定时器(Quartz)使用说明

本文目前主要包括如下几个部分：

Quartz功能简介 ：介绍Quartz的特点及概念。

使用Quartz的常见问题 ：很多用户在使用过程中遇到常见问题的解答。

快速开始：让读者尽快掌握Quartz开发。

Quartz官方开发指南：通过一些列的课程来指导如何使用Quartz.

Quartz 功能简介

Quartz 特点：

- Quartz 能嵌入到任何独立的应用中运行。
- Quartz 能在应用服务器或者 Servlet 容器中实例化，并且能够参与 XA 事务。
- Quartz 能够以独立的方式运行（在它自己的 Java 虚拟机中），可以通过 RMI 使用 Quartz。
- Quartz 可以被实例化为独立程序的集群（有负载均衡和容错能力）。

Job Scheduling（任务日程安排）

任务在给定的触发器（Trigger）触发时执行。触发器可以通过几乎以下所有形式的组合方式进行创建：

- 在一天中的任意时刻（可以精确到毫秒）。
- 一周中特定的一些天。
- 一个月中特定的一些天。
- 一年中特定的一些天
- 不在日历列表中注册的一些天（比如节假日）。
- 循环特定的次数。
- 循环到特定的时间。
- 无限循环。
- 按照一定的时间间隔循环。

“任务”由创建者给定名字，并且可以加入到给定名称的“任务组”中。为了简化触

发器在日程中的管理，“触发器”也可以被给定名字和分组。任务只能加入到日程中一次，但是可以为其注册多个触发器。在 J2EE 环境中，任务可以作为分布(XA)事务的一部分执行。

Job Execution（任务执行）

- 任务是任何实现简单 Job 接口的 Java 类，这样开发者能够执行任何完成他们工作的任务。
- 任务类的实例可以由 Quartz 实例化，也可以由你的程序框架实例化。

当触发器被触发时，日程管理器将会通知某个或者多个实现了 JobListener 或 TriggerListener 的对象（监听器可以是简单的 Java 对象，或者 EJBs,或者 JMS 消息发布器，等等）。这些监听器在任务执行完毕后也会接到通知。

- 任务被完成后，他们会返回一个“任务完成码（JobCompletionCode）”，这个“任务完成码”告知日程管理器任务执行的结果是成功还是失败。日程管理器会根据成功或者失败码来采取措施，比如：立即重新执行任务。

Job Persistence（任务持久化）

- Quartz 设计中包括了一个 JobStore 接口，这样，实现这个接口的 Job 类可以以多种机制实现 Job 的存储。
- 通过使用 JDBCJobStore,所有的 Jobs 和 Triggers 被配置为“non-volatile”(不轻快)的方式。即，通过 JDBC 存储在关系数据库中。
- 通过使用 RAMJobStore，所有 Jobs 和 Triggers 被存储在 RAM。因此，在程序执行中没有被持久化，但这种方式的优点就是不需要外部数据库。

Transactions（事务）

- Quartz 通过 JobStoreCMT（JDBCJobStore 的一个子类）可参与 JTA 事务。
- Quartz 可以管理 JTA 事务（开始或者提交事务）。

Clustering（集群）

- Fail-over.（容错）
- Load balancing.（负载均衡）

Listeners & Plug-Ins（监听器及插件）

- 应用可以通过实现一个或者多个监听器接口来实现捕捉日程事件，以监视或控制任务/触发器的行为。
- 可以通过插件的机制来扩展 Quartz 的功能。例如:记录任务执行历史的日志，或者从文件中载入任务和触发器的定义。

- Quartz 自带了一些“factory built（内建）”的插件和监听器。

常见问题

一般问题

- Quartz是什么？
- 为什么不使用java.util.Timer?
- 如何build Quartz源码？

杂项问题

- Quartz可以运行多少任务？
- 通过RMI使用Quartz存在一些问题。

关于 Jobs 的问题

- 如何能够控制Jobs的实例化。
- 如何避免一个任务在完成后被移（删）除？
- 如何避免一个Job被并发激活？
- 如何停止一个正在执行的任务？

关于触发器的问题：

- 如何串行任务的执行？或者，如何能够创建一个工作流程?
- 为什么触发器没有被触发？
- 夏令时和触发器

关于 JDBCJobStore 的问题

- 如何提高JDBC-JobStore的性能？
- 如果数据服务器重新启动，DB连接不能够正确恢复

关于事务的问题

- 使用JobStoreCMT并且发现了死锁，该如何做？

General Questions 一般问题

Quartz 是什么？

Quartz 是一个任务日程管理系统，这个系统可以与任何其他软件系统集成或者一起使用。术语“日程进度管理器”可能对于不同的人有不同的理解。当你阅读这个指南之后，

你会对这个术语有固定的理解。简而言之，“任务进度管理器”就是一个在预先确定（被纳入日程）的时间到达时，负责执行（或者通知）其他软件组件的系统。

为了达到预想目的，或者是能够写出与工程最“自然”衔接的软件代码，Quartz 相地灵活，并且包括了多个用法范例，可以单独运用这些范例或者组合运用这些范例。

Quartz 相当“轻量”，并且需要非常少的步骤/配置，如果需求比较基本，Quartz 确实非常容易使用。

Quartz 具有容错性，并且可以在你系统重起的时候持久化（记住）被纳入日程的任务。

虽然 Quartz 对于按照简单地给定日程运行的系统时非常有用，但是，当你学会如何使用它来驱动你应用中的商务过程，那么你才会认识到它的全部潜能。

从软件组件的角度来看，Quartz 是什么？

Quartz 用一个小 Java 库发布文件（.jar 文件），这个库文件包含了所有 Quartz 核心功能。这些功能的主要接口(API)是 Scheduler 接口。它提供了简单的操作，例如：将任务纳入日程或者从日程中取消，开始/停止/暂停日程进度。

如果你想将软件组件的执行纳入到日程中，它们只需简单地实现 Job 接口，这个接口有一个 execute()方法。如果希望在日程安排的时间到达时通知组件，那么这些组件应实现 TriggerListener 或者 JobListener 接口。

Quartz 主过程可以在应用中启动或者运行，也可以作为一个独立的应用（带有 RMI 接口），或者在一个 J2EE 应用服务器中运行，并且可作为其它 J2EE 组件的一种引用资源。

为什么不使用 java.util.Timer?

从 JDK1.3 开始，Java 有了内建的定时器功能，即，通过 java.util.Timer 和 java.util.TimerTask 来实现，为什么有人用 Quartz 而不用这些标准特性呢？有很多原因，下面是其中的一些：

1. Java 定时器没有持久化机制。
2. Java 定时器的日程管理不够灵活（只能设置开始时间、重复的间隔，设置特定的日期、时间等）
3. Java 定时器没有使用线程池(每个 Java 定时器使用一个线程)
4. Java 定时器没有切实的管理方案，你不得不自己完成存储、组织、恢复任务的措施。

…当然，对于某些简单的应用来说，这些特性可能不那么重要，在这种情况下，不使用 Quartz 也是正确的选择。

如何 build Quartz 源码？

尽管 Quartz 以“预先编译”的形式打包。很多人可能想做他们自己的改变或者 buil 从 CVS 上得到的最新的“未发布”的 Quartz 版本。阅读随同 Quartz 一起打包的 "README.TXT"文件就知道怎么做。

Miscellaneous Questions 杂项问题

Quartz 可以运行多少任务？

这是一个比较难以回答的问题，答案基本上是“看它依赖于什么环境”。

你可能不喜欢这个答案，这里有一些关于它所依赖环境的信息。

首先，JobStore 对性能有重要的影响.基于 RAM 的 JobStore 要比基于 JDBC 的 JobStore 快的多。基于 JDBC 的 JobStore 的速度几乎完全取决于对数据库连接的速度，以及使用的数据系统以及数据库运行的硬件。Quartz 本身实际上只做很少的处理，差不多所有的时间都花费在数据库上。当然，RAMJobStore 对于有多少“任务”或者“触发器”可以存储是有数量限制的，因为，内存的数量要比数据库的硬盘空间小的多。你可以参考问题“如何提高 JDBC-JobStore 的性能？”。

因此,对于 Quartz 能存储和监控的触发器和任务的数量限制因素时间上是 JobStore 有多少可用的存储空间的数量。（内存数量或者是硬盘数量）。

现在，除了“能存储多少？”的问题之外，就是“Quartz 能同时运行多少个任务？”

能够使 Quartz 本身运行变慢的就是使用大量的监听器（TriggerListeners, JobListeners, 和 SchedulerListeners），除了任务本身实际的运行时间外，花费在每个监听器上的时间会直接记入“处理”任务的执行时间上。这不意味着因此而害怕使用监听器，而是说要明智地使用监听器。如果能够使用特定的监听器就不要创建大量“全局”监听器，也不要使用监听器作“耗时”的工作，除非必须做。另外，也要留心，很多插件实际上也是监听器（例如：“history”插件）。

