Quratz2.3.0 官方教程中文完整版

Quartz 是一个开源的作业调度框架，它完全由 Java 写成，并设计用于 J2SE 和 J2EE 应用中。它提供了巨大的灵活性而不牺牲简单性。你能够用它来为执行作业而创建简单的或复杂的调度。它有很多特征，例如数据库支持、集群、插件、EJB 作业预构、JavaMail 等等，并支持 cron-like 表达式。

Quartz 目前已经被 Terracotta 收购。

Quartz 最新版本包括 2.3.1 和 2.4.0 版，但是稳定版本是 2.3.0，本文档即翻译稳定版 2.3.0。

项目官网：http://www.quartz-scheduler.org

GITHUB：https://github.com/quartz-scheduler/quartz

下载页面：http://www.quartz-scheduler.org/downloads/

文档页面：http://www.quartz-scheduler.org/documentation/

陆印章

2019 年 04 月 07 日

前言一 下载与配置 Quartz

下载地址：http://www.quartz-scheduler.org/downloads/ 。

1.1 直接/手工下载

可以手工下载完整的 Quartz 发行版（包括示例、源代码、依赖项以及 doc 等）。如果使用 Maven，则可以将 Quartz 依赖项添加到 pom.xml 文件中。

1.2 下面是 2.3.0 版本的 Maven 依赖项

~~~

```
  <dependency>
    <groupId>org.quartz-scheduler</groupId>
    <artifactId>quartz</artifactId>
    <version>2.3.0</version>
  </dependency>
  <dependency>
    <groupId>org.quartz-scheduler</groupId>
    <artifactId>quartz-jobs</artifactId>
    <version>2.3.0</version>
  </dependency>
```

~~~

注意：如果要使用其他版本的 Quartz，只需替换上面的版本号即可。

前言二 快速开始

Quartz 是一个开源的作业调度框架，它完全由 Java 编写，一般用于 J2SE 和 J2EE 应用。Quartz 灵活且简单易用。可用来为执行作业而创建简单的或复杂的调度。它有很多特征，例如数据库支持、集群、插件、EJB 作业预构、JavaMail 等等，并支持 cron-like 表达式。

Quartz 目前已经被 Terracotta 收购。

欢迎来到 Quartz 的快速开始教程，本章介绍以下内容：

1）下载 Quartz

2）安装 Quartz

3）设置 Quartz 配置

4）运行简单示例

熟悉 Quratz 调度器的基本功能后，可以考虑使用他的高级特性，例如“where”，一个 Enterprise 特征，允许在 Terracotta 客户机上运行作业和触发器，而不是随机地选择功能。

2.1 下载与安装

首先，下载最近的发行版本（本文档使用目前稳定版 2.3.0），下载时无需注册。下载后解压并安装即可。

2.1.1 Quartz 的 JAR 文件

Quartz 包包含多个 JAR 文件，位于发行包的根目录中。主要的 Quartz 库命名为 quartz-xxx.jar（其中 xxx 是版本号）。为了使用 Quartz 的特征，这个 jar 包必须引入到应用程序的类路径中。

解压 Quartz 后，将它复制到项目的 lib（或其他加载包的文件夹）中。

一般是在应用服务器环境中使用 Quratz，所以首选是在企业应用程序中引入 Quartz 的 JAR 包（.ear 或.war 文件）。然而，如果你想将 Quratz 用到其它的应用程序上，则需要确保 Jar 包引入到正确的类路径上。如果是一个独立的应用程序，那么 jar 包的位置与其他所有 jar 包加载过程是相同的。

Quartz 会依赖于第三方库（同时以 jar 的形式），这些库包含在解压文件的“lib”文件夹中。所有的这些 jar 包也必须引入到类路径中。如果是独立的 Quartz 应用，则只需添加这些 jar 包即可。如果在应用程序服务器环境中使用的 Quartz，那么有一部分 jar 包可能已经存在，那么只需选择里面没有的 jar 添加即可。

2.1.2 Properties 配置文档

Quartz 使用名字为 quartz.properties 的文件配置属性。开始使用时，这个配置并不是必需的，但在使用到一些功能或者最基本的配置，则必须配置这个文件，并通过类路径引入。

在本例，我的应用程序是使用 Weblogic Workshop 开发的。在应用程序的根目录下，我将所有的配置文件（包括 quartz.properties 文件）保存在项目中。当把所有内容打包到.ear 文件中时，配置项目会打包成.jar 包并且包含在.ear 中。quartz.properties 会自动打包入类路径径。

如果在 Web 应用程序中使用 Quartz（即.war 文件的形式），可以将 quartz.properties 文件放在 WEB-INF/classes 文件夹中，文件会自动加到类路径径。

2.2 配置

这是重要的一点。Quartz 是一个可以配置的应用程序。配置的最好方法是使用 quartz.properties 配置文件，并且将其放到应用程序的类路径中（见前一节）。

Quartz 发行版中包含几个属性文件示例，位于 examples/目录下。建议创建自定义的 quartz.properties 文件，而不是复制例子然后删除那些你不需要的。这是了解更多 Quartz 配置的好方法。

完整的文档配置在 Quartz Configuration Reference(http://www.quartz-scheduler.org/documentation/quartz-2.3.0/configuration)中介绍。

为了快速建立和运行，基本的 quartz.properties 如下例：

~~~

```
org.quartz.scheduler.instanceName = MyScheduler
org.quartz.threadPool.threadCount = 3
org.quartz.jobStore.class = org.quartz.simpl.RAMJobStore
```

~~~

此配置创建的调度程序具有以下特性：

1）org.quartz.scheduler.instanceName - 设置这个调度器的名字是“MyScheduler”。

2）org.quartz.threadpool.threadCount - 设置线程池开启的最大线程数，这意味着最多有 3 个 jobs 可以同时运行。

3）org.quartz.jobStore.class - 设置 Quartz 数据，如作业和触发器的详细信息，保存在内存中（而不是在数据库）。即使你有数据库，想它与 Quartz 整合，我建议你在使用数据库工作前先使用 RamJobStore 运行。

