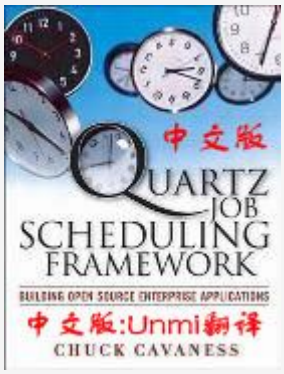
写在最前面的



Quartz Job Scheduling Framework

中　文　版

Chuck Cavaness 著　　Unmi(隔叶黄莺) 译

译者博客：http://unmi.cc

下载地址：Quartz Job Scheduling Framework 中文版.chm

写在最前面的

以此篇权当译者序。再次翻看一下我的博客，最早的那篇《Quartz Job Scheduling Framwork》翻译是在 2007-10-17 02:17 ，距离今日那是一个造人的时间，足见这个翻译过程有多难产。一方面是源于语言水平，再者其间也发生了许多事情。

Quartz 是什么，大概不需多加说明，简单讲就是一个纯 Java 实现的作业调度工具，相当于数据库中的 Job、Windows 的计划任务、Unix/Linux 下的 Cron，但 Quartz 可以把排程控制的更精细。也许大多数人听说 Quartz 是在学习或使用 Spring 的时候，也就是 Spring 整合了 Quartz。而我也不例外，同样是在春天里得悉了这块石英。

当初公司一个项目中有些定时候任务，原来是用 JDK 的 TaskTimer 来实现的，个人觉得比较笨拙——未能跳开节假日，也不能依赖于其他的业务操作灵活安排计划。于是想起了 Spring 框架整合了一个作业调度框架 Quartz，其实一直没用过它，而且这回是要脱离 Spring 框架来使用 Quartz。于是就到网上找来 Quartz 相关的资料，介绍使用倒也不含糊，但缺乏系统与深度，继而找到的就是这本英文版的《Quartz Job Scheduling Framework》。

起初对它的翻译不过是一时之兴——反正都是阅读，何妨用文字再次转录下来呢？后来发现对英文的阅读和翻译全然不是那么一回事，有太多的磕磕碰碰，那是对两种语言的双重考验。阅读可以是眼观六路，一知半解的，完全转换成中文就要字句斟酌，有时候还得为音韵的抑扬顿挫考究了起来。

数篇之后，确实觉得很耗时间且无用功太多，但又不想虎头蛇尾，感情上也无法割舍，加之有各位同仁网友的鼓励与期盼，使我紧持了下来。到目前为止，该书除前言部分、配置参考及附录未翻译之外，主体内容的翻译已大功告成。也算是基本了却了一桩心愿，当然对于落下的几小部分还会补充进来。

现把本人博客上零零散散，顺序错乱的各篇译章汇集成册，做成了一个 《Quartz Job Scheduling Framework 中文版.chm 》文件与各位分享，以方便大家的阅读。同时感谢大家一路来的支持，也希望大家能提出宝贵意见，或有问题拿出来共同探讨。另外，每篇译章都有到达我的博客上所对应页面的链接，所以有疑问可点击页面上方的 我要评论 按钮进入到博客对应页面来***对本篇进行评论，或阅读他人的相关评论***。

说明一下的是，由于本 CHM 文件是直接通过博客上相应日志来自动生成的，所以博客上日志内容有修改也能很快反映到《Quartz Job Scheduling Framework 中文版.chm》中，弊端是未能与原版 CHM 文件的章节目录保持一致，优点是能及时让 CHM 文件与博客日志的内容保持同步，也方便于阅读时就某一篇章共同探讨。

现如今虽有电脑自动全文翻译，而且比较智能化了，为何还去人工翻译呢？电脑总缺少人性化，许多地方无法到位，二来也为了锻练自己。实际上，在对 《Quartz Job Scheduling Framework》的翻译过程所能获得的好处也是不言而喻的。主要表现在两方面：

1. 对技术把握的更精细。阅读是放眼而瞟，只求个大概；翻译则不同，本身未能理解个相当，何以能用中文向他人解译的清楚呢？不得蕴责任于其中。对于多数例子，并非照搬了事，都有再次测试感受过的。译章置于网上之后，亦有许多朋友就 Quartz 提出疑问，毕竟文字出自我手，也就当仁不让的尽我能作出解答，也非常有助于自身对该项技术的掌握。

2. 阅读与翻译的速度提升也是显而易见的。最初时的每字每句的爬梳，须频繁请求各方资源才能完成一篇，现在与那时相比，可谓顺畅多了。许多篇章纵使离开英文词典也无碍了。以后的前行中需要面对更多的英文资料，通过对 Quartz 这个手册翻译算是好好锤炼了自己的英文阅读能力，写作能力亦在其内。

如果，除了诸位同仁网友的鼓励与期盼要感谢之外，以及他们在仔细品读后发现的许多错误之处，才得已使此译者更趋完美，同时非常感激他们阅读每一篇章的态度，这些可以在博客上的评论看得到；那么还要感谢的就是那些飘洒着过早离我而去的头发，曾经多少个夜晚，是我让你们迟迟不能进入到本该属于你们的色彩当中去，所以才不得不选择舍我而去。

中文版目录总汇及内容提要

1. 第一章. 企业应用中的作业调度
   内容提要：什么是作业调度，作业调度为什么说是重要的，企业应用中的作业调度，非企业应用中的作业调度，作业调度与工作流，关于作业调度其他可选择方案
2. 第二章. Quartz 起步
   内容提要：本章对 Quartz 框架一个快速的入门介绍，同时也大略指导你从哪里下载，构建和安装这个框架
3. 第三章. Hello Quartz (第一部分)
   内容提要：建立 Hello Quartz 工程，并创建一个 Quartz Job 类 ScanDirectoryJob.
4. 第三章. Hello Quartz (第二部分)
   内容提要：创建一个 Quartz Scheduler，关联上一个 Quartz Trigger 以编程方式调度前面编写的 ScanDirectoryJob 运行。
5. 第三章. Hello Quartz (第三部分)
   内容提要：通过配置 quartz.properties、quartz_jobs.xml 以声明的方式调度 ScanDirectoryJob 运行。
6. 第三章. Hello Quartz (第四部分)
   内容提要：让我们最后简单讨论打包一个用到了 Quarts 框架的应用程序的流程，需要依赖于哪些包，也以此来结束本章的内容。
7. 第四章. 部署 Job (第一部分)
   内容提要：介绍 Scheduler 和 SchedulerFactory 有哪些类型、SchedulerFactory 的关键 API 方法；以及如何通过 java.util.Properties 实例或默认 quartz.properties 文件创建 Scheduler。
8. 第四章. 部署 Job (第二部分)
   内容提要：如何管理 Scheduler(启动、停止、Standby 模式)。还介绍了 Job、JobExecutionContext、JobDetail、JobDataMap，及如何访问 JobDataMap 中的数据。有状态和无状态的 Job。
9. 第四章. 部署 Job (第三部分)
   内容提要：Job 的易失性、持久性和可恢复性，如何从 Scheduler 中移除、中断 Job。Quartz 已为我们提供了哪些 Job。最后是 Java 线程的简单介绍。
10. 第四章. 部署 Job (第四部分)
    内容提要：线程在 Quartz 中的用法，主处理线程：QuartzSchedulerThread 和 Quartz 工作者线程。Quartz Trigger 和 Calendar 各有哪些类型和如何使用。
11. 第五章. Cron 触发器及相关内容 (第一部分)
    内容提要：引入 Quartz CronTrigger 及简单使用 CronTrigger 来部署一个 Job
12. 第五章. Cron 触发器及相关内容 (第二部分)
    内容提要：详细介绍了 cron 表达式的格式和像 , - * ? / L W C # 特殊符号的使用
13. 第五章. Cron 触发器及相关内容 (第三部分)
    内容提要：CronTrigger 使用起(startTime) 迄(endTime) 日期的使用。TriggerUtils 简单方便的创建 Trigger。应用 JobInitializationPlugin 在 quartz_jobs.xml 配置文件中写 Cron 表达式。
14. 第五章. Cron 触发器及相关内容 (第四部分)
    内容提要：Cron 表达式 Cookbook，列举了各种 Cron 表达式的写法和意义，有助于更好的理解 Cron 表达式；还用了 TriggerUtils 创建了一个即刻触发的 Trigger。
15. 第六章. Job 存储和持久化 (第一部分)
    内容提要：介绍 Quartz 中的 Job 存储，JobStore 接口相关 API 方法。使用 RAMJobStore 来实现 Job 存储及它的优缺点。
16. 第六章. Job 存储和持久化 (第二部分)
    内容提要：使用持久性的 JobStore，可用类型 JobStoreTX 和 JobStoreCMT。持久性 JobStore 是通过数据库来完成的，哪可支持哪些数据及需要创建些什么表。
17. 第六章. Job 存储和持久化 (第三部分)
    内容提要：使用和配置 JobStoreTX，需要为不同数据库平台指定不同的驱动代理(DriverDelegate)，和

quartz.properties 中与 JobStoreTX 相关配置说明。

18. 第六章. Job 存储和持久化（第四部分）
内容提要：为 JobStoreTX 通过在 quartz.properties 配置来创建数据源，并在 Scheduler 中使用数据源

19. 第六章. Job 存储和持久化（第五部分）
内容提要：从数据库中加载 Job 等信息；配置和使用 JobStoreCMT；详细说明了在 quartz.propertis 中关于 JobStoreCMT 的配置属性。

20. 第六章. Job 存储和持久化（第六部分）
内容提要：为 JobStoreCMT 数据源；和 JobStoreTX 有所不同，需要为 JobStoreCMT 配置两个数据源，一个是不爱管理的，另一个是受容器管理的数据源。

21. 第六章. Job 存储和持久化（第七部分）
内容提要：有关改善 JobStore 性能的讨论--主要是 JDBC JobStore；还有如何创建自定义的 JobStore--需实现 40 个接口方法。

22. 第七章. 实现 Quartz 监听器（第一部分）
内容提要：简单介绍了监听器是 Quartz 框架的一个扩展点，实现一个监听器的基本步骤，最后说明了全局监听器和非全局监听器的区别。

23. 第七章. 实现 Quartz 监听器（第二部分）
内容提要：JobListener (Job 监听器) 的介绍和使用方法，代码演示了如何注册了全局 Job 监听器和非全局 Job 监听器。

24. 第七章. 实现 Quartz 监听器（第三部分）
内容提要：TriggerListener (Trigger 监听器) 的介绍和使用方法，代码演示了如何注册全局 Trigger 监听器和非全局 Trigger 监听器。

25. 第七章. 实现 Quartz 监听器（第四部分）
内容提要：SchedulerListener (Scheduler 监听器) 的介绍和使用方法，多是关于对 Scheduler 管理事件的监听，而不只专注于 Job 或 Trigger 的。

26. 第七章. 实现 Quartz 监听器（第五部分）
内容提要：Quartz 专门提供了一个与 FileScanJob 一同使用的 FileScanListener，用于监视文件 lastModifiedDate 的改变。

27. 第七章. 实现 Quartz 监听器（第六部分）
内容提要：除前面用编程式使用监听器外，Quartz 还支持在 quartz_jobs.xml 中以声明式使用监听器。

28. 第七章. 实现 Quartz 监听器（第七部分）
内容提要：调用监听方法的线程，按什么顺序调用监听方法，最后列了一些能够把监听器应用到什么地方。

29. 第八章. 使用 Quartz 插件（第一部分）
内容提要：开始介绍 Quartz 插件，以及它要实现的接口和其中的三个接口方法的使用时机和用途。

30. 第八章. 使用 Quartz 插件（第二部分）
内容提要：讲述如何创建 Quartz 插件，并用一个从指定目录中加载所有 Job 文件的 Quart 插件作为例子来说明。

31. 第八章. 使用 Quartz 插件（第三部分）
内容提要：插件类写好了，就是关于如于注册插件的话题了。本节对此详细讲解，并以前面写的插件例子进行示范。

32. 第八章. 使用 Quartz 插件（第四部分）
内容提要：Quartz 从属性文件中加载多个插件类时不能保证加载的顺序，所以本节引入一个自定义的统一按顺序加载其他插件的，名之为插件加载器的东西，其实也就是其他插件类的父亲。

33. 第八章. 使用 Quartz 插件（第五部分）
内容提要：Quartz 还为我们提供了几个开箱即用的工具插件：JobInitializationPlugin,JobInitializationPluginMulitple,LogginJobHistoryPlugin,LoggingTriggerHistoryPlugin,ShutdownHookPlugin。

34. 第九章. 使用 Quartz 的远程方式（第一部分）
内容提要：不在同一地址空间的 Quartz 需要一种远程管理的管理，Quartz 选用了 RMI。本部分主要是简单介绍了 RMI 技术。

35. 第九章. 使用 Quartz 的远程方式（第二部分）
内容提要：配置、创建并运行 Quartz RMI 服务端。

36. 第九章. 使用 Quartz 的远程方式 (第三部分)
内容提要：配置、创建并运行 Quartz RMI 端户端，演示了 Quartz RMI 客户端通过远程调度器部署一个 Job 的 的例子。

37. 第十章. J2EE 中使用 Quartz (第一部分)
内容提要：J2EE 中引入 Quartz。在 J2EE 环境中作为 J2SE 客户端运行 Quartz。演示了一个 Quartz 为我们提供的 EJBInvokerJob 的例子。

38. 第十章. J2EE 中使用 Quartz (第二部分)
内容提要：借助于 QuartzInitializerServlet 或 QuartzInitializerListener 在 J2EE 容器上运行 Quartz，并使用容器的相关资源。

39. 第十一章. Quartz 集群 (第一部分)
内容提要：Quartz 应用也能进行集群。及 Quartz 集群能提供高可用性、伸缩性、进行负载均衡。

40. 第十一章. Quartz 集群 (第二部分)
内容提要：介绍集群中的 Quartz 应用是如何工作的。集群中的 Quartz 应用是通过中心数据库来感知道其他节点的存在。

41. 第十一章. Quartz 集群 (第三部分)
内容提要：如何配置使 Quartz 节点工作在集群环境中。

42. 第十一章. Quartz 集群 (第四部分)
内容提要：运行 Quartz 集群节点，及提供了一个关于使用 Quartz 集群的 Cookbook 参考。

43. 第十二章. Quartz Cookbook (第一部分)
内容提要：Scheduler 相关的 Cookbook，Scheduler 的创建、启动、停止、暂停。

44. 第十二章. Quartz Cookbook (第二部分)
内容提要：Job 相关的 Cookbook，Job 的创建、部署。和如何用 TriggerUtils 创建一个只需触发一次的 Job。

45. 第十二章. Quartz Cookbook (第三部分)
内容提要：如何替换、更新已部署的 Job。更新已存在的 Trigger。如何列示出 Scheduler 中的所有 Job 和 Trigger。

46. 第十三章. Quartz 和 Web 应用 (第一部分)
内容提要：Web 应用中引入 Quartz 及如何集成。

47. 第十三章. Quartz 和 Web 应用 (第二部分)
内容提要：在 Struts 框架中使用 Quartz，虚构了一个叫做 Job 管理控制台的 Web 应用，。需要在 Web 应用启动时创建一个 SchedulerFactory。

48. 第十三章. Quartz 和 Web 应用 (第三部分)
内容提要：所幸 QuartzInitializerServlet 帮了我们大忙，介绍了 QuartzInitializerServlet 在 web.xml 中的配置，指定属性文件及应用启动时 Scheduler 是否启动等。Quartz 还为我们提供了 ActionUtil 类方便了访问 SchedulerFactory 和 Scheduler。

49. 第十三章. Quartz 和 Web 应用 (第四部分)
内容提要：还有，别忘了我们还有一个 QuartzServletContextListener 可选择的，从 2.3 版本的 Servlet API 开始就可以用这个。

50. 第十三章. Quartz 和 Web 应用 (第五部分)
内容提要：最后介绍了 Quartz 官方的一个 Quartz Web 应用程序，它是以 Velocity 作为视图实现的。

51. 第十四章. 工作流中使用 Quartz (第一部分)
内容提要：可以把 Quartz 引入到工作流中，主要讲了单独用 Quartz 来把 Job 组成 Job 链，模拟成一个酷似工作流的东西。

52. 第十四章. 工作流中使用 Quartz (第二部分)
内容提要：OSWorkflow 工作流快速入门，讲了 OSWorkflow 工作流中各种概念，为它与 Quartz 集成作个铺垫。

53. 第十四章. 工作流中使用 Quartz (第三部分)
内容提要：讲了如何把 OSWorkflow 与 Quartz 进行集成，做了使用 Action 自定义函数的例子。

54. 第十四章. 工作流中使用 Quartz (第四部分)
内容提要：用 Workflow Job 来启动、运行工作流，通 JobDataMap 来传递工作流名称，transientVars 等。

55. 附录 A. Quartz 配置参考 (第一部分)
内容提要：一些主要的、公共的 Quartz 属性配置参考。

56. 附录 A. Quartz 配置参考（第二部分）
内容提要：线程池、各种监听器和插件的配置参考。

57. 附录 A. Quartz 配置参考（第三部分）
内容提要：介绍 Quartz RMI 的相关配置，及引入 JobStore 选项。

58. 附录 A. Quartz 配置参考（第四部分）
内容提要：Quartz JobStore 中的 JobStoreTX 的配置参考。

59. 附录 A. Quartz 配置参考（第五部分）
内容提要：Quartz JobStore 中的 JobStoreCMT 的配置参考。

60. 附录 A. Quartz 配置参考（第六部分）
内容提要：关于数据源的配置参考。

第一章. 企业应用中的作业调度

第一章. 企业应用中的作业调度

- ·什么是作业调度
- ·作业调度为什么说是重要的
- ·企业应用中的作业调度
- ·非企业应用中的作业调度
- ·作业调度与工作流
- ·关于作业调度其他可选择方案

1. 什么是作业调度

“作业”，这一技术述语上的概念，又让我们回到了大型机的年代，那时候，用户/程序员提交一叠穿孔卡片或者纸带（上面描述了一个作业）给操作人员，由操作人员帮忙执行那些作业。用户等待作业执行完后，回到主机那边取自己的卡片和打印出来的输出结果。

因为不是每一个作业要求立即被执行，所以作业可以被安排在将来的某个时候执行。比如说，一个系统管理员每天晚上可能有一份要执行的作业列表：

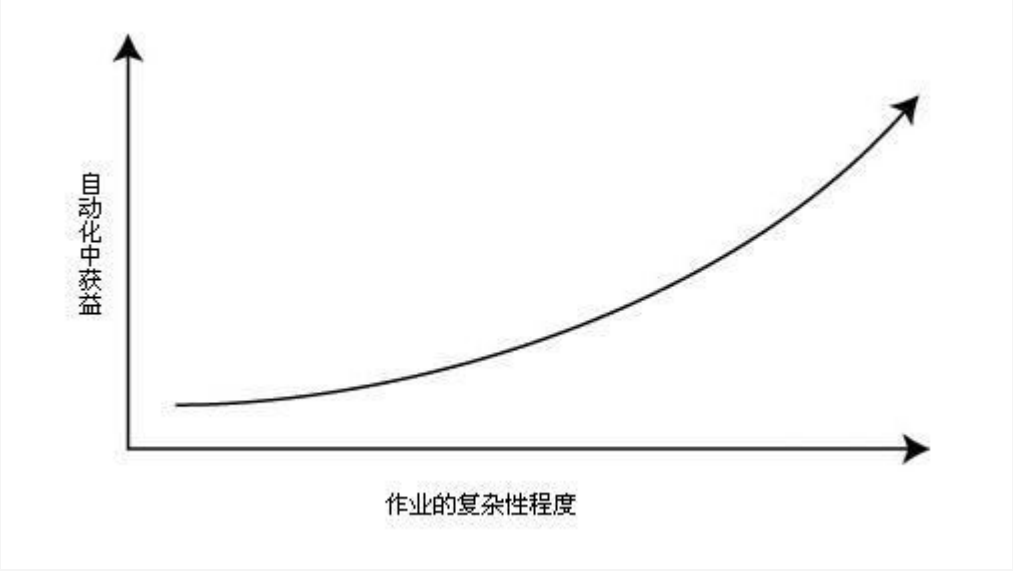
- · 10:00 PM: 运行患者信息文件的上载作业
- · 11:00 PM: 运行销售数据报表生成
- · 11:59 PM: 进行数据库的备份

作业调度通常是指运行一个批量的作业或称之谓批处理。这种批处理作业一般都是放在后台运行并且不需要与用户交互。现在，显著增多的多样性的任务已代替了早先的批量作业。在一个大的组织中每天的每小时跑上百个作业已属普遍。并且作业的规模与复杂性仍在持续的上扬，因此批量作业和作业调度器也就随需应生。

2. 作业调度为什么说是重要的

俗话说，“时间就是金钱。”，过高的资源投入到枯燥的任务中无疑是金钱和资源的浪费。随着业务流程复杂性的提升，自动化流程也更能显现出它的有益之处来。图 1.1 说明了这一问题。

图1.1 任务规规模越大、越复杂、越频繁的被执行，那么能从自动化中获益也越大



人之所以称之为人，国为我们犯错误的频度远高于电脑。把一系列任务自动安置到一个作业中，然后再为这个作业创建一个调度器，到时这个作业就会自动执行了。相对于人的手工处理，我们可以减少大多数的出错机会。

作业调度器的另一个优点体现在伸缩性。我们也许能一个小时中手工完成10或20个作业，但是随着每小时处理作业数量的增加，我们就更难杜绝不在作业中引入错误。但如果借助于作业调度，只会受到硬件资源的影响了。

所以我们能着实的说通过作业调度那样自动化处理相对于手工来说，至少为我们提供了以下三个优点：

·资源使用效率更高

·更少的出错机率

·更高的伸缩性

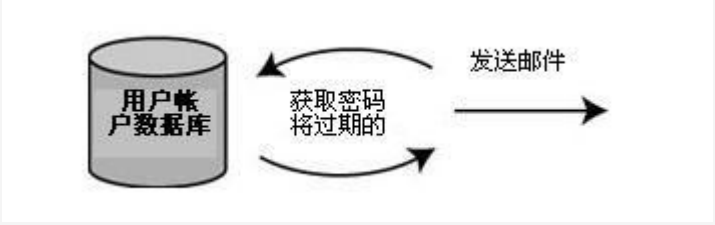
3. 企业应用中的作业调度

“企业应用”一词，如今经常会被我们无意识间提及，然而似乎现在还没有对它一个准确的定义。但对于在本书中这一词的意义，我们只要建立起这样一个概念：作为某一组织的一部份而存在的软件系统或程序。这个系统可以是一个大型机上的、或者是一个C/S结构的、或者就是一个J2EE应用。真实世界中的例子就是，作业调度器能在那个系统上大量的使用。以下的几个场景，尽管没有详尽的进行描绘，也涉及到了现今应用软件常常遇到的场景。

场景 #1: 邮件提醒和告警

许多网站（不管是商业的还是别的）允许用户提供用户名和密码注册一个帐户。出于安全考虑，一个好的做法是让用户密码每隔一段时间过期失效，比如说90天的周期。这种情况下，你可以创建一个作业，让它每天午夜运行一次，并且向离过期时间不到三天的所有用户发邮件提醒。这里可以恰到好处的用到作业调度器。图 1.2 描绘了密码这个提醒作业。

图 1.2 密码过期的作业每晚发送邮件给密码很快会过期的用户



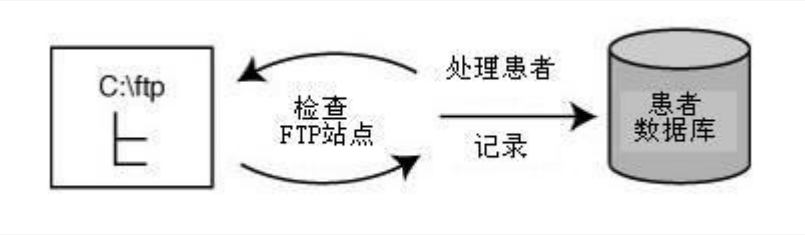
除了发过期的密码信息，网站还可以发送其他的告警或提醒（可不是垃圾邮件哦）。一个作业调度器还能够用在类似的其他方面。

场景 #2: 执行文件传输操作

许多商家需要和他们的供应商或客户作信息集成。一种集成的方式就是进行数据文件的交换。可以采用实时的方式，例如SOAP协议，但是许多时候却不需要实时性，代之以异步的方式，譬如用FTP协议来发出或取所要的文件。

下图描绘了一个劳工补偿局每天早上收到一些包含患者及事故信息的文件。公司可以雇一个人每天早上手工的检出FTP服务器上的文件。作为另一个更好的选择就是可以写一个作业，让它每天早上扫描FTP服务器，如果有文件的话，把文件内容处理后插入到患者数据库中去。让作业调度器代劳后，这个职员再也不用手工去上FTP检查文件，而可以为公司做更多别的更有意义的事情。图 1.3 描绘了文件传输的操作。

图 1.3. 文件传输的作业检查FTP站点，把患者信息文件处理到数据库中。



场景 #3: 创建销售报表

公司经营由盈亏账目所驱动，其中一个很重要的事情就是经营管理者和财务人员需要拿到最终收入和毛利数据进行分析。抽取销售报表数据可能非常的慢并且很耗资源，因为这通常需要联合多个表从中查询出上千条记录。一个更好的解决途径是在晚上计帐和计价结束后，运行一个作业，让它去生成一些临时表或视图为报表程序所用。创建临时表或视图的方式，使报表生成更具动态特性，而且用户也用不着平白去等待报表的生成，一些报表工具，如水晶报表 XI(Crystal Reports XI) 本身就包含了作业调度器（见图 1.4）。

图 1.4. 销售数据报表程序执行为销售团队产生收入和毛利信息

4. 非企业应用中的作业调度

Quartz 对于许多非企业环境的应用也是很有帮助的。例如，假定你有一个独立的应用程序，事件是基于时钟而不是鼠标的点击激发的。这时候你就可以把Quartz构建到这个应用程序中来，并且安排事件能周期性的被触发。

另一例子是，你也许正想查询数据库并发送邮件，而邮件接受者正是基于这些数据得到的。（译者注：真有些搞不明白）

5. 作业调度与工作流

作业调度不是工作流，理解这一点很重的要。它们常被同时应用于一个方案中，但它们是两个截然不同的解决办法，并且都可孤立使用。一个作业通常由几个步骤组成。我们回过头来看前面提到的那个密码过期的作业，实质上它是由三个步骤所组成。

1. 获取到密码将要过期的用户列表
2. 为列表中的用户各自发送一个邮件
3. 更新记录，下次就能知道哪些邮件发送过

这个作业可以使用工作流的优点，作业的每一部份恰好对应着工作流的每一个步骤。这并非意味着离开了工作流，作业调度会有些糟。这是普通的下一步、下一步简单操作。只要作业调度框架与第三方工作流能轻便的解决问题，就是好的。更多的关于Quartz和工作流的内容将会在第十四章，“使用Quartz和工作流”详解。

6. 关于作业调度其他可选择方案

正如你所知，这本书是讲Quartz的，但是Quartz的可替代方案呢？当我们比较作业调度方案的时候肯定要提到别的同类应用，那么现在就来简单介绍一下它们。

Java SDK Timer 和 TimerTask 类

java.util.Timer和java.util.TimerTask这两个类是自1.3版本才加入到JDK中来的。这两个新类可以实现一个最基本的调度器。也就只能作为我们理想的完整调度器框架的一个小的部件。任何严格意义的作业调度器都提供直接指定执行时间，存储作业信息到多种介绍和使用钩子进行定制及其他更多的功能。单纯靠JDK的那两个类还不足以构建一个真正的作业调度器。JAVA的Timer类也没办法对作业和触发器作相应的组织，使用每任务一个线程，而不是线程池的方式，还有其他不足之处难以成全其实现一个完全意义的作业调度器。

本土方案

从前面提到的Timer和TimerTask类来看，我们很容易低估创建一个灵活的、并且日后可扩展的作业调度器所作出的努力。创建一个作业调度器也不视之为微不足道的活儿。它需要的不仅仅是Java线程方面的专业技术，还需要解决其他更复杂的课题。如果你没有这方面成熟的专业知识，别想着作业调度器能直接随意一份草稿能一蹴而就。

商业解决方案

现在市面上也能见到不少商业的作业调度产品。在本书中，我们不打算花功夫去了解和评估那些商业化产品。表1.1 中列出了当下流行的几个解决方案，你可以通过所给相应的URL获得更详尽的信息。

表 1.1 商业作业调度器

名称	网址
Flux Scheduler	www.fluxcorp.com/
Enterprise Batching Queuing	www.argent.com/p/qe/qe.html
Unicenter AutoSys Job Management 4.5	www.ca.com
BMC Software ControlM	www.bmc.com
Cybermation ESP Espresso 4.2	www.cybermation.corly;9'm
Vexus consulting Avatar Job Scheduling Suite 4.5.5	www.vexus.ca
Argent software The Argent Job Scheduler 4.5A	www.argent.com
Tidal Enterprise Scheduler	www.tidalsoftware.com

第二章. Quartz 起步

第二章. Quartz 起步

本章对 Quartz 框架一个快速的入门介绍，同时也大略指导你从哪里下载，构建和安装这个框架

1. Quartz 框架的发展历程

和现今许多在用的开源项目一样，Quartz之初也只是为个人开发者提供了一个简单的实现方案。但是随着日益增多的关键人员的积极参与和慷慨的贡献，Quartz 已经成为了一个为众人所知，并且能帮助人们解决更大问题的框架。

Quartz 项目 是由 James House 创立的，它在1998年就有该框架最初的构思。包括作业队列的概念，使用线程池来处理作业，也许它最早的模型已不为现今的Quartz使用者所知了。

在接下来的数年中，House 自己说他一直在关注着同一个需求：需要一个灵活的作业调度工具。他在找寻便宜且具有丰富特征的Java作业调度工具时，让他面临着以下几个选择：

·一个昂贵的商业化工具

·嵌入在大框架之中的，根本用不着这么一个大框架

·类似 Unix Cron 或者 Windows 的计划任务

·自己亲自定制的方案

House 有限的选择和在这个问题上的兴趣促成了他为作业调度器创建一个开源的项目。在2001年春天，他在 SourceForge 上创立了该项目，这一网址 http://sourceforge.net/projects/quartz 现在还是有效的，只是已经不再维护了。

自从 Quartz 的雏形一出来，众多的捐助者和开发人员加入到这个项目中来。然而应该说，Quartz 能象今天这么存在还是要感谢 House 以及他在作业调度领域中的兴趣。在众多人眼中，他那解决问题的决心很值得称颂的。

2. 下载和安装 Quartz

在 Quartz 的主页面 http://www.opensymphony.com/quartz 中有下载链接（由 OpenSymphony 提供的主机服务）。在这里你可获取到最新版，也有几个早期版本供下载。Quartz 下过来是一个完整的发行版，其中包括源代码和已编译好可直接使用的 JAR 文件。Quartz 的 JAR 包还存在于 ibiblio (译者注：http://www.ibiblio.org/maven/) maven 仓库中，很方便于你用 Maven (译者注：一个比ANT更为强大的构建工具) 或者 Ivy(译者注：一个免费基于Java的依赖管理器) 来构建系统。

下载到的是一个 ZIP 格式文件，因此你需要一个像 WINZIP 那样的工具，你还可以用 Java 的 jar 命令来解压缩该文件：

```
jar -xvf quartz-1.5.0-rc1.zip
```

Quartz 发行包中的文件将会解压到当前目录中。

解开来的 Quartz zip 文件包含以下几个子目录。表 2.1 描述了每一个子目录的内容。

表 2.1 Quartz 的目录结构和内容

目录名	存放内容
Docs	
docs/api	Quartz 框架的JavaDoc Api 说明文档
docs/dbTables	创建 Quartz 的数据库对象的脚本
docs/wikidocs	Quartz 的帮助文件，点击 index.html 开始查看
Examples	多方面使用 Quartz 的例子
Lib	Quartz 使用到的第三方包

src/java/org/quartz	使用 Quartz 的客户端程序源代码，公有 API
src/java/org/quartz/core	使用 Quartz 的服务端程序源代码，私有 API
src/java/org/quartz/simpl	Quartz 提供的不衣赖于第三方产品的简单实现
src/java/org/quartz/impl	依赖于第三方产品的支持模块的实现
src/java/org/quartz/utils	整个框架要用到的辅助类和工具组件
src/jboss	提供了特定于 JBoss 特性的源代码
src/oracle	提供了特定于 Oracle 特性的源代码
src/weblogic	提供了特定于 WebLogic 特性的源代码

·安装必要的 JAR 文件

如果你急于想让 Quartz 工作起来，那么最快捷的方法是获取到已编译打包好的 Quartz JAR 文件（它存在于解压后的根目录下），并把它加到你的应用程序的 Classpath 上，你还需要获取到 Quartz 所依赖的包。表 2.2 列出了要创建一个 Quartz 应用最基本的包。

Quartz 基本应用所需的 JAR 包

名称	位置	备注
Commons BeanUtils	<quartz-download>/ lib/optional	依赖于怎么使用 Quartz, 最好是包含进来
Commons Collections	<quartz-download>/ lib/core	需要
Commons Digester	<quartz-download>/ lib/optional	依赖于怎么使用 Quartz, 最好是包含进来
Commons Logging	<quartz-download>/ lib/core	需要

就像使用 Quartz JAR 包一样，你同样需要把所依赖的包加到应用程序的 Classpath 中。

> **当心版本冲突**
>
> Quartz 同特定版本的第三方包构建并作过测试。许多其他的项目，包括一些非常知名的应用服务器也使用着这些第三方库，在某些情况下，这些库已然成了应用服务器的组成部份。类加载器是一个很神奇的东西。假如你是在应用服务器环境中使用 Quartz，小心不要用到了重复的库，不然你可能会得到奇怪的执行结果。表 2.1 中的包如果它们已存在于应用服务器中，你的程序应该能运行的很好。然而，像 servlet.jar 和 ejb.jar 这样的包要是重复出现在 classpath 中，恐怕会给你带来麻烦。这个时候你可以试着不把这类包加到 classpath 中，看看程序的运行表现。

·quartz.properties 文件

Quartz 包括一个名为 quartz.properties 的配置文件，它允许你对 Quartz 的很多方面的配置。在 Quartz JAR 包中有一个默认的 quartz.properties 文件，但是假如你需要修改任何默认配置项时，你需要放置一个 quartz.properties 文件持贝在 classpath 下。

下一章将详细描述 quartz.properties 中哪些选项可配置和如何配置。你将有极大的可能性要去修改其中的一或多项设置，因此你应该拷贝一份 quartz.properties 文件到你的 classpath 下。

3. 从源代码构建 Quartz

下载到的Quartz包括源代码和可部署的JAR文件。有了源代码的好处之一是你可以深入去理解它是如何实现以及实现了什么。阅读源代码方便了想深入研究它的开发人员。如果是用像 Eclipse 那样的 IDE 想要单步跟进到代码中，那么源代码肯定是少不了的。(译者注：怎么感觉原文挺啰嗦的)

·Quartz CVS 仓库

Quartz 的 CVS 仓库和别的 OpenSymphony 项目一块，都是着落在 Java.net 主机上。想要从 CVS 仓库中下载任何东西的话，你必须要有一个 CVS 帐号。当然，你也能在 xwork.dev.java.net/source/browse/quartz 上以匿名的方式浏览源代码，但是要下载的话就必须要有 CVS 帐号。

> **创建一个 Java.net 帐号**
>
> 你可以登录到网址：http://www.dev.java.net/servlets/Join 注册一个免费的 Java.net 帐

号。使用申请的帐号不仅能存取 Quartz CVS 仓库中的代码，而且还能够在这个站点上找到大量的有用的信息和技巧（见 http://www.java.net）。

创建好了帐号之后，你可以下载所需的文件用来构建 Quartz. 大致步骤是，打开命令行，进入到你希望工程存放的目录。通过 CVS 取代码后会在当前目录中创建两个子目录，quartz 和 opensymphony.

```
cvs -d :pserver:[username]@cvs.dev.java.net:/cvs login
cvs -d :pserver:[username]@cvs.dev.java.net:/cvs checkout quartz
```

和

```
cvs -d :pserver:[username]@cvs.dev.java.net:/
cvs checkout opensymphony
```

替换上面的[username]为你自己的帐号（命令中不带中括号）。

下载完这两个模块之后，命令行下进入到 quartz 子目录，并输入：

```
ant -projecthelp
```

该命令会列出所有的 target 和对应的描述。默认的 target 是构建 Quartz JAR 文件的 "jar"；你可以在命令行下只输入 "ant"，将会给你编译打包出一个 Quartz JAR 文件来。

IDE中由源代码进行构建

需要指出的是，几个 Quartz 包依赖于第三方的产品的，比如 JBoss 和 WebLogic. 当使用 Ant 的 build 文件从源代码进行构建时，默认的设置是，找不到第三方的包时会忽略构建那些组件。假如你的IDE中包括了完整的 Quarts 源代码，而又没有第三方产品的包，将会出现编译错误。最简单的方法是不要在IDE中包含那些源码。幸运的是，Quart 源码的层次和结构性很清晰，因此你可以在IDE中选择不包含某些目录/包。

4. 从 Quartz 社区获得帮助

通常评定一个开源项目的一个关键指标是它的用户社区的健康状况。参与到开源项目中完全是自愿的，并且是没有任何回报的，这也是为大家所认同的，用户应该是充满着热花费他们的时间在项目中。因此，自然给人的感觉就是，假如有一个令人满意的社区的话，用户就会相信这个项目的价值。

Quartz 的用户社区相当活跃的。和其他许多开源项目一样，大部份社区成员以匿名的方式查看列表和消息；少部分人发周占贴和回答别人的问题。Quartz 用户社区六个月以来就达到了一个很高的发贴记录，用户论坛收到了超过分属于500个不同主题的1500条消息，和惊人的25000人查看了贴子。这些甚至还不包括开发者列表中的消息。假如你注意到这六个月来的数字变化，你就能看出是一个上升趋势。从下载数量来看，Quartz 平均每月下载量在2000和3000之间。这些数字在有新版本发布时变的更高。

你能在网页 http://forums.opensymphony.com 找到进入用户或开发者论坛的链接，你也能够从 Quartz 的首页面 http://www.opensymphony.com/quartz 进入那两个论坛。也建议你注册加入到邮件列表中；假如感觉不错的话，可以加入到项目中，以任何自己所能的方式做有助于项目的事情。

5. 谁在用 Quartz?

开源软件一个常被人询问的问题是，"谁在使用它?"。如果有人在使用它，人们头脑中的观念就会觉得它一定很好，并且可安全的使用。尽管 Quartz 出来有好长一阵子了，但它最近才得到开发社区应得的关注。

使用了 Quartz 的用户和项目列表许多是我们耳熟能详的。你能在 Quartz Wike 网站 http://wiki.opensymphony.com/display/QRTZ1/Quartz+Users 看到这个列表。Quartz 是作为其中的那些知名开源项目的一部分存在，这些开源项目包括有：JIRA, Spring 和几个 Jakarta 项目。

没有一个准确的方法获知到底有多少用户在实际的项目中使用 Quartz。但是从下载次数以及某些当前流行的项目正在使用 Quartz 可推断出，Quartz 的使用者应该是数以千计。

第二章到此结束，会继续未尽的革命事业。

看别人译作，不时也会指手划脚起来，可真待到自己去把英文转成中文时，才有些体会，有些一眼看过去很明了的原句，换成中文愣是难找到一个合适的词，同时对原句一知半解时，转换起来就要受些阻，强行过去难免要跑意了.....

第三章. Hello Quartz (第一部分)

[译者注：后面的章由于每章的内容较多，每章聚于一篇之中，过于臃肿，屏幕不比书本，三屏之后的内容一般不为读者乐意去阅读，此为第一部分]

第三章：Hello Quartz

多数读者都较容易从一个简明扼要的例子中明白一个东西。作为写作者，要注意避免把一章的内容精简地几乎什么都没了；作为读者呢，需要有耐心并且要进一步相信其后相关的章节应该去阅读，尽管这个例子看起来是如此之简单。

有了这种初衷，这一章将为你介绍如何用 Quartz 框架创建一个简单的应用程序，它展示了一个典型的应用。这个例子将让你领略到创建和执行一个简单应用的必要步骤。通过本章的学习，为你学习本书的后续章节打下了坚实的基础。

1. "Hello, World" Quartz 工程

本示例应用比起众所周知的 System.out.println("Hello world from Quartz") 来还是要有趣些。当我们用 Quartz 执行一个作业时，总是希望它能为我们执行一些有趣且有意义的任务。因此，接下来我们就要做一些有趣且有用的事情。

本章向您演示如何创建这么一个 Quartz 作业，Quartz 应用通知它要做事情的时候，就会去扫描指定的目录寻找 XML 文件。假如在指定目录中找到了一个或多个 XML 文件的话，它将会打印出文件的一些概要信息。是不是很有意义且有趣的，你说呢？但愿，你还能进一步延伸：作业在检测到某一目录下的特定文件后，还要依据那些文件做其他许多你感兴趣的事。你可能会想把它们 FTP 到一台远程机器上，或者把它们作为电子邮件的附件发送。也许那是些客户发过来的订单文件，我们需要读取它们后插入到数据库中。无限可能性；我们会在本书的后面部分讨论它们。

我们努力让这一部分阐述地直截了当并且只涉及本质要义。然而，我们也会研究到一些会影响到 Quartz 应用程序运行行为的常用配置选项。我们假定你很好的掌握了 Java 语言；我们基本不会花时间去解翻译 Java 语言方面东西。

最后，本章的结束部分会简单的讨论怎么打包这个示例应用。在构建和打包 Java 工程时 Apache Ant 是我们的选择；Quartz 应用程序也不例外。

·建立 Quartz 工程

首要步骤是要建立起本工程的开发环境。你可以选择任何自己喜欢的开发工具或者感觉比较好的IDE；Quartz 并不发固执的要求你用哪一个工具。假如你还是接触Java没多久的开发者，Eclipse 会让你感觉特别的舒适；我们在本书的所有例子都是在 Eclipse 中讲解。

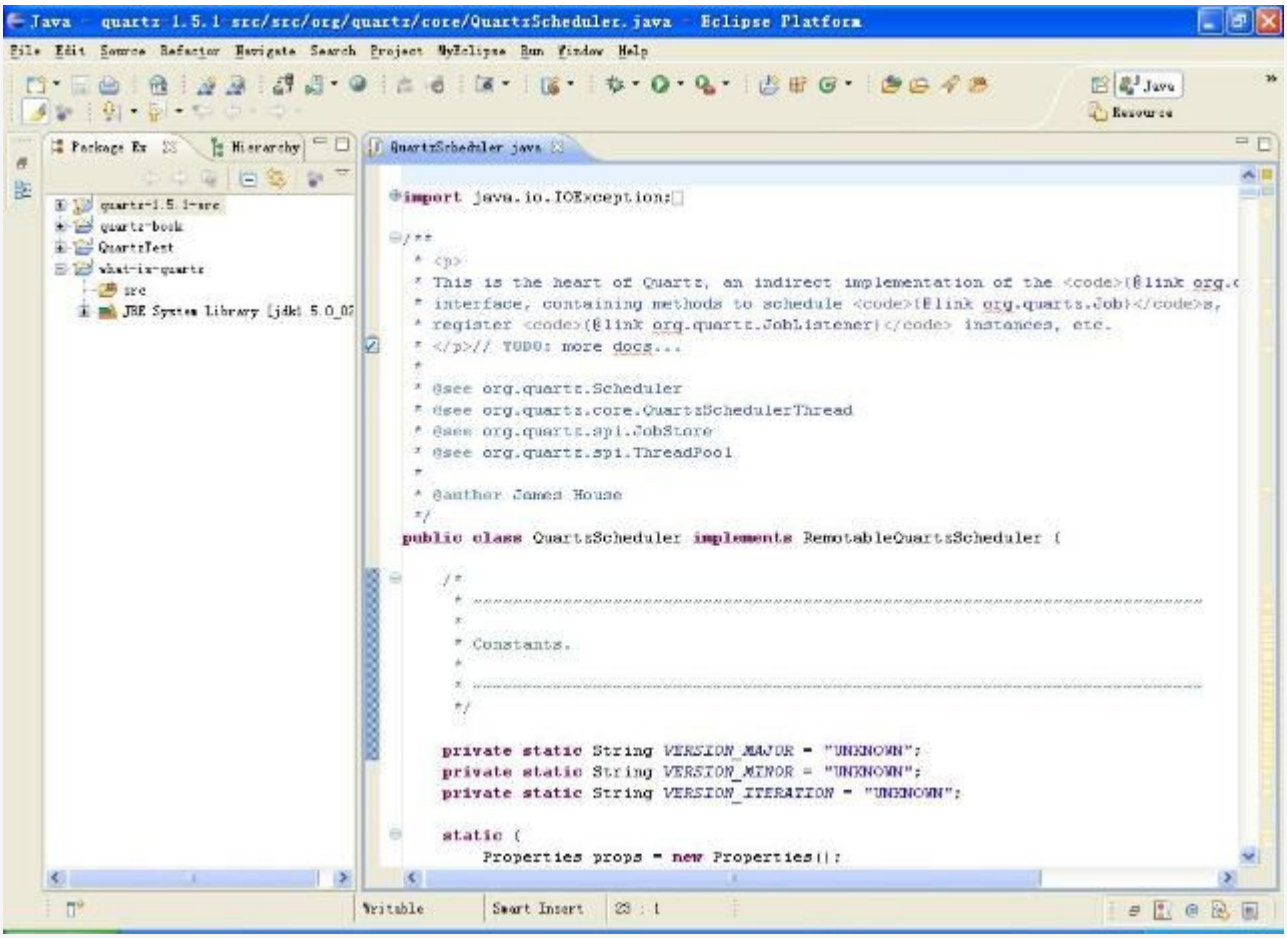
如果你还没有 Eclipse 的话，你可以从 http://eclipse.org 下载。你可以选择下载 3.x 的某个版本。在 http://www.eclipse.org/eclipse/index.html 可查看 Eclipse 的文档；你会找到能帮助你上手 Eclipse 的所有需要的资料。

·在 Eclipse 中配置使用 Quartz

我们只为本书中的所有例子创建一个 Java 工程；每一章的源代被放在单独的目录中。图 3.1 显示了Eclipse 中的 Quartz 工程。

图 3.1 在 Eclipse 中创建一个 Quartz Java 工程

[点击看大图]



你必须引入几个 JAR 到工程中才能成功构建它们。首先，你需要 Quartz 的二进制版本，包的名字是 quartz-<version>.jar。Quartz 还需要几个第三方库；这依赖于你要用到框架的什么功能而定，Commons Digester 库可以在 <QUARTZ_HOME>/lib/core 和 <QUARTZ_HOME>/lib/optional 目录中找到。表 3.1 列出了Quartz 依赖包的更多信息。

把 Quartz 源代码加入到 Eclipse 中来是个很好主意。这可以给你带来两方面的益处。其一，它允许你设置断点并跟入到 Quartz 源代码中。其二，它还帮助你深入浅出的学习这一框架。如果你要探察 Quartz 是怎么工作的，或者有时候为什么不能正常工作，那么这时候你真正需要手边有它的一套源代码。这是商业软件无法给你的便利。

> **Quartz Application 和 Job 的区别**
>
> 我们在这里打断一下，有必要解释这个容易搞混的概念。我们用述语“Quartz Application”来指代任何使用到 Quartz 框架的软件程序。通常一个应用程序中会使用许多库，只要用上了 Quartz，我们认为这就是我们在本书上所讨论的 Quartz Application。另一方面，我们所说的 Quartz Job，是指执行一些作业的特定的 Java 类。正常地，在一个 Quartz Application 中会包括许多不同类型的 Job，每一个作业会对应有一个具体 Java 类。这两个概念不能交换使用。

·创建一个 Quartz Job 类

每一个 Quartz Job 必须有一个实现了 org.quartz.Job 接口的具体类。这个接口仅有一个要你在 Job 中实现的方法，execute()，方法 execute() 的原型如下：

```
public void execute(JobExecutionContext context) throws JobExecutionException;
```

当 Quartz 调度器确定到时间要激发一个 Job 的时候，它就会生成一个 Job 实例，并调用这个实例的 execute() 方法。调度器只管调用 execute() 方法，而不关心执行的结果，除了在作业执行中出问题抛出的 org.quartz.JobExecutionException 异常。

你可以在 execute() 方法中执行你的业务逻辑：例如，也许你会调用其他构造的实例上的方法，发送一个电子邮件、FTP 传一个文件、调用一个 Web 服务、调用一个EJB、执行一个工作流，或者像我们的例子中那样，检查某个特定的目录下是否存在文件。

代码 3.1 是我们的第一个 Quartz job，它被设计来扫描一个目录中的文并显示文件的详细信息。

代码 3.1. ScanDirectoryJob 例子

```
1.  package org.cavaness.quartzbook.chapter3;
2.
3.  import java.io.File;
4.  import java.util.Date;
5.
6.  import org.apache.commons.logging.Log;
7.  import Org.apache.commons.logging.LogFactory;
8.  import org.quartz.Job;
9.  import org.quartz.JobDataMap;
10. import org.quartz.JobDetail;
11. import org.quartz.JobExecutionContext;
12. import org.quartz.JobExecutionException;
13.
14. /**
15.  * <p>
16.  * A simple Quartz job that, once configured, will scan a
17.  * directory and print out details about the files found
18.  * in the directory.
19.  * </p>
20.  * Subdirectories will filtered out by the use of a
21.  * <code>{@link FileExtensionFileFilter}</code>.
22.  *
23.  * @author Chuck Cavaness
24.  * @see java.io.FileFilter
25.  */
26. public class ScanDirectoryJob implements Job {
27.      static Log logger = LogFactory.getLog(ScanDirectoryJob.class);
28.
29.      public void execute(JobExecutionContext context)
30.                throws JobExecutionException {
31.
32.           // Every job has its own job detail
33.           JobDetail jobDetail = context.getJobDetail();
34.
35.           // The name is defined in the job definition
36.           String jobName = jobDetail.getName();
37.
38.           // Log the time the job started
39.           logger.info(jobName + " fired at " + new Date());
40.
41.           // The directory to scan is stored in the job map
42.           JobDataMap dataMap = jobDetail.getJobDataMap();
43.                     String dirName = dataMap.getString("SCAN_DIR");
44.
45.           // Validate the required input
46.           if (dirName == null) {
47.                throw new JobExecutionException( "Directory not configured" );
48.           }
49.
50.           // Make sure the directory exists
51.           File dir = new File(dirName);
52.           if (!dir.exists()) {
53.            throw new JobExecutionException( "Invalid Dir "+ dirName);
54.           }
55.
56.           // Use FileFilter to get only XML files
57.           FileFilter filter = new FileExtensionFileFilter(".xml");
58.
```

```
59.            File[] files = dir.listFiles(filter);
60.
61.            if (files == null || files.length <= 0) {
62.                 logger.info("No XML files found in " + dir);
63.
64.                 // Return since there were no files
65.                 return;
66.            }
67.
68.            // The number of XML files
69.            int size = files.length;
70.
71.            // Iterate through the files found
72.            for (int i = 0; i < size; i++) {
73.
74.                 File file = files[i];
75.
76.                 // Log something interesting about each file.
77.                 File aFile = file.getAbsoluteFile();
78.                 long fileSize = file.length();
79.                 String msg = aFile + " - Size: " + fileSize;
80.                 logger.info(msg);
81.            }
82.       }
83. }
```

让我们来细细看看代码 3.1 中做了些什么。

当 Quartz 调用 execute() 方法，会传递一个 org.quartz.JobExecutionContext 上下文变量，里面封装有 Quartz 的运行时环境和当前正执行的 Job。通过 JobexecutionContext，你可以访问到调度器的信息，作业和作业上的触发器的信息，还有更多更多的信息。在代码 3.1 中，JobExecutionContext 被用来访问 org.quartz.JobDetail 类，JobDetail 类持有 Job 的详细信息，包括为 Job 实例指定的名称，Job 所属组，Job 是否被持久化(易失性)，和许多其他感兴趣的属性。

JobDetail 又持有一个指向 org.quartz.JobDataMap 的引用。JobDataMap 中有为指定 Job 配置的自定义属性。例如，在代码 3.1中，我们从 JobDataMap 中获得欲扫描的目录名，我们可以在 ScanDirectoryJob 中硬编码这个目录名，但是这样的话我们难以重用这个 Job 来扫描别的目录了。在后面有一节“编程方式调度一个 Quartz Job”,你将会看到目录是如何配置到 JobDataMap 的。

execute() 方法中剩下的就是标准 Java 代码了：获得目录名并创建一个 java.io.File 对象。它还对目录名作为简单的校验，确保是一个有效且存在的目录。接着调用 File 对象的 listFiles() 方法得到目录下的文件。还创建了一个 java.io.FileFilter 对象作为参数传递给 listFiles() 方法。org.quartzbook.cavaness.FileExtensionFileFilter 实现了 java.io.FileFilter 接口，它的作用是过滤掉目录仅返回 XML 文件。默认情况下，listFiles() 方法是返回目录中所有内容，不管是文件还是子目录，所以我们必须过滤一下，因为我们只对 XML 文件感兴趣。

> **注：**
>
> FileExtensionFileFilter 并非 Quartz 框架的一部分；它是 java.io.FileFilter 的子类，而是 Java 核心的一部分。FileExtensionFileFilter 被创建为我们例子的一部分，用来滤除其他内容而只保留 XML 文件。它相当有用，你可以考虑为你的应用建一系列的文件过滤器，然后在你的 Quartz Job 中重用。

代码 3.2 是 FileExtensionFileFilter

```
1.  package  org.cavaness.quartzbook.chapter3;
2.
3.  import  java.io.File;
4.  import  java.io.FileFilter;
5.
6.  /**
7.   * A FileFilter that only passes Files of the specified extension.
8.   * <p>
```

```
 * Directories do not pass the filter.
 *
 * @author Chuck Cavaness
 */
public class FileExtensionFileFilter implements FileFilter {

      private String extension;

      public FileExtensionFileFilter(String extension) {
           this .extension = extension;
      }

       /*
       * Pass the File if it has the extension.
       */
       public boolean accept(File file) {
            // Lowercase the filename for easier comparison
           String lCaseFilename = file.getName().toLowerCase();

            return (file.isFile() &&
                        (lCaseFilename.indexOf(extension) > 0 )) ? true : false ;
      }
}
```

FileExtensionFileFilter 被用来屏蔽名称中不含字符串 “.xml” 的文件。它还屏蔽了子目录——这些子目录原本会让 listFiles() 方法正常返回。过滤器提供了一种很便利的方式选择性的向你的 Quartz 作业提供它能接受的作为输入的文件。

> **声明式之于编程式配置**
>
> 在 Quartz 中，我们有两种途径配置应用程序的运行时属性：声明式和编程式。有一些框架是使用外部配置文件的方式；我们都知道，在软件中硬编码设置有它的局限性。
> 从其他方面来讲，你将要根据具体的需求和功能来选择用哪一种方式。下一节强调了何时用声明式何时选择编程式。因为多数的 Java 行业应用都偏向于声明的方式，这也是我们所推荐的。

在下一节中，我们讨论如何为调度器配置 Job 和运行 ScanDirectoryJob。

[译者注] 翻译中还得细细考量，那些词要转换成中文，那些词保留成英文。例如，前面的 Job->作业、Application->应用、Task->任务、Scheduler之于调度器；具体下来 Quartz Application 可能比 Quartz 应用好理解，Quart Job 也没有 Quart 作业读来生硬。平时不细究，只是阅读英文资料的话，一眼掠过是不会太在意的，本人翻译中也有混用，还没能太明晰。

第三章. Hello Quartz (第二部分)

2. 调度 Quartz ScanDirectoryJob

到目前为止，我们已经创建了一个 Quartz job，但还没有决定怎么处置它——明显地，我们需以某种方式为这个 Job 设置一个运行时间表。时间表可以是一次性的事件，或者我们可能会安装它在除周日之外的每个午夜执行。你即刻将会看到，Quartz Schduler 是框架的心脏与灵魂。所有的 Job 都通过 Schduler 注册；必要时，Scheduler 也会创建 Job 类的实例，并执行实例的 execute() 方法。

> **Scheduler 会为每一次执行创建新的 Job 实例**
>
> Scheduler 在每次执行时都会为 Job 创建新的实例。这就意味着 Job 的任何实例变量在执行结束之后便会丢失。与此相反概念则可用述语*有状态的*（J2EE世界里常见语）来表达，但是应用 Quartz ，一个有状态的 Job 并不用多少开销，而且很容易的配置。当你创建一个有状态的 Job 时，有一些东西对于 Quartz 来说是独特的。最主要的就是不会出现两个有着相同状态的 Job 实例并发执行。这可能会影响到程序的伸缩性。这些或更多的问题将在以后的章节中详细讨论。

·创建并运行 Quartz Scheduler

在具体谈论 ScanDirectoryJob 之前，让我们大略讨论一下如何实例化并运行 Quartz Scheduler 实例。代码 3.3 描述了创建和启动一个 Quartz Scheduler 实例的必要且基本的步骤。

代码 3.3 运行一个简单的 Quartz 调度器

```
1.  package org.cavaness.quartzbook.chapter3;
2.
3.  package org.cavaness.quartzbook.chapter3;
4.
5.  import java.util.Date;
6.
7.  import org.apache.commons.logging.Log;
8.  import org.apache.commons.logging.LogFactory;
9.  import org.quartz.Scheduler;
10. import org.quartz.SchedulerException;
11. import org.quartz.impl.StdSchedulerFactory;
12.
13. public class SimpleScheduler {
14.      static Log logger = LogFactory.getLog(SimpleScheduler.class);
15.
16.      public static void main(String[] args) {
17.           SimpleScheduler simple = new SimpleScheduler();
18.           simple.startScheduler();
19.      }
20.
21.      public void startScheduler() {
22.           Scheduler scheduler = null;
23.
24.           try {
25.                // Get a Scheduler instance from the Factory
26.                scheduler = StdSchedulerFactory.getDefaultScheduler();
27.
28.                // Start the scheduler
29.                scheduler.start();
30.                logger.info("Scheduler started at " + new Date());
31.
32.           } catch (SchedulerException ex) {
33.                // deal with any exceptions
```

```
                logger.error(ex);
            }
        }
    }
```

运行上面 3.3 的代码，会有日志输出，你会看到类似如下的输出：

INFO [main] (SimpleScheduler.java:30) - Scheduler started at Mon Sep 05 13:06:38 EDT 2005

关闭 Quartz Info 级别的日志信息

假如你搭配着 Log4J 使用 Commons Logging 日志框架，就像本书的例子那样，你也许需要把除本书例子外，其他的所有 Info 级别以上的日志信息关闭掉。这是因为 Quartz 中 Debug 和 Info 级别的日志信息数量上大体相当。当你明白了 Quartz 在做什么的时候，你真正关注的信息却淹没在大量的日志信息中。为了不至于这样，你可以创建一个文件 log4j.properties 指定只输出 ERROR 级别信息，但是要为本书中的例子设置定显示 INFO 级别的信息。这里有一个 log4j.properties 的例子文件来达到这个目的：

```
# Create stdout appender
log4j.rootLogger=error, stdout

# Configure the stdout appender to go to the Console
log4j.appender.stdout=org.apache.log4j.ConsoleAppender

# Configure stdout appender to use the PatternLayout
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout

# Pattern output the caller's filename and line #
log4j.appender.stdout.layout.ConversionPattern=%5p [%t] (%F:%L) - %m%n

# Print messages of level INFO or above for examples
log4j.logger.org.cavaness.quartzbook=INFO
```

文件 log4j.properties 配置了默认向标准输出只输出 ERROR 级别以上的日志信息，但是任何在包 org.cavaness.quartzbook 中的 INFO 以上级别的信息也会输出。这有赖于以上属性文件最后一行配置。

代码 3.3 展示了启动一个 Quartz 调度器是那么的简单。当调度器起来之后，你可以利用它做很多事情或者获取到它的许多信息。例如，你也许需要安排一些 Job 或者改变又安排在调度器上 Job 的执行次数。你也许需要让调度器处于暂停模式，接着再次启动它以便重新执行在其上安排的作业。当调度器处于暂停模式时，不执行任何作业，即使是作业到了它所期待的执行时间。代码 3.4 展示了怎么把调度器置为暂停模式然后又继续运行，这样调度器会从中止处继续执行。

代码 3.4 设置调度器为暂停模式

```
private void modifyScheduler(Scheduler scheduler) {

    try {
        if (!scheduler.isInStandbyMode()) {
            // pause the scheduler
            scheduler.standby();
        }

        // Do something interesting here

        // and then restart it
        scheduler.start();

    } catch (SchedulerException ex) {
        logger.error(ex);
```

```
    }
}
```

代码 3.4 中的片断仅仅是一个最简单的例子，说明了当你执有一个 Quartz 调度器的引用，你可以利用它做一些你有感兴趣的事情。当然了，并非说 Scheduler 只有处于暂停模式才能很好的利用它。例如，你能在调度器处于运行状态时，安排新的作业或者是卸下已存在的作业。我们将通过本书的一个调度器尽可能的去掌握关于 Quartz 更多的知识。

上面的例子看起来都很简单，但千万不要被误导了。我们还没有指定任何作业以及那些作业的执行时间表。虽然代码 3.3 中的代码确实能启动运行，可是我们没有指定任何作业来执行。这就是我们下一节要讨论的。

·编程式安排一个 Quartz Job

所有的要 Quartz 来执行的作业必须通过调度器来注册。大多情况下，这会在调度器启动前做好。正如本章前面说过，这一操作也提供了声明式与编程式两种实现途径的选择。首先，我们讲解如何用编程的方式；接下来在本章，我们会用声明的方式重做这个练习。

因为每一个 Job 都必须用 Scheduler 来注册，所以先定义一个 JobDetail，并关联到这个 Scheduler 实例。见代码 3.5。

代码 3.5. 编程式安排一个 Job

```
package org.cavaness.quartzbook.chapter3;

import java.util.Date;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.quartz.JobDetail;
import org.quartz.Scheduler;
import org.quartz.SchedulerException;
import org.quartz.Trigger;
import org.quartz.TriggerUtils;
import org.quartz.impl.StdSchedulerFactory;

public class Listing_3_5 {
     static Log logger = LogFactory.getLog(Listing_3_5.class);

     public static void main(String[] args) {
          Listing_3_5 example = new Listing_3_5();

          try {
               // Create a Scheduler and schedule the Job
               Scheduler scheduler = example.createScheduler();
               example.scheduleJob(scheduler);

               // Start the Scheduler running
               scheduler.start();

               logger.info( "Scheduler started at " + new Date() )

          } catch (SchedulerException ex) {
               logger.error(ex);
          }
     }

     /*
      * return an instance of the Scheduler from the factory
      */
     public Scheduler createScheduler() throws SchedulerException {
          return StdSchedulerFactory.getDefaultScheduler();
```

```
40.     }
41.
42.     // Create and Schedule a ScanDirectoryJob with the Scheduler
43.     private void scheduleJob(Scheduler scheduler)
44.          throws SchedulerException {
45.
46.          // Create a JobDetail for the Job
47.          JobDetail jobDetail =
48.                       new JobDetail("ScanDirectory",
49.                  Scheduler.DEFAULT_GROUP,
50.                            ScanDirectoryJob.class);
51.
52.          // Configure the directory to scan
53.          jobDetail.getJobDataMap().put("SCAN_DIR",
54.                    "c:\\quartz-book\\input");
55.
56.          // Create a trigger that fires every 10 seconds, forever
57.          Trigger trigger = TriggerUtils.makeSecondlyTrigger(10);
58.          trigger.setName("scanTrigger");
59.          // Start the trigger firing from now
60.          trigger.setStartTime(new Date());
61.
62.          // Associate the trigger with the job in the scheduler
63.          scheduler.scheduleJob(jobDetail, trigger);
64.     }
65. }
```

上面程序提供了一个理解如何编程式安排一个 Job 很好的例子。代码首先调用 createScheduler() 方法从 Scheduler 工厂获取一个 Scheduler 的实例。得到 Scheduler 实例之后，把它传递给 schedulerJob() 方法，由它把 Job 同 Scheduler 进行关联。

首先，创建了我们想要运行的 Job 的 JobDetail 对象。JobDetail 构造器的参数中包含指派给 Job 的名称，逻辑组名，和实现 org.quartz.Job 接口的全限类名称。我们可以使用 JobDetail 的别的构造器。

```
public JobDetail();
public JobDetail(String name, String group, Class jobClass);
public JobDetail(String name, String group, Class jobClass,
    boolean volatility, boolean durability, boolean recover);
```

> **注**
>
> 一个 Job 在同一个 Scheduler 实例中通过名称和组名能唯一被标识。假如你增加两个具体相同名称和组名的 Job，程序会抛出 ObjectAlreadyExistsException 的异常。

在本章前面有说过，JobDetail 扮演着某一 Job 定义的角色。它带有 Job 实例的属性，能在运行时被所关联的 Job 访问到。其中在使用 JobDetail 时，的一个最重要的东西就是 JobDataMap，它被用来存放 Job 实例的状态和参数。在代码 3.5 中，待扫描的目录名称就是通过 scheduleJob() 方法存入到 JobDataMap 中的。

·理解和使用 Quartz Trigger

Job 只是一个部分而已。注意到代码 3.5，我们没有在 JobDetail 对象中为 Job 设定执行日期和次数。这是 Quartz Trigger 该做的事。顾名思义，Trigger 的责任就是触发一个 Job 去执行。当用 Scheduler 注册一个 Job 的时候要创建一个 Trigger 与这个 Job 相关联。Quartz 提供了四种类型的 Trigger，但其中两种是最为常用的，它们就是在下面章节中要用到的 SimpleTrigger 和 CronTrigger.