同时能够运行多少个任务受限于线程池的大小。如果池中有 5 个线程，则同时可运行的任务不超过 5 个。使用大量线程时要小心，因为 JVM,操作系统和 CPU 处理大量线程时很耗时，并且，线程管理也会导致性能下降。在多数情况下，性能开始下降是因为使用多达数百个线程。如果在应用服务器中运行程序，要留心应用服务器本身已经创建了不少的线程。

除了这些因素外，还同要执行的任务有关，如果任务需要花费很长时间才能完成它们的工作，并且（或者）它们的工作很消耗 CPU,那么很明显不要同时运行多个任务，也不要在一个时间段内运行过多这样的任务。

最后，如果不能通过一个 Quartz 实例获得足够的性能，你可以将多个 Quartz 实例（放在不同的机器上）进行负载均衡。每个 Quartz 都按照“先到先服务”的原则在共享的数据库之外运行任务。

目前为止，你已经知道有关“多少个”的答案很多知识，而且还是没有给出一个具体的数量，并且由于以上提及的各种因素，我也不愿意给出。因此，只是说，很多地方使用 Quartz 管理成百上千的任务和触发器，并且在任何给定的时刻，都有很多任务在执行，并且这些不包括负载均衡。有了这个印象，很多人都会对使用 Quartz 充满信心。

通过 RMI 使用 Quartz 存在一些问题。

RMI 可能会有一些小的问题，尤其当你不理解 RMI 如何加载类的时候。强烈推荐阅读所有关 RMI 的 JavaDOC,并且强烈建议阅读一下参考文档，这些参考资料由一个好心的 Quartz 用户整理（Mike Curwen）。

关于 RMI 的出色的描述和 codebase：

http://www.kedwards.com/jini/codebase.html其中的一个重点就是了解用于客户端的”codebase”。

有关安全管理的快讯：
http://gethelp.devx.com/techtips/java_pro/10MinuteSolutions/10min0500.asp.

来自于Java API 文档，阅读关于RMISecurityManager 的文档
http://java.sun.com/j2se/1.3/docs/api/java/rmi/RMISecurityManager.html

API 中需要重点掌握的就是：

如果没有安全管理器的设置，RMI 的类加载器不会从远程位置下载任何类。

Questions About Jobs

如何能够控制 Jobs 的实例化?

看有关 org.quartz.spi.JobFactory 和 org.quartz.Scheduler.setJobFactory() 方法的说明。

如何避免一个任务在完成后被移（删）除？

设置属性JobDetail.setDurability(true)，这将指示*Quartz*在任务变为“孤儿”（当任务不在被任何触发器所引用时）时不要删除任务。

如何避免一个 Job 被并发激活？

使 Job 类实现 StatefulJob 接口，而不是 Job 接口。阅读 JavaDOC 获得有关 StatefulJob 接口的更多信息。

如何停止一个正在执行的任务？

参见*org.quartz.InterruptableJob*接口和*Scheduler.interrupt(String, String)方法。*

Questions About Triggers

如何串行任务的执行？或者，如何能够创建一个工作流程。

目前 Quartz 没有“直接”或者“便捷”的方法能够串行触发器。但是有几种方法可以完成这些工作而不需要太多的工作。下面是这些方法的介绍：

一个方法就是使用监听器（例如：TriggerListener，JobListener 或者 SchedulerListener）来通知任务/触发器的执行完成，并且立即触发新的触发器。这个方法会有一些小麻烦，因为你不得不通知接下来要执行任务的监听器，并且你可能会担心这个信息的持久化问题。

另一个方法就是建立一个任务，这个任务将下一个要执行任务的名字放在这个任务的 JobDataMap 中，当这个任务完成时（它的 exceute()方法中的最后一步）就会产生执行下一个任务的日程安排。有些人使用这个方法并且取得成功。大多数创建一个（虚）基类，在这个类中，一个任务知道如何通过特定的键来获得 JobDataMap 以外的任务名和组，并且包含了将已识别的任务纳入日程的代码，然后，简单地扩展这个类使其包含任务需要做的附加工作。

在将来，Quartz 将提供一个非常明确的方法去完成这些功能，但目前为止，你只能使用以上这些方法中的一种，或者考虑更适合你的方法。

为什么触发器没有被触发？

最常见的原因就是没有调用*Scheduler.start()，这个方法告知*scheduler开始触发触发器。

另外一个常见的原因就是触发器或者触发器组被暂停了。

夏令时和触发器

CronTrigger 和 SimpleTrigger 分别用它们自己的方式来处理夏令时，每种方式对与触发器类型来说都很直观。

首先，回顾一下什么是夏令时，请阅读这个资源：
http://webexhibits.org/daylightsaving/g.html 。

（夏令时开始的时刻，时钟要向前（未来）拨一个小时，夏令时结束的时刻，时钟向后（过去）拨一个小时）

有的读者可能没有意识到不同的国家和地区的夏令时规则是不同的。例如：2005 年美国夏令时开始于 4 月 3 日，而埃及则开始于 4 月 29 日。不仅夏令时切换日期在不同的地区不同，时间也不同，有些地方于凌晨两点切换，而其他地方则于凌晨 1：00 切换，其他的地方则于凌晨 3 点切换，并且也有的地方就正好在午夜切换。

SimpleTrigger 允许你每隔若干毫秒来触发纳入进度的任务。因此，对于夏令时来说，根本不需要做任何特殊的处理来“保持进度”。它只是简单地保持每隔若干毫秒来触发一次，无论你的 SimpleTrigger 每隔 10 秒触发一次还是每隔 15 分钟触发一次，还是每隔 24 小时触发一次。但是这隐含了一个混乱，对于那些每隔 12 个小时触发一次的 SimpleTrigger 来说，在实行夏令时之前，任务是在凌晨 3 点及下午 3 点触发，但是在执行夏令时后，人物是在凌晨 4 点及下午 4 点触发。这不是 Bug,触发器仍然保持了每隔若干毫秒触发一次，只是时间的“名字”被人为地强行改变了。

CronTrigger 能在特定 “格林日历”时刻触发纳入进程的任务，因此，如果创建一个在每天上午 10 点触发的触发器，那么，在夏令时执行之前，系统将继续如此运作。但是，取决于是春季还是秋季夏令时，因为对于特定的星期日，从星期六上午 10 点到星期日上午 10 点之间的时间间隔将不是 24 小时，而可能是 23 或者 25 个小时。

当使用夏令时的时候，对于 CronTrigger 来说，有一个特别的“时间点”用户必须理解。这就是你应注意考虑在午夜和凌晨 3 点（临界的窗口时间取决于所在的地区）之间触发的任务，其原因就是由于触发器的进度以及特定的夏令时时间造成。触发器可能被忽略或者一两个小时都没有触发。例如：假设你在美国，这里的夏令时发生在凌晨两点（时钟会调整为凌晨 3：00）。如果 CronTrrigger 触发在那天的凌晨 2:15 分，那么，在夏令时开始的那天，触发器将被忽略。因为，那天凌晨 2；15 这个事件根本不存在。如果 CronTrigger 每 15 分钟触发一次，那么在夏令时结束的那天，你将会有一个小时的时间没有触发发生。因为当凌晨 2 点到达时，时间将重新变成凌晨 1：00，由于凌晨一点钟内的所有触发都已经触发过了，所以，这个时间不会有触发发生，直到时间又到了凌晨 2：00，这时触发才重新开始。

总之，如果你记住下面的两条规则，则会感觉良好并且很容易记忆：

- SimpleTrigger 总是每隔若干秒触发，而同夏令时没有关系。
- CronTrigger 总是在给定的时间出发然后计算它下次触发的时间。如果在给定的日期内没有该时间，则触发器将会被忽略，如果在给定的日期内该时间发生了两次，它只触发一次。因为是在第一次触发发生后计算当天下次触发的时间。

关于 JDBCJobStore 的问题

如何提高 JDBC-JobStore 的性能？

关于加速 JDBC-JobStore 的听闻很少。只有一个比较特别。

首先，很明显的，但不是很特别：

- 购买运行较好（较快）的用于连接 Quartz 的计算机和运行数据库的计算机之间网络设备。
- 购买较好的（强大的）用于运行数据库的计算机。
- 购买较好的关系型数据库。

另外就是比较简单，但是很有效的方法：为 Quartz 表建立索引。

大多数数据库系统都自动按照主键字段自动建立索引，很多数据库也自动按照外键建立索引。确保作了这些工作，或者对每个表的所有键字段手工建立索引。

下一步，手工增加额外的索引：最重要的索引就是 TRIGGER 表的"next_fire_time"字段和"state"字段上建立的索引。最后（不是很重要），为 FIRED_TRIGGERS 的每个字段建立索引。

