详见使用蜘蛛模式在actor系统中发现消息流。

调度周期性消息" class="reference-link">调度周期性消息

此模式描述了如何安排周期性消息给自己，有两种不同方式。

第一种方法是在actor构造函数中设置定期消息调度，并在 `postStop` 中取消定时发送，否则我们可能会有多个已注册的消息发送到相同的actor。

> 注意
>
> 用这种方法被调度的定期消息发送将在*actor*重启时被重新启动。这也意味着在重新启动期间，两个*tick*消息之间的时间间隔可能会漂移，它基于你重新启动调度预定的消息时的时间到最后一个消息的发送时间，以及初始延迟是多长时间。最糟糕的情况是 `interval` 加上 `initialDelay` 。

```
class ScheduleInConstructor extends Actor {
  import context.dispatcher
  val tick =
    context.system.scheduler.schedule(500 millis, 1000 millis,
self, "tick")

  override def postStop() = tick.cancel()

  def receive = {
    case "tick" =>
      // do something useful here
  }
}
```

第二种变体在actor的 `preStart` 方法中建立了一个初始的消息发送，然后当actor接收到此消息时设置一个新的消息发送。你还必须重写 `postRestart` ，所以我们不会调用 `preStart` 并重新调度一个初始消息的发送。

> 注意
>
> 用这种方法，即使*actor*负载很高，我们也不会被*tick*消息填满邮箱，而只会在收到前一个*tick*消息之后安排新*tick*消息发送。

```
class ScheduleInReceive extends Actor {
  import context._

  override def preStart() =
    system.scheduler.scheduleOnce(500 millis, self, "tick")

  // override postRestart so we don't call preStart and schedule a
new message
  override def postRestart(reason: Throwable) = {}

  def receive = {
    case "tick" =>
      // send another periodic tick after the specified delay
```

```
13.         system.scheduler.scheduleOnce(1000 millis, self, "tick")
14.         // do something useful here
15.     }
16. }
```

模板模式" class="reference-link">模板模式

贡献者：*N. N.*

这是一种特别好的模式，因为它甚至伴随着一些空的示例代码：

```
1.  class ScalaTemplate {
2.    println("Hello, Template!")
3.    // uninteresting stuff ...
4.  }
```

> 注意
>
> 流传一句话：这是成名的最简单方法！

请在该文件的结尾保留这个模式。

实验模块

实验模块

> 注：本节未经校验，如有问题欢迎提*issue*

以下几个Akka模块被标记作为实验性的，这意味着它们处于早期访问模式，这也意味着他们没有包含在商业支持中。将它们作为实验模块提早发布的目的是使其更容易获得和根据反馈改进，或甚至发现该模块并不是很有用。

实验模块并非都要服从微版本间二进制兼容的规定。在根据用户反馈完善和简化时，破坏 API 的更改可能在不注意的情况下在次要版本中引入。实验模块可能在没有标记为废弃的情况下在次要版本被弃用。

- 持久性
- 多节点测试
- Actor（Java Lambda 的支持）[不在此scala文档中介绍]
- FSM（Java Lambda 的支持）[不在此scala文档中介绍]

标记模块为实验性的另一个原因是模块没有足够时间来证明有一个可以长时间肩负维护责任的人。这些模块位于 `akka-contrib` 子项目中：

- 外部贡献TODO

持久化

持久化

Akka持久化使有状态的actor能留存其内部状态，以便在因JVM崩溃、监管者引起，或在集群中迁移导致的actor启动、重启时恢复它。Akka持久化背后的关键概念是持久化的只是一个actor的内部状态的的变化，而不是直接持久化其当前状态 （除了可选的快照）。这些更改永远只能被附加到存储，没什么是可变的，这使得高事务处理率和高效复制成为可能。有状态actor通过重放保存的变化来恢复，从而使它们可以重建其内部状态。重放的可以是完整历史记录，或着从某一个快照开始从而可以大大减少恢复时间。Akka持久化也提供了“至少一次消息传递语义”的点对点通信。

> 注意
>
> 本模块被标记为*“experimental”*直到*Akka 2.3.0*引入它。我们将基于用户的反馈继续改善此*API*，这就意味着我们对维护版本的二进制不兼容性降到最低的保证不适用于 `akka.persistence` 包的内容。

Akka持久化受eventsourced启发，并且是其正式的替代者。它遵循eventsourced相同的概念和体系结构，但在API和实现层上则显著不同。又见《迁移指南：从Eventsourced到Akka Persistence 2.3》。

Akka 2.3.4的变化

在Akka 2.3.4中，较早版本中的几个概念被推倒和简化。大体上讲；`Processor`和`EventsourcedProcessor`被替换为`PersistentActor`。`Channel`和`PersistentChannel`被替换为`AtLeastOnceDelivery`。`View`被替换为`PersistentView`。

更改的全部细节请参阅《迁移指南：从Akka Persistence (experimental) 2.3.3到Akka Persistence 2.3.4 (和2.4.x)》。老的类在一段时间内仍被包含并标记为废弃，以便用户顺利过渡。如果你需要的旧的文档，可以参考这里。

依赖

Akka持久化是一个单独的jar文件。请确保你的项目中有以下依赖关系：

```
1.  "com.typesafe.akka" %% "akka-persistence-experimental" % "2.3.6"
```

体系结构

- *PersistentActor*：是一个持久的、有状态的actor。它能够持久化消息到一个日志，并以线程安全的方式对它们作出响应。它是可被用于执行命令*[command]*和事件来源*[event sourced]*的actor。当一个持久化的actor被启动或重新启动时，该actor会被重播日志消息，从而可以从这些消息恢复内部状态。
- *PersistentView*：一个视图是一个持久的、有状态的actor，来接收已经由另一个持久化actor写下的日志消息。视图本身并没有新的日志消息，相反，它只能从一个持久化actor复制消息流来更新内部状态。
- *AtLeastOnceDelivery*：使用至少一次的传递语义将消息发送

到目的地，以防发送者和接收者 JVM崩溃。

- *Journal*：日志存储发送到一个持久化actor的消息序列。应用程序可以控制actor接收的消息中，哪些需要在日记中记录，哪些不需要。日志的存储后端是可插拔的。默认日志存储插件是写入本地文件系统，复制日志在社区插件中可以获得。
- *Snapshot store*：快照存储区持久化一个持久化actor或一个视图的内部状态的快照。快照可用于优化恢复时间。快照存储区的存储后端是可插拔的。默认快照存储插件写入本地文件系统。

事件来源 Event sourcing" class="reference-link">事件来源 Event sourcing

事件来源背后的基本思想很简单。一个持久化actor接收一个（非持久化）命令，它首先会被验证是否可以被应用到当前状态。在这里，验证可以意味着任何东西，例如从对命令消息字段的简单检查，到引用若干外部服务。如果验证成功，从该命令生成事件，表示命令的效果。然后这些事件被持久化，在成功的持久化后，用于改变actor的状态。当持久化actor需要恢复时，仅重播持久化的事件，因为我们知道他们可以被成功地应用。换句话说，与命令不同，被重播到一个持久化actor的事件不能失败。事件来源的actor当然也可以处理不改变应用程序状态的命令，例如查询命令。

Akka持久化通过 `PersistentActor` 特质支持事件来源。一个actor可以扩展这个特质来使用 `persist` 方法持久化和处理事件。 `PersistentActor` 的行为是通过实现 `receiveRecover` 和 `receiveCommand` 定义的。下面的示例演示了这一点。

```
1.  import akka.actor._
```

```
2.  import akka.persistence._
3.
4.  case class Cmd(data: String)
5.  case class Evt(data: String)
6.
7.  case class ExampleState(events: List[String] = Nil) {
8.    def updated(evt: Evt): ExampleState = copy(evt.data :: events)
9.    def size: Int = events.length
10.   override def toString: String = events.reverse.toString
11. }
12.
13. class ExamplePersistentActor extends PersistentActor {
14.   override def persistenceId = "sample-id-1"
15.
16.   var state = ExampleState()
17.
18.   def updateState(event: Evt): Unit =
19.     state = state.updated(event)
20.
21.   def numEvents =
22.     state.size
23.
24.   val receiveRecover: Receive = {
25.     case evt: Evt                                 =>
    updateState(evt)
26.     case SnapshotOffer(_, snapshot: ExampleState) => state =
    snapshot
27.   }
28.
29.   val receiveCommand: Receive = {
30.     case Cmd(data) =>
31.       persist(Evt(s"${data}-${numEvents}"))(updateState)
32.       persist(Evt(s"${data}-${numEvents + 1}")) { event =>
33.         updateState(event)
34.         context.system.eventStream.publish(event)
35.       }
36.     case "snap"  => saveSnapshot(state)
37.     case "print" => println(state)
```

```
38.   }
39.
40. }
```

该示例定义了两种数据类型，`Cmd` 和 `Evt` 分别代表命令和事件。`ExamplePersistentActor` 的 `state` 包含在 `ExampleState` 中的持久化的事件数据的列表。

持久化actor的 `receiveRecover` 方法定义如何通过在恢复过程中处理 `Evt` 和 `SnapshotOffer` 消息来更新 `state`。持久化actor的 `receiveCommand` 方法是一个命令处理程序。在此示例中，命令处理是通过生成两个事件，然后被持久化和处理的。事件通过调用 `persist` 方法持久化，该方法第一个参数是事件（或一系列事件），第二个参数是事件处理程序。

`persist` 方法以异步方式持久化事件，而事件处理程序对成功持久化的事件进行处理。成功持久化的事件在内部作为独立消息发送回给持久化actor，来触发事件处理执行。事件处理程序可能会包含持久化actor的状态并修改它。持久化事件的发送者也是相应命令的发送者。这使事件处理程序可以回复命令的发送者（未显示）。

事件处理程序的主要任务是：使用事件数据更改持久化actor状态，并通过发布事件通知其他人成功的状态变化。

当使用 `persist` 持久化事件的时候，可以保证持久化actor在 `persist` 调用和相应的事件处理程序的（多次）执行之间不会进一步收到命令。这在单个命令的上下文中多次调用 `persist` 的情况下也成立。

运行该示例最简单的方法是下载Typesafe Activator，并打开Akka Persistence Samples with Scala这个教程。它包含如何运行 `PersistentActorExample` 的说明。

> 注意
>
> 还有可能在正常处理过程中使用不同的命令处理程序，并使用 `context.become()` 和 `context.unbecome()` 来恢复。恢复后使*actor*进入相同的状态，你需要特别谨慎地使用 `receiveRecover` 方法中的 `become` 和 `unbecome` 进行相同的状态转换，就像你会在命令处理程序中做的一样。

标识符

一个持久化actor必须具有跨不同actor化身而不改变的标识符。必须使用 `persistenceId` 方法定义该标识符。

```
1.  override def persistenceId = "my-stable-persistence-id"
```

恢复" class="reference-link">恢复

默认情况下，一个持久化actor通过在启动和重启时重放日志消息实现自动恢复。恢复过程中发送给持久化actor的新消息不会干扰重放消息。新消息只会在持久化actor恢复完成后被收到。

自定义恢复

通过使用空实现重写 `preStart` ，可以禁用启动时的自动恢复。

```
1.  override def preStart() = ()
```

在这种情况下，必须显式地通过 `Recover()` 消息的发送恢复一个持久化actor。

```
1.  processor ! Recover()
```

如果没有重写， `preStart` 将发送一个 `Recover()` 消息到 `self` 。应用程序还可能重写 `preStart` 来定义进一步的 `Recover()` 参数如序列号范围上界，例如。

```
1.  override def preStart() {
```

```
2.   self ! Recover(toSequenceNr = 457L)
3. }
```

序列号范围上界可以用来恢复持久化actor到过去的某个状态，而不是当前状态。通过使用空实现重写 `preRestart` ，可以禁用重新启动时的自动恢复。

```
1. override def preRestart(reason: Throwable, message: Option[Any]) =
   ()
```