SimpleTrigger 是两个之中简单的那个，它主要用来激发单事件的 Job，Trigger 在指定时间激发，并重复 n 次--两次激发时间之间的延时为 m，然后结束作业。CronTrigger 非常复杂且强大。它是基于通用的公历，当需要用一种较复杂的时间表去执行一个 Job 时用到。例如，四月至九月的每个星期一、星期三、或星期五的午夜。

为更简单的使用 Trigger，Quartz 包含了一个工具类，叫做 org.quartz.TriggerUtils. TriggerUtils 提供了许多便捷的方法简化了构造和配置 trigger. 本章的例子中有用的就是 TriggerUtils 类；SimpleTrigger 和 CronTrigger 会在后面章节中用到。

正如你从代码3.5中看到的那样，调用了 TriggerUtils 的方法 makeSecondlyTrigger() 来创建一个每10秒种激发一次的 trigger(实际是由 TriggerUtils 生成了一个 SimpleTrigger 实例，但是我们的代码并不想知道这些)。我们同样要给这个 trigger 实例一个名称并告诉它何时激发相应的 Job；在代码3.5 中，与之关联的 Job 会立即启动，因为由方法 setStartTime() 设定的是当前时间。

代码 3.5 演示的是如何向 Scheduler 注册单一 Job。假如你有不只一个个 Job (你也许就是)，你将需要为每一个 Job 创建各自的 JobDetail。每一个 JobDetail 必须通过 scheduleJob() 方法一一注册到 Scheduler 上。

> **注**
>
> 回到代码 3.1 中，我们从代码中看到要扫描的目录名属性是从 JobDataMap 中获取到的。
> 再看代码 3.5，你能发现这个属性是怎么设置的。

如果你想重用了一个 Job 类，让它产生多个实例运行，那么你需要为每个实例都创建一个 JobDetail。例如，假如你想重用 ScanDirectoryJob 让它检查两个不同的目录，你需要创建并注册两个 JobDetail 实例。代码 3.6 显示了是如何做的。

代码 3.6. 运行 ScanDirectoryJob 的多个实例

```
1.  package org.cavaness.quartzbook.chapter3;
2.
3.  import java.util.Date;
4.
5.  import org.apache.commons.logging.Log;
6.  import org.apache.commons.logging.LogFactory;
7.  import org.quartz.JobDetail;
8.  import org.quartz.Scheduler;
9.  import org.quartz.SchedulerException;
10. import org.quartz.Trigger;
11. import org.quartz.TriggerUtils;
12. import org.quartz.impl.StdSchedulerFactory;
13.
14. public class Listing_3_6 {
15.      static Log logger = LogFactory.getLog(Listing_3_6.class);
16.
17.      public static void main(String[] args) {
18.           Listing_3_6 example = new Listing_3_6();
19.
20.           try {
21.                // Create a Scheduler and schedule the Job
22.                Scheduler scheduler = example.createScheduler();
23.
24.                // Jobs can be scheduled after Scheduler is running
25.                scheduler.start();
26.
27.                logger.info("Scheduler started at " + new Date());
28.
29.                // Schedule the first Job
30.                example.scheduleJob(scheduler, "ScanDirectory1",
31.                          ScanDirectoryJob.class,
32.                                    "c:\\quartz-book\\input", 10);
33.
34.                // Schedule the second Job
35.                example.scheduleJob(scheduler, "ScanDirectory2",
36.                          ScanDirectoryJob.class,
37.                                "c:\\quartz-book\\input2", 15);
38.
```

```
            } catch (SchedulerException ex) {
                logger.error(ex);
            }
        }

        /*
         * return an instance of the Scheduler from the factory
         */
        public Scheduler createScheduler() throws SchedulerException {
            return StdSchedulerFactory.getDefaultScheduler();
        }

        // Create and Schedule a ScanDirectoryJob with the Scheduler
        private void scheduleJob(Scheduler scheduler, String jobName,
                Class jobClass, String scanDir, int scanInterval)
                throws SchedulerException {

             // Create a JobDetail for the Job
            JobDetail jobDetail =
                        new JobDetail(jobName,
                                Scheduler.DEFAULT_GROUP, jobClass);

            // Configure the directory to scan
            jobDetail.getJobDataMap().put("SCAN_DIR", scanDir);

            // Trigger that repeats every "scanInterval" secs forever
            Trigger trigger =
                        TriggerUtils.makeSecondlyTrigger(scanInterval);

            trigger.setName(jobName + "-Trigger");

            // Start the trigger firing from now
            trigger.setStartTime(new Date());

            // Associate the trigger with the job in the scheduler
            scheduler.scheduleJob(jobDetail, trigger);
        }
    }
```

代码 3.6 和代码 3.5 非常的类似，只存在一点小小的区别。主要的区别是代码 3.6 中重构了允许多次调用 schedulerJob() 方法。在设置上比如 Job 名称和扫描间隔名称通过参数传。因此从 createScheduler() 方法获取到 Scheduler 实例后，两个 Job(同一个类) 用不同的参数就被安排到了 Scheduler 上了。(译者注：当用调 createScheduler() 方法得到 Scheduler 实例后，都还没有往上注册 Job，何来两个 Job 呢)。

> **在 Scheduler 启动之前还是之后安排 Job 代码**
>
> 代码 3.6 中，我们在安排 Job 之前就调用了 Scheduler 的 start() 方法。回到代码 3.5 中，采用了另一种方式：我们是在 Job 安排了之后调用了 start() 方法。Job 和 Trigger 可在任何时候在 Scheduler 添加或删除 (除非是调用了它的 shutdown()方法)。

·运行代码 3.6 中的程序

如果我们执行类 Listing_3_6，会得到类似如下的输出：

```
INFO [main] (Listing_3_6.java:35) - Scheduler started at Mon Sep 05 15:12:15 EDT 2005
 INFO [QuartzScheduler_Worker-0] ScanDirectory1 fired at Mon Sep 05 15:12:15 EDT 2005
 INFO [QuartzScheduler_Worker-0] - c:\quartz-book\input\order-145765.xml - Size: 0
 INFO [QuartzScheduler_Worker-0] - ScanDirectory2 fired at Mon Sep 05 15:12:15 EDT 2005
 INFO [QuartzScheduler_Worker-0] - No XML files found in c:\quartz-book\input2
 INFO [QuartzScheduler_Worker-1] - ScanDirectory1 fired at Mon Sep 05 15:12:25 EDT 2005
```

```
INFO [QuartzScheduler_Worker-1] - c:\quartz-book\input\order-145765.xml - Size: 0
INFO [QuartzScheduler_Worker-3] - ScanDirectory2 fired at Mon Sep 05 15:12:30 EDT 2005
INFO [QuartzScheduler_Worker-3] - No XML files found in c:\quartz-book\input2
```

第三章. Hello Quartz (第三部分)

3. 声明式部署一个 Job

前面我们讨论过，尽可能的用声明式处理软件配置，其次才才虑编程式。再来看代码 3.6，如果我们要在 Job 启动之后改变它的执行时间和频度，必须去修改源代码重新编译。这种方式只适用于小的例子程序，但是对于一个大且复杂的系统，这就成了一个问题了。因此，假如能以声明式部署 Quart Job 时，并且也是需求允许的情况下，你应该每次都选择这种方式。

·配置 quartz.properties 文件

文件 quartz.properties 定义了 Quartz 应用运行时行为，还包含了许多能控制 Quartz 运转的属性。本章只会讲到它的基本配置；更多的高级设置将在以后讨论。在现阶段也不用太深入到每一项配置有效值的细节。

现在我们来看看最基础的 quartz.properties 文件，并讨论其中一些设置。代码 3.7 是一个修剪版的 quartz.propertis 文件。

> **注**
>
> Quartz 框架会为几乎所有的这些属性设定默认值。

代码 3.7. 基本的 Quartz Properties 文件

```
#============================================================================
#Configure Main Scheduler Properties
#============================================================================
org.quartz.scheduler.instanceName = QuartzScheduler
org.quartz.scheduler.instanceId = AUTO

#============================================================================
#Configure ThreadPool
#============================================================================
org.quartz.threadPool.threadCount =  5
org.quartz.threadPool.threadPriority = 5
org.quartz.threadPool.class = org.quartz.simpl.SimpleThreadPool

#============================================================================
#Configure JobStore
#============================================================================
org.quartz.jobStore.class = org.quartz.simpl.RAMJobStore

#============================================================================
#Configure Plugins
#============================================================================
org.quartz.plugin.jobInitializer.class =
org.quartz.plugins.xml.JobInitializationPlugin

org.quartz.plugin.jobInitializer.overWriteExistingJobs = true
org.quartz.plugin.jobInitializer.failOnFileNotFound = true
org.quartz.plugin.jobInitializer.validating=false
```

在代码 3.7 所示的 quartz.properties 文件中，属性被逻辑上分为了四部分。属性在写法上无须要求分组或按特定的顺序。有 # 的行是注释行。

> **注**
>
> 这里讨论的并没有涉及到所有可能的设置，仅仅是一些基本的设置。也是你需要去熟悉的，能使声明式例子运转起来的必须的设置项。quartz.properties 中的所有属性配置将会分散在本书中的各章节中依据所在章节涉及内容详细讨论。

·调度器属性

第一部分有两行，分别设置调度器的实例名(instanceName) 和实例 ID (instanceId)。属性 org.quartz.scheduler.instanceName 可以是你喜欢的任何字符串。它用来在用到多个调度器区分特定的调度器实例。多个调度器通常用在集群环境中。(Quartz 集群将会在第十一章，“Quartz 集群”中讨论)。现在的话，设置如下的一个字符串就行：

org.quartz.scheduler.instanceName = QuartzScheduler

实际上，这也是当你没有该属性配置时的默认值。

代码 3.7 中显示的调度器的第二个属性是 org.quartz.scheduler.instanceId。和 instaneName 属性一样，instanceId 属性也允许任何字符串。这个值必须是在所有调度器实例中是唯一的，尤其是在一个集群当中。假如你想 Quartz 帮你生成这个值的话，可以设置为 AUTO。如果 Quartz 框架是运行在非集群环境中，那么自动产生的值将会是 NON_CLUSTERED。假如是在集群环境下使用 Quartz，这个值将会是主机名加上当前的日期和时间。大多情况下，设置为 AUTO 即可。

·线程池属性

接下来的部分是设置有关线程必要的属性值，这些线程在 Quartz 中是运行在后台担当重任的。threadCount 属性控制了多少个工作者线程被创建用来处理 Job。原则上是，要处理的 Job 越多，那么需要的工作者线程也就越多。threadCount 的数值至少为 1。Quartz 没有限定你设置工作者线程的最大值，但是在多数机器上设置该值超过100的话就会显得相当不实用了，特别是在你的 Job 执行时间较长的情况下。这项没有默认值，所以你必须为这个属性设定一个值。

threadPriority 属性设置工作者线程的优先级。优先级别高的线程比级别低的线程更优先得到执行。threadPriority 属性的最大值是常量 java.lang.Thread.MAX_PRIORITY，等于10。最小值为常量 java.lang.Thread.MIN_PRIORITY，为1。这个属性的正常值是 Thread.NORM_PRIORITY，为5。大多情况下，把它设置为5，这也是没指定该属性的默认值。

最后一个要设置的线程池属性是 org.quartz.threadPool.class。这个值是一个实现了 org.quartz.spi.ThreadPool 接口的类的全限名称。Quartz 自带的线程池实现类是 org.quartz.smpl.SimpleThreadPool，它能够满足大多数用户的需求。这个线程池实现具备简单的行为，并经很好的测试过。它在调度器的生命周期中提供固定大小的线程池。你能根据需求创建自己的线程池实现，如果你想要一个随需可伸缩的线程池时也许需要这么做。这个属性没有默认值，你必须为其指定值。

·作业存储设置

作业存储部分的设置描述了在调度器实例的生命周期中，Job 和 Trigger 信息是如何被存储的。我们还没有谈论到作业存储和它的目的；因为对当前例子是非必的，所以我们留待以后说明。现在的话，你所要了解的就是我们存储调度器信息在内存中而不是在关系型数据库中就行了。

把调度器信息存储在内存中非常的快也易于配置。当调度器进程一旦被终止，所有的 Job 和 Trigger 的状态就丢失了。要使 Job 存储在内存中需通过设置 org.quartz.jobStrore.class 属性为 org.quartz.simpl.RAMJobStore，就像在代码 3.7 所做的那样。假如我们不希望在 JVM 退出之后丢失调度器的状态信息的话，我们可以使用关系型数据库来存储这些信息。这需要另一个作业存储(JobStore) 实现，我们在后面将会讨论到。第五章“Cron Trigger 和其他”和第六章“作业存储和持久化”会提到你需要用到的不同类型的作业存储实现。

·插件配置

在这个简单的 quartz.properties 文件中最后一部分是你要用到的 Quart 插件的配置。插件常常在别的开源框架上使用到，比如 Apache 的 Struts 框架(见 http://struts.apache.org)。

一个声明式扩框架的方法就是通过新加实现了 org.quartz.spi.SchedulerPlugin 接口的类。SchedulerPlugin 接口中有给调度器调用的三个方法。

> **注**
>
> Quartz 插件会在第八章“使用 Quartz 插件”中详细讨论

要在我们的例子中声明式配置调度器信息，我们会用到一个 Quartz 自带的叫做 org.quartz.plugins.xml.JobInitializationPlugin 的插件。

默认时，这个插件会在 classpath 中搜索名为 quartz_jobs.xml 的文件并从中加载 Job 和 Trigger 信息。

在下一节中讨论 quartz_jobs.xml 文件，这是我们所参考的非正式的 Job 定义文件。

> **注**
>
> 默认时，插件 JobInitializationPlugin 在 classpath 中寻找 quartz_jobs.xml 文件。你可以覆盖相应设置强制这个插件使用不同的文件名查找。要做到这个，你必须设置上一节讨论的 quartz.properties 中的文件名。目前，我们就使用默认的文件名 quartz_jobs.xml，至于如何修改 quartz.properties 中相应设置会在本章中后面讲到。

·使用 quartz_jobx.xml 文件

代码 3.8 就是目录扫描例子的 Job 定义的 XML 文件。正如代码 3.5 所示例子那样，这里我们用的是声明式途径来配置 Job 和 Trigger 信息的。

代码 3.8. ScanDirectory Job 的 quartz_jobs.xml

```
<?xml version='1.0' encoding='utf-8'?>

<quartz>

  <job>
    <job-detail>
     <name>ScanDirectory</name>
     <group>DEFAULT</group>
     <description>
          A job that scans a directory for files
     </description>
     <job-class>
            org.cavaness.quartzbook.chapter3.ScanDirectoryJob
     </job-class>
     <volatility>false</volatility>
     <durability>false</durability>
     <recover>false</recover>
     <job-data-map allows-transient-data="true">
         <entry>
         <key>SCAN_DIR</key>
         <value>c:\quartz-book\input</value>
       </entry>
     </job-data-map>
    </job-detail>

    <trigger>
     <simple>
       <name>scanTrigger</name>
       <group>DEFAULT</group>
       <job-name>ScanDirectory</job-name>
       <job-group>DEFAULT</job-group>
       <start-time>2005-06-10 6:10:00 PM</start-time>
       <!-- repeat indefinitely every 10 seconds -->
       <repeat-count>-1</repeat-count>
       <repeat-interval>10000</repeat-interval>
     </simple>
    </trigger>

  </job>
</quartz>
```

<job> 元素描述了一个要注册到调度器上的 Job，相当于我们在前面章节中使用 scheduleJob() 方法那样。你所看到的<job-detail> 和 <trigger> 这两个元素就是我们在代码 3.5 中以编程式传递给方法 schedulerJob() 的参数。前面本质上是与这里一样的，只是现在用的是一种较流行声明的方式。你还可以对照着代码 3.5 中的例子来看在代码3.8 中我们是如何设置 SCAN_DIR 属性到 JobDataMap 中的。

<trigger>元素也是非常直观的：它使用前面同样的属性，但更简单的建立一个 SimpleTrigger。因此代码 3.8 仅仅是一种不同的(可论证的且更好的)方式做了代码 3.5 中同样的事情。显然，你也可以支持多个 Job。在代码3.6 中我们编程的方式那么做的，也能用声明的方式来支持。代码 3.9 显示了与代码 3.6 可比较的版本

代码 3.9. 你能在一个 quartz_jobs.xml 文件中指定多个 Job

```
<?xml version='1.0' encoding='utf-8'?>

<quartz>
  <job>
    <job-detail>
     <name>ScanDirectory1</name>
     <group>DEFAULT</group>
     <description>
           A job that scans a directory for files
     </description>
     <job-class>
           org.cavaness.quartzbook.chapter3.ScanDirectoryJob
     </job-class>
     <volatility>false</volatility>
     <durability>false</durability>
     <recover>false</recover>

     <job-data-map allows-transient-data="true">
     <entry>
       <key>SCAN_DIR</key>
         <value>c:\quartz-book\input1</value>
     </entry>
    </job-data-map>
  </job-detail>

  <trigger>
    <simple>
     <name>scanTrigger1</name>
     <group>DEFAULT</group>
     <job-name>ScanDirectory1</job-name>
     <job-group>DEFAULT</job-group>
     <start-time>2005-07-19 8:31:00 PM</start-time>
     <!-- repeat indefinitely every 10 seconds -->
     <repeat-count>-1</repeat-count>
     <repeat-interval>10000</repeat-interval>
    </simple>
  </trigger>
</job>

<job>
  <job-detail>
    <name>ScanDirectory2</name>
    <group>DEFAULT</group>
    <description>
          A job that scans a directory for files
    </description>
    <job-class>
          org.cavaness.quartzbook.chapter3.ScanDirectoryJob
    </job-class>
```

```
    <volatility>false</volatility>
    <durability>false</durability>
    <recover>false</recover>

    <job-data-map allows-transient-data="true">
      <entry>
       <key>SCAN_DIR</key>
       <value>c:\quartz-book\input2</value>
      </entry>
    </job-data-map>
  </job-detail>

  <trigger>
    <simple>
     <name>scanTrigger2</name>
     <group>DEFAULT</group>
     <job-name>ScanDirectory2</job-name>
     <job-group>DEFAULT</job-group>
     <start-time>2005-06-10 6:10:00 PM</start-time>
     <!-- repeat indefinitely every 15 seconds -->
     <repeat-count>-1</repeat-count>
     <repeat-interval>15000</repeat-interval>
    </simple>
  </trigger>
 </job>
</quartz>
```

·为插件修改 quartz.properties 配置

在本章前面，告诉过你的是，JobInitializationPlugin 找寻 quartz_jobs.xml 来获得声明的 Job 信息。假如你想改变这个文件名，你需要修改 quartz.properties 来告诉插件去加载那个文件。例如，假如你想要 Quartz 从名为 my_quartz_jobs.xml 的 XML 文件中加载 Job 信息，你不得不为插件指定这一文件名。代码 3.10 显示了怎么完成这个配置；我们现在是最后一次在这里重复说明这一插件部分。

代码 3.10. 为 JobInitializationPlugin 修改 quartz.properties

```
org.quartz.plugin.jobInitializer.class = org.quartz.plugins.xml.JobInitializationPlugin

org.quartz.plugin.jobInitializer.fileName = my_quartz_jobs.xml

org.quartz.plugin.jobInitializer.overWriteExistingJobs = true
org.quartz.plugin.jobInitializer.validating = false
org.quartz.plugin.jobInitializer.overWriteExistingJobs = false
org.quartz.plugin.jobInitializer.failOnFileNotFound = true
```

在代码 3.10中，我们添加了属性 org.quartz.plugin.jobInitializer.fileName 并设置该属性值为我们想要的文件名。这个文件名要对 classloader 可见，也就是说要在 classpath 下。

当 Quartz 启动后读取 quartz.properties 文件，然后初始化插件。它会传递上面配置的所有属性给插件，这时候插件也就得到通知去搜寻不同的文件。

译者后记：
想了又想，关于动词的 “Schedule” 还是选择“部署”，此前用的是“安排”，感觉不那么正式。当然英语中“部署”基本都用“Deploy”对应，平时与同事交流 Quartz 方面的技术都是说“往调度器上部署一个 Job”的，只要词能达意就行。

对于 “register with the Scheduler”，有时候是用的“通过调度器来注册”，有时候是“注册到调度器上”，意思基本一致的。

第三章. Hello Quartz (第四部分)

4. 打包 Quartz 应用程序

让我们最后简单讨论打包一个用到了 Quarts 框架的应用程序的流程，也以此来结束本章的内容。

·Quartz 第三方依赖包

从 1.5 版的发行包开始，你会看到一个 <QUARTZ_HOME>\lib 目录，在这个目录，你会发现几个子目录：

·<QUARTZ_HOME>\lib\core

·<QUARTZ_HOME>\lib\optional

·<QUARTZ_HOME>\lib\build

作为开发呢，你绝对是需求 Quartz JAR 包，也需要其他一些依赖包。需要哪些第三方包还依赖于你是运行在独立环境中还是作为一个 J2EE 发行包的一部份。典型的，jakarta Commons 库 (commons-loggin, commons-beanutils，还有其他的) 总是要用到。然而，当你是部署到一个应用服务器环境中，你需要确保不能把那些在应用服务器上已存在的包拷过去；如果你这样做的，你可能回得到非常奇怪的结果。

表 3.1 列出了第三方包的信息，帮助你确定是否需要在应用中包含它们

表 3.1. Quartz 第三方包，必须/可选

名称	必须/备注	网址
activation.jar	主要是 JavaMail 要用到	http://java.sun.com/products/javabeans/glasgow/jaf.html
commons-beanutils.jar	是	http://jakarta.apache.org/commons/beanutils
commons-collections.jar	是	http://jakarta.apache.org/commons/collections
commons-dbcp-1.1.jar	是，假如用到数据库作为作业存储	http://jakarta.apache.org/commons/dbcp
commons-digester.jar	是	假如你使用了某些插件，就需要它
commons-logging.jar	是	http://jakarta.apache.org/commons/logging/
commons-pool-1.1.jar		http://jakarta.apache.org/commons/pool/
javamail.jar	发送 e-mail 用	http://java.sun.com/products/javamail/
jdbc2_0-stdext.jar	是，假如用到数据库作为作业存储	http://java.sun.com/products/jdbc/
jta.jar	是，假如用到数据库作为作业存储	http://java.sun.com/products/jta/database
quartz.jar	是	Quart 框架核心包
servlet.jar	假如使用了Servlet 容器，但容器中应该存在	http://java.sun.com/products/servlet/
log4j.jar	来吧，谁没用过它呢？	http://logging.apache.org/

·配置和属性文件

你还必须在你的应用中包含 quartz.properties。假如你是以散装(exploded format) 形式部署应用，你应该把 quartz.properties 文件放置在类加载器能够加载的目录中。(所谓的 “exploed” 形式指不打成一个 JAR、WAR、EAR 或者其他 Java 包，以独立文件存在于文件系统中) 例如，如果你有一个 classes 目录(比如一个 Web 应用的 WEB-INF/classes 目录)，就可以把 quartz.properties 文件放在那儿。假如你以 Java 打包形式部署，应该放属性文件放在包的根下。在对待 quartz_job.xml 文件时也使用同样的规则。

[译者注]
OK，至此，第三章的内容即告完成，接下来将会跳到对第六章的翻译。

第四章. 部署 Job (第一部分)

第四章. 部署 Job

在上一章中，你首次尝试使用了 Quartz 来部署 Job。无可否认地，那些 Job 都不是很复杂，但这个不是重点。你应该轻松的对如何构造并部署 Job 有了相当的了解，更重要的是，由此热情的希望学得更多的东西。在本章中将会继续给你讲述。

第四章将带领你深入到 Quart 框架的核心部分。可证明的是，这一章对于阅读和理解本书是非常之重要的。调度器(Scheduler)是此框架的心脏。本章关注于如何使用 Scheduler 来管理你的 Job；如何创建并关联触发器以使 Job 能被触发执行；以及如何选择 calendar 为给定的时程安排提供更多的灵活性。

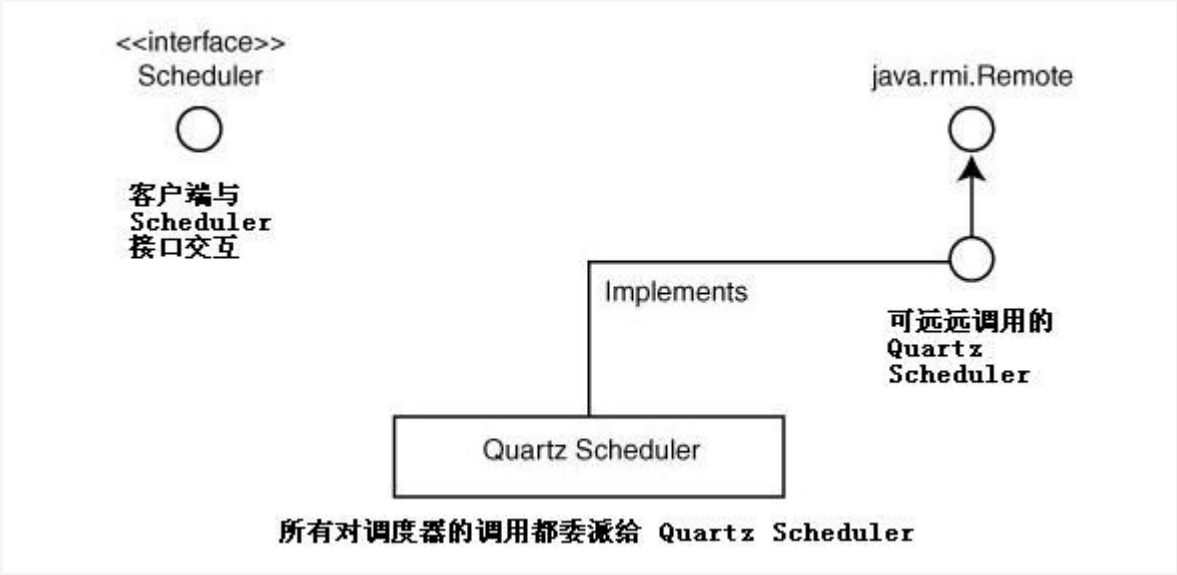
> *此刻，什么也没发生，下一刻，仍旧什么也没发生。*
>
> *道格拉斯.亚当斯，《银河系漫游指南》*

1. Quartz 调度器

Quartz 框架包含许多的类和接口，它们分布在大概 11 个包中。多数你所要使用到的类或接口放置在 org.quartz 包中。这个包含盖了 Quartz 框架的公有 API.

我们不打算对这个框架的所有类和接口都面面俱到。而所要介绍的是那些有助于你理解 Quartz 如何做它该做事情的组件的子集。图 4.1 展示了一个只留下必要的调度器的精简类图。

图 4.1. Quartz 类图(仅显示主要组件)



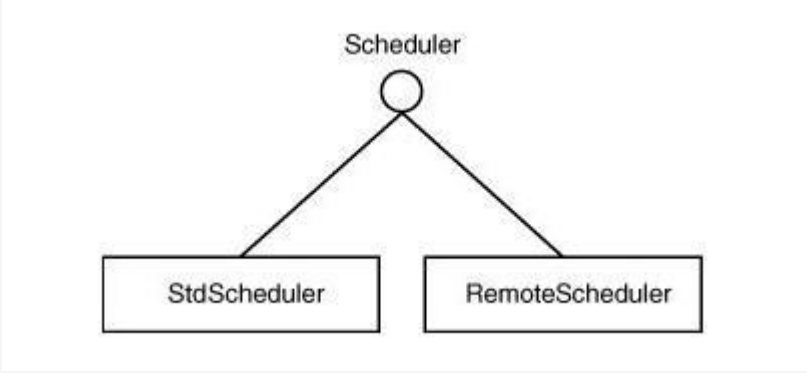
Scheduler 是 Quartz 的主要 API。对于 Quartz 用户来说，多数时候与框架的交互是发生于 Scheduler 之上的。客服端与 Scheduler 交互是通过 org.quartz.Scheduler 接口的。这个 Scheduler 的实现，在这种情况下，是一个代理，对其中方法调用会传递到 QuartzScheduler 实例上。QuartzScheduler 对于客户端是不可见的，并且也不存在与此实例的直接交互。

QuartzScheduler 处在框架根的位置，它是一个引擎驱动着整个框架。并非所有的功能都直接内建到 QuartzScheduler，然而，框架为灵活性和可配置性考虑而设计，所以许多重要的功能由分离的组件和子框架实现。这就意味着 Quartz 用户可以用自己某个关键特征实现来替换原有默认实现。即使 QuartzScheduler 代理了它的一些职责，但它仍然掌控着整个作业调度流程。

·Quartz Scheduler 类层次

客户端会同两种类型的 Scheduler 交互，如图 4.2. 它们都实现了 org.quartz.Scheduler 接口。

图 4.2. org.quartz.Scheduler 类层次



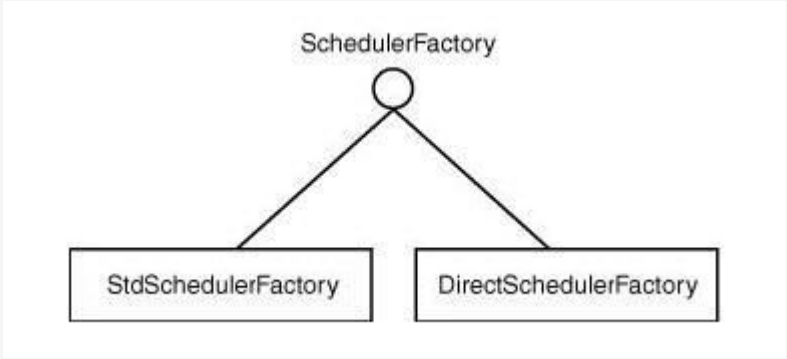
作为一个 Quartz 用户，你要与实现了 org.quartz.Scheduler 接口的类交互。在你调用它的任何 API 之前，你需要知道如何创建一个 Scheduler 的实例。

2. Quartz SchedulerFactory

尽管你已使用到了 Scheduler 类型了，但你未曾显式的去创建 Scheduler 的实例。取而代之的是用了某个工厂方法来确保了构造出 Sheduler 实例并正确的得到初始化。(工厂设计模式之所以谓之工厂模式是因为它承担了生产对象的职责。在这里是生产了一个 Scheduler 实例) Quartz 框架为这一目的提供了 org.quartz.SchedulerFactory 接口。角色 SchedulerFactory 就是用来产生 Scheduler 实例的。当 Scheduler 实例被创建之后，就会存到一个仓库中(org.quartz.impl.SchedulerRepository)，这个仓库还提供了通过一个 class loader 查询实例的机制。要使用 Scheduler 实例，客户端必须从工厂（和随同的仓库中）使用不同方法调用来获取到它们。换句话说，要通过工厂创建一个 Scheduler 实例并获取到它需要经由两次方法调用。有一些方便的方法封装了那两个方法，你将很快能看到。

你可使用两种不同类型的 SchedulerFactory 来创建 Scheduler 实例 (看图 4.3)

图 4.3. 所有的 Scheduler 实例应该由 SchedulerFactory 来创建



这两个 Scheduler 工厂分别是 org.quartz.impl.DirectoSchedulerFactory 和 org.quartz.impl.StdSchedulerFactory. 让我们来逐个检视它们。

·使用 DirectSchedulerFactory

DirectSchedulerFactory 是为那些想绝对控制 Scheduler 实例是如何生产出的人所设计的。代码 4.1 显示了最简单的方式去使用 DirectSchedulerFactory 来创建一个 Scheduler 实例。

代码 4.1. 使用 DirectSchedulerFactory

```
1.  public class Listing_4_1 {
2.       static Log logger = LogFactory.getLog(Listing_4_1.class);
3.
4.       public static void main(String[] args) {
5.            Listing_4_1 example = new Listing_4_1();
6.            example.startScheduler();
7.       }
8.
9.       public void startScheduler() {
10.           DirectSchedulerFactory factory=DirectSchedulerFactory.getInstance();
11.
12.           try {
13.               // Initialize the Scheduler Factory with 10 threads
14.               factory.createVolatileScheduler(10);
```

```
15.
16.                 // Get a scheduler from the factory
17.                 Scheduler scheduler = factory.getScheduler();
18.
19.                 // Start the scheduler running
20.                 logger.info("Scheduler starting up...");
21.                 scheduler.start();
22.
23.             } catch (SchedulerException ex) {
24.                  logger.error(ex);
25.             }
26.         }
27.     }
```

当使用 DirectSchedulerFactory 时，有三个基本的步骤。首先，你必须用静态方法 getInstance() 获取到工厂的实例。当你持有了工厂的实例之后，你必须调用其中的一个 createXXX 方法去初始化它。如代码 4.1 所示例子中是调用 createVolatileScheduler() 方法告诉工厂以十个工作者线程初始化它自己（至于工作者线程的更多内容将在本章的后部分讨论到）。第三步也就是最后一步是通过工厂的 getScheduler() 方法拿到 Scheduler 的实例。

> **在调用 getScheduler() 方法之后调用其中的一个 createXXX 方法**
>
> 方法 createVolatileScheduler() 方法不会返回 scheduler 的实例。createXXX() 方法是告诉工厂如何配置要创建的 Scheduler 实例。你必须调用方法 getScheduler() 获取到在工厂上执行方法 createXXX() 产生的实例。实际上，在调用 getScheduler() 方法之前，你必须调用其中一个 createXXX() 方法；否则，你将有收到一个 SchedulerException 错误，因为根本没有 Scheduler 实例存在。

你可从数个不同的 createXXX() 方法中选择，依赖于你想要的 Scheduler 类型和你需要怎样的配置。代码 4.1 中用的是 createVolatileScheduler() 方法创建 Scheduler 实例的。方法 createVolatileScheduler() 带有单个参数：要创建的线程数量。在图 4.2 中，你已看到还有一个 RemoteScheduler 类。你必须要用一个不同的 createXXX() 方法去创建 RemoteScheduler 实例。有两个版本的方法可用：

```
1.  public void createRemoteScheduler(String rmiHost, int rmiPort)
2.    throws SchedulerException;
3.
4.  protected void createRemoteScheduler(String schedulerName,
5.    String schedulerInstanceId, String rmiHost, int rmiPort)
6.    throws SchedulerException;
```

RemoteScheduler 会在第十章，“J2EE 中使用 Quartz”讨论。假如你就是想要一个标准的 Scheduler, 可以调用以下三个版本的方法中的一个：

```
1.  public void createScheduler(ThreadPool threadPool, JobStore jobStore)
2.    throws SchedulerException;
3.
4.  public void createScheduler(String schedulerName,
5.    String schedulerInstanceId, ThreadPool threadPool, JobStore jobStore)
6.    throws SchedulerException;
7.
8.  public void createScheduler(String schedulerName,
9.    String schedulerInstanceId, ThreadPool threadPool,
10.   JobStore jobStore, String rmiRegistryHost, int rmiRegistryPort,
11.   long idleWaitTime, long dbFailureRetryInterval)
12.   throws SchedulerException;
```

在上一章上，我们用了一个属性文件来初始化 Scheduler。要通过 DirectSchedulerFactory 创建一个 Scheduler 实例，你必须传递配置参数给其中的一个 createXXX() 方法。在下一节中，我们讨论 StdSchedulerFactory，一个 SchedulerFactory 版本，它依赖于一系列的属性来配置 Scheduler，而不是通过 createXXX() 方法参数来传递配置参数。这样也避免了在代码中对 Scheduler 的配置选项的硬编码。

·使用 StdSchedulerFactory

与 DirectSchedulerFactory 形成鲜明对比的是，org.quartz.impl.StdSchedulerFactory 依赖于一系列的属性来决定如何生

产出 Scheduler 实例。你可以通过以下三种途径向工厂提供那些属性：

·通过 java.util.Properties 实例提供

·通过外部属性文件提供

·通过含用属性文件内容的 java.io.InputStream 实例提供

> **Java 属性文件**
>
> 我们这里使用述语“属性文件”，对于 Java 传统来说就是：在一个外部文件中指定一系列的 key=value 对，并且每个 key=value 对独占一行。

代码 4.2 演示了第一种途径，通过一个 java.util.Properties 实例来提供属性。

代码 4.2. 使用 java.util.Properties 实例创建 StdSchedulerFactory

```
1.  public class Listing_4_2 {
2.       static Log logger = LogFactory.getLog(Listing_4_2.class);
3.
4.       public static void main(String[] args) {
5.            Listing_4_2 example = new Listing_4_2();
6.            example.startScheduler();
7.       }
8.
9.       public void startScheduler() {
10.
11.           // Create an instance of the factory
12.           StdSchedulerFactory factory = new StdSchedulerFactory();
13.
14.           // Create the properties to configure the factory
15.           Properties props = new Properties();
16.
17.           // required to supply threadpool class and num of threads
18.
19.           props.put(StdSchedulerFactory.PROP_THREAD_POOL_CLASS,
20.                     "org.quartz.simpl.SimpleThreadPool");
21.
22.           props.put("org.quartz.threadPool.threadCount", "10");
23.
24.           try {
25.
26.               // Initialize the factory with properties
27.               factory.initialize(props);
28.
29.               Scheduler scheduler = factory.getScheduler();
30.
31.               logger.info("Scheduler starting up...");
32.               scheduler.start();
33.
34.           } catch (SchedulerException ex) {
35.               logger.error(ex);
36.           }
37.      }
38.  }
39.
```

代码 4.2 提供了一个使用 StdSchedulerFactory 创建 Scheduler 实例的很简单的例子。在这个例子中向工厂传递了两个属性，它们分别是实现了 org.quartz.spi.ThreadPool 接口的类名和 Scheduler 用来处理 Job 的线程的数量。这两个属性是必须的，因为它们并没有被指定默认值（我们很快在后面讨论它们）

在代码 4.1 中的 DirectSchedulerFactory，我们调用它其中一个 createXXX() 方法来初始化工厂。而对于

StdSchedulerFactory，你使用的是一个有效的 initialize() 方法。在工厂初始化之后，你就可以调用它的 getScheduler() 方法获取到 Scheduler 的实例。使用 java.util.Properties 对象传递属性给工厂是一种配置 SchedulerFactory 的方式之一。硬编码配置属性的做法基本不推荐，也是可能的时候应尽力避免的做法。假如你需要修改上一例子中的线程数量，你将不得不修改代码然后重新编译。

幸运的是，StdSchedulerFactory 还有其他的方式来提供必须的属性。工厂也能通过传入一个外部文件名而被初始化，在这个外部文件中包含了这些配置项。应使用 initialize() 的替代方法形式如下：

```
1.  public void initialize(String filename) throws SchedulerException;
```

要使文件和属性能被成功加载的话，这个文件必须对于 classloader 是可见的。也就是说它必须在你的应用程序的 classpath 中。假如你用的是 java.io.InputStream 去加载文件，你可以使用另一个 initialize() 的替代方法如下：

```
1.  public void initialize(InputStream propertiesStream) throws SchedulerException;
```

在第三章，“Hello, Quartz”你已看到为 SchedulerFactory 从一个叫做 quartz.properties 的外部文件中加载设置的例子。这个外部属性文件就是要用前面的方法来加载。假如你没有为 initialize() 方法指定从哪儿读取属性，那么 StdSchedulerFactory 会试图从名为 quartz.properties 的文件中加载它们。这就是你在第三章看到的行为。

·使用默认的 quartz.properties 文件创建 Scheduler

假如你使用无参的 initialize() 方法，StdSchedulerFactory 会执行以下几个步骤去尝试为工厂加载属性：

1. 检查 System.getProperty("org.quartz.properties") 中是否设置了别的文件名

2. 否则，使用 quartz.properties 作为要加载的文件名

3. 试图从当前工作目录中加载这个文件

4. 试图从系统 classpath 下加载这个文件

> **在 Quartz Jar 包中的默认 quartz.properties 文件**
>
> 上面第4步总是能成功的，因为在 Quartz Jar 包中有一个默认的 quartz.properties 文件。假如你想使用另一个替代文件，你必须自己创建一个并确保它在 classpath 上。

使用 StdSchedulerFactory 来创建 Scheduler 实例的方式很普遍，因此在 StdSchedulerFactory 直接提供了一个方便的静态方法 getDefaultScheduler()，它就是使用前面列出的几个步骤来初始化工厂的。这如代码 4.3 所示。

代码 4.3. 使用静态的 getDefaultScheduler() 方法创建 Scheduler

```
1.  public class Listing_4_3 {
2.       static Log logger = LogFactory.getLog(Listing_4_3.class);
3.
4.       public static void main(String[] args) {
5.            Listing_4_3 example = new Listing_4_3();
6.            example.startScheduler();
7.       }
8.
9.       public void startScheduler() {
10.
11.           try {
12.               // Create a default instance of the Scheduler
13.               Scheduler scheduler =
14.                         StdSchedulerFactory.getDefaultScheduler();
15.               logger.info("Scheduler starting up...");
16.               scheduler.start();
17.
18.           } catch (SchedulerException ex) {
19.                logger.error(ex);
20.           }
21.      }}
```

在静态方法 getDefaultScheduler() 方法中调用了空的构造方法。假如之前未调用过任何一个 initialize() 方法，那么无参的 initialize() 方法会被调用。这会开始去按照前面说的顺序加载文件。默认情况下，quartz.properties 会被定位到，并从中加载属性。

·Scheduler 的功能

本章到此为止大部分笔墨都在着重论述如何获得一个 Scheduler 的实例。那么一旦你拿到 Scheduler 的实例之后，你能能此做些什么呢？好，现在开始，上面的例子中告诉了你可以调用它的 start() 方法。Scheduler 的 API 大概包括了 65 个不同的方法。我们不在此全部枚举出来，但是你需要理解其中的一小部分 API。

Scheduler 的 API 可以分组成以下三个类别：

·管理 Scheduler

·管理 Job

·管理 Trigger 和 Calendar

第四章. 部署 Job (第二部分)

3. 管理 Scheduler

除了启动 Scheduler, 在应用的生命周期中你也许还要执行 Scheduler 的别的一些操作。这些 Scheduler 操作包括查询、设置 Scheduler 为 standby 模式、继续、停止。很多情况下，当一个 Scheduler 启动后，除让它运行之外你不需要对它做任何事情的。在某些情形下，你也可能会要临时的终止 Scheduler 而转入到 standby 模式。

·启动 Scheduler

启动一个 Scheduler 也不总是一目了然的。当你有了 Scheduler 的实例，并得到正确的初始化，你的 Job 和 Triiger 也已注册上去了，你只需要简单的调用 start() 方法：

```
1.  //Create an instance of the Scheduler
2.  Scheduler scheduler =
3.     StdSchedulerFactory.getDefaultScheduler();
4.
5.  //Start the scheduler
6.  scheduler.start();
```

一旦 start() 方法被调用，Scheduler 就开始搜寻需要执行的 Job。在你刚得到一个 Scheduler 新的实例时，或者 Scheduler 被设置为 standby 模式后，你才可以调用 start() 方法。只要调用了 shutdown() 方法之后，你就不能再调用 Scheduler 实例的 start() 方法了。

> **别在 shutdown() 之后调用 start()**
>
> Scheduler 实例被关闭之后你就不能调用它的 start() 方法了。这是因为 shutdown() 方法销毁了为 Scheduler 创建的所有的资源(线程，数据库连接等)。假如你在 shutdown() 之后调用 start()你将收到 SchedulerException 的异常。

·Standby 模式

设置 Scheduler 为 standby 模式会导致 Scheduler 暂时停止查找 Job 去执行。例如，假定你的 Scheduler 是从数据库中获取到的 Job 信息，这时候你需要重启数据库。在数据库恢复之后你也需要重新启动 Scheduler，或者仅仅是设置为 standby 模式就行了。你能通过调用 standby() 方法让 Scheduler 进入到 standby 模式：

```
1.  public void standby() throws SchedulerException;
```

在 standby 模式，Scheduler 不再试图去执行 Job，因为那些搜寻要执行的 Job 的线程被暂停了下来。

·停止 Scheduler

你能使用两个版本的 shutdonw() 方来停止 Scheduler:

```
1.  public void shutdown(boolean waitForJobsToComplete)
2.  throws SchedulerException;
3.
4.  public void shutdown() throws SchedulerException;
```

上面那两个方法唯一不同之处是其中一个方法可接受一个 boolean 型参数，表示是否让当前正在进行的 Job 正常执行完成才停止 Scheduler。无参的 shutdown() 方法相当于调用 shutdown(false).

4. 管理 Job

前面章节我们粗略的看了 Quartz Job，现在来详细正式的讨论 Quartz Job 和怎样使用它们。

·什么是 Quartz Job?

很简单，一个 Quart Job 就是一个为你执行一个任务的 Java 类。这个任务是你能用 Java 编码的任何东西。下面就是一些任务的例子：

·使用 JavaMail (或别的 Mail 框架、如 Commons Net) 发送 e-mail

·创建一个远程接口并调用 EJB 上的方法

·获得 Hibernate 会话，查询、更新关系数据库中的数据

·使用 OSWorkflow 并在 Job 中调用一个工作流

上面的例子仅仅是一部份；你一定能列出不少你自己的任务。任何 Java 能做的事情都可以成为一个 Job.

·org.quartz.Job 接口

把 Quartz 作用到 Java 类上唯一要做的就是让它实现 org.quartz.Job 接口。你的 Job 类可以实现任何其他想要的接口或继承任何需要的基类，但是它自己或是它的超类必须实现这个 Job 接口。这个 Job 接口只定义了单个方法：

```
1.  public void execute(JobExecutionContext context)
2.      throws JobExecutionException;
```

当 Scheduler 决定了是时候运行 Job 时，方法 execute() 就会被调用，并传递一个 JobExecutionContext 对象给这个 Job。Quartz 加给方法 execute() 要承担的唯一合约责任就是如果在 Job 中出现严重问题时，必须抛出一个 org.quartz.JobExecutionException 异常。

·JobExecutionContext

当 Scheduler 调用一个 Job，一个 JobexecutionContext 传递给 execute() 方法。JobExecutionContext 对象让 Job 能访问 Quartz 运行时候环境和 Job 本身的明细数据。这就类似于在 Java Web 应用中的 servlet 访问 ServletContext 那样。通过 JobExecutionContext，Job 可访问到所处环境的所有信息，包括注册到 Scheduler 上与该 Job 相关联的 JobDetail 和 Triiger。代码 4.4 展示了一个叫做 PrintInfoJob 的 Job 打印出相关的一些信息。

从代码 4.4 中可以看到，Quartz Job 的一个非常基础的代码。PrintInfoJob 获得存储在 JobExecutionContext 中的 JobDetail 对象，进而打印出 Job 相关的基本明细。还JobDetail 类还会被给予更多的一些讨论。

代码 4.4. PrintInfoJob 显示了如何访问 JobExecutionContext

```
1.  public class PrintInfoJob implements Job {
2.      static Log logger = LogFactory.getLog(PrintInfoJob.class);
3.
4.      public void execute(JobExecutionContext context)
5.              throws JobExecutionException {
6.
7.          // Every job has its own job detail
8.          JobDetail jobDetail = context.getJobDetail();
9.
10.         // The name and group are defined in the job detail
11.         String jobName = jobDetail.getName();
12.         logger.info("Name: " + jobDetail.getFullName());
13.
14.         // The name of this class configured for the job
15.         logger.info("Job Class: " + jobDetail.getJobClass());
16.
```

```
            // Log the time the job started
            logger.info(jobName + " fired at " + context.getFireTime());

           logger.info("Next fire time " + context.getNextFireTime());
      }
}
```

·JobDetail

你第一次看到 org.quartz.JobDetail 类是在前面的第三章。对于部署在 Scheduler 上的每一个 Job 只创建了一个 JobDetail 实例。JobDetail 是作为 Job 实例进行定义的。注意到在代码 4.5 中不是把 Job 对象注册到 Scheduler；实际注册的是一个 JobDetail 实例。

代码 4.5. 注册到 Scheduler 上的是一个 JobDetail，而不是 Job

```
public class Listing_4_5 {
     static Log logger = LogFactory.getLog(Listing_4_5.class);

     public static void main(String[] args) {
          Listing_4_5 example = new Listing_4_5();
          example.runScheduler();
     }

     public void runScheduler() {

          try {

               // Create a default instance of the Scheduler
               Scheduler scheduler =
                              StdSchedulerFactory.getDefaultScheduler();

              logger.info("Scheduler starting up...");
              scheduler.start();

              // Create the JobDetail
              JobDetail jobDetail =
                            new JobDetail("PrintInfoJob",
                        Scheduler.DEFAULT_GROUP,
                                    PrintInfoJob.class);
                     // Create a trigger that fires now and repeats forever
              Trigger trigger = TriggerUtils.makeImmediateTrigger(
                        SimpleTrigger.REPEAT_INDEFINITELY, 10000);
              trigger.setName("PrintInfoJobTrigger");

             // register with the Scheduler
             scheduler.scheduleJob(jobDetail, trigger);

         } catch (SchedulerException ex) {
              logger.error(ex);
         }
     }
  }
```

在代码 4.5 中你可以看到，JobDetail 被加到 Scheduler 中了，而不是 job。Job 类是作为 JobDetail 的一部份，但是它直到 Scheduler 准备要执行它的时候才会被实例化的。

> **直到执行时才会创建 Job 实例**
>
> Job 的实例要到该执行它们的时候才会实例化出来。每次 Job 被执行，一个新的 Job 实例会

被创建。其中暗含的意思就是你的 Job 不必担心线程安全性，因为同一时刻仅有一个线程去执行给定 Job 类的实例，甚至是并发执行同一 Job 也是如此。

·使用 JobDataMap 对象设定 Job 状态

你能使用 org.quartz.JobDataMap 来定义 Job 的状态。JobDataMap 通过它的超类 org.quartz.util.DirtyFlagMap 实现了 java.util.Map 接口，你可以向 JobDataMap 中存入键/值对，那些数据对可在你的 Job 类中传递和进行访问。这是一个向你的 Job 传送配置的信息便捷方法。代码 4.6 描述的就是，我们特意创建了一个叫做 PrintJobDataMapJob 的 Job，使用了这种方式。

代码 4.6. 使用 JobDataMap 向你的 Job 传递配置信息

```
1.  public class Listing_4_6 {
2.       static Log logger = LogFactory.getLog(Listing_4_6.class);
3.
4.       public static void main(String[] args) {
5.            Listing_4_6 example = new Listing_4_6();
6.            example.runScheduler();
7.       }
8.
9.       public void runScheduler() {
10.           Scheduler scheduler = null;
11.
12.           try {
13.                // Create a default instance of the Scheduler
14.                scheduler = StdSchedulerFactory.getDefaultScheduler();
15.                scheduler.start();
16.                logger.info("Scheduler was started at " + new Date());
17.
18.                // Create the JobDetail
19.                JobDetail jobDetail =
20.                          new JobDetail("PrintJobDataMapJob",
21.                     Scheduler.DEFAULT_GROUP,
22.                          PrintJobDataMapJob.class);
23.
24.                // Store some state for the Job
25.                jobDetail.getJobDataMap().put("name", "John Doe");
26.                jobDetail.getJobDataMap().put("age", 23);
27.                jobDetail.getJobDataMap().put("balance",
28.                                                    new BigDecimal(1200.37));
29.                // Create a trigger that fires once
30.                Trigger trigger =
31.                          TriggerUtils.makeImmediateTrigger(0, 10000);
32.                trigger.setName("PrintJobDataMapJobTrigger");
33.
34.                scheduler.scheduleJob(jobDetail, trigger);
35.
36.           } catch (SchedulerException ex) {
37.                logger.error(ex);
38.           }
39.      }
40.  }
```

在代码 4.6 中，我们想要传递给 PrintJobDataMapJob 的信息存储到了 JobDetail 的 JobDataMap 中。因为 JobDataMap 实现了 java.util.Map 接口，所以我们可以以键/值对的形式在其中存储状态。JobDataMap 包含了许多精细的方法使处理起对象来方便简易。一般使用 map 的话，我们不得不要显式把 Object 类型转换成已知类型。JobDataMap 包含的方法能帮你完成这些工作。

当 Scheduler 最后调用了 Job，Job 可以通过 JobDetail 来访问到 JobDataMap 中的键/值对。代码 4.7 就是这个 PrintJobDataMapJob.

代码 4.7. Job 能通过 JobExecutionContext 对象访问 JobDataMap

```java
public class PrintJobDataMapJob implements Job {
     static Log logger = LogFactory.getLog(PrintJobDataMapJob.class);

     public void execute(JobExecutionContext context)
               throws JobExecutionException {

          logger.info("in PrintJobDataMapJob");

          // Every job has its own job detail
          JobDataMap jobDataMap =
                    context.getJobDetail().getJobDataMap();

          // Iterate through the key/value pairs
          Iterator iter = jobDataMap.keySet().iterator();

          while (iter.hasNext()) {
               Object key = iter.next();
               Object value = jobDataMap.get(key);

               logger.info("Key: " + key + " - Value: " + value);
          }
       }
  }
```

当你获得了 JobDataMap，你可以当它是任何 map 实例一样调用它的方法。一般的，你会自己选择一个预定义的键值来访问 JobDataMap 中的数据。你也可以像代码 4.7 那样遍历其中的所有数据。

像 PrintJobDataMapJob 那样的 Job，JobDataMap 中的属性成为了一个在部署 Job 的客户端与 Job 自身之间的一个非正式的契约。Job 的创建者应该非常仔细的文档化规定哪些属性是必须的，哪些又是可选的。这样有助于确保 Job 能被团队中的其他成员重用。

> **自 Quartz 1.5 始，JobDataMap 对 Trigger 也是可用的**
>
> 在 Quartz 1.5 中，JobDataMap 在 Trigger 级也是可用的。它的用途类似于 Job 级的 JobDataMap，此外它还能支持应用在同一个 JobDetail 上的多个 Trigger 上。伴随着加入到 Quartz 1.5 中的这一增强特性，可以使用 JobExecutionContext 的一个新的更方便的方法获取到 Job 和 Trigger 级的并集的 map 中的值。这个方法就是 getMergedJobDataMap()，它能够在 Job 中使用。从 Quartz 1.5 之后，使用这个方法被认为是获取 JobDataMap 最佳实践。

·有状态的之于无状态的 Job

你从前一节中学到，信息可插入到 JobDataMap 中然后被 Job 访问到。然而，对于每一次的 Job 执行，都会为特定的 Job 取用存储在某处(例如，数据库中)的值创建一个新的 JobDataMap 实例。因此，无法为两次 Job 调用之间持有那些信息，除非你使用有状态的 Job.

同样的方式，J2EE 中有状态的 Session Bean(SFSB) 能在两个调用之间保持状态，Quartz 的 StatefulJob 也能在两次 Job 执行间保持它的状态。然而，正如 SFSB 那样，Quartz 的有状态的 Job 与无状态的 Job 比起来也有一些不利的方面。

·使用有状态的 Job

当你需要在两次 Job 执行间维护状态的话，Quartz 框架为此提供了 org.quartz.StatefulJob 接口。StatefulJob 接口仅仅是扩展了 Job 接口，未加入新的方法。你只需要通过使用与 Job 接口相同的 execute() 方法简单的实现 StatefulJob 接口即可。假如你有已存在的 Job 类，你所有要做的只是改变 Job 的接口为 org.quartz.StatefulJob。

Job 和 StatefulJob 在框架中使用中存在两个关键差异。首先，JobDataMap 在每次执行之后重新持久化到 JobStore 中。这样就确保你对 Job 数据的改变直到下次执行仍然保持着。

> **改变有状态 Job 的 JobDataMap**
>
> 你可以在有状态 Job 中简单的通过 map 的 put() 方法来修改 JobDataMap.已存在的任何数据会被新的数据覆盖掉。你也能对无状态的 Job 这么做，但是因为对于无状态 Job 来说，

JobDataMap 不会持久化，所以数据不会保存下来。对于 Trigger 和 JobExecutionContext 上的 JobDataMap 的数据修改也是没能保存下来的。

另一个无状态和有状态 Job 重大区别就是：两个或多个有状态的 JobDetail 实例不能并发执行。说的是你创建并注册了一个有状态 JobDetail 到 Scheduler 上。你还建立了两个 Trigger 来触发这个 Job：一个每五分钟触发，另一个也是每五分钏触发。假如这两个 Trigger 试图在同一时刻触发 Job，框架是不允许这种事情发生的。第二个 Trigger 一直会被阻塞直到第一个结束。

这就产生了处理 JobDataMap 存储的需求了。因为 JobDataMap 的存储是伴随着 JobDetail 的，而 JobDetail 定义了 Job 实例，所以线程安全性问题必须纳入到我们考虑的范畴。同一时刻只能由一个线程去运行并更新 JobDataMap 存储。然而，由于在第一个 Trigger 有机会更新存储之前第二个就会试图执行 Job，所以数据有可能会出错。甚至可能的话还会第二个 Trigger 先于第一个执行完成(依赖于你的 Job 所做的事情)，这时就可能会出现奇怪的结果。

因为这些区别，在你使用 StatefulJob 时可要谨慎了。当你需要避免并发执行一个 Job 时，那么有状态 Job 就是你最简单的筹码了。在 J2EE 的世界里，有状态一词已经引起了一些负面影响，但对于 Quartz 却非如此。

第四章. 部署 Job (第三部分)

5. 易失性、持久性和可恢复性

这三个属性有些类似的，由于它们影响的都是 Job 的运行时行为。我们下面依次讨论它们。

·Job 的易失性

一个易失性的 Job 是在程序关闭之后不会被持久化。一个 Job 是通过调用 JobDetail 的 setVolatility(true) 被设置为易失性的。

> **当你需要持久化 Job 时不应使用 RamJobStore**
>
> RamJobStore 使用的是非永久性存储器，所有关于 Job 和 Trigger 的信息会在程序关闭之后丢失。保存 Job 到 RamJobStore 有效的使得它们是易失性的。假如你需要让你的 Job 信息在程序重启之后仍能保留下来，你就该考虑另一种 JobStore 类型，比如 JobStoreTX 或者 JobStoreCMT。它们会在第六章“作业存储与持久化”中讲到。

Job 易失性的默认值是 false.