创建表索引

```
create index idx_qrtz_t_next_fire_time on qrtz_triggers(NEXT_FIRE_TIME);
create index idx_qrtz_t_state on qrtz_triggers(TRIGGER_STATE);
create index idx_qrtz_t_nf_st on qrtz_triggers(TRIGGER_STATE,NEXT_FIRE_TIME);
create index idx_qrtz_ft_trig_name on qrtz_fired_triggers(TRIGGER_NAME);
create index idx_qrtz_ft_trig_group on qrtz_fired_triggers(TRIGGER_GROUP);
create index idx_qrtz_ft_trig_name on qrtz_fired_triggers(TRIGGER_NAME);
create index idx_qrtz_ft_trig_n_g on
qrtz_fired_triggers(TRIGGER_NAME,TRIGGER_GROUP);
create index idx_qrtz_ft_trig_inst_name on
qrtz_fired_triggers(INSTANCE_NAME);
create index idx_qrtz_ft_job_name on qrtz_fired_triggers(JOB_NAME);
create index idx_qrtz_ft_job_group on qrtz_fired_triggers(JOB_GROUP);
```

如果数据服务器重新启动，DB 连接不能够正确恢复

如果你已经使用 Quartz 创建了数据源连接（通过在 quartz 属性文件中定义连接参数），确保使用一个连接验证的查询语句，例如：

连接验证查询语句

```
org.quartz.dataSource.myDS.validationQuery=select 0 from dual
```

这个特定的查询语句只是针对 Oracle。对于其他的数据库，可能要考虑一个只要在连接正常的时候就能有效执行的查询语句。

如果数据源被应用服务器所管理，确保数据源被配置成能够检查连接失败的方式。

关于事务的问题

使用 JobStoreCMT 并且发现了死锁，该如何做？

JobStoreCMT 是一个多人使用的、大量负载的重量级应用。确信它是没有导致死锁的 Bug。但是，我们不时地收到有关死锁的抱怨。到目前为止，这些问题都被证明为“用户错误”。因此，如果你遇到了死锁，下面列表列出了你需要的检查项：

- 当一个 TX 执行较长的时间时，数据库错误地检测为死锁。确保你对表建立了索引（参见提高 JDBCJobStore 性能）。
- 确保数据源中有至少线程池中线程数量+2 的连接。
- 确保为使用 Quartz 配置了管理数据源和非管理数据源。
- 确保所有使用 Scheduler 接口完成的工作都在一个事务中。完成这些需要在一个 TX 被设置为"Required"或者"Container"的 SessionBean 中使用 Scheduler，或者在一个有相同设置的 MessageDrivenBean 中使用 Scheduler。最后，自己开启一个 UserTransaction，并且在工作做完时提交它。
- 如果在任务 execute()方法中使用了 Scheduler，确保通过 UserTransaction 或者设置 Quartz 的属性："org.quartz.scheduler.wrapJobExecutionInUserTransaction=true"来使事务在进行中。

快速开始

- 下载Quartz
- 安装 Quartz
- 从源码自己构建Quartz
- 按照自己特定的需要配置Quartz
- 开始一个简单的应用

注意，写这篇文档的时候，Quartz 的版本是 1.4.3。无论何时，指南都是版本无关的，只是需要了解这个事实。

下载和安装

访问 http://sourceforge.net/projects/quartz/ 并且点击**Quartz 1.x**包旁边的download连接下载quratz。

- Quartz 1.x(.zip) – zip格式的主Quartz 包。(主要为Windows用户准备)
- Quartz 1.4.3 (.tar.gz) - in tar/gzip主Quartz 包。(主要为*nix用户准备)
- Quartz Web-App - Quartz web 应用,它能让你通过web接口来监控scheduler。

下载和解压缩这些文件，然后将其安装在应用可以看到的地方就可以了。

所需的 JAR 文件

Quartz 包括了一系列的 jar 文件，这些文件位于 Quartz 主目录的 **lib** 目录中。主 Quartz 类库被形象地命名为 **quartz.jar**。为了使用 Quartz 的特性，这个 jar 文件必须位于 classpath 中。

Quartz 依赖一系列 Jar 文件,要使用 Quartz,所有这些 Jar 文件都必须位于 classpath 中。

The properties file

Quartz 使用一个名为 **quartz.properties**.的配置文件，开始时这个文件不是必须的,但是如果使用除了最基本的功能之外的其它功能,则该文件必须位于 classpath 中。

从源码自己构建 Quartz

查看 release 文档来获得 buildQuartz 的信息。(一般不需要开发者自己构造)

配置

Quartz 是一个可配置性非常强的应用,最佳的配置方法就是编辑 quartz.properties 文件。

首先看的是 **example_quartz.properties**,这个文件位于 Quartz 主目录下的 **docs\wikidocs** 目录中。也可以点击下列连接查看。

- example_quartz.properties

建议你在例子文件的基础上创建自己的 quartz.properties 文件而不是拷贝 example_quartz.properties 文件并且删除那些你不需要的内容。通过这种方式，你会发现更多 Quartz 所提供的功能。

注意： example_quartz.properties 并没有包括所有可能的属性。

为了快速建立和运行基于 quartz 的应用，基本的 quartz.properties 内容如下：

```
org.quartz.scheduler.instanceName = Sched1
org.quartz.scheduler.instanceId = 1
org.quartz.scheduler.rmi.export = false
org.quartz.scheduler.rmi.proxy = false
org.quartz.threadPool.class = org.quartz.simpl.SimpleThreadPool
org.quartz.threadPool.threadCount = 3
org.quartz.jobStore.class = org.quartz.simpl.RAMJobStore
```

依据这个配置文件创建的 scheduler 有如下特性：

- org.quartz.scheduler.instanceName – 它叫做"Sched1" (doh)
- org.quartz.scheduler.rmi.export – 这个scheduler是本地的，这意味着它不能通过RMI(Remote Method Invocation)进行访问。
- org.quartz.threadPool.threadCount –在线程池中有 3 个线程，这意味着最多有 3 个线程可以并发运行。
- org.quartz.jobStore.class – 所有的Quartz数据,例如Job和Trigger的细节信息被存储在内存（而不是数据库）中。

尽管你有一个数据库并且希望Quartz使用这个数据库，建议你在使用数据库之前先使用RamJobStore进行工作。

开始一个简单的应用

现在你已经下载安装了 Quartz，应当创建一个简单的应用并且运行这个应用。下面的代码包括了如何实例化一个 scheduler，启动它并且关闭它。

QuartzTest.java

```
import org.quartz.Scheduler;
import org.quartz.SchedulerException;
import org.quartz.impl.StdSchedulerFactory;
public class QuartzTest {
    public static void main(String[] args) {
        try {
            // Grab the Scheduler instance from the Factory
            Scheduler scheduler = StdSchedulerFactory.getDefaultScheduler();
            // and start it off
            scheduler.start();
            scheduler.shutdown();
        } catch (SchedulerException se) {
```

```
            se.printStackTrace();
        }
    }
}
```

注意：一旦通过StdSchedulerFactory.getDefaultScheduler()获得一个scheduler ，那么应用将不会结束，除非调用scheduler.shutdown()方法。

如果没有建立日志机制，所有的日志都将被发送到控制台，并且输出如下样子的信息：

```
16-Dec-2004 16:15:21 org.quartz.simpl.SimpleThreadPool initialize
INFO: Job execution threads will use class loader of thread: main
16-Dec-2004 16:15:22 org.quartz.simpl.RAMJobStore initialize
INFO: RAMJobStore initialized.
16-Dec-2004 16:15:22 org.quartz.impl.StdSchedulerFactory instantiate
INFO: Quartz scheduler 'DefaultQuartzScheduler' initialized from default
resource file in Quartz package: 'quartz.properties'
16-Dec-2004 16:15:22 org.quartz.impl.StdSchedulerFactory instantiate
INFO: Quartz scheduler version: 1.4.2
16-Dec-2004 16:15:22 org.quartz.core.QuartzScheduler start
INFO: Scheduler DefaultQuartzScheduler_$_NON_CLUSTERED started.
16-Dec-2004 16:15:22 org.quartz.core.QuartzScheduler shutdown
INFO: Scheduler DefaultQuartzScheduler_$_NON_CLUSTERED shutting down.
16-Dec-2004 16:15:22 org.quartz.core.QuartzScheduler pause
INFO: Scheduler DefaultQuartzScheduler_$_NON_CLUSTERED paused.
16-Dec-2004 16:15:22 org.quartz.core.QuartzScheduler shutdown
INFO: Scheduler DefaultQuartzScheduler_$_NON_CLUSTERED shutdown complete.
```

Quartz 官方开发指南

1. 第一课：使用Quartz
2. 第二课:Jobs And Triggers
3. 第三课:更多关于Jobs和JobDetails的内容
4. 第四课:关于Triggers更多的内容
5. 第五课:SimpleTriggers
6. 第六课: CronTriggers
7. 第七课: TriggerListeners和JobListeners
8. 第八课: SchedulerListeners
9. 第九课: JobStores
10. 第十课: Configuration, Resource 使用及SchedulerFactory
11. 第十一课: 高级（企业级）特性
12. 第十二课: 其他特性