恢复状态

一个持久化actor可以通过以下方法查询自身的恢复状态

```
1. def recoveryRunning: Boolean
2. def recoveryFinished: Boolean
```

有时持久化actor在恢复完成时，处理任意其他消息之前，需要执行额外的初始化。持久化actor会在恢复完成后，处理任意其他消息之前，收到一个特别的 `RecoveryCompleted` 消息。

如果该actor在从日志中恢复状态出现问题，该actor将发送 `RecoveryFailure` 消息并可以选择在 `receiveRecover` 中处理。如果该actor不处理 `RecoveryFailure` 消息，它将被停止。

```
1. def receiveRecover: Receive = {
2.   case RecoveryCompleted => recoveryCompleted()
3.   case evt               => //...
4. }
5.
6. def receiveCommand: Receive = {
7.   case msg => //...
8. }
9.
10. def recoveryCompleted(): Unit = {
```

```
11.   // perform init after recovery, before any other messages
12.   // ...
13. }
```

放宽的局部一致性要求和高吞吐量的用例

如果面临放宽的局部一致性要求和高吞吐量，有时 `PersistentActor` 及其 `persist` 在处理大量涌入的命令时可能会不够，因为它必须等待知道给定命令相关的所有事件都处理完成后，才开始处理下一条命令。虽然这种抽象在大多数的情况下非常有用，有时你可能会放宽一致性要求——例如你会想要尽可能快速地处理命令，假设事件最终会持久化并在后台恰当处理，并在需要时追溯性地回应持久性故障。

`persistAsync` 方法提供了一个工具，用于实现高吞吐量的持久化 actor。在日志仍在致力于持久化和（或） 执行用户事件回调代码时，它不会贮藏传入的命令。

在下面的示例中，事件回调可能在"任何时候"被调用，甚至在处理下一条命令之后。两个事件之间的顺序仍能得到保证（"evt-b-1"将在"evt-a-2"后发送，而它又在"evt-a-1"后发送，以此类推）。

```
1. class MyPersistentActor extends PersistentActor {
2.
3.   override def persistenceId = "my-stable-persistence-id"
4.
5.   def receiveRecover: Receive = {
6.     case _ => // handle recovery here
7.   }
8.
9.   def receiveCommand: Receive = {
10.    case c: String => {
11.      sender() ! c
12.      persistAsync(s"evt-$c-1") { e => sender() ! e }
13.      persistAsync(s"evt-$c-2") { e => sender() ! e }
14.    }
```

```
15.   }
16. }
17.
18. // usage
19. processor ! "a"
20. processor ! "b"
21.
22. // possible order of received messages:
23. // a
24. // b
25. // evt-a-1
26. // evt-a-2
27. // evt-b-1
28. // evt-b-2
```

> 注意
>
> 为了实现"命令源"模式，只需对所有传入消息马上调用 *persistAsync(cmd)(...)* ，并在回调中处理它们。
>
> 警告
>
> 如果在调用 *persistAsync* 和日志确定写操作之间，*actor*被重启（或停止）时，将不会调用回调。

推迟行动，直到持久化处理程序已执行

使用 persistAsync 时，有时你会发现定义一些''在 persistAsync 处理程序调用之后发生''的行动是很好的。 PersistentActor 提供了一个工具方法 defer ，它类似于 persistAsync ，可是并不持久化过去的事件。推荐它用于读取的操作，和在你的域模型中没有相应事件的行动。

使用这种方法和持久化系列方法的使用是非常相似的，但它不会持久化过去的事件。它将保留在内存中，并在调用处理程序时使用。

```
1. class MyPersistentActor extends PersistentActor {
2.
3.   override def persistenceId = "my-stable-persistence-id"
4.
```

```
  def receiveRecover: Receive = {
    case _ => // handle recovery here
  }

  def receiveCommand: Receive = {
    case c: String => {
      sender() ! c
      persistAsync(s"evt-$c-1") { e => sender() ! e }
      persistAsync(s"evt-$c-2") { e => sender() ! e }
      defer(s"evt-$c-3") { e => sender() ! e }
    }
  }
}
```

注意 `sender()` 是可以在处理程序回调中安全访问的，并将指向 `defer` 处理程序被调用的命令的原始发送者。

调用方将以这样的顺序（保证）获得响应：

```
processor ! "a"
processor ! "b"

// order of received messages:
// a
// b
// evt-a-1
// evt-a-2
// evt-a-3
// evt-b-1
// evt-b-2
// evt-b-3
```

> 警告
>
> 如果该*actor*在调用 *defer* 和日志处理与确认所有写入之间的回调，将不会在*actor*重启（或停止）时调用。

批处理写操作

为了优化吞吐量，一个持久化actor在高负荷下，会内部将一批事件先储存，然后再（作为一个批处理）写到日志中。批处理大小可以调整，从低和中等载荷作用下的1，动态增长到高负荷下可配置的最大大小（默认为 `200` ）。在使用 `persistAsync` 时，这极大地增加了最大吞吐量。

```
1.  akka.persistence.journal.max-message-batch-size = 200
```

只要一个batch达到最大大小或日志完成前一批写操作，就会立即触发持久化actor新的批处理写操作。批处理写操作永远不会是基于计时器的，从而将延迟保持在最低限度。

批处理也在内部使用确保事件写操作的原子性。在单个命令上下文中的所有事件将作为单个批处理写入到日志中（即使在一个命令中多次调用 `persist` ）。因此， `PersistentActor` 的恢复将永远不会部分完成（只持久化单个命令中事件的一个子集）。

删除邮件

若要删除所有消息（由一个持久化actor记录）到指定的序列号，持久化actor可以调用 `deleteMessages` 方法。

一个可选的 `permanent` 参数指定是否应从日志中永久删除消息，或仅标记为已删除。在这两种情况下，消息都不会重播。Akka持久化以后的扩展将允许重播标记为已删除的消息，例如可用于调试。

持久化视图

持久化视图可以通过扩展 `PersistentView` 特质以及实现 `receive` 和 `persistenceId` 方法实现。

```
1.  class MyView extends PersistentView {
```

```
  override def persistenceId: String = "some-persistence-id"
  override def viewId: String = "some-persistence-id-view"

  def receive: Actor.Receive = {
    case payload if isPersistent =>
    // handle message from journal...
    case payload                 =>
    // handle message from user-land...
  }
}
```

`PersistenceId` 标识从视图中接收的日志消息来自的持久化actor。该引用持久化actor实际并非必须正在运行。视图直接从一个持久化actor日志中读取消息。当一个持久化actor后来启动，并开始写新消息时，将默认自动更新相应的视图。

可以确定一条消息是从日志中发送，还是由用户定义的另一个调用 `isPersistent` 方法的actor发送。尽管有这样的功能，很多时候你根本不需要此信息，并可以简单地将相同的逻辑应用于这两种情况（跳过 `if isPersistent` 检查）。

更新

actor系统的所有视图的默认更新间隔是可配置的：

```
akka.persistence.view.auto-update-interval = 5s
```

`PersistentView` 实现类还可以重写 `autoUpdateInterval` 方法，以返回对特定的视图类或视图实例自定义的更新时间间隔。应用程序也可以在任何时候通过对一个视图发送 `Update` 消息触发额外的数据更新。

```
val view = system.actorOf(Props[MyView])
view ! Update(await = true)
```

如果 `await` 参数设置为 `true` ，在 `Update` 请求后面的消息在增量消息重播时会被处理，在这个更新请求处理完成时触发。如果设置为 `false` （默认值），更新请求后的消息可能与重播的消息流交织。自动更新始终以 `await = false` 运行。

actor系统中所有视图的自动更新可以在配置中关闭：

```
1.  akka.persistence.view.auto-update = off
```

实现类可以通过重载 `autoUpdate` 方法重写配置的默认值。若要限制的每个更新请求的重播消息数量，应用程序可以配置自定义的 `akka.persistence.view.auto-update-replay-max` 值或重载 `autoUpdateReplayMax` 方法。手动更新的重播消息数目可以通过 `Update` 消息的 `replayMax` 参数进行限制。

恢复

持久化视图的初始化恢复过程和持久化actor的工作方式相同（即通过发送一个 `Recover` 消息到自己）。初始化恢复的最大重放消息数由 `autoUpdateReplayMax` 确定。关于自定义初始化恢复更多的可能性参见恢复一节。

标识符

一个持久化视图必须具有跨不同actor化身而不改变的标识符。必须使用 `viewId` 方法定义该标识符。

`ViewId` 必须不同于引用的 `persistenceId` ，除非快照视图和其持久化actor是共享的（即应用程序通常不需要做的东西）。

快照" class="reference-link">快照

快照可以大幅减少持久化actor和视图的恢复时间。下面讨论的快照内

容是基于持久化actor的上下文，但这也同样适用于持久化视图。

持久化actor可以通过调用 `saveSnapshot` 方法保存内部状态的快照。如果快照保存成功，持久化actor接收 `SaveSnapshotSuccess` 消息，否则 `SaveSnapshotFailure` 消息

```
1. class MyProcessor extends Processor {
2.   var state: Any = _
3.
4.   def receive = {
5.     case "snap"                                 =>
   saveSnapshot(state)
6.     case SaveSnapshotSuccess(metadata)          => // ...
7.     case SaveSnapshotFailure(metadata, reason) => // ...
8.   }
9. }
```

这里 `metadata` 的类型是 `SnapshotMetadata`：

```
1. case class SnapshotMetadata(@deprecatedName('processorId)
   persistenceId: String, sequenceNr: Long, timestamp: Long = 0L) {
2.   @deprecated("Use persistenceId instead.", since = "2.3.4")
3.   def processorId: String = persistenceId
4. }
```

在恢复期间，持久化actor可以通过 `SnapshotOffer` 消息获取以前保存的快照，从中可以初始化内部状态。

```
1. class MyProcessor extends Processor {
2.   var state: Any = _
3.
4.   def receive = {
5.     case SnapshotOffer(metadata, offeredSnapshot) => state =
   offeredSnapshot
6.     case Persistent(payload, sequenceNr)          => // ...
7.   }
```

```
8. }
```

紧随着 `SnapshotOffer` 的重播消息，如果有的话，是比快照年轻的。他们帮助持久化actor恢复到其当前（即最新的）状态。

一般情况下，如果持久化actor之前保存了多份快照，且这些快照中至少有一个满足 `SnapshotSelectionCriteria` 并可被指定用于恢复的情况下，才会给持久化actor提供一个快照。

```
1. processor ! Recover(fromSnapshot = SnapshotSelectionCriteria(
2.   maxSequenceNr = 457L,
3.   maxTimestamp = System.currentTimeMillis))
```

如果未指定，他们默认为 `SnapshotSelectionCriteria.Latest` ，即选择最新（= 最小）的快照。若要禁用基于快照的恢复，应用程序应使用 `SnapshotSelectionCriteria.None` 。如果已保存的快照没有匹配指定的 `SnapshotSelectionCriteria` ，恢复时将重播所有日志消息。

快照删除

一个持久化actor可以通过调用 `deleteSnapshot` 方法并指定快照的序列号与的时间戳作为参数，来删除单个快照。要批量删除匹配 `SnapshotSelectionCriteria` 的快照，持久化actor应该使用 `deleteSnapshots` 方法。

至少一次投递

要在至少一次投递语义下发送消息到目的地，你可以在发送端的 `PersistentActor` 混入 `AtLeastOnceDelivery` 特质。如果他们在可配置的超时时间内未得到确认，它负责重新发送消息。

> 注意
>
> 至少一次投递意味着原始消息发送顺序并不总是保留的，以及目的地可能接收重复的消息。这意

> 味着语义不匹配那些正常的 `ActorRef` 发送操作：
>
> - 它不是在最多一次投递
> - 同一个发件人-接收人对的消息顺序不保留，因为可重新发送
> - 崩溃并重新启动后，消息仍然会发送到目的地—向新*actor*化身
>
> 这些语义和 `ActorPath` 所表示的相似（见*actor*生命周期），因此你在发送消息时需要提供的是一个路径而不是一个引用。消息被发送到一个指向*actor selection*的路径。