·Job 持久性

一个持久的 Job 是它应该保持在 JobStore 中的，甚至是在没有任何 Trigger 去触发它的时候。我们说，你设置了一个单次触发的 Trigger，触发之后它就变成了 STATE_COMPLETE 状态。Job 执行一次后就不再被触发了，这个 Trigger 部署之后只为了执行一次。这个 Trigger 指向的 Job 现在成了一个孤儿 Job，因为不再有任何 Trigger 与之相关联了。

假如你设置一个 Job 为连续性的，即使它成了孤儿 Job 也不会从 JobStore 移除掉。这样可以保证在将来，无论何时你的程序决定为这个 Job 增加另一个 Trigger 都是可用的。假如调用了 JobDetail 的 setDurability(false) 方法，那么在所有的触发器触发之后 Job 将从 JobStore 中移出。连续性的默认值是 false。因此，Job 和 Trigger 的默认行为是：当 Trigger 完成了所有的触发、Job 在没有 Trigger 与之关联时它们就会从 JobStore 中移除。

·Job 的可恢复性

当一个 Job 还在执行中，Scheduler 经历了一次非预期的关闭，在 Scheduler 重启之后可恢复的 Job 还会再次被执行。这个 Job 会再次重头开始执行。Scheduler 是没法知道在程序停止的时候 Job 执行到了哪个位置，因此必须重新开始再执行一遍。

要设置 Job 为可恢复性，用下面的方法：

```
1.  public void setRequestsRecovery(boolean shuldRecover);
```

默认时，这个值为 false，Scheduler 不会试着去恢复 job 的。

·从 Scheduler 中移除 Job

你可用多种方式移除已部署的 Job。一种方式是移除所有与这个 Job 相关联的 Trigger；如果这个 Job 是非持久性的，它将会从 Scheduler 中移出。一个更简单直接的方式是使用 deleteJob() 方法：

```
1.  public boolean deleteJob(String jobName, String groupName)
2.  throws SchedulerException;
```

·中断 Job

有时候需要能中断一个 Job，尤其是对于一个长时间执行的 Job。例如，假定你有一个 Job 运行过程要花费一个小时，你发现在 5 分钟的时候因某个非受控的错误被中断需要接着执行。你或许也会中断 Job，修复问题，然后又继续运行。

Quartz 包括一个接口叫做 org.quartz.InterruptableJob，它扩展了普通的 Job 接口并提供了一个 interrupt() 方法：

```
public void interrupt() throws
UnableToInterruptJobException;
```

可以提供 Job 部署时所用的 Job 的名称和组名调用 Scheduler 的 interrupte() 方法：

```
public boolean interrupt(SchedulingContext ctxt, String
jobName, String groupName) throws
UnableToInterruptJobException;
```

代码 4.8 就是一个叫做 ChedkForInterruptJob 的 InterruptableJob 例子。

Scheduler 接着会调用你的 Job 的 interrupt() 方法。这时就取决于你和你的 Job 决定如何中断 Job 了。Quartz 有几种如何处理中断的方式，代码 4.8 中提供的是比较通用的。Quartz 框架可向 Job 发出中断请求的信号，但此时是你在控制着 Job，因此需要你为中断信号作出响应。

代码 4.8. InterrupableJob 能用来决定是否调用了 Scheduler 的 interrupte()

```
public class CheckForInterruptJob implements InterruptableJob {
     static Log logger = LogFactory.getLog(CheckForInterruptJob.class);

     private boolean jobInterrupted = false;

     private int counter = 5;

     private boolean jobFinished = false;

     public void interrupt() throws UnableToInterruptJobException {
          jobInterrupted = true;
     }

     public void execute(JobExecutionContext context)
               throws JobExecutionException {

          while (!jobInterrupted && !jobFinished) {

               // Perform a small amount of processing
               logger.info("Processing the job");
               counter--;

               if (counter <= 0) {
                    jobFinished = true;
                    }

               // Sleep and wait for 3 seconds
               try {
                    Thread.sleep(3000);
               } catch (InterruptedException e) {
                    // do nothing
               }
          }
          if (jobFinished) {
               logger.info("Job finished without interrupt");
          } else if (jobInterrupted) {
               logger.info("Job was interrupted");
          }
```

```
39.         }
40.     }
```

·框架所提供的 Job

Quartz 框架提供了几个可在你的应用程序中轻松使用的 Job，表 4.1 列出了那种 Job 及用法

表 4.1. Quartz 框架所提供的 Job

Job 类	Job 用法
org.quartz.jobs.FileScanJob	检查某个指定文件是否变化，并在文件被改变时通知到相应监听器的 Job
org.quartz.jobs.FileScanListener	在文件被修改后通知 FileScanJob 的监听器
org.quartz.jobs.NativeJob	用来执行本地程序(如 windows 下 .exe 文件) 的 Job
org.quartz.jobs.NoOpJob	什么也不做，但用来测试监听器不是很有用的。一些用户甚至仅仅用它来导致一个监听器的运行
org.quartz.jobs.ee.mail.SendMailJob	使用 JavaMail API 发送 e-mail 的 Job
org.quartz.jobs.ee.jmx.JMXInvokerJob	调用 JMX bean 上的方法的 Job
org.quartz.jobs.ee.ejb.EJBInvokerJob	用来调用 EJB 上方法的 Job

6. 快速了解Java 线程

本章很短小却很有必要暂时从对 Quartz 框架的讨论中转移到这个话题来。线程在 Quartz 框架中扮演着一个很重要的角色。要是没有线程，Java(以及 Quartz) 就不得不使用重量级的进程来执行并发的任务(Job，对于 Quartz 来说)。这个材料对于已经理解了 Java 中线程如何工作的人来说是很基本的。假如你是这类人，请宽容一下。假如你还没有机会去学习有关 Java 线程的知识，现在是个相当好的时机去快速了解它。尽管主要是关注于 Java 线程的讨论，我们在后面部份会由此进一步讨论线程在 Quartz 是如何运用的。

·Java 中的线程

线程允许程序同一时间做很多任务，至少，看起来那些任务是并发执行的。本章的讨论不考虑并行处理的情形，在任一特定时刻仅有一个线程在执行，但是 CPU 给每个线程一小片时间运行(通过时间片)然后来回在线程间快速的切换。这就是我们所看到的多线程运行的样子。

Java 语言使用 Thread 类内建了对线程的支持。当一个线程被通知运行，该线程的 run() 方法就被执行。正是因为你创建了一个线程实例并且调用的是 start() 方法，所以并不意意味着相应的 run() 方法会得到立即执行；这个线程实例必须等待，直到 JVM 告诉它可以运行。

·线程的生命周期

线程在它的生命周期中会是几种可能的状态中的一种。在同一时刻只能处于一种状态，这些状态是由 JVM 指示的，而不是操作系统。线程状态列示如下：

·新建

·可运行

·阻塞

·等待中

·指定时间的等待

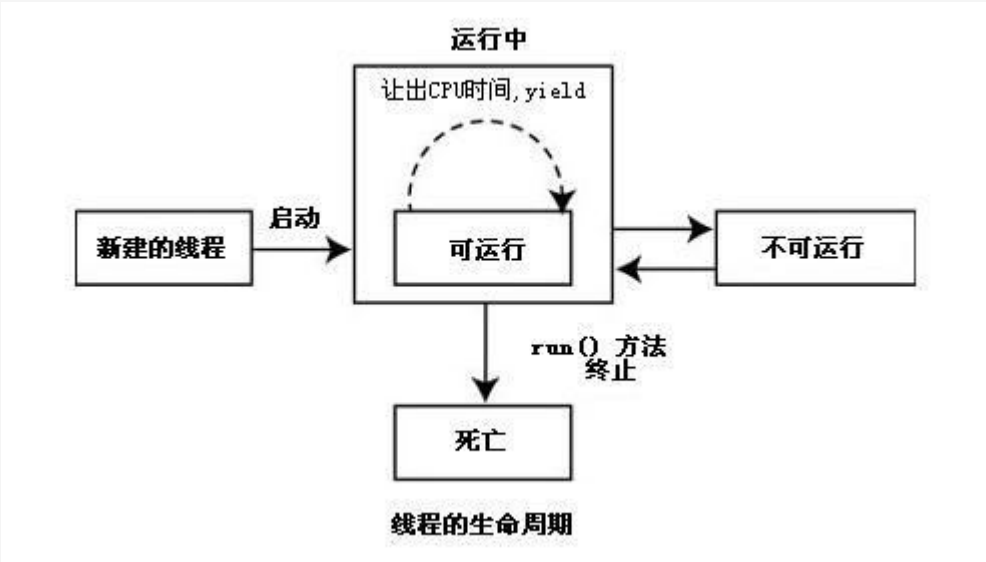
·结束

当一个新的线程被创建后，它就被指定为新建状态。线程在此状态时不会被分配任何系统资源。要使线程转到可运行状态，必须调

用 start() 方法。

当调用了新建线程的 start() 方法，线程进入到就绪状态并被认为是运行中的。这不代表这个线程实际正在执行指令。即使线程的状态被设置为可运行，已安排去运行，它或许不得不等到其他执行中的线程结束或被临时挂起。JVM 来决定哪个线程该获得下次运行的机会。JVM 决定下个线程的行为依赖于数个因素，包括操作系统、特定的 JVM 实现和线程优先级及其他因素。图 4.4 显示了 Java 线程的生命周期。

图 4.4. Java 线程的生命周期



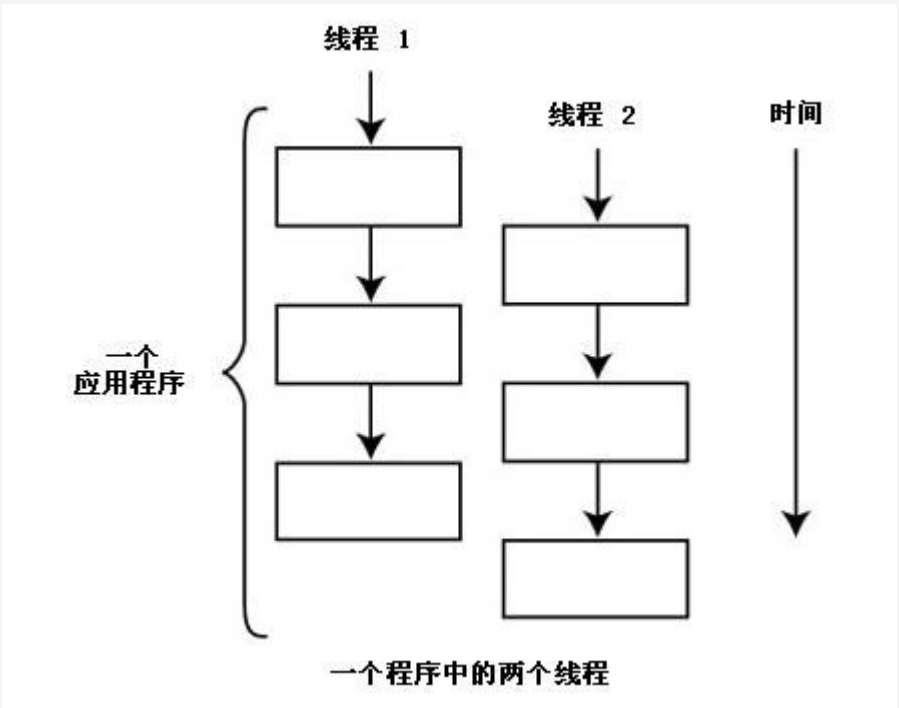
阻塞和等待状态对于本书来说讨论的余地不大，也是要好分几个主题才能讲清，我们在此不必对此深入。假如你想要关于这些主题更多信息的话，你可以看 Sun 站点的 Java 指南关于线程的内容，
http://java.sun.com/docs/books/tutorial/essential/threads/index.html。

·进程和线程

每一个 Java 程序被指派一个进程。程序接着把任务分派到线程中。甚至是你在写一个仅包含一个 main() 方法的 "Hello, World" 程序，程序仍然要使用线程，虽然只有一个。

有时候，所谓“轻量级进程”指的就是线程。这与实际的操作系统进程“重量级”一词形成对比。假如你正在使用 Windows 系统，你可以在任务管理器中看到运行着的重量级进程，但是你看不到线程。图 4.5 描绘了一个 Java 程序中的多线程如何操作。

图 4.5. 运行在一个程序中的两个线程



从图 4.5 中看到，线程通常不会并发着运行的。这里还是先排除多 CPU 和并行处理的情况的话，同一时刻只能有一个线程才能得到 CPU 的参与执行。每一个 JVM 和操作系统对此处理方式可能有所不同，然而，有时，一个线程可能要执行完成后，另一线程

才能接着运行。这就是所知的“非抢先式”。其他的，一个线程中途被打断来让其他线程做它们的事。这被称做“抢先式”。一般的，线程必须完成运行后另一线程才能获得 CPU，这种对 CPU 资源的竞争通常是基于线程优先级的。

·理解和使用线程优先级

我们已多次提到，可运行线程是否能获得下次运行机会是由 JVM 来决定的。JVM 支持的一种知名的调度方案“固定优先级调度”，它调度线程是基于它们与其他可运行线程优先级相比较而定的。

优先级是一个指定给新线程的整数值，它还应参考创建它的父线程的优先级。这个值从 Thread.MIN_PRIORITY (等于 1)，到 Thread.MAX_PRIORITY (等于 10)。这些常量值在 java.lang.Thread 类中有定义。

这个值越大 (最大到 MAX_PRIORITY)，所具有的优先级就越高。除非特别指定，Java 程序中多数刚创建的线程运线在 Thread.NORMAL_PRIORITY (恰好是个中值 5) 级别上。你可能用 setPriority() 方法来修改线程的优先级。

通常，一个运行着的线程持续运行直到下面情况中的一个：

·线程让出 CPU (可能是调用了它的 sleep() 方法、yield() 方法或者是 Object.wait() 方法)。

·线程的 run() 结束了。

·OS 支持时间片的话，它的时间到了

·另一高优先级的线程变为了可运行状态

·守护线程

守护线程有时指的是服务线程，因为他们运行在很低的优先级别上(在后台)，完成些非紧急但却本质的任务。例如，Java 的垃圾回收器就是一个守护线程的例子。这个线程运行在后台，查找并回收程序不再使用的内存。

你可以通过传递 true 给 setDaemon() 方法使线程成为守护线程。不然，这个线程就是一个用户线程。你仅能在线程启动之前把它设置为守护线程。守护线程有点特别的就是假如只有守护线程在运行而没有活着的非守护线程，此时 JVM 仍然是存在的。

· Java 线程组和线程池

Java 的 ThreadGroup 由 java.lang.ThreadGroup 类实现的，描绘的是一组行为如单一实体般的线程。每一个 Java 线程都是线程组的成员。在一个线程组中，你可以停止或启动所有的线程组成员。你还可以设置线程优先级和执行其他线程的通用功能。线程组对于构建像 Quartz 那样的多线程程序是非常有用的。

当一个 Java 程序启动后，它就创建了一个叫做 main 的 ThreadGrop。除非特别指定，所有创建的线程都会成为这个组的成员。当一个线程被指定到某个 ThreadGroup 时，它才会被改变。

·Java 线程池 (ThreadPool)

Java 1.5 引入一个新的概念到 Java 语言中，称之线程池。第一眼看来，这有些像是线程组，但是它实际是用于不同目的的。尽量多个线程能从属于一个相同的 ThreadGroup 中，组是享用不到典型的池化资源带来的好处的。这就是说，线程组仅仅是用于对成员的组织。

线程池是可共享的资源，是一个能被用来执行任务的可管理的线程集合。线程池(通常讲的资源池) 比非池化资源提供了几点好处。首先也是首要的，资源池通过减少过度的对象创建改善了性能。假如你实例化十个线程，并重复的使用它们，而不是每次需要一个都创建一个新的线程，你将会改善程序的性能。这个概念在 Java 和 J2EE 领域中比比皆是。另外的优点包括能限制资源的数量，这有助于程序的稳定性和可扩展性。这是一个非常有意义的特征，而不管你所用的是什么语言，或是什么程序。

第四章. 部署 Job (第四部分)

七. 线程在 Quartz 中的用法

线程与 Quartz 来说尤为重要，因为 Quartz 就是设计为支持同时运行多个 Job。为达到此效果，Quartz 非常倚重于内建于 Java 语言的线程，借助于自己的类和借口还有所增强。你已经在本章或前面章节中看到过这方面的例子。

当 Quartz Schduler 首次由某个工厂方法创建时，工厂配置了 Scheduler 会在它的整个生命周期中用到的几个重要的资源。其中一些重要的资源是与线程相关的。

·主处理线程：QuartzSchedulerThread

Quartz 应用第一次运行时，main 线程会启动 Scheduler。QuartzScheduler 被创建并创建一个 org.quartz.core.QuartzSchedulerThread 类的实例。QuartzSchedulerThread 包含有决定何时下一个 Job 将被触发的处理循环。顾名思义，QuartzSchedulerThread 是一个 Java 线程。它作为一个非守护线程运行在正常优先级下。

QuartzSchedulerThread 的主处理循环的职责描述如下：

1. 当 Scheduler 正在运行时：

   A. 检查是否有转换为 standby 模式的请求。

      1. 假如 standby 方法被调用，等待继续的信号

   B. 询问 JobStore 下次要被触发的 Trigger.

      1. 如果没有 Trigger 待触发，等候一小段时间后再次检查

2. 假如有一个可用的 Trigger，等待触发它的确切时间的到来

   D. 时间到了，为 Trigger 获取到 triggerFiredBundle.

   E. 使用 Scheduler 和 triggerFiredBundle 为 Job 创建一个 JobRunShell 实例

   F. 告诉 ThreadPool 可能时运行 JobRunShell.

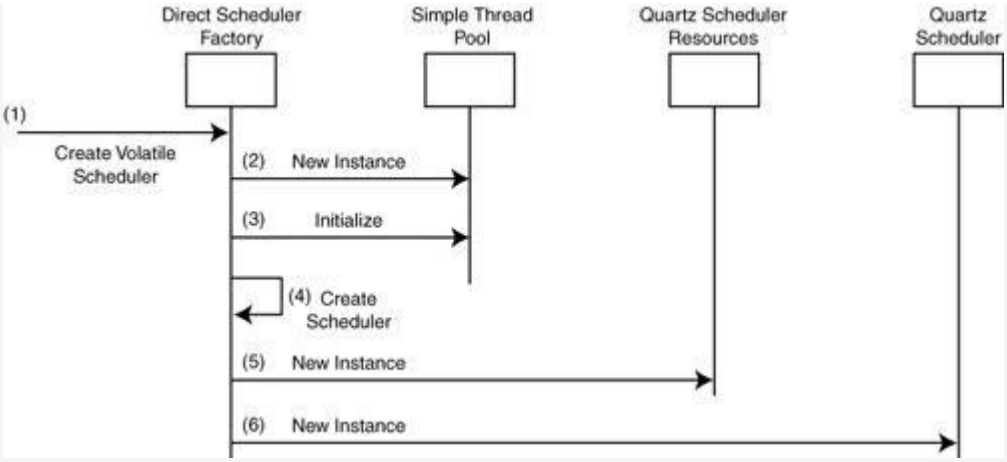
这个逻辑存在于 QuartzSchedulerThread 的 run() 方法中。

·QuartzSchedulerResources

当工厂创建 Scheduler 实例时，它还会传递给 Scheduler 一个 org.quartz.core.QuartzSchedulerResoures 实例。QuartzSchedulerResourecs 除包含以上东西之后，还有一个 ThreadPool 实例，它提供了一个工作者线程池来负责执行 Job。在 Quartz 中，ThreadPool 是由 org.quartz.spi.ThreadPool 接口 (因为 Quartz 是在 JDK 1.5 之前产生的，所以需要自己的 ThreadPool 类确保向后的兼容性，Quartz 仍然用自己的 ThreadPool 替代 Java 的) 表示的，并提供一个名为 org.quartz.simp.SimpleThreadPool 的具体实现类。SimpleThreadPool 有一个固定数目的工作者线程，在加载之后就不再减少或增多。图 4.6 是在框架启动时与线程相关的时序图。

图 4.6. 在 Quartz 启动时创建的几个与线程相关的资源

[看全尺寸图]



·什么是 Quartz 工作者线程？

Quartz 不会在 main 线程中处理你的 Job。如果这么做，会严重降低应用的可扩展性。相应的，Quartz 把线程管理的职责委托给分散的组件。对于一般的 Quartz 设置 (这部分还是很费功夫的)，都是用SimpleThreadPool 类处理线程的管理。SimpleThreadPool 创建了一定数量的 WorkerThread 实例来使得 Job 能够在分散的线程中进行处理。WorkerThread 是定义在 SimpleThreadPool 类中的内部类，它实质上就是一个线程。要创建 WorkerThread 的数量以及为他们的优先级是配置在文件 quartz.properties 中并传入工厂的。

当 QuartzSchedulerThread 请求 ThreadPool 去运行 JobRunShell 实例，ThreadPool 就检查看是否有一个可用的工作者线程。假如所以已配置的工作者线程都是忙的，ThreadPool 就等待直到有一个变为可用。当一个工作者线程是可用的，并且有一个 JobRunShell 等待执行，工作者线程就会调用 JobRunShell 类的 run() 方法。

> **配置可选择的 ThreadPool**
>
> Quartz 框架允许你改变所用的 ThreadPool 实现。替换类必须实现 org.quartz.spi.ThreadPool 接口，但是框架只支持通过在文件中配置的方式改变 ThreadPool 的实现类。例如，你可以使用更为高级的 ThreadPool 实现——随时基于需求改变线程的数量，甚至是从应用服务器中获得工作者线程。对于大多数用户，默认的实现就足够了。

·JobRunShell 的 run() 方法

虽然 WorkerThread 是真正的 Java 线程，JobRunShell 类也还是实现了 Runable。那意味着它可以作为一个线程并包含一个 run() 方法。在本章前面讨论过，JobRunShell 的目的是调用 Job 的 execute() 方法。不仅如此，它还要负责通知 Job 和 Trigger 监听器，在运行完成后还得更新此次执行的 Trigger 的信息。

八. 理解 Quartz 的 Trigger

Job 包含了要执行任务的逻辑，但是 Job 对何时该执行却一无所知。这个事情留给了 Trigger。Quartz Trigger 继承了抽象的 org.quartz.Trigger 类。当前，Quartz 有三个可用的 Trigger：

- ·org.quartz.SimpleTrigger
- ·org.quartz.CronTrigger
- ·org.quartz.NthIncludeDayTrigger

还有第四个 Trigger，叫做 UICronTrigger，但是到 Quartz 1.5 就不推荐使用了。它主要是用在 Quartz 的 Web 程序中，而不是用于 Quartz 自身中。[译者注：在 Quartz 1.6 中，UICronTrigger 已被彻底摈弃]

·使用 org.quartz.SimpleTrigger

正如其名所示，SimpleTrigger 对于设置和使用是最为简单的一种 Quartz Trigger。它是为那种需要在特定的日期/时间启动，且以一个可能的间隔时间重复执行 *n* 次的 Job 所设计的。代码 4.9 提供了一个使用 SimpleTrigger 的例子。

代码 4.9. 使用 SimpleTrigger 部署一个 Job

```
public class Listing_4_9 {
     static Log logger = LogFactory.getLog(Listing_4_9.class);

     public static void main(String[] args) {
          Listing_4_9 example = new Listing_4_9();
          example.startScheduler();
     }

     public void startScheduler() {
          try {
               // Create and start the scheduler
               Scheduler scheduler =
                              StdSchedulerFactory.getDefaultScheduler();
               scheduler.start();
              logger.info("Scheduler has been started");

               JobDetail jobDetail =
                         new JobDetail("PrintInfoJob",
                              Scheduler.DEFAULT_GROUP,
                                     PrintInfoJob.class);
               /*
                * Create a SimpleTrigger that starts immediately,
                * with a null end date/time, repeats forever and has
                * 1 minute (60000 ms) between each firing.
                */
                Trigger trigger =
                               new SimpleTrigger("myTrigger",
                                      Scheduler.DEFAULT_GROUP, new Date(), null,
                                             SimpleTrigger.REPEAT_INDEFINITELY,
                                             60000L);

                scheduler.scheduleJob(jobDetail, trigger );

           } catch (SchedulerException ex) {
                logger.error(ex);
           }
      }
  }
```

SimpleTrigger 存在几个变种的构造方法。他们是从无参的版本一路到带全部参数的版本。下面代码版断显示了一个仅带有 trigger 的名字和组的简单构造方法

```
//No Argument Constructor
SimpleTrigger sTrigger =
   new SimpleTrigger("myTrigger", Scheduler.DEFAULT_GROUP);
```

这个 Trigger 会立即执行，而不重复。还有一个构造方法带有多个参数，配置 Triiger 在某一特定时刻触发，重复执行多次，和两次触发间的延迟时间。

```
public SimpleTrigger(String name, String group,
   String jobName, String jobGroup, Date startTime,
   Date endTime, int repeatCount, long repeatInterval);
```

·使用 org.quartz.CronTrigger

CronTrigger 允许设定非常复杂的触发时间表。然而有时也许不得不使用两个或多个 SimpleTrigger 来满足你的触发需求，这时候，你仅仅需要一个 CronTrigger 实例就够了。

顾名思义，CronTrigger 是基于 Unix 类似于 cron 的表达式。例如，你也许有一个 Job，要它在星期一和星期五的上午 8:00-9:00 间每五分钟执行一次。假如你试图用 SimpleTrigger 来实现，你或许要为这个 Job 配置多个 Trigger。然而，你可以使用如下的表达式来产生一个遵照这个时间表触发的 Trigger：

"0 0/5 8 ? * MON,FRI"

```
try {
     CronTrigger cTrigger = new CronTrigger("myTrigger",
               Scheduler.DEFAULT_GROUP, "0 0/5 8 ? *
MON,FRI");
} catch (ParseException ex) {
ex.printStackTrace();
}
```

因为 CronTrigger 内建的如此强的灵活性，也与生俱来可用于创建几乎无所限制的表达式，所以下一章专注于你想知道的关于 CronTrigger 的东西及 cron 表达式。第五章，"CronTrigger 及更多" 也列了一系列的关于如何为特定触发时间表创建的 CronTrigger 的例子。

·使用 org.quartz.NthIncludedDayTrigger

org.quartz.NthIncludedDayTrigger 是 Quartz 开发团队最新加入到框架中的一个 Trigger。它设计用于在每一间隔类型的第几天执行 Job。例如，你要在每个月的 15 号执行开票的 Job，用 NthIncludedDayTrigger 就再合适不过了。Quartz 的 Caldendar 也可与 Trigger 关联以此把周末与节假日考虑进来，并在必要时跳开这些日期。接下来的代码片断描绘了如何创建一个 NthIncludedDayTrigger.

```
NthIncludedDayTrigger trigger =
     new NthIncludedDayTrigger(
          "MyTrigger", Scheduler.DEFAULT_GROUP);
               trigger.setN(15);
trigger.setIntervalType(
     NthIncludedDayTrigger.INTERVAL_TYPE_MONTHLY);
```

·为一个 Job 使用多个 Trigger

你并未被强制要接受单个 Trigger 每个 Job。如果你需要一个更复杂的触发计划，你可以创建多个 Trigger 并指派它们给同一个 Job。Scheduler 是基于配置在 Job 上的 Trigger 来决定正确的执行计划的。为同一个 JobDetail 使用多个 Trigger 方法片断如下：

```
try {
     // Create and start the scheduler
     Scheduler scheduler =
          StdSchedulerFactory.getDefaultScheduler();
     scheduler.start();
     logger.info("Scheduler has been started");

     JobDetail jobDetail =
          new JobDetail("PrintInfoJob",
               Scheduler.DEFAULT_GROUP,
               PrintInfoJob.class);

     // A trigger that fires every 5 seconds
```

```
14.      Trigger trigger1 =
15.           TriggerUtils.makeSecondlyTrigger("trigger1",
16.                5000, SimpleTrigger.REPEAT_INDEFINITELY);
17.
18.      // A trigger that fires every 10 minutes
19.      Trigger trigger2 =
20.           TriggerUtils.makeMinutelyTrigger("trigger2", 10,
21.                SimpleTrigger.REPEAT_INDEFINITELY);
22.
23.      // Schedule job with first trigger
24.      scheduler.scheduleJob(jobDetail, trigger1);
25.
26.      // Schedule job with second trigger
27.      scheduler.scheduleJob(jobDetail, trigger2);
28.
29.    } catch (SchedulerException ex) {
30.         logger.error(ex);
31.    }
```

> **每 Trigger 一个 Job**
>
> 虽然单个 JobDetail 能够支持多个 Trigger，但一个 Trigger 只能被指派给一个 Job。

·Quartz Calendar

不要混淆了 Quartz 的 Calendar 对象与 Java API 的 java.util.Calendar。它们是应用于不同目的不一样的组件。正如你大概所知，Java 的 Calender 对象是通用的日期和时间工具；许多过去由 Java 的 Date 类提供的功能现在加到了 Calendar 类中了。

另一方面，Quartz 的 Calender 专门用于屏闭一个时间区间，使 Trigger 在这个区间中不被触发。例如，让我们假如你是为一个财务机构（如银行）工作。对于银行普遍的都有许多 "银行节日"。假设你不需要（或不想） Job 在那些日子里运行。你可以采用以下几种方法中的一种来实现：

·让你的 Job 总是运行。(这会让银行一团糟)

·在节假日期间手动停止你的 Job。(需要专门的人来负责做这个事情)

·创建不包含这些日子的多个 Trigger。(这对于设置和维护会较耗时的)

·设立一个排除这些日子的银行节日的 Calendar。(很轻松的实现)

虽然你可以用其中的一个方法来解决这样的问题，Quartz 的 Calender 却是特意为此而设计的。

·org.quartz.Calender 接口

Quartz 定义了 org.quartz.Calendar 接口，所有的 Quartz Calendar 必须实现它。它包含了几个方法，但是有两个是最重要的：

```
1.  public long getNextIncludedTime(long timeStamp);
2.  public boolean isTimeIncluded(long timeStamp);
```

> **Calender 排除时间的粒度**
>
> Calendar 接口方法参数的类型是 Long。这说明 Quartz Calender 能够排除的时间细致毫秒级。你很可能永远都不需要这么细小的位度，因为大部分的 Job 只需要排除特别的日期或许会是小时。然而，假如你真需要排除到毫秒一级的，Calender 能帮你做到。

作为一个 Quartz Job 的创建者和开发者，你可不必去熟悉 Calender 接口。这主要是因为已有的 Calendar (Quartz 自带的) 需要应付的情况就不够多。开箱即用的，Quartz 包括许多的 Calender 实现足以满足你的要求。表 4.1 列出了 Quartz 自带的随时可用的 Calendar。

表 4.1. Quartz 包含了你的应用可用的许多的 Calender 类型

Calender 名称	类	用法
BaseCalender	org.quartz.impl.calendar.BaseCalender	为高级的 Calender 实现了基本的功能，实现了 org.quartz.Calender 接口
WeeklyCalendar	org.quartz.impl.calendar.WeeklyCalendar	排除星期中的一天或多天，例如，可用于排除周末
MonthlyCalendar	org.quartz.impl.calendar.MonthlyCalendar	排除月份中的数天，例如，可用于排除每月的最后一天
AnnualCalendar	org.quartz.impl.calendar.AnnualCalendar	排除年中一天或多天
HolidayCalendar	org.quartz.impl.calendar.HolidayCalendar	特别的用于从 Trigger 中排除节假日

·使用 Quartz 的 Calendar

要使用 Quartz Calendar，你只需简单的实例化，并加入你要排除的日期，然后用 Scheduler 注册它。最后把这个 Calender 实例与你想要使用该 Calender 的每一个 Trigger 实例关联起来。

> **多个 Job 共同一个 Calendar**
>
> 你不能仅仅是把 Calendar 加入到 Scheduler 来为所有 Job 安排 Calendar。你需让 Calendar 实例关联到每一个 Trigger。Calender 实例被加到 Scheduler 中后，它只允许由使用中的 JobStore 存储；你必须让 Calender 依附于 Trigger 实例。

代码 4.10 显示的是一个使用 Quartz AnnualCalender 类执行银行节日的例子。

代码 4.10. 使用 AnnualCalender 来排除银行节日

```
1.  public class Listing_4_10 {
2.       static Log logger = LogFactory.getLog(Listing_4_10.class);
3.
4.       public static void main(String[] args) {
5.            Listing_4_10 example = new Listing_4_10();
6.            example.startScheduler();
7.       }
8.
9.       public void startScheduler() {
10.           try {
11.               // Create and start the scheduler
12.               Scheduler scheduler =
13.                                 StdSchedulerFactory.getDefaultScheduler();
14.               scheduler.start();
15.
16.               scheduleJob(scheduler, PrintInfoJob.class);
17.
18.               logger.info("Scheduler starting up...");
19.               scheduler.start();
20.
21.           } catch (SchedulerException ex) {
22.               logger.error(ex);
23.           }
24.      }
25.
26.      private void scheduleJob(Scheduler scheduler, Class jobClass) {
27.           try {
28.               // Create an instance of the Quartz AnnualCalendar
29.               AnnualCalendar cal = new AnnualCalendar();
30.
31.               // exclude July 4th
32.               Calendar gCal = GregorianCalendar.getInstance();
```

```
                gCal.set(Calendar.MONTH, Calendar.JULY);
                gCal.set(Calendar.DATE, 4);

                cal.setDayExcluded(gCal, true);

                // Add to scheduler, replace existing, update triggers
                scheduler.
                                addCalendar("bankHolidays", cal, true, true);

                /*
                         * Set up a trigger to start firing now, repeat forever
                         * and have (60000 ms) between each firing.
                 */
                Trigger trigger =
                                TriggerUtils.makeImmediateTrigger("myTrigger",
                       -1,60000);

                // Trigger will use Calendar to exclude firing times
                trigger.setCalendarName("bankHolidays");

                JobDetail jobDetail =
                                new JobDetail(jobClass.getName(),
                           Scheduler.DEFAULT_GROUP, jobClass);

                // Associate the trigger with the job in the scheduler
                scheduler.scheduleJob(jobDetail, trigger);

          } catch (SchedulerException ex) {
                logger.error(ex);
          }
     }
}
```

当你运行代码 4.10 中的例子时，除 7 月 4 号之外，你都可以看到 Job 会执行。作为一个留下来给你做的练习，在方法 scheduleJob() 中改变被排除的日期为你当前的日期。假如你再次跑这段代码，你将会看到当前日期被排除了，下次被触发的时间会是明天了。

> **为什么我们不用 HolidayCalender？**
>
> 你也许会有所疑惑，为什么我们在上个例子中不选择使用 HolidayCalender 呢？HolidayCalender 类考虑的是年份。因此，如果你希望在后续三年中排除 7 月 4 日，你需要把三年中的每个日期都作为要排除的项。而 AnnualCalender 可简单的为每年设定要排除的日期，也就更容易的应用于这种情况

·创建你自己的 Calender

这最后一节演示了创建你自己的 Calender 类是多么的容易。假定你需要一个 Calender 去排除小时当中的某分钟。例如，假定你需要排除每小时中的前 5 分钟或者后 15 分钟。你能创建一个新的 Calender 来支持这种功能。

> **我们也许可以使用 CronTrigger**
>
> 我们也许可以写出一个 Cron 表达式来排除这些时间，但那样就没了创建一个新 Calender 的乐趣了。

代码 4.11 是 HourlyCalender，我们能用它让排除小时中的一些分钟。

代码 4.11. HourlyCalender 能从每小中排除某些分钟

```
public class HourlyCalendar extends BaseCalendar {

```

```
3.      // Array of Integer from 0 to 59
4.      private List excludedMinutes = new ArrayList();
5.
6.      public HourlyCalendar() {
7.           super();
8.      }
9.      public HourlyCalendar(Calendar baseCalendar) {
10.          super(baseCalendar);
11.     }
12.
13.     public List getMinutesExcluded() {
14.          return excludedMinutes;
15.     }
16.
17.     public boolean isMinuteExcluded(int minute) {
18.
19.          Iterator iter = excludedMinutes.iterator();
20.          while (iter.hasNext()) {
21.               Integer excludedMin = (Integer) iter.next();
22.
23.               if (minute == excludedMin.intValue()) {
24.                    return true;
25.               }
26.
27.               continue;
28.          }
29.          return false;
30.     }
31.
32.     public void setMinutesExcluded(List minutes) {
33.          if (minutes == null)
34.               return;
35.
36.          excludedMinutes.addAll(minutes);
37.     }
38.
39.     public void setMinuteExcluded(int minute) {
40.
41.          if (isMinuteExcluded(minute))
42.               return;
43.
44.          excludedMinutes.add(new Integer(minute));
45.     }
46.
47.     public boolean isTimeIncluded(long timeStamp) {
48.
49.          if (super.isTimeIncluded(timeStamp) == false) {
50.               return false;
51.          }
52.
53.          java.util.Calendar cal = getJavaCalendar(timeStamp);
54.          int minute = cal.get(java.util.Calendar.MINUTE);
55.
56.          return !(isMinuteExcluded(minute));
57.     }
58.
59.     public long getNextIncludedTime(long timeStamp) {
60.          // Call base calendar implementation first
61.          long baseTime = super.getNextIncludedTime(timeStamp);
62.          if ((baseTime > 0) && (baseTime > timeStamp))
63.              timeStamp = baseTime;
```

```
64.
65.            // Get timestamp for 00:00:00
66.            long newTimeStamp = buildHoliday(timeStamp);
67.            java.util.Calendar cal = getJavaCalendar(newTimeStamp);
68.            int minute = cal.get(java.util.Calendar.MINUTE);
69.
70.            if (isMinuteExcluded(minute) == false)
71.                 return timeStamp; // return the
72.            // original value
73.
74.            while (isMinuteExcluded(minute) == true) {
75.                 cal.add(java.util.Calendar.MINUTE, 1);
76.            }
77.            return cal.getTime().getTime();
78.       }
79.  }
```

如果你使用 HourlyCalender 去部署 Job，需要你做的事情是设置小时中你希望排除的分钟；由 Calender 和 Scheduler 做剩下的事情。你能看到在代码 4.12 中对 HourlyCalender 的演示。

代码 4.12. HourlyCalender 基于小时中排除某些分钟来执行

```
1.  public class Listing_4_12 {
2.       static Log logger = LogFactory.getLog(Listing_4_12.class);
3.
4.       public static void main(String[] args) {
5.            Listing_4_12 example = new Listing_4_12();
6.            example.startScheduler();
7.       }
8.
9.       public void startScheduler() {
10.           try {
11.                // Create a default instance of the Scheduler
12.                Scheduler scheduler =
13.                          StdSchedulerFactory.getDefaultScheduler();
14.
15.                // Using the NoOpJob, but could have been any
16.                scheduleJob(scheduler, PrintInfoJob.class);
17.
18.                logger.info("Scheduler starting up...");
19.                scheduler.start();
20.
21.           } catch (SchedulerException ex) {
22.                logger.error(ex);
23.           }
24.      }
25.
26.      private void scheduleJob(Scheduler scheduler, Class jobClass) {
27.           try {
28.                // Create an instance of the Quartz AnnualCalendar
29.                HourlyCalendar cal = new HourlyCalendar();
30.                cal.setMinuteExcluded(47);
31.                cal.setMinuteExcluded(48);
32.                cal.setMinuteExcluded(49);
33.                cal.setMinuteExcluded(50);
34.
35.                // Add Calendar to the Scheduler
36.                scheduler.
37.                          addCalendar("hourlyExample", cal, true, true);
38.
```

```
                Trigger trigger =
                          TriggerUtils.makeImmediateTrigger("myTrigger",
                          -1, 10000);

                // Trigger will use Calendar to exclude firing times
                trigger.setCalendarName("hourlyExample");

                JobDetail jobDetail =
                          new JobDetail(jobClass.getName(),
                          Scheduler.DEFAULT_GROUP, jobClass);

                // Associate the trigger with the job in the scheduler
                scheduler.scheduleJob(jobDetail, trigger);

           } catch (SchedulerException ex) {
                logger.error(ex);
           }
      }
 }
```

当你运行 4.12 中的代码，你会看到 PrintInfoJob 在被排除的分钟是不被执行，在方法 setMinuteExcluded() 方法中依你要求改变需排除的分钟，来看看新的 Calender 是如何工作的。

第五章. Cron 触发器及相关内容（第一部分）

第五章. Cron 触发器及相关内容

我们在上章中有承诺过会花更多时间来讲 Quartz 的 CronTrigger，所以不会让你失望的。SimpleTrigger 对于需要在指定的毫秒处及时执行的作业还是不错的，但是假如你的作业需要更复杂的执行计划时，你也就要 CronTrigger 给你提供更强更灵活的功能。

一. Cron 的快速一课

cron 这一观念是来自于 UNIX 世界。在 UNIX 中，cron 是一个运行于后台的守护程序，它负责所有基于时间的事件。尽管 Quartz 除相同的名字和相似的表达式语法外，并未分享到 UNIX cron 别的东西，我们还是值得花几个段落去理解 cron 背后的历史。我们这里的目标不是搞混 UNIX cron 表达式和 Quartz 的 cron 表达式，但是你应该了解 Quartz 表达式的历史，并探索为什么他们运作起来很像。这儿明显有许多有意图的相似性。

> **有许多不同版本的 UNIX Cron**
>
> 你会发现不同版本的 cron，每一种都有些微地差异。我们仅着眼于与 Quartz CronTrigger 的比较，因此我们只讨论不同版本 UNIX cron 共性的东西。

UNIX cron 守护进程每隔一分钟被唤醒一次去检查叫做 *crontabs* 的配置文件。(*Crontab* 是 CRON 和 TABLE 的连写，其中配置有 cron 守护进程的作业列表和其他的指令。)守护进程检查存储在 crontabs 中的命令并决定是否有要执行的任务。

·UNIX Cron 的格式

你可以认为 UNIX crontab 是 Trigger 和 Job 的组合，因为它们同时列出来执行计划和要执行的命令(job)。

Cron Expression 的格式

crontab 格式包含六段，前五段为执行计划，第六段为要执行的命令。(Quartz cron 表达式有七段。) 下面这些是执行计划的五个字段：

- ·分 (00-59)
- ·时 (00-23)
- ·日 (1-31)
- ·月 (1-12)
- ·周 (0-6 或 sun-sat)

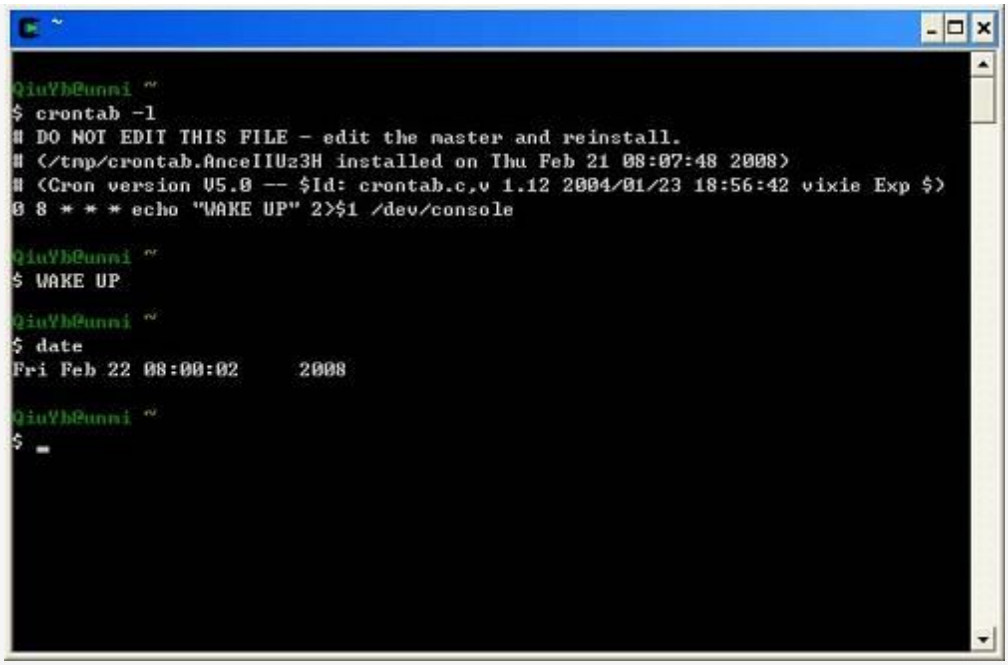
UNIX cron 格式表达式中允许出现一些特殊的字符，例如星号(*)，它用来匹配所有值。这作有一个 UNIX crontab 的例子：

```
0 8 * * * echo "WAKE UP" 2>$1 /dev/console
```

这一 crontab 条目在每天早上8点打印字符串 "WAKE UP" 到 UNIX 的设置 /dev/console 上。图 5.1 显示了这个动作。

图 5.1. UNIX Cron 执行 0 8 * * * echo "WAKE UP" 2>$1 /dev/console 表达式

[点击看全图]



2. 使用 Quartz CronTrigger

在现实世界里，作业计划通常比 SimpleTrigger 能支持的复杂得多。CronTrigger 就可用于指定非常复杂的计划，这种计划不错，因为也是我们发现需要这么做的。在我们深入到 CronTrigger 之前，让我们先看一个例子。代码 5.1 所示的是一个使用 CronTrigger (连同一个 Quartz cron 表达式) 来部署前面例子中的 PrintInfoJob。代码中的大部分内容与前面章节的例子相同。唯一不同点是我们使用了 CronTrigger 替代了 SimpleTrigger。正因为如此，我们不得不为它提供了一个 cron 表达式。

代码 5.1. 简单使用 CronTrigger 来部署一个 Job

```
1.  public class Listing_5_1 {
2.       static Log logger = LogFactory.getLog(Listing_5_1.class);
3.
4.       public static void main(String[] args) {
5.            Listing_5_1 example = new Listing_5_1();
6.            example.runScheduler();
7.       }
8.
9.       public void runScheduler() {
10.           Scheduler scheduler = null;
11.
12.           try {
13.               // Create a default instance of the Scheduler
14.               scheduler = StdSchedulerFactory.getDefaultScheduler();
15.               scheduler.start();
16.               logger.info("Scheduler was started at " + new Date());
17.
18.               // Create the JobDetail
19.               JobDetail jobDetail =
20.                            new JobDetail("PrintInfoJob",
21.                       Scheduler.DEFAULT_GROUP,
22.                                    PrintInfoJob.class);
23.
24.               // Create a CronTrigger
25.               try {
26.                    // CronTrigger that fires @7:30am Mon - Fri
27.                    CronTrigger trigger = new
28.                                    CronTrigger("CronTrigger", null,
29.                               "0 30 7 ? * MON-FRI");
30.
31.                    scheduler.scheduleJob(jobDetail, trigger);
32.               } catch (ParseException ex) {
```

```
                    logger.error("Error parsing cron expr", ex);

                }


            } catch (SchedulerException ex) {
                logger.error(ex);
            }
        }
    }
```

代码 5.1 中的例子使用了如下 cron 表达式：

0 30 7 ? * MON-FRI

当被调度器解释后，它会引起 Trigger 在星期一至星期五的早上 7:30 被激发。让我们来看看 Quartz CronTrigger 的 cron 表达式的格式。

第五章. Cron 触发器及相关内容（第二部分）

三. cron 表达式的格式

Quartz cron 表达式的格式十分类似于 UNIX cron 格式，但还是有少许明显的区别。区别之一就是 Quartz 的格式向下支持到秒级别的计划，而 UNIX cron 计划仅支持至分钟级。许多我们的触发计划要基于秒级递增的(例如，每45秒)，因此这是一个非常好的差异。

在 UNIX cron 里，要执行的作业（或者说命令）是存放在 cron 表达式中的，在第六个域位置上。Quartz 用 cron 表达式存放执行计划。引用了 cron 表达式的 CronTrigger 在计划的时间里会与 job 关联上。

另一个与 UNIX cron 表达式的不同点是在表达式中支持域的数目。UNIX 给出五个域(分、时、日、月和周)，Quartz 提供七个域。表 5.1 列出了 Quartz cron 表达式支持的七个域。

表 5.1. Quartz Cron 表达式支持到七个域

名称	是否必须	允许值	特殊字符
秒	是	0-59	, - * /
分	是	0-59	, - * /
时	是	0-23	, - * /
日	是	1-31	, - * ? / L W C
月	是	1-12 或 JAN-DEC	, - * /
周	是	1-7 或 SUN-SAT	, - * ? / L C #
年	否	空 或 1970-2099	, - * /

月份和星期的名称是不区分大小写的。FRI 和 fri 是一样的。

域之间有空格分隔，这和 UNIX cron 一样。无可争辩的，我们能写的最简单的表达式看起来就是这个了：

```
* * * ? * *
```

这个表达会每秒钟(每分种的、每小时的、每天的)激发一个部署的 job。

·理解特殊字符

同 UNIX cron 一样，Quartz cron 表达式支持用特殊字符来创建更为复杂的执行计划。然而，Quartz 在特殊字符的支持上比标准 UNIX cron 表达式更丰富了。

* 星号

使用星号(*) 指示着你想在这个域上包含所有合法的值。例如，在月份域上使用星号意味着每个月都会触发这个 trigger。

表达式样例：

```
0 * 17 * * ?
```

意义：每天从下午5点到下午5:59中的每分钟激发一次 trigger。它停在下午 5:59 是因为值 17 在小时域上，在下午 6 点时，小时变为 18 了，也就不再理会这个 trigger，直到下一天的下午5点。

在你希望 trigger 在该域的所有有效值上被激发时使用 * 字符。

? 问号

? 号只能用在日和周域上，但是不能在这两个域上同时使用。你可以认为 ? 字符是 "我并不关心在该域上是什么值。" 这不同于星号，星号是指示着该域上的每一个值。? 是说不为该域指定值。

不能同时这两个域上指定值的理由是难以解释甚至是难以理解的。基本上，假定同时指定值的话，意义就会变得含混不清了：考虑一下，如果一个表达式在日域上有值11，同时在周域上指定了 WED。那么是要 trigger 仅在每个月的11号，且正好又是星期三那天被激发？还是在每个星期三的11号被激发呢？要去除这种不明确性的办法就是不能同时在这两个域上指定值。

只要记住，假如你为这两域的其中一个指定了值，那就必须在另一个字值上放一个 ?。

表达式样例：

0 10,44 14 ? 3 WEB

意义：在三月中的每个星期三的下午 2:10 和 下午 2:44 被触发。

, 逗号

逗号 (,) 是用来在给某个域上指定一个值列表的。例如，使用值 0,15,30,45 在秒域上意味着每15秒触发一个 trigger。

表达式样例：

0 0,15,30,45 * * * ?

意义：每刻钟触发一次 trigger。

/ 斜杠

斜杠 (/) 是用于时间表的递增的。我们刚刚用了逗号来表示每15分钟的递增，但是我们也能写成这样 0/15。

表达式样例：

0/15 0/30 * * * ?

意义：在整点和半点时每15秒触发 trigger。

- 中划线

中划线 (-) 用于指定一个范围。例如，在小时域上的 3-8 意味着 "3,4,5,6,7 和 8 点。" 域的值不允许回卷，所以像 50-10 这样的值是不允许的。

表达式样例：

0 45 3-8 ? * *

意义：在上午的3点至上午的8点的45分时触发 trigger。

L 字母

L 说明了某域上允许的最后一个值。它仅被日和周域支持。当用在日域上，表示的是在月域上指定的月份的最后一天。例如，当月域上指定了 JAN 时，在日域上的 L 会促使 trigger 在1月31号被触发。假如月域上是 SEP，那么 L 会预示着在9月30号触发。换句话说，就是不管指定了哪个月，都是在相应月份的时最后一天触发 trigger。

表达式 0 0 8 L * ? 意义是在每个月最后一天的上午 8:00 触发 trigger。在月域上的 * 说明是 "每个月"。

当 L 字母用于周域上，指示着周的最后一天，就是星期六 (或者数字7)。所以如果你需要在每个月的最后一个星期六下午的 11:59 触发 trigger，你可以用这样的表达式 0 59 23 ? * L。

当使用于周域上，你可以用一个数字与 L 连起来表示月份的最后一个星期 X。例如，表达式 0 0 12 ? * 2L 说的是在每个月的最后一个星期一触发 trigger。

不要让范围和列表值与 L 连用

虽然你能用星期数(1-7)与 L 连用，但是不允许你用一个范围值和列表值与 L 连用。这会产生不可预知的结果。

W 字母

W 字符代表着平日 (Mon-Fri)，并且仅能用于日域中。它用来指定离指定日的最近的一个平日。大部分的商业处理都是基于工作周的，所以 W 字符可能是非常重要的。例如，日域中的 15W 意味着 "离该月15号的最近一个平日。" 假如15号是星期六，那么 trigger 会在14号(星期五)触发，因为星期四比星期一（这个例子中是17号）离15号更近。（译者Unmi注：不会在17号触发的，如果是15W，可能会是在14号(15号是星期六)或者15号(15号是星期天)触发，也就是只能出现在邻近的一天，如果15号当天为平日直接就会当日执行）。W 只能用在指定的日域为单天，不能是范围或列表值。

井号

字符仅能用于周域中。它用于指定月份中的第几周的哪一天。例如，如果你指定周域的值为 6#3，它意思是某月的第三个周五 (6=星期五，#3意味着月份中的第三周)。另一个例子 2#1 意思是某月的第一个星期一 (2=星期一，#1意味着月份中的第一周)。注意，假如你指定 #5，然而月份中没有第 5 周，那么该月不会触发。

第五章. Cron 触发器及相关内容（第三部分）

四. 为 CronTrigger 使用起迄日期

Cron 表达式是用来决定一个 Trigger 被触发执行一个 Job 的日期和次数。当你创建一个 CronTrigger 实例，假如没为它指定一个开始时间，这个 Trigger 当然就会假定是在依赖于 Cron 表达式尽早的被触发。例如，如果你用这个表达式

0 * 14-20 * * ?

这个 Trigger 会在每天的从下午 2 点到下午的 7:59 间的每分钟触发一次。一旦你运行了这个表达式的 CronTrigger，假如当前是下午 2 点后(不能超过 7:59 PM--译者注)，它将会立即触发。它会在每天无限期的被触发。

另一方面，倘若你希望这个计划直到下一天才开始，并且只执行两天，你就可以用 CronTrigger 的 setStartTime() 和 setEndTime() 方法来形成一个 "定时箱" 来触发。代码 5.2 描述了限定 CronTrigger 仅触发两天的例子。

代码 5.2. 你可以对 CronTrigger 用 startTime 和 endTime

```
1.  public class Listing_5_2 {
2.       static Log logger = LogFactory.getLog(Listing_5_2.class);
3.
4.       public static void main(String[] args) {
5.            Listing_5_2 example = new Listing_5_2();
6.            example.runScheduler();
7.       }
8.
9.       public void runScheduler() {
10.           Scheduler scheduler = null;
11.
12.           try {
13.
14.               // Create a default instance of the Scheduler
15.               scheduler = StdSchedulerFactory.getDefaultScheduler();
16.               scheduler.start();
17.               logger.info("Scheduler was started at " + new Date());
18.
19.               // Create the JobDetail
20.               JobDetail jobDetail = new JobDetail("PrintInfoJob",
21.                         Scheduler.DEFAULT_GROUP,
22.                                   PrintInfoJob.class);
23.
24.               // Create a CronTrigger
25.               try {
26.                    // cron that fires every min from 2  8pm
27.                    CronTrigger trigger =
28.                                   new CronTrigger("MyTrigger", null,
29.                                                 "0 * 14-20 * * ?");
30.
31.                    Calendar cal = Calendar.getInstance();
32.                    // Set the date to 1 day from now
33.                    cal.add(Calendar.DATE, 1);
34.                    trigger.setStartTime(cal.getTime());
35.
36.                    // Move ahead 2 days to set the end time
37.                    cal.add(Calendar.DATE, 2);
38.                    trigger.setEndTime(cal.getTime());
39.
40.                    scheduler.scheduleJob(jobDetail, trigger);
```

```
                } catch (ParseException ex) {
                    logger.error("Couldn't parse cron expr", ex);
                }

            } catch (SchedulerException ex) {
                logger.error(ex);
            }
        }
    }
```

代码 5.2 中的例子使用了 java.util.Calendar 来为 Trigger 选择一个开始和结束时间周期。在上面例子中，Trigger 将会在 Scheduler 启动后的下一天开始触发，并只在开始触发后的两天内有效。

使用 CronTrigger 的 startTime 和 endTime 属性的效果有点像 SimpleTrigger。

五. 为 CronTrigger 使用 TriggerUtils

在第四章，"安排 Job" 中介绍了 org.quartz 包中的 TriggerUtils 类，它简化了两种类型的 Trigger 的创建。只要可能的话，你应该尝试用 TriggerUtils 类的方法来创建你的 Trigger。

例如，假如你需要在每天的下午 5:30 执行一个 Job，你可以用下面的代码：

```
try {

  // A CronTrigger that fires @ 5:30PM
  CronTrigger trigger = new CronTrigger("CronTrigger", null, "0 30 17 ? * *");
} catch (ParseException ex) {
  logger.error("Couldn't parse cron expression", ex);
}
```

或者你能用上 TriggerUtils，如下：

```
  // A CronTrigger that fires @ 5:30PM
Trigger trigger = TriggerUtils.makeDailyTrigger(17, 30);
trigger.setName("CronTrigger");
```

TriggerUtils 使得我们更简单方便的使用 Trigger，而又未放弃太多的灵活性。

六. 在 JobInitializationPlugin 中使用 CronTrigger

尽管我们要到第八章，"使用 Quartz 插件" 才会讲到插件，但还是值得提前展现一下 CronTrigger 如何应用于 quartz_jobs.xml 文件中来指定 Job 信息的。JobInitialzationPlugin 可用来从 XML 文件中加载 Job 的信息。

正如 SimpleTrigger 一样，你可在 XML 文件中指定 CronTrigger 的表达式，并且 Quartz 的 Scheduler 将会利用这一信息来安排你的 Job。这对于你想在你的程序代码之外声明你的 Job 信息时特别方便。代码 5.3 显示了 quartz_jobs.xml 文件内容，它被 JobInitializationPlugin 用来加作 Job 信息。

代码 5.3. CronTrigger 可在 XML 文件中指定，并由 JobInitializationPlugin 加载

```
<?xml version='1.0' encoding='utf-8'?>

<quartz>
  <job>
    <job-detail>
```

```
      <name>PrintInfoJob</name>
    <group>DEFAULT</group>
    <description>
      A job that prints out some basic information.
    </description>
    <job-class>
      org.cavaness.quartzbook.common.PrintInfoJob
    </job-class>
   </job-detail>

   <trigger>
    <cron>
     <name>printJobInfoTrigger</name>
     <group>DEFAULT</group>
     <job-name>PrintInfoJob</job-name>
     <job-group>DEFAULT</job-group>

      <!-- Fire 7:30am Monday through Friday -->
     <cron-expression>0 30 7 ? * MON-FRI</cron-expression>
    </cron>
   </trigger>
  </job>
</quartz>
```

代码 5.3 中的 Cron 表达式与 代码 5.1 中。当 Quartz 加载这个 XML 后，就会安排 PrintInfoJob (也已在 XML 中列出) 在从星期一到星期五的早上 7:30 执行。关于 JobInitializationPlugin 更多的说明见第八章。

第五章. Cron 触发器及相关内容（第四部分）

七. Cron 表达式 Cookbook

此处的 Cron 表达式 cookbook 旨在为常用的执行需求提供方案。尽管不可能列举出所有的表达式，但下面的应该为满足你的业务需求提供了足够的例子。

·分钟的 Cron 表达式

表 5.1. 包括了分钟频度的任务计划 Cron 表达式

用法	表达式
每天的从 5:00 PM 至 5:59 PM 中的每分钟触发	0 * 17 * * ?
每天的从 11:00 PM 至 11:55 PM 中的每五分钟触发	0 0/5 23 * * ?
每天的从 3:00 至 3:55 PM 和 6:00 PM 至 6:55 PM 之中的每五分钟触发	0 0/5 15,18 * * ?
每天的从 5:00 AM 至 5:05 AM 中的每分钟触发	0 0-5 5 * * ?