第一课：使用 Quartz

使用 scheduler 之前应首先实例化它。使用 SchedulerFactory 可以完成 scheduler 的实例化。有些 Quartz 用户将工厂类的实例放在 JNDI 中存储，其他用户可能直接地实例化这个工厂类并且直接使用工厂的实例（例如下面的例子）。

一旦一个 scheduler 被实例化，它就可以被启动(start),并且处于驻留模式，直到被关闭(shutdown)。注意，一旦 scheduler 被关闭（shutdown）,则它不能再重新启动，除非重新实例化它。除非 scheduler 被启动或者不处于暂停状态，否则触发器不会被触发(任务也不能被执行)。

下面是一个代码片断，这个代码片断实例化并且启动了一个 scheduler，接着将一个要执行的任务纳入了进程。

Using Quartz

```
SchedulerFactory schedFact = new org.quartz.impl.StdSchedulerFactory();
Scheduler sched = schedFact.getScheduler();
sched.start();
JobDetail jobDetail = new JobDetail("myJob",null,DumbJob.class);
Trigger trigger = TriggerUtils.makeHourlyTrigger(); //每个小时激活一次
trigger.setStartTime(TriggerUtils.getEvenHourDate(new Date()));//在下一个小时启动。
trigger.setName("myTrigger");
sched.scheduleJob(jobDetail, trigger);
```

如您所见，使用 quartz 相当简单，在第二课中，我们将给出一个 Job 和 Trigger 的快速预览，这样就能够充分理解这个例子。

第二课 Jobs And Triggers

正如前面所提到的那样，通过实现 Job 接口来使你的 Java 组件可以很简单地被 scheduler 执行。下面是 Job 接口：

Job Interface

```
package org.quartz;
public interface Job {
   public void execute(JobExecutionContext context)
      throws JobExecutionException;
}
```

这样，你会猜想出，当 Job 触发器触发时（在某个时刻），execute(..)就被 scheduler 所调用。***JobExecutionContext*** 对象被传递给这个方法，它为 Job 实例提供了它的“运行时”环境-一个指向执行这个 Job 实例的 Scheduler 句柄，一个指向触发该次执行的触发器的句柄，Job 的 JobDetail 对象以及一些其他的条目。

JobDetail 对象由 Quartz 客户端在 Job 被加入到 scheduler 时创建。它包含了 Job 的各种设置属性以及一个 ***JobDataMap*** 对象，这个对象被用来存储给定 Job 类实例的状态信息。

Trigger 对象被用来触发 jobs 的执行。你希望将任务纳入到进度，要实例化一个 Trigger 并且“调整”它的属性以满足你想要的进度安排。Triggers 也有一个 JobDataMap 与之关联，这非常有利于向触发器所触发的 Job 传递参数。Quartz 打包了很多不同类型的 Trigger,但最常用的 Trigge 类是 SimpleTrigger 和 CronTrigger。

SimpleTrigger 用来触发只需执行一次或者在给定时间触发并且重复 N 次且每次执行延迟一定时间的任务。CronTrigger 按照日历触发，例如“每个周五”，每个月 10 日中午或者 10：15 分。

为什么要分为 Jobs 和 Triggers?很多任务日程管理器没有将 Jobs 和 Triggers 进行区分。一些产品中只是将“job”简单地定义为一个带有一些小任务标识的执行时间。其他产品则更像 Quartz 中 job 和 trigger 的联合。而开发 Quartz 的时候，我们决定对日程和按照日程执行的工作进行分离。（从我们的观点来看）这有很多好处。

例如：jobs 可以被创建并且存储在 job scheduler 中，而不依赖于 trigger,而且，很多 triggers 可以关联一个 job.另外的好处就是这种“松耦合”能使与日程中的 Job 相关的 trigger 过期后重新配置这些 Job,这样以后就能够重新将这些 Job 纳入日程而不必重新定义它们。这样就可以更改或者替换 trigger 而不必重新定义与之相连的 job 标识符。

当向 Quartz scheduler 中注册 Jobs 和 Triggers 时，它们要给出标识它们的名字。Jobs 和 Triggers 也可以被放入“组”中。“组”对于后续维护过程中，分类管理 Jobs 和 Triggers 非常有用。Jobs 和 Triggers 的名字在组中必须唯一，换句话说，Jobs 和 Triggers 真实名字是它的名字+组。如果使 Job 或者 Trigger 的组为‘null’，这等价于将其放入缺省的 *Scheduler.DEFAULT_GROUP* 组中。

现在对什么是Jobs 和 Triggers有了一般性的认识，可以通过第三课:更多关于Jobs和JobDetails的内容及第四课:关于Triggers更多的内容来深入地学习它们。

第三课：更多关于 Jobs 和 JobDetails 的内容

如你所见,Job 相当容易实现。这里只是介绍有关 Jobs 本质，Job 接口的 execute(..)方法以及 JobDetails 中需要理解的内容。

在所实现的类成为真正的“Job”时，期望任务所具有的各种属性需要通知给 Quartz。通过 JobDetail 类可以完成这个工作，这个类在前面的章节中曾简短提及过。软件“考古学家”们可能对了解 Quartz 早期版本的样子感兴趣。早期的 Quartz 中，JobDetail 实现的功能被强加给实现 Job 的类，实现 Job 的类必须实现 Job 接口中所有同 JobDetail 类一样的'getter'方法。这样为每个 Job 类强加了一个重新实现虚拟识别码的笨重工作，这实在糟糕，因此，我们创建了 JobDetail 类。

现在，我们花一些时间来讨论 Quartz 中 Jobs 的本质和 Job 实例的生命周期。首先让我们回顾一下第一课中所看到的代码片断：

Using Quartz

```
JobDetail jobDetail = new JobDetail("myJob",                    // job 名
                                    sched.DEFAULT_GROUP,    // job 组(你可以指定为'null'以使用缺省的组)
                    DumbJob.class); // 要被执行的Java类。
  Trigger trigger = TriggerUtils.makeDailyTrigger(8, 30);
  trigger.setStartTime(new Date());
  trigger.setName("myTrigger");
  sched.scheduleJob(jobDetail, trigger);
```

现在考虑如下定义的 DumbJob 类：

DumbJob

```
public class DumbJob implements Job {
    public DumbJob() {
    }
    public void execute(JobExecutionContext context)
```

```
        throws JobExecutionException
    {
        System.err.println("DumbJob is executing.");
    }
}
```

注意，我们给 scheduler 传入了一个 JobDetail 实例，而且这个 JobDetail 实例只是简单提供了类名来引用被执行的 Job。每次 scheduler 执行这个任务时，它就创建这个类的新实例，然后调用该实例的 execute(..)方法。对这种行为的一个推论就是 Job 类必须有一个无参数的构造函数。另外一个推论就是它使得 Job 类中定义的成员数据失去意义，因为这些成员数据值在每次执行的时候被“清空”了。

你可能要问，如何才能为每个 Job 实例提供属性和配置呢？而且，在执行中如何跟踪 Job 的状态呢？这些问题的答案是相同的：关键就是 JobDataMap，这是 JobDetail 对象的一部分。

JobDataMap

JobDataMap 被用来保存一系列的（序列化的）对象，这些对象在 Job 执行时可以得到。JobDataMap 是 Java Map 接口的一个实现，而且还增加了一些存储和读取主类型数据的便捷方法。

下面是将 Job 加入到 scheduler 前使用的一些向 JobDataMap 加入数据的方法。

Setting Values in a JobDataMap

```
jobDetail.getJobDataMap().put("jobSays", "Hello World!");
jobDetail.getJobDataMap().put("myFloatValue", 3.141f);
jobDetail.getJobDataMap().put("myStateData", new ArrayList());
```

下面的代码展示了在 Job 执行过程中从 JobDataMap 获取数据的代码：

Getting Values from a JobDataMap

```
public class DumbJob implements Job {
    public DumbJob() {
    }
    public void execute(JobExecutionContext context)
        throws JobExecutionException
    {
        String instName = context.getJobDetail().getName();
        String instGroup = context.getJobDetail().getGroup();
        JobDataMap dataMap = context.getJobDetail().getJobDataMap();
```

```
      String jobSays = dataMap.getString("jobSays");
      float myFloatValue = dataMap.getFloat("myFloatValue");
      ArrayList state = (ArrayList)dataMap.get("myStateData");
      state.add(new Date());
      System.err.println("Instance " + instName + " of DumbJob says: " +
jobSays);
    }
  }
```

如果使用一个持久的 JobStore（在本指南的 JobStore 章节中讨论），那么必须注意存放在 JobDataMap 中的内容。因为放入 JobDataMap 中的内容将被序列化，而且容易出现类型转换问题。很明显，标准 Java 类型将是非常安全的，但除此之外的类型，任何时候，只要有人改变了你要序列化其实例的类的定义，就要注意是否打破了程序的兼容性。有关这方面的更多信息可以在 Java Developer Connection Tech Tip: Serialization In The Real World 中找到。另外，你可以对 JobStore 和 JobDataMap 采用一种使用模式：就是只把主类型和 String 类型存放在 Map 中，这样就可以减少后面序列化的问题。