`deliver` 方法用于将消息发送到目的地。当目的地已回复一条确认消息，调用 `confirmDelivery` 方法。

```
1.  import akka.actor.{ Actor, ActorPath }
2.  import akka.persistence.AtLeastOnceDelivery
3.
4.  case class Msg(deliveryId: Long, s: String)
5.  case class Confirm(deliveryId: Long)
6.
7.  sealed trait Evt
8.  case class MsgSent(s: String) extends Evt
9.  case class MsgConfirmed(deliveryId: Long) extends Evt
10.
11. class MyPersistentActor(destination: ActorPath)
12.   extends PersistentActor with AtLeastOnceDelivery {
13.
14.   def receiveCommand: Receive = {
15.     case s: String           => persist(MsgSent(s))(updateState)
16.     case Confirm(deliveryId) => persist(MsgConfirmed(deliveryId))
    (updateState)
17.   }
18.
19.   def receiveRecover: Receive = {
20.     case evt: Evt => updateState(evt)
21.   }
22.
23.   def updateState(evt: Evt): Unit = evt match {
24.     case MsgSent(s) =>
25.       deliver(destination, deliveryId => Msg(deliveryId, s))
26.
```

```
27.     case MsgConfirmed(deliveryId) => confirmDelivery(deliveryId)
28.   }
29. }
30.
31. class MyDestination extends Actor {
32.   def receive = {
33.     case Msg(deliveryId, s) =>
34.       // ...
35.       sender() ! Confirm(deliveryId)
36.   }
37. }
```

`deliver` 和 `confirmDelivery` 之间的相关性，是通过传入 `deliveryIdToMessage` 函数的 `deliveryId` 参数进行的。通常在消息中包含 `deliveryId` 传递到目的地，然后用一个包含相同 `deliveryId` 的消息进行答复。

`deliveryId` 是无间隙严格单调递增序列号。相同的序列将用于所有目标actor，即当发送到多个目标时会看到序列中的空白，如果没有执行转译。

`AtLeastOnceDelivery` 特质具有未经确认的消息和一个序列号组成的一个状态。它并不存储这个状态本身。你必须持久化从你的 `PersistentActor` 调用 `deliver` 和 `confirmDelivery` 所对应的事件，从而可以通过调用相同的方法在 `PersistentActor` 的恢复阶段恢复状态。有时这些事件可以来自其他业务级别的事件，而有时你必须创建单独的事件。在恢复过程中 `deliver` 的调用不会发出消息，但如果没有匹配的 `confirmDelivery` 执行，它将稍后发送。

支持快照功能是 `getDeliverySnapshot` 和 `setDeliverySnapshot` 提供的。 `AtLeastOnceDeliverySnapshot` 包含完整的投递状态，包括未经确认的消息。如果你需要一个自定义的快照保存actor其他部分的状态，你还必须包括 `AtLeastOnceDeliverySnapshot` 。它使用 `protobuf` 序列化，

即利用Akka的通用序列化机制。最简单的方法是将 `AtLeastOnceDeliverySnapshot` 中的字节作为blob包含在你自定义的快照中。

重发尝试之间的间隔是由 `redeliverInterval` 方法定义的。其默认值可以用 `akka.persistence.at-least-once-delivery.redeliver-interval` 配置键来配置。可以在实现类中重写该方法来返回非默认值。

经过若干次尝试后，一个 `AtLeastOnceDelivery.UnconfirmedWarning` 消息将发送到 `self` 。重新发送仍会继续，但你可以选择调用 `confirmDelivery` 来取消重新发送。 `warnAfterNumberOfUnconfirmedAttempts` 方法定义发出警告之前传递尝试的次数。其默认值可以用 `akka.persistence.at-least-once-delivery.warn-after-number-of-unconfirmed-attempts` 配置键配置。可以用实现类重写该方法来返回非默认值。

`AtLeastOnceDelivery` 特质将消息保留在内存中，直到他们成功投递已被确认。actor能保留在内存中的未经确认的消息的最大数目限制是由 `maxUnconfirmedMessages` 方法定义的。如果超过了此限制 `deliver` 方法将不会接受更多的消息，它将抛出 `AtLeastOnceDelivery.MaxUnconfirmedMessagesExceededException` 。可以用 `akka.persistence.at-least-once-delivery.max-unconfirmed-messages` 配置键配置其默认值。可以用实现类重写该方法来返回非默认值。

存储插件

对于日志和快照存储的存储后端在Akka持久化中是可插拔的。默认日志插件将消息写入LevelDB（见本地LevelDB日志）。默认快照存储插件将快照作为单独的文件写入本地文件系统（请参阅本地快照存储区）。应用程序可以通过实现一个插件API并通过配置激活它们来提供

他们自己的插件。插件开发需要以下引入：

```
import akka.actor.ActorSystem
import akka.persistence._
import akka.persistence.journal._
import akka.persistence.snapshot._
import akka.testkit.TestKit
import com.typesafe.config._
import org.scalatest.WordSpec

import scala.collection.immutable.Seq
import scala.concurrent.Future
import scala.concurrent.duration._
```

日志插件API

日志插件要扩展 `SyncWriteJournal` 或 `AsyncWriteJournal` 。 `SyncWriteJournal` 是一个actor，当存储后端的API只支持同步、阻塞写入时应扩展它。在这种情况下，要实现的方法是：

```
/**
 * Plugin API: synchronously writes a batch of persistent messages to the journal.
 * The batch write must be atomic i.e. either all persistent messages in the batch
 * are written or none.
 */
def writeMessages(messages: immutable.Seq[PersistentRepr]): Unit

/**
 * Plugin API: synchronously writes a batch of delivery confirmations to the journal.
 */
@deprecated("writeConfirmations will be removed, since Channels will be removed.", since = "2.3.4")
def writeConfirmations(confirmations:
```

```
    immutable.Seq[PersistentConfirmation]): Unit
13.
14.  /**
15.   * Plugin API: synchronously deletes messages identified by
    `messageIds` from the
16.   * journal. If `permanent` is set to `false`, the persistent
    messages are marked as
17.   * deleted, otherwise they are permanently deleted.
18.   */
19.  @deprecated("deleteMessages will be removed.", since = "2.3.4")
20.  def deleteMessages(messageIds: immutable.Seq[PersistentId],
    permanent: Boolean): Unit
21.
22.  /**
23.   * Plugin API: synchronously deletes all persistent messages up to
    `toSequenceNr`
24.   * (inclusive). If `permanent` is set to `false`, the persistent
    messages are marked
25.   * as deleted, otherwise they are permanently deleted.
26.   */
27.  def deleteMessagesTo(persistenceId: String, toSequenceNr: Long,
    permanent: Boolean): Unit
```

当存储后端的API支持异步、非阻塞写入时，应扩展 `AsyncWriteJournal` 这个actor。在这种情况下，要实现的方法是：

```
1.  /**
2.   * Plugin API: asynchronously writes a batch of persistent messages
    to the journal.
3.   * The batch write must be atomic i.e. either all persistent
    messages in the batch
4.   * are written or none.
5.   */
6.  def asyncWriteMessages(messages: immutable.Seq[PersistentRepr]):
    Future[Unit]
7.
8.  /**
```

```
 9.   * Plugin API: asynchronously writes a batch of delivery
     confirmations to the journal.
10.   */
11.  @deprecated("writeConfirmations will be removed, since Channels
     will be removed.", since = "2.3.4")
12.  def asyncWriteConfirmations(confirmations:
     immutable.Seq[PersistentConfirmation]): Future[Unit]
13.
14.  /**
15.   * Plugin API: asynchronously deletes messages identified by
     `messageIds` from the
16.   * journal. If `permanent` is set to `false`, the persistent
     messages are marked as
17.   * deleted, otherwise they are permanently deleted.
18.   */
19.  @deprecated("asyncDeleteMessages will be removed.", since =
     "2.3.4")
20.  def asyncDeleteMessages(messageIds: immutable.Seq[PersistentId],
     permanent: Boolean): Future[Unit]
21.
22.  /**
23.   * Plugin API: asynchronously deletes all persistent messages up to
     `toSequenceNr`
24.   * (inclusive). If `permanent` is set to `false`, the persistent
     messages are marked
25.   * as deleted, otherwise they are permanently deleted.
26.   */
27.  def asyncDeleteMessagesTo(persistenceId: String, toSequenceNr:
     Long, permanent: Boolean): Future[Unit]
```

消息重播和序列号恢复始终是异步的，因此任何日志插件必须实现：

```
1.  /**
2.   * Plugin API: asynchronously replays persistent messages.
     Implementations replay
3.   * a message by calling `replayCallback`. The returned future must
     be completed
4.   * when all messages (matching the sequence number bounds) have
```

```
    been replayed.
 5.  * The future must be completed with a failure if any of the
    persistent messages
 6.  * could not be replayed.
 7.  *
 8.  * The `replayCallback` must also be called with messages that have
    been marked
 9.  * as deleted. In this case a replayed message's `deleted` method
    must return
10.  * `true`.
11.  *
12.  * The channel ids of delivery confirmations that are available for
    a replayed
13.  * message must be contained in that message's `confirms` sequence.
14.  *
15.  * @param persistenceId persistent actor id.
16.  * @param fromSequenceNr sequence number where replay should start
    (inclusive).
17.  * @param toSequenceNr sequence number where replay should end
    (inclusive).
18.  * @param max maximum number of messages to be replayed.
19.  * @param replayCallback called to replay a single message. Can be
    called from any
20.  *                       thread.
21.  *
22.  * @see [[AsyncWriteJournal]]
23.  * @see [[SyncWriteJournal]]
24.  */
25. def asyncReplayMessages(persistenceId: String, fromSequenceNr:
    Long, toSequenceNr: Long, max: Long)(replayCallback: PersistentRepr
    ⇒ Unit): Future[Unit]
26.
27. /**
28.  * Plugin API: asynchronously reads the highest stored sequence
    number for the
29.  * given `persistenceId`.
30.  *
31.  * @param persistenceId persistent actor id.
```

```
32.   * @param fromSequenceNr hint where to start searching for the
      highest sequence
33.   *                      number.
34.   */
35.  def asyncReadHighestSequenceNr(persistenceId: String,
      fromSequenceNr: Long): Future[Long]
```

日志插件可以以下最小配置下激活：

```
1.  # Path to the journal plugin to be used
2.  akka.persistence.journal.plugin = "my-journal"
3.
4.  # My custom journal plugin
5.  my-journal {
6.    # Class name of the plugin.
7.    class = "docs.persistence.MyJournal"
8.    # Dispatcher for the plugin actor.
9.    plugin-dispatcher = "akka.actor.default-dispatcher"
10. }
```

指定的插件 `class` 必须具有一个无参数构造函数。 `plugin-dispatcher` 是用于插件actor的调度程序。如果未指定，则默认是对 `SyncWriteJournal` 插件的 `akka.persistence.dispatchers.default-plugin-dispatcher` 和对 `AsyncWriteJournal` 插件的 `akka.actor.default-dispatcher` 。