·日上的 Cron 表达式

表 5.2. 基于日的频度上任务计划的 Cron 表达式

用法	表达式
每天的 3:00 AM	0 0 3 * * ?
每天的 3:00 AM (另一种写法)	0 0 3 ? * *
每天的 12:00 PM (中午)	0 0 12 * * ?
在 2005 中每天的 10:15 AM	0 15 10 * * ? 2005

·周和月的 Cron 表达式

表 5.3. 基于周和/或月的频度上任务计划的 Cron 表达式

用法	表达式
在每个周一,二, 三和周四的 10:15 AM	0 15 10 ? * MON-FRI
每月15号的 10:15 AM	0 15 10 15 * ?
每月最后一天的 10:15 AM	0 15 10 L * ?
每月最后一个周五的 10:15 AM	0 15 10 ? * 6L
在 2002, 2003, 2004, 和 2005 年中的每月最后一个周五的 10:15 AM	0 15 10 ? * 6L 2002-2005
每月第三个周五的 10:15 AM	0 15 10 ? * 6#3
每月从第一天算起每五天的 12:00 PM (中午)	0 0 12 1/5 * ?
每一个 11 月 11 号的 11:11 AM	0 11 11 11 11 ?
三月份每个周三的 2:10 PM 和 2:44 PM	0 10,44 14 ? 3 WED

八. 创建一个即刻触发的 Trigger

有时候，你需要立即执行一个 job。例如，想像一下，你正在构建一个 GUI 程序并允许用户能立刻执行。另一个例子，你或许已经检测到了某个 Job 未执行成功，因此你想要即刻重跑一次。在 Quartz 1.5，有几个方法被加入到了 TriggerUtils 类中，使得实现那些事很容易了。代码 5.4 展示了如何部署一个 job，只让它立即执行一次。

代码 5.4. 你可以用 TriggerUtils 来立即执行一个 Job

```
1.  public class Listing_5_4 {
2.       static Log logger = LogFactory.getLog(Listing_5_4.class);
3.
4.      public static void main(String[] args) {
5.           Listing_5_4 example = new Listing_5_4();
6.           example.runScheduler();
7.      }
8.
9.      public void runScheduler() {
10.          Scheduler scheduler = null;
11.
12.          try {
13.               // Create a default instance of the Scheduler
14.               scheduler = StdSchedulerFactory.getDefaultScheduler();
15.               scheduler.start();
16.               logger.info("Scheduler was started at " + new Date());
17.
18.               // Create the JobDetail
19.               JobDetail jobDetail = new JobDetail("PrintInfoJob",
20.                                        Scheduler.DEFAULT_GROUP,
21.                                        PrintInfoJob.class);
22.
23.               // Create a trigger that fires once right away
24.               Trigger trigger = TriggerUtils.makeImmediateTrigger(0, 0);
25.
26.               trigger.setName("FireOnceNowTrigger");
27.
28.               scheduler.scheduleJob(jobDetail, trigger);
29.          } catch (SchedulerException ex) {
30.              logger.error(ex);
31.          }
32.     }
33. }
```

在代码 5.4 中，TriggerUtils 的 makeImmediateTrigger() 方法被用来立即执行一个 Job。第一个参数是将要触发的次数。第二个参数是执行的间隔时间。为方便起见，这个方法的签名显示如下：

```
public static Trigger  makeImmediateTrigger(int repeatCount, long repeatInterval);
```

TriggerUtils 类提供了许多便利的方法简化了 Trigger 的使用。确切地检查一下这个工具类中看看是否有你想要的东西。你还将在本书上看到更多的使用 TriggerUtils 的例子。

第六章. Job 存储和持久化（第一部分）

第六章. Job 存储和持久化

Quartz 用 JobStores 对 Job、Trigger、calendar 和 Schduler 数据提供一种存储机制。Scheduler 应用已配置的 JobStore 来存储和获取到部署信息，并决定正被触发执行的 Job 的职责。所有的关于哪个 Job 要执行和以什么时间表来执行他们的信息都来自于 JobStore。本章就来看 Quartz 中可用的各种类型的 JobStore，和如何使用他们，以及哪一个能适应你的需求。

> *"罗马非一日建成"*
>
> 道格拉斯.亚当斯，《宇宙环游指南》

一. Job 存储

在前面章节中，我们未曾花过任何时间来讨论 Scheduler 的 Job 和 Trigger 是保存在哪儿的。我们也许已经实现了，然而，当你停止了 Scheduler 后，那些有关哪些 Job 已经运行和哪些 Job 没有运行的信息就会丢失掉。实际上，所有的关于正在运行中的 Job 的信息也被销毁。

当程序被重启后，Trigger 和 Job 的信息被加回去，且所有的一切又都正常了。我们假定，有一个 Job 是安排为 5 PM 执行，然而 Scheduler 在这个时间之前的五分钟(4:55 PM) 时停掉了。如果你在 5:05 PM 时重新启动了 Scheduler 的话将会发生什么事情呢？Scheduler 还会记得要在 5 PM 触发这个 Job 的吗？答案就是看它是依赖于你使用的哪种类型的 JobStore，以及是如何对它配置的。

二. Quartz 中的 Job 存储

Quartz 支持对 Scheduler 信息的几种不同类型的存储机制。在 Quartz 中两种可用的 Job 存储类型是：

- 内存(非持久化) 存储
- 持久化存储

默认时，我们在前面几章的例子中已经使用了内存存储机制。两种类型都是用来服务于相同的目的：存储 Job 信息。然而他们各自是如何运作的，而且他们提供给 Scheduler 的功能是很不一样的。

·JobStore 接口

Quartz 为所有类型的 Job 存储提供了一个接口。这个接口位于 org.quartz.spi 包中，叫做 JobStore。所有的 Job 存储机制，不管是在哪里或是如何存储他们的信息的，都必须实现这个接口。

JobStore 可以列出太多的方法来，但是 JobStore 接口的 API 可归纳为下面几类：

- ·Job 相关的 API
- ·Trigger 相关的 API
- ·Calendar 相关的 API
- ·Scheduler 相关的 API

Quartz 的使用者几乎从不访问或是查看实现了 JobStore 接口的具体类；他们被 Quartz Scheduler 在运行期间内部使用来获取 Job 和 Trigger 信息。不过很值得练习一下，使你自己能熟悉每一种类型，这样你就能更好的理解这些为你所提供的存储机制，并有助于你在 Quartz 应用中选择一个正确的类型。

三. 使用内存来存储 Scheduler 信息

Quartz 直接可用的配置就是把 Job 和 Trigger 信息存储在内存中的。这个解释了为什么，对于前面章节中的例子，每次我们重启了 Quartz 应用程序后，Scheduler 的状态，包括 Job 和 Trigger 信息都丢失了。每回 Java 虚拟机(JVM) 关闭之后，它所占用的内存就释放回给了操作系统，因此任何关于 Job 和 Trigger 的信息都随 JVM 而丢失。

Quartz 的内存 Job 存储的能力是由一个叫做 org.quartz.simple.RAMJobStore 类提供了，当如我们所说，它实现了 JobStore 接口的。RAMJobStore 是 Quartz 的开箱即用的解决方案。对此，我们的意思是说，除非你改变了配置，否则在任何 Quartz 应用中都将使用 RAMJobStore。相比于其他的，使用这种 JobStore 可带来几个好处。

首先，RAMJobStore 是配置最简单的 JobStore：已给你配置好了的。当你下载并安装 Quartz 后，就已为你配置了使用 RAMJobStore 作为存储机制。你能在默认的 quartz.properties 文件中看到这个，如代码 6.1 所示。

代码 6.1. 没有其他配置时默认的 quartz.properties 文件

```
# Default Properties file for use by StdSchedulerFactory
# to create a Quartz Scheduler Instance, if a different
# properties file is not explicitly specified.

org.quartz.scheduler.instanceName = DefaultQuartzScheduler
org.quartz.scheduler.rmi.export = false
org.quartz.scheduler.rmi.proxy = false
org.quartz.scheduler.wrapJobExecutionInUserTransaction = false
org.quartz.threadPool.class = org.quartz.simpl.SimpleThreadPool
org.quartz.threadPool.threadCount = 10
org.quartz.threadPool.threadPriority = 5
org.quartz.threadPool.threadsInheritContextClassLoaderOfInitializingThread = true

org.quartz.jobStore.misfireThreshold = 60000

org.quartz.jobStore.class = org.quartz.simpl.RAMJobStore
```

代码 6.1 中显示了包含在 Quartz 二进制包中的默认的 quartz.properties 文件。当你没在你自己的程序中引入一个 quartz.properties 文件，这个属性文件就会得到使用。你可以从默认的 quartz.properties 文件的最后一行看到，RAMJobStore 是名为 org.quartz.jobStore.class 的配置属性的默认值。甚至是未在 quartz.propterties 中设置 org.quartz.jobStore.class 属性时，RAMJobStore 也是默认所用的 JobStore。这是硬编码到 Scheduler 工厂初始化程序中的。

另一使用 RAMJobStore 是优点是它的速度。因为所有的 Scheduler 信息都保存在计算机内存中，访问这些数据随着电脑而变快。这儿不存在进程外的调用，没有数据库连，仅仅是原始而简单的内存访问。再也找不到比这更快的方式了。

·RAMJobStore 的 Job 易失性

你也许还记得在第四章，"部署 Job" 中讲过，Job 可以配置一个易失性属性。当这个易失性属性设置为 false，Job 将会在应用关闭之间持久化下来。这个属性对于用 RAMJobStore 时是不起作用的；那一行为是显式的为持久性的 JobStore 所保留的。

·配置 RAMJobStore

配置你的 Quartz 应用来使用 RAMJobStore 是非常简单的。假如你正用一个定制的 quartz.properties 文件，而不是来自于 Quartz JAR 文件中的，那么加上这行到你的属性文件中即可：

org.quartz.jobStore.class = org.quartz.simpl.RAMJobStore

这就你要用 RAMJobStore 所要做的事情，正如我们所说的，没有比这更简单的了。

·加载 Job 到 RAMJobStore 中

既然 RAMJobStore 的目的是存储 Scheduler 信息，那么那些信息一开始是如何被加载到内存中的呢？你可以用两种方式加载

Job 信息。首先，你能硬编码你的 Job、Trigger、calendar 和 listener 到你的代码中。如第三章，"Hello，Quarz"，第四章，"部署 Job" 所指出的，然而，这总是一件很危险的事情，因为这对于维护来说将是个梦魇。任何的改变，即使是微不足道的，都要修改代码然后重新编译。甚至是只改变触发的时间，也要改代码然后重编译。这没什么大不了的，你说呢？那也许对于小小的程序是这样的，但对于有大量的 Job 和 Trigger 的程序却成了一个大问题。

第二种途径是使用 JobInitializationPlugin，这会在第八章，"使用 Quartz 插件" 中详细讨论。这个 Quartz 插件使用一个 XML 文件来加载 Job、Trigger、Calendar 和其他你需要加载的东西。这种方式的优点是，当有改变时只需要对这个 XML 文件作改动，不用改代码，不用重编译，仅用一个文本编辑器。阅读第八章可获得关于 Quartz 插件更多的信息。

·RAMJobStore 的缺点

你要问了，"RAMJobStore 不能全是正面的，对吗？"。没错，确实如此。我们前面提到几个使用 RAMJobStore 的优点。现在，让我们来谈谈它的一个负面的地方：因为计算机的内存是易失性的，当你的 Quartz 程序被停止了，它会把内存释放回操作系统，当然了，伴随着存储在所释放内存里的别的内容就是这些部署信息了。

假如你的程序的 Scheduler 信息需要在程序重启之间能保持着，那么你需要看看持久性的 JobStore 了。

第六章. Job 存储和持久化（第二部分）

四. 使用持久性的 JobStore

在很多方面，JobStore 有用内存来存储的，还有些使用某种能长期持久的方式来共享相拟的特征。这不该有什么惊奇的，因为他们都服务于同一目的。

和 RAMJobStore 一样，特久性的 JobStore 有优点也有其缺点。在你选择持久性的 JobStore 之前应该认真理解其利与弊。这节就来解释它们的区别，以及在什么情况下你会希望使用持久性的 JobStore。

目前，Quartz 提供了两种类型的持久性 JobStore，每一种类型都有其独特的持久化机制。

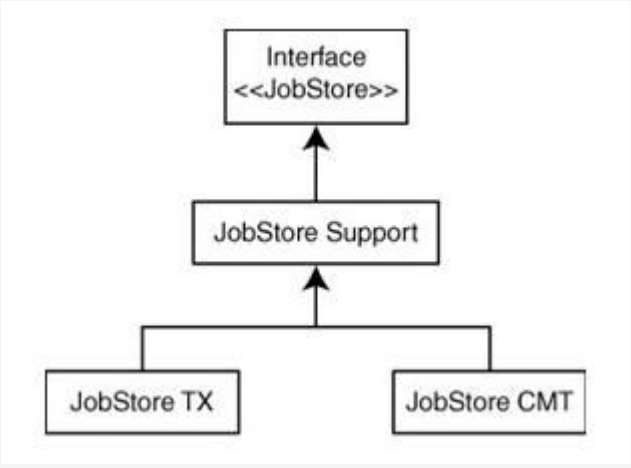
> **持久性 JobStore = JDBC + 关系型数据库**
>
> 尽管有几种不同的持久化机制可被 Quartz 用于持久化 Scheduler 信息，Quartz 依赖于一个关系型数据库管理系统(RDMS) 来持久化存储。假如你想用某种别的而不是数据库来持久化存储，那么你必须通过实现 JobStore 接口自己构建它。假定你想用文件系统来持久化存储。你可以创建一个类，这个类要实现 JobStore 接口，在本章中，当我们说 "持久化"，我们隐式的是说用 JDBC 来持久化 Scheduler 状态到数据库中

Quartz 所带的所有的持久化的 JobStore 都扩展自 org.quartz.impl.jdbcjobstore.JobStoreSupport 类。

·JobStoreSupport 类

JobStoreSupport 是个抽象类，并实现了 JobStore 接口，在此章前面就讨论过的。它为所有基于 JDBC 的 JobStore 提供了基本的功能。图 6.1 显示了 JobStore 类型的层次关系。

图 6.1. JobStore 类型层次



如图 6.1 所描绘的，JobStoreSupport 实现了 JobStore 接口，是作为 Quartz 提供的两个具体的持久性 JobStore 类的基类。

> **JobStoreSupport 本该名之为 JDBCJobStoreSupport**
>
> 作为这个类的一个更好的名字本应该是 JDBCJobStoreSupport，因为这个类专门是为基于 JDBC 存储方案而设置的，然而，这个名并没有减损到它为持久性 JobStore 所提供的功能。

因为 JobStoreSupport 类是抽象的，因此 Quartz 提供了两种不同类型的具体化的 JobStore，每一个设计为针对特定的数据库环境和配置：

·org.quartz.impl.jdbcjobstore.JobStoreTX

·org.quartz.impl.jdbcjobstore.JobStoreCMT

这两个持久性 JobStore 前面简单讨论过。但现在，我们来讨论两个版本所需要的数据库。

五. 为 Job 存储使用数据库

Quartz 中的持久性 JobStore 有时候就是指 JDBC JobStore，因为他们基本是依赖于一个 JDBC 驱动和一个关系型数据库通信。持久性 JobStore 会用到许多的 JDBC 特性，包括支持事特，锁定和隔离级别，只列了这几个特性罢。

> **要是我的数据库不支持 JDBC 呢？**
>
> 假如你的数据库不支持 JDBC，那你肯定是碰到什么问题了。并不能全归咎于你运气不好，只是在这之前你还有很多工作要做。你最好是切换到某一种支持的数据库平台上来。假如你的数据库不支持 JDBC，你将需要创建一个新的实现(实现了 JobStore 接口)。你也许想检查一下 Quartz 用户论坛里的用户，看谁是否已经做了这样的工作，并是否愿意共享他们的代码或者至少告诉你实现的方法。

·持久性 JobStore 所支持的数据库

Quartz 中的持久性 JobStore 被设计能与如下数据库平台一同使用：

- ·Oracle
- ·MySQL
- ·MS SQL Server 2000
- ·HSQLDB
- ·PostgreSQL
- ·DB2
- ·Cloudscape/Derby
- ·Pointbase
- ·Informix
- ·Firebird
- ·大多数别的有完全 JDBC 兼容性驱动的 RDBMS

·独立环境中的持久性存储

JobStoreTX 类设计为用于独立环境中。这里的 "独立"，我们是指这样一个环境，在其中不存在与应用容器的事物集成。这里并不意味着你不能在一个容器中使用 JobStoreTX，只不过，它不是设计来让它的事特受容器管理。区别就在于 Quartz 的事物是否要参与到容器的事物中去。

·程序容器中的持久性存储

JobStoreCMT 类设计为当你想要程序容器来为你的 JobStore 管理事物时，并且那些事物要参与到容器管理的事物边界时使用。它的名字明显是来源于容器管理的事物(Container Managed Transactions (CMT))。

六. 创建 Quartz 数据库结构

JobStore 是基于 JDBC 的，它需要一个数据用于 Scheduler 信息的持久化。Quartz 需要创建 12 张数据库表。表的名字和描述在表 6.1 中列出。

表 6.1. Quartz 需要下列表用于所有的 JDBC 的持久性 JobStore

表名	描述
QRTZ_CALENDARS	以 Blob 类型存储 Quartz 的 Calendar 信息
QRTZ_CRON_TRIGGERS	存储 Cron Trigger，包括 Cron 表达式和时区信息
QRTZ_FIRED_TRIGGERS	存储与已触发的 Trigger 相关的状态信息，以及相联 Job 的执行信息
QRTZ_PAUSED_TRIGGER_GRPS	存储已暂停的 Trigger 组的信息
QRTZ_SCHEDULER_STATE	存储少量的有关 Scheduler 的状态信息，和别的 Scheduler 实例(假如是用于一个集群中)
QRTZ_LOCKS	存储程序的非观锁的信息(假如使用了悲观锁)
QRTZ_JOB_DETAILS	存储每一个已配置的 Job 的详细信息
QRTZ_JOB_LISTENERS	存储有关已配置的 JobListener 的信息
QRTZ_SIMPLE_TRIGGERS	存储简单的 Trigger，包括重复次数，间隔，以及已触的次数
QRTZ_BLOG_TRIGGERS	Trigger 作为 Blob 类型存储(用于 Quartz 用户用 JDBC 创建他们自己定制的 Trigger 类型，JobStore 并不知道如何存储实例的时候)
QRTZ_TRIGGER_LISTENERS	存储已配置的 TriggerListener 的信息
QRTZ_TRIGGERS	存储已配置的 Trigger 的信息

中表 6.1 中，所有的表都是以前缀 QRTZ_ 开始。这是默认的，但是你可以通过在 quartz.properties 文件中提供一个替代的前缀来改变它。如果你对不同的 Scheduler 实例使用了多套的表，那么改变这个前缀则是必须的。这在你需要用到多个非集群的 Scheduler，但只想用一个单独的数据库实例时也是要做的。

·安装 Quartz 数据库表

Quartz 包括了所有被支持的数据库平台的 SQL 脚本。你能在 <quartz_home>/docs/dbTables 目录下找到那些 SQL 脚本，这里的 <quartz_home> 是解压 Quartz 分发包后的目录。

大约有 18 种不同的数据库平台的脚本。这差不多能覆盖到你所能想出来的任何数据库。假如你的不在其列，你可以使用其中一个已存在的脚本，略做修改以适应你的数据库平台。

要安装必须的数据库表，先打开那个专为你的数据库平台制作的 .sql 文件并用你喜爱的查询工具执行其中的命令。比如说是 MS SQL Server，你需要用数据库所带的查询分析器(Quary Analyzer) 运行文件 tables_sqlServer.sql 中的命令。SQL 命令不负责创建数据库。你还要特别留意 SQL 文件最前面的注释。通常，在运行命令之前都必须执行几条指令。例如，还是 MS SQL Server 的 SQL 文件，你需要修改文件顶端的这条命令，填入数据库名称，在你刚创建它的时候还是一个空数据库。

```
USE [enter_db_name_here]
```

SQL 文件创建了必须的表结构，还给表加上了基本的约束和索引。在本章后面，我们会讨论如何通过对表结构做些额外的变动来改进性能。

第六章. Job 存储和持久化（第三部分）

七. 使用 JobStoreTX

我们首先要讨论的持久性 JobStore 是 JobStoreTX。名字中的 "TX" 代表着 "事物"。我们在前面提过，JobStoreTX 是设计用于想要 Quartz 来管理事物的环境中。例如，假如你正构建一个 J2EE 应用，并且不使用到应用服务器，如 WebLogic 或者 JBoss 等，那么 JobStoreTX 会是持久性 JobStore 正确的选择。

在之前章节中，我们看到配置 RAMJobStore 是多么的容易。我们提到 RAMJobStore 的其中一个优点就是易于配置。我们已经讨论过让数据库准备就绪该做的事情；现在我们讲述使 Quartz 应用支持 JDBC JobStore 需要对它配置些什么。

·配置 JobStoreTX

要告诉 Quartz 运行环境你想使用一个别的 JobStore 而不是默认的 RAMJobStore，你必须配置几个属性。配置它们的顺序无关紧要，只要保证在第一次运行程序之前都做了设置。

设置 JobStore 属性

欲告知 Scheduler 应该使用 JobStoreTX，你必须加上下面一行到 quartz.properties 文件中：

```
org.quartz.jobStore.class = org.quartz.ompl.jdbcjobstore.JobStoreTX
```

在切换到 JobStoreTX 时确保移动了 RAMJobStore 行(假如存在)。

配置驱动代理

JDBC API 依赖于专属于某个数据库平台的 JDBC 驱动，同样的，Quartz 依赖于某个 DriverDelegate 来与给定数据库进行通信。顾名思义，从 Scheduler 通过 JobStore 对数据库的调用是委托给一个预配置的 DriverDelegate 实例。这个代理承担起所有与 JDBC driver 也就是数据库的通信。

所有的 DriverDelegate 类都继承自 org.quartz.impl.jdbcjobstore.StdDriverDelegate 类。StdDriverDelegte 只有所有代理可用的，平台无关性的基本功能。然而，在不同的数据库平台间还是存在太多的差异，因此需要为某个平台创建特定的代理。表 6.2 列出特定的代理。

表 6.2. 你必须为你的平台配置其中一个 DriverDelegate

数据库平台	Quartz 代理类
Cloudscape/Derby	org.quartz.impl.jdbcjobstore.CloudscapeDelegate
DB2 (version 6.x)	org.quartz.impl.jdbcjobstore.DB2v6Delegate
DB2 (version 7.x)	org.quartz.impl.jdbcjobstore.DB2v7Delegate
DB2 (version 8.x)	org.quartz.impl.jdbcjobstore.DB2v8Delegate
HSQLDB	org.quartz.impl.jdbcjobstore.PostgreSQLDelegate
MS SQL Server	org.quartz.impl.jdbcjobstore.MSSQLDelegate
Pointbase	org.quartz.impl.jdbcjobstore.PointbaseDelegate
PostgreSQL	org.quartz.impl.jdbcjobstore.PostgreSQLDelegate
(WebLogic JDBC Driver)	org.quartz.impl.jdbcjobstore.WebLogicDelegate
(WebLogic 8.1 with Oracle)	org.quartz.impl.jdbcjobstore.oracle.weblogic.WebLogicOracleDelegate
Oracle	org.quartz.impl.jdbcjobstore.oracle.OracleDelegate

> **假如我的数据库平台在表 6.2 中未列出该怎么办?**
>
> 如果你的 RDBMS 没在上面列出，那么最好的选择就是，直接使用标准的 JDBC 代理 org.quartz.impl.jdbcjobstore.StdDriverDelegate 就能正常的工作。

在你决定好了基于你的数据库平台使用哪个代理，你就需要加入下面的行到 quartz.properties 文件中：

org.quartz.jobStore.driverDelegateClass = <FQN of driver delegate class>

例如，假如你使用 MS SQL Server 作为一你的数据库平台，你就需要加下面这行到属性文件中：

org.quartz.jobStore.driverDelegateClass = org.quartz.impl.jdbcjobstore.MSSQLDelegate

配置数据库表的前缀

前面我们首先讨论 Quartz 要用的数据库表的时候，我们提到所有的表都加有一个前缀 QRTZ_。在某些情况下，你也许需要创建多套的 Quartz 数据库表。在这时候，你就需要改变每一套表的前缀。

表名的前缀配置在 quartz.properties 文件中，使用属性 org.quartz.jobStore.tablePrefix。要改变这一前缀，只要设置这个属性为不同的值：

org.quartz.jobStore.tablePrefix = SCHEDULER2_

确定所有的表名都开始于这一前缀。

> **数据库表和列的命名**
>
> 假使你还有所疑惑，数据库表的名字(除却前缀) 和表的列名定义在 org.quartz.impl.jdbcjobstore.Constants 接口中。这个接口为 JobStoreSupport 类所实现，因而那些常量值在 JobStoreTX 或 JobStoreCMT 类中是可用的。

表 6.3 显示了一系列的属性可用于调节 JobStoreTX。

表 6.3. 可用于设置 JobStoreTX 的配置属性

属性	默认值
org.quartz.jobStore.driverDelegateClass **描述**：能理解不同数据库系统中某一特定方言的驱动代理	
org.quartz.jobStore.dataSource **描述**：用于 quartz.properties 中数据源的名称	
org.quartz.jobStore.tablePrefix **描述**：指定用于 Scheduler 的一套数据库表名的前缀。假如有不同的前缀，Scheduler 就能在同一数据库中使用不同的表。	QRTZ_
org.quartz.jobStore.userProperties **描述**："use properties" 标记指示着持久性 JobStore 所有在 JobDataMap 中的值都是字符串，因此能以名-值 对的形式存储，而不用让更复杂的对象以序列化的形式存入 BLOB 列中。这样会更方便，因为让你避免了发生于序列化你的非字符串的类到 BLOB 时的有关类版本的问题。	False
org.quartz.jobStore.misfireThreshold **描述**：在 Trigger 被认为是错过触发之前，Scheduler 还容许 Trigger 通过它的下次触发时间的毫秒数(译者注：据原文翻译，真的不好理解，实际效果可参看：http://www.blogjava.net/Unmi/archive/2007/10/23/153413.html 我在评论中的实验)。默认值(假如你未在配置中存在这一属性条目) 是 60000(60 秒)。这个不仅限于 JDBC-JobStore；它也可作为 RAMJobStore 的参数	60000
org.quartz.jobStore.isClustered **描述**：设置为 true 打开集群特性。如果你有多个 Quartz 实例在用同一套数据库时，这个属性就必须设置为 true。	False
org.quartz.jobStore.clusterCheckinInterval **描述**：设置一个频度(毫秒)，用于实例报告给集群中的其他实例。这会影响到侦测失败实例的敏捷度。它只用于设置了 isClustered 为 true 的时候。	15000
org.quartz.jobStore.maxMisfiresToHandleAtATime **描述**：这是 JobStore 能处理的错过触发的 Trigger 的最大数量。处理太多(超过两打) 很快会导致数据库表被锁定够长的时间，这样就妨碍了触发别的(还未错过触发) trigger 执行的性能。	20

org.quartz.jobStore.dontSetAutoCommitFalse	False

描述：设置这个参数为 true 会告诉 Quartz 从数据源获取的连接后不要调用它的 setAutoCommit(false) 方法。这在少些情况下是有帮助的，比如假如你有这样一个驱动，它会抱怨本来就是关闭的又来调用这个方法。这个属性默认值是 false，因为大多数的驱动都要求调用 setAutoCommit(false)。

org.quartz.jobStore.selectWithLockSQL	SELECT * FROM {0}LOCKS WHERE LOCK_NAME = ? FOR UPDATE

描述：这必须是一个从 LOCKS 表查询一行并对这行记录加锁的 SQL 语句。假如未设置，默认值就是 SELECT * FROM {0}LOCKS WHERE LOCK_NAME = ? FOR UPDATE，这能在大部分数据库上工作。{0} 会在运行期间被前面你配置的 TABLE_PREFIX 所替换。

org.quartz.jobStore.txIsolationLevelSerializable	False

描述：值为 true 时告知 Quartz(当使用 JobStoreTX 或 CMT) 调用 JDBC 连接的 setTransactionIsolation (Connection.TRANSACTION_SERIALIZABLE) 方法。这有助于阻止某些数据库在高负载和长时间事物时锁的超时。

第六章. Job 存储和持久化（第四部分）

八. 为 JobStroreTX 创建数据源

当使用持久性 JobStore 时，Quartz 需要一个数据源。数据源扮演着产生数据库连接工厂的角色。在 Java 中，所有的数据源要实现 java.sql.Datasource 接口。Quartz 自身并不提供 Datasource 的所有功能；它只代表了那一职责。默认的，Quartz 能使用另一开源的框架，叫做 Commons DBCP，或者可以通过 JNDI 查找应用服务器中定义的 DataSource。

DBCP 是一个 Jakarta Commons 项目，网址是 http://jakarta.apache.org/commons/dbcp。这个框架的二进制版包含在 Quartz 的发行版中，你应该把它加到你的 Quartz 应用中来。你还需要加入 Commons Pool 库，它也包含中 Quartz 发行版中，是 DBCP 要用到的。

使用 JobStoreTX 时，你必须在 quartz.properties 文件中指定 Datasource 属性。这允许 Quartz 为你创建并管理 Datasource。表 6.4 列示了使用 JobStoreTX 时需要的 Datasource 配置属性。

表 6.4. 配置 Quartz Datasource 的可用属性

属性	必须
org.quartz.dataSource.NAME.driver	是
描述：JDBC 驱动类的全限名	
org.quartz.dataSource.NAME.URL	是
描述：连接到你的数据库的 URL(主机，端口等)	
org.quartz.dataSource.NAME.user	否
描述：用于连接你的数据库的用户名	
org.quartz.dataSource.NAME.password	否
描述：用于连接你的数据库的密码	
org.quartz.dataSource.NAME.maxConnections	否
描述：DataSource 在连接接中创建的最大连接数	
org.quartz.dataSource.NAME.validationQuary	否
描述：一个可选的 SQL 查询字串，DataSource 用它来侦测并替换失败/断开的连接。例如，Oracle 用户可选用 select table_name from user_tables，这个查询应当永远不会失败，除非直的就是连接不上了。	

表 6.4 中列出的每一个属性，你需要用你选择的名字替换掉属性的 NAME 部分。只要保证 Datasource 的所有属性的 NAME 部分相同就行了。这个名字用于唯一的标识 Datasource。假如你需要配置多个 Datasource (在使用 JobStoreCMT 时你将会这么做)，每一个 Datasource 应该有一个唯一的 NAME 值。

代码 6.2 展示了一个为 JobStoreTX 配置 Datasource 的例子，它需要加到 quartz.properties 文件中

代码 6.2. 一个用于非 CMT 环境的 Quartz Datasource 的例子

```
1. org.quartz.dataSource.myDS.driver = net.sourceforge.jtds.jdbc.Driver
2. org.quartz.dataSource.myDS.URL = jdbc:jtds:sqlserver://localhost:1433/quartz
3. org.quartz.dataSource.myDS.user = admin
4. org.quartz.dataSource.myDS.password = myPassword
5. org.quartz.dataSource.myDS.maxConnections = 10
```

像上面代码 6.2 那样加入了 Datasource 部分到 quartz.properties 文件后，你仍然需要使之对于已配置的 Quartz JobStoreTX 是可用的。你可以通过把下面的属性加到属性文件中来做到这一点：

org.quartz.jobStore.DataSource = <DS_NAME>

这个 <DS_NAME> 应该与指定给个 Datasource 配置的名字相匹配。对于代码 6.2 中的例子来使用 Datasource，你应当在 quartz.properties 文件中加入下面这行：

org.quartz.jobStore.dataSource = myDS

这个值然后会传递给 JobStoreSupport 并且对于你的 JobStoreTX 就可用了，这样连接就可以被获取并传递到 DriverDelegate 实例。

九. 应用 JobStoreTX 运行 Quartz

当你已完成前面的配置步骤时，你的程序就可以准备启动了。正如前面所有的例子那样，你仍然需要一个启动类来从工厂创建一个 Scheduler 实例，并调用它的 start() 方法。一个如代码 6.3 中的类就足够了。

代码 6.3. 简单的启动类，从命令行调用来启动 Scheduler

```
1.  public class SchedulerMain {
2.      static Log logger = LogFactory.getLog(SchedulerMain.class);
3.
4.     public static void main(String[] args) {
5.           SchedulerMain app = new SchedulerMain();
6.           app.startScheduler();
7.       }
8.     public void startScheduler() {
9.           try {
10.               // Create an instance of the Scheduler
11.               Scheduler scheduler =
12.                        StdSchedulerFactory.getDefaultScheduler();
13.
14.              logger.info("Scheduler starting up...");
15.               scheduler.start();
16.
17.          } catch (SchedulerException ex) {
18.                logger.error(ex);
19.          }
20.      }
21.  }
```

当你使用代码 6.3 中的 SchedulerMain 类来测试 JobStoreTX 配置，你将得到类似于如下那样的输出：

```
INFO [main] - Quartz Scheduler v.1.5.0 created.
INFO [main] - Using thread monitor-based data access locking (synchronization).
INFO [main] - Removed 0 Volatile Trigger(s).
INFO [main] - Removed 0 Volatile Job(s).
INFO [main] - JobStoreTX initialized.
INFO [main] - Quartz scheduler 'QuartzScheduler' initialized from default resource
file in Quartz package: 'quartz.properties'
INFO [main] - Quartz scheduler version: 1.5.0
INFO [main] - Scheduler starting up...
INFO [main] - Freed 0 triggers from 'acquired' / 'blocked' state.
INFO [main] - Recovering 0 jobs that were in-progress at the time of the last
shut-down.
INFO [main] - Recovery complete.
INFO [main] - Removed 0 'complete' triggers.
```

```
INFO [main] - Removed 0 stale fired job entries.
INFO [main] - Scheduler QuartzScheduler_$_NON_CLUSTERED started.
```

日志信息是用 Log4J 显示的，因而它们可能和你实际的输出略有差别。一些事情从输出来看是很明显的。首先，在数据库中没有发现 Trigger 或 Job。这是很重要的，有些时候也是使人困惑之处。用了数据库却未给你加载任何的 Job 或 Trigger：是这样的，因为它无法知道谁来为你加载。这是你自己不得不做的事情，你可以几种方式把 Scheduler 信息存入到数据库中。

第六章. Job 存储和持久化（第五部分）

十. 使用数据库存储 Scheduler 信息

·加载 Job 到数据库中

在前面有一节，"使用内存存储 Scheduler 信息"，我们谈到关于在使用 RAMJobStore 时如何加载 Job 和 Trigger 信息到内存中。那么 Job 和 Trigger 又是如何加载到数据库中的呢？存在以下几个方法把 Job 信息存入到数据库：

- 在你的程序中加入 Job 信息
- 使用 JobInitializationPlugin
- 使用 Quartz Web 应用程序

我们在前面的 RAMJobStore 章节中讨论过前面两种途径。当它们用于 JDBC JobStore 时，并没有多大不同，只些许例外。首先，你需要知道，当使用这两个方法式，Job 信息是在数据库中的。甚至在你停止了程序后，这些信息仍然保留在数据库中。甚至是你不在你的程序中使用 JobInitializationPlugin 时，这些信息也还在数据库中。基于这一点，它是会从数据库中找寻 Job 信息。第八章涵盖了 JobInitializationPlugin 和常用的 Quartz 插件。

最后一种方法可能是最有意思的。我们还没有谈论到 Quartz Web 应用，但是我们在第十三章 "Quartz 和 Web 应用" 是这么做的。在现在呢，你应当知道 Quartz Web 应用是一个基于浏览器的 GUI 程序，它是设计用来管理 Quartz Scheduler。它是由 Quartz 用户设计的，它为加入 Job 和 Trigger、启动和暂停 Scheduler 和发布其他功能呈现了一个相当好的界面。

> **通过 SQL 工具加载 Job**
>
> 最后还有一种能用于加载 Job 信息的方法，但仅在这儿提一下，并不鼓励你去尝试它。这种方法是使用本地 SQL 直接操作 Quartz 表来尝试加载和/或修改信息。使用本地查询工具来加入 Job 信息到数据库中只在少数时候这样做，但是很容易破坏数据进而导致所有 Job 不能正确运行。无论如何应尽力避免用这种方法。

十一. 使用 JobStoreCMT

许多我们在前面章节对 JobStoreTX 所说的和做的对于另一版本的 JDBC JobStore，JobStoreCMT 来也是适用的。再说，也没说所有，这不足为奇，因为它们都是 JobStore 类型，它们都是设计成用 JDBC 来与关系型数据库交互。也都是继承自共同的基类。

JobStoreCMT 被设计成参与到容器的事物边界内。这意味着容器创建一个 JTA 事物并使之对于 JobStore 可用。Quartz 与 JobStore 的交互保持在这个事物中。假如出现任何问题，Quartz 能给容器一个信号，它希望通过调用事物的 setRollbackOnly() 使事物回滚。

·配置 JobStoreCMT

同之前的 JobStoreTX 和 RAMJobStore 一样，要使用 JobStoreCMT 的第一步是告知 Scheduler 你打算用 JobStoreCMT。和以前类似，也是通过在 quartz.properties 文件中设置 JobStore 类属性来做到这一点的：

```
org.quartz.jobStore.class = org.quartz.impl.jdbcjobstore.JobStoreCMT
```

如果属性文件中存在 RAMJobStore 行，要确保移除了它。

·配置 DriverDelegate 类

你也是需要像为 JobStoreTX 所做的那样选择 DriverDelegate。Quartz 依靠一个 DriverDelegate 与给定的数据库通信。代理负责了与 JDBC Driver，也就是数据库的所有通信。

回到表 6.2 中的 DriverDelegate 列表，并基于你的数据库平台和环境选择一个。要加 MS SQLServer 代理到 quart.properties 文件，那就加入下一行：

```
org.quartz.jobStore.driverDelegateClass = org.quartz.impl.jdbcjobstore.MSSQLDelegate
```

你可以使几个属性来帮助调整 JobStoreCMT。表 6.5 列举了全部设置项。

表 6.5. 用于设置 JobStoreCMT 的配置属性

属性	默认值

org.quartz.jobStore.driverDelegateClass

描述：能够理解不同数据库系统中特定方言的驱动代理

org.quartz.jobStore.dataSource

描述：这是一个用于 quartz.properties 文件的数据源配置块的名字。

org.quartz.jobStore.nonManagedTXDataSource

描述： JobStoreCMT 需要一个(第二个) 数据源，它所包含的连接不作为容器管理事特的一部分。 这个属性值必须是一个定义在配置属性文件中的数据源的名字。这个数据源必须包含非容器管理事物(non-CMT) 连接，换名话说就是, 它产生的连接可让 Quartz 直接合法的调用它的 commit() 和 rollback() 方法。

org.quartz.jobStore.tablePrefix QRTZ_

描述：这是指定给 Scheduler 的一套数据库表名的前缀。Schedulers 在指定了不同前缀时可在同一数据库中使用不同的表。

org.quartz.jobStore.useProperties False

描述："use properties" 标记指示着持久性 JobStore 所有在 JobDataMap 中的值都是字符串，因此能以名-值 对的形式存储，而不用让更复杂的对象以序列化的形式存入 BLOB 列中。这样会更方便，因为让你避免了发生于序列化你的非字符串的类到 BLOB 时的有关类版本的问题。

org.quartz.jobStore.misfireThreshold 60000

描述：在 Trigger 被认为是错过触发之前，Scheduler 还容许 Trigger 通过它的下次触发时间的毫秒数(译者注：据原文翻译，真的不好理解，实际效果可参看：http://www.blogjava.net/Unmi/archive/2007/10/23/153413.html 我在评论中的实验)。默认值(假如你未在配置中存在这一属性条目) 是 60000(60 秒)。这个不仅限于 JDBC-JobStore；它也可作为 RAMJobStore 的参数

org.quartz.jobStore.isClustered False

描述：设置此为 true 来打开集群特性。假如你有多个 Quartz 实例使用同一套数据库表时这个属性必须设置为 true。

org.quartz.jobStore.clusterCheckinInterval 15000

描述：设置一个频度(毫秒)，用于实例报告给集群中的其他实例。这会影响到侦测失败实例的敏捷度。它只用于设置了 isClustered 为 true 的时候。

org.quartz.jobStore.maxMisfiresToHandleAtATime 20

描述： 这是 JobStore 能处理的错过触发的 Trigger 的最大数量。处理太多(超过两打) 很快会导致数据库表被锁定够长的时间，这样就妨碍了触发别的(还未错过触发) trigger 执行的性能。

org.quartz.jobStore.dontSetAutoCommitFalse False

描述：设置这个参数为 true 则告诉 Quartz 不要调用从 DataSource 获取到的连接的 setAutoCommit (false) 方法. 这在少些情况下是有帮助的、例如你有一个驱动在已是 off 时又调用了这个方法会有所抱怨. 这个属性默认为 false，因为多数驱动需要调用 setAutoCommit(false) 方法。

org.quartz.jobStore.selectWithLockSQL SELECT * FROM {0}LOCKS WHERE LOCK_NAME = ? FOR UPDATE

描述：这必须是一个从 LOCKS 表查询一行并对这行记录加锁的 SQL 语句。假如未设置，默认值就是 SELECT * FROM {0}LOCKS WHERE LOCK_NAME = ? FOR UPDATE，这能在大部分数据库上工作。{0} 会在运行期间被前面你配置的 TABLE_PREFIX 所替换。

org.quartz.jobStore.dontSetNonManagedTX ConnectionAutoCommitFalse False

描述：这个属性同 org.quartz.jobStore.dontSetAutoCommitFalse, 只是它还可以应用于不受管理事物的数据源(nonManagedTXDataSource)。

org.quartz.jobStore.txIsolationLevelSerializable False

描述：值为 True 时告诉 Quartz (当使用 JobStoreTX 或 CMT 时) 调用 JDBC 连接的 setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE) 方法。这能助于防止某些数据库在高负荷和长事物时的锁超时。

org.quartz.jobStore.txIsolationLevelReadCommitted False

描述：当设置为 true 时，这一属性告诉 Quartz 调用不受管理的 JDBC 连接的 setTransactionIsolation (Connection.TRANSACTION_READ_UNCOMMITTED) 方法。这能助于防止某些数据库(如 DB2) 在高负荷和长事物时的锁超时。

第六章. Job 存储和持久化（第六部分）

十二. 为 JobStoreCMT 配置数据源

跟 JobStoreTX 一样，我们需要配置一个 Datasource 才能使用 JobStoreCMT。然而，JobStoreCMT 需要两个 Datasource 而不是像 JobStoreTX 只要一个。其中一个 Datasource 和我们为 JobStoreTX 设置的类同：作为不受管理的数据源。同时呢，我们还需配置第二个数据源，是作为受管理的数据源，它由应用服务器来进行管理。

> **为什么 JobStoreCMT 需要两个 Datasource 呢？**
>
> JobStoreCMT 的原始作者，Jeffrey Wescott，设计 JobStoreCMT 使用一个标准的 JDBC 连接来做它“自己的工作”，同时，代表客户端(如部署 Job) 的工作在执行时是使用一个在容器控制之下有自身事物的 JDBC 连接。即使 Quartz 处在一个大事物中，这种设计也允许用户与 Quartz 交互，而无需 JobStoreCMT 非得使用应用服务器的事物管理器(例如，经由 UserTransaction) 在做自己内部工作时(如处理已错过执行的 Trigger) 来创建和终止事物。如果是 JobStoreCMT 使用 UserTransation 只给它配置一个数据源，从配置方面来看确实方便。然而，在相比于别的特性需求和改进的必要性时，作此变化并不会成为团队中首要的问题，因而 JobStoreCMT 还是继续要两个数据源。

·配置不受管理的数据源

我们在设置不受管理的数据源的多数操作与为 JobStoreTX 所做是相同的，只是我们还要加上一行来指定这是 nonManagedTXDataSource:

```
# Add the property for the nonManagedTXDataSource
org.quartz.jobStore.nonManagedTXDataSource = myDS

org.quartz.dataSource.myDS.driver = net.sourceforge.jtds.jdbc.Driver
org.quartz.dataSource.myDS.URL = jdbc:jtds:sqlserver://localhost:1433/quartz
org.quartz.dataSource.myDS.user = admin
org.quartz.dataSource.myDS.password = myPassword
org.quartz.dataSource.myDS.maxConnections = 10
```

这是配置不受管理的数据源，并让 JobStore 知道这个 nonManagedTXDataSource 叫做 "myDS"。

·配置受管理的 Datasource

第二个数据源需配置为一个受管理的 Datasource。这意味着 Quartz 在执行 Scheduler 操作时使用一个容器已创建好的 Datasorce 与数据库交互。当 Quartz 从 Datasource 上取得了连接后，在 Quartz 部署 Job 和 Trigger 时应有一个 JTA 事物。例如，代码要求 Quartz 在 SessionBean 的一个方法上的事物描述符设置为 REQUIRED。另一个应用是客户端程序要通过使用 javax.transaction.UserTransaction 直接启动一个事物。

和不受管理的 Datasoure 一样，也是要在 quartz.properties 文件中配置容器管理的 Datasource。下面的例子描述了如何设置受管理的 Datasource:

```
org.quartz.dataSource.NAME.jndiURL=jdbc/quartzDS

org.quartz.dataSource.NAME.java.naming.factory.initial=weblogic.jndi.WLInitialContextFactory

org.quartz.dataSource.NAME.java.naming.provider.url=t3://localhost:7001
org.quartz.dataSource.NAME.java.naming.security.principal=weblogic
org.quartz.dataSource.NAME.java.naming.security.credentials=weblogic
```

表 6.6 列出了受管理的 Datasource 可用的属性。

表 6.6. 在应用服务器上所用的 Datasource 的属性

属性	必须
org.quartz.dataSource.NAME.jndiURL	是
描述：受你的应用服务器管理的 DataSource 的 JNDI URL	
org.quartz.dataSource.NAME.java.naming.factory.initial	否
描述：可选项，你想用的 JNDI InitialContextFactory 的类名称	
org.quartz.dataSource.NAME.java.naming.provider.url	否
描述：可选项，连接到 JNDI 上下文的 URL	
org.quartz.dataSource.NAME.java.naming.security.principal	否
描述：可选项，连接到 JNDI 上下文的用户主体(Unmi 注：用户名)	
org.quartz.dataSource.NAME.java.naming.security.credential	否
描述：可选项，连接到 JNDI 上下文的用户凭证(Unmi 注：密码)	

使用到表 6.6 中的属性，这儿有一个在 quartz.properties 中配置受管理的 Datasource 的例子。

```
org.quartz.dataSource.WL.jndiURL = OraDataSource
org.quartz.dataSource.WL.jndiAlwaysLookup = DB_JNDI_ALWAYS_LOOKUP
org.quartz.dataSource.WL.java.naming.factory.initial = weblogic.jndi.WLInitialContextFactory
org.quartz.dataSource.WL.java.naming.provider.url = t3://localhost:7001
org.quartz.dataSource.WL.java.naming.security.principal = weblogic
org.quartz.dataSource.WL.java.naming.security.credentials = weblogic
```

[Unmi 后记] 又半月有途未动笔墨了，不过仍然保持着只要有机会上网就一定到 BlogJava 来阅读最新本章的习惯。部门人员锐减，事情也就杂乱不堪起来，对于开发技术的选择上客观上完全能由我自行决断，没有人与我争议了，甚是悲凉。

起初对 Quartz 这个系列的翻译只为一时之兴，一路过来翻译工作其实蛮费时间的。一篇英文阅读完之后，还需花费几近于 20 倍的时间才能用中文记录下来。因为阅读总是眼观六路，一知半解的，完全转换成中文就要字句斟酌了。

翻译进行到这个阶段，我当然还会继续坚持，从其中获得的好处也是不言而喻的。主要表现在两方面：

1. 对技术把握的更精细。阅读是放眼而瞟，只求个大概；翻译则不同，本身未能理解个相当，何以能用中文向他人解译的清楚呢？蕴责任于其中。对于多数例子，并非照搬了事，都有再次测试感受过的。译章置于网上之后，亦有许多朋友就 Quartz 使用时的疑问，本人也会带着某种责任心，尽我能力作解答，也非常有助于自身对该项技术的掌握。

2. 阅读与翻译的速度提升也是显而易见的。最初时的每字每句的爬梳，须频繁请求各方资源才能完成一篇，现在与那时相比，可谓顺畅多了。许多篇章纵使离开英文词典也无碍了。用数据来说吧，现在翻译一篇的速度大概是以前的四至五倍。以后的前行中需要面对更多的英文资料，通过对 Quartz 这个手册翻译扎实锤炼了自己的英文阅读能力，写作能力亦在其内。

之于以上两点能对我产生的影响，尤其是 Quartz 手册的翻译已完成大部分时，我一定还会继续完成它，也非常感谢网络上各位朋友们的支持。

第六章. Job 存储和持久化（第七部分）

十三. 改善持久性 JobStore 的性能

当在只有最少量时间做任何相关事情的时候，性能是一个广受人瞩目的课题。作为有经验的开发者，我们知道从项目之初它就成为一个需要考虑的事。

在使用 Quartz 的 JobStore 时，最大的关注面就是有关于与关系型数据库的交互。数据库 I/O(就像文件 I/O) 通常不是很快。你可以通过采取一些措施，如调优 SQL、增加索引和操作表和列等来改善性能。因为性能问题在写 Quartz 框架的时候就已有考虑到，而你又不想在未出现实际的性能问题时扎入到 Quartz 中做些手脚，那么可试图通过配置来解决它，或者尝试所有可能的方式，只要不是维护源代码。最好的消息是 Quartz 是开源的，你完全可以窥入其中了解它做了什么和如何实现的。假如你不喜欢它现有的查询数据库的方式，你有权去修正它。不过，在你采取行动之前，确定检查了 Quartz 论坛上的用户和开发者，看看是否其他人也遇到了相关的问题并浏览推荐的做法。

一个很简单(也是很有效的) 改善性能的方式是确保在所有适当的列上创建了索引。Quartz 所带的某些数据库创建脚本已有创建索引的命令。如果你的没有，你可以简单参考定义在 tables_oracle_sql 底端的语句并针对你的 RDBM 需要作些语法上改变。

不管你怎么做的，假如你修改了 Quartz 来改善性，一定要反馈到社区和 Quartz 项目。

十四. 创建新的 JobStore

对于多数用户，JobStore 所提供开箱即用的实现已是足够了。当你的应用在重启之间不需要维护状态，那么 RAMJobStore 就是你的第一选择。它速度快，易于配置，也不会带来什么麻烦。另一方面，如果你需要在重启之间维护 Scheduler 的状态，且正使用一个数据库或可以访问数据，那么用一个 JDBC JobStore 或许是你最后的选择。

那会在什么时候你需要一个完全不同的 JobStore 类型？你将需要创建一个新的类型。本节讨论几种途径，并针对如何创建一个新的 JobStore ，在所提供的方案无法满足你时给出一些主意。

·实现 JobStore 接口

无论它们是否是用的数据库、文件系统、甚至是内存，所有的 JobStore 必须实现 JobStore 接口。你创建的新的 JobStore 也不例外。回头看本章前面部分，你会发现 RAMJobStore 直接实现了 JobStore 接口，JDBC JobStore 是 JobStroeSupport 的子类，它本身实现了 JobStore 接口。

JobStore 接口有 40 个方法，它要求任何 JobStroe 实现都必须实现这些方法，你的也一样。你如何实现那些方法完全依赖于你正构建的 JobStore 的类型。那不是说你的 JobStore 将只能有 40 个方法；这仅仅是接口需要的最小数量。这 40 个方法体现 JobStore 和 Scheduler 之间的公共契约。

让我们拣出其中一个方法来简短的讨论它。我们就选 JobStore 接口方法：

```
public void schedulerStarted() throws SchedulerException;
```

Scheduler 调用 JobStore 的 SchedulerStarted() 方法去通知 JobStore Scheduler 已经启动了。如果你看了 RAMJobStore 的实现，你能发现它在这个方法实现中什么也没做：

```
public void schedulerStarted() throws SchedulerException{
   // nothing to do
}
```

然而，假如你去看那两个 JobStore 的实现，你会看到在 Scheduler 在首次启动时进行了一些工作：

```
1.  public void schedulerStarted() throws SchedulerException {
2.
3.      if (isClustered()) {
4.          clusterManagementThread = new ClusterManager(this);
```

```
            clusterManagementThread.initialize();
        } else {
            try {
                recoverJobs();
            } catch (SchedulerException se) {
                throw new SchedulerConfigException("Failure occurred during job recovery.", se);
            }
        }
        misfireHandler = new MisfireHandler(this);
        misfireHandler.initialize();
    }
```

每一个 JobStore 实现会是唯一的，在接口方法内部实现的功能也是不同的。如果你是认真的去创建你自己的 JobStore，你就应当好好看看 RAMJobStore 的源代码来完全理解 JobStore 所有职责。RAMJobStore 应该作为你需要定制任何 JobStore 时的指南。

第七章. 实现 Quartz 监听器（第一部分）

第七章. 实现 Quartz 监听器

在某个所关注事件发生时，监听器提供了一种方便且非侵入性的机制来获得这一通知。Quartz 提供了三种类型的监听器：监听 Job 的，监听 Trigger 的，和监听 Scheduler 自己的。本章解释如何应用每一种类型来更好的管理你的 Quartz 应用，并获悉到什么事件正在发生。

一. 监听器作为扩展点

术语 "*扩展点*" 在软件开发中用于指示框架或应用的某个位置，在这一位置在创建者期望用户扩展或定制这一框架来适合于他们的需要。(你也将会听到 *hook*(钩子) 一词，是一样意思的)

Quartz 监听器是某一类型的扩展点，在这里你，作为一个 Quartz 用户，可以扩展这框架并定制它来做些新的事情。定制化发生成监听类的实现当中，我们会在本章中告诉你如何构建它。

监听器并非框架中仅有的扩展点。还有插件和一些其他的定制选项，不过监听器提供了一个简单的方式来定制框架，使之做你需要它做的事情。因为针对于监听器的扩展点是通过公有化接口来支持，所以你用不着担心创建了你自己的分支代码，到后来又不被支持的情况。

二. 实现监听

在接下来的讨论中，实现监听器的方法通用于所有的三种类型。可以分成以下步骤：

1. 创建一个 Java 类，实现监听器接口

2. 用你的应用中特定的逻辑实现监听器接口的所有方法

3. 注册监听器

·创建监听器类

监听器是一个 Java 接口，它必须由一个具体类来实现。你也不需要只为这个目的创建一个专门的类；它可以是任何一个你希望能接收到方法回调的类。为符合良好的程序设计，你应当注意保持高内聚和松耦合性。认真的考虑哪个类你决定用于实现监听器类；这从总体设计的视角来看是很重要的。

·实现监听器方法

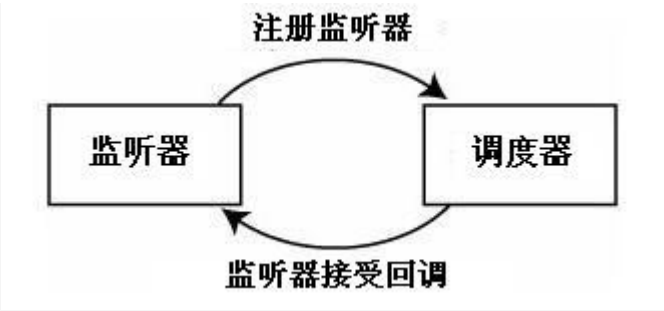
因为监听器是普通的 Java 接口，每个方法都必须在你的监听器实现类中实现。假如有一些监听器接口方法你不感兴趣，允许你使用空的方法体；只是，你仍必须提供一个有效的方法实现它。例如，下面的代码片断显示了 SchedulerListerner 中一个方法的空方法体：

```
SchedulerListener methods:

... rest of the SchedulerListener not shown

public void schedulerShutdown(){
  // Don't care about the shutdown event
}
```

·注册监听器

要接收到方法回调，Scheduler 必须能获知监听实例。图 7.1 描绘了用 Scheduler 注册一个监听器和接受回调的过程。

图 7.1. 监听器被注册并接受 Scheduler 的回调



全局之于非全局监听器

JobListener 和 TriggerListener 可被注册为全局或非全局监听器。一个全局监听器能接收到所有的 Job/Trigger 的事件通知。而一个非全局监听器(或者说是一个标准的监听器) 只能接收到那些在其上已注册了监听器的 Job 或 Triiger 的事件。

你要注册你的监听器为全局或非全局的需依据你特定的应用需要。我们在以下章节中提供了两种方式的例子。从另一方面来认识全局和非全局的监听器是来自于 Quartz 框架的创建者。James House 在描述全局和非全局监听器时是这样的：

> 全局监听器是主动意识的，它们为了执行它们的任务而热切的去寻找每一个可能的事件。通常，全局监听器要做的工作不用指定到特定的 Job 或 Trigger。非全局监听器一般是被动意识的，它们在所关注的 Trigger 激发之前或是 Job 执行之前什么事也不做。因此，非全局的监听器比起全局监听器而言更适合于修改或增加 Job 执行的工作。这有点像知名的装饰设计模式的装饰器。

第七章. 实现 Quartz 监听器（第二部分）

三. 监听 Job 事件

org.quartz.JobListener 接口包含一系列的方法，它们会由 Job 在其生命周期中产生的某些关键事件时被调用。JobListener 可用的方法显示在代码 7.1 中。

代码 7.1. org.quartz.JobListener 接口中的方法

```
1.  public interface JobListener {
2.     public String getName();
3.     public void jobToBeExecuted(JobExecutionContext context);
4.     public void jobExecutionVetoed(JobExecutionContext context);
5.
6.     public void jobWasExecuted(JobExecutionContext context,
7.             JobExecutionException jobException);
8.  }
```

JobListener 接口中的方法用途是十分明了的。然后，我们还是要对他们加以简单说明。

·getName() 方法

getName() 方法返回一个字符串用以说明 JobListener 的名称。对于注册为全局的监听器，getName() 主要用于记录日志，对于由特定 Job 引用的 JobListener，注册在 JobDetail 上的监听器名称必须匹配从监听器上 getName() 方法的返回值。在你看完一些例子之后就会很清楚了。

·jobToBeExecuted() 方法

Scheduler 在 JobDetail 将要被执行时调用这个方法。

·jobExecutionVetoed() 方法

Scheduler 在 JobDetail 即将被执行，但又被 TriggerListener 否决了时调用这个方法。

·jobWasExecuted() 方法

Scheduler 在 JobDetail 被执行之后调用这个方法。

代码 7.2 展示了一个很简单的 JobListener 实现。

代码 7.2. 一个简单的 JobListner 实现

```
1.  package org.cavaness.quartzbook.chapter7;
2.
3.  import org.apache.commons.logging.Log;
4.  import org.apache.commons.logging.LogFactory;
5.  import org.quartz.JobExecutionContext;
6.  import org.quartz.JobExecutionException;
7.  import org.quartz.JobListener;
8.
9.  public class SimpleJobListener implements JobListener {
10.      Log logger = LogFactory.getLog(SimpleJobListener.class);
11.
12.      public String getName() {
13.          return getClass().getSimpleName();
```

```
    }

    public void jobToBeExecuted(JobExecutionContext context) {
        String jobName = context.getJobDetail().getName();
        logger.info(jobName + " is about to be executed");
    }
    public void jobExecutionVetoed(JobExecutionContext context) {
        String jobName = context.getJobDetail().getName();
        logger.info(jobName + " was vetoed and not executed()");
    }
    public void jobWasExecuted(JobExecutionContext context,
            JobExecutionException jobException) {

        String jobName = context.getJobDetail().getName();
        logger.info(jobName + " was executed");
    }
}
```

代码 7.2 中的 JobListener 打印一个日志消息，很明显，只是监听器最基本的用法。你要实现的逻辑完全由你和你的应用需要而定。你也许想在 Job 成功完成后发送一个电子邮件，或者在 Job 被否决后部署另一个。你有在回调方法中执行几乎任何动作的自由。

前面，我们提到过 JobListener (和 TriggerListener) 能注册为全局或非全局的。注意了，我们并不需要事先知道在代码 7.2 中的 JobListener 是一个全局或是非全局的；我们仅仅是实现了接口和提供了监听器方法。代码 7.3 描绘了如何使用代码 7.2 中的 SimpleJobListner 使之注册为一个全局的 JobListener。

代码 7.3. 使用 SimpleJobListener 作为一个全局 JobListener

```
package org.cavaness.quartzbook.chapter7;

import java.util.Date;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.cavaness.quartzbook.common.PrintInfoJob;
import org.quartz.JobDetail;
import org.quartz.JobListener;
import org.quartz.Scheduler;
import org.quartz.SchedulerException;
import org.quartz.Trigger;
import org.quartz.TriggerUtils;
import org.quartz.impl.StdSchedulerFactory;

public class Listing_7_3 {
    static Log logger = LogFactory.getLog(Listing_7_3.class);

    public static void main(String[] args) {
        Listing_7_3 example = new Listing_7_3();

        try {
            example.startScheduler();
        } catch (SchedulerException ex) {
            logger.error(ex);
        }
    }

    public void startScheduler() throws SchedulerException {

        // Create an instance of the factory
        Scheduler scheduler = null;
```

```
33.
34.            // Create the scheduler and JobDetail
35.            scheduler = StdSchedulerFactory.getDefaultScheduler();
36.            JobDetail jobDetail = new JobDetail("PrintInfoJob",
37.                      Scheduler.DEFAULT_GROUP, PrintInfoJob.class);
38.
39.           /*
40.            * Set up a trigger to start firing now, with no end
41.            * date/time, repeat forever and have
42.            * 10 secs (10000 ms) between each firing.
43.            */
44.           Trigger trigger = TriggerUtils.makeSecondlyTrigger(10);
45.           trigger.setName("SimpleTrigger");
46.           trigger.setStartTime(new Date());
47.
48.           // Register the JobDetail and Trigger
49.           scheduler.scheduleJob(jobDetail, trigger);
50.
51.           // Create and register the global job listener
52.           JobListener jobListener =
53.                new SimpleJobListener("SimpleJobListener");
54.
55.           scheduler.addGlobalJobListener(jobListener);
56.           // Start the scheduler
57.           scheduler.start();
58.           logger.info("Scheduler was started at " + new Date());
59.       }
60.   }
```

代码 7.3 中的代码现在看来是相当直截的。创建了一个 JobDetail 和 Trigger 并注册到了 Scheduler 实例上，这在前面我们已是做过许多次了。

代码 7.2 中的 SimpleJobListener 初始化后通过 Scheduler 调用 addGlobalJobListener() 方法注册为一个全局的 JobListener。最后，启动 Scheduler。

因为我们只配置了单个 Job (PrintInfoJob)，我们获得回调也只是那个 JobDetail。不过，假如我们部署了其他 Job，我们也能看到第二个 Job 的回调日志信息，因为这个监听顺是配置为全局的。

·注册非全局的 JobListener

你还能使用代码 7.2 中的 SimpleJobListener 作为一个非全局的 JobListener。要做到这点，你仅需要修改代码 7.3 的 startScheduler() 方法中的代码。代码 7.4 显示了这一需要做的小小的改变。

代码 7.4. 使用 SimpleJobListener 作为非全局的 JobListener

```
1.  package org.cavaness.quartzbook.chapter7;
2.
3.  import java.util.Date;
4.
5.  import org.apache.commons.logging.Log;
6.  import org.apache.commons.logging.LogFactory;
7.  import org.cavaness.quartzbook.common.PrintInfoJob;
8.  import org.quartz.JobDetail;
9.  import org.quartz.JobListener;
10. import org.quartz.Scheduler;
11. import org.quartz.SchedulerException;
12. import org.quartz.Trigger;
13. import org.quartz.TriggerUtils;
14. import org.quartz.impl.StdSchedulerFactory;
```

```
15.
16.  public class Listing_7_4 {
17.       static Log logger = LogFactory.getLog(Listing_7_4.class);
18.
19.       public static void main(String[] args) {
20.            Listing_7_4 example = new Listing_7_4();
21.
22.            try {
23.                 example.startScheduler();
24.            } catch (SchedulerException ex) {
25.                 logger.error(ex);
26.            }
27.       }
28.
29.       public void startScheduler() throws SchedulerException {
30.
31.            Scheduler scheduler = null;
32.
33.            try {
34.                // Create the scheduler and JobDetail
35.                scheduler = StdSchedulerFactory.getDefaultScheduler();
36.                JobDetail jobDetail =
37.                          new JobDetail("PrintInfoJob",
38.                          Scheduler.DEFAULT_GROUP,
39.                          PrintInfoJob.class);
40.
41.                /*
42.                 * Set up a trigger to start firing now, with no end
43.                 * date/time, repeat forever and have
44.                 * 10 secs (10000 ms) between each firing.
45.                 */
46.                Trigger trigger =
47.                     TriggerUtils.makeSecondlyTrigger(10);
48.
49.                trigger.setName("SimpleTrigger");
50.                trigger.setStartTime(new Date());
51.
52.                // Create the job listener
53.                JobListener jobListener =
54.                     new SimpleJobListener("SimpleJobListener");
55.
56.                // Register Listener as a nonglobal listener
57.                scheduler.addJobListener(jobListener);
58.
59.                // Listener set on JobDetail before scheduleJob()
60.                     jobDetail.addJobListener(jobListener.getName());
61.
62.                // Register the JobDetail and Trigger
63.                scheduler.scheduleJob(jobDetail, trigger);
64.
65.                // Start the scheduler
66.                scheduler.start();
67.                logger.info("Scheduler started at " + new Date());
68.
69.            } catch (SchedulerException ex) {
70.                logger.error(ex);
71.            }
72.       }
73.  }
```

代码 7.4 很类似于代码 7.4 中的代码。因为 JobListener 是要注册为一个非全局的监听器，你就要调用 Scheduler 的

addJobListener() 方法而不是 addGlobalJobListener() 方法了。对于非全局的 JobListener，它应于任何引用到它的 JobDetail 使用 schedulerJob() 或 addJob() 方法注册之前被注册。

接下来，JobListener 的名字要设置给 JobDetail。注意，设置的不是 JobListener 实例，仅仅是它的名称。这是通过调用 addJobListener() 方法并传入名称来完成的。传递给 addJobListener() 方法的名称必须匹配从监听器的 getName() 方法返回的名称。如果 Scheduler 不能根据名称找到监听器，它会抛出一个 SchedulerException 异常。

最后，启动 Scheduler。

非全局 JobListener 相关步骤的顺序

加入一个非全局 JobListener 的步骤必须是依序完成。JobListener 必须首先加入到 Scheduler 中。接着，JobListener 才能够设置给 JobDetail 对象。之后，你就能使用 scheduleJob() 方法安全的把 JobDetail 加入到 Scheduler 中。

第七章. 实现 Quartz 监听器（第三部分）

四. 监听 Trigger 事件

正如 JobListener, org.quartz.TriggerListener 接口也包含一系列给 Scheduler 调用的方法。然而，与 JobListener 有所不同的是， TriggerListener 接口还有关于 Trigger 实例生命周期的方法。代码 7.5 列出了 TriggerListener 接口的方法。

代码 7.5. org.quartz.TriggerListener 接口的方法

```
1.  public interface TriggerListener {
2.      public String getName();
3.
4.      public void triggerFired(Trigger trigger,
5.           JobExecutionContext context);
6.
7.      public boolean vetoJobExecution(Trigger trigger,
8.            JobExecutidonContext context);
9.
10.     public void triggerMisfired(Trigger trigger);
11.
12.     public void triggerComplete(Trigger trigger,
13.           JobExecutionContext context,
14.           int triggerInstructionCode);
15. }
```

·getName() 方法

和前面的 JobListener 一样，TriggerListner 接口的 getName() 返回一个字符串用以说明监听器的名称。对于非全局的 TriggerListener，在 addTriggerListener() 方法中给定的名称必须与监听器的 getName() 方法返回值相匹配。

·triggerFired() 方法

当与监听器相关联的 Trigger 被触发，Job 上的 execute() 方法将要被执行时，Scheduler 就调用这个方法。在全局 TriggerListener 情况下，这个方法为所有 Trigger 被调用。

·vetoJobExecution() 方法

在 Trigger 触发后，Job 将要被执行时由 Scheduler 调用这个方法。TriggerListener 给了一个选择去否决 Job 的执行。假如这个方法返回 true，这个 Job 将不会为此次 Trigger 触发而得到执行。

·triggerMisfired() 方法

Scheduler 调用这个方法是在 Trigger 错过触发时。如这个方法的 JavaDoc 所指出的，你应该关注此方法中持续时间长的逻辑：在出现许多错过触发的 Trigger 时，长逻辑会导致骨牌效应。你应当保持这上方法尽量的小。

·triggerComplete() 方法

Trigger 被触发并且完成了 Job 的执行时，Scheduler 调用这个方法。这不是说这个 Trigger 将不再触发了，而仅仅是当前 Trigger 的触发(并且紧接着的 Job 执行) 结束时。这个 Trigger 也许还要在将来触发多次的。

代码 7.6 展示了一个很简单的 TriggerListener 实现

代码 7.6. 一个简单的 TriggerListener 实现

```
1.  package org.cavaness.quartzbook.chapter7;
```

```
2.
3.  import org.apache.commons.logging.Log;
4.  import org.apache.commons.logging.LogFactory;
5.  import org.quartz.JobExecutionContext;
6.  import org.quartz.Trigger;
7.  import org.quartz.TriggerListener;
8.
9.  public class SimpleTriggerListener implements TriggerListener {
10.      Log logger = LogFactory.getLog(SimpleTriggerListener.class);
11.
12.      private String name;
13.
14.      public SimpleTriggerListener(String name) {
15.          this.name = name;
16.      }
17.
18.      public String getName() {
19.           return name;
20.      }
21.
22.      public void triggerFired(Trigger trigger,
23.           JobExecutionContext context) {
24.
25.           String triggerName = trigger.getName();
26.           logger.info(triggerName + " was fired");
27.      }
28.
29.      public boolean vetoJobExecution(Trigger trigger,
30.                JobExecutionContext context) {
31.
32.           String triggerName = trigger.getName();
33.           logger.info(triggerName + " was not vetoed");
34.           return false;
35.      }
36.
37.      public void triggerMisfired(Trigger trigger) {
38.          String triggerName = trigger.getName();
39.          logger.info(triggerName + " misfired");
40.      }
41.
42.      public void triggerComplete(Trigger trigger,
43.          JobExecutionContext context,
44.          int triggerInstructionCode) {
45.
46.          String triggerName = trigger.getName();
47.          logger.info(triggerName + " is complete");
48.      }
49.  }
```

正如代码7.2 中的 JobListener 一样，代码 7.6 中的 TriggerListener 也是初步的。 它不过是在 Scheduler 调用它的方法时打印了一条日志信息。代码 7.7 中代码测试了这个简单的 TriggerListener。

代码 7.7. 使用 SimpleTriggerListener 作为一个全局的 TriggerListener

```
1.  package org.cavaness.quartzbook.chapter7;
2.
3.  import java.util.Date;
4.
5.  import org.apache.commons.logging.Log;
6.  import org.apache.commons.logging.LogFactory;
```

```
import org.cavaness.quartzbook.common.PrintInfoJob;
import org.quartz.JobDetail;
import org.quartz.Scheduler;
import org.quartz.SchedulerException;
import org.quartz.Trigger;
import org.quartz.TriggerListener;
import org.quartz.TriggerUtils;
import org.quartz.impl.StdSchedulerFactory;

public class Listing_7_7 {
     static Log logger = LogFactory.getLog(Listing_7_7.class);

     public static void main(String[] args) {
          Listing_7_7 example = new Listing_7_7();
          try {
               example.startScheduler();
          } catch (SchedulerException ex) {
               logger.error(ex);
          }
    }

    public void startScheduler() throws SchedulerException {

          // Create an instance of the factory
          Scheduler scheduler = null;

          // Create the scheduler and JobDetail
          scheduler = StdSchedulerFactory.getDefaultScheduler();
          JobDetail jobDetail = new JobDetail("PrintInfoJob",
                    Scheduler.DEFAULT_GROUP, PrintInfoJob.class);

          // Create and register the global job listener
          TriggerListener triggerListener =
               new SimpleTriggerListener("SimpleTriggerListener");

          scheduler.addGlobalTriggerListener(triggerListener);

          /*
           * Set up a trigger to start firing now, with no end
           * date/time, repeat forever and have 10 secs
           * (10000 ms) between each firing.
           */
          Trigger trigger = TriggerUtils.makeSecondlyTrigger(10);
          trigger.setName("SimpleTrigger");
          trigger.setStartTime(new Date());

          // Register the JobDetail and Trigger
          scheduler.scheduleJob(jobDetail, trigger);

          // Start the scheduler
          scheduler.start();
          logger.info("Scheduler was started at " + new Date());
    }
}
```

代码 7.7 显示了如何注册 SimpleTriggerListener 为一个全局的 TriggerListener。它看起来与代码 7.3 中用来注册一个全局 JobListener 的代码完全相似。你只需要调用 addGloabelTriggerListener() 方法并传入这个 TriggerListener 实例。

·注册为非全局的 TriggerListener

要注册为一个非全局的 TriggerListener，你必须调用 addTriggerListener() 方法并传入这个 TriggerListener 实例。接着调用 Trigger 实例的 addTriggerListener() 方法并传入这个 TriggerListener 的名称。