2.3 简单的运行例子

现在确认已经下载并安装了 Quartz，现在是时候运行一个示例应用程序了。下面的代码

先获取调度程序的实例，启动然后关闭:
~~~

```
  import org.quartz.Scheduler;
  import org.quartz.SchedulerException;
  import org.quartz.impl.StdSchedulerFactory;
  import static org.quartz.JobBuilder.*;
  import static org.quartz.TriggerBuilder.*;
  import static org.quartz.SimpleScheduleBuilder.*;

  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：一旦使用 StdSchedulerFactory.getDefaultScheduler()开启调度器，你的应用程序终端将不可用直到使用 scheduler.shutdown()关闭方法，因为这个线程正在活动。

请注意代码示例中的静态导入；这些类将在下面的代码示例中发挥作用。

如果没有设置日志记录，那么所有日志都会从控制台输出，可能如下：
~~~

```
[INFO] 21 Jan 08:46:27.857 AM main [org.quartz.core.QuartzScheduler]
Quartz Scheduler v.2.0.0-SNAPSHOT created.

[INFO] 21 Jan 08:46:27.859 AM main [org.quartz.simpl.RAMJobStore]
RAMJobStore initialized.

[INFO] 21 Jan 08:46:27.865 AM main [org.quartz.core.QuartzScheduler]
Scheduler meta-data: Quartz Scheduler (v2.0.0) 'Scheduler' with
instanceId 'NON_CLUSTERED'
```
~~~

```
      Scheduler class: 'org.quartz.core.QuartzScheduler' - running locally.
      NOT STARTED.
      Currently in standby mode.
      Number of jobs executed: 0
      Using thread pool 'org.quartz.simpl.SimpleThreadPool' - with 50
threads.
      Using job-store 'org.quartz.simpl.RAMJobStore' - which does not
support persistence. and is not clustered.


    [INFO] 21 Jan 08:46:27.865 AM main
[org.quartz.impl.StdSchedulerFactory]
    Quartz scheduler 'Scheduler' initialized from default resource file in
Quartz package: 'quartz.properties'

    [INFO] 21 Jan 08:46:27.866 AM main
[org.quartz.impl.StdSchedulerFactory]
    Quartz scheduler version: 2.0.0

    [INFO] 21 Jan 08:46:27.866 AM main [org.quartz.core.QuartzScheduler]
    Scheduler Scheduler$NONCLUSTERED started.

    [INFO] 21 Jan 08:46:27.866 AM main [org.quartz.core.QuartzScheduler]
    Scheduler Scheduler$NONCLUSTERED shutting down.

    [INFO] 21 Jan 08:46:27.866 AM main [org.quartz.core.QuartzScheduler]
    Scheduler Scheduler$NONCLUSTERED paused.

    [INFO] 21 Jan 08:46:27.867 AM main [org.quartz.core.QuartzScheduler]
    Scheduler Scheduler$NONCLUSTERED shutdown complete.
```

在 start()与 shutdown()调用之间编写作业代码。

```
  // define the job and tie it to our HelloJob class
  JobDetail job = newJob(HelloJob.class)
      .withIdentity("job1", "group1")
      .build();

  // Trigger the job to run now, and then repeat every 40 seconds
  Trigger trigger = newTrigger()
      .withIdentity("trigger1", "group1")
      .startNow()
            .withSchedule(simpleSchedule()
              .withIntervalInSeconds(40)
```

~~~
              .repeatForever())
          .build();

        // Tell quartz to schedule the job using our trigger
        scheduler.scheduleJob(job, trigger);
    `````
~~~

在调用 shutdown()之前，需要为作业的触发和执行保留时间，作为一个简单的例子，可以添加 Thread.sleep(60000)到程序中。

第一章 使用 Quartz

在使用调度器（scheduler）之前，需要先进行实例化（谁已经猜到了？）。没错，可以使用 SchedulerFactory 进行实例化。有一些 Quartz 框架的用户可能会将 Factory 的实例存储在 JNDI 中，为了便于举例子可以直接使用 Factory 的实例，也是最容易的入门流程。

一旦调度器实例化后，即可启动它，等待执行和关闭。注意，一旦调度器调用了 shutdown 方法关闭后，如果不重新实例化，它就不会再启动了。触发器（Triggers）在调度器未启动时，或是终止状态时，都不会被触发（不会触发那么作业也就无法执行）。

下面的代码实例化和启动调度器，执行了 Job 对象：

~~~
`````
 SchedulerFactory schedFact = new org.quartz.impl.StdSchedulerFactory();
 Scheduler sched = schedFact.getScheduler();
 sched.start();
 // define the job and tie it to our
    HelloJobclassJobDetail job = newJob(HelloJob.class)
    .withIdentity("myJob", "group1")
    .build();
 //Trigger the job to run now, andthen every 40 seconds
 Trigger trigger = newTrigger()
      .withIdentity("myTrigger", "group1")
      .startNow()
      .withSchedule(simpleSchedule()
          .withIntervalInSeconds(40)
          .repeatForever())
      .build();
 //Tell quartz to schedule the job using our trigger
 sched.scheduleJob(job, trigger);
`````
~~~

由上可知，使用 Quartz 框架非常简单，第二课将会快速介绍 Jobs 类、Triggers 类和 Quartz 的 API，以便加深对这个示例的理解。

第二章 Quartz API、Jobs 和 Triggers

Quartz API 关键的几个接口如下：

1）Scheduler：跟任务调度相关的，最主要的 API 接口。

2）Job：任务调度执行的组件定义（调度器执行的内容），都必须实现该接口。

3）JobDetail：用来定义 Job 的实例。

4）Trigger：定义指定的 Job 何时被执行的组件，也叫触发器。

5）JobBuilder：用来定义或创建 JobDetail 的实例，JobDetail 限定了只能是 Job 的实例。

6）TriggerBuilder：用来定义或创建触发器的实例。

调度器的生命周期，起始于 SchedulerFactory 的创建，终止于调用 shutdown 方法。当调度器接口实例创建完成后，就可以添加、删除和查询 Jobs 和 Triggers 对象，也可以执行其它的跟调度器相关的操作（例如中止触发器的触发）。并且，调度器在调用 start 方法之前，不会触发任何一个触发器（执行作业），如第一章所示的例子。