快照存储插件API

一个快照存储插件必须扩展 `SnapshotStore` actor并实现以下方法：

```
1.  /**
2.   * Plugin API: asynchronously loads a snapshot.
3.   *
4.   * @param persistenceId processor id.
5.   * @param criteria selection criteria for loading.
6.   */
```

```
 7.  def loadAsync(persistenceId: String, criteria:
     SnapshotSelectionCriteria): Future[Option[SelectedSnapshot]]
 8.
 9.  /**
10.   * Plugin API: asynchronously saves a snapshot.
11.   *
12.   * @param metadata snapshot metadata.
13.   * @param snapshot snapshot.
14.   */
15.  def saveAsync(metadata: SnapshotMetadata, snapshot: Any):
     Future[Unit]
16.
17.  /**
18.   * Plugin API: called after successful saving of a snapshot.
19.   *
20.   * @param metadata snapshot metadata.
21.   */
22.  def saved(metadata: SnapshotMetadata)
23.
24.  /**
25.   * Plugin API: deletes the snapshot identified by `metadata`.
26.   *
27.   * @param metadata snapshot metadata.
28.   */
29.
30.  def delete(metadata: SnapshotMetadata)
31.
32.  /**
33.   * Plugin API: deletes all snapshots matching `criteria`.
34.   *
35.   * @param persistenceId processor id.
36.   * @param criteria selection criteria for deleting.
37.   */
38.  def delete(persistenceId: String, criteria:
     SnapshotSelectionCriteria)
```

可通过以下最小配置激活快照存储插件：

```
# Path to the snapshot store plugin to be used
akka.persistence.snapshot-store.plugin = "my-snapshot-store"

# My custom snapshot store plugin
my-snapshot-store {
  # Class name of the plugin.
  class = "docs.persistence.MySnapshotStore"
  # Dispatcher for the plugin actor.
  plugin-dispatcher = "akka.persistence.dispatchers.default-plugin-dispatcher"
}
```

指定的插件 `class` 必须具有一个无参数构造函数。 `plugin-dispatcher` 是用于插件actor的调度程序。如果未指定，则默认为 `akka.persistence.dispatchers.default-plugin-dispatcher` 。

插件TCK

为了帮助开发人员构建正确和高质量存储插件，我们提供技术兼容性工具包（简称TCK）。

TCK可用于Java或Scala项目中，对Scala你需要引入 `akka-persistence-tck-experimental` 依赖关系：

```
"com.typesafe.akka" %% "akka-persistence-tck-experimental" % "2.3.5" % "test"
```

要在你的测试套件中包括日志TCK的测试，只需要扩展提供的 `JournalSpec` ：

```
class MyJournalSpec extends JournalSpec {
  override val config = ConfigFactory.parseString(
    """
      |akka.persistence.journal.plugin = "my.journal.plugin"
    """.stripMargin)
```

```
6. }
```

我们还提供一个简单的基准测试类 `JournalPerfSpec` ，包括所有 `JournalSpec` 有的测试，还会执行日志上的一些长操作，并打印性能统计数据。虽然它并不旨在提供一个适当的基准测试环境，它仍可以被用于对典型的应用场景下你的日志表现进行粗略的感受。

要在你的测试套件中包括 `SnapshotStore` TCK测试，只需要扩展 `SnapshotStoreSpec` ：

```
1. class MySnapshotStoreSpec extends SnapshotStoreSpec {
2.   override val config = ConfigFactory.parseString(
3.     """
4.       |akka.persistence.snapshot-store.plugin = "my.snapshot-
   store.plugin"
5.     """.stripMargin)
6. }
```

在你的插件需要一些初始设置的情况下（启动模拟数据库，删除临时文件等），你可以重写 `beforeAll` 和 `afterAll` 来钩入测试生命周期：

```
1. class MyJournalSpec extends JournalSpec {
2.   override val config = ConfigFactory.parseString(
3.     """
4.       |akka.persistence.journal.plugin = "my.journal.plugin"
5.     """.stripMargin)
6.
7.   val storageLocations = List(
8.     new
   File(system.settings.config.getString("akka.persistence.journal.level
9.     new File(config.getString("akka.persistence.snapshot-
   store.local.dir")))
10.
11.   override def beforeAll() {
12.     super.beforeAll()
13.     storageLocations foreach FileUtils.deleteRecursively
```

```
14.   }
15.
16.   override def afterAll() {
17.     storageLocations foreach FileUtils.deleteRecursively
18.     super.afterAll()
19.   }
20.
21. }
```

我们强烈建议在你的测试套件包括这些规格，因为从头编写一个插件时，它们涵盖了广泛的，你可能会遗忘的测试用例。

预先包装好的插件

本地LevelDB日志" class="reference-link">本地LevelDB日志

默认日志插件是 `akka.persistence.journal.leveldb` ，它将消息写入到本地的LevelDB实例。LevelDB文件的默认位置是当前工作目录中一个名为 `journal` 的目录。此位置可以由配置中指定的相对或绝对的路径更改：

```
1. akka.persistence.journal.leveldb.dir = "target/journal"
```

用这个插件，每个actor系统可运行其自己私有的LevelDB实例。

共享LevelDB日志

一个LevelDB实例还可以由多个actor系统（在相同或不同节点上）共享。它，例如，允许持久化actor进行故障转移到备份节点，并从备份节点继续使用共享的日志实例。

> 警告
>
> 共享的*LevelDB*实例是单点故障，因此应仅用于测试目的。高可用、带复本的日志可以从社区插件中获得。

通过实例化 `SharedLeveldbStore` actor可以启动一个共享的LevelDB实例。

```
  }
}

class MyJournal extends AsyncWriteJournal {
  def asyncWriteMessages(messages: Seq[PersistentRepr]): Future[Unit] = ???
  def asyncWriteConfirmations(confirmations: Seq[PersistentConfirmation]): Future[Unit] = ???
  def asyncDeleteMessages(messageIds: Seq[PersistentId], permanent: Boolean): Future[Unit] = ???
  def asyncDeleteMessagesTo(persistenceId: String, toSequenceNr: Long, permanent: Boolean): Future[Unit] = ???
  def asyncReplayMessages(persistenceId: String, fromSequenceNr: Long, toSequenceNr: Long, max: Long)(replayCallback: (PersistentRepr) => Unit): Future[Unit] = ???
  def asyncReadHighestSequenceNr(persistenceId: String, fromSequenceNr: Long): Future[Long] = ???
}

class MySnapshotStore extends SnapshotStore {
  def loadAsync(persistenceId: String, criteria: SnapshotSelectionCriteria): Future[Option[SelectedSnapshot]] = ???
  def saveAsync(metadata: SnapshotMetadata, snapshot: Any): Future[Unit] = ???
  def saved(metadata: SnapshotMetadata): Unit = ???
  def delete(metadata: SnapshotMetadata): Unit = ???
  def delete(persistenceId: String, criteria: SnapshotSelectionCriteria): Unit = ???
}

object PersistenceTCKDoc {
  new AnyRef {
    import akka.persistence.journal.JournalSpec

    class MyJournalSpec extends JournalSpec {
```

```
26.       override val config = ConfigFactory.parseString(
27.         """
28.           |akka.persistence.journal.plugin = "my.journal.plugin"
29.         """.stripMargin)
30.     }
31.   }
32.   new AnyRef {
33.     import akka.persistence.snapshot.SnapshotStoreSpec
34.
35.     class MySnapshotStoreSpec extends SnapshotStoreSpec {
36.       override val config = ConfigFactory.parseString(
37.         """
38.           |akka.persistence.snapshot-store.plugin = "my.snapshot-
    store.plugin"
39.         """.stripMargin)
40.     }
41.   }
42.   new AnyRef {
43.     import java.io.File
44.
45.     import akka.persistence.journal.JournalSpec
46.     import org.iq80.leveldb.util.FileUtils
47.
48.     class MyJournalSpec extends JournalSpec {
49.       override val config = ConfigFactory.parseString(
50.         """
51.           |akka.persistence.journal.plugin = "my.journal.plugin"
52.         """.stripMargin)
53.
54.       val storageLocations = List(
55.         new
    File(system.settings.config.getString("akka.persistence.journal.level
56.         new File(config.getString("akka.persistence.snapshot-
    store.local.dir")))
57.
58.       override def beforeAll() {
59.         super.beforeAll()
60.         storageLocations foreach FileUtils.deleteRecursively
```

```
61.       }
62.
63.       override def afterAll() {
64.         storageLocations foreach FileUtils.deleteRecursively
65.         super.afterAll()
66.       }
67.
68.     }
69.   }
70. }
```

默认情况下，共享的实例将日志消息写入到当前的工作目录中一个名为 `journal` 的本地目录。可以通过配置更改存储位置：

```
1. akka.persistence.journal.leveldb-shared.store.dir = "target/shared"
```

使用共享的LevelDB存储的actor系统必须激活 `akka.persistence.journal.leveldb-shared` 插件。

```
1. akka.persistence.journal.plugin =
   "akka.persistence.journal.leveldb-shared"
```

这个插件必须由注入（远程的） `SharedLeveldbStore` actor引用来进行初始化。注入是通过以actor引用作为参数调用 `SharedLeveldbJournal.setStore` 方法完成的。

```
1. trait SharedStoreUsage extends Actor {
2.   override def preStart(): Unit = {
3.
   context.actorSelection("akka.tcp://example@127.0.0.1:2552/user/store"
    ! Identify(1)
4.   }
5.
6.   def receive = {
7.     case ActorIdentity(1, Some(store)) =>
```

```
8.         SharedLeveldbJournal.setStore(store, context.system)
9.     }
10. }
```

内部日志命令（由持久化actor发送的）会缓冲直到注入完成。注入是幂等的，即只有第一次的注入被使用。

本地快照存储区" class="reference-link">本地快照存储区

默认快照存储插件是 `akka.persistence.snapshot-store.local` 。它将快照文件写入本地文件系统。默认的存储位置是当前工作目录中一个名为 `snapshots` 的目录。这可以通过配置中指定的相对或绝对的路径来更改：

```
1. akka.persistence.snapshot-store.local.dir = "target/snapshots"
```

自定义序列化

快照序列化和 `Persistent` 消息的有效载荷是可以通过Akka序列化基础架构配置的。例如，如果应用程序想要序列化

- 有效载荷的 `MyPayload` 类型与自定义的 `MyPayloadSerializer` 和
- 快照的类型 `MySnapshot` 与自定义的 `MySnapshotSerializer`

它必须添加

```
1. akka.actor {
2.   serializers {
3.     my-payload = "docs.persistence.MyPayloadSerializer"
4.     my-snapshot = "docs.persistence.MySnapshotSerializer"
5.   }
6.   serialization-bindings {
7.     "docs.persistence.MyPayload" = my-payload
8.     "docs.persistence.MySnapshot" = my-snapshot
```

```
  }
}
```

在应用程序配置中。如果未指定，则使用默认的序列化程序。

测试

运行测试时使用 `sbt` 的LevelDB默认设置，请确保在你的sbt项目中设置 `fork := true` ，否则你将看到一个 `UnsatisfiedLinkError` 。或者，你可以切换到一个LevelDB Java 端口，通过这样的设置

```
akka.persistence.journal.leveldb.native = off
```

或

```
akka.persistence.journal.leveldb-shared.store.native = off
```

在Akka配置中。LevelDB 的Java端口仅用于测试目的。

杂项

状态机

状态机可以通过将 `FSM` 特质混入持久化actor来实现持久化。

```
import akka.actor.FSM
import akka.persistence.{ Persistent, Processor }

class PersistentDoor extends Processor with FSM[String, Int] {
  startWith("closed", 0)

  when("closed") {
    case Event(Persistent("open", _), counter) =>
      goto("open") using (counter + 1) replying (counter)
  }
```

```
11.
12.    when("open") {
13.      case Event(Persistent("close", _), counter) =>
14.        goto("closed") using (counter + 1) replying (counter)
15.    }
16.  }
```

配置

配置中有几个属性为持久化模块使用，请参阅参考配置。

多节点测试

多节点测试

> 注：本节未经校验，如有问题欢迎提*issue*

多节点测试概念

当我们谈论Akka多节点测试时，我们指的是多个actor系统在不同Jvm上运行协同测试的过程。多节点测试套件由三个主要部分组成。

- `测试导体`。协调和控制测试的节点。
- `多节点规格`。启动`TestConductor`的方便的包装器，并允许所有节点连接到它。
- `SbtMultiJvm`插件。启动可能在多个机器上的多个Jvm的测试。