在代码 7.8 中展示了这一过程。

代码 7.8. 使用一个非全局的 TriggerListener

```
1.  package org.cavaness.quartzbook.chapter7;
2.
3.  import java.util.Date;
4.
5.  import org.apache.commons.logging.Log;
6.  import org.apache.commons.logging.LogFactory;
7.  import org.cavaness.quartzbook.common.PrintInfoJob;
8.  import org.quartz.JobDetail;
9.  import org.quartz.Scheduler;
10. import org.quartz.SchedulerException;
11. import org.quartz.Trigger;
12. import org.quartz.TriggerListener;
13. import org.quartz.TriggerUtils;
14. import org.quartz.impl.StdSchedulerFactory;
15.
16. public class Listing_7_8 {
17.      static Log logger = LogFactory.getLog(Listing_7_8.class);
18.
19.      public static void main(String[] args) {
20.           Listing_7_8 example = new Listing_7_8();
21.
22.           try {
23.                example.startScheduler();
24.           } catch (SchedulerException ex) {
25.                logger.error(ex);
26.           }
27.     }
28.
29.     public void startScheduler() throws SchedulerException {
30.
31.          // Create an instance of the factory
32.          Scheduler scheduler = null;
33.
34.          // Create the scheduler and JobDetail
35.          scheduler = StdSchedulerFactory.getDefaultScheduler();
36.          JobDetail jobDetail = new JobDetail("PrintInfoJob",
37.                    Scheduler.DEFAULT_GROUP, PrintInfoJob.class);
38.
39.          // Create and register the nonglobal job listener
40.          TriggerListener triggerListener =
41.               new SimpleTriggerListener("SimpleTriggerListener");
42.
43.          scheduler.addTriggerListener( triggerListener );
44.
45.          /*
46.           * Set up a trigger to start firing now, with no end
47.           * date/time, repeat forever and have 10 secs
48.           * (10000 ms) between each firing.
49.           */
50.          Trigger trigger = TriggerUtils.makeSecondlyTrigger(10);
51.          trigger.setName("SimpleTrigger");
52.          trigger.setStartTime(new Date());
53.
```

```
54.             // Set the listener name for the trigger
55.             trigger.addTriggerListener( triggerListener.getName() );
56.
57.             // Register the JobDetail and Trigger
58.             scheduler.scheduleJob(jobDetail, trigger);
59.
60.             // Start the scheduler
61.             scheduler.start();
62.             logger.info("Scheduler was started at " + new Date());
63.         }
64.     }
```

针对于前面的非全局 JobListener 提到的相同的警告可以应用到这里来；你必须在把它设置给 Trigger 实例并存储了 Trigger 之前把 TriggerListener 加入到 Scheduler 中。

第七章. 实现 Quartz 监听器（第四部分）

五. 监听 Scheduler 事件

org.quartz.SchedulerListener 接口包含了一系列的回调方法，它们会在 Scheduler 的生命周期中有关键事件发生时被调用。代码 7.9 列出了包括在 SchedulerListener 接口的方法。

代码 7.9. org.quartz.SchedulerListener 接口中的方法

```
1.  public interface SchedulerListener {
2.      public void jobScheduled(Trigger trigger);
3.      public void jobUnscheduled(String triggerName, String triggerGroup);
4.      public void triggerFinalized(Trigger trigger);
5.      public void triggersPaused(String triggerName, String triggerGroup);
6.      public void triggersResumed(String triggerName,String triggerGroup);
7.      public void jobsPaused(String jobName, String jobGroup);
8.      public void jobsResumed(String jobName, String jobGroup);
9.      public void schedulerError(String msg, SchedulerException cause);
10.     public void schedulerShutdown();
11. }
```

正如你从代码 7.9 中列示看到的方法那样，SchedulerListener 是在 Scheduler 级别的事件产生时得到通知，不管是增加还是移除 Scheduler 中的 Job，或者是 Scheduler 遭遇到了严重的错误时。那些事件多是关于对 Scheduler 管理的，而不是专注于 Job 或 Trigger 的。

· jobScheduled() 和 jobUnscheduled() 方法

Scheduler 在有新的 JobDetail 部署或卸载时调用这两个中的相应方法。

· triggerFinalized() 方法

当一个 Trigger 来到了再也不会触发的状态时调用这个方法。除非这个 Job 已设置成了持久性，否则它就会从 Scheduler 中移除。

·triggersPaused() 方法

Scheduler 调用这个方法是发生在一个 Trigger 或 Trigger 组被暂停时。假如是 Trigger 组的话，triggerName 参数将为 null。

·triggersResumed() 方法

Scheduler 调用这个方法是发生成一个 Trigger 或 Trigger 组从暂停中恢复时。假如是 Trigger 组的话，triggerName 参数将为 null。

·jobsPaused() 方法

当一个或一组 JobDetail 暂停时调用这个方法。

·jobsResumed() 方法

当一个或一组 Job 从暂停上恢复时调用这个方法。假如是一个 Job 组，jobName 参数将为 null。

·schedulerError() 方法

在 Scheduler 的正常运行期间产生一个严重错误时调用这个方法。错误的类型会各式的，但是下面列举了一些错误例子：

·初始化 Job 类的问题

·试图去找到下一 Trigger 的问题

·JobStore 中重复的问题

·数据存储连接的问题

你可以使用 SchedulerException 的 getErrorCode() 或者 getUnderlyingException() 方法或获取到特定错误的更详尽的信息。

·schedulerShutdown() 方法

Scheduler 调用这个方法用来通知 SchedulerListener Scheduler 将要被关闭。

代码 7.10 展示了一个 SchedulerListener 实现

代码 7.10. 一个简单的 SchedulerListener 实现

```
1.  package org.cavaness.quartzbook.chapter7;
2.
3.  import org.apache.commons.logging.Log;
4.  import org.apache.commons.logging.LogFactory;
5.  import org.quartz.SchedulerException;
6.  import org.quartz.SchedulerListener;
7.  import org.quartz.Trigger;
8.
9.  public class SimpleSchedulerListener implements SchedulerListener {
10.      Log logger = LogFactory.getLog(SimpleSchedulerListener.class);
11.
12.      public void jobScheduled(Trigger trigger) {
13.           String jobName = trigger.getJobName();
14.           logger.info(jobName + " has been scheduled");
15.      }
16.
17.      public void jobUnscheduled(String triggerName,
18.           String triggerGroup) {
19.
20.           if (triggerName == null) {
21.                // triggerGroup is being unscheduled
22.                logger.info(triggerGroup + " is being unscheduled");
23.           } else {
24.                logger.info(triggerName + " is being unscheduled");
25.           }
26.      }
27.
28.      public void triggerFinalized(Trigger trigger) {
29.           String jobName = trigger.getJobName();
30.           logger.info("Trigger is finished for " + jobName);
31.      }
32.
33.      public void triggersPaused(String triggerName,
34.           String triggerGroup) {
35.
36.           if (triggerName == null) {
37.                // triggerGroup is being unscheduled
38.                logger.info(triggerGroup + " is being paused");
39.           } else {
```

```
            logger.info(triggerName + " is being paused");
        }
    }

    public void triggersResumed(String triggerName,
        String triggerGroup) {

        if (triggerName == null) {
            // triggerGroup is being unscheduled
            logger.info(triggerGroup + " is now resuming");
        } else {
            logger.info(triggerName + " is now resuming");
        }
    }

    public void jobsPaused(String jobName, String jobGroup) {
        if (jobName == null) {
            // triggerGroup is being unscheduled
            logger.info(jobGroup + " is pausing");
        } else {
            logger.info(jobName + " is pausing");
        }
    }

    public void jobsResumed(String jobName, String jobGroup) {
        if (jobName == null) {
            // triggerGroup is being unscheduled
            logger.info(jobGroup + " is now resuming");
        } else {
            logger.info(jobName + " is now resuming");
        }
    }

    public void schedulerError(String msg, SchedulerException cause) {
        logger.error(msg, cause.getUnderlyingException());
    }

    public void schedulerShutdown() {
        logger.info("Scheduler is being shutdown");
    }
}
```

和前面的例子一样，代码 7.10 中的 SimpleSchedulerListener 只提供了监听方法的简单实现。代码 7.11 使用了 SimpleSchedulerListener 类。

代码 7.11. 使用 SimpleSchedulerListener

```
package org.cavaness.quartzbook.chapter7;

import java.util.Date;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.cavaness.quartzbook.common.PrintInfoJob;
import org.quartz.JobDetail;
import org.quartz.Scheduler;
import org.quartz.SchedulerException;
import org.quartz.SchedulerListener;
import org.quartz.Trigger;
import org.quartz.TriggerUtils;
```

```
14.  import org.quartz.impl.StdSchedulerFactory;
15.
16.  public class Listing_7_11 {
17.       static Log logger = LogFactory.getLog(Listing_7_11.class);
18.
19.       public static void main(String[] args) {
20.            Listing_7_11 example = new Listing_7_11();
21.            try {
22.                 example.startScheduler();
23.            } catch (SchedulerException ex) {
24.                 logger.error(ex);
25.            }
26.       }
27.
28.       public void startScheduler() throws SchedulerException {
29.
30.            // Create an instance of the factory
31.            Scheduler scheduler = null;
32.
33.            // Create the scheduler and JobDetail
34.            scheduler = StdSchedulerFactory.getDefaultScheduler();
35.
36.            // Create and register the scheduler listener
37.            SchedulerListener schedulerListener =
38.                 new SimpleSchedulerListener();
39.
40.            scheduler.addSchedulerListener(schedulerListener);
41.
42.            // Start the scheduler
43.            scheduler.start();
44.            logger.info("Scheduler was started at " + new Date());
45.
46.            // Create the JobDetail
47.            JobDetail jobDetail = new JobDetail("PrintInfoJob",
48.                      Scheduler.DEFAULT_GROUP, PrintInfoJob.class);
49.
50.            /*
51.             * Set up a trigger to start firing now, with no end
52.             * date/time, repeat forever and have 5 secs
53.             * between each firing.
54.             */
55.            Trigger trigger = TriggerUtils.makeSecondlyTrigger(5);
56.            trigger.setName("SimpleTrigger");
57.            trigger.setStartTime(new Date());
58.
59.            // Register the JobDetail and Trigger
60.            scheduler.scheduleJob(jobDetail, trigger);
61.       }
62.  }
```

相比于前面的例子，我们在代码 7.11 中作了些小许改动，来实际促使更多的 SchedulerListener 方法被调用。在代码 7.11 中，Scheduler 创建后是在 Job 注册之前被启动的。这就使得在 Job 部署时 jobScheduled() 方法能得到调用。我们也改变了 Trigger 只重复两次而不是无限的运行。这样能强制 triggerFinalized() 方法被调用，因为这个 Trigger 不再有机会触发了。除了这些人为的条件外，使用 SchedulerListener 就和使用 Job 或 Trigger 监听器是一样的了。

第七章. 实现 Quartz 监听器（第五部分）

六. 使用 FileScanListener

Quartz 框架还包含一个我们未曾提及的监听器。这个监听器不像别的，因为它是为特定目的而设计的：同框架所带的一个工具 Job 一起用的。

这个监听器就是 org.quartz.jobs.FileScanListener 接口，它显式的设计为 FileScanJob 所用的，这一 Job 也在 org.quartz.jobs 包中。FileScanJob 检查某一指定文件的 lastModifiedDate。当某人改变了这个文件，这个 Job 就调用 FileScanListener 的 fileUpdated() 方法。

就像使用其他类型的 Quartz 监听器一样，你必须创建一个实现了 FileScanListener 接口的具体类。只有一个方法需要实现：

```
public void fileUpdated(String fileName);
```

代码 7.12 展了了我们的一个极简单的 FileScanListener 实现。

代码 7.12. 一个简单的 FileScanListener 实现

```
1.  package org.cavaness.quartzbook.chapter7;
2.
3.  import java.io.File;
4.  import java.sql.Timestamp;
5.
6.  import org.apache.commons.logging.Log;
7.  import org.apache.commons.logging.LogFactory;
8.
9.  public class SimpleFileScanListener implements org.quartz.jobs.FileScanListener {
10.      private static Log logger = LogFactory.getLog(SimpleFileScanListener.class);
11.
12.      public void fileUpdated(String fileName) {
13.          File file = new File(fileName);
14.          Timestamp modified = new Timestamp(file.lastModified());
15.
16.          logger.info( fileName + " was changed at " + modified );
17.      }
18.  }
```

显然，你会想做些更有意义的事情，而不仅仅是写下一条日志信息，但是你从代码 7.12 中的简单例子中明白了要旨。我们也必须在部署 FileScanJob 时为它使用这一新型监听器。代码 7.13 展示了如何部署 FileScanJob。

代码 7.13. 部署 FileScanJob

```
1.  package org.cavaness.quartzbook.chapter7;
2.
3.  import java.util.Date;
4.
5.  import org.apache.commons.logging.Log;
6.  import org.apache.commons.logging.LogFactory;
7.  import org.quartz.JobDataMap;
8.  import org.quartz.JobDetail;
9.  import org.quartz.Scheduler;
10. import org.quartz.SchedulerException;
11. import org.quartz.Trigger;
12. import org.quartz.TriggerUtils;
13. import org.quartz.impl.StdSchedulerFactory;
```

```
14. import org.quartz.jobs.FileScanJob;
15.
16. public class Listing_7_13 {
17.      private static Log logger = LogFactory.getLog(Listing_7_13.class);
18.
19.      public static void main(String[] args) {
20.           Listing_7_13 example = new Listing_7_13();
21.
22.           try {
23.                Scheduler scheduler = example.createScheduler();
24.                example.scheduleJob(scheduler);
25.                scheduler.start();
26.
27.           } catch (SchedulerException ex) {
28.                logger.error(ex);
29.           }
30.      }
31.
32.      protected Scheduler createScheduler() throws
33.           SchedulerException {
34.
35.           return StdSchedulerFactory.getDefaultScheduler();
36.      }
37.
38.      protected void scheduleJob(Scheduler scheduler) throws
39.           SchedulerException {
40.
41.           // Store the FileScanListener instance
42.           scheduler.getContext().put("SimpleFileScanListener",
43.                     new SimpleFileScanListener());
44.
45.           // Create a JobDetail for the FileScanJob
46.           JobDetail jobDetail = new JobDetail("FileScanJob", null,
47.                     FileScanJob.class);
48.
49.           // The FileScanJob needs some parameters
50.           JobDataMap jobDataMap = new JobDataMap();
51.           jobDataMap.put(FileScanJob.FILE_NAME,
52.                     "C:\\quartz-book\\input1\\test.txt");
53.           jobDataMap.put(FileScanJob.FILE_SCAN_LISTENER_NAME,
54.                     "SimpleFileScanListener");
55.           jobDetail.setJobDataMap(jobDataMap);
56.
57.           // Create a Trigger and register the Job
58.           Trigger trigger = TriggerUtils.makeSecondlyTrigger(30);
59.           trigger.setName("SimpleTrigger");
60.           trigger.setStartTime(new Date());
61.
62.           scheduler.scheduleJob(jobDetail, trigger);
63.      }
64. }
```

代码 7.13 中的程序像几乎所有别的 必须部署一个 Job 的 Quartz 应用。FileScanJob 需要两个参数：要监视文件的 FILE_NAME，和 FileScanListener(FILE_SCAN_LISTENER_NAME) 的名称。这两个参数的值会存在 JobDataMap 中，因此 FileScanJob 能访问到它们。

仅有一个特别要注意，容易出错的地方就是要确保添加了一个 FileScanListener 到 SchedulerContext 中。这在代码 7.13 中是通过如下代码片断完成的：

```
scheduler.getContext().put("SimpleFileScanListener",
```

```
    new SimpleFileScanListener());
```

这一步是必须的，因为 FileScanJob 获取到 SchedulerContext 的引用，然后使用设置到 JobDataMap 中的名称找寻 FileScanListener。

```
jobDataMap.put(FileScanJob.FILE_SCAN_LISTENER_NAME,
    "SimpleFileScanListener");
```

如果你还有所困惑，别担心：看一看 org.quartz.jobs.FileScanJob 类的源代码吧。这是对待开源软件最好的方式了。

第七章. 实现 Quartz 监听器（第六部分）

七. 在 quartz_jobs.xml 文件中实现监听器

本章的所有例子告诉了你如何以编程的方式设置监听器。假如我们一个关于在 quartz_jobs.xml 文件中以声明式配置监听器的例子都不提供本章就不能算是完结。

自 Quartz 1.5 开始，你能够在 Job 定义文件中指定监听器，当然就是知名的 quartz_jobs.xml 文件了。代码 7.14 显示了一个使用全局监听器的例子。

代码 7.14. Quartz 监听器能在 quartz_jobs.xml 文件中实现

```
1.  <?xml version='1.0' encoding='utf-8'?>
2.
3.  <quartz>
4.   <job-listener
5.     class-name="org.cavaness.quartzbook.chapter7.SimpleJobListener"
6.     name="SimpleJobListener">
7.  </job-listener>
8.
9.  <job>
10.   <job-detail>
11.    <name>PrintInfoJob</name>
12.    <group>DEFAULT</group>
13.    <job-listener-ref>SimpleJobListener</job-listener-ref>
14.    <job-class>
15.      org.cavaness.quartzbook.common.PrintInfoJob
16.    </job-class>
17.  </job-detail>
18.
19.  <trigger>
20.    <simple>
21.      <name>printJobTrigger</name>
22.      <group>DEFAULT</group>
23.      <job-name> PrintInfoJob</job-name>
24.      <job-group>DEFAULT</job-group>
25.      <start-time>2005-09-13 6:10:00 PM</start-time>
26.       <! repeat indefinitely every 10 seconds >
27.      <repeat-count>-1</repeat-count>
28.      <repeat-interval>10000</repeat-interval>
29.    </simple>
30.  </trigger>
31. </job>
32. </quartz>
```

在代码 7.14 中你看到那个附加的 <job-listener> 元素，它有两个必须的属性：

```
<job-listener
  class-name="org.cavaness.quartzbook.chapter7.SimpleJobListener"
  name="SimpleJobListener">
```

class-name 属性标识了监听器类的全限名称。name 属性指派给这个监听器一个逻辑名，在 <job-detail> 元素中用到。

下一步就是为在同一个文件中的每一个要用到监听器的 <job-detail> 元素中定义一个 <job-listener-ref> 元素。该元素的值必须与文件中所定义的其中一个 <job-listener> 元素的 name 属性相匹配。

做完那之后，要确保你已经在 文件中通过设置属性让 使用了 。

做完那之后，要确保你已经在 quartz.properties 文件中通过设置属性让 Scheduler 使用了 JobInitializationPlugin。Quartz 插件会在下一章中详细讨论。目前，只要加入以下行到你的 quartz.properties 文件中：

```
org.quartz.plugin.jobInitializer.class = org.quartz.plugins.xml.JobInitializationPlugin

org.quartz.plugin.jobInitializer.overWriteExistingJobs = true
org.quartz.plugin.jobInitializer.failOnFileNotFound = true
org.quartz.plugin.jobInitializer.validating=false
```

然后命名你的 XML 文件为 quartz_jobs.xml 并放到你的 classpath 下。

> **一些需注意的易出错的地方**
>
> 很有必要提醒你在尝试 XML 文件中设置监听器时很可能会遇到的两个问题。在 Quartz 1.5 中，至少，监听器的的 setName() 方法未包含中接口中。getName() 方法是有的，但不存在相应的 setName()。这在以编程式使用监听器时似乎不会导致什么问题，但是对于声明式时就有问题了。你需要简单的为你的监听器创建一个 setName() 方法。
>
> 另一个提示就是要确保你的监听器有一个无参构造方法。在某些情况下，Quartz 框架不会在乎，但当以声明式使用它时，你将会得到一个错误。最好还是声明一个无参的构造方法，这样总是安全的。

[译者 Unmi 后记] 配置文件中写成 <start-time>2005-09-13 6:10:00 PM</start-time>，让 Quartz 来解析这个时间字符串可能会出现异常，这在我的一个回复(http://www.blogjava.net/Unmi/archive/2007/11/17/159830.html#210032) 中有论及，现摘录如下：

quart_jobx.xml 中时间格式的问题，写成 2008-06-20 7:23:00 PM 的话 JobSchedulingDataProcessor.parseDate(value) 没办法解析

在 quarts_jobs.xml 中 <start-time> 的格式是：

```
<start-time>2008-06-23T21:23:00</start-time>
```

T隔开日期和时间，默认时区

或者：

```
<start-time>2008-06-23T21:23:00+08:00</start-time>
```

+08:00 表示东八区

我觉得这是 Quartz 的一个 Bug，其实 Quartz 在解析时间时准备了两个 Pattern 的，分别是：

```
yyyy-MM-dd'T'hh:mm:ss
yyyy-MM-dd hh:mm:ss a
```

但是在 JobSchedulingDataProcessor.parseDate(value) 方法中只会以第一个 Pattern 解析时间，并不会尝试使用第二个 Pattern 去解析时间，第二个 Pattern 是可以认识 2008-06-20 7:23:00 PM 的。

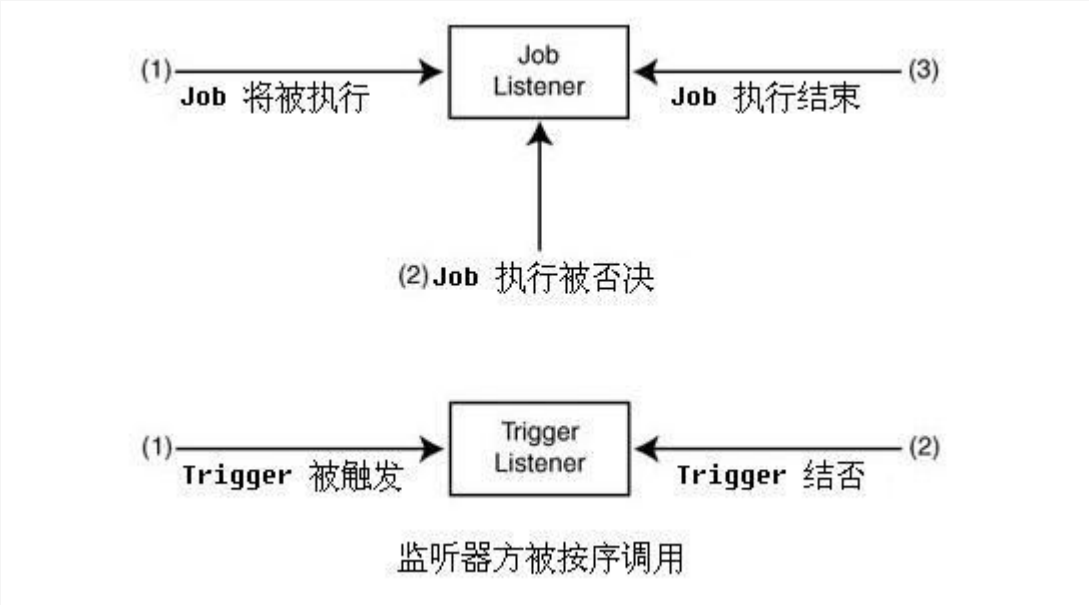
所以为了规避这个问题，还是应该写成 yyyy-MM-dd'T'hh:mm:ss 格式。

第七章. 实现 Quartz 监听器（第七部分）

八. 监听器中的线程使用

你看到了监听器接口中的方法后，你或许想知道是线程在调用监听器方法中饰演着什么样的角色。基实监听器方法是存在一个时序的，正如你看到方法名能想像到的那样。在一个 Job 执行的生命周期中，调用监听器方法以的顺序通常是固定的。图 7.2 描绘了监听方法的调用顺序和所涉及到的工作者线程。

图 7.2. 监听器方法按某一特定的时序被调用



调用监听器方法的时序是固定的。如图 7.2 所示，在 Job 的执行前后，调用 Job 的 execute() 方法相同的线程被用于调用 JobListener 和 TriggerListener 的方法。假如你使用任何类类型的第三方线程管理工具或者打算实现你自己的线程池管理，知道这一点是很重要的。假如你在监听方法中实现了一个长运行逻辑时，这也会带来对性能上的负面影响。因为调用监听方法的线程和执行 Job 是同一个工作者线程，你不应该把监听方法实现的太复杂并要花费较长时间才能完成。保持它们的执行时间尽可能短。

九. Quartz 监听器的使用

了解了这所有的知识之后，那你能拿这些监听器做什么呢？实际上，你可以做相当多的事情。首先，值得注意的是，在内部，Quartz 使用这些监听器来帮助管理 Scheduler 和你的 Job 和 Trigger。框架还包含两个实现了监听器接口的插件，它们记录所有 Job 的日志和触发历史：分别为 org.quartz.plugins.history.LoggingJobHistoryPlugin 和 org.quartz.plugins.history.LogginTriggerHistoryPlugin。我们在下一章讲到 Quartz 插件。

这里有一些监听的使用：

- 捕获错过触发和重新的部署
- 成功执行完一个 Job 后发送一个 e-mail
- 基于数据库中设置的标记否决 Job 的执行
- 基于一个 Job 执行的成功或失败部署其他的 Job
- 记录一个 Job 的实际运行时间

这些仅是一些想法。当你的 Quartz 应用运行期间出现特定的事件时，Quartz 监听器为你提供了一种方法接收到编程角度上的通知。你选择什么来应用那些知道，如果有的话，完全由你而定。

第八章. 使用 Quartz 插件（第一部分）

第八章. 使用 Quartz 插件

Quartz 框架提供了几种用于扩展平台能力的方式。通过使用各种 "钩子" (通常指的就是扩展点)，Quartz 变得很容易被扩展和定制化来适应你的需要。其中一个最简单的扩展框架的方法就是使用 Quartz 插件。本章就来看看如何使用插件机制让 Quartz 进入到之前 Quartz 用户没去过的领域。

一. 什么是插件？

假如你使用过其他的开源框架，例如 Apache Struts，你应该已经熟悉了插件的概念和它们的用法。非常简单，一个 Quartz 插件就是一个实现了 org.quartz.spi.SchedulerPlugin 接口的 Java 类，并且被作为插件注册给了 Scheduler。这个插件接口包含了三个方法，显示在代码 8.1 中。

代码 8.1. Quartz 插件必须实现的 SchedulerPlugin 接口

```
1.  public interface SchedulerPlugin {
2.
3.      public void initialize(String name, Scheduler scheduler)
4.                  throws SchedulerException;
5.
6.      public void start();
7.      public void shutdown();
8.
9.  }
```

SchedulerPlugin 的方法是在 Scheduler 的初始化和启动期间被调用。Scheduler 会调用每一个已注册插件的这三个方法。下面的几节描述了插件的每一个方法会在什么时候被调用。

·initialize() 方法

initialize() 方法在 Scheduler 的创建期间被调用。当 StdSchedulerFactory 的 getScheduler() 方法被调用后，这个工厂就调用所有注册的插件的 initialize() 方法。

> **插件不能同 DirectSchedulerFactory 正常工作**
>
> 插件只是设计来用与 StdSchedulerFactory 的。这是在框架中的限制。如果你想要用插件，那么就需要使用 StdSchedulerFactory 来获取你的 Scheduler 实例。

每个插注册为一个唯一的名字。这个给定的名字和 Scheduler 实例被包含在对 initialize() 方法的调用中。你需要在你的插件初始化的时候做些事情。如，你的插件也许需要从文件或数据库中读取并解析数据。

> **Scheduler 在 initialize() 时并未完全创建好**
>
> 当这个方法在调用的时候，Scheduler 还未完全初始化好，因此它和 Scheduler 的交互应保持在尽量短的时间。例如，你不应该试图在 initialize() 方法中去部署任何 Job。

假如在初始化插件的时候发生了问题，你应当抛出一个 org.quartz.SchedulerConfigException 异常，这个异常是 SchedulerException 的子类。这能阻止插件的继续加载，并停止了它与 Scheduler 后面的交互。

·start() 方法

Scheduler 实例调用 start() 方法让插件知道它可以执行任何需要的启动动作了。例如，假如你有 Job 要部署，那这个时候就可以部署他们了。

·shutdown() 方法

shutdown() 方法被调用来通知插件 Scheduler 将要关闭了。这是给插件的一个机会去清理任何打开的资源。例如，数据库连接或是打开的文件应该要关闭。

Scheduler 实例不会传递给 start() 或 shutdown()

注意到，Scheduler 实例并不会作为参数传递给 start() 或 shutdown() 方法。如果你的插件需要在 start() 或 shutdown() 中访问 Scheduler，你必须把 Scheduler 存放在插件的一个实例变量中。

第八章. 使用 Quartz 插件（第二部分）

二. 创建 Quartz 插件

创建一个新的插件很简单。你所有要做的就是创建一个 Java 类(或重用一个现有的类)，让它实现 org.quartz.spi.SchedulerPlugin 接口。Scheduler 将会在启动期间创建这个插件的实例。这个插必须有一个无参的构造方法，很显然它不能是抽象的。

·JobInitializationPlugin

Quartz 框架有一个用来从 XML 文件中加载 Job 和 Trigger 信息的插件。这个插件就是 org.quartz.plugins.xml.JobInitializationPlugin，并且它在前面第三章 "Hello, Quartz" 中简略的讨论过。当你使用这个插件的时候，Quartz 框架就会搜寻一个叫做 quartz_jobs.xml 的文件并试图从中加载 Job 和 Trigger 信息。

> **改变 JobInitializtionPlugin 加载的 XML 文件**
>
> 插件允许你改变它要查找来加载 Job 和 Trigger 信息的文件的名字。你可以通过在 quartz.properties 文件中设置一个别的文件名。我们会在本章后续中讲到更多的关于设置插件参数的内容。

如第三章所解释的，这个插件在你的应用需求不涉及到从数据库中加载 Job 信息时是很方便的。它在开发和测试期间也是很有用的，因为你可以快速的配置哪些 Job 和 Trigger 要被触发。就是说，无可争辩的，修改一个 XML 总比一系列的数据库表要简单。

对于从一个 XML 文件中加载 Job 和 Trigger 信息做法的一个很好的延展就是可以有一个目录来存储 Job XML 文件，然后过使用一个插件，Scheduler 就会加载任何存在的 Job 文件了。这允许你在 Scheduler 启动时简单从指定的目录中添加或移除 Job 文件来方便的增加或删除 Job。在本章剩下的部分，我们向你展示如何构建这个插件。

·创建 JobLoaderPlugin

我们把这个新插件命名为 JobLoaderPlugin。代码 8.2 中显示了这个 JobLoaderPlugin 类。

代码 8.2. 从一个目录中加载多个 Job 文件的 Quartz SchedulerPlugin

```
1.  package org.cavaness.quartzbook.chapter8;
2.
3.  import java.io.File;
4.
5.  import org.apache.commons.logging.Log;
6.  import org.apache.commons.logging.LogFactory;
7.  import org.quartz.Scheduler;
8.  import org.quartz.SchedulerConfigException;
9.  import org.quartz.SchedulerException;
10. import org.quartz.spi.SchedulerPlugin;
11. import org.quartz.xml.JobSchedulingDataProcessor;
12.
13. public class JobLoaderPlugin implements SchedulerPlugin {
14.
15.      private static Log logger =
16.           LogFactory.getLog(JobLoaderPlugin.class);
17.
18.      // The directory to load jobs from
19.      private String jobsDirectory;
20.
21.      // An array of File objects
22.      private File[] jobFiles = null;
23.
```

```
24.      private String pluginName;
25.
26.      private Scheduler scheduler;
27.
28.      private boolean validateXML = true;
29.
30.      private boolean validateSchema = true;
31.
32.      public JobLoaderPlugin() {
33.      }
34.
35.      public File[] getJobFiles() {
36.           return jobFiles;
37.      }
38.
39.      public void setJobFiles(File[] jobFiles) {
40.           this.jobFiles = jobFiles;
41.      }
42.
43.      public boolean isValidateSchema() {
44.           return validateSchema;
45.      }
46.
47.      public void setValidateSchema(boolean validatingSchema) {
48.           this.validateSchema = validatingSchema;
49.      }
50.
51.      public void initialize(String name, final Scheduler scheduler)
52.                throws SchedulerException {
53.
54.           this.pluginName = name;
55.           this.scheduler = scheduler;
56.
57.           logger.debug("Registering Plugin " + pluginName);
58.           // Load the job definitions from the specified directory
59.       loadJobs();
60.  }
61.  private void loadJobs() throws SchedulerException {
62.
63.      File dir = null;
64.
65.      // Check directory
66.      if (getJobsDirectory() == null
67.                || !(dir =
68.                new File(getJobsDirectory())).exists()) {
69.           throw new SchedulerConfigException(
70.                     "The jobs directory was missing "
71.                               + jobsDirectory);
72.
73.      }
74.      logger.info("Loading jobs from " + dir.getName());
75.
76.      // Only XML files, filtering out any directories
77.      this.jobFiles = dir.listFiles(new XMLFileOnlyFilter());
78.  }
79.
80.  public void start() {
81.      processJobs();
82.  }
83.
84.  public void shutdown() {
```

```
85.       // nothing to clean up
86.   }
87.
88.   public void processJobs() {
89.       // There should be at least one job
90.       if (getJobFiles() == null || getJobFiles().length == 0) {
91.           return;
92.       }
93.
94.       JobSchedulingDataProcessor processor =
95.           new JobSchedulingDataProcessor(
96.               true, isValidateXML(), isValidateSchema());
97.
98.       int size = getJobFiles().length;
99.       for (int i = 0; i < size; i++) {
100.          File jobFile = getJobFiles()[i];
101.
102.          String fileName = jobFile.getAbsolutePath();
103.          logger.debug("Loading job file: " + fileName);
104.
105.          try {
106.
107.              processor.processFileAndScheduleJobs(
108.                  fileName, scheduler, true);
109.
110.          } catch (Exception ex) {
111.              logger.error("Error loading jobs: " + fileName);
112.              logger.error(ex);
113.          }
114.       }
115.   }
116.
117.   public String getJobsDirectory() {
118.       return jobsDirectory;
119.   }
120.
121.   public void setJobsDirectory(String jobsDirectory) {
122.       this.jobsDirectory = jobsDirectory;
123.   }
124.
125.   public String getPluginName() {
126.       return pluginName;
127.   }
128.
129.   public void setPluginName(String pluginName) {
130.       this.pluginName = pluginName;
131.   }
132.
133.   public boolean isValidateXML() {
134.       return validateXML;
135.   }
136.
137.   public void setValidateXML(boolean validateXML) {
138.       this.validateXML = validateXML;
139.   }
140.  }
```

代码 8.2 中 JobLoaderPlugin 的实际工作仅是由两个方法：initialize() 和 start() 来完成的。它们是 SchedulerPlugin 接口所必须的。其他的方法只是 setXXX 和 getXXX 方法是用于实现 JavaBean 规范的，因为声明了私有属性。

·JobLoaderPlugin initialize() 方法

正如你看到的，由 Scheduler 调用的 initialize() 方法会调用 loadJobs() 方法。loadJobs() 方法使用从 quartz.properties 文件传入的 jobsDirectory 所指示的目录中获取所有的 XML 文件。这个插件还不会试图部署 Job，因为在插件的 initialize() 方法被调用的时候 Scheduler 还没有完全初始化好。JobLoaderPlugin 只简单的持有一个 File 对象的数组，直等 start() 方法被调用。我们还持有了一个 Scheduler 实例，以便我们在 start() 方法在调用时能访问它。

·JobLoaderPlugin start() 方法

当 Scheduler 调用 JobLoaderPlugin 的 start() 方法时，start() 方法就调用 processJobs()。processJobs() 方法遍历那个 Job 文件的数组并把每个都加载到 Scheduler 实例中。

对 Job 文件的处理是通过一个 org.quartz.xml.JobSchedulingDataProcessor 实例来完成的。调用 processFileAndScheduleJobs() 方法并传入文件名，Scheduler 实例和一个布尔值告诉它是否覆盖已有的 Job。

当 processJobs() 方法完成后，这个指定 jobsDirectory 下的所有 Job 文件就已经被加载并被部署了。

第八章. 使用 Quartz 插件（第三部分）

三. 注册你的插件

当 SchedulerFactory 首次初始化的时候，会从 quartz.properties 文件中搜寻你所配置的 Quartz 插件。它会通过 java.lang.Class 的 newInstance() 方法创建插件的实例。你的插件必须有一个无参的构造方法，像代码中 JobLoaderPlugin 所做的那样。

要在 quartz.properties 文件中注册你的插件的话，需在 quartz.properties 文件中使用如下的格式创建一个属性：

org.quartz.plugin.<pluginName>.class=<fully_qualified_class_name_of_plugin>

Quartz 找寻属性文件中所有含这个关键词的项：

org.quartz.plugin.<pluginName>.class

接着会试图创建等号右边的类的实例，并假定它是一个插件。你通过为 <pluginName> 字段提供一个唯一的名字来命你这个插件。

代码 8.3 展示了一个使用 JobLoaderPlugin 的 quartz.properties 文件。

代码 8.3. 在 quartz.properties 文件中注册 JobLoaderPlugin

```
#============================================================================
# Configure Main Scheduler Properties
#============================================================================

org.quartz.scheduler.instanceName = QuartzScheduler
org.quartz.scheduler.instanceId = AUTO

#============================================================================
# Configure ThreadPool
#============================================================================

org.quartz.threadPool.class = org.quartz.simpl.SimpleThreadPool
org.quartz.threadPool.threadCount =  5
org.quartz.threadPool.threadPriority = 5

#============================================================================
# Configure JobStore
#============================================================================

org.quartz.jobStore.misfireThreshold = 60000
org.quartz.jobStore.class = org.quartz.simpl.RAMJobStore

#============================================================================
# Configure Plugins
#============================================================================
org.quartz.plugin.jobLoader.class = org.cavaness.quartzbook.chapter8.JobLoaderPlugin

org.quartz.plugin.jobLoader.jobsDirectory = c:\\quartz-book\\sample\\chapter8
```

你应该已经看过了在代码 8.3 的 quartz.properties 文中的大部分设置。最后一台就是用来注册 JobLoaderPlugin 的。

·在 quartz.properties 中指定插件

在初始化和启动期间，Quartz Scheduler 从 quartz.properties 文件中加载属性。你必须在 quartz.properties 文件中指定插

件的时候遵循某一特定的格式。这一格式显示如下：

<plugin prefix>.<pluginName><.class>=<fully qualified Plugin class name>

·<plugin prefix> 总是 org.quartz.plugin.

·<pluginName> 是你指派的唯一的名字

·要指定插件类，使用后缀 .class

·右边是插件类的全限名

回头看代码 8.3，你会发现我们的 JobLoaderPlugin 是用了这一格式：

org.quartz.plugin.jobLoader.class=org.cavaness.quartzbook.chapter8.JobLoaderPlugin

指定给这个插件的名字是 jobLoader，可以任意的。我们可以使用任意的名字，只要它区别于其他已注册的插件是唯一的就行。在等号的右边，你必须指定插件类的全限名。这个类必须在 classpath 中或是对于类加载器可见。

·向插件传递参数

多数插件需要配置值，并以此作为好的编程实践，我们不希望把这些值硬编码到插件类中。Quartz 提供了一种机制向你的插件类传递参数，就是在 quartz.properties 文件中提供参数值。

Scheduler 找寻所有的能匹配如下格式的其他属性：

<plugin prefix>.<plugin name>.<property name>=<someValue>

它把把们视为插件类的 JavaBean 属性。从代码 8.3 中来看，意味着这一属性引发对 setJobDirectory() 方法的调用，并传递字符串值 c:\\quartz-book\\sample\\chapter8 作为该方法的参数：

org.quartz.plugin.jobLoader.jobsDirectory=c:\\quartz-book\\sample\\chapter8

你可以有配置给你的插件所需的更多的属性

> **插件属性必须 set() 方法**
>
> 你必须为你打算传递给插类的每一个属性提供 setXXX() 方法。Quartz 在找不到属生对应的公有 setXXX() 方法时抛出 SchedulerException 异常并终止 Scheduler。基于 JavaBean 规范，你应该为属性提供 get() 和 set() 方法。

Quartz 框架转换属性值成插件指定的类型，假定是原始类型，例如，你能指定属性的类型为 int，并期望 Quartz 把 quartz.properties 文件中的字符串转换成一个 int。然而框架不会把 1 转换成一个 Integer 类。

> **Quartz 使用内省来设值**
>
> Quartz 使用内省和反射来把 quartz.properties 文件中的参数值转换成插件类中的正确类型。你大约已经猜到它使用了 Jakarata 的 common BeanUtils，但是还不仅如此。

·为 JobLoaderPlugin 创建 Job 文件

JobLoaderPlugin 查找指定目录下的所有 XML 文件，并假定每个文件都是有铲的 Quartz Job 文件。我们说 "有效"，意思是说 XML 文件是符合最新的 job-scheduling XSD 文件的，写这个的时候是 job_scheduling_data_1_5.xsd。

为使得 JobLoaderPlugin 更有用，我们把每一个 Job，和伴随着它的 JobDetail 和 Trigger 信息放在一个独立的 XML 文件中。这让我们在添加或完全移除 Job 时仅仅是把文件放入到这个目录中或是从中提出来。这对于你在开发环境中只是想测试某些 Job 的时候很有帮助。一个单一个 Job XML 文件如代码 8.4 所示。

代码 8.4. 一个由 JobLoaderPlugin 读取的 Job XML 文件

```
<?xml version='1.0' encoding='utf-8'?>


<quartz xmlns="http://www.opensymphony.com/quartz/JobSchedulingData"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.opensymphony.com/quartz/JobSchedulingData
 http://www.opensymphony.com/quartz/xml/job_scheduling_data_1_5.xsd"
 version="1.5">

 <job>
   <job-detail>
     <name>PrintInfoJob1</name>
     <group>DEFAULT</group>
     <job-class>
       org.cavaness.quartzbook.chapter3.ScanDirectoryJob
     </job-class>
     <volatility>false</volatility>
     <durability>false</durability>
     <recover>false</recover>

     <job-data-map allows-transient-data="true">
       <entry>
         <key>SCAN_DIR</key>
         <value>c:\quartz-book\input1</value>
       </entry>
     </job-data-map>
   </job-detail>

   <trigger>
     <simple>
       <name>trigger1</name>
       <group>DEFAULT</group>
       <job-name>PrintInfoJob1</job-name>
       <job-group>DEFAULT</job-group>
       <start-time>2005-07-30T16:04:00</start-time>

       <!-- repeat indefinitely every 10 seconds -->
       <repeat-count>-1</repeat-count>
       <repeat-interval>10000</repeat-interval>
     </simple>
   </trigger>
 </job>


</quartz>
```

像代码 8.4 的 Job 文件中包含所有 JobLoaderPlugin 用来部署 Job 所需的信息。这个文件也包含一个 JobDataMap 项，这对于 Job 类运行时是可用的。代码 8.4 中的例子使用一个已配置的 SimpleTrigger 来部署一个以10秒为间隔不断重复的 Trigger。为进一步测试这个插件，我们创建了第二个 Job 文件，与前面的相比只有些许差别。代码 8.5 显示了第二个 Job 文件。

代码 8.5. 由 JobLoaderPlugin 加载的第二个 Job XML 文件

```
<?xml version='1.0' encoding='utf-8'?>

<quartz xmlns="http://www.opensymphony.com/quartz/JobSchedulingData"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
  xsi:schemaLocation="http://www.opensymphony.com/quartz/JobSchedulingData
  http://www.opensymphony.com/quartz/xml/job_scheduling_data_1_5.xsd"
  version="1.5">

  <job>
    <job-detail>
      <name>PrintInfoJob2</name>
      <group>DEFAULT</group>
      <job-class>
        org.cavaness.quartzbook.chapter3.ScanDirectoryJob</job-class>
      <volatility>false</volatility>
      <durability>false</durability>
      <recover>false</recover>

      <job-data-map allows-transient-data="true">
         <entry>
          <key>SCAN_DIR</key>
           <value>c:\quartz-book\input2</value>
         </entry>
      </job-data-map>

    </job-detail>

     <trigger>
      <simple>
        <name>trigger2</name>
        <group>DEFAULT</group>
        <job-name>PrintInfoJob2</job-name>
        <job-group>DEFAULT</job-group>
        <start-time>2005-07-30T16:04:00</start-time>
        <!-- repeat indefinitely every 10 seconds -->
        <repeat-count>-1</repeat-count>
        <repeat-interval>60000</repeat-interval>
      </simple>
    </trigger>
  </job>

</quartz>
```

代码 8.5 所示的第二个 Job 文件与代码 8.4 的第一次相比只有稍稍不同。我们改变了 Job 所用来扫描的目录和触发计划。这里的重点就是在 Job 目录中你可以有多个 Job，而且 JobLoaderPlugin 将会加载它们并分别部署到 Scheduler 上。

第八章. 使用 Quartz 插件（第四部分）

四. 使用多个插件

你喜欢多少个，就可以在 quartz.properties 文件中注册多少个插件。然而，加载和初始化的顺序却不能保证，因为 Quartz 加载先把所有的属性到一个 Map 中，然后按照从 Map 中取出的顺序遍历插件。

为规避这一限制，你可以创建一个 Quartz 插件作为父插件，然后以给定的顺序加载其他多个插件。代码 8.6 显示了 ParentPlugin 长什么样子。

代码 8.6. ParentPlugin 能以特定的顺序加载子插件

```
1.  package org.cavaness.quartzbook.chapter8;
2.
3.  import java.util.ArrayList;
4.  import java.util.List;
5.  import java.util.StringTokenizer;
6.
7.  import org.apache.commons.logging.Log;
8.  import org.apache.commons.logging.LogFactory;
9.  import org.quartz.Scheduler;
10. import org.quartz.SchedulerConfigException;
11. import org.quartz.SchedulerException;
12. import org.quartz.spi.SchedulerPlugin;
13.
14. public class ParentPlugin implements SchedulerPlugin {
15.     private static Log logger = LogFactory.getLog(ParentPlugin.class);
16.
17.     // A list of child plug-ins
18.     private List childPlugins = new ArrayList();
19.
20.     private String childPluginNames;
21.
22.     private String pluginName;
23.
24.     private Scheduler scheduler;
25.
26.     /**
27.      * Default no-arg Constructor
28.      *
29.      */
30.     public ParentPlugin() {
31.     }
32.
33.     /**
34.      * Pass the initialize call on to the child plug-ins.
35.      *
36.      * @throws SchedulerConfigException
37.      *             if there is an error initializing.
38.      */
39.
40. public void initialize(String name, final Scheduler scheduler)
41.               throws SchedulerException {
42.
43.          this.pluginName = name;
44.          this.scheduler = scheduler;
45.
```

```
46.          logger.info("Searching for child plugins to load");
47.
48.          // The child plug-ins are comma-separated
49.          StringTokenizer tokenizer =
50.               new StringTokenizer(childPluginNames, ",");
51.
52.          while (tokenizer.hasMoreElements()) {
53.               String pluginClassname = tokenizer.nextToken();
54.
55.               try {
56.                    Class pluginClass =
57.                         Class.forName(pluginClassname);
58.
59.                    Object obj = pluginClass.newInstance();
60.
61.                    // Make sure the specified class is a plug-in
62.                    if (obj instanceof SchedulerPlugin) {
63.                         // Initialize the Plugin
64.                         SchedulerPlugin childPlugin =
65.                               (SchedulerPlugin) obj;
66.
67.                              logger.info("Init child Plugin " +
68.                                   pluginClassname);
69.
70.                         childPlugin.initialize(pluginClassname,
71.                              scheduler);
72.
73.                         // Store the child plug-in in the list
74.                         childPlugins.add(childPlugin);
75.                    } else {
76.                         // Skip loading class
77.                         logger.error("Class is not a plugin " +
78.                              pluginClass);
79.                    }
80.
81.               } catch (Exception ex) {
82.                    // On error, log and go to next child plug-in
83.                    logger.error("Error loading plugin " +
84.                         pluginClassname, ex);
85.               }
86.          }
87.     }
88.     public void start() {
89.          // Start each child plug-in
90.          int size = childPlugins.size();
91.          for (int i = 0; i < size; i++) {
92.               SchedulerPlugin childPlugin =
93.                    ((SchedulerPlugin) childPlugins.get(i));
94.
95.               logger.info("Starting Child Plugin " + childPlugin);
96.               childPlugin.start();
97.          }
98.     }
99.
100.     public void shutdown() {
101.          // Stop each child plug-in
102.          int size = childPlugins.size();
103.          for (int i = 0; i < size; i++) {
104.               SchedulerPlugin childPlugin =
105.                    ((SchedulerPlugin) childPlugins.get(i));
106.
```

```
                logger.info("Stopping Plugin " + childPlugin);
                childPlugin.shutdown();
            }
        }

        public String getPluginName() {
            return pluginName;
        }

        public void setPluginName(String pluginName) {
            this.pluginName = pluginName;
        }

        public String getChildPluginNames() {
            return childPluginNames;
        }

        public void setChildPluginNames(String childPluginNames) {
            this.childPluginNames = childPluginNames;
        }
    }
```

代码 8.6 中的插件基什么也不做，也就只能算作一个插件，但是它扮演着子插件的加载器。一个子插件是个有效的 Quartz 插件，它可以是你所编写的或者是包含在框架中的。

·配置文件 quartz.properties 中的 ParentPlugin

要配置 quartz.properties 文件中的 ParentPlugin，仅要加入作为其他插件的父类。那就是，添加如下的行：

org.quartz.plugin.parentPlugin.class = org.cavaness.quartzbook.chapter8.ParentPlugin

接着，添加子插件，并按你想要的加载顺序排列，就是指定一个逗号分隔的插件列表：

org.quartz.plugin.parentPlugin.childPluginNames=org.quartz.plugins.history
.LoggingJobHistoryPlugin,org.quartz.plugins.history.LoggingTriggerHistoryPlugin

[译者 Unmi 注：注意，上面是写在一行里的，因页面排版本的原因写成两行了]

如代码 8.6 所示，ParentPlugin 拆开逗号分隔的字符串并以它们在列表中的顺序来加载插件。这看起来好像相当复杂，但却能很好的完成工作。将来的 Quartz 框架版本也许支持插件的按序加载机制，但是现在，ParentPlugin 工作的很好。

第八章. 使用 Quartz 插件（第五部分）

五. Quartz 工具插件

Quartz 框架包括几个你能用于你的应用中的几个插件。本节简单描述它们和它们的用途。

·JobInitializationPlugin

我们已经多次谈到过这个插件。它从一个 XML 文件中加载 Job 和 Trigger 信息(默认文件名是 quartz_jobs.xml) 。你可以通过在 quartz.properties 文件中为这个插件设定 filename 参数来配置文件名。假如你不需要数据库来存储你的 Job 或者是需要能快速测试特定 Job 的话，这个插件非常有帮助。

·JobInitializationPluginMulitple

显然，由其相似的名字，JobInitializationPluginMultiple 类似于 JobInitializationPlugin。不同点在于它支持从多个 XML 文件加载而非只是一个。它也类似于代码 8.2 中的 JobLoaderPlugin，只是代码 8.2 的插件查找一个目录，而不是一系列的文件。

指定给 JobInitializationPluginMultiple 的文件是以逗分进行分隔，配置在 quartz.properties 文件中的。这个插件的一个最好的特征就是能定期的扫描是否有修改，当它们有改变时能重新加载 Job 信息。它是通过实现 org.quartz.jobs.FileScanListener 来加入这一行为的。扫描间隔(定义为秒) 可以在属性文件中指定。

·LogginJobHistoryPlugin

org.quartz.plugins.history.LogginJobHistoryPlugin 是用来记录 Job 历史的，应用的是 commons-logging 框架。这包括 Job 的执行还有任何 Job 被否决的日志。这个插件允许配置日志消息，但已提供了默认的消息格式。格式可以指定 Job 的哪些字段会包括在消息中。你能为下列事件提供分离的消息格式：

- ·jobFailedMessage 在 Job 执行失败时记录
- ·jobSuccessMessage 在 Job 完成执行时记录
- ·jobToBeFiredMessage 在 Job 即将执行时记录
- ·jobWasVetoedMessage 在 Job 要被否决时记录

例如，当你想要覆盖掉 Job 即将时的默认消息，并且你所关心，想在日志消息中看到的是 Job 的名称和执行时间，你可以加入以下行到 quartz.properties 文件：

```
org.quartz.plugin.jobHistory.class=org.quartz.plugins. history.LoggingJobHistoryPlugin
org.quartz.plugin.jobHistory.jobToBeFiredMessage=Job {0} is about to be fired at: {2, date, HH:mm:ss MM/dd/yyyy}
```

你可以在日志消息在包含多个有关 Job 的数据元素。表 8.1 列出了这些元素和它们的数据类型。

表 8.1. 能用于 Job 的消息格式中的元素

元素	数据类型	描述
0	String	Job 的名称
1	String	Job 组的名称
2	Date	当前日期
3	String	Trigger 的名称
4	String	Trigger 组的名称
5	Date	调度的触发时间
6	Date	调度的下一触发时间
7	Integer	JobExecutionContext 的触发次数

每个事件能分开来配置在 quartz.properties 文件中。使用这个插件唯一负面是，假如你有许多的 Job，日志文件很快就被撑大，几乎就是过度的信息了。

·LoggingTriggerHistoryPlugin

这个插件相当于 Job 的历史插件，只是它用于 Trigger 的历史信息罢了。你可以为如下 Trigger 事件提供日志消息格式：

·triggerCompleteMessage 在 Trigger 完成了所有的触发再也不被触发时记录

·triggerFireMessage 在 Trigger 触发时记录

·triggerMisfiredMessage 在错过触发之后记录

就像是 LoggingJobHistoryPlugin，你可以在属性文件中覆盖掉默认的消息格式。

你可以在日志消息中包括有关 Trigger 的几个数据元素。表 8.2 列出了这些无素及数据类型。表 8.2 列出了能用于 Trigger 历史日志消息的元素。

表 8.2. 能用于 Trigger 消息格式的元素

元素	数据类型	描述
0	String	Trigger 的名称
1	String	Trigger 组的名称
2	Date	调度的触发时间
3	Date	Trigger 下一次触发时间
4	Date	Trigger 实际触发时间
5	String	Job 的名称
6	String	Job 组的名称
7	Integer	JobExecutionContext 的触发次数

像是 LoggingJobHistoryPlugin 一样，这个插件也能记录大量的 ，特别是几个 Trigger 要经常触发。

·ShutdownHookPlugin

这个插件捕获 JVM 的关闭事件并强制关闭 Scheduler。你或许会说，“我为何需要在 JVM 已经在关闭时告诉 Scheduler 关闭自己呢？” 原因主要是让 Scheduler 能执行一个“干净”的关闭。

当这个插件的 initialize() 方法被调用时，它会加入一个新的 java.lang.Thread 到 JVM 中。在 JVM 获取到关闭事件后，这由两个事件之一所导致：

·当最后一个非守护线程退出或调用了 System.exit() 方法时的程序正常退出。

·JVM 响应用户输入后被终止，如用户按下了 Ctrl+C，或是系统范围的事件，像用户注销或系统关闭。

当 JVM 获取到关闭通知后，它对关闭线程执行一次回调并给这个线程以运行的机会。对 ShutdownHookPlugin 来说，run() 方法就会调用 Scheduler，并告诉它作出关闭操作。默认时，会传递一个布尔值 true 来调用 Scheduler 的 shutdown() 方法，这就等于告诉 Scheduler 执行一个“干净”的关闭。 这意味着 Scheduler 将要等待所有正在执行着的 Trigger 结束后才停止。

你可以告诉这个插件不去执行一个“干净”的关闭，那就要在 quartz.properties 文件中把这作为插件的参数。

[译者 Unmi 注：最后一句原文写得也有些含混不清，其实就是说要调用 Scheduler 的 shutdown() 方法时传递 false 参数的话，就要在 quartz.properties 中配置 ShutdownHookPlugin 的属性 cleanShutdown 为 false]

第九章. 使用 Quartz 的远程方式（第一部分）

第九章. 使用 Quartz 的远程方式

以独立方式运行的 Quartz 应用程序，仅限于在 JVM 内部访问调度器。和其他任何 J2SE 程序一样，不使用其他某种机制的话，是决不允许从外部访问 JVM 中的对象的。

幸运的是，有几种技术(机制) 可让你做到这一点。Quartz 框架很好的支持其中一种机制－－远程方法调用(RMI)。本章就是关注于如何部署 Quartz 为一个 RMI 服务，以便于你能够从 JVM 外部访问到调度器。这样做有几个好处，这也是我们本章要讨论内容。

1. 为什么要把 RMI 和 Quartz 相结合

想像一下，你需要构建一个这样的作业调度器，它要应多个客户端请求进行动态 Job 部署。在这个案例中，一个单一的、自包容的 Quartz 调度器是没法做到的，因为这些客户端程序在自己所在的地址空间或者说 JVM 中需要以一种方式与调度器交谈。

借助于 RMI，运行在一个地址空间(或JVM) 的对象可自由的调用另一 JVM 中的对象。这也扩充了 Quartz 这一工具集，使这一框架更有利。

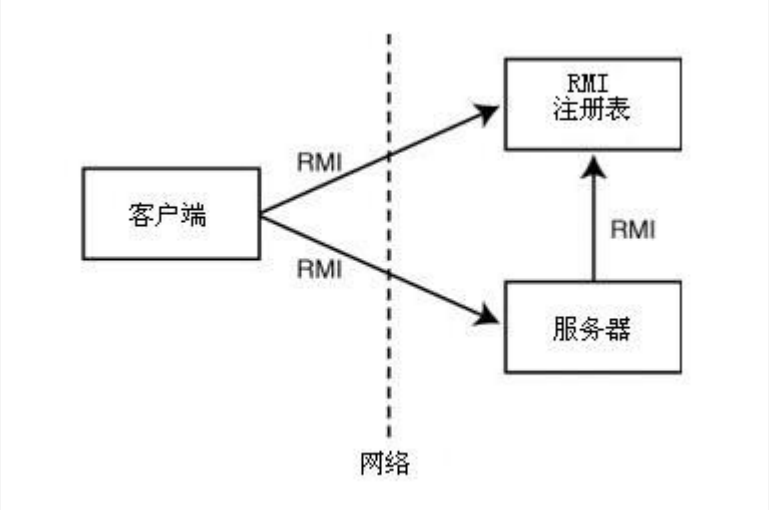
2. Java RMI 概览

假如 RMI 对你来说还是个新鲜玩艺，那么在进一步具体理解 Quartz 中使用 RMI 之前值得对它作个大概的了解。如果你完全适应了RMI，特别是能灵活运用它了，可以跳过这一节。

RMI 是一种允行在一个地址空间的对象与别一地址空间的对象进行通行的机制。这两个地址空间可能存在于同一机器上或者根本就在不同的机器上。一般而言，你可以认为 RMI 是一种面向对象的远程过程调用(RPC) 机制。

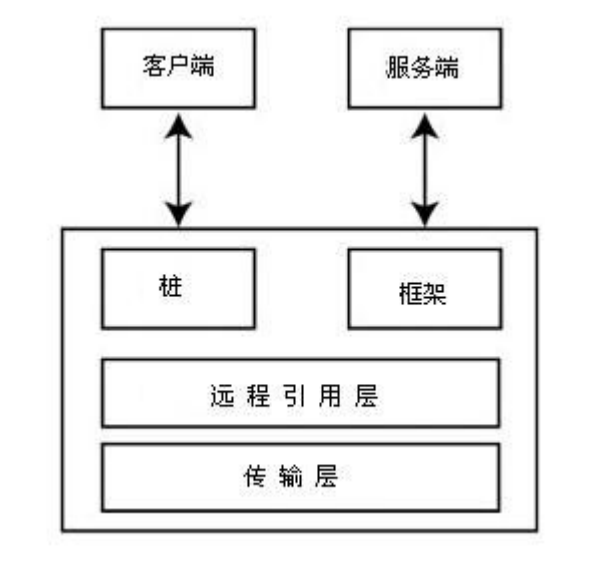
RMI 并不是特定于 Java 东西，但是 Java 有自己的实现：Java RMI，这是作为 Sun (http://java.sun.com/products/jdk/rmi) Java SDK 的一部份发布并提供支持的。Java RMI 允许运行在一个 JVM 中的对象与运行在另一 JVM 中的对象通信或调用其方法。这两个 JVM 可运行在同一机器，或者更有意义的就是让它们分布在不同的机器上。它们甚至是在不同的平台之上的。例如，一个 JVM 运行在 Windows 平台，而另一个在 Linux 上。使用 RMI 的话，这些都不在乎。图 9.1 描绘了这个场景。

图9.1. Java RMI 允许开发者创建分布式应用



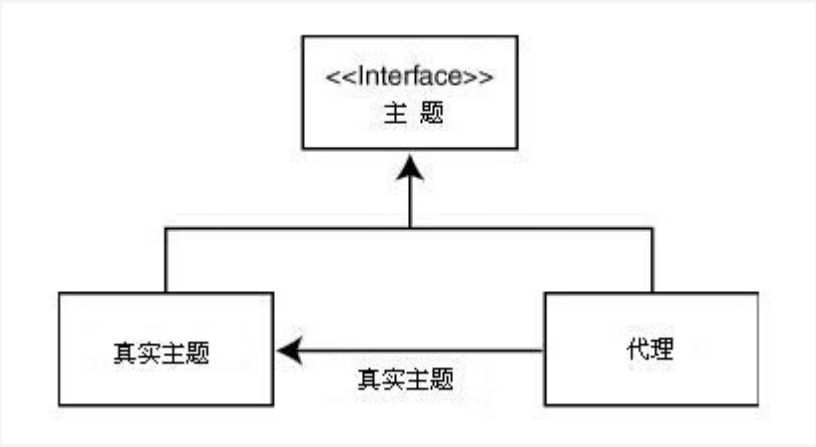
RMI 是基于职责分离原则的。接口定义与实现分离。这很好的符合了 RMI 的目标：允许一个主要关注于接口定义的客户端与实现了接口定义的服务端分离(可能是物理上的)。换句话就是，客户端有一个接口，通过接口可以调用到服务，然而服务的实现是驻留在另一台机器上的。图9.2 描绘了这种 Java 编程语言的分离实现。

图 9.2 Java 支持的服务接口定义与实现的分离



当 RMI 客户端调用一个对象上的方法时，它实际上调用的是客户端 JVM 的一个代理对象的方法。这个代理对象知道如何与服务器上的对象进行通信。所有的通信，包括数值传递给服务器对像和返回都是经由代理对象进行的。这一过程请看图 9.3。

图 9.3. 所有的从客户端到服务端的通信都要走代理对象



·Java RMI 应用的组成

每一人 Java RMI 应用都包括以下三部分

- ·RMI 服务程序
- ·RMI 客户程序
- ·RMI 对象注册表

·RMI 服务器

RMI 服务器的责任是创建服务对象(客户端要调用其上的方法) 并使之对于远程客户端来说是可见的。RMI 服务器跑在一个标准的 JVM 中。这些服务对象也是标准的 Java 对象，不仅提供远程调用，也能在本地调用。

·RMI 客户端

RMI 客户端是一个跑在(典型的) 不同于服务器的JVM 中的Java 程序；它可以是与 RMI 服务端同在一台机器，或者是分布在不同的机器上。客户端程序通过从 RMI 注册表中查找获取到服务对象的引用。

·RMI 对象注册表

RMI 客户端和服务器都要用到 RMI 对象注册表。当服务端想要使一个对象对远程客户端可用时，它就要把这个对象(连同一个唯一名称) 注册到注册表上。服务对象注册之后，RMI客户端和在这个对象上调用的方法就能被找到了。

3. RMI 必须具备的

RMI 是 Java 库很好的特性。一个 JVM 中对象的方法能够被世界另一个角落的机器上的 JVM 来调用的能力确实很强大。但是要让 RMI 工作起来，有几件事情必须统筹起来。Java 是一个动态编程语言。它支持(运行时) 下载未在 JVM 的类路径上定义的新的类文件。灵活性的同时也带来了危险性。你可以想象一个糟糕的情形就是假如你的程序会下载别人写的类，而你却不知道他会在这个类中作些什么。

·使用 RMISecurityManager

使用 RMI 时，不存在客户端的代码可从服务器程序中动态下载。默认的，RMI 应用是不安装安全管理器(SecurityManager)的。假如一个没有任何防范的客户端下载到了恶意的代码那会是很危险的。为保障应用的安全性，RMI 服务程序必须安装一个特定的安全管理器(SecurityManger) 叫做 RMISecurityManager。