Quartz 框架提供许多构造器来定义一套领域特定语言，简称 DSL，有时候也称为“流接口”。在下面代码是第一章提到的示例：

`````
  // define the job and tie it to our HelloJob class
  JobDetail job = newJob(HelloJob.class)
      .withIdentity("myJob", "group1") // name "myJob", group "group1"
      .build();

  // Trigger the job to run now, and then every 40 seconds
  Trigger trigger = newTrigger()
      .withIdentity("myTrigger", "group1")
      .startNow()
      .withSchedule(simpleSchedule()
          .withIntervalInSeconds(40)
          .repeatForever())
      .build();

  // Tell quartz to schedule the job using our trigger
  sched.scheduleJob(job, trigger);
`````

生成 Job 定义调用的方法代码块静态地导入了 JobBuilder 类。同样，生成触发器调用的方法代码块静态地导入了 TriggerBuilder 类，SimpleScheduleBuilder 类也是静态导入的。

DSL 静态导入的实现可以通过以下几个 import 语句完成：

`````
import static org.quartz.JobBuilder.*;
import static org.quartz.SimpleScheduleBuilder.*;
import static org.quartz.CronScheduleBuilder.*;
import static org.quartz.CalendarIntervalScheduleBuilder.*;
import static org.quartz.TriggerBuilder.*;
import static org.quartz.DateBuilder.*;
`````

功能各样的 ScheduleBuilder 类提供多种方法来定义不同类型的调度器。

DateBuilder 类包含了许多方法可以更简捷地构建 java.util.Date 实例对象，尤其针对特定的时间点(比如下一个的整点时间，换言之现在是 9:43:27，我需要得到 10:00:00 的时间对象)。

2.1 Jobs 和 Triggers

作业任务类实现 Job 接口，它只有一个方法:

Job 接口：

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

当作业任务的触发器被触发的那一刻，调度器的一个工作线程，将会调用该 Job 的 execute(..)方法。JobExecutionContext 对象会向 execute 方法传递运行时环境的工作任务信息：执行该方法的调度器引用，触发该方法执行的触发器引用，Job 实例的 JobDetail 对象，以及一些其它信息。

当调度器添加 Job 实例时，会在 Quartz 客户端程序中（你开发的代码）创建 JobDetail 对象。JobDetail 为 Job 实例提供了许多设置属性，以及 JobDataMap 成员变量属性，它用来存储特定 Job 实例的状态信息。这是从本质上定义 Job 实例，我们将会在下一节中深入地讨论相关的细节。

触发器对象用来触发 Job 对象的执行。当你希望调度一项作业任务，可以实例化一个触发器并且将触发时间属性调整为你期望的时间表。触发器也可以关联 JobDataMap 对象 -- 这是将参数传递到特定的触发器。Quartz 有几个不同的触发器类型，但最常用的类型是 SimpleTrigger 和 CronTrigger。

SimpleTrigger 适用于一次性的任务执行（在给定的时间段只执行一次的作业任务），或者你需要在指定时间多次触发作业任务，每次触发都延迟固定的时间。CronTrigger 适用于基于类似日历时间表的触发，比如“每个周五的下午”或是“每月 10 号的 10:15”。

为什么要分别定义作业任务和触发器？许多作业调度并没有区分作业任务和触发器的概念。有些框架把作业任务简单定义成执行时间（或计划）以及一些小作业标识符，其他框架更像是整合了 Quartz 框架的作业和触发器对象。因此我们设计 Quartz 时，我们决定构建调度器时拆分开调度和任务是有意义的，在我看来，这样做还是有许多好处。

例如，作业任务可以独立于触发器在作业调度中创建和存储，多个触发器可以关联到相同的作业任务中。另一个好处是在作业任务关联的触发器失效后，仍然能够在调度器上松耦合地配置作业任务，因此该作业不需要重新定义，一段时间后能够重新调度。它还允许你在不重新实例化关联的作业任务下修改和替换触发器。

2.2 标识

作业任务和触发器被注册到 Quartz 调度器时需要提供标识信息。这种标识信息（也称作作业任务键和触发器键）允许作业任务和触发器按组存放，这样可以很方便地将你的作业任务和触发器分组，比如分为“报表类 Job”和“维护类 Job”。作业任务和触发器的键的名称部分在同一个组内必须唯一，换句话说，作业任务和触发器的键（或标识符）的名字是由键名和组名共同组成的。

现在你对作业任务和触发器有一个大概的了解，然后可以在“第三章 Jobs 和 JobDetails 的详解”和“第四章 Triggers 详解”学到更多关于它们的知识。

第三章 Jobs 和 Job Details 详解

在第二章我们已经学习到，Jobs 接口非常容易实现，只有一个 execute 方法。我们需要再学习一些知识去理解 jobs 的本质，Job 接口的 execute 方法以及 JobDetails 接口。

当你实现 Job 接口类，Quartz 需要你提供 job 实例的各种参数，Job 接口实现类中的代码才知道如何去完成指定类型 Job 的具体工作。这个过程是通过 JobDetail 类来完成的，该类会在下一个章节作简要介绍。

JobDetail 的实例是调用 JobBuilder 类创建的。通常你可以使用静态导入方式获得该类所有方法的调用，这样可以在你的代码体验 DSL 的感觉。

`````
import static org.quartz.JobBuilder.*;
`````

现在花点时间来讨论一下关于 Jobs 的本质和 job 实例在 Quartz 中的生命周期。首先回顾一下第一章提及的代码片断：

`````
// define the job and tie it to our HelloJob class
  JobDetail job = newJob(HelloJob.class)
      .withIdentity("myJob", "group1") // name "myJob", group "group1"
      .build();

  // Trigger the job to run now, and then every 40 seconds
  Trigger trigger = newTrigger()
      .withIdentity("myTrigger", "group1")
      .startNow()
      .withSchedule(simpleSchedule()
          .withIntervalInSeconds(40)
          .repeatForever())
      .build();

  // Tell quartz to schedule the job using our trigger
  sched.scheduleJob(job, trigger);
`````