测试导线" class="reference-link">测试导线

多节点测试的基础是 `TestConductor` 。它是插入到网络栈的一个Akka

扩展，它用来协调参与测试的节点，并提供多种功能，包括：

- 节点地址查找： 找出另一个测试节点的完整路径 （不需要在测试节点之间共享配置）
- 节点屏障协调： 在指定屏障等待其他节点。
- 网络故障注入： 流量限制、丢失的数据包、拔出节点并重新插入。

这是测试指挥示意性概述。

.. image:: ../images/akka-remote-testconductor.png

测试导体服务器负责协调屏障，并向对他们采取行动的测试导体客户端发送命令，例如限制从/到另一个客户端的网络流量。有关可能操作的详细信息见 `akka.remote.testconductor.Conductor` API 文档。

多节点规范" class="reference-link">多节点规范

多节点规范由两部分组成。负责对常见配置，并且枚举和命名测试节点的 `MultiNodeConfig` 。 `MultiNodeSpec` 中包含许多方便的函数使测试节点彼此交互。有关可能的操作的详细信息见 `akka.remote.testkit.MultiNodeSpec` API 文档。

`MultiNodeSpec` 的启动设置是通过java 系统属性配置的，你需要对要运行测试的节点的所有Jvm设定的。这些可以轻松地通过在 JVM 命令行中使用 `-Dproperty=value` 设置。

以下是是可用的属性：

- `multinode.max-nodes`

  一个测试可以有的节点的最大数目。

- `multinode.host`

  此节点的主机名或 IP 。必须能够使用 `InetAddress.getByName` 解析。

- `multinode.port`

  此节点端口号。默认值为 0，此时将使用一个随机端口。

- `multinode.server-host`

  主机名或服务器节点 IP 。必须能够使用 `InetAddress.getByName` 解析。

- `multinode.server-port`

  服务器节点的端口号。默认值为 4711。

- `multinode.index`

  为测试定义的角色序列中本节点的索引。索引 0 是特别指定为服务器的机器。所有的故障注入和节流必须在这个节点进行。

SbtMultiJvm 插件" class="reference-link"> SbtMultiJvm 插件

SbtMultiJvm 插件已更新以支持运行多节点测试，通过自动生成相关的 `multinode.*` 属性。这意味着你可以轻松地在单台计算机上运行多节点测试，而不用任何特殊配置，只要像运行正常多jvm测试一样运行即可。这些测试无需任何更改就可以分布运行在多台机器上，只需使用插件的多节点附加。

多节点特定附加

该插件也有大量的新的 `multi-node-*` sbt 任务和支持多台机器上运

行测试的设置。必要的测试类和依赖项通过SbtAssembly被打包入一个 jar 文件分发给其他机器，其命名格式为 `<projectName>_<scalaVersion>-<projectVersion>-multi-jvm-assembly.jar` 。

> 注意
>
> 要能分发并在多个机器上启动测试，需假定系统主机和目标系统是类 *POSIX* 系统，并支持 *`ssh`* 和 *`rsync`* 。

以下都是可用的 sbt 多节点的设置：

- `multiNodeHosts`

  用于运行测试的主机，以 `user@host:java` 的形式，其中主机是唯一必须的部分。将覆盖文件中的设置。

- `multiNodeHostsFileName`

  用于在运行测试的主机读取的文件。与上述格式相同，每行一个。默认为项目基目录中的 `multi-node-test.hosts` 。

- `multiNodeTargetDirName`

  目标系统上的目录名称，用于复制 jar 文件。默认为ssh 用户基目录的 `multi-node-test` ，用于 rsync jar 文件。

- `multiNodeJavaName`

  目标机器上的默认 Java 可执行文件的名称。默认值为 `java` 。

这里是如何定义主机的一些示例：

- `localhost`

  本地主机上的当前用户，使用缺省 java。

- `user1@host1`

  `host1` 主机上的用户 `user1`，使用缺省 java。

- `user2@host2:/usr/lib/jvm/java-7-openjdk-amd64/bin/java`

  `host2` 主机上的用户 `user2`，使用 java 7。

- `host3:/usr/lib/jvm/java-6-openjdk-amd64/bin/java`

  `host3` 主机上的当前用户 `user2`，使用 java 6。

运行多节点测试

若要在多节点模式下从sbt中运行所有的多节点测试 （即分发 jar 文件并开启远程测试），使用 `multi-node-test` 任务：

```
1.  multi-node-test
```

若要在多 jvm 模式中运行所有测试（即在本地计算机上的所有 Jvm），执行：

```
1.  multi-jvm:test
```

若要运行单个测试使用 `multi-node-test-only` 任务：

```
1.  multi-node-test-only your.MultiNodeTest
```

若要在多 jvm 模式下运行单个测试，执行：

```
1.  multi-jvm:test-only your.MultiNodeTest
```

可以列出多个测试名称来运行多个特定的测试。sbt的 tab 键提示可以帮你很容易地写出测试的名称。

为你的项目准备多节点测试

多节点测试套件是一个单独的 `jar` 文件。请确保你的项目中包含以下依存关系：

```
1.  "com.typesafe.akka" %% "akka-multi-node-testkit" % "2.3.6"
```

如果你正在使用最新的每夜构建，你应该从“http://repo.typesafe.com/typesafe/snapshots/com/typesafe/akka/akka-multi-node-testkit_2.10/”选择一个带时间戳的Akka版本。我们建议不要使用``SNAPSHOT``以获得稳定的构建。

多节点测试示例

首先，我们需要一些脚手架来讲 `MultiNodeSpec` 与你最喜爱的测试框架钩起来。允许定义一个 `STMultiNodeSpec` 特质使用 `ScalaTest` 来启动和停止 `MultiNodeSpec` 。

```
1.  package sample.multinode
2.
3.  import org.scalatest.{ BeforeAndAfterAll, WordSpecLike }
4.  import org.scalatest.Matchers
5.  import akka.remote.testkit.MultiNodeSpecCallbacks
6.
7.  /**
8.   * Hooks up MultiNodeSpec with ScalaTest
9.   */
10. trait STMultiNodeSpec extends MultiNodeSpecCallbacks
11.   with WordSpecLike with Matchers with BeforeAndAfterAll {
12.
13.   override def beforeAll() = multiNodeSpecBeforeAll()
14.
15.   override def afterAll() = multiNodeSpecAfterAll()
```

```
16. }
```

然后我们需要定义一个配置。让我们利用两个节点 `"node1"` 和 `"node2"` ，并称之为 `MultiNodeSampleConfig` 。

```
1. package sample.multinode
2. import akka.remote.testkit.MultiNodeConfig
3. 
4. object MultiNodeSampleConfig extends MultiNodeConfig {
5.   val node1 = role("node1")
6.   val node2 = role("node2")
7. }
```

然后最终到节点测试代码。启动两个节点，并演示了一个屏障，和一个远程actor消息发送和接收。

```
1. package sample.multinode
2. import akka.remote.testkit.MultiNodeSpec
3. import akka.testkit.ImplicitSender
4. import akka.actor.{ Props, Actor }
5. 
6. class MultiNodeSampleSpecMultiJvmNode1 extends MultiNodeSample
7. class MultiNodeSampleSpecMultiJvmNode2 extends MultiNodeSample
8. 
9. object MultiNodeSample {
10.   class Ponger extends Actor {
11.     def receive = {
12.       case "ping" => sender() ! "pong"
13.     }
14.   }
15. }
16. 
17. class MultiNodeSample extends MultiNodeSpec(MultiNodeSampleConfig)
18.   with STMultiNodeSpec with ImplicitSender {
19. 
20.   import MultiNodeSampleConfig._
21.   import MultiNodeSample._
```

```
22.
23.   def initialParticipants = roles.size
24.
25.   "A MultiNodeSample" must {
26.
27.     "wait for all nodes to enter a barrier" in {
28.       enterBarrier("startup")
29.     }
30.
31.     "send to and receive from a remote node" in {
32.       runOn(node1) {
33.         enterBarrier("deployed")
34.         val ponger = system.actorSelection(node(node2) / "user" /
    "ponger")
35.         ponger ! "ping"
36.         expectMsg("pong")
37.       }
38.
39.       runOn(node2) {
40.         system.actorOf(Props[Ponger], "ponger")
41.         enterBarrier("deployed")
42.       }
43.
44.       enterBarrier("finished")
45.     }
46.   }
47. }
```

运行此示例最简单的方法是下载Typesafe Activator并打开名为Akka Multi-Node Testing Sample with Scala的教程。

需要记住的事情

当写多节点测试时有几件事情要牢记在心，否则你的测试行为可能会出现意想不到的行为。

- 不要关闭第一个节点。第一个节点是控制器，而如果它关闭则你的

测试将中断。

- 要能够使用`blackhole`，`passThrough`和`throttle`，你必须激活故障注入器和节流传输适配器，通过在你的 MultiNodeConfig 中指定`testTransport(on = true)`。
- 限流，关闭和其他失败注解只可以从第一个节点，又即控制器发出。
- 不要再借点关闭后使用`node(address)`询问节点地址。要在关闭节点之前抓取其地址。
- 不要在主线程之外的其他线程使用 MultiNodeSpec 方法，比如地址查找，屏障进入等。这也意味着你不应该在actor内部或一个计划的任务中使用他们。

配置

多节点测试模块的几个配置属性，请参考配置一节。

Actors(使用Java的Lambda支持)

Actors(使用Java的Lambda支持)

FSM(使用Java的Lambda支持)

FSM(使用Java的Lambda支持)

外部贡献

外部贡献

> 注：本节未经校验，如有问题欢迎提*issue*

该子项目用来作为外部开发人员贡献模块的地方，随着时间的推移它可能会，也可能不会进入正式支持的代码中。这种转变可以发生的条件包括：

- 必须有足够的兴趣把模块列入标准的发行版中，
- 必须积极维护模块
- 代码质量必须足够好，允许由Akka核心开发团队有效的维护

如果贡献被发现不足以”起飞”，它可能会再次在稍后的时间移除。

买者自负

在此子项目中的模块并不都得服从小版本之间的二进制兼容规则。次要版本中可能引入破坏 API 的更改并且没有通知，只是基于用户的反馈意见。一个模块可能未经事先通知而丢弃。Typesafe订阅并不包括对这些模块的支持。

当前模块的列表

- 可靠的代理模式

    - 可靠代理简介
    - 它保证的到底是什么？
        - 连接到目标
    - 如何使用它
        - 配置
    - Actor契约
        - 它处理的消息
        - 它发送的消息
        - 它升级的异常
        - 它的参数

- Actor消息节流
    - 介绍
    - 如何使用它
    - 保障
- Java 日志记录 (JUL)
- 使用显式确认的邮箱[
- 群集单例
    - 需要意识到的潜在问题
    - 一个例子
- 群集Sharding
    - 一个Java示例
    - 一个Scala示例
    - 它是如何工作
    - 仅代理模式

    - 钝化
    - 配置
- 分布式发布订阅群集
    - 一个Java示例
    - 一个Scala示例
    - DistributedPubSubExtension
- 群集客户机
    - 一个例子
    - ClusterReceptionistExtension
- 聚合器模式
    - 介绍
    - 使用
    - 样本用例 - AccountBalanceRetriever
    - 样本用例 - 多个响应聚合和链接
    - 陷阱

利用这些贡献的建议方式

由于Akka团队不限制到此子项目的更新，即使在二进制兼容的切换版本中，而且有可能在没有警告的情况下删除模块，因此建议将源文件复制到你自己的代码基中，更改包名称。通过这种方式，你可以选择何时更新或哪些修补程序需要包括进来（以保持二进制兼容性，如果需要的话），而且更高版本的Akka不可能破坏你的应用程序。