有状态和无状态任务

Triggers 也可以有 JobDataMaps 与之相关联。当 scheduler 中的 Job 被多个有规律或者重复触发的 Triggers 所使用时非常有用。对于每次独立的触发，你可为 Job 提供不同的输入数据。

从 Job 执行时的 JobExecutionContext 中取得 JobDataMap 是惯用手段，它融合了从 JobDetail 和从 Trigger 中获的 JobDataMap，当有相同名字的键时，它用后者的值覆盖前者值。下面的代码就是在 Job 执行过程中从 JobExecutionContext's 获取融合的 JobDataMap。

Getting Values from the JobExecutionContext convenience/merged JobDataMap

```
public class DumbJob implements Job {

    public DumbJob() {
    }

    public void execute(JobExecutionContext context)
      throws JobExecutionException
    {
      String instName = context.getJobDetail().getName();
```

```
        String instGroup = context.getJobDetail().getGroup();

        JobDataMap dataMap = context.getJobDataMap();  // 注意同前面例子的不同

        String jobSays = dataMap.getString("jobSays");
        float myFloatValue = dataMap.getFloat("myFloatValue");
        ArrayList state = (ArrayList)dataMap.get("myStateData");
        state.add(new Date());
        System.err.println("Instance " + instName + " of DumbJob says: " +
jobSays);
    }
  }
```

StatefulJob——有状态任务

现在，一些关于 Job 状态数据的附加论题：一个 Job 实例可以被定义为“有状态的”或者“无状态的”。“无状态的”任务只拥有它们被加入到 scheduler 时所存储的 JobDataMap。这意味着，在执行任务过程中任何对 Job Data Map 所作的更改都将丢失而且任务下次执行时也无法看到。你可能会猜想出，有状态的任务恰好相反，它在任务的每次执行之后重新存储 JobDataMap。有状态任务的一个副作用就是它不能并发执行。换句话说，如果任务有状态，那么当触发器在这个任务已经在执行的时候试图触发它，这个触发器就会被阻塞（等待），直到前面的执行完成。

想使任务有状态，它就要实现*StatefulJob*接口而不是实现Job接口。

Job 'Instances' 任务“实例”

这个课程的最终观点或许已经很明确，可以创建一个单独的 Job 类，并且通过创建多个 JobDetails 实例来将它的多个实例存储在 scheduler 中，这样每个 JobDetails 对象都有它自己的一套属性和 JobDataMap，而且将它们都加入到 scheduler 中。

当触发器被触发的时候，通过 Scheduler 中配置的 JobFactory 来实例化与之关联的 Job 类。缺省的 JobFactory 只是简单地对 Job 类调用 newInstance()方法。创建自己 JobFactory 可以利用应用中诸如 Ioc 或者 DI 容器所产生或者初始化的 Job 实例。

Other Attributes Of Jobs ——Jobs 的其它属性

这里简短地总结一下通过 JobDetail 对象可以定义 Job 的其它属性。

- Durability（持久性）-如果一个 Job 是不持久的， 一旦没有触发器与之关联，它就会被从 scheduler 中自动删除。
- Volatility（无常性）-如果一个 Job 是无常的,在重新启动 Quartz i scheduler 时它不能被保持。

- RequestsRecovery（请求恢复能力） -如果一个 Job 具备“请求恢复”能力，当它在执行时遇到 scheduler “硬性的关闭”（例如：执行的过程崩溃，或者计算机被关机），那么当 scheduler 重新启动时，这个任务会被重新执行。这种情况下，JobExecutionContext.isRecovering() 方法的返回值将是 true。
- JobListeners（任务监听器） -一个 Job 如果有 0 个或者多个 JobListeners 监听器与之相关联，当这个 Job 执行时，监听器被会被通知。更多有关 JobListeners 的讨论见 TriggerListeners & JobListeners 章节。

JobExecutionException 任务执行异常

最后，需要告诉你一些关于 Job.execute(..)方法的细节。在 execute 方法被执行时，仅允许抛出一个 JobExecutionException 类型异常。因此需要将整个要执行的内容包括在一个'try-catch'块中。应花费一些时间仔细阅读 JobExecutionException 文档，因为 Job 能够使用它向 scheduler 提供各种指示，你也可以知道怎么处理异常。

第四课：关于 Triggers 更多内容

同 Job 一样，trigger 非常容易使用，但它有一些可选项需要注意和理解，同时，trigger 有不同的类型，要按照需求进行选择。

Calendars——日历

Quartz ***Calendar***（不是java.util.Calendar）对象在trigger被存储到scheduler时与trigger相关联。Calendar对于在trigger触发日程中的采用批量世间非常有用。例如：你想要创建一个在每个工作日上午 9：30 触发一个触发器，那么就添加一个排除所有节假日的日历。

Calendar 可以是任何实现 Calendar 接口的序列化对象。看起来如下；

Calendar Interface

```
package org.quartz;
public interface Calendar {
public boolean isTimeIncluded(long timeStamp);
public long getNextIncludedTime(long timeStamp);
}
```

注意，这些方法的参数都是 long 型，你可以猜想出，它们的时间戳是毫秒的格式。这意味日历能够排除毫秒精度的时间。最可能的是，你可能对排除整天的时间感兴趣。为了提供方便，Quartz 提供了一个 org.quartz.impl.HolidayCalendar，这个类可以排除整天的时间。

Calendars 必须被实例化，然后通过 addCalendar(..)方法注册到 scheduler 中。如果使用 HolidayCalendar，在实例化之后，你可以使用它的 addExcludedDate(Date date)方法来定义你想要从日程表中排除的时间。同一个 calendar 实例可以被用于多个 trigger 中，如下：

Using Calendars

```
HolidayCalendar cal = new HolidayCalendar();
cal.addExcludedDate(someDate);
sched.addCalendar("myHolidays", cal, false);
Trigger trigger = TriggerUtils.makeHourlyTrigger(); // 每一个小时触发一次
trigger.setStartTime(TriggerUtils.getEvenHourDate(new Date()));//从下一个小时开始
trigger.setName("myTrigger1");
trigger.setCalendarName("myHolidays");// ..用trigger来安排任务。
Trigger trigger2 = TriggerUtils.makeDailyTrigger(8, 0); //每天 8:00 触发
trigger.setStartTime(new Date()); // begin immediately
trigger2.setName("myTrigger2");
trigger2.setCalendarName("myHolidays");//用trigger2 来安排任务
```

传入 SimpleTrigger 构造函数的参数的细节将在下章中介绍。现在只要确信上面的代码创建了两个 trigger,一个每隔 60 分钟重复一次，另一个每 5 天重复 5 次。但是，任何在日历中被排除的时间所要进行的触发都被取消。

Misfire Instructions——未触发指令

Trigger的另一个重要属性就是它的“misfire instruction(未触发指令)”。如果因为scheduler被关闭而导致持久的触发器“错过”了触发时间，这时，未触发就发生了。不同类型的触发器有不同的未触发指令。缺省情况下，他们会使用一个“智能策略”指令——根据触发器类型和配置的不同产生不同动作。当scheduler开始时，它查找所有未触发的持久triggers，然后按照每个触发器所配置的未触发指令来更新它们。开始工程中使用Quartz的时，应熟悉定义在各个类型触发器上的未触发指令，以及它们在JavaDoc中的解释说明。关于未触发指令信息的详细说明将在每种特定的类型触发器的指南课程中给出。可以通过*setMisfireInstruction(..)*来为给定的触发器实例配置未触发指令。

TriggerUtils - Triggers Made Easy（TriggerUtils——使 Triggers 变得容易）

TriggerUtils 类包含了创建触发器以及日期的便捷方法，而无须围绕 java.util.Calendar 对象。使用这个类可以轻松地使触发器在每分钟，小时，日，星期，月等触发。使用这个类也可以产生距离触发最近的妙、分或者小时，这对设定触发开始时间非常有用。

TriggerListeners

最后，如同job一样，triggers可以注册监听器，实现***TriggerListener***接口的对象将可以收到触发器被触发的通知。

第五课: SimpleTrigger

如果需要让任务只在某个时刻执行一次，或者，在某个时刻开始，然后按照某个时间间隔重复执行，简单地说，如果你想让触发器在 2005 年 1 月 13 日，上午 11：23：54 秒执行，然后每个隔 10 秒钟重复执行一次，并且这样重复 5 次。那么***SimpleTrigger*** 就可以满足你的要求。

通过这样的描述，你可能很惊奇地发现 SimpleTrigger 包括这些属性：开始时间，结束时间，重复次数，重复间隔。所有这属性都是你期望它所应具备的，只有 end-time 属性有一些条目与之关联。