Job 接口的实现类 HelloJob 可以这样定义：

`````
public class HelloJob implements Job {

    public HelloJob() {
    }

    public void execute(JobExecutionContext context)
      throws JobExecutionException
    {
      System.err.println("Hello!  HelloJob is executing.");
`````

```
    }
  }
`````

注意，调度器包含了 JobDetail 实例对象，在构建 JobDetail 对象时仅提供了 job 的 class 对象，调度器就知道它要执行的 job 类型。每次调度器执行 job 时，它在调用 excecute 方法前会创建一个新的 job 实例。当调用完成后，关联的 job 对象实例会被释放，释放的实例会被垃圾回收机制回收。这种调用过程导致的其中一个结果是 jobs 对象必须要有一个无参数构造器(使用默认的 JobFacotry 实现时),另外一个结果 jobs 实现类不能定义状态数据字段，因为这些状态数据字段的值在调用 job 任务时不会被保留。

你现在可能想问"我要怎样才能为 Job 实例提供配置参数？在执行任务时我要如何跟踪 job 对象的状态？"这两个问题的答案都一样：使用 JobDataMap，它是 JobDetail 对象的一部分。

3.1 JobDataMap

JobDataMap 可以用来装载任何可序列化的数据对象，当 job 实例对象被执行时这些参数对象会传递给它。JobDataMap 实现了 JDK 的 Map 接口，并且添加了一些非常方便的方法用来存取基本数据类型。

下面的代码片断演示了在定义/构建 JobDetail 对象时，在 job 对象添加到调度器之前，如何将数据存放至 JobDataMap 中：

````
// define the job and tie it to our DumbJob class
  JobDetail job = newJob(DumbJob.class)
      .withIdentity("myJob", "group1") // name "myJob", group "group1"
      .usingJobData("jobSays", "Hello World!")
      .usingJobData("myFloatValue", 3.141f)
      .build();
````

下面的例子显示了在 job 执行过程中如何从 JobDataMap 取数据：

~~~
public class DumbJob implements Job {

    public DumbJob() {
    }

    public void execute(JobExecutionContext context)
      throws JobExecutionException
    {
      JobKey key = context.getJobDetail().getKey();

      JobDataMap dataMap = context.getJobDetail().getJobDataMap();

      String jobSays = dataMap.getString("jobSays");
      float myFloatValue = dataMap.getFloat("myFloatValue");
```
~~~

````
      System.err.println("Instance " + key + " of DumbJob says: " + jobSays + ",
and val is: " + myFloatValue);
    }
  }
````

如果使用持久化的 JobStore(在教程 JobStore 部分讨论)，应该要多考虑放在 JobDataMap 中的数据对象，因为此时的对象会被序列化，因此这更容易出现类版本问题。显然官方版本的类很安全，但是非官方的版本，任何时候有人变更你序例化实例的类定义，都要注意不要破坏兼容性。更多关于此主题讨论的信息都能在 Java Developer Connection 技术贴士中找到：Serialization In The Real World。当然，可以选择将 JDBC-JobStore 和 JobDataMap 设计成只有基本数据类型和 String 类型才允许存储的 map 对象，从而从根本上消除序列化问题。

如果你在 job 类中添加 setter 方法对应 JobDataMap 的键值（例如 setJobSays(String val) 方法对应上面例子里的 jobSays 数据），Quartz 框架默认的 JobFactory 实现类在初始化 job 实例对象时会自动地调用这些 setter 方法，从而防止在调用执行方法时需要从 map 对象取指定的属性值。

触发器也可以关联 JobDataMap 对象，当存储在调度器中的 job 对象需要定期/重复执行，被多个触发器共用时，这种场景下使用 JobDataMap 将非常方便，然而每个独立的触发器，你都可以为 job 对象提供不同的输入参数。

在进行任务调度时 JobDataMap 存储在 JobExecutionContext 中非常方便获取。它整合了 JobDetail 和 Trigger 里的 JobDataMap 数据对象，后面的对象会把前面对象相同键值对象的值覆盖。

接下来的例子展示了任务执行过程中从 JobExecutionContext 取合并的 JobDataMap 数据：

````
public class DumbJob implements Job {

    public DumbJob() {
    }

    public void execute(JobExecutionContext context)
      throws JobExecutionException
    {
      JobKey key = context.getJobDetail().getKey();

      JobDataMap dataMap = context.getMergedJobDataMap();  // Note the difference
from the previous example

      String jobSays = dataMap.getString("jobSays");
      float myFloatValue = dataMap.getFloat("myFloatValue");
      ArrayList state = (ArrayList)dataMap.get("myStateData");
      state.add(new Date());

      System.err.println("Instance " + key + " of DumbJob says: " + jobSays + ",
and val is: " + myFloatValue);
````

````
    }
  }
````
或者如果你想在类中依赖 JobFactory 注入 map 数据，可以参照如下代码：
````
public class DumbJob implements Job {



    String jobSays;
    float myFloatValue;
    ArrayList state;

    public DumbJob() {
    }

    public void execute(JobExecutionContext context)
      throws JobExecutionException
    {
      JobKey key = context.getJobDetail().getKey();

      JobDataMap dataMap = context.getMergedJobDataMap();  // Note the difference
from the previous example

      state.add(new Date());