建议的贡献格式

每个贡献应该是一个自包含的单元，包括一个源文件或一个专门使用的

包，没有对此子项目中其他模块的依赖；不过它可以依赖于Akka发布。这将确保贡献可以单独移动到标准发布中。该模块应在 `akka.contrib` 的一个子包中。

每个模块必须伴随一个测试套件验证其提供的功能，可能辅之以集成和单元测试。测试应遵守开发者指南，并放在 `src/test/scala` 或 `src/test/java` 目录中（被测试的模块需要正确匹配包名）。例如，如果该模块被称为 `akka.contrib.pattern.ReliableProxy`，则其测试套件应命名为 `akka.contrib.pattern.ReliableProxySpec`。

每个模块还必须有用reStructured Text编写的恰当的文档。文档应该是一个单一的 `<module>.rst` 文件，在 `akka-contrib/docs` 目录中，包括从 `index.rst`（此文件）的链接。

Akka开发者信息

Akka开发者信息

构建Akka

构建Akka

> 注：本节未经校验，如有问题欢迎提*issue*

此页介绍如何从最新的源代码构建和运行Akka。

获取源代码

Akka使用 Git 并托管于Github。

你首先需要在你的计算机上安装了 Git。然后你可以从 http://github.com/akka/akka源代码存储库克隆。

举个例子：

```
1.  git clone git://github.com/akka/akka.git
```

如果你以前已经被克隆存储库，则你可以使用 `git pull` 更新代码：

sbt-简单生成工具

Akka使用优秀的 sbt 构建系统。所以你要做的第一件事是要下载并安装sbt。你可以在 sbt 安装文档阅读更多。

所有你需要构建Akka的 sbt 命令会在下面介绍。如果你想要了解更多有关 sbt 的内容，并在自己项目中使用它，请阅读 sbt 文档。

Akka sbt 构建文件是 `project/AkkaBuild.scala` 。

构建Akka

首先请确保你在Akka代码目录中：

```
1. cd akka
```

构建

要编译的Akka核心模块使用 `compile` 命令：

```
1. sbt compile
```

可以使用 `test` 命令来运行所有测试：

```
1. sbt test
```

如果编译和测试成功然，则你就会拥有一份可以工作的Akka最新的开发版本。

并行执行

默认情况下测试按顺序执行。他们可以并行执行来减少构建时间，如果硬件能控制更大的内存和 cpu 使用率。将以下系统属性添加到sbt的启动脚本来激活并行执行：

```
-Dakka.parallelExecution=true
```

长时间运行和时间敏感试验

默认情况下长时间运行的测试（主要是群集测试）和时间敏感测试（取决于运行它的机器的性能） 是禁用的。你可以通过添加以下标志启用它们：

```
-Dakka.test.tags.include=long-running
-Dakka.test.tags.include=timing
```

或者如果你需要启用两者：

```
-Dakka.test.tags.include=long-running,timing
```

发布到Ivy本地存储库

如果你想要将项目部署到Ivy本地资源库 （例如，从 sbt 项目中使用）， 使用 `publish-local` 命令：

```
sbt publish-local
```

> 注意
>
> *Akka*使用 *ScalaDoc* 为 *API* 文档生成类图。这需要安装 *Graphviz* 软件包的 `dot` 命令来避免错误。你可以通过添加 `-Dakka.scaladoc.diagrams=false` 标志来禁用图生成

sbt 交互模式

请注意在上面的例子我们调用 `sbt compile` 和 `sbt test` 等等，但sbt也有一种互动的模式。如果你键入 `sbt` 就进入了 sbt 交互式提示符，并可以直接输入命令。这节省了为每个命令启动一个新的 JVM 实例的开销，从而可以更快、更方便。

例如，Akka构建一般是这样做的：

```
% sbt
[info] Set current project to default (in build file:/.../akka/project/plugins/)
[info] Set current project to akka (in build file:/.../akka/)
> compile
...
> test
...
```

sbt 批处理模式

也可以在单个调用中组合命令。例如，可以像这样测试和发布Akka到本地的Ivy资源库中：

```
sbt test publish-local
```

依赖

你可以查看通过 `sbt update` 创建的Ivy依赖解析信息，并在 `~/.ivy2/cache` 中找到。例如， `~/.ivy2/cache/com.typesafe.akka-akka-remote-compile.xml` 文件包含Akka远程模块编译依赖的解析信息。如果在 web 浏览器中打开此文件，你会得到一个易于导航的依赖关系视图。

多JVM测试

多JVM测试

> 注：本节未经校验，如有问题欢迎提*issue*

同时在多个 Jvm 中运行应用程序 （main方法的对象） 和 ScalaTest 测试的支持。对于需要多个系统相互沟通的集成测试很有用。

安装程序

多JVM测试是 sbt 插件，你可以在 http://github.com/typesafehub/sbt-multi-jvm 找到。

你可以作为一个插件添加它，在 project/plugins.sbt 添加以下内容：

```
1.  addSbtPlugin ("com.typesafe.sbt"%"sbt 多 jvm"%"0.3.8")
```

然后，你可以通过在 `build.sbt` 或 `project/Build.scala` 包含 `MultiJvm` 和设置添加多JVM 测试。请注意 MultiJvm 测试源代码

位于 `src/multi-jvm/...` 而不是在 `src/test/...` 。

下面是使用 MultiJvm 插件的 sbt 0.13 下的 build.sbt 文件示例:

```
1.  import com.typesafe.sbt.SbtMultiJvm
2.  import com.typesafe.sbt.SbtMultiJvm.MultiJvmKeys.MultiJvm
3.
4.  val akkaVersion = "2.3.6"
5.
6.  val project = Project(
7.    id = "akka-sample-multi-node-scala",
8.    base = file("."),
9.    settings = Project.defaultSettings ++
    SbtMultiJvm.multiJvmSettings ++ Seq(
10.     name := "akka-sample-multi-node-scala",
11.     version := "2.3.6",
12.     scalaVersion := "2.10.4",
13.     libraryDependencies ++= Seq(
14.       "com.typesafe.akka" %% "akka-remote" % akkaVersion,
15.       "com.typesafe.akka" %% "akka-multi-node-testkit" %
    akkaVersion,
16.       "org.scalatest" %% "scalatest" % "2.0" % "test"),
17.     // make sure that MultiJvm test are compiled by the default
    test compilation
18.     compile in MultiJvm <<= (compile in MultiJvm) triggeredBy
    (compile in Test),
19.     // disable parallel tests
20.     parallelExecution in Test := false,
21.     // make sure that MultiJvm tests are executed by the default
    test target,
22.     // and combine the results from ordinary test and multi-jvm
    tests
23.     executeTests in Test <<= (executeTests in Test, executeTests in
    MultiJvm) map {
24.       case (testResults, multiNodeResults)  =>
25.         val overall =
26.           if (testResults.overall.id < multiNodeResults.overall.id)
```

```
27.             multiNodeResults.overall
28.           else
29.             testResults.overall
30.         Tests.Output(overall,
31.           testResults.events ++ multiNodeResults.events,
32.           testResults.summaries ++ multiNodeResults.summaries)
33.     }
34.   )
35. ) configs (MultiJvm)
```

你可以为分支 Jvm 指定 JVM 选项:

```
1. jvmOptions in MultiJvm := Seq("-Xmx256M")
```

运行测试

多JVM任务是类似于正常的任务: `test`, `test-only` 和 `run`, 但在 `multi-jvm` 配置下。

所以在Akka中, 要运行akka-remote 中的所有多JVM测试 (在sbt 提示符中) :

```
1. akka-remote-tests/multi-jvm:test
```

或可以首先修改 `akka-remote-tests` 项目, 然后运行测试:

```
1. project akka-remote-tests
2. multi-jvm:test
```

使用 `test-only` 运行单个测试:

```
1. multi-jvm:test-only akka.remote.RandomRoutedRemoteActor
```

可以列出多个测试名称来运行多个特定的测试。使用sbt的 tab 键可

以很容易地完成测试的名称。

也可以通过在测试名称后和 `--` 包括这些选项，来指定 JVM 选项为 `test-only` 。举个例子：

```
multi-jvm:test-only akka.remote.RandomRoutedRemoteActor -- -Dsome.option=something
```

创建应用程序测试

测试通过一种命名约定被发现并结合起来。MultiJvm 测试源代码位于 `src/multi-jvm/...` 。测试按以下模式命名：

```
{TestName}MultiJvm{NodeName}
```

也就是，每个测试在其名字中有 `MultiJvm` 。前面的部分将测试/应用划分在单个 `TestName` 组下并将一起运行。后面的部分 `NodeName` ，是为每个分叉的 JVM分配的不同的名称。

因此若要创建名为 `Sample` 的3-节点测试，你可以如下创建三个应用程序：

```
package sample

object SampleMultiJvmNode1 {
  def main(args: Array[String]) {
    println("Hello from node 1")
  }
}

object SampleMultiJvmNode2 {
  def main(args: Array[String]) {
    println("Hello from node 2")
  }
}
```

```
14.
15.     object SampleMultiJvmNode3 {
16.       def main(args: Array[String]) {
17.         println("Hello from node 3")
18.       }
19.     }
```

当你在sbt命令行中调用 `multi-jvm:run sample.Sample` ，会产生三个Jvm，分别用于每个节点。看起来会像这样：

```
1. > multi-jvm:run sample.Sample
2. ...
3. [info] * sample.Sample
4. [JVM-1] Hello from node 1
5. [JVM-2] Hello from node 2
6. [JVM-3] Hello from node 3
7. [success] Total time: ...
```

更改默认设置

你可以通过在项目中添加以下配置来更改多JVM测试的源代码目录的名称：

```
1.    unmanagedSourceDirectories in MultiJvm <<=
2.       Seq(baseDirectory(_ / "src/some_directory_here")).join
```

你可以更改 `MultiJvm` 标识符。例如，使用 `multiJvmMarker` 设置更改它为 `ClusterTest` ：

```
1. multiJvmMarker in MultiJvm := "ClusterTest"
```

现在，你的测试应该命名为 `{TestName}ClusterTest{NodeName}` 。

JVM 实例配置

你可以为每个生成的 Jvm定义特定 JVM 选项。通过创建一个以节点的名字命名的带有 `.opts` 后缀的文件，把它们放在测试的同一个目录中。

例如，为 `SampleMultiJvmNode1` 提供 JVM 选项 `-Dakka.remote.port=9991` 和 `-Xmx256m`，让我们创建三个 `*.opts` 文件并向其中添加选项。使用空格分隔多个选项。

`SampleMultiJvmNode1.opts`：

```
-Dakka.remote.port=9991 -Xmx256m
```

`SampleMultiJvmNode2.opts`：

```
-Dakka.remote.port=9992 -Xmx256m
```

`SampleMultiJvmNode3.opts`：

```
-Dakka.remote.port=9993 -Xmx256m
```

ScalaTest

除了应用程序外，它还支持创建 ScalaTest 测试。要这样做，要如上所述使用相同的命名约定，但创建 ScalaTest 套件，而不是对象的main方法。你需要在类路径上有 ScalaTest。这里是一个类似于上面例子的ScalaTest代码：

```
package sample

import org.scalatest.WordSpec
import org.scalatest.matchers.MustMatchers

class SpecMultiJvmNode1 extends WordSpec with MustMatchers {
```

```
 7.   "A node" should {
 8.     "be able to say hello" in {
 9.       val message = "Hello from node 1"
10.       message must be("Hello from node 1")
11.     }
12.   }
13. }
14.
15. class SpecMultiJvmNode2 extends WordSpec with MustMatchers {
16.   "A node" should {
17.     "be able to say hello" in {
18.       val message = "Hello from node 2"
19.       message must be("Hello from node 2")
20.     }
21.   }
22. }
```