RMISecurityManager 运用了 Java 安全策略文件来决定什么权限应赋予 RMI 应用。默认的，Java 查找 JAVA.HOME/lib/security 目录下的策略文件。一个策略文件的配置条目示例如下：

```
grant {
  permission java.security.AllPermission;
};
```

上面这个策略文件配置条目给予了程序完全的访问权限。你可以用上面的设置让程序先工作起来，而后再紧缩你的安全策略。

[译者注] 图9.2 中的“框架”是单词 “Skeleton”的翻译，不是很贴切，但比较通用的译法就是这样子的。“桩”是 “Stub”的翻译。RMIRegistry 这里译作 RMI 注册表，后面又为 RMI 注册服务，感觉后者要贴切一些。

本章内容对于理解 RMI 也是不错的。

第九章. 使用 Quartz 的远程方式（第二部分）

4. 创建 Quartz RMI 服务端

你务必按几个步骤来配置 Quartz 来使用 RMI。其中的一些步骤会在创建 Quartz RMI 服务端用到，还有些步骤会在 Quartz 客户端连接服务端。我们先来阐述服务端的配置步骤。

·配置 Quartz RMI 服务端

第一步就是修改要部署到 Quartz RMI 服务端的 quartz.properties 文件。当在 Quartz 中使用 RMI，你还必须添加几个新的属性。表 9.1 包括了完整 RMI 属性列表。

表 9.1. RMI 服务端必要的属性

属性	默认值
org.quartz.scheduler.rmi.export **注：**假如你要使 Quartz 调度作为一个可用的 RMI 对象，这个标记必须设置为 true	false
org.quartz.scheduler.rmi.registryHost **注：**这是运行 RMI 注册表所在的主机	localhost
org.quartz.scheduler.rmi.registryPort **注：**这是 RMI 注册服务监听所用的端口号(通常是1099)	1099
org.quartz.scheduler.rmi.createRegistry **注：**这项决定了 Quartz 是否会创建 RMI 注册服务。如果你不希望 Quartz 创建注册服务就设置为 false 或 never。如果是希望 Quartz 首先尝试去使用已存在的注册服务，如果失败的话自行创建一个就设置为 true 或 as_needed。假如注册服务创建好了，它会使用给定的 registryPort 绑定到所给的 registryHost 上。	never
org.quartz.scheduler.rmi.serverPort **注：**这是 Quartz 调度器服务所绑定的端口号，在其中监听到来的连接。默认，RMI 服务会随机选择一个端口号作为调度器绑定到 RMI 注册服务的端口。	-1

表 9.1 中的所有属性都必须加到Quartz RMI 服务端所使用的 quartz.properties 文件中去。虽然这些属性都有默认值的，但最好还是显式的为它们指定值，以免产生混乱。代码 9.1 是一个 Quartz RMI 服务端所用的属性文件的样例。

代码 9.1. 用于 Quartz RMI 服务端的 quartz.properties 文件样例

```
#============================================================
# Configure Main Scheduler Properties
#============================================================
org.quartz.scheduler.instanceName = RMIScheduler


#============================================================
# Configure RMI Properties
#============================================================
org.quartz.scheduler.rmi.export = true
org.quartz.scheduler.rmi.registryHost = localhost
org.quartz.scheduler.rmi.registryPort = 1099
org.quartz.scheduler.rmi.serverPort = 0
org.quartz.scheduler.rmi.createRegistry = true

#============================================================
# Configure ThreadPool
#============================================================
```

```
org.quartz.threadPool.class = org.quartz.simpl.SimpleThreadPool
org.quartz.threadPool.threadCount = 10
org.quartz.threadPool.threadPriority = 5

#============================================================
# Configure JobStore
#============================================================
org.quartz.jobStore.misfireThreshold = 60000
org.quartz.jobStore.class = org.quartz.simpl.RAMJobStore
```

你认得代码 9.1 的 quartz.properties 文件的大部分设置的，这些在前面章节中讲过。我们只是往期中加了表 9.1 所列的属性。

·创建 Quartz RMI 服务端启动类

为启动 Quartz RMI 服务端，你必须创建一个启动类，该类从工厂中获取到调度器实例，然后运行这个调度器。即使是不用 RMI 也需要这一步。因为我们要在这个例子中运用 RMI，所以还有一些新的步骤要做。

首先，为清晰起见，我们把 quartz.properties 文件更名为 server.properties，这时候要告诉 Quartz RMI 服务端去加载新命名的文件而不是默认的 quartz.properties 文件。更改文件名会让我们调试问题变得容易些。这样，我们可以确保 Quartz 加载的是正确的设置文件。

第二个改变是：我们加载了一个新的安全管理器(SecurityManager)，以便能够赋予 RMI 服务端必须的权限。我们在本章的前面讨论过 RMI 安全管理器(RMISecurityManager)。

除了这些改变，代码 9.2 中的启动类看起还是很亲切的

代码 9.2. QuartzRMIServer 可用于启动 Quartz RMI 服务

```
package org.cavaness.quartzbook.chapter9;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.Date;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.quartz.Scheduler;
import org.quartz.SchedulerFactory;
import org.quartz.impl.StdSchedulerFactory;

public class QuartzRMIServer {

     public void run() throws Exception {
          Log log = LogFactory.getLog(QuartzRMIServer.class);

          // Use this properties file instead of quartz.properties
          System.setProperty("org.quartz.properties",
               "server.properties");

          // RMI with Quartz requires a special security manager
          if (System.getSecurityManager() == null) {
               System.setSecurityManager(new
                    java.rmi.RMISecurityManager());
          }
          // Get a reference to the Scheduler
          Scheduler scheduler =
               StdSchedulerFactory.getDefaultScheduler();

          /*
```

```
            * Due to the server.properties file, our Scheduler will
            * be exported to RMI Registry automatically.
            */
           scheduler.start();

           log.info("Quartz RMI Server started at " + new Date());
           log.info("RMI Clients may now access it. ");

           System.out.println("\n");
           System.out.println(
                "The scheduler will run until you type \"exit\"");

           BufferedReader rdr = new BufferedReader(
                     new InputStreamReader(System.in));

           while (true) {
                System.out.print("Type 'exit' to shutdown server: ");
                if ("exit".equals(rdr.readLine())) {
                     break;
                }
           }

           log.info("Scheduler is shutting down...");
           scheduler.shutdown(true);
           log.info("Scheduler has been stopped.");
      }
      public static void main(String[] args) throws Exception {

           QuartzRMIServer example = new QuartzRMIServer();
           example.run();
      }
}
```

在代码 9.2 中，安装了 RMISecurityManager 之后，通过工厂方法获得调度器实例，并调用它的 start() 方法。服务端是设计成在控制台运行的，因此一旦调度器启动之后，直至用户在控制台上键入 exit 。接着调度器被关闭也不再为远程的客户端提供服务了。

除了要使用 RMISecurityManager，我们注意到用不着在代码中做任何特别的事情，就能让 Quartz 调度器作为一个远程调度器来用。那些全是托 server.properties 文件的福所致。当调度器被创建后，假如属性文件告诉它这么做，调度器就会把自己导出并注册到 RMI 注册服务器上，并使之可被远程调用。

5. 使用 RMI 注册服务

需要运行一个 RMI 注册服务让客户端能访问到服务对象。你可以选择在命令行下使用 Java 的 rmiregistry 命令来运行注册服务，或者你可以允许 Quartz 自动启动注册服务。完全由你自己选择，但是，假如你没什么偏爱的话，让 Quartz 自己在需要的时候启动注册服务大概要简单些。

假如你要通过命令行启动注册服务，要确保你启动时所用的端口号要与属性文件所指定的一致。要从命令行启动，你应先进入到 <JAVA_HOME>/bin 目录下，然后键入如下命令：

rmiregistry <port>

假如你不指定端口号，会使用默认的 1099。这个默认值与 Quartz 所用的默认端口是一样的。

假如你不想从命令行中运行注册服务，在你为属性 org.quartz.scheduler.rmi.createRegistry 设置了正确值的情况下，Quartz 会自动启动注册服务。看表 9.1，这个属性可取以下几个值：

·false (never)

- true (as_needed)

- always

如果你想要 Quartz 来启动注册服务，为这个属性设置 true 或者 always 即可。

[译者注] 在为“Registry”和“Registry Server”用词选择上感觉不怎么贴切，如今都是用的“注册服务”与之对等，确也有译作注册表的，但总觉不妥，太容易与 Windows 系统注册表搞混。

第九章. 使用 Quartz 的远程方式（第三部分）

6. 创建 RMI 客户端

你需要创建一个客户端，用来调用远程 Quartz 调度器上的方法。客户端会同 RMI 注册服务器进行通信，进而定位到远程调度器对象，然后就能够调用其上的方法了。这些方法包括有暂停和停止调度器、部署和卸下 Job，和执行所有其他对与远程客户端可见的方法。

·配置 Quartz RMI 客户端

类似于表 9.1 所示服务端的配置，表 9.2 所列出的属性也是必须加到 Quartz RMI 客户端的。这两份属性列表必须分别应用到服务端和客户端的。

表 9.2 Quartz RMI 客户端所必须的属性

属性	默认值
org.quartz.scheduler.rmi.registryHost **注：**这是运行 RMI 注册服务所在的主机	localhost
org.quartz.scheduler.rmi.registryPort **注：**这是运行 RMI 注册服务所监听的端口(通常是 1099)	1099
org.quartz.scheduler.rmi.proxy **注：**假如你希望连接到远程服务端的调度器，设置 org.quartz.scheduler.rmi.proxy 标志为 true。你同时必须指定 RMI 注册服务进程的主机和端口号。	false

为了能让客户端定位到服务对象，它需要知道 RMI 注册服务运行在哪里，以便能查找到远程对象。org.quartz.scheduler.rmi.registryHost 和 org.quartz.scheduler.rmi.registryPort 属性必须是运行着 RMI 注册服务的主机和端口。假如你配置了 Quartz RMI 服务端自动启动注册服务，那么 RMI 注册服务器与 RMI 服务端就是同在一个机器上的。

因为你想要客户端能联系到远程调度器去部署 Job，你必须设置属性 org.quartz.scheduler.rmi.proxy 为 true。

代码 9.3 就是一个 RMI 客户端用来与服务器进行通信的 quartz.properties 示例文件

代码 9.3 一个用于 Quartz RMI 客户端的 quartz.properties 文件例子

```
#============================================================
# Configure Main Scheduler Properties
#============================================================
org.quartz.scheduler.instanceName = RMIScheduler
#org.quartz.scheduler.instanceId = AUTO

#============================================================
#Configure RMI Properties
#============================================================
org.quartz.scheduler.rmi.registryHost=localhost
org.quartz.scheduler.rmi.registryPort=1099
org.quartz.scheduler.rmi.proxy= true
```

除了上面所说的三个 RMI 属性之外，你曾经见过像代码 9.3 所示的 quartz.properties 属性文件。

> **客户端和服务器的实例名必须相匹配**
>
> 属性 org.quartz.scheduler.instanceName 在 RMI 客户端和服务端必须一致。不然，客户将无法在注册服务中查找到服务对象，会收一个客户端无法获取到远程调度器句柄的异常。

·创建 RMI 客户端类

在为客户端配置好了属性文件之后，你需要构建一个客户端 Java 类，去获取指向远程调度器的句柄并据此做点什么。创建这个类并不用我们做太多的事情，正如创建服务端类那般，我们把 quartz.properties 文件更名为 client.properties 并且告诉客户端从更名后的文件中加载属性。同样的，这样做的目的只是有助于我们清楚的知道是加载的哪一个属性文件以免产生混乱。除了这点这改变之外，你已经是多次的在前面章节中看到像代码 9.4 那样的配置了。

代码 9.4. 通过远程调度器部署一个 Job 的 Quartz RMI 客户端的例子

```
1.  package org.cavaness.quartzbook.chapter9;
2.
3.  import java.util.Date;
4.
5.  import org.apache.commons.logging.Log;
6.  import org.apache.commons.logging.LogFactory;
7.  import org.quartz.CronTrigger;
8.  import org.quartz.JobDataMap;
9.  import org.quartz.JobDetail;
10. import org.quartz.Scheduler;
11. import org.quartz.impl.StdSchedulerFactory;
12.
13. public class RMITestClient {
14.
15.      public void run() throws Exception {
16.
17.           Log log = LogFactory.getLog(RMITestClient.class);
18.
19.           // Use this properties file instead of quartz.properties
20.           System.setProperty("org.quartz.properties",
21.                 "client.properties");
22.
23.           // Get a reference to the remote scheduler
24.           Scheduler scheduler =
25.                StdSchedulerFactory.getDefaultScheduler();
26.
27.           // Define the job to add
28.           JobDetail job = new JobDetail("remotelyAddedJob", "default",
29.                     SimpleJob.class);
30.           JobDataMap map = new JobDataMap();
31.           map.put("msg", "Your remotely added job has executed!");
32.           job.setJobDataMap(map);
33.           CronTrigger trigger =
34.                new CronTrigger("remotelyAddedTrigger",
35.                   "default", "remotelyAddedJob", "default", new
36.                         Date(), null, "/5 * * ? * *");
37.
38.
39. // schedule the remote job
40.           scheduler.scheduleJob(job, trigger);
41.
42.           log.info("Remote job scheduled.");
43.      }
44.
45.     public static void main(String[] args) throws Exception {
46.           RMITestClient example = new RMITestClient();
47.           example.run();
48.      }
49. }
```

我们观察到一件有趣的事，也近乎魔术般的就是根本不用告诉工厂类我们想要的是一个远程调度器。工厂类是依据我们告诉它所加

载的 client.properties 文件知道这么做的。明确的讲就是，设置了 RMI 属性导引着工厂类创建了一个远程调度器：

org.quartz.scheduler.rmi.proxy = true

下面是 StdSchedulerFactory 的一个代码片断，它们决定了客户端去连接到一个远程调度器

```
if (rmiProxy) {

    if (autoId)
        schedInstId = DEFAULT_INSTANCE_ID;

    schedCtxt = new SchedulingContext();
    schedCtxt.setInstanceId(schedInstId);

    String uid = QuartzSchedulerResources.getUniqueIdentifier(
            schedName, schedInstId);

    RemoteScheduler remoteScheduler = new RemoteScheduler(schedCtxt,
            uid, rmiHost, rmiPort);

    schedRep.bind(remoteScheduler);

    return remoteScheduler;
}
```

以上代码出现在 StdSchedulerFactory 的 instantiate() 方法中。在这段代码中，工厂类检查为客户端设置的 rmiProxy 是否为 true。如果为 true，一个新的 RemoteScheduler 将被初始化并返回。这就是为什么我们的客户端并不需做任何特别的事情就行。返回到我们客户端的调度器实例实质上就是一个远程调度器(RemoteScheduler) 实例，因为 RemoteScheduler 实现了 Scheduler 接口，所以客户端代码可以对此毫不知情。

7. 测试 RMI 服务端和客户端

我们最后终于来到了可以运行客户端和服务端去测试 RMI 配置的时刻了。第一件要做的事情是启动 Quartz RMI 服务端。除了确保基本的 Quartz JAR 包和 server.properties 文件包含在 classpath 下之外，不用去做其他特别的事情了。运行 RMI 用不着附加的 JAR 文件，只是需要运行任何一个 Quartz 程序所必须的包而已。

·运行 Quartz RMI 服务端

要运行 Quartz RMI 服务端，仅仅是像运行别的 Java 类一样运行 QuartzRMIServer 类。就在前面提到过，要确保 classpath 下包含了所需的 JAR 文件和 server.properties 文件。要完成这个，最简单的方式就是创建一个批处理文件(或shell 脚本)，把需要的东西写在其中。代码 9.4 就是一个用来启动服务端的简单的批处理文件。

代码 9.4. 一个简单的用来启动 Quartz RMI 服务端的 startserver.bat 批处理文件

```
java org.cavaness.quartzbook.chapter9.QuartzRMIServer
```

你还需要在代码9.4 的命令行中包含所需的 JAR 文件到 classpath 上才能正常工作。这包括有 quartz.jar、commons-logging.jar、commons-logging-api.jar、commons-collections3.1.jar、beanutils.jar、commons-beanutils-bean-collections.jar、commons-beanutils-core.jar。

·运行 Quartz RMI 客户端

当服务端运行起来之后，你就可以运行 RMI 客户端了。客户端能以与服务端同样的方式运行--创建一个批处理文件或 shell 脚本。当启动客户端后，假如一切正常的话，你将不仅仅能看到从客户端控制台输出了一个远程 Job 被部署了的消息，还能看到在服务端控制台打印出一条消息，是说接收到了远程 Job 并部署运行。

尽管这个 RMI 客户端被设计为部署一个 Job 后就退出，Quartz RMI 服务端被设计为持续运行直到键入 exit 为止。服务端可以写成是接收到一个远程 Job，进行部署，然后执行自己的逻辑去等待更多的客户连接。这就意味着你能多次的运行客户端，像这样

的设计应该才是你想要的。

当你完成了运行这个例子之后，只要键入 exit 即可。

·接下来是什么？

本章介绍了与 Quartz 调度器打交道的一种新的方式，这在之前是不可能的。在 Quartz 中使用 RMI，允许你构建组件分离的程序并能分布在跨平台的机器中。这是一个非常好的概念，因为它在没有增加额外开发工作量的情况下提供了更好的可伸缩性。

第十章. J2EE 中使用 Quartz (第一部分)

Java 2 平台企业版定义了基于组件的企业应用开发标准。不管你是否倾向于使用一种开源的 J2EE 服务器，比如 JBoss 或 Geronimo，或者你更希望得到可靠安全的商业服务支持，比如 WebSphere 和 WebLogic，你都能在那些应用服务器中使用几种不同的部署方式使用 Quartz。本章给你示范了在 J2EE 应用服务器中以不同策略部署 Quartz，你也会看到 Quartz 框架更加丰富了 J2EE。

一：假如我有 J2EE，为什么还需要 Quartz?

自从 20 世纪 90 年代末期 J2EE 首次登上舞台以来，开发人员就被某些规格决议和一些表面看来明显缺失的特征所困惑。这没必要去批判规格的制定者，更多的是说明了当有不同意见和议程的独立团体，在尝试达成统一的优先级时，就像联合国进行决议时那样，并不如所想像的那样好。许多必须的特征获得认可，但是一些关键的特征被略去留待以后加入。其中一个让早期规范遗漏的关键特征就是定时器服务。

·我需要定时器服务

许多商业流程需要异步的作业调度。例如，Web 站点通常需要向注册用户发邮件告知他们有新的特性或是邮寄最新期刊。医疗索赔处理公司典型的也是需要在晚上下载医疗数据然后拿来处理后续事情。一个销售多种类型产品公司也许要每个晚上生成销售和佣金信息的数据报表。所有前面的场景都需要一个定时器服务来执行异步的作业。

Java/J2EE 社区对解决定时器服务作过多次的尝试。在早期，厂商在他们的 J2EE 服务器中加入适当的解决方案。（在本章上，述语 J2EE 服务器和 J2EE 容器是指同一概念。）例如，WebLogic 产品中有一些定制的扩展，在 IBM 的 J2EE 服务器中也这么做的。不幸的是，它们是不能兼容的，你不能把组件从一种服务器迁移到另一种服务器中。后来，WebLogic 和 IBM 还有其他的厂商试着开发一套通用的定时器组件。

从 Java 1.3 开始，Sun 在 Java 核心中加入了 java.util.Timer 和 java.util.TimerTask 类来帮助实现基本的定时器功能。尽管 Timer 和 TimerTask 能满足简单的需求，但有更多的作业调度仅用这两个具体类做不到的，希望你已经明白了这一点。

·Quartz/J2EE 部署模型

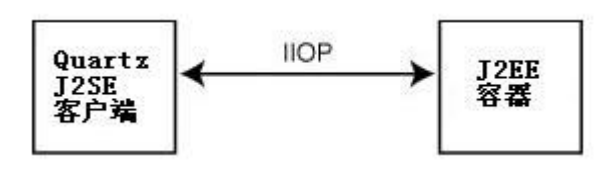
现在有两个策略用来在 J2EE 中构架和部署 Quartz。其中之一是，你可以把 Quartz 设计在 J2EE 容器外部作为一个标准的 J2SE 客户端。你很快就会看到，这种方法是最简单的。第二种策略是部署 Quartz 驻留在 J2EE 容器中作为其客户端。这种场景下，J2EE 客户端就是一个像其他 Web 应用程序一样部署 Web 归档 (WAR) 文件。策略的选择有赖于你确切的需求。每一种策略都有其好处与缺点。

·作为 J2SE 客户端运行 Quartz

假如你正需要调用 Enterprise JavaBean (EJB) 的服务，或者要把消息送入到 JMS 的队列或主题中，最简单的配置方式就是在 J2EE 容器之外作为一个 J2SE 程序运行。这时它的功能就像任何别的在容器外运行的程序一样，只是它需要去调用容器内分布式组件上的方法。

我们实际上在前面的章节中使用过这种方式，只是未提及调用 EJB 。你可以创建一个 Quartz 应用，包含了 Quartz 库和作业调度信息，并通过 Home 和 Remote 接口连接到 J2EE 服务器。然后你就可以像其他分布式客户端那样调用 EJB 上的方法了。你也可以新建并插入 JMS 消息让容器中运行的消息驱动 Bean (MDB) 来处理它们。在图 10.1 中显示了这种用法。

图 10.1. Quartz 可作为一个独立的 J2SE 客户端与 J2EE 协同工作

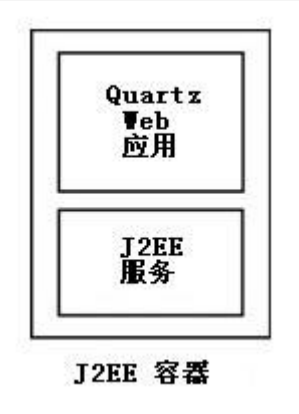


如果你有已存在的 J2EE 组件需要 Quartz 与其交互，并且你不希望或无法对服务器作任何改变，用这种这种方法很适合。你只需要像我们前面章节介绍的那样构建一个 Quartz 应用，并使用 Quartz 自带的 EJBInvokerJob。我们马上就会讨论到 EJBInvokerJob 的。

·部署 Quartz 到 J2EE 服务器中

Quartz 也直接能部署在容器中，这样就不需要外部的 Quartz 应用了。这通常被称之为 J2EE 客户端。你选择这种方式也不是之前的方式也许有多个理由。其中之一是你仅需要维护一个应用，而另一种方式需要维护两个应用。中，可以让 Quartz 使用某些容器提供的资源，例如邮件会话、数据源和其他的适配器资源。假如你是在集群环境中使用 J2EE 服务器，这也让部署在容器中的 Quartz 集群更简单和易于管理。图10.2 描绘了 Quartz 如何部署与 J2EE 应用一同部署的。

图 10.2. Quartz 可与 J2EE 应用一同部署



当 Quartz 被部署在 J2EE 容器中，你必须清楚并能处理一些兼容性的问题，在论及这个话题之前，让我们来讨论如何部署 Quartz 为 J2SE 客户端并从容器外访问容器。

·作为 J2SE 客户端运行 Quartz

到目前为止，最简单容易的在 J2EE 中使用 Quartz 的方式是把它部署在容器之外。这种方式这所以简单一方面是因为你无需去处理许多问题，比如 Quartz 如何尝试在 J2EE 容器中创建线程并执行他们的；另一方面是因为即使你使用最新的工具和技术，例如 XDoclet 或管理控制台，部署应用到 J2EE 容器中也是让人感到沮丧的。

·使用 Quartz 的 EJBInvokerJob 去调用 EJB

Quartz 框架中包含 org.quartz.jobs.ee.ejb.EJBInvokerJob 类，它允许你部署一个 Job 当触发时去调用 EJB 的方法。不管你选择的是哪种 Quartz 部署方式，这个 Job 都很容易建立并使用。我们假定，你有一个像代码 10.1 所列的无状态会话 Bean (SLSB)。

代码 10.1. 一个无状态会话 Bean 的例子

```
1.  import java.rmi.RemoteException;
2.
3.  import javax.ejb.EJBException;
4.  import javax.ejb.SessionBean;
5.  import javax.ejb.SessionContext;
6.
7.  public class TestBean implements SessionBean {
8.       /** The session context */
9.       private SessionContext context;
10.
11.      public TestBean() {
12.           super();
13.      }
14.
15.      // EJB Lifecycle Methods not shown for brevity
16.
17.      public void helloWorld() throws EJBException {
18.           System.out.println("Hello World");
19.      }
20.
21.      public void helloWorld(String msg) throws EJBException {
```

```
            System.out.println("Hello World - " + msg);
        }
}
```

把这个 EJB 部署到你所选的 J2EE 应用服务器中并准备就绪，你就可以使用 EJBInvokerJob 去调用其中的一个对于远程客户端可见方法 helloWorld()。

你只要像其他的 Job 那般建立 EJBInvokerJob。代码10.2 展示了如何用 EJBInvokerJob 去调用那个 SLSB 上的 helloWorld() 方法。

代码 10.2. 一个使用 EJBInvokerJob 的简单例子

```
package org.cavaness.quartzbook.chapter10;

import java.util.Date;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.quartz.JobDetail;
import org.quartz.Scheduler;
import org.quartz.SchedulerException;
import org.quartz.Trigger;
import org.quartz.TriggerUtils;
import org.quartz.impl.StdSchedulerFactory;
import org.quartz.jobs.ee.ejb.EJBInvokerJob;

public class Listing_10_2 {
     static Log logger = LogFactory.getLog(Listing_10_2.class);

     public static void main(String[] args) {
          Listing_10_2 example = new Listing_10_2();

          try {
               // Create a Scheduler and schedule the Job
               Scheduler scheduler = example.createScheduler();
               example.scheduleJob(scheduler);

               // Start the Scheduler running
               scheduler.start();

               logger.info("Scheduler started at " + new Date());


          } catch (SchedulerException ex) {
               logger.error(ex);
          }
     }

     // Schedule the EJBInvokerJob
     private void scheduleJob(Scheduler scheduler) throws SchedulerException {

          // Create a JobDetail for the Job
          JobDetail jobDetail = new JobDetail("HelloWorldJob",
                    Scheduler.DEFAULT_GROUP,
                    org.quartz.jobs.ee.ejb.EJBInvokerJob.class);

          loadJobDataMap(jobDetail);

          // Create a trigger that fires every 10 seconds, forever
```

```
48.             Trigger trigger = TriggerUtils.makeSecondlyTrigger(10);
49.             trigger.setName("helloWorldTrigger");
50.             // Start the trigger firing from now
51.             trigger.setStartTime(new Date())
52.
53.             // Associate the trigger with the job in the scheduler
54.
55.                scheduler.scheduleJob(jobDetail, trigger);
56.           }
57.
58.           /*
59.            * Configure the EJB parameters in the JobDataMap
60.            */
61.           public JobDetail loadJobDataMap(JobDetail jobDetail) {
62.                jobDetail.getJobDataMap().put(EJBInvokerJob.EJB_JNDI_NAME_KEY,
63.                          "ejb/HelloWorldSession");
64.
65.                jobDetail.getJobDataMap().put(EJBInvokerJob.EJB_METHOD_KEY,
66.                          "helloWorld");
67.
68.                jobDetail.getJobDataMap().put(EJBInvokerJob.PROVIDER_URL,
69.                          "t3://localhost:7001");
70.
71.                jobDetail.getJobDataMap().put(
72.                     EJBInvokerJob.INITIAL_CONTEXT_FACTORY,
73.                          "weblogic.jndi.WLInitialContextFactory");
74.
75.                return jobDetail;
76.
77.           }
78.
79.           /*
80.            * return an instance of the Scheduler from the factory
81.            */
82.           public Scheduler createScheduler() throws SchedulerException {
83.                return StdSchedulerFactory.getDefaultScheduler();
84.           }
85. }
```

正如你在代码 10.2 中看到的，EJBInvokerJob 采用和别的 Job 一样的配置方式。创建了一个相关的 JobDetail 和 trigger 并注册到调度器中。需要针对不同的 J2EE 容器来配置 JobDataMap 中的参数，才能让 Job 正确的执行。表 10.1 列出了这个 Job 所支持的 JobDataMap 参数。

要往 JobDataMap 中加入什么参数依赖于所使用的 J2EE 服务器和自身的需求。例如，假如你使用的是 BEA 的 WebLogic，你必须至少指定代码 10.1 中所列的参数。显示地，必须替换为你特定环境的参数值。假如你是用的 WebSphere，大部份参数值是不一样的。

当我们建立好 Job 并在我们的外部 Quartz 应用程序中运行 10.2 中的代码，每隔 10 秒钟 EJB 上的方法 helloWorld() 就被调用一次。这种方式好就好在我们不需要为部署 Quartz 应用到 J2EE 容器上而费心。这也强制了 Job 信息与商业处理逻辑的分离。

表 10.2. EJBInvokerJob 所用的参数，具体依赖于特定的 J2EE 服务器

静态常量	字面值
EJB_JNDI_NAME_KEY 注：Bean 的 home 接口的 JNDI 名	ejb
PROVIDER_URL 注：特定于厂商的 URL，指示哪里能找到服务器	java.naming.provider.url

INITIAL_CONTEXT_FACTORY **注：**特定于厂商的上下文工厂，用于查找资源的	java.naming.factory.initial
EJB_METHOD_KEY **注：**在 EJB 上要调用的方法名称	method
EJB_ARGS_KEY **注：**Object[] 类型，传递给方法的参数值对象数组(可选，如果不设表明无参数)	args
EJB_ARG_TYPES_KEY 注：Class[] 类型，传递给方法参数类型的类数组(可选，如果不调表示每个参数的类型都源自于相应参数的 getClass() 返回)	argsTypes
PRINCIPAL **注：**调用 EJB 方法的主体(用户)	java.naming.security.principal
CREDENTIALS **注：**调用 EJB 方法的凭证	java.naming.security.credentials

在代码 10.2 所示例子中，在 EJB 上被调用的方法 helloWorld() 没有定义任何参数。EJBInvokedJob 类允许你通过指定 EJB_ARGS_KEY 和 EJB_ARG_TYPES_KEY 属性来传递方法参数，见表 10.1。

代码 10.3 展示了另一个简单的例子，传递一个参数来调用运行在 Apache Geronimo J2EE 服务器中的 EJB 的另一版本的 helloWord() 方法。

代码 10.3 和代码10.2 十分相像，除多包含了 EJB_ARGS_KEY 和 EJB_ARG_TYPES_KEY 参数设置。还因为 EJB 是运行在 Geronimo J2EE 应用服务器中，所以还需要加上参数 PRINCIPAL 和 CREDENTIALS。

代码 10.3. 使用 EJBInvokerJob 的简单例子

```
1.  package org.cavaness.quartzbook.chapter10;
2.
3.  import java.util.Date;
4.
5.  import org.apache.commons.logging.Log;
6.  import org.apache.commons.logging.LogFactory;
7.  import org.quartz.JobDetail;
8.  import org.quartz.Scheduler;
9.  import org.quartz.SchedulerException;
10. import org.quartz.Trigger;
11. import org.quartz.TriggerUtils;
12. import org.quartz.impl.StdSchedulerFactory;
13. import org.quartz.jobs.ee.ejb.EJBInvokerJob;
14.
15. public class Listing_10_3 {
16.      static Log logger = LogFactory.getLog(Listing_10_3.class);
17.
18.      public static void main(String[] args) {
19.
20.           Listing_10_3 example = new Listing_10_3();
21.
22.           try {
23.
24.               // Create a Scheduler and schedule the Job
25.               Scheduler scheduler = example.createScheduler();
26.               example.scheduleJob(scheduler);
27.
28.               // Start the Scheduler running
29.               scheduler.start();
30.
31.               logger.info("Scheduler started at " + new Date());
32.
33.           } catch (SchedulerException ex) {
34.                logger.error(ex);
```

```
35.             }
36.         }
37.
38.         // Schedule the EJBInvokerJob
39.         private void scheduleJob(Scheduler scheduler)
40.             throws SchedulerException {
41.
42.             // Create a JobDetail for the Job
43.             JobDetail jobDetail = new JobDetail("HelloWorldJob",
44.                     Scheduler.DEFAULT_GROUP,
45.                     org.quartz.jobs.ee.ejb.EJBInvokerJob.class);
46.
47.             // Load all of the necessary EJB parameters
48.             loadJobDataMap(jobDetail);
49.
50.             // Create a trigger that fires every 10 seconds, forever
51.             Trigger trigger = TriggerUtils.makeSecondlyTrigger(10);
52.
53.             trigger.setName("helloWorldTrigger");
54.             // Start the trigger firing from now
55.             trigger.setStartTime(new Date());
56.
57.             // Associate the trigger with the job in the scheduler
58.             scheduler.scheduleJob(jobDetail, trigger);
59.
60.         }
61.
62.         /*
63.          * Configure the EJB parameters in the JobDataMap
64.          */
65.         public JobDetail loadJobDataMap(JobDetail jobDetail) {
66.             jobDetail.getJobDataMap().put(
67.                 EJBInvokerJob.EJB_JNDI_NAME_KEY, "ejb/Test");
68.
69.             jobDetail.getJobDataMap().put(EJBInvokerJob.EJB_METHOD_KEY,
70.                     "helloWorld");
71.
72.             Object[] args = new Object[1];
73.             args[0] = " from Quartz";
74.             jobDetail.getJobDataMap().put(
75.                 EJBInvokerJob.EJB_ARGS_KEY, args);
76.
77.             Class[] argTypes = new Class[1];
78.             argTypes[0] = java.lang.String.class;
79.             jobDetail.getJobDataMap().put(
80.                 EJBInvokerJob.EJB_ARG_TYPES_KEY, argTypes);
81.
82.             jobDetail.getJobDataMap().put(
83.                 EJBInvokerJob.PROVIDER_URL, "127.0.0.1:4201");
84.
85.             jobDetail.getJobDataMap().put(
86.                 EJBInvokerJob.INITIAL_CONTEXT_FACTORY,
87.                         "org.openejb.client.RemoteInitialContextFactory");
88.             jobDetail.getJobDataMap().put(
89.                 EJBInvokerJob.PRINCIPAL, "system");
90.
91.             jobDetail.getJobDataMap().put(
92.                 EJBInvokerJob.CREDENTIALS, "manager");
93.
94.             return jobDetail;
95.         }
```

```
96.
97.        /*
98.         * return an instance of the Scheduler from the factory
99.         */
100.       public Scheduler createScheduler() throws SchedulerException {
101.            return StdSchedulerFactory.getDefaultScheduler();
102.       }
103.  }
```

EJBInvokerJob 参数和序列化

在论及 Java 和分布式程序时会遇到一个典型的问题就是序列化，你应该尽量的传递 String 或原始数据类型给 EJB 的方法。假如你需要传递复杂的类型，你的代码必须使对象在服务端和客户端之间序列化传输。要知道关于 Java 序列化的深层次的信息，请在 http://java.sun.com/j2se/1.5.0/docs/guide/serialization 查看 Sun 的序列化规范。

由于 Quartz 需要拿到 EJB 的 home 和远程接口，你必须部署某些 J2EE 客户端 JAR 包到你外部的 Quartz 应用中。需要哪些 JAR 文件依赖于你所用的 J2EE 容器。例如，假如你是用的 WebLogic，则要在 Quartz 应用程序中引入 weblogic.jar 包。是 Geronimo 的话，就得引入多个相关包。查看特定应用服务器的文档来确保这一点。

第十章. J2EE 中使用 Quartz (第二部分)

·在 J2EE 应用服务器中运行 Quartz

作为 J2EE 客户端运行 Quartz 比运行为外部 J2SE 应用程序稍显繁琐。这主要是因为部署一个应用到容器中有点了儿复杂，J2EE 规范对容器中的组件会有些约束。其中最大的原则之一就是涉及到该由谁来创建线程。因为 J2EE 容器有责任去管理所有资源，所以它并非允许谁想谁就能创建线程的。假如是这样的话，容器就会要更艰难的去管理环境和保证一切稳定性。Quartz 会创建自己的工作者线程，所以你必须依照一些步骤来保证它能正常的运转。

确保代像代码 10.1 那样的一个无状态会话 Bean 已部署到容器中。最简单的部署 Quartz 到容器中的方式是构建一个包含所必须文件的 WAR 包，然后使用管理工具或 Eclipse 部署这个 Web 应用到容器中。

这个 Web 应用的目录结构像其他任何 Web 应用是一样的。你必须添加以下文件至其中：

- ·web.xml (放置到 WEB-INF 下)
- ·quartz.properties (放置在 WEB-INF/classes 下)
- ·quartz_jobs.xml (放置在 WEB-INF/classes 下)
- ·Quartz 二进制包 (放置在 WEB-INF/lib 下)
- ·所需第三方包 (放置在 WEB-INF/lib 下)

因为正在构建一个 Web 应用，所以要加入一个必备的 web.xml 作为部署描述文件。代码10.4 中显示了我们要安装到容器中的客户端应用的 web.xml 文件。

代码 10.4. Quartz J2EE 客户端程序的 web.xml 文件

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.
3.  <web-app>
4.    <servlet>
5.      <servlet-name>QuartzServlet</servlet-name>
6.      <servlet-class>
7.        org.quartz.ee.servlet.QuartzInitializerServlet
8.      </servlet-class>
9.      <load-on-startup>1</load-on-startup>
10.   </servlet>
11.
12.   <servlet-mapping>
13.     <servlet-name>QuartzServlet</servlet-name>
14.     <url-pattern>/servlet/QuartzServlet</url-pattern>
15.   </servlet-mapping>
16. </web-app>
```

Quartz 框架包含有一个名为 QuartzInitializerServlet 的 Java Servlet，当被调用时它会初始化 Quartz 调度器并加载 Job 信息。在代码 10.4 中， 我们看到有设置 <load-on-startup> 标记值为 1，这指示着在容器启动的时候这个 servlet 会被自动加载并初始化。通过使用这个 servlet 去启动 Quartz 调度器，我们规避了容器中创建线程的约束，因为容器将允许 servlet 去创建用户线程的。

> **加入 QuartzInitializerListerner 到 Quartz 中**
>
> 不久前，一个新的名为 QuartzInitializerListener 被加入到 Quartz 中来，它实现了 javax.servlet.ServletContextListener 接口。前面也有提过，这个类可用来替代

QuartzInitializerServlet 类。

接下来，你需要把标准的 quartz.properties 文件放入到 Web 应用的 WEB-INF/classes 目录中去。在这里的这个文件没什么特别的；实质上，这一步与前面同类的操作是一样的。然而，我们这里使用到 JobInitializationPluin (这在第八章，"使用 Quartz 插件"，它设计为从 XML 文件中加载 Job 信息)。默认情况下，这个插件查找一个叫做 quartz_jobs.xml 文件并从中加载所配置的 Job。如第八章描述的，使用这一特定的插件可以避免你写加载作业的代码，且在代码改变时不得不重新编译。quartz_jobs.xml 的内容如 代码 10.5 所示。

代码 10.5. J2EE 客户端所用的 quartz_jobs.xml 文件

```
<?xml version='1.0' encoding='utf-8'?>

<quartz>
  <job>
    <job-detail>
        <name>HelloWorldJob</name>
        <group>DEFAULT</group>
        <job-class>org.quartz.jobs.ee.ejb.EJBInvokerJob</job-class>
        <volatility>false</volatility>
        <durability>false</durability>
        <recover>false</recover>

        <job-data-map allows-transient-data="true">
          <entry>
            <key>ejb</key>
            <value>ejb/Test</value>
          </entry>
          <entry>
            <key>java.naming.factory.initial</key>
            <value>org.openejb.client.RemoteInitialContextFactory</value>
          </entry>
          <entry>
            <key>java.naming.provider.url</key>
            <value>127.0.0.1:4201</value>
          </entry>
          <entry>
            <key>method</key>
            <value>helloWorld</value>
          </entry>
          <entry>
            <key>java.naming.security.principal</key>
            <value>system</value>
          </entry>
          <entry>
            <key>java.naming.security.credentials</key>
            <value>manager</value>
          </entry>
        </job-data-map>
    </job-detail>
    <trigger>
      <simple>
          <name>helloWorldTrigger</name>
          <group>DEFAULT</group>
          <job-name>HelloWorldJob</job-name>
          <job-group>DEFAULT</job-group>
          <start-time>2005-06-10 6:10:00 PM</start-time>
          <!- repeat indefinitely every 10 seconds >
          <repeat-count>-1</repeat-count>
          <repeat-interval>10000</repeat-interval>
      </simple>
```

```
    </trigger>
  </job>
</quartz>
```

你能看出在代码 10.5 中，我们仍然使用 EJBInvokerJob，只是声明在了 quartz_jobs.xml 文件中了。

> **在 quartz.properties 中指定插件**
>
> 第八章已讲过，在使用 Quartz 插件就必须在 quartz.properties 文件中指定相应的插件信息。对于 JobInitialzationPluin，你必须在这个属性文件中加入下面一行代码：
>
> org.quartz.plugin.jobInitializer.class = org.quartz.plugins.xml.JobInitializationPlugin

上面的文件都配置好后，就可以构建一个 WAR 文件并部署到你的容器中。当容器启动之后，就会加载那个 servlet 并初化，进而启动了调度器。调度器利用 JobInitializerPluin 去从 quartz_jobs.xml 文件中加载作业信息。自此，EJBInvokerJob 就会调用 EJB 上的 helloWorld() 方法了。

·包含 J2EE 客户端 JAR 包

在打包 J2EE 客户程序时，你需要把所需的特定于服务器的 J2EE 客户端 JAR 包打进来。针对不同的容器会有所不同，你需要参照具体文档予于决定。在构建一个独立运行的 Quartz 应用时，你还要加入所需的 Quartz 包。

二：使用 J2EE 容器的数据源

直到此刻，我们有意没去提到 JobStores 和 DataSources。在第六章，"作业存储和持久化"，你已经学到可以把作业信息存储在内存中，或者假你需要作业信息在两次程序重启之间能持久保存下来，你就可以把作业信息存储到关系型数据库中。存在有两种类型的 JDBC 作业存储方式：

·JobStoreTX 在持久化操作过程中自己管理自己的事特

·JobStoreCMT 在持久化操作过程中技术容器管理事物(CMT)

如果你使用了 J2EE 容器并选择了上面的种类型的 JDBC JobStores，这时候你最好应该用容器提供的数据源。参考前面第六章，当在 J2EE 容器中使用 JDBC JobStores 时如何设置 quartz.properties 文件。

三：使用其他的 J2EE 资源

当部署 Quartz 到 J2EE 容器中，你可以利用 J2EE 组件可用的其他资源。例如，假如你需要发送 email，一种途径是得用 Quartz 的 SendMailJob，它依赖于 JavaMail。另一可用途径是，假如你把 Quartz 部署在容器中，能使用所有 J2EE 服务器都可提供的 mail session 资源，当然，前提是你在容器中已配置好的 mail session。那是作为 J2EE 客户端来部署 Quartz 好处之一。

```
InitialContext initialContext = new InitialContext();
Session session = (Session)
initialContext.lookup(urlToMailSession);

Message msg = new MimeMessage(session);
// ... build up msg
Transport.send(msg);
```

四：EJB 2.1 规范：最后的曙光

在 EJB 2.1 的第 22 章中讨论到企业 JavaBean 的一个新的特性，定时器服务。这是一个容器管理的服务，给予需要基于时间事件的组件回调服务。实质上就是 EJB 能通过这一服务注册他们自己，当到了它们要执行的时间时就会接收到一个通知。定时器服务是被 EJB 容器实现并管理着的。现在还不知道 J2EE 厂商还要在那些规范后面加上多少功能。有些人争辩说 EJB 定时器的提议内容还不充分，同样的原因，java.util.Timer 类也不足以真正应付调度程序。最好也看看 EJB 的架构规范，可加入对框架的插件支持，例如 Quartz 就可以用来增强定时器服务的灵活性。

第十一章. Quartz 集群（第一部分）

第十一章. Quartz 集群

不可避免的，我们还是要说到集群。虽然单个 Quartz 实例能给予你很好的 Job 调度能力，但它不能令典型的企业需求，如可伸缩性、高可靠性满足。假如你需要故障转移的能力并能运行日益增多的 Job，Quartz 集群势必成为你方言的一部分了。本章就告诉你如何使用 Quartz 的集群能力来更好的支持你的业务需求，并且即使是其中一台机器在最糟的时间崩溃了也能确保所有的 Job 得到执行。

一. 集群对 Quartz 来说意味着什么？

集群扮演着运行一个组件或应用的多个实例，它们以透明的方式提供服务。集群是企业范畴的事物，而不局限于 Java 的世界里。当部署 J2EE 应用时，例如，供应商为应用服务器提供了集群的能力，以便于像 EJB、JNDI 和 Web 组件能获得高可用性。然当客户端请求这些服务时候，它们就能更可靠的提供服务。

这和一些用户要求他们的 Quartz 应用的行为是完全一样的。用户希望构建并设定 Quartz 应用是当一个 Job 一定要执行时，它就要得到执行。随着你的 Quartz 应用变得更普及，还要安排不断增多的需求，Quartz 应用的集群将能让你安下心来，你能处理你的需求并确保一切按计划进行。而且你只需要付出很小的努力来搭建和维护便能得到这些好处。

·Quartz 应用集群的好处

集群的 Quartz 应用比起非集群环境，能提供两个关键的好处

·高可用性

·伸缩性

·高可用性

高可用性的应用是指能以高百分率的时间服务于客户。在某些情况下，可能就是一周 7 天，一天 24 小时。对于其他的应用，可能只是“尽量多的时间”。可用性通常表示为在 0 到 100 之间的一个百分数。一个应用也许经常失败但仍可达到高可用性。另一方面，一个应用可能只萨掉一次但是随后处在萨掉状态很长时间，可用性就低了。计量可用性不是应用的萨掉次数，而是萨掉状态经历总的时间。显然，作为开发者来说，我们希望我们的应用从不失败。这也是可以出现的，不过你必须对此有所准备。

硬件和软件的可用性级别有时候是以九的级别来衡量的。九的级别指示了在可用性百分数中有多少个数字九。例如，99.999 说的是达到五个九的级别，因为这里有五个九。表 11.1 显示了在某一级别的近似宕机时间百分比。

表 11.1. 应用的可用性级别

可用性	每年近似的宕机小时数
99%	87.6 小时
99.9%	8.8 小时
99.99%	.9 小时
99.999%	0.09 (大约 5 分钟)

看到了表 11.1，你或许能得到结论，四个九的级别(大约每年宕机一小时) 就是一个令人叹为观止的高用性了，通常就是这样子的。然而，假如是一个 Quartz 应用，它是被设计来传送发票的，要是它连续 12 天里每天宕机 5 分钟的话，那些发票就会被认作过期，那么 生意上就会承受大量的损失，你也很可以要去寻到一份新的工作了(我可不是说的 Quartz 的那种 Job)。这不仅仅是宕机时间的总和，而是一旦宕机，程序就要罢工。

其中一个能获得尽可能高可用性的分子就是故障转移的概念。故障转移确保了即使是系统出现故障时，其他冗余的服务组件能处理请求，使得客户端(或 Job) 能从失败中恢复过来。从故障组件或服务中倒换到另一相同功能的组件或服务的能力增强了应用的可用性。这一切换或故障转换应当是透明的。

·伸缩性

伸缩性意味着为增进应用自身能力，可以动态增加新的资源(如硬件) 到应用环境中的能力。在一个可伸缩的应用中，为达到增进能力的做法不涉及到改变代码或是设计。

获得伸缩性可不是变魔术那样。一个程序必须从一开始就要有合理的设计；支撑额外的能力通常要耗费管理上的努力来增加新的硬件(如内存) 或是启动程序更多的实例。

·负载均衡

作为获得更好的伸缩性，在集群中跨节点分布式工作的能力是很重要的。把工作散布开能确保集群中的每一个节点会共同承担工作负载。想像一下假如集群中所有的工作都指定给某一个节点，同时其他节点处于空闲状态。继续这种模式，终究这个过载的节点将无法处理增加的工作而导致失败。

最好的场景是，工作均匀的分布在集群中的所有实例上。有几个不同的算法能用来分布工作，包括随机法、循环法和加权循环法，只列了这些。

当前，Quartz 使用的是随机算法来提供最低限度负载均衡的能力。集群中的每个 Scheduler 实例尝试 Scheduler 所允许尽量快的速度触发已布署的 Trigger。Scheduler 实例通过触发自己的 Trigger 来竞争(使用数据库锁) 得到执行 Job 的权利。当某个 Job 的 Trigger 被触发时，别的 Scheduler 实例就不再试图触发这一 Trigger 了，直到下一次触发时间的到来。这种机制可能比你从它的简单性所推断的还要工作的更好。这是因为 Scheduler "多数时候是忙的"，作为至少的一个都有希望找到下次要触发的 Job。因此，这样就可能获得近乎完全合理的负载均衡性。

第十一章. Quartz 集群（第二部分）

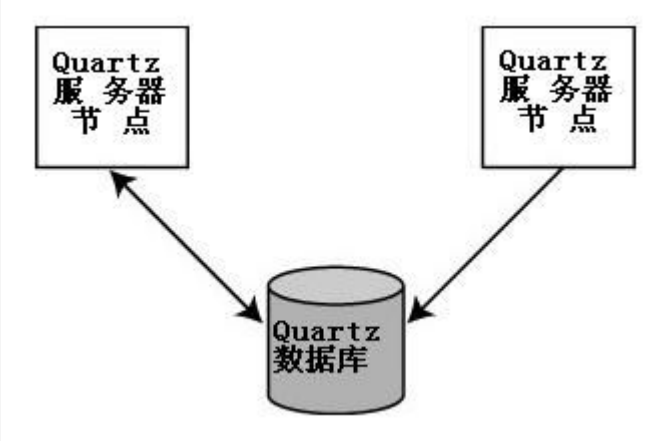
二. Quartz 中集群是如何工作的

一个 Quartz 集群中的每个节点是一个独立的 Quartz 应用，它又管理着其他的节点。意思是你必须对每个节点分别启动或停止。不像许多应用服务器的集群，独立的 Quartz 节点并不与另一其的节点或是管理节点通信。(将来的 Quartz 版本将会设计成让节点能与其他节点直接通信，而不是借助于数据库。) 取而代之的是，Quartz 应用是通过数据库表来感知到另一应用的。

> **Quartz 集群仅能使用 JDBC JobStore 工作**
>
> 因为集群中节点依赖于数据库来传播 Scheduler 实例的状态，你只能在使用 JDBC JobStore 时应用 Quartz 集群。这意味着你必须使用 JobStoreTX 或是 JobStoreCMT 作为 Job 存储；你不能在集群中使用 RAMJobStore 的。在将来的释放版中非常可能移除这个需求，节点也将能直接与另一节点直接通过网络协议，可能使用 JGroup 进行通信。

图 11.1 显示了每个节点直接与数据库通信，若离开数据库将对其他节点一无所知

图 11.1. Quartz 集群中的每个节点感知到其他实例只能经由数据库



·Quartz Scheduler 在集群中的启动

Quartz Scheduler 自身是察觉不到被集群的，只有配置给 Scheduler 的 JDBC JobStore 才知道。当 Quartz Scheduler 启动时，它调用 JobStore 的 schedulerStarted() 方法，顾名思义，它告诉 JobStore Scheduler 已经启动了，SchedulerStarted() 方法在 JobStoreSupport 类中被实现了。

JobStoreSupport 类使用 quartz.properties 文件(很快就会讨论到) 中适当的设置确定 Scheduler 实例是否参与到集群中。假如配置了集群，一个新的 ClusterManager 类的实例就被创建、初始化并启动。ClusterManager 是在 JobStoreSupport 类中的一个内嵌类。ClusterManager 继承了 java.lang.Thread，它会定期运行，并对 Scheduler 实例执行检入的功能。Scheduler 也要查看是否有任何一个别的集群节点失败了。检入操作发生的周期是基于一个配置属性的(很快会讲到)。

·侦测失败的 Scheduler 节点

当一个 Scheduler 实例执行检入的例程时，它会查看是否有其他的 Scheduler 实例在到达他们所预期的时间还未检入。这是通过检查 SCHEDULER_STATE 表并寻找 Scheduler 记录在 LAST_CHEDK_TIME 列的值是否早于 org.quartz.jobStore.clusterCheckinInterval(在下节中会讨论) 属性值。如果一个或多个节点还没有检入，那么运行中的 Scheduler 就假定它(们) 失败了。

> **时钟不同步的独立的机器上运行节点**
>
> 到现在你可以弄清了，如果你在不同的时钟不同步的机器上运行节点的话，你会得到异外的结果。这是因为节点用时间戳来通知其他实例它自己的最后检入时间。假如节点的时钟被设置为将来的时间，那么运行中的 Scheduler 也许再也意识不到那个结点已经宕掉了。另一方面，如果某个节点的时钟被设置为过去了，也许另一节点就会认定那个节点已宕掉并试图接过它的 Job 重运行。这两种情况下，都不是你想要的表现。当你在集群中正使用不同的机器(通常情况下)，要确定同步了时钟。具体参考本章后面的 "Quartz 集群 Cookbook" 一节了解如何做。

·从故障实例中恢复 Job

当一个 Sheduler 实例在执行某个 Job 时失败了，有可能由另一正常工作的 Scheduler 实例接过这个 Job 重新运行。要实现这

种行为，配置给 JobDetail 对象的 Job 可恢复属性必须设置为 true。

如果可恢复属性被设置为 false(默认时)，当某个 Scheduler 在运行一个 Job 时失败，它将不会重新运行；而是由另一个 Scheduler 实例在下一次触发时间触发，如果还会被触发的话。Scheduler 实例出现故障后多快能被侦测到决定于每个 Scheduler 的检入间隔。这会在下节中讨论。

第十一章. Quartz 集群（第三部分）

三. 配置 Quartz 使用集群

为 Quartz 配置集群环境的步骤比设置类似的 J2EE 集群环境容易的多：

1. 配置每个节点的 quartz.properties 文件。

2. 配置 JDBC JobStore。

3. 使用 Scheduler 信息(Job 和 Trigger) 装载数据库。

4. 启动每个 Quartz 节点。

·配置节点的 quartz.properties 文件

就像是运行 Quartz 在非集群环境中那样，每个 Quartz 应用需要一个 quartz.properties 文件。在第三章，“Hello, Quartz”中提到过，假如你不指定它，会使用默认的 quartz.properties 文件(存在于 quartz.jar 文件中)。最好是指定这个文件，因为你终究是需求修改一个或多个的设置项。

当使用 Quartz 的集群特性，你需要为每个节点修改 quartz.properties 文件。代码 11.1 显示了一个用于集群实例的 quartz.properties 文件的例子。属性将在之后讨论。

代码 11.1. 集群实例的 quartz.properties 文件示例

```
#============================================================================
#Configure Main Scheduler Properties
#============================================================================
org.quartz.scheduler.instanceName = TestScheduler1
org.quartz.scheduler.instanceId = instance_one
#============================================================================
#Configure ThreadPool
#============================================================================
org.quartz.threadPool.class = org.quartz.simpl.Simple ThreadPool
org.quartz.threadPool.threadCount = 5
org.quartz.threadPool.threadPriority = 5
#============================================================================
#Configure JobStore
#============================================================================
org.quartz.jobStore.misfireThreshold = 60000
org.quartz.jobStore.class = org.quartz.impl.jdbcjobstore.JobStoreTX
org.quartz.jobStore.driverDelegateClass =
org.quartz.impl.jdbcjobstore.MSSQLDelegate
org.quartz.jobStore.tablePrefix = QRTZ_
org.quartz.jobStore.dataSource = myDS

org.quartz.jobStore.isClustered = true
org.quartz.jobStore.clusterCheckinInterval = 20000
#============================================================================
#Non-Managed Configure Datasource
#============================================================================
org.quartz.dataSource.myDS.driver = net.sourceforge.jtds.jdbc.Driver
org.quartz.dataSource.myDS.URL = jdbc:jtds:sqlserver://localhost:1433/quartz
org.quartz.dataSource.myDS.user = admin
org.quartz.dataSource.myDS.password = admin
org.quartz.dataSource.myDS.maxConnections = 10
```

·配置主要的 Scheduler 属性

在这一节中有两个属性应该配置

·org.quartz.scheduler.instanceName

·org.quartz.scheduler.instanceId

这两属性用于多处，用在 JDBC JobStore 中和数据库来唯一标识实例。

> **集群时为实例 ID 使用 AUTO**
>
> AUTO 为专门为集群准备的。不幸的是，在某些早期的版本，1.4.5 版所用的机制任何情况下都不会清理旧的实例 ID。1.5.1 版中有一些插入式的实例 ID 生成器，其中一个是基于节点的 IP 地址来生成 ID；只要你在一台给定的机器上仅有一个 Quartz 集群节点时这个生成器工作的很好。集群时应当使用 AUTO，因为很多人把 Quartz 放在 EAR 中，分布部署在集群的应用服务器上。这时候，EAR 中必然只有一个 quartz.properties 文件，因此它在所有的节点上是相同的。假如实例 ID 是硬编码的，Quartz 集群将不能正常工作，因为所有的节点有着一样的 ID。AUTO 就是为解决这个问题的。
>
> 一些其他严重的集群问题在 Quartz 1.5.1 被引入。如果你需要对 Quartz 集群，你或许该避免这个版本。

·配置 JobStore 块

为使用 Quartz 的集群，你需要用到 JobStoreTX 或者 JobStoreCMT 作为 Scheduler 的 JobStore。第六章，“Job 存储和持久化” 详细说明了如何设置和使用所提供的这两个 JDBC JobStore。从代码 11.1 中，你能发现和第六章所显示的设置是一样的，还有两个附加的属性：

·org.quartz.jobStore.isClustered

·org.quartz.jobStore.clusterChedkinInterval

通过设置 org.quartz.jobStore.isClustered 属性为 true，你就告诉了 Scheduler 实例要它参与到一个集群当中。这一属性会贯穿于调度框架的始终，用于修改集群环境中操作的默认行为。

org.quartz.jobStore.clusterCheckinInterval 属性定义了Scheduler 实例检入到数据库中的频率(毫秒为单位)。Scheduler 检查是否其他的实例到了它们应当检入的时候未检入；这能指出一个失败的 Scheduler 实例，且当前 Scheduler 会以此来接管任何执行失败并可恢复的 Job。通过检入操作，Scheduler 也会更新自身的状态记录。

clusterChedkinInterval 越小，Scheduler 节点检查失败的 Scheduler 实例就越频繁。默认值是 15000 (即15 秒)。

·配置 JobStore 的 数据源

因为你必须用 JDBC JobStore (JobStoreTX 或 JobStoreCMT) 于集群中，你也需要配置一个不受管理的数据源(所谓 "不受管理的"，我们意思是说数据源不受应用服务器管理)。看代码 11.1 中给 JobStoreTX 设置不受管理的数据源的例子。参考第六章中列出的可用属性和它们的允许值。

·使用 Scheduler 信息装载数据库

就如所有的使用数据库作为 Job 存储的 Quartz 应用一样，你必须加载 Job 信息到数据库中。我们在第六章已谈论过数种完成这个的方法。

其中一种方式是写一个配置经由 JDBC JobStore 使用数据库的独立的 Quartz 应用，创建一个 Scheduler 实例，部署所有的 Job 和 Trigger 信息。你可以让应用能通过传递一个参数给命令行来从数据库清除 Job 信息。这种用法的问题在于维护数据库变得非常笨拙的。

另一方式是使用你指定的 RDBMS 所带的查询工具来手工加载信息。这使得更新和删除相当容易，但是这在首次加载数据时可能有

问题，除非你明确你正在做什么；我们强烈不建议这么做。

有人发现了使用在第十三章，“Quartz 和 Web 应用” 讨论的 Quartz Web 应用是一种很方便的做法。管理 Job 信息简单，甚至能由非技术性的资源来完成。看第十三章中关于 Quartz Wet 应用的内容，假如你选择这么做，也能够以类似的方式集成 Quartz 到你自己的 GUI 程序中来。

［译者 Unmi 注：对于最后一节的原文，感觉到一阵雾水洒过，仿佛说了个东西含含糊糊，不够明确，不具体。］

第十一章. Quartz 集群（第四部分）

四. 运行 Quartz 集群节点

在启动集群中的 Quartz 应用真的没什么差别。每个实例(或节点) 必须单独启动。启动时，实例连接到数据库，获取 Scheduler 信息，并开始部署 Job。

因为 Quartz 使用了一个随机的负载均衡算法，你将会看到 Job 以随机的行方式由不同的实例执行。没有固定的模式或预告定义的节点来执行特定的 Job。

下一节会讨论一些在处理集群环境中的 Quartz 较常见的问题和任务。

五. Quartz 集群 Cookbook

本节旨在为开发者便于解决 Quartz 集群的具体问题而提供了资源。

·指派 Job 给集群中特定的实例

当前，还不存在一个方法来指派(钉住) 一个 Job 到集群中特定的节点。假如你需要这种行为，你可以创建一个非集群的 Quartz 应用与集群中的节点并行运行，并且要使用独立的一套数据库表或单独的 JobInitializationPlugin 用到的 XML 文件。

不要让非集群的实例指向到集群所用的同一套数据库表。不然你会得到莫名其妙的结果。

·在集群中的每一个节点上运行 Job

正如前面所回答的，当前还没有一种方式能让某一个 Job 实例在集群中的每一个节点上都运行。最好的办法是使用一个非集群的实例与集群的每一个节点并行运行，并且要使用独立的一套数据库表或单独的 JobInitializationPlugin 和 RAMJobStore 用到的 XML 文件。

·在不同的机器上运行节点

Quartz 实际并不关心你是在相同的还是不同的机器上运行节点。当集群是放置在不同的机器上时，通常称之为水平集群。节点是跑在同一台机器是，称之为垂直集群。对于垂直集群，存在着单点故障的问题。这对高可用性的应用来说是个坏消息，因为一旦机器崩溃了，所有的节点也就被有效的终止了。

·使用时间同步服务

当你在是在不同的机器上运行 Quartz 集群时，时钟应当要同步，以免出现离奇且不可预知的行为。我们已经提及过，假如时钟没能够同步，Scheduler 实例将对其他节点的状态产生混乱。有几种简单的方法来保证时钟何持同步，而且也没有理由不这么做。

最简单的同步计算机时钟的方式是使用某一个 Internet 时间服务器(Internet Time Server ITS)。关于如何基于其中一个国际可接受标准来设置你的时钟的信息请看 http://tf.nist.gov/service/its.html。

·从集群获取正在执行的 Job 列表

当前，如果不直接进到数据库的话，还没有一个简单的方式来得到集群中所有正在执行的 Job 列表。如果你请求一个 Scheduler 实例，你将只能得到在那个实例上正运行 Job 的列表。你可以写一些访问数据库 JDBC 代码来从适当的表中获取信息。当然，这是用的 Quartz 之外的方法，但确是能解决问题的。另一个途是使用 Quartz 的 RMI 特性来依次连接到每一个节点，并从中查询到当前正在执行的 Job。

·让集群和非集群实例一起运行

没什么会阻止你在相同环境中使用集群的和非集群的 Quartz 应用。唯一要注意的是这两个环境不要混用在相同的数据库中(译者 Unmi 注：应用是数据库表)。意思是非集群环境不要使用与集群应用相同的一套数据库表；否则将得到希奇古怪的结果，集群和非集群的 Job 都会遇到问题。

假如你让一个非集群的 Quartz 应用与集群节点并行着运行，设法使用 JobInitializationPlugin(随之的 XML 文件) 和 RAMJobStore。这会让你的生活更轻松。

·在集群环境中使用全局监听器

你仍然可以使用 Job 和 Trigger 监听器在集群环境中。唯一能让你有所迷惑的是要努力理解哪个 Scheduler 实例将收到方法回调。

要记住这个最简单的方法是：Job 或 Trigger 是在哪个 Scheduler 实例上执行的，通知的就是这个 Scheduler 实例上的监听器。因为 Job 和 Trigger 只会在单个节点上执行，也就只会通知那个节点上的监听器。

第十二章. Quartz Cookbook (第一部分)

第十二章. Quartz Cookbook

本章的目的是为在构建 Quartz 应用时常遇到的情形提供一系列的例子和解决方案。本章也可作为对本书剩余部分的一个参考和补充。

一. 与 Scheduler 一同工作

本节提供了使用 Quartz Scheduler 管理功能的几个例子。

·创建和启动 Scheduler

你能通过几种方式来启动 Quartz Scheduler，但是最简单的方式是使用两种 SchedulerFactory 实现中的一个。特别的，org.quartz.impl.stdSchedulerFactory 使用很简单，要执行对 Scheduler 的所有设置工作只需要调用 getDefaultScheduler () 这一静态方法即可，如代码 12.1 所演示的那般。

代码 12.1. 启动默认的 Scheduler