      System.err.println("Instance " + key + " of DumbJob says: " + jobSays + ",
and val is: " + myFloatValue);
    }

    public void setJobSays(String jobSays) {
      this.jobSays = jobSays;
    }

    public void setMyFloatValue(float myFloatValue) {
      myFloatValue = myFloatValue;
    }

    public void setState(ArrayList state) {
      state = state;
    }

  }
````
你可能会注意到类的整体代码比较长，但 execute 方法很简洁。有人会认为虽然代码比

较长，如果程序员的集成开发平台（IDE）自动生成 setter 方法的话，可以编写更少的代码，而不必手工编写那些单独的调用方法从 JobDataMap 中取值。你可以自主选择编写代码的方式。

3.2 Job 实例化

许多用户对 Job 实例对象确切的结构是什么疑惑了很长时间，将尝试在这为大家解答，并且在下一个板块讲述 Job 状态和并发机制。

你可以创建一个单独的 Job 实现类，创建多个不同的 JobDetails 实例，将不同 Job 实例定义存储在调度器中，每个 JobDetails 实例都有各自的参数和 JobDataMap，并且把这些 JobDetails 添加到调度器中。

例如：你创建一个 Job 接口的实现类，类名为“SalesReportJob”，Job 类可以预先传入一些假想的参数（通过 JobDataMap）来指定销售报表中业务员的名字。接下来创建多个 Job 实例的定义（即 JobDetails），如“SalesReportForJoe”和“SalesReportForMike”通过“Joe”和“Mike”指定到相应的 JobDataMaps 中作为参数输入到各自的 Job 对象中。

当触发器被触发时，相关的 JobDetail 实例会被加载，通过在调度器中配置的 JobFactory 会将关联的 Job 类实例化，默认的 JobFactory 只是在 Job 类中调用 newInstance 方法，然后尝试调用匹配 JobDataMap 键值的 setter 方法。你可以开发自己的 JobFactory 实现类通过应用 IOC 或 DI 机制完成 Job 实例的创建和初始化。

用 Quartz 框架的话来说，我们将每个存储的 JobDetail 称为 Job 定义或 JobDetail 实例，将每个执行的作业任务（Job）称为 Job 实例或 Job 定义实例。通常我们只用“job”单词来对应命名的 Job 定义或是 JobDetail。当我们指 Job 接口的实现类时，一般使用“job class”术语。

3.3 Job 状态与并发机制

现在介绍一些关于 Job 状态值和并发的信息。有一对加在 Job 类上面的注解，可以影响 Quartz 框架的这些方面的行为。

在 Job 类上使用@DisallowConcurrentExecution 注解，会告知 Quartz 不要并发执行相同 Job 定义创建的多个实例对象。

注意这里的措辞，要慎重地选择。引用上一章节的例子，如果 SalesReportJob 添加这个注解，在给定的时间段内只能执行一个 SalesReportJobForJoe 实例对象，但是可以并发执行 SalesReportJobForMike 实例。然而，在 Quartz 设计阶段决定在该类中携带注解，因为该注解会影响 JobDetail 类的编码。

在 Job 类上使用@PersistJobDataAfterExecution 注解，会告知 Quartz 成功执行完 execute 方法后（有异常抛出的情况除外）更新 JobDetail 的 JobDataMap 中存储的数据。例如同一个 JobDetail 下一次执行时将接收更新的值而不是初始值。跟@DisallowConcurrentExecution 注解类似，@PersistJobDataAfterExecution 注解适用于 Job 定义实例，而不是 Job 类实例。只是该注解是附着在 Job 类的成员变量中，因为它不会影响整个类的编码（例如 statefulness 只需要在 execute 方法代码内正确使用即可）。

如果使用@PersistJobDataAfterExecution 注解，强烈建议也应该考虑使用@DisallowConcurrentExecution 注解，为了避免当两个相同 JobDetail 实例并发执行时可能由于最后存储状态数据不一致导致执行混乱。

3.4 Jobs 的其它属性

接下来浏览 Job 实例的其它属性，这些属性是通过 JobDetail 对象传递给 Job 实例的。

1)Durability-如果 Job 是非持久化的，一旦没有任何活跃的触发器关联这个 Job 实例时，这个实例会自动地从调度器中移除。换句话说，非持久化的 jobs 的生命周期是以存在的触发器为界限的。

2）RequestsRecovery-如果一个 Job 设置了请求恢复参数，并且在调度器强制关闭过程中恰好在执行（强制关闭的情况例如：运行的线程崩溃，或者服务器宕机），当调度器重启时，它会重新被执行。这种情况下，JobExecutionContext 的 isRecovery 方法会返回 true。

3.5 JobExecutionException 异常

最后，我们需要告知你 Job.execute 方法的一些细节。允许从 execute 方法抛出的唯一一种异常类型是 JobExecutionException（运行时异常除外，可以正常抛出），由于这个限制，你应该在 execute 方法内的 try-catch 代码块中包装好要处理的异常。你也可以花些时间查阅一下 JobExecutionException 的文档，便于你在开发的 Job 类中需要捕获处理异常时，为调度器提供各种信息。

第四章 Triggers 详解

跟作业任务类似，触发器也非常易于使用，但是在能够充分掌握 Quartz 之前，需要知道并理解许多触发器的自定义参数。前面已经提到过，有许多不同类型的触发器供选择，适用不同的调度需求。

第五章和第六章将介绍两种常用的触发器类型。

4.1 通用的 Trigger 属性

所有类型的触发器都有 TriggerKey 属性用于跟踪触发器标识，除了这个外，还有许多其他的属性，对所有触发器类型都适用。这些通用属性在创建触发器定义时通过 TriggerBuilder 类来设定的（参考接下来的例子）。

下面的属性列表对所有类型的触发器都通用：

1)jobKey 表示 job 实例的标识，触发器被触发时，该指定的 job 实例会执行。

2)startTime 属性表示触发器的时间表首次被触发的时间。它的值是定义由指定日历时间的 Java.util.Date 对象。对于一些类型的触发器，会在启动时间触发，另一些触发器则仅仅是标示了调度器将要被触发的时间。这意味着你可以在调度器中存储一个触发器，例如每月的 5 号，如果现在是一月份，而 startTime 参数又设置为 4 月 1 日，那这样需要几个月后触发器才会第一次被触发。

3)endTime 属性指定触发器的不再被触发的时间，换言之，调度器中的触发器定义为“每月的 5 号”，而且 endTime 设置为 7 月 1 日，那么 6 月 5 日将会是最后一次触发的日期。

其它属性的讲解将会在接下来的子章节中进行讨论。

4.2 优先级（Priority）属性

有些时候，当创建了多个触发器（或 Quartz 线程池中只有少数几个工作线程），Quartz

可能没有足够的资源去触发所有的在同一时间段内排定好的触发器。既然这样，你可能期望控制哪个触发器能第一个获得 Quartz 空闲工作线程的调用。为了达到这个目的，你可以设定触发器的 Priority 属性。如果 N 个触发器在同一时间内被触发，但只有 Z 工作线程当前空闲可用，那么拥有最高优先级的 Z 触发器将会第一个被触发。如果你没有设置触发器的优先级，它将会使用默认的优先级，优先级值为 5。任何 Integer 类型的值都可以作为优先级，正数负数都可以（优先级数字越大触发器优先级越高，数字越小优先级越低）。

提示：优先级只是用于在同一时间被触发的触发器进行比较。一个安排在 10:59 分触发的触发器永远要比安排在 11:00 的触发器先执行。

提示：当一个触发器的作业任务发现设置了请求恢复参数，在恢复调度执行时的优先级和原来的一样。

4.3 触发失败指令（Misfire Instructions）

触发器另外一个重要的属性就是触发失败指令（Misfire Instructions）。触发失败的情况是由于调度器被关闭导致存储的触发器错过了触发的时间，或是由于 Quartz 线程池内没有空闲的线程去执行作业任务。不同类型的触发器有不同的触发失败处理机制。默认情况下使用“智能策略（smart policy）”指令——基于触发器类型和配置的动态机制。当调度器启动时，它会查询所有存储的、触发失败的触发器，然后根据各自配置的触发失败指令更新触发器。当你开始在你的项目中使用 Quartz 时，你应该熟悉在给定触发器类型上定义的触发失败指令和 JavaDoc 上的文档解释。更多关于触发失败指令的详细信息将会在每个触发器类型中作详细介绍。

4.3.1 日历对象（Calendars）

当触发器在调度器中创建和存储时，Quartz 日历对象（区别于 java.util.Calendar 对象）可以与触发器相关联。日历对触发器调度定义不包含的时间段非常方便。例如，你可以创建一个触发器，定义在每个工作日上午 9：30 分触发作业任务，另外添加一个日历表排除当中所有的假期。

Calendar 对象可以是实现 Calendar 接口的任何可序列化的对象，如下所示：

The 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是 long 类型。你可能已经猜到，这些是毫秒单位的时间戳格式。这意味着日历“屏蔽”的时间段可以精确到毫秒级。最有可能的是，我们喜欢“屏蔽”一整天的时间。为了方便起见，Quartz 包含的类 org.quartz.impl.HolidayCalendar，可以完成刚刚提及

的功能。

日历实例化和注册到调度器中必须通过 addCalendar 方法。如果你使用 HolidayCalendar，初始化对象完成后，应该使用 addExcludedDate(Date date)方法方便将你希望从调度时间表中排除的日期添加到日历实例对象中。同一个日历实例对象可以应用于多个触发器，如下代码：

Calendar 示例：