你需要在sbt提示中运行 `multi-jvm:test-only sample.Spec` 来执行这些测试。

多节点添加

此外对 `SbtMultiJvm` 插件有一些补充，以适应实验模块一节中描述的多节点测试。

I/O层设计

I/O层设计

> 注：本节未经校验，如有问题欢迎提*issue*

`akka.io` 包已由Akka和 `spray.io` 团队协作开发。其设计采用 `spray-io` 模块的开发经验，并加以改进为更一般基于actor的服务使用。

要求

为了形成通用的可扩展的IO层基础，使之适合于广泛的应用，在Akka remoting 和spray HTTP 是原有基础上，为设计的关键驱动因素建立了以下要求：

- 数以百万计的并发连接的可扩展性
- 从数据输入通道到进入目标actor邮箱要有最低可能的延迟
- 最大的吞吐量
- 在两个方向的可选back-pressure （即在议定允许的地方对本地发送者限流，同时允许本地读者对远程发送者限流）
- 一个纯粹基于actor的 API 和不可变的数据表示形式

- 通过很瘦的SPI集成新运输方式的可扩展性;目标是不会强迫迫I/O机制到最低共同标准，而是相反，完全允许特定协议的用户级API。

基本体系结构

每个传输实现，可作为一个单独的Akka扩展，提供一个 `ActorRef` 代表联系客户端代码的初始点。这个”管理器”可接受请求建立一个通信通道（例如连接或侦听一个TCP套接字）。每个通信通道都由一个特定actor代表，暴露给客户端代码，在其整个生命周期服务于与此通道的所有交互。

实现的核心要素是传输特定”selector”的actor ；例如TCP中它将包装一个 `java.nio.channels.Selector` 。通道actor通过发送相应消息到其指定的选择器actor，来注册其感兴趣的渠道的读或写。然而，实际的通道读写是由通道actor自己完成的，这样将选择器actor从耗时的任务中解放出来，从而保证了低延迟。选择器actor唯一的责任是管理的底层的选择器的键集合和实际选择操作，这是唯一会阻塞的操作。

通道到选择器的指定由管理actor操作。并且在通道的整个生存期内都保持不变。从而管理actor基于一些特定于实现的分配逻辑在一个或多个选择器actor中 “编制（stripes)” 新渠道。这种逻辑可能 （部分地） 委派给选择器actor，例如，当他们认为自己有能力处理的时候可以选择拒绝被指定一个新渠道。

管理actor创建 （并因此监督） 选择器actor，它们相应地创建并监督其频道actor。某一单一传输实现的actor层次结构因此包含三个截然不同的actor层次，管理actor在顶部，通道actor在叶子节点，选择器actor在中间。

背压(Back-pressure)输出通过允许用户能够在其 `Write` 消息中指

定它是否想收到会写入操作系统内核的排队确认消息来实现。背压输入通过向通道actor发送消息，暂时禁用通道的阅读兴趣，直到通过相应的恢复命令重新启用读操作来实现。在流量控制的运输情况下 —例如TCP — 在接收端不消费数据的行为 （从而使数据保持在内核读取缓冲区中） 会传播回给发送者，跨网络链接这两个机制。

设计的好处

整个实现保持在actor模型内，允许我们删除显式线程处理逻辑的需要，同时也意味着有没有涉及锁 （除了那些底层传输工具库的部分）。只编写actor代码会使实现更加简洁，同时Akka高效的actor消息传递没有对这一好处带来很大的开销。事实上 I/O 基于事件的性质能很好地映射到actor模型，在此模型中我们期望明确的性能和可伸缩性优势，而不是传统的解决方案的显式线程管理和同步。

监督层次结构的另一个好处是，清理资源是自然而然的： 关闭一个选择器的actor会自动清理所有通道actor，允许通道的恰当关闭，并将适当的消息发送到用户级客户端actor。DeathWatch允许通道actor注意到其用户级别的处理器actor的消亡，并在这种情况下也以一种有序的方式终止 ；这自然就减少了泄露未关闭通道的可能性。

使用 `ActorRef` 暴露所有功能的选择，确保了这些引用可以被自由的分发和代理，并在用户认为合适的时候处理一般情况，包括使用远程处理和生命周期监测（只是随便列出两个）。

怎样去添加一个新的传输

最好的开始是研究 TCP 引用的实现，从中可以良好的获取其基本工作原理，然后设计和实施，对新的协议是类似的，但也会有问题。有 I/O 机制之间差异巨大 （例如比较文件 I/O 和消息代理） ，此

I/O 层的目标是明确不能把他们都硬塞进一个统一的 API，这就是为什么只有基本体系结构的思想记录在这里。

开发指南

开发指南

> 注：本节未经校验，如有问题欢迎提*issue*
>
> 注意
>
> 首先阅读*Akka*贡献者指南。

代码风格

Akka代码风格遵循 Scala 风格指南。唯一的例外是块注释的样式：

```
1.   /**
2.     * Style mandated by "Scala Style Guide"
3.     */
4.
5.   /**
6.    * Style adopted in the Akka codebase
7.    */
```

Akka使用 `Scalariform` 作为构建的一部分的格式化源代码。所以随便修改，然后运行的 `sbt compile` ，它将重新格式化为Akka标准代

码。

过程

- 请确保你已经签署了Akka CLA，如果没有，在这里签署。
- 选择一个ticket，如果没有适合你的就创建一个。
- 首先在功能分支工作。类似 `wip-<ticket number>-<descriptive name>-<your username>` 这样命名。
- 当你完成后，创建一个 GitHub Pull-Request 到目标分支，并且向Akka邮件列表发送邮件表示你希望它被审阅。
- 审查达成一致后，Akka核心团队将对它进行合并。

提交消息

请按照如下指导方针创建公共提交和编写提交消息。

1. 如果你的工作跨越多个本地提交 （例如 ； 如果你在主题分支工作时的安全点提交，或在长时间工作分支做合并/rebase 等）则请不要提交所有，而是重写历史，将多个提交挤压到单个大提交中，并在此编写良好的提交消息（如下讨论）。有一篇好的文章介绍了如何做到这一点。每个提交都应该是能够用于隔离、cherry picked等。

2. 第一行应该是一个描述性的句子，说明提交在做什么。应该达到只是读这一行就能够充分了解提交做了什么。它不是只列出ticket编号、打入”次要修复”或类似的话。在第一行末尾加上以 `#` 开头的ticket编号。如果提交是一个小的修复程序，则已完成。否则请看下一条。

3. 单行描述之后应该是一个空行，然后跟一个提交细节的枚举列表。

示例：

```
Completed replication over BookKeeper based transaction log. Fixes
#XXX

  * Details 1
  * Details 2
  * Details 3
```

测试

签入的所有代码都应该有测试。所有的测试都是用 `ScalaTest` 和 `ScalaCheck` 编写。

- 命名测试为 **`Test.scala`**，如果他们不依赖于任何外部的东西。这使得必经之快乐。
- 命名测试为 **`Spec.scala`**，如果他们具有外部依赖项。

Actor测试工具包

测试actor的有用套件: `akka.util.TestKit`。它提供了确认收到的答复及其时间的断言，更多文档参考测试actor系统模块中。

多JVM测试

包含在示例中的是一个多 `jvm` 测试的 `sbt` 特质，将分枝 `Jvm`用于多节点测试。它支持运行应用程序 （含main方法的对象） 和运行 `ScalaTest` 测试。

NetworkFailureTest

你可以使用 ‘NetworkFailureTest’ 特质来测试网络故障。

文档指南

文档指南

The Akka documentation uses `reStructuredText` *as its markup language and is built using* `Sphinx` .

.. _reStructuredText: http://docutils.sourceforge.net/rst.html
.. _sphinx: http://sphinx.pocoo.org

Sphinx

For more details see `The Sphinx Documentation <http://sphinx.pocoo.org/contents.html>` _

reStructuredText

For more details see `The reST Quickref <http://docutils.sourceforge.net/docs/user/rst/quickref.html>`_

Sections

Section headings are very flexible in reST. We use the following convention in
the Akka documentation:

- `#` (over and under) for module headings
- `=` for sections
- `-` for subsections
- `^` for subsubsections
- `~` for subsubsubsections

Cross-referencing

Sections that may be cross-referenced across the documentation should be marked
with a reference. To mark a section use `.. _ref-name:` before the section

heading. The section can then be linked with ```` `ref-name` ````. These are
unique references across the entire documentation.

For example::

```
.. _akka-module:
```

#

Akka Module

#

This is the module documentation.

.. _akka-section:

Akka Section

Akka Subsection

Here is a reference to “akka section”:  `akka-section` which will have the
name “Akka Section”.

Build the documentation

First install `Sphinx` _. See below.

Building

For the html version of the docs::

```
1. sbt sphinx:generate-html
2.
3. open <project-dir>/akka-docs/target/sphinx/html/index.html
```

For the pdf version of the docs::

```
1. sbt sphinx:generate-pdf
```

```
2.
3. open <project-dir>/akka-docs/target/sphinx/latex/AkkaJava.pdf
4. or
5. open <project-dir>/akka-docs/target/sphinx/latex/AkkaScala.pdf
```

Installing Sphinx on OS X

Install `Homebrew <https://github.com/mxcl/homebrew>` _

Install Python and pip:

::

```
1.   brew install python
2.   /usr/local/share/python/easy_install pip
```

Add the Homebrew Python path to your $PATH:

::

```
1.  /usr/local/Cellar/python/2.7.5/bin
```

More information in case of trouble: https://github.com/mxcl/homebrew/wiki/Homebrew-and-Python

Install sphinx:

::

```
1.  pip install sphinx
```

Add sphinx_build to your $PATH:

::

```
/usr/local/share/python
```

Install BasicTeX package from: http://www.tug.org/mactex/morepackages.html

Add texlive bin to $PATH:

::

```
 /usr/local/texlive/2013basic/bin/universal-darwin
```

Add missing tex packages:

::

```
  sudo tlmgr update --self
  sudo tlmgr install titlesec
  sudo tlmgr install framed
  sudo tlmgr install threeparttable
  sudo tlmgr install wrapfig
  sudo tlmgr install helvetic
  sudo tlmgr install courier
  sudo tlmgr install multirow
```

If you get the error “unknown locale: UTF-8” when generating the documentation the solution is to define the following environment variables:

::

```
  export LANG=en_US.UTF-8
  export LC_ALL=en_US.UTF-8
```

团队

团队

```
==================== ==========================
Name Role
==================== ==========================
Jonas Bonér Founder, Despot, Committer
Viktor Klang Honorary Member
Roland Kuhn Project Lead
Patrik Nordwall Core Team
Björn Antonsson Core Team
Endre Varga Core Team
Mathias Doenitz Committer
Johannes Rudolph Committer
Raymond Roestenburg Committer
Piotr Gabryanczyk Committer
Helena Edelson Committer
Martin Krasser Committer
Henrik Engström Alumnus
Peter Vlugter Alumnus
Derek Williams Alumnus
Debasish Ghosh Alumnus
Ross McDonald Alumnus
Eckhart Hertzler Alumnus
Mikael Högqvist Alumnus
```

Tim Perrett Alumnus

Jeanfrancois Arcand Alumnus

Jan Van Besien Alumnus

Michael Kober Alumnus

Peter Veentjer Alumnus

Irmo Manie Alumnus

Heiko Seeberger Alumnus

Hiram Chirino Alumnus

Scott Clasen Alumnus

========== ==========

工程信息

工程信息

迁移指南

迁移指南

问题追踪

问题追踪

Akka使用 GitHub Issues作为其问题跟踪系统。

浏览

Tickets

在提交一张ticket之前，请检查现有的Akka tickets是否在早些时候报告了同样的问题。非常欢迎你在现有的ticket中发表评论，尤其是当你有可以分享的重复性测试用例的时候。

路线图

请参看Akka路线图，以找出即将发行的Akka版本中的总体主题。

创建ticket

请包括 *Scala* 和*Akka*的版本及相关配置文件。

如果你已经注册了 GitHub 的用户，你可以创建一张新问题单。

非常感谢报告 bug 以及建议功能 ！

许可证

许可证

Akka License

```
1.  This software is licensed under the Apache 2 license, quoted below.
2.
3.  Copyright 2009-2014 Typesafe Inc. <http://www.typesafe.com>
4.
5.  Licensed under the Apache License, Version 2.0 (the "License"); you
    may not
6.  use this file except in compliance with the License. You may obtain
    a copy of
7.  the License at
8.
9.      http://www.apache.org/licenses/LICENSE-2.0
10.
11. Unless required by applicable law or agreed to in writing, software
12. distributed under the License is distributed on an "AS IS" BASIS,
    WITHOUT
13. WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
    See the
14. License for the specific language governing permissions and
    limitations under
15. the License.
```

Akka Committer License Agreement

All committers have signed this CLA.
It can be signed online.