重复次数可能是 0，正数或者一个常量值 SimpleTrigger.REPEAT_INDEFINITELY。重复间隔时间属性可能是 0，正的 long 型，这个数字以毫秒为单位。注意：如果指定的重复间隔时间是 0，那么会导致触发器按照‘重复数量’定义的次数并发触发（或者接近并发）。

如果不熟悉java.util.Calendar类，按照开始时间（***startTime***）或者结束时间（***endTime***）来计算触发器的初次触发时间很方便，*org.quartz.helpers.TriggerUtils* 类对处理这样的循环也提供了很多支持。

endTime（如果这个属性被设置）属性会覆盖重复次数属性，这对创建一个每隔 10 秒就触发一次直到某个时间结束的触发器非常有用，这就可以不计算开始时间和结束时间之间的重复数量。也可以指定一个结束时间，然后使用REPEAT_INDEFINITELY作为重复数量。（甚至可以指定一个大于结束时间之前实际重复次数的整数作为重复次数）。一句话，***endTime***属性控制权高于重复次数属性。

SimpleTrigger 有几个不同的构造函数，但是我们将测试这个，并且在下面的例子中使用它：

SimpleTrigger 的一个构造函数

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

SimpleTrigger Example 1 – 从现在 10 秒钟后开始触发，并且只触发一次。

```
long startTime = System.currentTimeMillis() + 10000L;
  SimpleTrigger trigger = new SimpleTrigger("myTrigger",
                                            null,
                                            new Date(startTime),
                                            null,
                                            0,
                                            0L);
```

SimpleTrigger Example 2 – 创建一个立即触发的触发器，并且每隔 60 秒钟触发一次，直到永远。

```
SimpleTrigger trigger = new SimpleTrigger("myTrigger",
                                            null,
                                            new Date(),
                                            null,
SimpleTrigger.REPEAT_INDEFINITELY,
                                            60L * 1000L);
```

SimpleTrigger Example 3 -创建一个立即触发的触发器，并且每隔 10 秒重复一次，直到 40 秒钟以后。

```
long endTime = System.currentTimeMillis() + 40000L;
SimpleTrigger trigger = new SimpleTrigger("myTrigger",
                                            "myGroup",
                                            new Date(),
                                            new Date(endTime),
                                SimpleTrigger.REPEAT_INDEFINITELY,
                                            10L * 1000L);
```

SimpleTrigger Example 4 - 创建一个触发器，开始触发时间为 2002 年 3 月 17 上午 10：30 分，并且重复 5 次。（总共触发 6 次）每次延迟 30 秒钟。

```
java.util.Calendar cal = new java.util.GregorianCalendar(2002, cal.MARCH, 17);
cal.set(cal.HOUR, 10);
cal.set(cal.MINUTE, 30);
cal.set(cal.SECOND, 0);
cal.set(cal.MILLISECOND, 0);
Data startTime = cal.getTime()
SimpleTrigger trigger = new SimpleTrigger("myTrigger",
                                            null,
                                            startTime,
                                            null,
```

```
                                                    5,
                                                    30L * 1000L);
```

花一些时间来看其他的构造函数（以及属性设置），这样可以选择能够完成工作的最方便的构造函数。

SimpleTrigger Misfire Instructions——SimpleTrigger 的未触发指令

“未触发”发生时，SimpleTrigger 有几个指令可以用来通知 Quartz 进行相关处理。（“未触发”在本指南中有关触发器的章节中介绍过）。这些指令以常量形式定义在 SimpleTrigger 本身（通过 JavaDOC 查看它们的行为描述），这些指令如下：

Misfire Instruction Constants of SimpleTrigger

```
MISFIRE_INSTRUCTION_FIRE_NOW
MISFIRE_INSTRUCTION_RESCHEDULE_NOW_WITH_EXISTING_REPEAT_COUNT
MISFIRE_INSTRUCTION_RESCHEDULE_NOW_WITH_REMAINING_REPEAT_COUNT
MISFIRE_INSTRUCTION_RESCHEDULE_NEXT_WITH_REMAINING_COUNT
MISFIRE_INSTRUCTION_RESCHEDULE_NEXT_WITH_EXISTING_COUNT
```

回顾前面的课程你可以知道，每个触发器都有一个 *Trigger.MISFIRE_INSTRUCTION_SMART_POLICY*指令可用，并且，这个指令对于每个类型的触发器都是缺省的。

第六课: CronTrigger

如果你需要像日历那样按日程来触发任务，而不是像 SimpleTrigger 那样每隔特定的间隔时间触发，CronTriggers 通常比 SimpleTrigger 更有用。

使用CronTrigger，你可以指定诸如“每个周五中午”，或者“每个工作日的 9:30”或者“从每个周一、周三、周五的上午 9：00 到上午 10：00 之间每隔五分钟”这样日程安排来触发。甚至，象SimpleTrigger一样，CronTrigger也有一个***startTime***以指定日程从什么时候开始，也有一个（可选的）***endTime***以指定何时日程不再继续。

Cron Expressions——Cron 表达式

Cron表达式被用来配置CronTrigger实例。Cron表达式是一个由 7 个子表达式组成的字符串。每个子表达式都描述了一个单独的日程细节。这些子表达式用空格分隔，分别表示：

1. Seconds 秒
2. Minutes 分钟
3. Hours 小时
4. Day-of-Month 月中的天
5. Month 月
6. Day-of-Week 周中的天
7. Year (optional field) 年（可选的域）

一个cron表达式的例子字符串为*"0 0 12 ? * WED"*,这表示“每周三的中午 12：00”。

单个子表达式可以包含范围或者列表。例如：前面例子中的周中的天这个域（这里是"WED"）可以被替换为"MON-FRI", "MON, WED, FRI"或者甚至"MON-WED,SAT"。

通配符（'*'）可以被用来表示域中“每个”可能的值。因此在"Month"域中的*表示每个月，而在 Day-Of-Week 域中的*则表示“周中的每一天”。

所有的域中的值都有特定的合法范围，这些值的合法范围相当明显，例如：秒和分域的合法值为 0 到 59，小时的合法范围是 0 到 23，Day-of-Month 中值得合法凡范围是 0 到 31，但是需要注意不同的月份中的天数不同。月份的合法值是 0 到 11。或者用字符串 JAN,FEB MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV 及 DEC 来表示。Days-of-Week 可以用 1 到 7 来表示（1＝星期日）或者用字符串 SUN, MON, TUE, WED, THU, FRI 和 SAT 来表示.

'/'字符用来表示值的增量，例如，如果分钟域中放入'0/15'，它表示“每隔 15 分钟，从 0 开始”，如果在份中域中使用'3/20'，则表示“小时中每隔 20 分钟，从第 3 分钟开始”或者另外相同的形式就是'3,23,43'。

'?'字符可以用在 day-of-month 及 day-of-week 域中，它用来表示“没有指定值”。这对于需要指定一个或者两个域的值而不需要对其他域进行设置来说相当有用。看下面例子（以及 CronTrigger JavaDOC）会更清楚。

'L'字符可以在 day-of-month 及 day-of-week 中使用，这个字符是"last"的简写，但是在两个域中的意义不同。例如，在 day-of-month 域中的"L"表示这个月的最后一天，即，一月的 31 日，非闰年的二月的 28 日。如果它用在 day-of-week 中，则表示"7"或者"SAT"。但是如果在 day-of-week 域中，这个字符跟在别的值后面，则表示"当月的最后的周 XXX"。例如："6L" 或者 "FRIL"都表示本月的最后一个周五。当使用'L'选项时，最重要的是不要指定列表或者值范围，否则会导致混乱。

'W' 字符用来指定距离给定日最接近的周几（在 day-of-week 域中指定）。例如：如果你为 day-of-month 域指定为"15W",则表示“距离月中 15 号最近的周几”。

'#'表示表示月中的第几个周几。例如：day-of-week 域中的"6#3" 或者 "FRI#3"表示“月中第三个周五”。

下面是一些表达式以及它们的含义，你可以在 CronTrigger 的 JavaDOC 中找大更多例子。

Example Cron Expressions ——Cron 表达式的例子

CronTrigger 例 1 – 一个简单的每隔 5 分钟触发一次的表达式

"0 0/5 * * * ?"

CronTrigger 例 2 – 在每分钟的 10 秒后每隔 5 分钟触发一次的表达式(例如. 10:00:10 am, 10:05:10 等.)。

"10 0/5 * * * ?"

CronTrigger 例 3 – 在每个周三和周五的 10：30，11：30，12：30 触发的表达式。

"0 30 10-13 ? * WED,FRI"

CronTrigger 例 4 – 在每个月的 5 号，20 号的 8 点和 10 点之间每隔半个小时触发一次且不包括 10 点，只是 8：30，9：00 和 9：30 的表达式。

"0 0/30 8-9 5,20 * ?"