~~~
````
HolidayCalendar cal = new HolidayCalendar();
cal.addExcludedDate( someDate );
cal.addExcludedDate( someOtherDate );

sched.addCalendar("myHolidays", cal, false);

Trigger t = newTrigger()
    .withIdentity("myTrigger")
    .forJob("myJob")
    .withSchedule(dailyAtHourAndMinute(9, 30)) // execute job daily at 9:30
    .modifiedByCalendar("myHolidays") // but not on holidays
    .build();

// .. schedule job with trigger

Trigger t2 = newTrigger()
    .withIdentity("myTrigger2")
    .forJob("myJob2")
    .withSchedule(dailyAtHourAndMinute(11, 30)) // execute job daily at 11:30
    .modifiedByCalendar("myHolidays") // but not on holidays
    .build();

// .. schedule job with trigger2
````
~~~

关于触发器结构/构建详细信息将会在后面两节课中介绍。现在，只需要相信上面的代码创建了两个触发器对象，每个触发器每天都会被触发。然而，任何发生在日历对象中排除的日期内的触发将会跳过。

可以查阅 org.quartz.impl.calendar 包目录下的几个 Calendar 实现类，估计有适合你需要的类。

第五章 SimpleTrigger 简单触发器

如果你需要在一个指定时间段内执行一次作业任务或是在指定的时间间隔内多次执行作业任务，SimpleTrigger 能满足你的调度需求。例如，你希望触发器在 2015 年 1 月 13 日上午 11:23:54 准时触发，或是希望在那个时间点触发，然后再重复触发 5 次，每隔 10 秒一次。有了这样的描述，你就不会对 SimpleTrigger 包含的参数感到奇怪：开始执行时间，结束执行时间，重复次数和重复执行间隔时间。所有的参数都是你期望的那样，只是关于结束执行时间参数有两条特别的提示。

重复次数可以为 0，正整数或是 SimpleTrigger.REPEAT_INDEFINITELY 常量值。重复执行间隔必须为 0 或长整数（long 类型），它表示毫秒数的值。注意如果重复执行间隔时间为 0 会导致数量为“重复次数”的触发器并发执行（或是在调度器控制下接近并发执行）。

如果你还不熟悉 Quartz 的 DateBuilder 类，你尝试创建日期对象时会发现它非常方便地根据 startTime 或 endTime 参数计算触发器的触发时间。

endTime 参数（如果被指定）会覆盖重复次数参数的效果。当你希望创建一个触发器，每隔 10 秒被触发一次直到给定的截止时间，而不是必须完成在给定的开始和结束时间段内的触发次数。使用 endTime 参数会非常方便，你可以仅仅指定 end-time 参数，并且将重复次数设置为 REPEAT_INDEFINITELY（你甚至可以将重复次数指定为非常大的值，确保比结束执行时间到达前实际要执行的次数大就行）。

SimpleTrigger 实例对象可以使用 TriggerBuilder（针对触发器主要的参数）和 SimpleScheduleBuilder（针对 SimpleTrigger 的指定参数）来创建。为了使用这些创建类时满足 DSL 格式，使用静态导入：

````
import static org.quartz.TriggerBuilder.*;
import static org.quartz.SimpleScheduleBuilder.*;
import static org.quartz.DateBuilder.*:
````

下面是使用简易调度器定义触发器的几个案例，请通读一遍，每个例子都至少展示了一个新的、不同的知识点：

创建触发器时指定具体的时间，不重复执行：