```
1.  public void startScheduler() {
2.    Scheduler scheduler = null;
3.
4.    try {
5.      // Get a Scheduler instance from the Factory
6.      scheduler = StdSchedulerFactory.getDefaultScheduler();
7.
8.      // Start the scheduler
9.      scheduler.start();
10.     logger.info("Scheduler started at " + new Date());
11.
12.     // Schedule jobs and triggers
13.
14.   } catch (SchedulerException ex) {
15.     // deal with any exceptions
16.     logger.error(ex);
17.   }
18.
19. }
```

当你从这个工厂中取得了一个 Scheduler 的实例后，你就可以启动它，并往其中加入需要的 Job 和 Trigger。你既可在 Scheduler 启动之前，也可以在 Scheduler 启动之后加入 Job 和 Trigger。

Quartz 框架支持多个配置文件，这就允许你创建不同版本的 Scheduler。举个例子说，某个版本的配置文件设置了 Scheduler 作为单个实例使用 RAMJobStore，而同时，另一配置文件可配置 Cheduler 作为集群的一部分并使用某种 JDBCJobStore。

要指定一个配置文件而不用默认配置文件，你可用 StdSchedulerFactory 的 initialize() 方法并指定一个配置文件名作为参数。代码 12.2 描绘了这个例子。

代码 12.2. 使用一个不同 Quartz 配置文件来启动 Scheduler

```
1.  public static void main(String[] args) {
2.
3.    Scheduler scheduler = null;
4.
5.        {
```

```
    try {
      StdSchedulerFactory factory =
                new StdSchedulerFactory();
      factory.initialize("myquartz.properties");
      scheduler = factory.getScheduler();
      scheduler.start();
      logger.info("Scheduler started at " + new Date());

      // Schedule jobs and triggers

    } catch (SchedulerException ex) {
      // deal with any exceptions
      logger.error(ex);
    }
  }
```

> **装入 Job 到 Scheduler 中**
>
> 上面这些例子启动了 Scheduler 但是没有往其中加入任何 Job。你可以首先启动 Scheduler，接着加入你的 Job，或者你也可以选择先加入 Job 然后再启动 Scheduler。两种方式都能工作的很好。本章接下来的部分，我们会有一些采用了两种方式的例子。

·停止 Scheduler

Scheduler API 包含有两个版本的 shutdown() 方法。其中一个带有一个布尔型参数，另一个没有参数。这一布尔型参数告诉 Scheduler 是否等待正在执行的 Job 执行完成。

使用无参版的方法相当于是传递了 false 给另一个方法。假如你对停止当前正在执行的 Job 并不在意，那就调用这个：

```
scheduler.shudown();
```

否则，就用这个：

```
scheduler.shutdown(false);
```

另一方面，如果你希望在 Scheduler 停止之前完成正在执行的 Job，就传递 true 给 shudown() 方法：

```
scheduler.shutdown(true);
```

·暂停 Scheduler (Standby 模式)

要临时停止触发任何 Trigger，你可以调用 Scheduler 的 standby() 方法。Scheduler 和它的资源并不被销毁，并且 Scheduler 能在任何时候被重启。代码 12.3 展示了一个使用 standby() 方法的例子。

代码 12.3. 使 Scheduler 转入到 Standby 模式

```
public void runScheduler() {
  Scheduler scheduler = null;

  try {
    // Get a Scheduler instance from the factory
    scheduler = StdSchedulerFactory.getDefaultScheduler();

    // Start the scheduler
    scheduler.start();

    // Pause the scheduler for some reason
    scheduler.standby();
```

```
13.
14.       // Restart the scheduler
15.       scheduler.start();
16.
17.     } catch (SchedulerException ex) {
18.       // deal with any exceptions
19.       logger.error(ex);
20.     }
21.   }
```

当一个 Scheduler 被置为 standby 模式，已部署的 Trigger 将不被触发。而当 Scheduler 在使用了 start() 方法重新启动之后，所有的已被触发 Trigger 将依据 misfire 设置来决定处理。

第十二章. Quartz Cookbook (第二部分)

二. 与 Job 一同工作

本节为使用 Quartz 的 Job 提供了一些例子。

·创建一个新的 Job 类

创建一个新的 Job 类很简单。仅需创建一个类，让它实现 org.quartz.Job 接口即可。这个接口需要你实现 execute() 方法，它会在 Scheduler 决定 Job 要执行时被调用。

代码 12.4 演示了一个简单的 Job，它会为某个用户检查邮件服务器上是否有新的邮件。当 Scheduler 执行这个 Job 时，方法 execute() 被调用，然后其中的代码就会连接到邮件服务器并获取任何邮件消息。这一 Job 只简单的打印邮件是谁发的和邮件的主题。

代码 12.4. 一个检查邮件服务器上的邮件的 Quartz Job

```
1.  package org.cavaness.quartzbook.chapter12;
2.
3.  import java.security.NoSuchProviderException;
4.  import java.util.Properties;
5.
6.  import javax.mail.Folder;
7.  import javax.mail.Message;
8.  import javax.mail.MessagingException;
9.  import javax.mail.Session;
10. import javax.mail.Store;
11.
12. import org.quartz.Job;
13. import org.quartz.JobExecutionContext;
14. import org.quartz.JobExecutionException;
15.
16. public class CheckEmailJob implements Job {
17.
18.   String mailHost = "some.mail.host";
19.   String username = "aUsername";
20.   String password = "aPassword";
21.
22.   // Default Constructor
23.   public CheckEmailJob() {
24.     super();
25.   }
26.
27.   public void execute(JobExecutionContext context) throws JobExecutionException {
28.     checkMail();
29.   }
30.
31.   protected void checkMail() {
32.
33.     // Get session
34.     Session session = null;
35.
36.     try {
37.
38.       // Get system properties
39.       Properties props = System.getProperties();
40.
```

```
      session = Session.getDefaultInstance(props, null);
    // Get the store
    Store store = session.getStore("pop3");
    store.connect(mailHost, username, password);

    // Get folder
    Folder folder = store.getFolder("INBOX");
    folder.open(Folder.READ_ONLY);

    // Get directory
    Message message[] = folder.getMessages();
    int numOfMsgs = message.length;

    if (numOfMsgs > 0) {
      for (int i = 0, n = numOfMsgs; i < n; i++) {
        System.out.println("(" + i + "  of "  + numOfMsgs + "): "
            + message[i].getFrom()[0] + "\t"
            + message[i].getSubject());
      }
    } else {
      System.out.println("No Messages for user");
    }

    // Close connection
    folder.close(false);
    store.close();
   } catch (NoSuchProviderException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
   } catch (MessagingException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
   }
  }
  public static void main(String[] args) {
    CheckEmailJob job = new CheckEmailJob();
    job.checkMail();
  }
}
```

代码 12.4 中大部分内部是使用 JavaMail API 访问邮件服务器的。就实现一个新的 Quartz Job 类来说，只有很少的事情要做。究其本质，就是要实现 Job 接口和 execute() 方法，然后为被部署准备就绪。这在下一个例子中有所显示。

> **硬编码之于向 Job 传参**
>
> 在代码 12.4 中，邮件属性，如主机、用户名和密码是硬编码在 Job 类本身中。这称不上是个好主意。在本章后面部份，我们把 Job 修改为传参到 JobDataMap 中。

·部署 Quartz Job

如上个例子所演示的，要创建一个 Quartz Job 确实很直截的。幸运的是，通过 Scheduler 配置一个 Job 也不是很难。代码 12.5 展示的就是部署前面的 CheckEmailJob。

代码 12.5. 显示了如何部署 CheckEmailJob 的例子

```
package org.cavaness.quartzbook.chapter12;

import org.apache.commons.logging.Log;
       org.apache.commons.logging.LogFactory;
```

```
4.  import org.apache.commons.logging.LogFactory;
5.  import org.quartz.JobDetail;
6.  import org.quartz.Scheduler;
7.  import org.quartz.SchedulerException;
8.  import org.quartz.Trigger;
9.  import org.quartz.TriggerUtils;
10. import org.quartz.impl.StdSchedulerFactory;
11.
12. public class Listing_12_5 {
13.   static Log logger = LogFactory.getLog(Listing_12_5.class);
14.
15.   public static void main(String[] args) {
16.     Listing_12_5 example = new Listing_12_5();
17.     example.runScheduler();
18.   }
19.
20.   public void runScheduler() {
21.     Scheduler scheduler = null;
22.
23.     try {
24.       // Get a Scheduler instance from the Factory
25.       scheduler = StdSchedulerFactory.getDefaultScheduler();
26.
27.       // Start the scheduler
28.       scheduler.start();
29.
30.       // Create a JobDetail for the Job
31.       JobDetail jobDetail = new JobDetail("CheckEmailJob",
32.           Scheduler.DEFAULT_GROUP, CheckEmailJob.class);
33.
34.       // Create a trigger that fires every 1 hour
35.       Trigger trigger = TriggerUtils.makeHourlyTrigger();
36.
37.       trigger.setName("emailJobTrigger");
38.
39.       // Start the trigger firing from now
40.       // trigger.setStartTime(new Date());
41.
42.       // Associate the trigger with the job in the scheduler
43.       scheduler.scheduleJob(jobDetail, trigger);
44.     } catch (SchedulerException ex) {
45.       // deal with any exceptions
46.       logger.error(ex);
47.     }
48.   }
49.
50. }
```

代码 12.5 中的内容先是从 StdSchedulerFactory 获得 Scheduler，然后启动它。接着为 CheckEmailJob 创建一个 JobDetail 和一个立即开始、每小时触发的 Trigger。

·触发 Job 一次

org.quartz.TriggerUtils 类是很方便的，包含有许多有用的方法。其中一个最有用的方法是能部署一个只立即触发一次的 Trigger。代码 12.6 演示了如何仅仅触发 CheckEmailJob 一次。

代码 12.6. 为 CheckEmailJob 使用仅触发一次的 Trigger

```
1.  org.cavaness.quartzbook.chapter12;
```

```java
package org.cavaness.quartzbook.chapter12;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.quartz.JobDetail;
import org.quartz.Scheduler;
import org.quartz.SchedulerException;
import org.quartz.Trigger;
import org.quartz.TriggerUtils;
import org.quartz.impl.StdSchedulerFactory;

public class Listing_12_6 {
  static Log logger = LogFactory.getLog(Listing_12_6.class);

  public static void main(String[] args) {
    Listing_12_6 example = new Listing_12_6();
    example.runScheduler();
  }

  public void runScheduler() {
    Scheduler scheduler = null;

    try {
      // Get a Scheduler instance from the Factory
      scheduler = StdSchedulerFactory.getDefaultScheduler();

      // Start the scheduler
      scheduler.start();
      // Create a JobDetail for the Job
      JobDetail jobDetail = new JobDetail("CheckEmailJob",
          Scheduler.DEFAULT_GROUP, CheckEmailJob.class);

      // Create a trigger that fire-once only
      Trigger trigger = TriggerUtils.makeImmediateTrigger(0, 0);
      trigger.setName("emailJobTrigger");

      // Associate the trigger with the job in the scheduler
      scheduler.scheduleJob(jobDetail, trigger);

    } catch (SchedulerException ex) {
      // deal with any exceptions
      logger.error(ex);
    }
  }
}
```

代码 12.6 中使用了 TriggerUtils 类的静态方法 makeImmediateTrigger()，并传递给 repeatCount 和 repeatInterval 这两个参数的值都是 0，也就使得这个 Trigger 仅触发一次。

第十二章. Quartz Cookbook (第三部分)

·替换已部署的 Job

Quartz 提供了对已部署 Job 进行修改的灵活性。它是通过允许用修改后的 JobDetail 替换已有的 JobDetail 来支持这一特性的。为展未这一例子，让我们更新代码 12.4 中的 CheckEmailJob 类。代码 12.4 是硬编码了邮件属性值到 Job 类中的。更好的做法是传入那些属性，如此则可以随意的改变它们；那让我们改动 CheckEmailJob 来做到这一点。代码 12.7 显示的是那个 Job 的更新后的版本。

代码 12.7. 更新后的允许传入属性的 CheckEmailJob

```
1.  package org.cavaness.quartzbook.chapter12;
2.
3.  import java.util.Properties;
4.
5.  import javax.mail.Folder;
6.  import javax.mail.Message;
7.  import javax.mail.MessagingException;
8.  import javax.mail.NoSuchProviderException;
9.  import javax.mail.Session;
10. import javax.mail.Store;
11. import org.quartz.Job;
12. import org.quartz.JobDataMap;
13. import org.quartz.JobExecutionContext;
14. import org.quartz.JobExecutionException;
15.
16. public class CheckEmailJob implements Job {
17.   public static String HOST_KEY = "mailHost" ;
18.   public static String USERNAME_KEY = "username";
19.   public static String PASSWORD_KEY = "password";
20.
21.   String mailHost = "some.mail.host";
22.   String username = "aUsername";
23.   String password = "aPassword";
24.
25.   public CheckEmailJob() {
26.     super();
27.   }
28.   public void execute(JobExecutionContext context) throws JobExecutionException {
29.     loadMailProperties(context.getJobDetail().getJobDataMap());
30.     checkMail();
31.   }
32.
33.   protected void loadMailProperties(JobDataMap map) {
34.     if (map.getString(HOST_KEY) != null) {
35.       mailHost = map.getString(HOST_KEY);
36.     }
37.
38.     if (map.getString(USERNAME_KEY) != null) {
39.       username = map.getString(USERNAME_KEY);
40.     }
41.
42.     if (map.getString(PASSWORD_KEY) != null) {
43.       password = map.getString(PASSWORD_KEY);
44.     }
45.   }
46.   protected void checkMail() {
```

```
    // Get session
    Session session = null;

    try {
      // Get system properties
      Properties props = System.getProperties();

      session = Session.getDefaultInstance(props, null);

      // Get the store
      Store store = session.getStore("pop3");
      store.connect(mailHost, username, password);

      // Get folder
      Folder folder = store.getFolder("INBOX");
      folder.open(Folder.READ_ONLY);

      // Get directory
      Message message[] = folder.getMessages();
      int numOfMsgs = message.length;
      if (numOfMsgs > 0) {
        for (int i = 0, n = numOfMsgs; i < n; i++) {
          System.out.println("(" + i + " of " + numOfMsgs + "): "
              + message[i].getFrom()[0] + "\t"
              + message[i].getSubject());
        }
      } else {
        System.out.println("No Messages for user");
      }

      // Close connection
      folder.close(false);
      store.close();
    } catch (NoSuchProviderException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    } catch (MessagingException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    }
  }
  public static void main(String[] args) {
    CheckEmailJob job = new CheckEmailJob();
    job.checkMail();
  }
}
```

代码 12.7 的 CheckEmailJob 与 12.4 中版本主要的不同是 loadMailProperties() 方法。这个方法在 Job 首先执行的时候被调用，并检查 JobDataMap 看邮件属性是否有设置在这个 Map 中。假如有就用已设值，如果没有的话就使用 Job 类中的默认值。

代码 12.8 展示了属性值是如何被设置到 JobDataMap 中并传递给 Job 的。这些代码也显示了你怎么样才能用修改后的 JobDetail 实例替换已有实例来改变 Job 的。

代码 12.8. 显示如何更新一个已部署 Job 的例子

```
package org.cavaness.quartzbook.chapter12;

import org.apache.commons.logging.Log;
```

```
import org.apache.commons.logging.LogFactory;
import org.quartz.JobDetail;
import org.quartz.Scheduler;
import org.quartz.SchedulerException;
import org.quartz.Trigger;
import org.quartz.TriggerUtils;
import org.quartz.impl.StdSchedulerFactory;
 public class Listing_12_8 {
  static Log logger = LogFactory.getLog(Listing_12_8.class);

  public static void main(String[] args) {
    Listing_12_8 example = new Listing_12_8();
    example.runScheduler();
  }

  public void runScheduler() {
    Scheduler scheduler = null;

    try {
      // Get a Scheduler instance from the Factory
      scheduler = StdSchedulerFactory.getDefaultScheduler();

      // Start the scheduler
      scheduler.start();

      // Create a JobDetail for the Job
      JobDetail jobDetail = new JobDetail("CheckEmailJob",
          Scheduler.DEFAULT_GROUP, CheckEmailJob.class);

      // Set the properties used by the job
      jobDetail.getJobDataMap().put(CheckEmailJob.HOST_KEY, "host1");
      jobDetail.getJobDataMap().put(CheckEmailJob.USERNAME_KEY, "username");
      jobDetail.getJobDataMap().put(CheckEmailJob.PASSWORD_KEY, "password");

      // Create a trigger that fires at 11:30pm every day
      Trigger trigger = TriggerUtils.makeDailyTrigger(23, 30);
      trigger.setName("emailJobTrigger");

      // Associate the trigger with the job in the scheduler
      scheduler.scheduleJob(jobDetail, trigger);

      // Update the Job with a different mail host
      jobDetail.getJobDataMap().put(CheckEmailJob.HOST_KEY, "host2");
      scheduler.addJob(jobDetail, true);

    } catch (SchedulerException ex) {
      // deal with any exceptions
      logger.error(ex);
    }
  }
}
```

代码 12.8 中内容显示了两件事。其一，它向你展示了如何能通过 JobDataMap 向 Job 类传递邮件属性。其二，它描绘了你如何能使用 addJob() 方法去更新一个已部署 Job 的 JobDetail。addJob() 方法有一个布尔型的参数，用来告诉 Scheduler 是否用将要传入的 JobDetail 替换已部署的 JobDetail。Job 名和组必须与 Scheduler 中相匹配，这样才能用新的替换掉旧的。典型的做法是，你的代码会获取到已存在的 Job，然后修改它的 JobDataMap 中的内容，最后重新保存即可。

·更新已存在的 Trigger

你或许也需要更新某个 Job 更新已存在的 Trigger。你能用另一个来替换某个 Trigger，只要它是应用于同一个 Job。你可以通过使用 Scheduler 的 rescheduleJob() 方法来替换某个已有的 Trigger。

```
Trigger newTrigger = // Create a new Trigger

// Replace the old trigger with a new one
sched.rescheduleJob(jobName, Scheduler.DEFAULT_GROUP, newTrigger);
```

·列示出 Scheduler 中的所有 Job

假如你正为 Quartz 构造一个 GUI，你可能需要列出已注册到 Scheduler 的所有 Job。代码 12.9 呈现了这个需求的一个应用。

代码 12.9. 列示出 Scheduler 中的所有 Job 的例子

```
1.  package org.cavaness.quartzbook.chapter12;
2.
3.  import org.apache.commons.logging.Log;
4.  import org.apache.commons.logging.LogFactory;
5.  import org.quartz.JobDetail;
6.  import org.quartz.Scheduler;
7.  import org.quartz.SchedulerException;
8.  import org.quartz.Trigger;
9.  import org.quartz.TriggerUtils;
10. import org.quartz.impl.StdSchedulerFactory;
11.
12. public class Listing_12_9 {
13.   static Log logger = LogFactory.getLog(Listing_12_9.class);
14.
15.   public static void main(String[] args) {
16.     Listing_12_9 example = new Listing_12_9();
17.     example.runScheduler();
18.   }
19.
20.   public void runScheduler() {
21.     Scheduler scheduler = null;
22.
23.     try {
24.       // Get a Scheduler instance from the Factory
25.       scheduler = StdSchedulerFactory.getDefaultScheduler();
26.
27.       // Start the scheduler
28.       scheduler.start();
29.
30.       // Create a JobDetail for the Job
31.       JobDetail jobDetail = new JobDetail("CheckEmailJob",
32.           Scheduler.DEFAULT_GROUP, CheckEmailJob.class);
33.
34.       // Create a trigger that fires at 11:30pm every day
35.       Trigger trigger = TriggerUtils.makeDailyTrigger(23, 30);
36.       trigger.setName("emailJobTrigger");
37.
38.       // Associate the trigger with the job in the scheduler
39.       scheduler.scheduleJob(jobDetail, trigger);
40.
41.       String[] jobGroups = scheduler.getJobGroupNames();
42.       int numOfJobGroups = jobGroups.length;
43.
44.       for (int i = 0; i < numOfJobGroups; i++) {
45.         System.out.println("Group: "  + jobGroups[i]
```

```
            + " contains the following jobs");

        String[] jobsInGroup = scheduler.getJobNames(jobGroups[i]);
        int numOfJobsInGroup = jobsInGroup.length;

        for (int j = 0; j < numOfJobsInGroup; j++) {
          System.out.println(" - "  + jobsInGroup[j]);

        }
      }

    } catch (SchedulerException ex) {
      // deal with any exceptions
      logger.error(ex);
    }
  }
}
```

代码 12.9 中只注册了一个 Job，就是代码 12.7 中的 CheckEmailJob，且演示了如何循环 JobGroups 并罗列出每一个 group 下的所有 Job。在 GUI 中，罗列的结果可以呈现在列表框中或者是下拉列表中。

·列示出 Scheduler 中的所有 Trigger

你也能用类似于代码 12.9 中的方式罗列出所有 Trigger 来。下面的代码看起来十分相似，只是替换为 Trigger 而已。

```
String[] triggerGroups = sched.getTriggerGroupNames();
int numOfTriggerGroups = triggerGroups.length;
for (i = 0; i < numOfTriggerGroups; i++) {
  System.out.println("Group: "
    + triggerGroups[i]
    + " contains the following triggers");

  String[] triggersInGroup = sched.getTriggerNames(triggerGroups[i]);
  int numOfTriggersInGroup = triggersInGroup.length;
  for (j = 0; j < numOfTriggersInGroup; j++) {
    System.out.println("- " + triggersInGroup[j]);
  }
}
```

如果你需要罗列出单个 Job 的所有 Trigger，你可用 Scheduler 的 getTriggersOfJob() 方法。这个方法返回一个与此 Job 相关联的 Trigger[] 数组。

第十三章. Quartz 和 Web 应用（第一部分）

第十三章. Quartz 和 Web 应用

到目前为止，我们与 Quartz 框架的交互主要还是通过命令行。对于有些使用者，比如我的一个大学计算机科学老教授(它曾每天都告诉我说，"GUI 是给能力差的人用的!")，使用命令行让他们很乐意接受。当应用程序被开发完成后，它们常要移交给终端用户或支持团队。在命令行应用程序上层架设一个 GUI 前端会非常有帮助也是很增值的。本章记述如何在 Web 应用中用 Quartz 来使得部署和维护 Job 更轻松。

一. 在 Web 应用中使用 Quartz

至此，你已经看到过许多在 J2SE 环境中独立运行的 Quartz 的例子。在第十章，"J2EE 中使用 Quartz"，你也学到了 Quartz 良好的运作于 J2EE 环境中。但是，我们还没有向你介绍的是如何部署 Quartz 到一个 Java Web 应用(通常简写为 Web app)中。这就是本章唯一意图

你也许有几个理由想把 Quartz 集成到 Web 应用中。其中一些很显然的理由如下：

- 使用 GUI 界面部署和运行 Job
- 改善 Job 的管理和监控
- 使更易于多人部署 Job
- 能从你的 Web 应用内部来部署 Job

当然，在 Web 应用中使用 Quartz 主要原因还是允许通过 GUI 界面部署和维护 Job 更简单。第二个原因包括更好的管理已运行和已部署的 Job，还有当有故障时能更快得到通知。大体上，你想在其他任何软件程序之上放置一个 GUI 界面的理由也能通行于使用了 Quartz 的应用： 使更易于使用程序。

二. 集成 Quartz

幸运的是，有两方面使得很轻松集成 Quartz 到一个 Web 应用中。首先，Quartz 框架所需要的第三方库用起来相当简单。大多数所依赖的第三方库已经包含在任何 Java Web 应用中，尤其是像 Apache Struts 那样的由开源框架构建的组件。当部署 Quartz 到 Web 应用中时，Quartz 需要以下第三方库：

- Commons BeanUtil
- Commons Collections
- Commons Logging
- Commons Digester

假如你先前构建过 Java Web 应用，你应当见识过以上列出的所有组件。还有一些别的 JAR 包也可能是必须的，这要依赖于你对 Quartz 确切的部署。例如，假如 Quartz 要存储 Job 信息到数据库中，那样标准的 JDBC API 库(jdbc2_0-stdext.jar) 就是必须的，一起也可能要用到 Java 事物 API(jta.jar)。

你可能也需要一些可选的库，这依赖于你的总体需求。例如，如果你的应用需要发送 e-mail，你就需要 activation 和 JavaMail 库。但是这与你是否部署 Quartz 在一个 Web 应用还是作为一个独立的程序都是一样考虑的。

Web 应用的结构

经历了过去的几年之后，Servlet 和 JSP 规范已得到改善，并允许更好的在不同的工具提供商之间移植。这在 Java 开发者社区中有着一种安定的效果，也让 Web 开发者把精力聚焦在 "真正的" 业务需求，而不用太关注应用的部署和运行的细节。

·安装 Quartz 库

像在任何其他的 Java Web 应用中一样，Servlet 规范指示了所有的 JAR 文件(第三方或别的) 必须放置于 WEB-INF/lib 中。因此，其中第一个步骤就是把 quartz.ar 和他所依赖的 JAR 文件放到 WEB-INF/lib 目录中。

> **留意 JAR 包版本和位置**
>
> 你必须关心的不仅是要把哪些 JAR 文件放到 Web 应用中，还要注意使用什么版本的 JAR 包和确切的放置目录。随着开发社区的不断成熟，更多的持续集成发生在各个独立的项目之间。因此某一个项目依赖于另一个项目过期版本的事情也时有发生。在升级到库的新版本前请确保检查依赖性。
>
> 另一个请记住的事情是要把库安放到哪里，这是非常重要的(有时也很令人困惑的)。所幸的是，Web 容器提供商开始遵循着相近的规范，开发人员也变得更有学识。对于 Web 应用，你几乎总是要安装一些第三方包(因你的应用需求而定) 到 WEB-INF/lib 目录中。与 XML 解析器和像 Sun Java 安全套接字扩展(JSSE) 那样的加密包的问题仍然会冒出来，但这些问题随着商业和开源提供商更新他们的发行版时会变得更少。

·选择一个 Web 应用框架

这完全由你来选择用哪个 Web 应用框架来与 Quartz 集成。如此多框架可选择，以致于着实无法明确的选用哪个。要说某个框架优于另一个那是很主观的，因为太多的关乎于你的需求和技术的东西。然而，有几个 Web 框架已被一段时间所证明了的。一个例子就是 Apache Struts 框架(以前是以 Jakarta Struts 著名)。因本节的用意，我们将使用 Struts 框架来演示如何与 Quartz 集成。

第十三章. Quartz 和 Web 应用（第二部分）

三. 在 Struts 框架中使用 Quartz

第一步就是要下载 Apache Struts 并创建好你的 Web 应用的目录结构。Struts 框架可从 Apache Struts 站点 http://struts.apache.org 上找到。也很欢迎你直接抓取到源代码来编译它，尽管你能够及时下载到最新版本的二进制版。

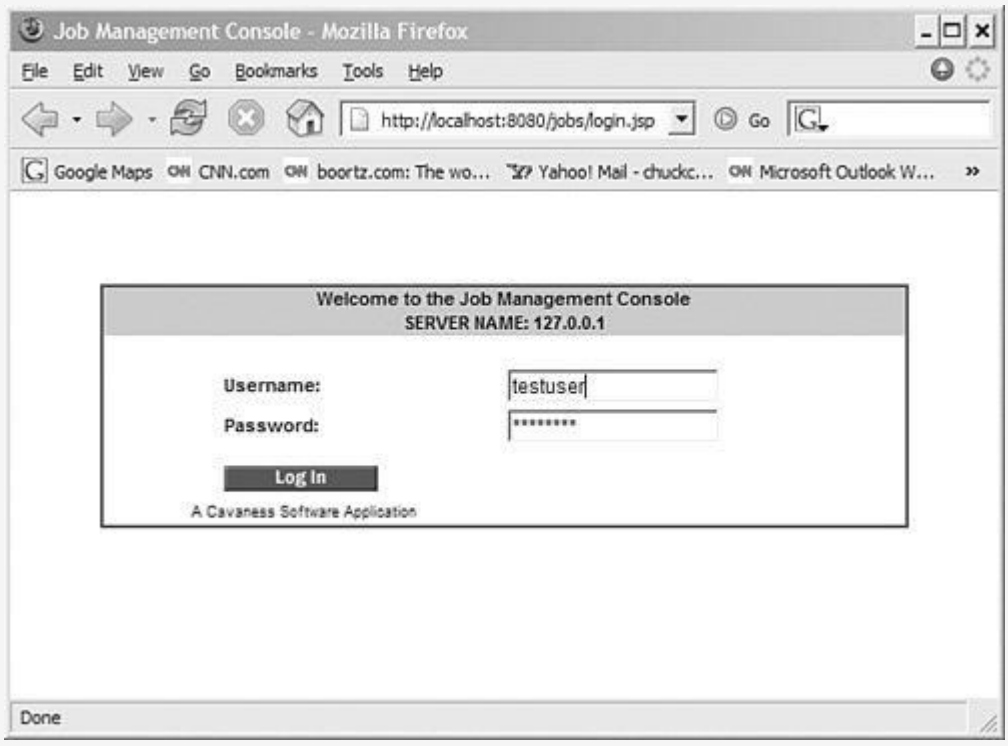
因为 Quartz 不直接依赖于 Struts 框架，所以你也就不必为该使用哪一版本的 Struts 而忧心。只管下载到当前的最新版本就是了。不过，你应该认识到，Struts 和 Quartz 框架共同依赖了一些第三方的包。实际上，在前面列出的 Quartz 所需要的库也是 Struts 框架所必须的。就是要留心混在一起不同的版本，如在最后一节的警告所注明的。

·创建你的 Web 应用目录结构

下载到了 Struts 之后，你就可以创建你的目录结构并引入必须的文件。作为例子，我们将创建一个虚构的 Web 应用，叫做 Job 管理控制台。因为这仅是一个假想的应用，我们并不打算完整的构建它。相应的，我们用它来来解释与 Quartz 集成的几个关键点，剩余的部分留给你的探索了。现在，假定老板给了我们一个在 Job 调度框架(这个吗，当然是基于 Quartz 的) 之上构建 GUI 的任务。图 13.1 和 13.2 显示了 Job 管理控制台应用的登录和主界面。

图 13.1. Job 管理控制台应用的登录界面

[看全尺寸图]



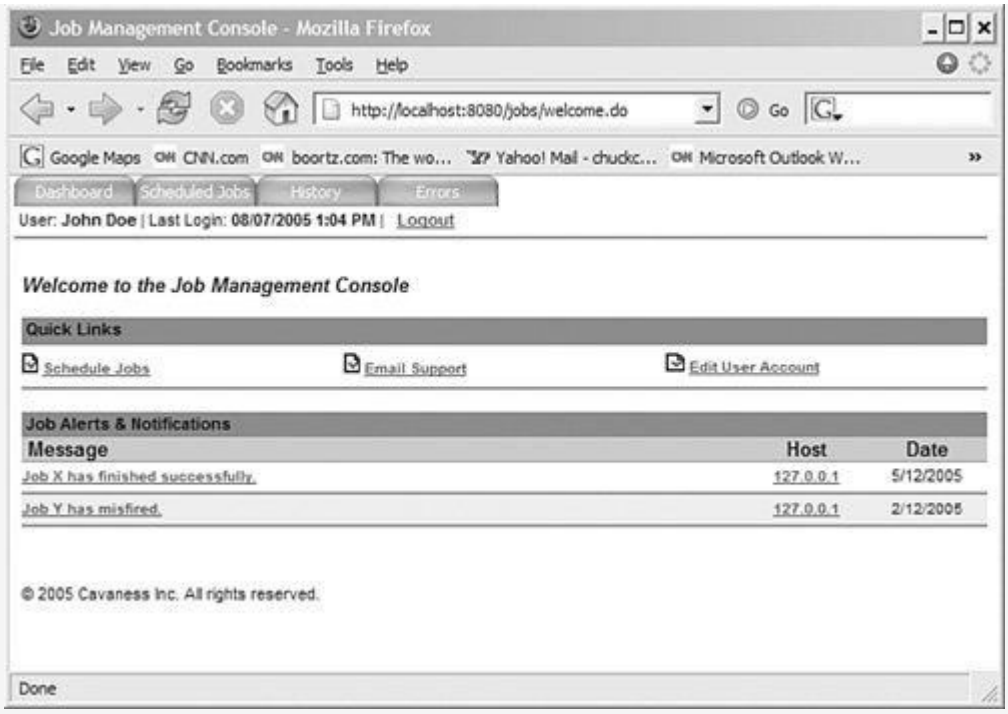
当用户按下 Login 按钮，应用就会带领用户到操控板界面，见图 13.2。

图 13.2 显示了所有用户登录后被带领到的控制台操控板。

图 13.2 中显示了操控板页面相当简单但足以应达到我们的目的。

图 13.2. Job 管理控制台的操控板是用户的主界面

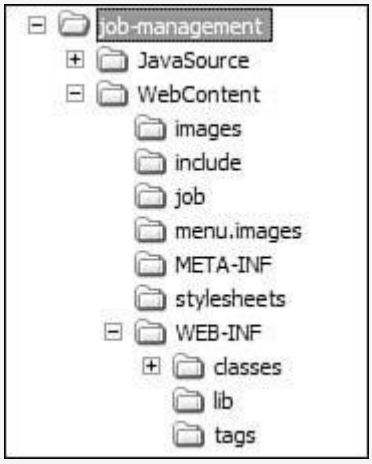
[看全尺寸图]



·创建 Job 管理控制台项目结构

Job 管理控制台项目结构在 Java Web 应用中是非常标准的，并且大部分 Java Web 开发者能清楚以下所示的多数目录的功用。WEB-INF 下的标签目录将存放 .tld 文件，应用据此来引用自定义标签库。Struts 框架提供了几个标签库，能用来简化 JSP 开发；.tld 目录中存有那些标签的标述文件。图 13.3 显示了这一应用的项目结构。

图 13.3. Job 管理控制台项目的结构



·在 Web 应用中初始化 Quartz

当 Quartz 应用于命令行中时，是用一个 Java 类来创建 SchedulerFactory 并实例化 Scheduler。由于 Quartz 现在将要运行在 Web 应用中，你就不容易访问到 main() 方法了，因为应用是由容器来启动的，而 main() 是埋在了代码中，可能甚至是实际不可执行的。所幸的是，解决办法很简单：你所需要做的就是确保当容器首先启动这个 Web 应用时，你有一些代码扮演着以工厂创建实例的逻辑。这就是说，当容器首次加载 Web 应用时，你需要创建一个 SchedulerFactory 并启动这个 Scheduler。

是 start() 还是不 start()

依赖于你的需求，你也许希望 Scheduler 在 Web 应用首次加载时立即启动。然而，或许你开始只是要 Scheduler 准备就绪， 待到某个其他的动作发生时才启动它。例如，也许在你的 Job 管理控制台应用中，Scheduler 直到用户进入到操控板页面并按下开始按钮才启动。如果是这种情况，可以获取到工厂，但 Scheduler 直到你准备好了才启动。

第十三章. Quartz 和 Web 应用（第三部分）

四. QuartzInitializerServlet 可谓救命草

Quartz 框架包括一个叫做 org.quartz.ee.servlet.QuartzInitializerServlet 的 Java 类，它继承自标准的 HttpServlet。你可应用这个 servlet 于你的 Web 应用中，它将会创建一个 StdSchedulerFactory 实例并在你的程序后续中一直可用。通常的，它就是做了命令行版本的 Quartz 程序的 main() 方法所做的事性。

> **QuartzInitializerServlet 在 Quartz 1.5 中有所改变**
>
> 在 Quartz 的 1.5 发布版中，QuartzInitializerServlet 被修改为会存储 StdSchedulerFactory 实例到 Web 应用的 ServletContext 中。这就允许你的程序在任何地方都能访问到 Scheduler 实例，只要获取到了 HttpServletRequest 或 HttpSession 对象，调用工厂的 getScheduler() 就访问到了 Scheduler 实例。
>
> 还新增了一个 start-scheduler-on-load 的 Servlet 初始化参数。这一参数指定 Scheduler 是否随 QuartzInitializerServlet 启动或是别处启动。假如未设置时默认为 True，Sheduler 将随 QuartzInitializerServlet 起来。否则，你的应用将不得不自己去获得 Scheduler 实例然后调用 start() 方法。

当容器加载 QuartzInitializerServlet，该 Servlet 的 init() 方法将被调用。这个 Servlet 读取几个初始化参数，创建 StdSchedulerFactory 类的实例，并使用指定(或默认) 的 Quartz 属性文件来初始化 Scheduler。

工厂创建之后，init() 方法就会决定 Scheduler 是否立即启动，或是让程序来决定何时启动它。代码 3.1 列出了 QuartzInitializerServlet 的 init() 方法。

代码 13.1. QuartzIniializerServlet 类的 init() 方法

```
1.  public void init(ServletConfig cfg) throws ServletException {
2.   super.init(cfg);
3.
4.   log("Quartz Initializer Servlet loaded, initializing Scheduler...");
5.
6.   StdSchedulerFactory factory;
7.   try {
8.
9.     String configFile = cfg.getInitParameter("config-file");
10.    String shutdownPref = cfg.getInitParameter("shutdown-on-unload");
11.
12.    if (shutdownPref != null)
13.
14.      performShutdown = Boolean.valueOf(shutdownPref).booleanValue();
15.
16.    // get Properties
17.    if (configFile != null) {
18.      factory = new StdSchedulerFactory(configFile);
19.    } else {
20.      factory = new StdSchedulerFactory();
21.    }
22.
23.    // Should the Scheduler being started now or later
24.    String startOnLoad =
25.      cfg.getInitParameter("start-scheduler-on-load");
26.
27.    /*
28.     * If the "start-scheduler-on-load" init-parameter is not specified,
29.     * the
30.     * scheduler will be started. This is to maintain backwards
```

```
     * compatability.
     */

     if (startOnLoad == null ||
      (Boolean.valueOf(startOnLoad).booleanValue())) {
      // Start now
      scheduler = factory.getScheduler();
      scheduler.start();
      log("Scheduler has been started...");
     } else {
      log("Scheduler has not been started. Use scheduler.start()");
     }

     log(
      "Storing the Quartz Scheduler Factory in the servlet context at key: " +
   QUARTZ_FACTORY_KEY);
     cfg.getServletContext().setAttribute(QUARTZ_FACTORY_KEY, factory);

   } catch (Exception e) {
     log("Quartz Scheduler failed to initialize: " +
  e.toString());
     throw new ServletException(e);
   }
  }
```

QuartzInitializerServlet 是 Quartz JAR 文件的一部分。只要你 Web 应用的 WEB-INF/lib 中有了 quartz.jar，这个 Servlet 在你的应用中就是可用的了。

·配置 Web 部署描述文件

Java Servlet 规范规定了每个 Web 应用都必须包含一个 Web 部署描述文件。部署文件(web.xml) 中包含了以下类型的信息：

·初始化参数

·Session 配置

·Servlet/JSP 定义

·Servlet/JSP 映射

·MIME 类型映射

·欢迎文件

·错误页面

·安全性

因为 QuartzInitializerServlet 是一个 Java Servlet，它必须配置到部署描述文件中让容器来加载它。代码 13.2 描绘了怎样在 web.xml 文件中配置 QuartzInitializerServlet。

代码 13.2. QuartzInitializerServlet 需要对 web.xml 文件作相应的修改

```
<web-app>
<servlet>
 <servlet-name>QuartzInitializer</servlet-name>
 <display-name>Quartz Initializer Servlet</display-name>
```

```
5.
6.    <servlet-class>
7.      org.quartz.ee.servlet.QuartzInitializerServlet
8.    </servlet-class>
9.
10.   <load-on-startup>1</load-on-startup>
11.
12.   <init-param>
13.     <param-name>config-file</param-name>
14.     <param-value>/some/path/my_quartz.properties</param-value>
15.   </init-param>
16.
17.   <init-param>
18.     <param-name>shutdown-on-unload</param-name>
19.     <param-value>true</param-value>
20.   </init-param>
21.
22.   <init-param>
23.     <param-name>start-scheduler-on-load</param-name>
24.     <param-value>true</param-value>
25.   </init-param>
26.
27.  </servlet>
28.
29.  <! other web.xml items here >
30.
31.  </web-app>
```

QuartzInitializerServlet 支持三个 Quartz 规范的初始化参数

初始化参数 config-file

config-file 参数用来指定 Quartz 属性文件的路径和文件名。StdSchedulerFactory 使用这个文件配置 Scheduler 实例。这个参数是可选的；假如未指定，就会用默认的 quartz.properties 文件。使用此参数最简单的方式(假定你想提供自己的属性文件)是把你的属性文件放到 WEB-INF/classes 目录中，然后如下方式指定 init-param:

```
1.  <init-param>
2.   <param-name>config-file</param-name>
3.    <param-value>/my_quartz.properties</param-value>
4.  </init-param>
```

初始化参数 shutdown-on-unload

shutdown-on-unload 参数用于在容器卸载本 Servlet 时引起 scheduler.shutdown() 方法的调用。一个容器在关闭，或某种条件时，在一个热部署环境中重新加载时就会卸载这个 Servlet。这个参数是可选的，默认值是 true。

初始化参数 start-scheduler-on-load

start-scheduler-on-load 参数用于告诉 Servlet 调用 Scheduler 实例的 start() 方法。假如它没有启动，Scheduler 就需要由程序在晚些时候来启动，在 start() 未被调用之前是不会有 Job 运行的。这一参数是可选的，如果未指定时默认为 true。这个参数是在 1.5 发行版中加载来，不存在于早期版本中。

·访问 SchedulerFactory 和 Scheduler

自 Quartz 1.5 开始，QuartzInitializerservlet 将自动将 StdSchedulerFactory 实例以某一预定义键存放于 ServletContext 中。

早期 Quartz 版本中的 QuartzInitializerServlet

> QuartzInitializerServlet 在早期版本的框架中就出现了。在以往版本中，StdSchedulerFactory 不会存放到 ServletContext 中。Sheduler 在 Servlet 的 init() 方法中初始化后即启动。欲从你的代码中获取 Scheduler 实例，你必须使用 StdSchedulerFactory 的某个 get 方法访问该 Servlet 创建的 Scheduler。从 ServletContext 中访问 Scheduler 的改变是在 1.5 版中加进来的。

你可在代码 13.1 的 init() 方法结束部分看出来。一旦工厂被存入 ServletContext，有多种方式可以获取访问到它。最简单的方式，尤其是你在用 Struts 框架的时候，是使用 request 对象。代码 13.3 展示了一个 Struts Action 类，叫做 StartSchedulerAction。当这个 Action 被调用(假定是通过 /startscheduler.do 这样的 URL 来访问)，SchedulerFactory 就能获取取，接着可以调用 getScheduler() 方法。

代码 13.3. SchedulerFactory 和 Scheduler 能简单的被访问

```
1.  import javax.servlet.ServletContext;
2.  import javax.servlet.http.HttpServletRequest;
3.  import javax.servlet.http.HttpServletResponse;
4.
5.  import org.apache.struts.action.Action;
6.  import org.apache.struts.action.ActionForm;
7.  import org.apache.struts.action.ActionForward;
8.  import org.apache.struts.action.ActionMapping;
9.  import org.cavaness.jobconsole.web.QuartzFactoryServlet;
10. import org.cavaness.jobconsole.web.WebConstants;
11. import org.quartz.Scheduler;
12. import org.quartz.impl.StdSchedulerFactory;
13.
14. public class StartSchedulerAction extends Action {
15.     public ActionForward execute(ActionMapping mapping,
16.     ActionForm form, HttpServletRequest request,
17.     HttpServletResponse response) throws Exception {
18.
19.          // Retrieve the ServletContext
20.          ServletContext ctx =
21.               request.getSession().getServletContext();
22.
23.          // Retrieve the factory from the ServletContext
24.          StdSchedulerFactory factory =
25.                (StdSchedulerFactory)
26.               ctx.getAttribute(
27.                  QuartzFactoryServlet.QUARTZ_FACTORY_KEY);
28.
29.          // Retrieve the scheduler from the factory
30.          Scheduler scheduler = factory.getScheduler();
31.
32.          // Start the scheduler
33.          scheduler.start();
34.
35.
36.          // Forward to success page
37.          return mapping.findForward(WebConstants.SUCCESS);
38.     }
39. }
```

当从 SchedulerFactory 获取到 Scheduler 之后，你就可以按正常方式般调用 Scheduler 实例的方法。在代码 13.3 中，Scheduler 是使用 start() 方法来启动的，这是你所习惯的。

把 StdSchedulerFactory 存于 ServletContext 中的优点是避免了你重复创建它。实际上，你可以使得访问 Scheduler 甚至更简单，做法是把所有访问 ServletContext 和创建 Scheduler 的逻辑放到一个工具类中。代码 13.4 展示了这样一个名为 ActionUtil 的类，它简化了获得一个 Scheduler 对象的引用。

代码 13.4. ActionUtil 类方便了访问 Scheduler

```
1.  import javax.servlet.ServletContext;
2.  import javax.servlet.http.HttpServletRequest;
3.
4.  import org.quartz.Scheduler;
5.  import org.quartz.SchedulerException;
6.  import org.quartz.impl.StdSchedulerFactory;
7.
8.  public class ActionUtil {
9.
10.     public static Scheduler getScheduler(HttpServletRequest request)
11.          throws SchedulerException {
12.
13.          ServletContext ctx =
14.               request.getSession().getServletContext();
15.
16.          StdSchedulerFactory factory =
17.                (StdSchedulerFactory) ctx.getAttribute(
18.                QuartzFactoryServlet.QUARTZ_FACTORY_KEY);
19.
20.          return factory.getScheduler();
21.     }
22. }
```

你能使用 ActionUtil.getScheduler() 方法很方便的获取到 Scheduler 实例。代码 13.3 中接下来的片断就是类 StartSchedulerAction 如何使用 Action.getScheduler() 方法了：

```
1.  public class StartSchedulerAction extends Action {
2.      public ActionForward execute(ActionMapping mapping,
3.      ActionForm form, HttpServletRequest request,
4.      HttpServletResponse response) throws Exception {
5.
6.          // Retrieve the scheduler from the factory
7.          Scheduler scheduler = ActionUtil.getScheduler();
8.
9.          // Start the scheduler
10.         scheduler.start();
11.
12.         // Forward to success page
13.         return mapping.findForward(WebConstants.SUCCESS);
14.
15.     }
16. }
```

代码 13.4 中的 ActionUtil 并非 Quartz 框架的一部份，但它或许会被加入到将来的版本中。你可在你的应用中需要的时候加入相同的东西。

第十三章. Quartz 和 Web 应用（第四部分）

五. 使用 ServletContextListener

很值得一提的是你可以配置和集成 Quartz 到 Web 应用的另一种方式。从 2.3 版本的 Servlet API 开始，你能创建监听器，由容器在其生命周期中的某个特定时间回调。其中的一个监听器接口叫做 java.servlet.ServletContextListener，它包括有两个方法：

```
public void contextInitialized(ServletContextEvent sce);
public void contextDestroyed(ServeltContextEvent sce);
```

容器会在启动和关闭的时候相应的调用这两个方法。这就可以在 contextInitialized() 方法中初始化 Quartz Scheduler，并通过 contextDestroyed() 方法关闭它。代码 13.5 描述了这种用法：

代码 13.5. ServletContextListener 也能被用于初始化 Quartz

```
1.  import javax.servlet.ServletContext;
2.  import javax.servlet.ServletContextEvent;
3.  import javax.servlet.ServletContextListener;
4.
5.  import org.apache.commons.logging.Log;
6.  import org.apache.commons.logging.LogFactory;
7.  import org.quartz.SchedulerException;
8.  import org.quartz.impl.StdSchedulerFactory;
9.
10. public class QuartzServletContextListener implements ServletContextListener {
11.     private static Log logger = LogFactory
12.               .getLog(QuartzServletContextListener.class);
13.
14.     public static final String QUARTZ_FACTORY_KEY =
15.          "org.quartz.impl.StdSchedulerFactory.KEY";
16.
17.     private ServletContext ctx = null;
18.
19.     private StdSchedulerFactory factory = null;
20.
21.     /**
22.      * Called when the container is shutting down.
23.      */
24.     public void contextDestroyed(ServletContextEvent sce) {
25.          try {
26.               factory.getDefaultScheduler().shutdown();
27.          } catch (SchedulerException ex) {
28.               logger.error("Error stopping Quartz", ex);
29.          }
30.
31.     }
32.
33.     /**
34.      * Called when the container is first started.
35.      */
36.     public void contextInitialized(ServletContextEvent sce) {
37.
38.          ctx = sce.getServletContext();
39.
40.          try {
41.
```

```
                factory = new StdSchedulerFactory();

                // Start the scheduler now

                factory.getScheduler().start();

                logger.info("Storing QuartzScheduler Factory at"
                          + QUARTZ_FACTORY_KEY);

                ctx.setAttribute(QUARTZ_FACTORY_KEY, factory);

          } catch (Exception ex) {
                logger.error("Quartz failed to initialize", ex);
           }
      }
}
```

正如我们在 QuartzInitializerServlet 中所做的，我们需要为这个监听器加入一些配置信息到 Web 部署描术文件(web.xml)中。针对我们的监听器，我们需要加一个 <listener> 元素到部署描述中。如下片断中显示：

```
<web-app>
 <listener>
   <listener-class>
  org.cavaness.jobconsole.web.QuartzServletContextListener
   </listener-class>
 </listener>
 <!--Other deployment descriptor info not shown -->
</web-app>
```

·新的 QuartzInitializerListener 已加入到 Quartz 中

ServletContextListener 为顺应用户社区而来的呼声被加入到 Quartz 框架中。13.5 中的代码应当认为是一个例子，实际应用时你需要开发你自己的监听器。

第十三章. Quartz 和 Web 应用（第五部分）

六. 介绍 Quartz Web 程序

早期的 Quartz 框架开发者意识到一个 GUI 对于某类用户群体是必需的。几年前，一个 Web 应用被创立，它可用于管理 Quartz 框架。虽说是历经了几年有相当投入的开发，但不得不说的，总是时断时续的。

近来出现有更多的要求对这个应用的更新与支持，因而又重新吸引了新的开发者自愿的工作并使之保持不断更新。这个应用就是知名的 Quartz Web 程序。(译者注：实际上这个应用程序基本驻步不前，当前版本还是 RC-1 2004-06-26 22:00 的，仅作参考来应用)

·Quartz Web 程序的截屏

Quartz Web 程序主界面的左上方展示了它所拥有的特征列表(看 图 13.4)。

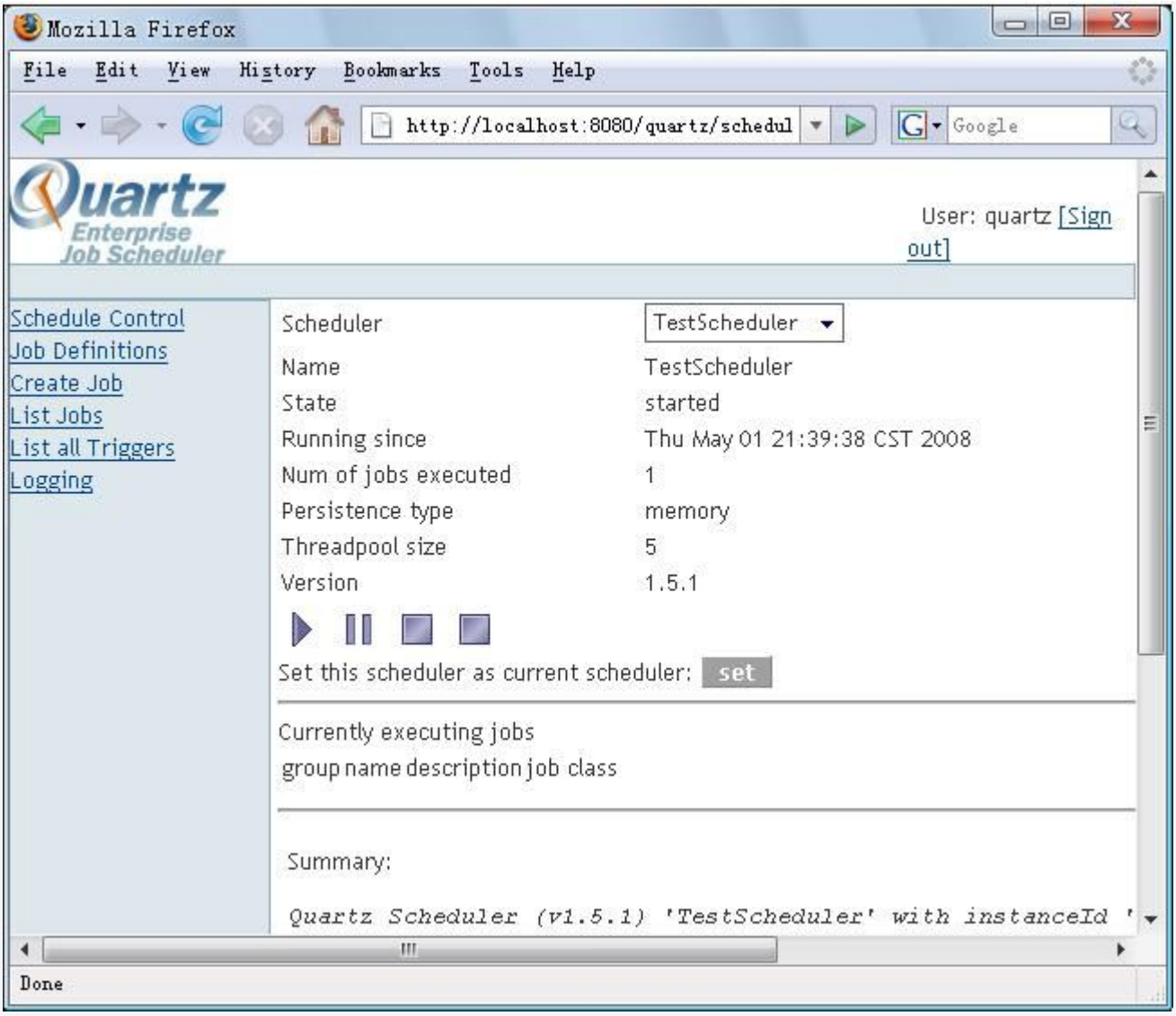
图 13.4. Quartz Web 程序主界面

[看全尺寸图]

图 13.5 显示的是 Scheduler 控制界面，允许你启动、停止和暂停 Scheduler。

图 13.5. Scheduler 控制界面包含 Scheduler 可管理的特点

[看全尺寸图]



·下载并构建 Quartz Web 程序

Quartz Web 程序是作为 OpenSymphony 上 Quartz 源程序库的一部分。它曾经是放在 SourceForge 上的，但最近已迁移到新家了(译者注：在 SourceForge 仍可下载到)。旧的站点是 http://sourceforge.net/projects/quartz；新的站点(包括 Quartz Web 程序) 可在 http://www.opensymphony.com/quartz/cvs.action 上找到。

当前，还没有 Web 程序的二进制版，而且标准的 Quartz 下载中也不包括 Web 程序，尽管程序的维护人员说过，他们会在 2006 前放出一个构建版。就现在来说，你需要下载到 Quartz 源代码树并使用 quartz/webapp 目录下的 Ant 构建文件来构建这个 Web 程序。

为了从 CVS 上下载 Quartz，你需要有一个 java.net 上的帐号，因为不允许匿名访问。在命令行下，进到你希望下载源代码树的目录中，输入：

```
cvs -d :pserver:[username]@cvs.dev.java.net:/cvs login
cvs -d :pserver:[username]@cvs.dev.java.net:/cvs checkout quartz
cvs -d :pserver:[username]@cvs.dev.java.net:/cvs checkout opensymphony
```

> **Quartz 需要 OpenSymphony 源码**
>
> OpenSymphony 的项目一直实践着持续集成。意思是说其上的项目紧紧的但支持性的依赖于另一项目。要成功构建 Quartz 还必须下载到 OpenSymphony 源码。使用前面的 checkout 命令，并把 quartz 替换为 opensymphony。你应该在与执行 checkout Quartz 相同的位轩处执行该命令。

你下载到了 Quartz 和 OpenSymphony 源码之后，你就可以在 quartz 目录中构建 Quartz 了，而且你可在 quartz 下的 webapp 目录中构建 Web 程序。改变目录到 webapp 并输入 ant。

·安装 Quartz Web 程序

当你成功构建了 Web 程序，就会在 webapp/dist 目录中创建一个 quartz.war 文件。部署这个 Web 包(WAR) 到你所选的容器中。

> **并非所有的容器设计成一样的**
>
> 我们应当指出的是，尽管各种 Servlet 和 JSP 规范被设计来使得程序从一个容器移动另一个容器更简单，但提供商也有不足之处。在最小限度内，我们能说每一个提供商对规范的解翻译会有差异。
>
> 一些 Quartz 用户报告了关于以 WAR 文件形式部署 Quartz Web 程序到 WebLogic 8.1 和特定版本的 WebSphere 的问题。假如你是用的 Tomcat，你一般不会有很多问题，但是你在用其他容器时或许会面临一些挑战。如果你遇到一些麻烦，试着以散装格式(exploded format) 替换WAR 来部署应用。

·配置和运行 Quartz Web 程序

现在，WAR 文件已经部署好了，到你的 Web 浏览器中引燃它吧。对于 Tomcat，默认的 URL 是 http://localhost:8080/quartz/。在首页面，你会有选项来管理 Scheduler、Job、Trigger 和 Logging。

在首页面上所有的菜单选项都是安全的。在最新版本中加入了一个安全过滤器用以模仿容器管理的安全性。这意味着，你可以调用 HttpServletRequest.getRemoteUser() 和 HttpRequest.isUserInRole() 方法，就像是在容器管理的安全性所做的那样。用户被定义在 SecurityFilter-config.xml 文件中。默认时，它包含一个用户，quartz，密码也是 quartz。你可以增加你的用户到 SecurityFilter-config.xml 文件中，或简单的继承 org.securityfilter.realm.SecurityRealmInterface 来提供一个安全域以适应你的需求。

以选择 Scheduler Control 菜单项开始，当提示时，以用户 quartz 登录。Quartz Web 程序预定义了一个测试的 Scheduler，这也是你在本页唯一可选项。深入到 quartz\webapp\config\resources 看看 Scheduler 是如何配置的。那儿你将找到目前熟悉的 quartz.properties 文件。在 Scheduler 控制页上下拉列表中的名字是 quartz.properties 的 org.quartz.scheduler.instanceName 属性。

接着点击 Job Definifions 菜单。那里你会看到三个 Job：NativeJob，NoJob 和 SendMailJob。假如你看遍了 \webapp\config\resources 目录，你将已经猜到了 Job 是定义在哪的：JobDefinitions.xml。这是在 Quartz Web 程序启动时由 ContextLoaderServlet 加载的。你可以编辑这个 JobDefinitions.xml 文件，或通过改变 web.xml 文件中 ContextLoaderServlet <init-param> 参数来提供你自己的 Job 定义文件。

如果你准备要运行一个 Job，你可以选择 Job 定义页面的 NativeJob 左边的 Create Job 链接。你要挑选一个本地的，可很清楚的看出你的 Job 已经运行的程序。假如你是在 Windows 平台下，你可以用 Notepad.exe。首先给这个 Job 一个名字如 Native Job Test。这个名字必须是在 Scheduler 中唯一的。接着，在命令(command) 框中输入 notepad 然后保存(Save)。你现在应当看到了你刚刚创建的 Job Detail 属性了。

点击 List Jobs 菜单项来看 Schedueler 上所有的 JobDetail。点击你的 JobDetail 左边的执行(Execute) 链接。一个记事本窗口弹了出来。虽然不怎么实用，但你可以想像其他可能的事。当你阅读此书时，你是否有些疑惑：你如何才能充分利于 Quartz 于非 Java 的 Job 呢？现在你会得到你的答案。

现在是个探索 Loggin 菜单的好时机了。点击 Loggin 并滚动到列表的底端。这儿，你将会找到你刚刚执行的 Job 的一个条目，伴随着自你的 Scheduler 启动以来的其他已发生的活动项。

点击列出所有 Trigger(List all Triggers) 菜单项来显示你的 Job 初始化加载 Trigger。注意到，这里没有选项用于从 Quartz Web 程序中增加一个 Trigger。Trigger 必须声明在 job.xml 文件中，这个文件也在 resources 目录中。job.xml 文件列表被声明在 quartz.properties 文件的 Job 初始器段中。

那就是 Quartz Web 程序的所有内容了。剩下要做的事就是增加你自己的 Job 和 Trigger 了。

第十四章. 工作流中使用 Quartz (第一部分)

第十四章. 工作流中使用 Quartz

Quartz 可以执行一个难以置信的 Job，来完成预计的任务。不幸的是，用来运行一个业务的 Job 经常比单一的 Job 或任务要稍稍复杂。每年百万计的美金花费到理解、设计和构建组织的业务流程。Quartz 框架包含一些设施用于把多个 Job 链接起来构建一个简单的业务流程模型 。本章讨论你能如何用 Quartz 连接 Job。为获得实际的工作流可操作性，你还需要一些来自于 Quartz 框架的东西。本章就来看为实现你的 Job 所构成的工作流可以如何扩展 Quartz 框架。

一. 什么是工作流

Web 上聚集了个人或团体关于工作流的定义和实例。有人定义工作流为“自动化的后台管理系统”。另一些人使用“业务流程建模” 一语，并收取许多的咨询费用向你解释这个概念。对于本章的要义，我们使用如下的工作流定义：

> 工作流是出现在某一特定时序中的一系列互为依懒的任务。

在我们进一步深入到本章内容后，该定义将变得越发明淅。

·工作流中什么地方需要 Quartz ？

如果你问，“工作流中什么地方要使用 Quartz 来效力？” 答案就是，“相当的多。” 甚至是像自动执行构建或只是发送电子邮件的简单任务里，都有工作流的位置。Quartz 支持一些基本的途径来把多个 Job 串起来。本章就讨论那些并展示如何集成 Quartz 到一个流行的开源的工作流解决方案。

二. Quartz 中的 Job 串联

Job 串联这一主题是在 Quartz 用户论坛中随着时间的推移而出现的。实际上，问的人多了，它已成为 Quartz FAQ (见 http://www.opensymphony.com/quartz/faq.html#chaining) 的一部分了。无论他们是否实现了他，多数问询了 Quartz 是否支持 Job 串联的 用户实际上问的是，“我该如何把工作流加到 Quartz 中？” 但是在我们深入到 OSWorkflow 之前，让我们瞧瞧使用 Quartz 框架自带的东西你该如何完成 Job 的串联。

·Job 串联不是工作流

为明确起见，我们应该认清：Job 串联是指 Quartz Job 在前一个 Job 完成时有条件或无条件的部署另一个 Job。单独使用 Quartz 框架来达成这一过程会受困于一些问题和限制。然而，还是值得去练习一下，以便于你能充分了解那些限制。

接下来的陈述我们将使你们中的某些人要抓狂了，但是你会从这份材料中获息，Quartz 中的 Job 串联并非工作流。它或许和工作流有些类似，有着工作流的气味让人感觉到像是工作流，但它确确实实不是工作流，你很快就会知道这一点的。你可以认为这是“懒汉式工作流”，在小额预算时权作一种工作流类型。严格意义上，像 OSWorkflow 那样的工作流系统提供了比你能从 Quartz 的 Job 串联获得的更多的功能。这不是在打击 Quartz：Quartz 是为 Job 调度设计的，它也做得非常好。工作流框架，像 OSWorkflow 是设计来实现工作流程的。两者都是伟大的工作。

·用监听器实现 Job 串联

第一种实现 Job 串联的方法是使用 Quartz 监听器。这是通过创建一个 JobListener 或 TriggerListener 来完成，当它们被 Scheduler 通知时，就为本次执行部署上下一个 Job。JobListener 的 jobWasExecuted() 方法，或 TriggerListener 的 TriggerComplete() 是用来“链接”到下一个 Job 的地方。我们假定，你有一个执行一些重要业务逻辑的 Job，叫做 ImportantJob。你就像先前创建其他任何的 Quartz Job 那样创建它。代码 14.1 显示了这个 Job 的概要，描述了你的 Quartz 应用需要执行某一重要的 Job。

代码 14.1. ImportantJob，描述了你可能为你的业务所要执行的 Job