注意，对于单独触发器来说，有些日程需求可能过于复杂而不能用表达式表述，例如：9：00 到 10：00 之间每隔 5 分钟触发一次，下午 1：00 到 10 点每隔 20 分钟触发一次。这个解决方案就是创建两个触发器，两个触发器都运行相同的任务。

第 7 课: TriggerListeners 和 JobListeners

监听器是在scheduler事件发生时能够执行动作的对象。可以看出，**TriggerListeners**接收与triggers相关的事件，而**JobListeners**则接收与Job相关的事件。

Trigger 相关的事件包括：trigger 触发、trigger 未触发，以及 trigger 完成（由 trigger 触发的任务被完成）。

The org.quartz.TriggerListener Interface

```
public interface TriggerListener {
    public String getName();
    public void triggerFired(Trigger trigger, JobExecutionContext context);
    public boolean vetoJobExecution(Trigger trigger, JobExecutionContext
context);
```

```
    public void triggerMisfired(Trigger trigger);
    public void triggerComplete(Trigger trigger, JobExecutionContext context,
            int triggerInstructionCode);
}
```

与任务相关的事件包括：即将被执行的任务的通知和任务已经执行完毕的通知。

The org.quartz.JobListener Interface

```
public interface JobListener {
    public String getName();
    public void jobToBeExecuted(JobExecutionContext context);
    public void jobExecutionVetoed(JobExecutionContext context);
    public void jobWasExecuted(JobExecutionContext context,
            JobExecutionException jobException);
}
```

Using Your Own Listeners 使用你自定义的监听器

创建监听器很简单，创建一个实现 org.quartz.TriggerListener 或（和）org.quartz.JobListener 的接口。监听器然后在执行的时候注册到 scheduler 中，而且必须给定一个名字（或者，它们必须通过他们的 getName()方法来介绍自己）。监听器可以被注册为“全局”的或者“非全局”。“全局”监听器接收所有 triggers/jobs 产生的事件，而“非全局”监听器只接受那些通过 getTriggerListenerNames() 或 getJobListenerNames()方法显式指定监听器名的 triggers/jobs 所产生的事件。

> 正如上面所说的那样，监听器在运行时向 scheduler 注册，并且不被存储在 jobs 和 triggers 的 JobStore 中。Jobs 和 Trigger 只存储了与他们相关的监听器的名字。因此，每次应用运行的时候，都需要向 scheduler 重新注册监听器。

向 Scheduler 中加入一个 JobListener

```
scheduler.addGlobalJobListener(myJobListener);
或者
scheduler.addJobListener(myJobListener);
```

Quartz 的大多数用户不使用监听器，但是当应用需要创建事件通知而 Job 本身不能显式通知应用，则使用监听器非常方便。

第八课: SchedulerListeners

SchedulerListeners同TriggerListeners及JobListeners非常相似，***SchedulerListeners***只接收与特定trigger 或job无关的Scheduler自身事件通知。

Scheduler 相关的事件包括：增加 job 或者 trigger,移除 Job 或者 trigger, scheduler 内部发生的错误，scheduler 将被关闭的通知，以及其他。

***org.quartz.SchedulerListener* 接口**

```
public interface SchedulerListener {
    public void jobScheduled(Trigger trigger);
    public void jobUnscheduled(String triggerName, String triggerGroup);
    public void triggerFinalized(Trigger trigger);
    public void triggersPaused(String triggerName, String triggerGroup);
    public void triggersResumed(String triggerName, String triggerGroup);
    public void jobsPaused(String jobName, String jobGroup);
    public void jobsResumed(String jobName, String jobGroup);
    public void schedulerError(String msg, SchedulerException cause);
    public void schedulerShutdown();
}
```

除了不分“全局”或者“非全局”监听器外，SchedulerListeners 创建及注册的方法同其他监听器类型十分相同。所有实现 org.quartz.SchedulerListener 接口的对象都是 SchedulerListeners。

第 9 课: JobStores

JobStore 负责保持对所有 scheduler “工作数据”追踪，这些工作数据包括：job（任务）,trigger（触发器）,calendar(日历)等。为你的 Quartz scheduler 选择合适的 JobStore 是非常重要的一步，幸运的是，如果你理解了不同的 JobStore 之间的差别，那么选择就变得非常简单。在提供产生 scheduler 实例的 SchedulerFactory 的属性文件中声明 scheduler 所使用的 JobStore（以及它的配置）。

> 不要在代码中直接使用 JobStore 实例，处于某些原因，很多人试图这么做。JobStore 是由 Quartz 自身在幕后使用。你必须告诉（通过配置）Quartz 使用哪个 JobStore，而你只是在你的代码中使用 Scheduler 接口完成工作。

RAMJobStore

RAMJobStore 是最简单的 JobStore，也是性能最好的（根据 CPU 时间）。从名字就可以直观地看出，RAMJobStore 将所有的数据都保存在 RAM 中。这就是为什么它闪电般的快速和如此容易地配置。缺点就是当应用结束时所有的日程信息都会丢失，这意味着 RAMJobStore 不能满足 Jobs 和 Triggers 的持久性（“non-volatility”）。对于有些应用来说，这是可以接受的，甚至是期望的行为。但是对于其他应用来说，这将是灾难。

为了使用 RAMJobStore（假设你使用 StdSchedulerFactory），指使简单地将类名 org.quartz.simpl.RAMJobStore 作为你的 quartz 的配置值。

配置 Quartz 使用 RAMJobStore

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

这里没有其他需要的担心的配置。

JDBCJobStore

JDBCJobStore 的命名也非常得体，它将所有的数据通过 JDBC 保存到数据库可中。它的配置要比 RAMJobStore 稍微复杂，同时速度也没有那么快。但是性能的缺陷不是非常差，尤其是如果你在数据库表的主键上建立索引。在一台主频为 800MHz 的 windows 计算机上运行一个使用一个在不是很新 Solaris 系统上的 Oracle 数据库的 Quartz 应用，接收和更新正在触发的触发器以及与之相关的任务的时间大约为 15 毫秒。

JDBCJobStore 几乎可以在任何数据库上工作，它广泛地使用 Oracle, MySQL, MS SQLServer2000, HSQLDB, PostreSQL 以及 DB2。要使用 JDBCJobStore，首先必须创建一套 Quartz 使用的数据库表，可以在 Quartz 的 docs/dbTables 找到创建库表的 SQL 脚本。如果没有找到你的数据库类型的脚本，那么找到一个已有的，修改成为你数据库所需要的。需要注意的一件事情就是所有 Quartz 库表名都以 QRTZ_作为前缀（例如：表"QRTZ_TRIGGERS",及"QRTZ_JOB_DETAIL"）。实际上，可以你可以将前缀设置为任何你想要的前缀，只要你告诉 JDBCJobStore 那个前缀是什么即可（在你的 Quartz 属性文件中配置）。对于一个数据库中使用多个 scheduler 实例，那么配置不同的前缀可以创建多套库表，十分有用。

一旦数据库表已经创建，在配置和启动JDBCJobStore之前，就需要作出一个更加重要的决策。你要决定在你的应用中需要什么类型的事务。如果不想将scheduling命令绑到其他的事务上，那么你可以通过对JobStore使用***JobStoreTX***来让Quartz帮你管理事务（这是最普遍的选择）。

如果想让Quartz同其他的事务协同工作（例如：J2EE应用服务器中的事务），那么你需要使用***JobStoreCMT***，这样，Quartz就会让应用服务器容器来管理事务。

最后的疑问就是如何建立获得数据库联接的数据源（DataSource）。Quartz 属性中定义数据源有几个不同的途径。一个途径就是通过提供所有联接数据库的信息，让 Quartz 自己创建和管理数据源。另一个途径就是通过提供数据源 JNDI 名字来让 Quartz 使用其

所在应用服务器提供的数据源。详细的属性设置请参照"docs/config"文件夹中的例子配置文件。

要使用 JDBCJobStore（假定使用 StdSchedulerFactory），首先需要设置 Quartz 配置中的 JobStore class 属性为 org.quartz.impl.jdbcjobstore.JobStoreTX 或者 org.quartz.impl.jdbcjobstore.JobStoreCMT，这取决于对事务的选择，前面已经介绍过。

配置 Quartz 使用 JobStoreTx

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

下一步，需要为 JobStore 选择一个 DriverDelegate ， DriverDelegate 负责做指定数据库的所有 JDBC 工作。StdJDBCDelegate 是一个使用 vanilla" JDBC 代码（以及 SQL 语句）来完成工作的代理。如果数据库没有其他指定的代理，那么就试用这个代理。只有当使用 StdJDBCDelegate 发生问题时，我们才会使用数据库特定的代理（这看起来非常乐观。其他的代理可以在 org.quartz.impl.jdbcjobstore 包或者它的下级包中找到。）。其他的代理包括 DB2v6Delegate（专为 DB2 6 版本或者更早版本），HSQLDBDelegate (专为 HSQLDB 数据库)，MSSQLDelegate（专为微软 SQL SERVER2000)，PostgreSQLDelegate (专为 PostgreSQL 7.x)，WeblogicDelegate（专为使用 Weblogic 所提供的 JDBC），OracleDelegate（专为 Oracle 8i 及 9i）。

）。

一旦选择好了代理，就将它的名字设置给 JDBCJobStore。

配置 JDBCJobStore 使用 DriverDelegate

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

Next, you need to inform the JobStore what table prefix (discussed above) you are using.