````
  SimpleTrigger trigger = (SimpleTrigger) newTrigger()
    .withIdentity("trigger1", "group1")
    .startAt(myStartTime) // some Date
    .forJob("job1", "group1") // identify job with name, group strings
    .build();
````

创建触发器时指定具体的时间，然后每隔 10 秒触发一次，共重复触发 10 次：

````
  trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startAt(myTimeToStartFiring)  // if a start time is not given (if this line
were omitted), "now" is implied
    .withSchedule(simpleSchedule()
        .withIntervalInSeconds(10)
        .withRepeatCount(10)) // note that 10 repeats will give a total of 11
firings
    .forJob(myJob) // identify job with handle to its JobDetail itself
    .build();
````

创建触发器，5 分钟后将会触发一次：

````
  trigger = (SimpleTrigger) newTrigger()
````

```
    .withIdentity("trigger5", "group1")
    .startAt(futureDate(5, IntervalUnit.MINUTE)) // use DateBuilder to create a date in the future
    .forJob(myJobKey) // identify job with its JobKey
    .build();
```

创建触发器时立即触发，然后每隔 5 分钟触发一次，直到 22:00：

```
  trigger = newTrigger()
    .withIdentity("trigger7", "group1")
    .withSchedule(simpleSchedule()
        .withIntervalInMinutes(5)
        .repeatForever())
    .endAt(dateOf(22, 0, 0))
    .build();
```

创建触发器时，在下一个整点小时触发，然后每隔 2 小时触发一次，永不停歇：

```
  trigger = newTrigger()
    .withIdentity("trigger8") // because group is not specified, "trigger8" will be in the default group
    .startAt(evenHourDate(null)) // get the next even-hour (minutes and seconds zero ("00:00"))
    .withSchedule(simpleSchedule()
        .withIntervalInHours(2)
        .repeatForever())
    // note that in this example, 'forJob(..)' is not called
    //  - which is valid if the trigger is passed to the scheduler along with the job
    .build();

    scheduler.scheduleJob(trigger, job);
```

花点时间查阅 TriggerBuilder 和 SimpleScheduleBuilder 类的所有可用方法，以便你能熟悉在上面演示代码中可能没有展示的有用操作。

注意 TriggerBuilder（或是 Quartz 别的创建类）在你没有明确设置参数值时一般会选择合理的值。例如，如果你没有调用 withIdentity 方法，TriggerBuilder 会为你的触发器产生一个随机的名字，如果你没有调用 startAt 方法，它会假设是立即触发。

5.1 SimpleTrigger 触发失败指令

SimpleTrigger 有几条指令，用来告知 Quartz 当触发失败时该如何操作。（在第四章更多关于触发器已经介绍过触发失败的情况）。这些指令在 SimpleTrigger 类中设计成常量（包含 JavaDoc 描述了它们的行为）。指令有：

SimpleTrigger 的触发失败指令常量：

````
MISFIRE_INSTRUCTION_IGNORE_MISFIRE_POLICY
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
````

回顾前面的章节可以知道：所有的触发器都可以使用 Trigger.MISFIRE_INSTRUCTION_SMART_POLICY 指令，并且这条指令也是所有触发器的默认指令。

如果使用"智能策略（smart policy）"指令，SimpleTrigger 会从多条触发失败指令集中根据配置和 SimpleTrigger 实例的状态动态地选择指令。JavaDoc 文档中 SimpleTrigger 的 updateAfterMisfire 方法解释了动态选择行为更详细的信息。

当创建 SimpleTrigger 时，可以通过 SimpleSchedulerBuilder 指定触发失败指令作为调度器的一部分。