```
1.  public class ImportantJob implements Job {
2.
3.      public void execute(JobExecutionContext context) {
```

```
            // Do something important in this Job
        }
}
```

注意到代码 14.1 中的 Job 什么也没指定。让我们进一步假如你需要在代码 14.1 中所示的 ImportantJob 结束时链接到第二个 Job。你可以选择任何 Job，不过我们让这个 Job 打印出之前运行的 Job 的一些细节信息。把它叫做 PrintJobResultJob (见 代码 14.2)。

代码 14.2. PrintJobResultJob 打印有关之前运行的被链接的 Job 的信息

```
public class PrintJobResultJob implements Job {
     Log logger = LogFactory.getLog(PrintJobResultJob.class);

     public void execute(JobExecutionContext context) {

          // Get the JobResult for the previous chained Job
          JobResult jobResult =
               (JobResult) context.getJobDataMap().get("JOB_RESULT");

          // If no Job was chained before this one, do nothing
          if (jobResult != null) {
               logger.info(jobResult);
          }
     }
}
```

PrintJobResultJob 设计为察看它的 JobDataMap 来确定是否存在一个 JobResult 对象。类 JobResult 不是 Quartz 框架的组成部分，但是你能不费力的创建它来表示 Job 的执行结果。在许多实例中，创建一个像 JobResult 的类是很有帮助的。代码 14.3 显示了我们例子中的 JobResult 类。

代码 14.3. JobResult 表示 Job 的执行结果

```
public class JobResult {

     private boolean success;
     private String jobName;
     private long startedTime;
     private long finishedTime;

     public JobResult(){
          startedTime = System.currentTimeMillis();
     }

     // getters and setters not shown in this listing

     public String toString() {
          StringBuffer buf = new StringBuffer();
          buf.append(jobName);
          buf.append(" executed in ");
          buf.append(finishedTime - startedTime);
          buf.append(" (msecs) ");

          if (success) {
               buf.append("and was successful. ");
          } else {
               buf.append("but was NOT successful. ");
          }

          return buf.toString();
```

```
        }
    }
```

代码 14.3 中 JobResult 类包含了有关 Job 执行结果的几块信息：启动时间，完成时间，和指示是否执行成功的标志。显然，你也可以在你的版本中加任何需要的字段；这个只是一个简单的例子。

在这个 Job 串联例子的下一步是创建一个监听器类来执行实际的串联操作。对于本例中，我们将用一个 JobListener，只要一个 TriggerListener 就可很好的工作。代码 14.5 显示了这个 Job 串联的 JobListener.

代码 14.5. 一个执行 Job 串联的非全局 JobListener

```
public class JobChainListener implements org.quartz.JobListener {
    Log logger = LogFactory.getLog(JobChainListener.class);

    public Class nextJobClass;

    public String listenerName;

    public JobChainListener() {
        super();
    }

    public JobChainListener(String listenerName, Class nextJob) {
        setName(listenerName);
        this.nextJobClass = nextJob;
    }

    public String getName() {
        return listenerName;
    }

    public void setName(String name) {
        this.listenerName = name;
    }

    public void jobToBeExecuted(JobExecutionContext context) {
        // Do nothing in this example
    }

    public void jobExecutionVetoed(JobExecutionContext context) {
        // Do nothing in this example
    }

    public void jobWasExecuted(JobExecutionContext context,
            JobExecutionException jobException) {
        Scheduler scheduler = context.getScheduler();

        try {
            // Create the chained JobDetail
            JobDetail jobDetail = new JobDetail("ChainedJob", null, nextJobClass);

            // Create a one-time trigger that fires immediately
            Trigger trigger =  TriggerUtils.makeSecondlyTrigger(0, 0);
            trigger.setName("FireNowTrigger");
            trigger.setStartTime(new Date());

            // Update the JobResult for the next Job
            JobResult jobResult = (JobResult) context.getJobDataMap().get("JOB_RESULT");
            jobResult.setFinishedTime(System.currentTimeMillis());
            jobResult.setSuccess(true);
```

```
50.
51.                 // Pass JobResult to next job through its JobDataMap
52.                 jobDetail.getJobDataMap().put("JOB_RESULT", jobResult);
53.
54.                 // Schedule the next job to fire immediately
55.                 scheduler.scheduleJob(jobDetail, trigger);
56.
57.                 logger.info(nextJobClass.getName() + " has been scheduled executed");
58.
59.             } catch (Exception ex) {
60.                 logger.error("Couldn't chain next Job", ex);
61.                 return;
62.             }
63.         }
64.     }
```

如你从代码 14.5 中看到的，链接到下一个 Job 的工作是在 jobWasExecuted() 方法中完成。你已从第七章，“实现 Quartz 监听器” 学习到，Scheduler 在 Job 完成执行时调用 jobWasExecuted() 方法的。使得这是把 Job 串链在一起的最佳方法。在代码 14.5 的 jobWasExecuted() 方法中，进行了几件事情。

首先，为被链接 Job 新建了一个 JobDetail 和 Trigger。接着从 JobDataMap 中获取了第一个 Job 的 JobResult，并设置了 finishedTime 和 success 字段的值。为了让被链接的(下一个) Job 能访问到 JobResult 对象，它被载入到被链接 Job 的 JobDataMap 中

> **传递数据从一个 Job 到另一个**
>
> JobResult 的想法用在这个例子中描绘了，尽管是可以传递数据从一个 Job 到被链接的的 Job，但还是有些笨拙的。在当前的例子中，数据是在监听器来传递的。这是可发生串联操作唯一的地方。假如你需要串联三个 Job，这种纠缠关系就会变得很糟。（译者 Unmi 注：不知所云）

代码 14.5 的最后部分部署了一个新的 Job 到 Scheduler 上。因为新 Job 的 Trigger 设置为立刻触发，PrintJobResultJob 将会立即执行。回看代码 14.2，你会发现当 PrintJobResultJob 类的 execute() 方法被执行时，它会获取到 JobResult 对象，并调用它的 toString() 方法。再次说一下，这是一个向你展示如何串联 Job 的非常简单的例子。你的 Job 显然会有比这更复杂的逻辑。然而 Job 串联工作是一样的。

因此，好消息是问题不在于使用监听器类来串联 Job 的难度上。坏消息是这种用法存在有几个严重的设计问题。首先，尽管我们足够的聪明把下一 Job 的名字传到了监听器类中，但这段代码有着相当紧的耦合性。监听器需要在创建之时被告知链接的 Job，所以监听在它的生命期内变得与某一特定被链接 Job 紧密的耦合起来了。你需要为每一个被链接的 Job 分别建立监听器，那么对于需要链接到多余两个 Job 时又该如何呢？事情很快就变得不好控制了。

> **监听器用法的变化**
>
> 当然，存在一些不同的方式让你能实现你的监听器使之工作。例如，你可以创建一个独立的 JobDetail 实例，设置它的 durability 标志为 true，并预存储到 Scheduler 中。这样的话，监听器就不需要每次都创建这个 JobDetail 和相应的 Trigger；只需要创建一个 JobDataMap，把 JobResult 置于其中，并调用 scheduler.triggerJob(jobName, groupName, jobDataMap)；这个已存在的 Job 就会被执行并传递 JobResult。

·使用 JobDataMap 进行 Job 串联

另一串联 Job 的方法是使用 JobDataMap 来存储下一个要执行的 Job。在早先监听器的例子中，我们使用了 JobDataMap 来存储和传递 JobResult，但是这一新用法可以摈弃监听器而用 JobDataMap 来完成这一切。

因为可以让我们摆脱监听器类了，这就意味着 Job 必须自己处理链接到下一 Job。需要的话，这个行为可以抽象到一个基础 Job 类中。不过，这儿的例子为保持尽可能的简单，所以不那么做。

代码 14.5中展示了新的 ImportantJob 类。在它完成之后，链中的下一 Job 就得到部署了。你可能还想基于某个标志或是上次执行的条件来决定部署下一 Job。例如，假如某标志设置为 true 时，你要执行 Job A；而在该标志为 false，你会要执行 Job B。

链中的下一 Job 由你选择一个 Key 存键在 JobDataMap 中。在本例中，我们使用 NEXT_JOB。这种用法的其中一个问题是必须在 Job 执行之前把下一 Job 存入到 JobDataMap 中。

代码 14.5. Job 串联也能用 JobDataMap 实现

```
1.  public class ImportantJob implements Job {
2.       static Log logger = LogFactory.getLog(JobChainListener.class);
3.
4.       public void execute(JobExecutionContext context) {
5.
6.            // Do something important in this Job
7.
8.            // Set some condition based on this Job execution
9.            boolean success = true;
10.
11.           // schedule the next Job if condition was successful
12.           if (success) {
13.                scheduleNextJob(context);
14.           } else {
15.                logger.info("Job was NOT chained");
16.           }
17.
18.      }
19.
20.      protected void scheduleNextJob(JobExecutionContext context) {
21.           JobDataMap jobDataMap = context.getJobDataMap();
22.
23.           String nextJob = jobDataMap.getString("NEXT_JOB");
24.           if (nextJob!= null && nextJob.length() > 0) {
25.
26.                try {
27.                     Class jobClass = Class.forName(nextJob);
28.                     scheduleJob(jobClass, context.getScheduler());
29.                } catch (Exception ex) {
30.                logger.error("error scheduling chained job", ex);
31.                }
32.           }
33.       }
34.
35.      protected void scheduleJob(Class jobClass, Scheduler scheduler) {
36.           JobDetail jobDetail =
37.                new JobDetail(jobClass.getName(), null, jobClass);
38.
39.           // Create a fire now, one time trigger
40.           Trigger trigger = TriggerUtils.makeSecondlyTrigger(0, 0);
41.           trigger.setName(jobClass.getName() + "Trigger");
42.           trigger.setStartTime(new Date());
43.
44.           // Schedule the next job to fire immediately
45.           try {
46.                scheduler.scheduleJob(jobDetail, trigger);
47.           } catch (SchedulerException ex) {
48.
49.           logger.error("error chaining Job "
50.                + jobClass.getName(), ex);
51.               }
52.       }
53.  }
```

从 JobDataMap 中获取到 Job 的的名称之后，创建了 JobDetail 和 Trigger，并部署了 Job。和 和前面监听器的例子一样，部署首个 Job 的代码需要加入最初被链接的 Job 到 JobDataMap 来让一切正常工作。这能在 Job 首次加到 Scheduler 和 Scheduler 启动之后来做。代码 14.6 展示了这些。

代码 14.6. 第一个被链接的 Job 需要在第一个 Job 被部署时配置

```
1.  public class NewScheduler {
2.       static Log logger = LogFactory.getLog(NewScheduler.class);
3.
4.       public static void main(String[] args) {
5.
6.            try {
7.                 // Create and start the Scheduler
8.                 Scheduler scheduler = StdSchedulerFactory.getDefaultScheduler();
9.                 scheduler.start();
10.
11.                JobDetail jobDetail = new JobDetail("ImportantJob", null, ImportantJob.class);
12.
13.                // Set up the first chained Job
14.                JobDataMap dataMap = jobDetail.getJobDataMap();
15.                dataMap.put("NEXT_JOB", "org.cavaness.quartzbook.chapter14.ChainedJob");
16.
17.                // Create the trigger and scheduler the Job
18.                Trigger trigger = TriggerUtils.makeSecondlyTrigger(10000, 0);
19.                trigger.setName("FireOnceTrigger");
20.                trigger.setStartTime(new Date());
21.
22.                scheduler.scheduleJob(jobDetail, trigger);
23.
24.           } catch (SchedulerException ex) {
25.                logger.error( ex );
26.           }
27.      }
28. }
```

14.6 中的代码是用来部署第一个 Job，并且同时设置了链中的下一个 Job。如果有第三个 Job，它就不得不在第二个 Job 的 JobDataMap 中设置了。你很明显能看出在链中当有多余两个 Job 时会有多笨拙。OSWorkflow 能帮上忙了，还能解决其他一些乱糟糟的问题。

·为 Job 串联使用 JobInitializationPlugin

如果你在 quartz_jobs.xml 文件中指明你的 Job 信息，并使用 JobInitializationPlugin 来加载那些信息。这种用法也许不那么的坏。那是因为你可以很容易的在 XML 文件中指定 Job 链。例如，就看代码 14.7 中的 quartz_jobs.xml。

代码 14.7. 使用 JobInitializationPlugin 时 Job 串联还有了一定程度的可理性

```
1.  <?xml version='1.0' encoding='utf-8'?>
2.
3.  <quartz>
4.    <job>
5.      <job-detail>
6.        <name>ImportantJob</name>
7.          <job-class>
8.            org.cavaness.quartzbook.chapter14.ImportantJob
9.          </job-class>
10.
11.         <job-data-map allows-transient-data="true">
12.           <entry>
```

```
            <key>NEXT_JOB</key>
            <value>org.cavaness.quartzbook.chapter14.ChainedJob</value>
          </entry>
        </job-data-map>
     </job-detail>

    <trigger>
        <simple>
        <name>FireOnceTrigger</name>
          <group>DEFAULT</group>
          <job-name>ImportantJob</job-name>
          <job-group>DEFAULT</job-group>
          <start-time>2005-07-19 8:31:00 PM</start-time>
          <repeat-count>0</repeat-count>
        </simple>
    </trigger>
   </job>
</quartz>
```

在代码 14.7 中 Job 信息的结果和前的例子是一样的，还更好些。如果你需要改变哪个 Job 会链接到 ImportatJob 的话，你仅需要修改这个 XML 文件。在前面的 Job 串联的例子中，不得不修改代码然后重新编译。

[译者 Unmi 注:] 本部分作者想要表达的内容确如他之前有交待的那样，会让人觉得有些抓狂。从前面各章节一路下来，唯见本部分有些描述让人难以琢磨。请读者朋友结合实际用意仔细理解。另：对 job-chaining 使用的是 Job 串联的说法，自我感觉还没有到位，但还没有想到更好的说法。单独的 chain 用的链(接)。

第十四章. 工作流中使用 Quartz (第二部分)

三. OSWorkflow 快速入门

像 Quartz 一样，OSWorkflow 是一个完全由 Java 构建的开源项目，而且也是 OpenSymphony 家族项目的成员。还有许多的工作流项目，商业的或是开源的。OSworkflow 在设计上与 Quartz 有很多相似性，所以把这两个框架进行集成不用太费我们的心思。

OSWorkflow 工作在有限状态机的原则之上。一个工作流由一系列状态组成，包括一个开始状态和一个或多个结束状态。从某一状态迁移到另一状态，需要发生一次转换。实际上从某一特定状态可能会有多种转换，你也可以在同一时间从某一状态发生多种转换。选择什么转换依赖于环境，对状态的输入，和一些我们将在后面讨论的条件信息。

> **无可替代的 OSWorkflow 文档**
>
> 本章中有关 OSWorkflow 的材料不应该取代在 OSWorkflow 站点上的经过严格审查过的文档。我们主要关注在解释如何应用 OSWorkflow 于 Quartz 中。比如，在本章中，我们不使用很多的 OSWorkflow 的可用特性，因为我们只想让你大体上了解 OSWorkflow 能做什么。我们没有时间去研究很多其他的特性，因此，一定要去阅读 OSWorkflow 网站上的文档和指导教材。

·Workflow 描述文件

OSWorkflow 的一个关键组件是工作流描述器，有时候也叫做工作流定义；我们在这儿会交替的用这两个述语。工作流描述器定义了某一特定工作流的所有方面。这个描述器实现为一个 XML 文件，框架包含了一个 DTD 用于验证。你用用 OSWorkflow 工作的多数时间将会涉及到理解描述器文件的布局与规则。你可以在 http://www.opensymphony.com/osworkflow/workflow_2_7.dtd 查看 2.7 版的 DTD 文件。

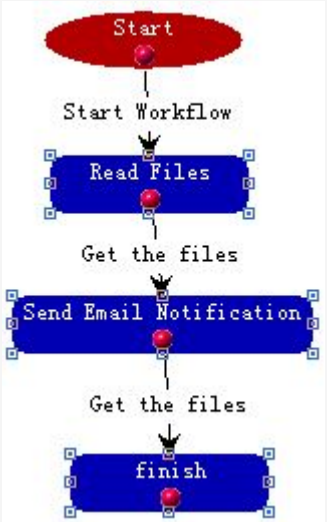
·OSWorkflow 的概念

为有助于讲解，我们创建一个假想的工作流并在本章的后续部分中使用。创建好后，这个工作流将扫描某个目录来查找文件。在医学领域中，这些可能是来自于客户或病人信息的电子单据。

假如检查时存在任何文件，这个工作流程就读取他们并存储信息到数据库中。工作流的最后一步是产生一个电子邮件，记载了插入到数据库中的记录的数量。你可以想像一下，假如我们正运行着一个全球范围的业务，我们会期望文件会在白天或黑夜的任何时间丢过来。那就是为什么在调度器方面 Quartz 是如此的完美。

我们想用 Quartz 和 OSWorkflow 构建一个系统，用它来定期的检查这些文件，当接收到一个或多个文件时，通过一个指定的工作流来处理收到的文件。图 14.1 显示了我们想用的工作流。

图 14.1. 一个处理电子数据文件的示例工作流



你能从图 14.1 中的工作流中看出，由几个步骤组成了工作流本身。开发一套可在许多不同工作流中重用的步骤是个很好的想法。让我们看看这个示例工作流的一些定义和说明。

首先，一个工作流描述符(定义) 文件具体以下格式：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE workflow PUBLIC
  "-//OpenSymphony Group//DTD OSWorkflow 2.7//EN"
  "http://www.opensymphony.com/osworkflow/workflow_2_7.dtd">

<workflow>
  <initial-actions>
    ...
  </initial-actions>

  <steps>
    ...
  </steps>
</workflow>
```

我们现来浏览工作流重要的部分。

·工作流步骤

一个工作流由一些状态和步骤构成。每个步骤由你给它指定一个名字。在 OSWorkflow 中，由某一步骤和另一步骤的转换是一个动作的结果。下面的 XML 片断描绘了一个步骤：

```
<step id="1" name="Read Files">
 <actions>
  <action id="1" name="Read the records from the files" auto="true">

  <results>
   ...
  </results>
  </action>

 </actions>
</step>
```

·工作流动作

简而言之，任何工作流引擎的目标都是使工作流从开始进行到结束。那意味着我们需要一种方式来转换工作流从某一步骤到另一步骤。在 OSWorkflow 中，动作就是用来决定选用哪一个转换路径，因此，也决定了要转换到的步骤。在工作流中一个单一的步骤可以有多条路径(或者叫转换) 可基于动作的结果来执行。有几方面因素有助于决定工作流应采取哪个路径，包括外部事件和来自用户的输入。工作流动作使用条件和函数去确定动作的结果，进而要采取的转换。每个动作必须有至少一个无条件的结果和零个或多少有条件的结果。这有助于确保一个工作流总是能从一个步骤进行转换，甚至是不存在有条件的结果时也能导致一个转换。通常，一个动作会从一个步骤产生单个转换。我们在后面要讨论到，这不完全对，因为在从在步骤上发生多个转换时可以导致一个分支。最后，这些多重转换必须归并回一条路径上。下面的 XML 片断提供了一个工作流动作元素的例子：

```
<action id="1" name="Start Workflow">
 <results>
  <unconditional-result old-status="Finished" status="Queued" step="1"/>
 </results>
</action>
```

叫做*初始化动作(initial actions)* 特殊类型的动作通过指示第一个步骤执行来启动一个工作流。这儿是一个 initial-actions 元素的例子：

```
<initial-actions>
```

```
 <action id="1" name="Start Workflow">
  <results>
    <unconditional-result old-status="Finished"
      status="Queued" step="1" />
  </results>
 </action>
</initial-actions>
```

·动作结果

动作结果告诉工作流下一步要做什么任务。OSWorkflow 提供了有条件的无条件的两种结果。一个结果可以用多个条件元素，由它们计算出 true 或 false。计算为 true 的结果元素中的第一个条件会得到执行并决定了下个步骤。假如只有一个无条件的结果或者不存在为 true 的有条件结果，那么这个无条件的结果会得到执行。

存在三种工作流结果元素类型，不论它们是有条件的还是无条件的：

·转换到单一步骤

·分支到两个或多个步骤

·合并多个步骤到单一步骤上

你要组合使用它们来构建你的工作流。

·工作流函数

在 OSWorkflow 中，多数它的强大和灵活之处体现在函数和条件的使用上。函数是在从某一步骤到另一步骤转换期间能执行的逻辑；这是可以做大部分工作的地方，尤其是对于我们的 Quartz + OSWorkflow 集成应用上来说。OSWorkflow 包括调用 EJB 方法的函数，使用 Java 消息服务(Java Message Service，JMS) 的函数，发送电子邮件的函数，还有更多。函数还支持了我们感兴趣的方面是它能调用一个普通的 Java 类。

OSWorkflow 包括了函数的接口 com.opensymphony.workflow.FunctionProvider. 所有我们必须做的是创建一个实现 FunctionProvider 接口的 Java 类，它只包含一个方法：

```
public void execute(java.util.Map transientVars,
                    java.util.Map args,
                    com.opensymphony.module.propertyset.PropertySet ps)
             throws WorkflowException;
```

创建好了 FunctionProvider 类之后，我们要像如下这样设置函数到工作流中：

```
<function type="class">
  <arg name="class.name">
    org.cavaness.quartzbook.chapter14.ReadFileFunction
  </arg>
</function>
```

当工作流来到了这个步骤和定义了函数的动作时，它就会呼叫我们的 Java 类，并调用函数的 execute() 方法。这真的是很强大的，因为你能易的集成新的或遗留的系统到你的工作流中。

·自动动作

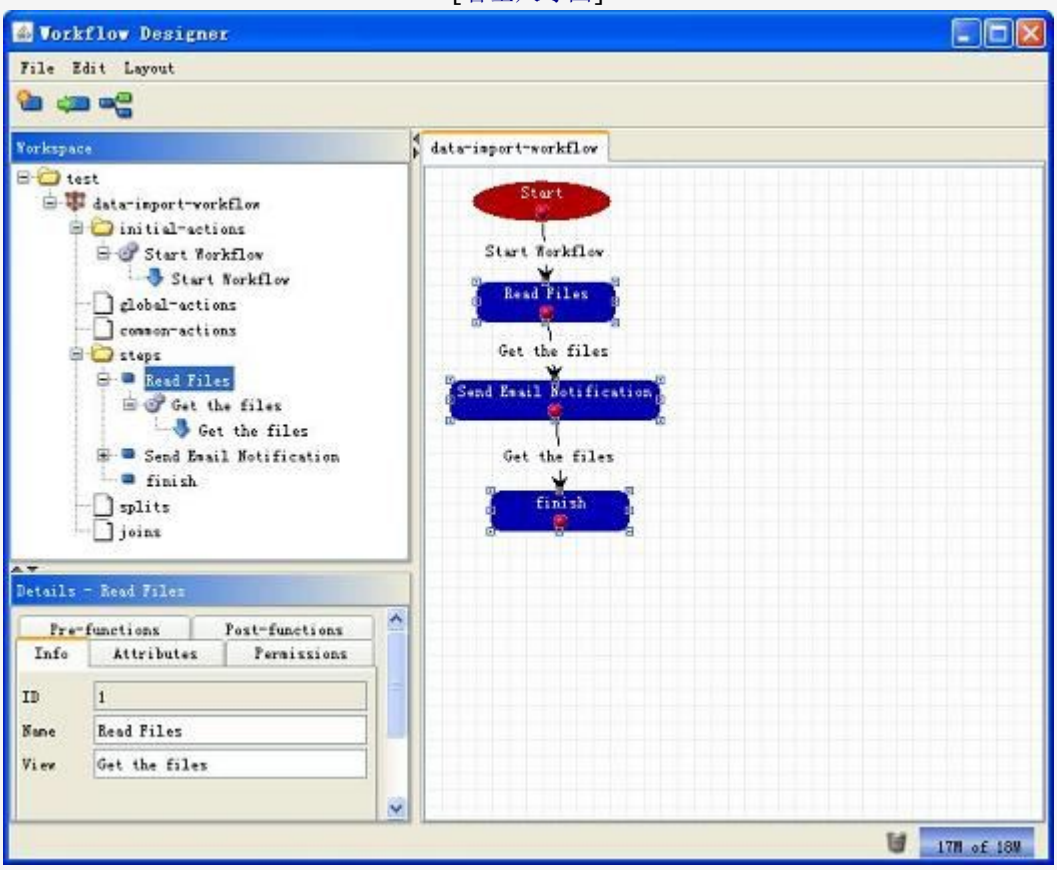
假如你构建这样一个工作流，你希望它的步骤在没有用户输入时自动转换，你可以在某个动作上使用 auto 属性来强制发生动作。这意味着工作流一旦启动后，就会缘着它的步骤自动运行，而不用等待一个外部事件来引发它的转换。[译者 Unmi 注：原文似乎未把某些个别的自动化动作与完全自动化的工作流分清。]

·工作GUI 设计器

我们这里没有使用 GUI 设计器，但还值的一提的是 OSWorkflow 团队已创建了一个图形化的工作来建立和编辑工作流定义。这个工具是一个客客户端的，可使用 Sun 的 Java Web Start 技术来运行。图 14.2 显示了这个 GUI 工具加载了示例工作流的截图。

图 14.2. OSWorkflow 设计器是一个建立和修改工作流好工具

[看全尺寸图]



第十四章. 工作流中使用 Quartz (第三部分)

四. Quartz 与 OSWorkflow 的集成

OSWorkflow 与 Quartz 集成的第一步是要改变关于 Job 的思维方式。当把 OSWorkflow 引入到你的 Quartz 应用时你需要以完全不同的方式来思考。那也不是说你当前的想法就是糟糕的或不正确的，只是与 Quartz 一同用工作流强迫你生发一些关于是什么组成 Job 的新的思维。你过去概念中的 Job 现成变成了一个 OSWorkflow 函数。你可以认为是你原有 Job 实质上存在的逻辑作为工作流中的步骤。你仍然需要使用 Quartz 的 Job，但是，当与 Quartz 框架集成工作流时，一个 Quartz Job 将用来初始化工作流。在工作流运行时，这个 Job 将会等待它直至结束。

在本章前面部分，当我们谈到串联 Job 时，每个 Job 代表了一个独立的任务。Jox X 执行后并完成一个任务，接着通知 Job Y 去执行一个有点关联却是独立的任务。在这两个任务间必须有一些依赖关系，否则你不能把它们链接在一起。

当加入工作流到这个流程中时，那些独立的 Job 就变身为工作流中的步骤了，而且你仅需要创建单个的 Job。当收到 Scheduler 的通知时，那个 Job 就起过工作流然后等待着工作流的完成。这其中有一些严格的寓意。好消息是，通过使用 OSWorkflow，你只需更少的 Quartz Job，因为早先的 Job 现在成了步骤(实际是函数)。坏消息是假如你有创建了大量的 Job，将要耗费一些工作量去转换 Job 到 OSWorkflow 的函数。

·下载和安装 OSWorkflow

你可以从它在 OpenSymphony 站点 http://www.opensymphony.com/osworkflow 的主页下载完整的发行版。从发布页的根目录获取到二进制版的 OSWorkflow 和其他第三方的库，它们在 <OSWORKFLOW_DISTRIBUTION>/lib/core 目录下。把这些二进制包扔到你的项目的 lib 目录中。这应当和 Quartz 二进制包所在的同一目录。

你必须建立两个配置文件并放置到你的 classes 目录中。第一个你需要创建的配置文件叫做 osworkflow.xml。这个文件在 OSWorkflow 启动并配置运行时环境时被加载。我们例子中的该文件如代码 14.8 中显示。

代码 14.8. osworkfllow.xml 文件用于配置 OSWorkflow 的运行时环境

```
<osworkflow>
  <persistence
    class="com.opensymphony.workflow.spi.memory.MemoryWorkflowStore"/>
  <factory class="com.opensymphony.workflow.loader.XMLWorkflowFactory">
    <property name="resource" value="workflows.xml" />
  </factory>
</osworkflow>
```

假如你再看看 OSWorkflow 的文档，你会发现你可以从多种类型的持久性存储和工作流工厂中作出选择。在代码 14.8 中所用的都是最简单的并且为我们的例子工作的很好的。

配置在代码 14.8 中的工作流工厂类叫做 XMLWorkflowFactory，它包括一个称为 resource 的属性。XMLWorkflowFactory 用于加载一个包含了所有工作流的资源文件。在这里的 resource 属性的值是 workflows.xml。只要你想要多少个，就允许你可以有多少个不同的工作流。每个工作流驻留在一个单独的 XML 文件中，但是你需要以某种方式指定可用的工作流列表给 OSWorkflow 引擎。因为我们已经指定工厂为 XMLWorkflowFactory，所以框架会查看 workflows.xml 文件来得到可用的工作流并加载它们。代码 14.9 显示了我们例子里的 workflows.xml 文件。

代码 14.9. workflows.xml 文件定义了应用可用的工作流列表

```
<workflows>
  <workflow
    name="data-import"
    type="resource"
    location="data-import-workflow.xml"/>
</workflows>
```

代码 14.9 只列了一个工作流：data-import，它将在我们启动工作流时用作参考。实际的工作流定义是破存储在文件 data-import-workflow.xml 中的。代码 14.10 显示了 data-import 工作流。

代码 14.10. data-import 工作流被定义在一个 XML 文件中

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE workflow PUBLIC
                 "-//OpenSymphony Group//DTD OSWorkflow 2.7//EN"
                 "http://www.opensymphony.com/osworkflow/workflow_2_7.dtd">
<workflow>
    <initial-actions>
        <action id="1" name="Start Workflow">
            <results>
                <unconditional-result old-status="Finished" status="Queued"
                    step="1" />
            </results>
        </action>
    </initial-actions>

    <steps>
        <step id="1" name="Read Files">
            <actions>
                <action id="2" name="Get the files" auto="true">
                    <pre-functions>
                        <function type="class">
                            <arg name="class.name">
                               org.cavaness.quartzbook.chapter14.ReadFileFunction
                            </arg>
                        </function>
                    </pre-functions>

                    <results>
                        <unconditional-result old-status="Finished"
                            status="Underway" step="2" />
                    </results>
                </action>
            </actions>
        </step>

        <step id="2" name="Send Email Notification">
            <actions>
                <action id="3" name="Get the files" auto="true">
                    <pre-functions>
                        <function type="class">
                            <arg name="class.name">
                               org.cavaness.quartzbook.chapter14.SendEmailFunction
                            </arg>
                        </function>
                    </pre-functions>

                    <results>
                        <unconditional-result old-status="Finished"
                            status="Underway" step="3" />
                    </results>
                </action>
            </actions>
        </step>

        <step id="3" name="finished" />
    </steps>
```

```
56.  </workflow>
```

定义在代码 14.10 的工作流包含两个步骤：“读文件” 和 "发 e-mail 通知"。当工作流启动后，initial-actions 块被执行，继而它调用步骤 1。当进入到步骤 1 后，org.cavaness.quartzbook.chapter14.ReadFileFunction 的 execute() 方法就被调用。代码 14.11 显示了这一函数。

代码 14.11. 工作流引擎在步骤 1 其间调用 ReadFileFunction

```
1.  public class ReadFileFunction implements FunctionProvider {
2.       static Log logger = LogFactory.getLog(ReadFileFunction.class);
3.
4.       public void execute(Map transientVars, Map args, PropertySet ps)
5.                 throws WorkflowException {
6.            logger.info("Entered " + this.getClass().getName());
7.
8.            // Read the files and process the data
9.            String dirName = (String)transientVars.get("SCAN_DIR");
10.           if ( dirName == null ) {
11.                throw new InvalidInputException( "Scan dir not set" );
12.           }
13.
14.           File dir = new File( dirName );
15.           File[] files = dir.listFiles();
16.
17.           int fileCount = files.length;
18.           ps.setInt( "FILE_COUNT", fileCount );
19.      }
20.  }
```

当 ReadFileFunction 的 execute() 方法完成后，工作流会转换到步骤 2。在步骤 2，org.cavaness.quartzbook.chapter14.SendEmailFunction 被调用并获得机会执行。SendEmailFunction 显示在代码 14.12 中。

代码 14.12. SendEmailFunction 在工作流的步骤 2 期间被调用

```
1.  public class SendEmailFunction implements FunctionProvider {
2.       static Log logger = LogFactory.getLog(SendEmailFunction.class);
3.
4.       public void execute(Map transientVars, Map args, PropertySet ps)
5.                 throws WorkflowException {
6.            logger.info("Entered " + this.getClass().getName());
7.
8.            int fileCount = ps.getInt("FILE_COUNT");
9.            logger.info( "File count " + fileCount );
10.
11.           // Email creation code not shown
12.      }
13.  }
```

显然我们漏下了这个函数的实例；也没有对此多加以说明。我们假定你知道如何使用 JavaMail 发送电子邮件。

第十四章. 工作流中使用 Quartz (第四部分)

四. 创建一个工作流 Job

最后，我们需要介绍启动工作流的 Quartz Job 。当 Scheduler 调用了它，Quartz Job 就查找工作流的名字，并启动、运行相应的工作流。如果没有在 JobDataMap 中配置工作流的名字，Job 就会直接退出。

代码 14.13 显示了 WorkflowJob.

代码 14.13. Quartz WorkflowJob 设计为调用一个 OSWorkflowJob

```
1.  public class WorkflowJob implements Job {
2.       static Log logger = LogFactory.getLog(WorkflowJob.class);
3.
4.       /**
5.        * Called by the scheduler to execute a workflow
6.        */
7.       public void execute(JobExecutionContext context)
8.                 throws JobExecutionException {
9.            JobDataMap jobDataMap = context.getJobDataMap();
10.
11.           String wfName = jobDataMap.getString("WORKFLOW_NAME");
12.           if (wfName != null && wfName.length() > 0) {
13.                try {
14.                     executeWorkflow(wfName, jobDataMap);
15.                } catch (Exception ex) {
16.                     logger.error(ex);
17.                          throw new
18.                               JobExecutionException(ex.getMessage());
19.                }
20.           } else {
21.                logger.error("No Workflow name in JobDataMap");
22.           }
23.      }
24.
25.      protected void executeWorkflow(String workflowName,
26.           JobDataMap jobDataMap) throws WorkflowException {
27.
28.           // Create the inputs for the workflow from JobDataMap
29.           Map workflowInputs = new HashMap();
30.           Iterator iter = jobDataMap.keySet().iterator();
31.           while (iter.hasNext()) {
32.                String key = (String) iter.next();
33.                Object obj = jobDataMap.get(key);
34.                workflowInputs.put(key, obj);
35.           }
36.           // Create and execute the workflow
37.           Workflow workflow = new BasicWorkflow("someuser");
38.           workflow.setConfiguration(new DefaultConfiguration());
39.
40.           long workflowId = workflow.initialize(workflowName, 1, workflowInputs);
41.
42.           workflow.doAction(workflowId, 1, workflowInputs);
43.       }
44.  }
```

这个工作流实际是在代码 14.13 的 executeWorkflow() 方法中启动的。一个新的工作流实例被创建了。它通过从

JobDataMap 中读取到的工作流名字来初始化的。工作流实例的 initialize() 和 doAction() 方法用了一个 java.util.Map 作为第三个参数。Map 中的值会通过 transientVars 参数传递到工作流的每一个函数中。如果你回头看看代码 14.11，你会看到 SCAN_DIR 是如何从 transientVars 抽取出来的。这个数据最初是在 JobDataMap 的。

本例中，我们从 Quartz Job 中获得 JobDataMap 并传值到工作流中。这是集成这两个框架的其中一种方式，简单也很直截。

最后，代码 14.14 显示了 Scheduler 的代码，它用于部署 WorkflowJob 并把工作流名称和 SCAN_DIR 存入到 JobDataMap 中。

代码 14.14. WorkflowJob 以正常方式部署，但必须存工作流名称到 JobDataMap 中

```
1.  public class WorkflowScheduler {
2.       static Log logger = LogFactory.getLog(WorkflowScheduler.class);
3.
4.       public static void main(String[] args) {
5.
6.            try {
7.                 // Create and start the Scheduler
8.                 Scheduler scheduler =
9.                      StdSchedulerFactory.getDefaultScheduler();
10.                scheduler.start();
11.
12.                JobDetail jobDetail =
13.                     new JobDetail("WorkflowJob", null,
14.                          WorkflowJob.class);
15.                // Store the scan directory and workflow name
16.                JobDataMap dataMap = jobDetail.getJobDataMap();
17.                dataMap.put("SCAN_DIR", "c:\\quartz-book\\input");
18.                dataMap.put("WORKFLOW_NAME", "data-import");
19.
20.                // Create a simple trigger
21.                Trigger trigger =
22.                     TriggerUtils.makeSecondlyTrigger(30000, -1);
23.                trigger.setName("WorkflowTrigger");
24.                trigger.setStartTime(new Date());
25.
26.                // schedule the job
27.                scheduler.scheduleJob(jobDetail, trigger);
28.
29.           } catch (SchedulerException ex) {
30.                logger.error(ex);
31.           }
32.      }
33. }
```

你看到代码 14.14 时不应感到惊讶。唯一要注意的事情是我们存储了工作流名称到 JobDataMap 中。和本章前面的 Job 串联例子一样，假如你想使用 JobInitializationPlugin，你可以简单的在文件中指定工作流的名称。

五. 小结

我们从本章中学到了什么？首先，Job 串联可由 Quartz 框架实现。你可用所介绍的两种方法，只是不借助于监听器实现的方式或许会让你有些头疼。我希望你带走的另一课是 Job 串联并非工作流。它也许看起来像工作流，你可能还对此存有疑虑，只要回到前面从头至尾构建一个自己的例子就能清楚了。一定要读一读 OSWorkflow 的文档并找出我们未提及的特性。你就能明白工作流比 Job 串联丰富多了。

最后，你应该了解到了同 Quartz 一同使用 OSWorkflow 实际是颇为简单的。所有要用到的就是几个二进制包，几个配置文件和一些工作流函数。把它们同工作流定义文件打包到一块，你就大功告成了。很快，你就将构建出一个可重用的函数库和一连串工作流来运行你的业务。不久，你就会有一个为之自豪的精致小巧的应用程序。

附录 A. Quartz 配置参考（第一部分）

附录 A. Quartz 配置参考

本附录编写作为配置一个 Quartz 应用的快速参考。尽管这些信息在 Quartz 文档中都有，但是这个附录提供了一种更快的方式来查找配置属性和它们可能的值。

一. 主要的 Quartz 属性

表 A.1 列出了主要的 Scheduler 属性。它们用于声明和标识 Scheduler 和其他高层次的设置。

表 A.1. 主要的 Quartz Scheduler 属性

名称	必须	类型	默认值
org.quartz.scheduler.instanceName	否	String	'QuartzScheduler'
org.quartz.scheduler.instanceId	否	String	'NON_CLUSTERED'
org.quartz.scheduler.instanceIdGenerator.class	否	String	org.quartz.simpl.SimpleInstanceIdGenerator
org.quartz.scheduler.threadName	否	String	instanceName+'_QuartzSchedulerThread'
org.quartz.scheduler.idleWaitTime	否	Long	30000
org.quartz.scheduler.dbFailureRetryInterval	否	Long	15000
org.quartz.scheduler.classLoadHelper.class	否	String	org.quaartz.simpl.CascadingClassLoadHelper
org.quartz.context.key.SOME_KEY	否	String	None
org.quartz.scheduler.userTransactionURL	否	String	'java:comp/UserTransaction'
org.quartz.scheduler.wrapJobExecutionIn UserTransaction	否	Boolean	false
org.quartz.scheduler.jobFactory.class	否	String	org.quartz.simple.SimpleJobFactory

·org.quartz.scheduler.instanceName

每个 Scheduler 必须给定一个名称来标识。当在同一个程序中有多个实例时，这个名称作为客户代码识别是哪个 Scheduler 而用。假如你用到了集群特性，你就必须为集群中的每一个实例使用相同的名称，以使它们成为“逻辑上” 是同一个 Scheduler 。

·org.quartz.scheduler.instanceId

每个 Quartz Scheduler 必须指定一个唯一的 ID。这个值可以是任何字符串值，只要对于所有的 Scheduler 是唯一的。如果你想要自动生成的 ID，那你可以使用 AUTO 作为 instanceId。从版本 1.5.1 开始，你能够定制如何自动生成实例 ID。见 instanceIDGenerator.class 属性，会在接下来讲到。

·org.quartz.scheduler.instanceIdGenerator.class

从版本 1.5.1 开始，这个属性允许你定制instanceId 的生成，这个属性仅被用于属性 org.quartz.scheduler.instanceId 设置为 AUTO 的情况下。默认是 org.quartz.simpl.SimpleInstanceIdGenerator，它会基于主机名和时间戳来产生实例 ID 的。

·org.quartz.scheduler.threadName

可以是对于 Java 线程来说有效名称的任何字符串。假如这个属性未予指定，线程将会接受 Scheduler 名称 (org.quartz.scheduler.instanceName) 前附加上字符串 '_QuartzSchedulerThread' 作为名称。

·org.quartz.scheduler.idelWaitTime

这个属性设置了当 Scheduler 处于空闲时转而再次查询可用 Trigger 时所等待的毫秒数。通常，你无需调整这个参数，除非你正使用 XA 事物，遇到了 Trigger 本该立即触发而发生延迟的问题。

·org.quartz.scheduler.dbFailureRetryInterval

这个属性设置 Scheduler 在检测到 JobStore 到某处的连接(比如到数据库的连接) 断开后，再次尝试连接所等待的毫秒数。这个参数在使用 RamJobStore 无效。

·org.quartz.scheduler.classLoadHelper.class

对于多数健状的应用，所使用的默认值为 org.quartz.simpl.CascadingClassLoadHelper 类，它会依序使用其他的 ClassLoadHelper 类，直到有一个能正常工作为止。你大概没必须为这个属性指定任何其他的类，除非有可能在应用服务器中时。当前所有可能的 ClassLoadHelper 实现可在 org.quartz.simpl 包中找到。

·org.quartz.context.key.SOME_KEY

这个属性用于向 "Scheduler 上下文" 中置入一个 名-值 对表示的字符串值。(见 Scheduler.getContext())。因此，比如设置了 org.quartz.context.key.MyEmail = myemail@somehost.com 就相当于执行了 scheduler.getContext().put ("MyEmail", myemail@somehost.com)

·org.quartz.scheduler.userTransactionURL

它设置了 Quartz 能在哪里定位到应用服务器的 UserTransaction 管理器的 JNDI URL。默认值(未设定的话) 是 java:comp/UserTransaction，这几乎能工作于所有的应用服务器中。Websphere 用户也许需要设置这个属性为 jta/usertransaction。这个属性仅用于 Quartz 配置使用 JobStoreCMT 的情况，并且 org.quartz.scheduler.wrapJobExecutionInUserTransaction 被设定成了 true。

·org.quartz.scheduler.wrapJobExecutionInUserTransaction

如果你要 Quartz 在调用你的 Job 的 execute 之前启动一个 UserTransaction 的话，设置这个属性为 true。这个事物将在 Job 的 execute 方法完成和 JobDataMap(假如是一个 StatefulJob) 更新后提交。默认值为 false。

·org.quartz.scheduler.jobFactory.class

这是所用的 JobFactory 的类名称。默认为 org.quartz.simpl.SimpleJobFactory。你也可以试试 org.quartz.simpl.PropertySettingJobFactory。一个 Job 工厂负责产生 Job 类的实例。SimpleFactory 类是调用 Job 类的 newInstance() 方法。PropertySettingJobFactory 也会调用 newInstance()，但还会使用 JobDataMap 中的内容以反射方式设置 Job Bean 的属性。

[译者 Unmi 本篇后记] 从正式发布《Quartz Job Scheduling Framework 中文版.chm 》之后到现在又快过去四个月的时间了，正如前面提到的那个 CHM 文件确实包含了绝大部份主体的内容，就差最后一个附录：Quartz 配置参考，说来也是个缺憾。耽搁的太久，每天都会发生很多事情，可是这几个月对我来太不平静，家庭的、个人的、工作上的事故接踵而至。既然想起来了，还是着手完成这个事吧，之后会汇入到先前那个 CHM 文件中的。况且也还不时有人提起关于翻译版权的问题，实际上通过了解确有不对，该如何呢？暂顶顶风了。

另外，在此提一下 org.quartz.scheduler.jobFactory，因为它简单的调用 Job 类的 newInstance() 方法来得到 Job 实例，所以你的 Job 要有一个无参构造方法。有一个网友使用 Quartz 在 Scheduler 初始化 Job 时碰到这样的错误：

```
严重: An error occured instantiating job to be executed. job= 'jobDetailGroup1.jobDetail1'
org.quartz.SchedulerException: Problem instantiating class 'steve.InvokeCmdAction$SimpleQuartzJob' [See
nested exception: java.lang.InstantiationException: steve.InvokeCmdAction$SimpleQuartzJob]
    at org.quartz.simpl.SimpleJobFactory.newJob(SimpleJobFactory.java:57)
    at org.quartz.core.JobRunShell.initialize(JobRunShell.java:132)
    at org.quartz.core.QuartzSchedulerThread.run(QuartzSchedulerThread.java:387)
Caused by: java.lang.InstantiationException: steve.InvokeCmdAction$SimpleQuartzJob
    at java.lang.Class.newInstance0(Class.java:340)
    at java.lang.Class.newInstance(Class.java:308)
    at org.quartz.simpl.SimpleJobFactory.newJob(SimpleJobFactory.java:55)
    ... 2 more
```

问题在于 Quartz 的 SimpleFactory 无法实例化 'steve.InvokeCmdAction$SimpleQuartzJob' 这个例，看这个类名，带个 $ 符号，很显然是写 InvokeCmdAction.java 文件中的，后来他把 SimpleQuartzJob 单独写在 SimpleQuartzJob.java 文件中问题即得到解决。如若有兴趣的话，定制自己的 JobFactory 配置给 org.quartz.scheduler.jobFactory 属性，那对于 'steve.InvokeCmdAction$SimpleQuartzJob' Job 类也是可以成功实例化的。

上一页 我要评论 下一页

附录 A. Quartz 配置参考（第二部分）

二. 配置 Quartz ThreadPool

表 A.2 列出了配置 Quartz ThreadPool 可用的属性。只有少些属性是必须的，剩下的都有合理的默认值。

表 A.2. 配置 Quartz ThreadPool 的属性

名称	必须	类型	默认值
org.quartz.threadPool.class	是	String	null
org.quartz.threadPool.threadCount	是	Integer	-1
org.quartz.threadPool.threadPriority	否	Integer	5
org.quartz.threadPool.makeThreadsDaemons	否	boolean	false
org.quartz.threadPool.threadsInheritGroup-OfInitializingThread	否	boolean	true
org.quartz.threadPool.threadsInheritContext-ClassLoaderOfInitializingThread	否	boolean	false

·org.quartz.threadPool.class

用于指定你想要使用的 ThreadPool 实现的类名。Quartz 自带的 ThreadPool 实现是 org.quartz.simpl.SimpleThreadPool，它在大多数情况下是够用的。它有着十分简单的行为并得到过很好的测试。它提供了一个固定大小的线程池，且线程在 Scheduler 的生命期内是 "活着" 的。

·org.quartz.threadPool.threadCount

这个属性指定了可用于并发执行 Job 的线程的数量。它可设置为一个 1 和 100 之间的正整数。大于 100 的值也是允许的但却不切实际。假如你一天中只有少数几个 Job 要触发，那一个线程就够了。如果你有数以万计的 Job，且每分钟有许多被触发，那么你或许要的线程数就会是 50 或者 100。

·org.quartz.threadPool.threadPriority

这个属性用于指定工作者线程运行的优先级。这个值可是介于 1 和 10 之间的整数。默认为 5，也就是 Thread.NORM_PRIORITY。

·org.quartz.threadPool.makeThreadsDaemons

设置为 true 会使得池中的线程被创建为守护线程。默认为 false。

·org.quartz.threadPool.threadsInheritGroupOfInitializingThread

假如你希望新线程继承自父线程组，就设置为 true。默认为 true。

·org.quartz.threadPool.threadInheritContextClassLoaderOfInitializingThread

假如你希望新线程继承自创建父线程的 Classloader，就设置为 true。默认为 false。

三. 配置 Quartz 监听器

在属性文件中配置 Quartz 监听器涉及到指定名称、类名和任何其他可设置给实例的属性。监听器类必须有一个无参的构造方法；属性可以反射的方法设置。仅支持原始数据类型（包括 String）。

·配置 JobListener

要配置一个 JobListener，你必须指定一个实现了监听接口的类

```
org.quartz.jobListener.NAME.class = com.foo.SomeListenerClass
```

NAME 可以是你想要提供给监听器的任何名称，但它应当与调用这个类的 getName() 的返回值相匹配。你能提供属性给监听器，它们会以反射方式设置给监听器的：

```
org.quartz.jobListener.NAME.propName = proValue
org.quartz.jobListener.NAME.prop2Name = prop2Value
```

四. 配置 TriggerListener

配置一个 TriggerListener 的过程很类似于 JobListener 的配置。事实上，除了 jobListener 换成了 TriggerListener 外，就是一样的。

```
org.quartz.triggerListener.NAME.class = com.foo.SomeListenerClass
```

你也可以相同的方式提供属性：

```
org.quartz.triggerListener.NAME.propName =  propValue
org.quartz.triggerListener.NAME.prop2Name = prop2Value
```

关于 Quartz 监听器更多的信息，见第七章，"实现 Quartz 监听器"。

五. 配置 Quartz 插件

配置 Quartz 插件的过程很类似于前面描述的配置监听器。当你有一个实现了 SchedulerPlugin 接口的类，你就可以通过加入如下行来配置插件：

```
org.quartz.plugin.NAME.class = com.foo.MyPluginClass
```

这里，NAME 是你指派给插件的名称。你可以提供像下面那样给插件实例传递参数：

```
org.quartz.plugin.NAME.propName = propValue
org.quartz.plugin.NAME.prop2Name = prop2Value
```

设置参数时的 NAME 必须与指派给插件的 NAME 相匹配。有关 Quartz 插件的更多信息，见第八章，"使用 Quartz 插件"。

附录 A. Quartz 配置参考（第三部分）

六. 配置 Quartz RMI 选项

当通过 RMI 使用 Quartz 启动一个 Quartz 实例时，你需要把它配置为经由 RMI "导出" 服务。然后你就能创建客户端，配置它们的 Quartz Scheduler 作为 "代理" 工作而连接到服务端来。表 A.3 列出了可用的 RMI 设定。

表 A.3. Quartz 使用 RMI 时的属性

名称	必须	类型	默认值
org.quartz.scheduler.rmi.export	否	Boolean	false
org.quartz.scheduler.rmi.registryHost	否	String	localhost
org.quartz.scheduler.rmi.registryPort	否	Integer	1099
org.quartz.scheduler.rmi.createRegistry	否	String	never
org.quartz.scheduler.rmi.serverPort	否	Integer	Random
org.quartz.scheduler.rmi.proxy	否	Boolean	false

·org.quartz.scheduler.rmi.export

假如你想要 Quartz Scheduler 经由 RMI 服务器被导出，就设置此项为 true。

·org.quartz.scheduler.rmi.registryHost

配置在哪个主机上能找到 RMI 服务。默认为 localhost。

·org.quartz.scheduler.rmi.registryPort

这个配置 RMI 注册表所监听的端口。默认为 1099。

·org.quartz.scheduler.rmi.createRegistry

要根据你想要 Quartz 如何创建一个 RMI 注册服务来设置该属性。假如你不希望 Quartz 创建一个注册服务 (例如，如果你已经有一个在运行的外部注册服务) 就使用 false 或者 never。要是你想要 Quartz 首先尝试去使用一个已存在的注册服务，失败时才创建时就用 true 或者 as_needed。倘若你要 Quartz 先尝试创建一个注册服务，不成功就使用一个已存在的注册服务的话就用 always。一旦注册服务创建好了，它将会被绑定到 org.quartz.scheduler.rmi.registryPort 属性指指定的端口上去，而此时 org.quartz.schedu.rmi.registryHost 应该是 localhost。

·org.quartz.scheduler.rmi.serverPort

这个指示了 Quartz Scheduler 服务将被绑定和监听连接的端口号。默认时，RMI 服务会随机选择一个端口号用来把 Scheduler 绑定到 RMI 注册服务。

·org.quartz.scheduler.rmi.proxy

如果你要连接到一个远程的 Scheduler 服务就设置本属性为 true。同时你必须指定 RMI 注册进程的主机和端口号，通常是 localhost 和 1099 端口。不要在同一个配置文件中指定 org.quartz.scheduler.rmi.export 和 org.quartz.scheduler.rmi.proxy 都为 true；假如你那样做的话，export 选项将会忽略。如果你不打算通过 RMI 来使用 Quartz 的话，export 和 proxy 属性同时设为 false 当然没问题了。

有关通过 RMI 使用 Quartz 的更多信息，见第十章，"J2EE 中使用 Quartz"。

七. 配置 JobStore 选项

你要提供了一个实现了 JobStore 接口的类的全限名来配置 JobStore。例如，下面行告诉了 Quartz 程序使用

RAMJobStore：

org.quartz.jobStore.class = org.quartz.simpl.RAMJobStore

除了 RAMJobStore 外，Quartz 还提供了两种类型的 JDBC JobStore：

·JobStoreTX

·JobStoreCMT

附录 A. Quartz 配置参考（第四部分）

八. 配置 JobStoreTX JobStore

你可以像下面那样设定类名来选择 JobStoreTX 类：

```
org.quartz.jobStore.class = org.quartz.impl.jdbcjobstore.JobStoreTX
```

表 A.4 列出了配置 Quartz JobStore 的可用属性。只有少数几个属性是必须的，而且其他的都有合理的默认值。

表 A.4. 配置 Quartz JobStore 可用的属性

名称	必须	类型	默认值
org.quartz.jobStore.driverDelegateClass	是	String	null
org.quartz.jobStore.dataSource	是	String	null
org.quartz.jobStore.tablePrefix	否	String	QRTZ_
org.quartz.jobStore.useProperties	否	Boolean	false
org.quartz.jobStore.misfireThreshold	否	Integer	60000
org.quartz.jobStore.isClustered	否	Boolean	false
org.quartz.jobStore.clusterCheckinInterval	否	Long	15000
org.quartz.jobStore.maxMisfiresToHandleAtATime	否	Integer	20
org.quartz.jobStore.dontSetAutoCommitFalse	否	Boolean	false
org.quartz.jobStore.selectWithLockSQL	否	String	"SELECT * FROM {0} LOCKS WHERE LOCK_NAME = ? FOR UPDATE"
org.quartz.jobStore.txIsolationLevelSerializable	否	Boolean	false

·org.quartz.jobStore.driverDelegateClass

Quartz 通过一个代理可以使用大部分流行的数据库平台。这里是 org.quartz.jobStore.driverDelegateClass 属性允许的值：

- ·org.quartz.impl.jdbcjobstore.StdJDBCDelegate
- ·org.quartz.impl.jdbcjobstore.MSSQLDelegate
- ·org.quartz.impl.jdbcjobstore.PostgreSQLDelegate
- ·org.quartz.impl.jdbcjobstore.WebLogicDelegate (for WebLogic drivers)
- ·org.quartz.impl.jdbcjobstore.oracle.OracleDelegate
- ·org.quartz.impl.jdbcjobstore.oracle.WebLogicOracleDelegate
- ·org.quartz.impl.jdbcjobstore.oracle.weblogic.WebLogicOracleDelegate
- ·org.quartz.impl.jdbcjobstore.CloudscapeDelegate
- ·org.quartz.impl.jdbcjobstore.DB2v6Delegate
- ·org.quartz.impl.jdbcjobstore.DB2v7Delegate
- ·org.quartz.impl.jdbcjobstore.HSQLDBDelegate
- ·org.quartz.impl.jdbcjobstore.PointbaseDelegate

·org.quartz.jobStore.dataSource

这个属性的值必须是本附录接下来的 DataSource 配置中定义的 DataSource 的名称。

·org.quartz.jobStore.tablePrefix

表前缀属性是作为在你的数据库中创建 Quartz 表的表名的前缀。如果你使用不同的表前缀，你就能够在同一个数据库中拥有多套 Quartz 表。

·org.quartz.jobStore.useProperties

"使用属性" 设置指示着 JDBC JobStore 所有在 JobDataMap 中的值会是字符串，因此，它们可以名-值对来存储，而无需把更复杂的对象以序列化的形式存储在 BLOB 列中。这样做很有好处的，因为避免了伴随着序列化非字符串类到 BLOB 时产生的类版本问题。

·org.quartz.jobStore.misfireThreshold

设定这个属性为一个毫秒数，Scheduler 允许一个 Triiger 在超过它的下次触发时多少毫秒才算是错过触发。默认值是 60000 (60 秒)。

·org.quartz.jobStore.isClustered

设置为 true 来打开集群特性。假如你正使用多个 Quartz 实例且用的是同一套数据库表的话，这个属性必须设置为 true。

·org.quartz.jobStore.clusterCheckinInterval

设置当前实例检查集群中的其他实例的频度 (毫秒)。这个值会影响侦测失败实例的灵敏性。

·org.quartz.jobStore.maxMisfiresToHandleAtATime

这是设定 JobStore 同一时刻能处理错过触发 Trigger 的最大数量。一次同时处理太多(超过数十个)的话，会导致数据库表被锁定过长的时间，从而影响到了触发其他的(还未错过触发) Trigger 的性能。

·org.quartz.jobStore.dontSetAutoCommitFalse

设置这一参数为 true 是告诉 Quartz 别调用从 DataSource 处获取的连接的 setAutoCommit(false) 方法。这在一些情况下是有帮助的，比如可能你有某个驱动在连接关闭的时候是不允许调用该方法的。这个属性默认为 false，因为多数驱动是需要调用 setAutoCommit(false) 方法的。

·org.quartz.jobStore.selectWithLockSQL

这必须为一个 SQL 字符串，用来从 LOCKS 表中查询一条记录并在其中加锁。倘若未予设置，默认为 SELECT * FROM {0}LOCKS WHERE LOCK_NAME = ? FOR UPDATE, 这可在多数数据库中正常工作。{0} 会在运行时替代为你早先配置的 TABLE_PREFIX 属性。

·org.quartz.jobStore.txIsolationLevelSerializable

值为 true 是告诉 Quartz (当使用 JobStoreTX 或是 CMT 的时候) 去调用 JDBC 连接的 setTransactionIsolation (Connection.TRANSACTION_SERIALIZABLE) 方法。这对于避免某些数据库在高负载及长事物发生时的锁超时是有帮助的。

附录 A. Quartz 配置参考（第五部分）

九. 配置 JobStoreCMT

JobStoreCMT 提供了另一类型的 JobStore，它能工作于一个关系型数据库之下。你能通过设置 org.quartz.jobStore.class 属性来选用 JobStoreCMT：

org.quartz.jobStore.class = org.quartz.impl.jdbcjobstore.JobStoreCMT

当使用 JobStoreCMT 时，可设置一些附加属性，它们是表 A.5 所示的。

表 A.5. 配置 Quartz JobStoreCMT 的属性

名称	必须	类型	默认值
org.quartz.jobStore.nonManagedTXDataSource	是	String	null
org.quartz.jobStore.dontSetNonManagedTX-ConnectionAutoCommitFalse	否	Boolean	false
org.quartz.jobStore.txIsolationLevelReadCommited	否	Boolean	false

·org.quartz.jobStore.nonManagedTXDataSource

JobStoreCMT 需要一个(即第二个) 数据源，其中包含的连接不作为容器管理事物的一部分。这个属性值必须是在属性配置文件中定义的某一个 DataSource 的名称。这个数据源必须包含非 CMT 的连接，换句话说就是，其中的连接的 commit() 和 rollback() 方法能够由 Quartz 直接且合法的调用。

·org.quartz.jobStore.dontSetNonManagedTXConnectionAutoCommitFalse

这个属性意义同 org.quartz.jobStore.dontSetAutoCommitFalse，只是它作用于 nonManagedTXDataSource。

·org.quartz.jobStore.txIsolationLevelReadCommited

当设置为 true，这一属性就告诉 Quartz 去调用不受管理 JDBC 连接的 setTransactionIsolation (Connection.TRANSACTION_READ_UNCOMMITED) 方法。这有助于防止某些数据库(如 DB2) 在高负载及长事物发生锁超时的现象。

关于 JDBC JobStore 更多的信息，参见第六章，"JobStore 和持久化"

附录 A. Quartz 配置参考（第六部分）

十. 配置 Quartz 数据源

如果你在使用 JDBC JobStore，就需定义要用到的数据源。而如果使用的是 JobStoreCMT，你实际上需要定义两个数据源。数据源可由三种方式来配置：

·在 quartz.properties 文件中指定连接池属性，这样 Quartz 能亲自创建数据源。

·指定应用服务器管理的数据源所在 JNDI 的位置，Quartz 直接使用它。

·还可使用自定义的 org.quartz.utils.ConnectionProvider 实现类。

每一个你定义的 datasource 都必须给予一个名称，并且你为它们配置的属性也必须包含这个名称。datasource 的 NAME 可以是你任何想要的；它没有特别的意思，只是用来指定给 JDBC JobStore 时标识 datasource。

当使用 quartz.properties 来定义连接池的所有属性来配置 datasource 时，表 A.6 是可用的属性。

表 A.6. 配置 Quartz Datasource 的属性

名称	必须	类型	默认值
org.quartz.dataSource.NAME.driver	是	String	null
org.quartz.dataSource.NAME.URL	是	String	null
org.quartz.dataSource.NAME.user	否	String	""
org.quartz.dataSource.NAME.password	否	String	""
org.quartz.dataSource.NAME.maxConnections	否	Integer	10
org.quartz.dataSource.NAME.validationQuery	否	String	null

·org.quartz.dataSource.NAME.driver

必须是你的数据库的 JDBC 驱动的 Java 类名。

·org.quartz.dataSource.NAME.URL

这是连接到你的数据库的 URL(主机、端口等)

·org.quartz.dataSource.NAME.user

用来连接到你的数据库的用户名。

·org.quartz.dataSource.NAME.password

用来连接到你的数据库的密码。

·org.quartz.dataSource.NAME.maxConnections

DataSource 创建的连接池中的最大连接数。

·org.quartz.dataSource.NAME.validationQuery

这是一个可选的 SQL 查询语句，DataSource 用它来检测失败/无用的连接。例如，Oracle 用户可选用 select table_name from user_tables，这个查询永不会失败，除非连接实际是无用的。

当你使用定义在应用服务器中的 Datasource 时，可用的属性如表 A.7 所列。

表 A.7. 配置 Quartz 使用应用服务器中中 Datasource 时的属性

名称	必须	类型	默认值
org.quartz.dataSource.NAME.jndiURL	是	String	null
org.quartz.dataSource.NAME.java.naming.factory.initial	否	String	null
org.quartz.dataSource.NAME.java.naming.provider.url	否	String	null
org.quartz.dataSource.NAME.java.naming.security.principal	否	String	null
org.quartz.dataSource.NAME.java.naming.security.credentials	否	String	null

·org.quartz.dataSource.NAME.jndiURL

这是受容器管理的 DataSource 的 JNDI URL。

·org.quartz.dataSource.NAME.java.naming.factory.initial

可选，你希望使用的 JNDI InitialContextFactory 类名。

·org.quartz.dataSource.NAME.java.naming.provider.url

可选，用来连接到 JNDI 上下文的 URL。

·org.quartz.dataSource.NAME.java.naming.security.principal

可选，连接到 JNDI 上下文的用户名。

·org.quartz.dataSource.NAME.java.naming.security.credentials

可选，连接到 JNDI 上下文的用户凭证。

十一. 使用自定义的 ConnectionProvider 配置数据源

自 Quartz 1.5.1 开始，你可以创建一个自定义的 ConnectionProvider，并在属性文件中提供类名来配置 Quartz 使用它。

```
org.quartz.dataSource.myConnProvider.connectionProvider.class = com.foo.MyConnectionProvider
```

初始化该类后，Quartz 就能自动以 Bean 风格来设置实例的属性。

```
org.quartz.dataSource.myConnProvider.someStringProperty = someValue
org.quartz.dataSource.myConnProvider.someIntProperty = 5
```