接下来，需要为 JobStore 指定所使用的数据库表前缀（前面讨论过）。

配置 JDBCJobStore 的数据库表前缀

```
org.quartz.jobStore.tablePrefix = QRTZ_
```

最后，需要设置 JobStore 所使用的数据源。必须在 Quartz 属性中定义已命名的数据源，比如，我们指定 Quartz 使用名为"myDS"的数据源（在配置文件的其他地方定义）。

配置 JDBCJobStore 使用数据源源的名字

```
org.quartz.jobStore.dataSource = myDS
```

> 如果 Scheduler 非常忙（比如，执行的任务数量差不多和线程池的数量相同，那么你需要正确地配置 DataSource 的连接数量为线程池数量

+1）

为了指示 JDBCJobStore 所有的 JobDataMaps 中的值都是字符串，并且能以“名字-值”对的方式存储而不是以复杂对象的序列化形式存储在 BLOB 字段中，应设置 org.quartz.jobStore.useProperties 配置参数的值为"true"(这是缺省的方式)。这样做，从长远来看非常安全，这样避免了对存储在 BLOB 中的非字符串的序列化对象的类型转换问题。

第 10 课:配置、资源使用以 SchedulerFactory

Quartz 以模块方式构架，因此，要使它运行，几个组件必须很好的咬合在一起。幸运的是，已经有了一些现存的助手可以完成这些工作。

在 Quartz 进行工作之前需要被配置的组件主要有：

- ThreadPool　线程池
- JobStore
- DataSources (如果需要)
- Scheduler 本身

ThreadPool（线程池）为Quartz运行任务时提供了一些线程。池中的线程越多，那么并发运行的任务数就越多。但是，过多的线程会降低系统的运行速度。大多数用户发现 5 个或者相近的线程就已经足够了，因为任何给定的时间段内都不超过 100 个任务要运行，而且这些任务不会在同一时刻运行，同时任务活动时间很短（很快就结束了）。其他的用户发现需要 10，15，50，甚至 100 个线程，因为每个schedules都有成千上万的触发器，并且在给定的时刻会有平均 10 到 100 个任务在运行。确定schedule的线程池中的线程数量的合理值取决于用scheduler来做什么。除了尽可能少地设置线程数量，使得任务执行时线程够用外（由于计算机资源的有限性），没有其他实用的准则。注意：如果触发器触发的时间到了，却没有可用的线程，那么Quartz将会让这个任务等待，直到有线程可用。这样，任务的执行将比它因该执行的时间晚一些毫秒。如果scheduler的配置的“未触发极限”时限中仍然没有线程可用，这甚至会导致“未触发(misfire)”。

ThreadPool 接口定义在 org.quartz.spi 中，你也可以创建一个自己的 ThreadPool（线程池）实现，Quartz 打包了一个简单（但非常满意的）的线程池，名为：org.quartz.simpl.SimpleThreadPool，这个线程池只是简单地在它的池中保持固定数量的线程，不增长也不缩小。但是它非常健壮且经过良好的测试，差不多每个 Quartz 用户都使用这个池。

JobStores和***DataSrouces***在第九课中已经讨论过，值得注意的一个事实是所有的JobStores都实现了org.quartz.spi.JobStore接口，如果捆绑的JobStores不能满足你的要求，你可以自己开发一个。

JobStores

Finally, you need to create your ***Scheduler*** instance. The Scheduler itself needs to be given a name, told its RMI settings, and handed instances of a JobStore and ThreadPool. The RMI settings include whether the Scheduler should create itself as an RMI server object (make itself available to remote connections), what host and port to use, etc.. StdSchedulerFactory (discussed below) can also produce Scheduler instances that are actually proxies (RMI stubs) to Schedulers created in remote processes.

最后你需要创建自己的***Scheduler***实例。***Scheduler***本身需要给定一个名字，告诉它的RMI设置，处理的JobStore和ThreadPool实例。RMI包括是否Scheduler将自己创建为一个RMI服务对象（使它可以被远程连接），所使用的主机和端口等。StdSchedulerFactory（下面将要讨论）也能够创建Scheduler实例来代理远程过程中的Schedulers。

StdSchedulerFactory

StdSchedulerFactory 是对 org.quartz.SchedulerFactory 接口的一个实现。是使用一套属性(java.util.Properties)来创建和初始化 Quartz Scheduler。这些属性通常在文件中存储和加载。也可以通过编写程序来直接操作工厂。简单地调用工厂的 getScheduler()就可以产生一个 scheduler，初始化（以及它的 ThreadPool、JobStore 和 DataSources），并且返回一个公共的接口。

在 Quartz 发布的"docs/config"目录中有一些配置的例子。你可以在 Quartz 文档的“参考”中找到完整的配置手册。

DirectSchedulerFactory

DirectSchedulerFactory 是 SchedulerFactory 的另一个实现。它对于那些希望用更加程序化的方式创建 Scheduler 非常有用。不鼓励使用它的原因如下：

(1) 它需要用户非常了解他们想要干什么。
(2) 它不允许声明式的配置。换句话说，它使用硬编码的方式设置scheduler。

Logging 日志

Quartz 用 org.apache.commons.logging 框架来满足它所有的日志需要。Quartz 不会产生太多的日志信息，通常只是一些初始化信息以及只有在任务执行时发生的一些严重问题的信息。要“调整”日志设置（例如输出量以及在哪输出），需要理解 Jakarta Commons Logging 框架，这不在本文档的讨论范围内。

第 11 课: 高级（企业级）属性

Clustering 集群

目前，集群只能用在使用 JDBC-Jobstore（JobStoreTX 中 JobStoreCMT）的情况。特新包括负载均衡和容错（如果 JobDetail 的"request recovery"标记被设置为 true）。

设置"org.quartz.jobStore.isClustered"属性为 true 才可以集群，集群中的每个实例都使用 quartz.properties 的相同拷贝。集群所使用属性文件的例外是一致的，下面是允许的例外：不同的线程池数量，"org.quartz.scheduler.instanceId"的不同属性值。集群中的每个节点必须有唯一的 instanceId，通过替换这个属性的值为"AUTO"就可以轻松做到（不要不同的属性文件）。

> 除非使用某些运行非常有整齐（时钟必须同步在一秒之内）的时间同步服务来同步不同计算机的时钟外，不要将集群运行在不同的计算机三行。如果你不熟悉如何做请参照 http://www.boulder.nist.gov/timefreq/service/its.htm
>
> 不要在一套数据库表上运行未集群的实例。这会导致严重的数据冲突，及不可预知的行为。

JTA Transactions ——JTA 事务

正如第 9 课JobStores中讲的那样，JobStoreCMT使Quartz安排的操作在更大的JTA事务中执行。

也可通过将"org.quartz.scheduler.wrapJobExecutionInUserTransaction"属性设置为 true 来让任务在 JTA 事务中执行，通过这种设置，JTA 事务只是在 Job 执行方法被调用前才调用 begin()方法，并且，在任务执行结束的时候调用 commit()方法。

Quartz 除了自动将任务的执行包装在 JTA 之内，当使用 JobStoreCMT 时，对 Scheduler 接口的调用也参与进了事务。即，确定在调用 scheduler 方法之前已经启动事务即可。也可以通过更加直接的方法，就是使用 UserTransaction 或者将调用 scheduler 的代码放在使用容器管理的事务的 SessionBean 中。

第 12 课: Quartz 的其他特性

Plug-Ins 插件

Quartz 提供了一个接口(org.quartz.spi.SchedulerPlugin)来插入附加的功能。

随Quartz打包儿来的插件有很多有用的功能，它们在***org.quartz.plugins***包中找到。他们提供了诸如自动安排任务的日程，将任务和触发器事件的历史记入日志以及JVM虚拟机退出时确保干净地关闭scheduler等的功能。

JobFactory

当触发器触发时，与之相关联的任务被 Scheduler 中配置的 JobFactory 所实例华。缺省的 JobFactory 只是简单地调用 job 类的 newInstance()方法。你也许想创建自己的 JobFactory 实现，以完成诸如让应用的 IoC 或者 DI 容器产生/初始化 job 实例的功能。

查看 **org.quartz.spi.JobFactory** 接口及与之相关的 **Scheduler.setJobFactory(fact)**方法。

'Factory-Shipped' Jobs

Quartz也提供了一些可以在你的应用中使用的实用的Jobs,比如，发邮件、调用EJBs。这些外来的Job可以在***org.quartz.jobs***包中找到。