Licenses for Dependency Libraries

Each dependency and its license can be seen in the project build file (the comment on the side of each dependency):
AkkaBuild.scala

赞助商

赞助商

Typesafe

Typesafe is the company behind the Akka Project, Scala Programming Language,
Play Web Framework, Scala IDE, Simple Build Tool and many other open source
projects. It also provides the Typesafe Stack, a full-featured development
stack consisting of Akka, Play and Scala. Learn more at
typesafe.com.

YourKit

YourKit is kindly supporting open source projects with its full-featured Java Profiler.

YourKit, LLC is the creator of innovative and intelligent tools for profiling Java
and .NET applications. Take a look at YourKit’s leading software products:

YourKit Java Profiler
and YourKit .NET Profiler

项目

项目

Commercial Support

Commercial support is provided by `Typesafe <http://www.typesafe.com>` .

Akka is part of the `*Typesafe Reactive Platform <http://www.typesafe.com/platform>*` .

Mailing List

Akka User Google Group

Akka Developer Google Group

Downloads

`http://akka.io/downloads`

Source Code

Akka uses Git and is hosted at Github.

- Akka: clone the Akka repository from `http://github.com/akka/akka`

Releases Repository

All Akka releases are published via Sonatype to Maven Central, see search.maven.org

Snapshots Repository

Nightly builds are available in http://repo.akka.io/snapshots/ as both `SNAPSHOT` and timestamped versions.

For timestamped versions, pick a timestamp from @binVersion@/"">http://repo.akka.io/snapshots/com/typesafe/akka/akka-actor_@binVersion@/.
All Akka modules that belong to the same build have the same timestamp.

sbt definition of snapshot repository

Make sure that you add the repository to the sbt resolvers::

```
1.  resolvers += "Typesafe Snapshots" at
    "http://repo.akka.io/snapshots/"
```

Define the library dependencies with the timestamp as version. For example::

```
libraryDependencies += "com.typesafe.akka" % "akka-remote_@binVersion@" %
  "2.1-20121016-001042"
```

maven definition of snapshot repository

Make sure that you add the repository to the maven repositories in pom.xml::

```
  <repositories>
    <repository>
      <id>akka-snapshots</id>
      <name>Akka Snapshots</name>
      <url>http://repo.akka.io/snapshots/</url>
      <layout>default</layout>
    </repository>
  </repositories>
```

Define the library dependencies with the timestamp as version. For example::

```
  <dependencies>
    <dependency>
      <groupId>com.typesafe.akka</groupId>
      <artifactId>akka-remote_@binVersion@</artifactId>
      <version>2.1-20121016-001042</version>
    </dependency>
  </dependencies>
```

附加信息

附加信息

常见问题

常见问题

Akka Project

Where does the name Akka come from?

It is the name of a beautiful Swedish mountain up in the northern part of Sweden called Laponia. The mountain is also sometimes

called ‘The Queen of Laponia’.

Akka is also the name of a goddess in the Sámi (the native Swedish population)
mythology. She is the goddess that stands for all the beauty and good in the
world. The mountain can be seen as the symbol of this goddess.

Also, the name AKKA is the a palindrome of letters A and K as in Actor Kernel.

Akka is also:

- the name of the goose that Nils traveled across Sweden on in The Wonderful Adventures of Nils by the Swedish writer Selma Lagerlöf.
- the Finnish word for ‘nasty elderly woman’ and the word for ‘elder sister’ in the Indian languages Tamil, Telugu, Kannada and Marathi.
- a font
- a town in Morocco
- a near-earth asteroid

Actor总体

当我在Actor中使用Future时，sender()/getSender()消失了，为什么呢？

当在Actor内部使用Future回调时需要小心，避免闭合包含Actor的引用，即对闭合的Actor不在回调内调用方法或访问其内的可变状态。这将打破Actor封装，因为回调将被调度与闭合的Actor并发执行，所

以可能会引入同步 bug 和竞态条件。不幸的是没有编译时的方法发现这些非法访问。

更多内容参见Actor与共享可变状态的文档。

为什么发生OutOfMemoryError？

产生OutOfMemoryError错误的原因很多。例如，在基于纯推送系统中，消费者的消息处理速度可能低于相应消息生产者，必须添加某种消息流量控制。否则邮件将在消费者的邮箱中排队，从而填满堆内存。

一些寻找灵感的文章：

- Balancing Workload across Nodes with Akka 2
- Work Pulling Pattern to prevent mailbox overflow, throttle and distribute work

Actors Scala API

我怎么会得到`receive`方法丢失消息的编译时错误呢？

一种解决方案来帮助你获得警告，因为基于你的actor输入输出消息实现基本特质的定义，编译时将检查的处理消息的全面性，如有遗漏将会告警。

这里有一个例子，编译器将发出警告你接收的比配并不是详尽无遗的：

```
1.  object MyActor {
2.    // these are the messages we accept
3.    sealed abstract trait Message
4.    case class FooMessage(foo: String) extends Message
5.    case class BarMessage(bar: Int) extends Message
6.
7.    // these are the replies we send
8.    sealed abstract trait Reply
```

```
 9.    case class BazMessage(baz: String) extends Reply
10.  }
11.
12.  class MyActor extends Actor {
13.    import MyActor._
14.    def receive = {
15.      case message: Message => message match {
16.        case BarMessage(bar) => sender ! BazMessage("Got " + bar)
17.        // warning here:
18.        // "match may not be exhaustive. It would fail on the
     following input: FooMessage(_)"
19.      }
20.    }
21.  }
```

Remoting

我想发送到远程系统，但它没有做任何事情

请确保你有在两端上都启用远程处理：客户端和服务器。两端都需要配置主机名和端口，而且你将需要知道服务器端口；客户端在大多数情况下可以使用一个自动的港口（即配置端口为零）。如果这两个系统在同一网络主机上运行，它们的端口必须是不同的。

如果仍然看不到任何东西，看看远程的生命周期事件的日志记录了什么（通常在 INFO 级别） 或打开辅助的远程日志记录选项，以查看所有发送和接收消息 （在DEBUG级别）。

调试远程处理问题时，应启用哪些选项？

详见远程配置，典型的候选人是：

- `akka.remote.log-sent-messages`
- `akka.remote.log-received-messages`
- `akka.remote.log-remote-lifecycle-events` (this also includes

deserialization errors)

远程Actor的名字是什么？

当你想要将消息发送到远程主机上的Actor时，你需要知道它的完整路径，格式如下：

```
1.  akka.protocol://system@host:1234/user/my/actor/hierarchy/path
```

所需要的部分是：

- `protocol` 是要用来与远程系统进行通信的协议。大多数情况下是 `tcp` 。
- `system` 是远程系统的名称 （必须完全匹配，区分大小写！）
- `host` 是远程系统的 IP 地址或 DNS 名称，它必须匹配该系统的配置 （即 `akka.remote.netty.hostname` ）
- `1234` 是侦听连接和接收消息的远程系统的端口号
- `/user/my/actor/hierarchy/path` 是远程Actor在远程系统监督层次结构中的绝对路径 ，包括系统的监护者(即 `/user` 还有其他如 `/system` 承载日志记录器， `/temp` 保存使用 `ask()` 创建的临时Actor引用， `/remote` 启用远程部署等)；这符合该Actor在远程主机上打印它自身的引用的方式，例如在日志输出。

为什么没有收到远程Actor的答复？

最常见的原因是本地系统名称 （即上述答案中的 `system@host:1234` 部分） 是从远程系统的网络位置不可以到达的，例如因为 `host` 被配置为为 `0.0.0.0` ， `localhost` 或 NAT’ed IP 地址。

消息如何可靠传递？

一般的规则是至多一次交付**(at-most-once delivery)**，即没有保证。可以在其上面建立更强的可靠性，Akka提供了工具来这样做。

详见消息传递可靠性。

调试

如何打开调试日志记录

若要打开actor系统调试日志记录，添加如下配置：

```
1.  akka.loglevel = DEBUG
```

若要打开不同类型的调试日志记录，添加如下配置：

- `akka.actor.debug.receive` 将记录发送到一个actor的所有消息，如果actor `receive`方法是 `LoggingReceive`
- `akka.actor.debug.autoreceive` 将记录发送给所有actor的所有特殊消息如`Kill`， `PoisonPill`等
- `akka.actor.debug.lifecycle` 将记录所有actor的所有actor生命周期事件

更多内容参见日志记录 和 追踪Actor调用 。

图书

图书

- Akka in Action, by Raymond Roestenburg and Rob Bakker, Manning Publications Co., ISBN: 9781617291012, est fall 2013
- Akka Concurrency, by Derek Wyatt, artima developer, ISBN: 0981531660, est April 2013
- Akka Essentials, by Munish K. Gupta, PACKT Publishing, ISBN: 1849518289, October 2012

其他语言绑定

其他语言绑定

JRuby

Read more here: https://github.com/iconara/mikka.

Groovy/Groovy++

Read more here: https://gist.github.com/620439.

Clojure

Read more here: http://blog.darevay.com/2011/06/clojure-and-akka-a-match-made-in/.

Akka与OSGi

Akka与OSGi

Configuring the OSGi Framework

To use Akka in an OSGi environment, the `org.osgi.framework.bootdelegation` property must be set to always delegate the `sun.misc` package to the boot classloader instead of resolving it through the normal OSGi class space.

Activator

To bootstrap Akka inside an OSGi environment, you can use the `akka.osgi.ActorSystemActivator` class to conveniently set up the ActorSystem.

```
1. import akka.actor.{ Props, ActorSystem }
2. import org.osgi.framework.BundleContext
3. import akka.osgi.ActorSystemActivator
4.
5. class Activator extends ActorSystemActivator {
6.
7.   def configure(context: BundleContext, system: ActorSystem) {
```

```
    // optionally register the ActorSystem in the OSGi Service Registry
    registerService(context, system)

    val someActor = system.actorOf(Props[SomeActor], name = "someName")
    someActor ! SomeMessage
  }

}
```

The `ActorSystemActivator` creates the actor system with a class loader that finds resources (`reference.conf` files) and classes from the application bundle and all transitive dependencies.

The `ActorSystemActivator` class is included in the `akka-osgi` artifact::

```
<dependency>
  <groupId>com.typesafe.akka</groupId>
  <artifactId>akka-osgi_@binVersion@</artifactId>
  <version>@version@</version>
</dependency>
```

Sample

A complete sample project is provided in akka-sample-osgi-dining-hakkers.

部分HTTP框架名单

部分HTTP框架名单

Play

The Play framework is built using Akka, and is well suited for building both full web applications as well as REST services.

Spray

The Spray toolkit is built using Akka, and is a minimalistic HTTP/REST layer.

Akka Mist

If you are using Akka Mist (Akka's old HTTP/REST module) with Akka 1.x and wish to upgrade to 2.x there is now a port of Akka Mist to Akka 2.x. You can find it here.

Other Alternatives

There are a bunch of other alternatives for using Akka with HTTP/REST. You can find some of them among the Community Projects.