````
  trigger = newTrigger()
    .withIdentity("trigger7", "group1")
    .withSchedule(simpleSchedule()
        .withIntervalInMinutes(5)
        .repeatForever()
        .withMisfireHandlingInstructionNextWithExistingCount())
    .build();
````

第六章 CronTrigger 定时触发器

CronTrigger 比 SimpleTrigger 更常用，当你需要一个基于日历概念的作业调度器，而不是像 SimpleTrigger 那样精确指定间隔时间。

使用 CronTrigger，你可以这样指定触发时间表例如"每周五的中午"，或是"每周末的上午 9:30"，甚至是"一月份每周一、三、五上午 9:00 到 10:00 之间每 5 分钟"。

虽然如此，跟 SimpleTrigger 一样，CronTrigger 也需要指定 startTime 让调度器生效，指定 endTime 让调度器终止。

6.1 Cron 表达式

Cron 表达式用于配置 CronTrigger 实例。Cron 表达式实际上是由 7 个子表达式组成的字符串，描述了时间表的详细信息。这些子表达式用空格隔开，分别代表：

秒
分
小时
月份中的天数

月
星期中的天数
年（可选）

Cron 表达式的字符串例子：“0 0 12 ? * WED”，意思是“每周三的上午 12:00:00”。

每个 Cron 表达式都包含“和”、“或”的两种排列，例如，上一个例子中星期中的天数字段（显示的是“WED”）可以替换为“MON－FRI”，“MON,WED,FRI”，甚至是“MON-WED,SAT”。

通配符（“”）可用来表示该字段的任意值，因此“Month”在上面的例子中的月份字段表示“每个月”，“*”在星期中的天数字段由此明显是表示“一周的任何一天”。

所有的字段都定义了一套可用的值。这些值应该非常明显易懂——例如秒和分的值是从 0 到 59，小时的值是从 0 到 23.月份中的天数是从 0 到 31，但是你需要特别注意这个月实际上有多少天！月份的值指定在 0 和 11 之间，或者可以使用字符串 JAN，FEB，MAR，APR，MAY，JUN，JUL，AUG，SEP，OCT，NOV 和 DEC。星期中的天数的值指定在 1 到 7 之间（1 表示星期天），或是使用字符串 SUN，MON，TUE，WED，THU，FRI 和 SAT。

“/”字符可用来表示增量的值。例如，如果你在分钟字段写“0/15”，这表示“每次从一小时中的第 0 分钟开始，每隔 15 分钟触发”，如果你在分钟字段上写“3/20”，这表示“每次从一小时中的第 3 分钟开始，每隔 20 分钟触发”——换句话说，这跟在分钟字段上指定“3，23，43”是一样的。注意细微的区别：“/35”不是表示“每隔 35 分钟”，而是表示“每次从一小时中的第 0 分钟开始，每隔 35 分钟触发”，相当于指定“0，35”。

“?”字符允许出现在月份中的天数和星期中的天数字段中。它一般用来指定“不关心的值”。当你需要在这两个字段中的一个指定不确定的值是非常方便的，这个字符不能用在其他的字段中。可以查看下面的例子（或是 CronTrigger 的 JavaDoc 文档）获得更详细的说明。

“L”字符允许出现在月份中的天数和星期中的天数字段中。这个字符是“last”的缩写，但是在这两个字段中有不同的含义。例如，“L”字符出现在月份中的天数字段中表示“每月的最后一天”——1 月 31 日，平年的 2 月 28 日。如果该字符单独用在星期中的天数字段时，仅仅是表示“7”或是“SAT”。但是在星期中的天数字段中该字符用在其他值的后面，表示“每月的最后一个星期几”——例如“6L”或是“FRIL”都表示“每月的最后一个星期五”。你也可以指定每月最后一天的偏移数，例如“L－3”表示日历月份的最后三天。当你使用“L”字符时，最好不要使用排列值或是带范围的值，否则你会对结果感到意外和难以理解。

“W”字符用来指定给定日期的最近一个工作日（工作日指的是从周一到周五）。例如，如果你在月份中的天数字段的值指定为“15w”，这表示“离每月 15 号最近的工作日”。

“#”字符用来指定每月的第 N 个工作日，例如，星期中的天数字段的值为“6#3”或是“FRI#3”表示“每月的第三个星期五”。

下面演示了一些表达式的例子和含义——你可以在 org.quartz.CronExpression 的 JavaDoc 找到更多信息。

6.2 Cron 表达式实例

“0 0/5 * * * ?”

Cron 案例 1——仅仅表示每隔 5 分钟触发一次："0 0/5 * * * ?"

“10 0/5 * * * ?”

Cron 案例 2——表示每隔 5 分钟，在过了 10 秒后触发一次（例如上午 10:00:10，10:05:10 等）："10 0/5 * * * ?"

“0 30 10-13 ? * WED,FRI”

Cron 案例 3——表示每个周三到周五，在上午 10:30，11:30，12:30 和 13:30 分触发："0 30 10-13 ? * WED,FRI"

“0 0/30 8-9 5,20 * ?”

Cron 案例 4——表示每月从 5 号到 20 号，上午 8 时到 10 时之间的每半小时触发，注意这个触发器只在 8:00，8:30，9:00 和 9:30 分触发，上午 10:00 不会触发："0 0/30 8-9 5,20 * ?"

注意有些调度需求因太复杂例如“上午 9:00 到 10:00 之间的每 5 分钟，下午 1:00 到 10:00 的每 20 分钟”，而不能用单一的触发器来表示。这种情况的解决方案是创建两个简单的触发器，将它们注册到调度器中去运行同一个作业任务。

6.2.1 创建 CronTriggers

CronTrigger 实例对象可以使用 TriggerBuilder（针对 触发器主要的参数）和 CronScheduleBuilder（针对 CronTrigger 的指定参数）来创建。为了使用这些创建类时满足 DSL 格式，使用静态导入：

````
import static org.quartz.TriggerBuilder.*;
import static org.quartz.CronScheduleBuilder.*;
import static org.quartz.DateBuilder.*:
````

创建一个触发器，每天从上午 8 点到下午 5 点，每隔 2 分钟触发一次：

````
  trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .withSchedule(cronSchedule("0 0/2 8-17 * * ?"))
    .forJob("myJob", "group1")
    .build();
````

创建一个触发器，每天的上午 10:42 分触发一次：

````
  trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .withSchedule(dailyAtHourAndMinute(10, 42))
    .forJob(myJobKey)
    .build();
````

或者：

````
  trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .withSchedule(cronSchedule("0 42 10 * * ?"))
    .forJob(myJobKey)
    .build();
````

创建一个触发器，每逢星期三的上午 10:42 分触发一次，并且制定与系统默认不同地方的 TimeZone：
````

  trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .withSchedule(weeklyOnDayAndHourAndMinute(DateBuilder.WEDNESDAY, 10, 42))
    .forJob(myJobKey)
    .inTimeZone(TimeZone.getTimeZone("America/Los_Angeles"))
    .build();
````

or -
````

  trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .withSchedule(cronSchedule("0 42 10 ? * WED"))
    .inTimeZone(TimeZone.getTimeZone("America/Los_Angeles"))
    .forJob(myJobKey)
    .build();
````

6.2.2 CronTrigger 触发失败指令

CronTrigger 有几条指令，用来告知 Quartz 当触发失败时该如何操作。（在第四课更多关于触发器已经介绍过触发失败的情况）。这些指令在 CronTrigger 类中设计成常量（包含 JavaDoc 描述了它们的行为）。指令有：
CronTrigger 触发失败指令常数
````

MISFIRE_INSTRUCTION_IGNORE_MISFIRE_POLICY
MISFIRE_INSTRUCTION_DO_NOTHING
MISFIRE_INSTRUCTION_FIRE_NOW
````

所有的触发器都可以使用 Trigger.MISFIRE_INSTRUCTION_SMART_POLICY 指令，并且这条指令也是所有触发器的默认指令。“智能策略”指令可以从 CronTrigger 的 MISFIRE_INSTRUCTION_FIRE_NOW 当中获得解释。JavaDoc 文档中 CronTrigger 的 updateAfterMisfire 方法解释了动态选择行为的更详细的信息。

当创建 CronTrigger 时，可以通过 CronSchedulerBuilder 指令触发失败指令作为调度器的一部分。
````

  trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .withSchedule(cronSchedule("0 0/2 8-17 * * ?")
        ..withMisfireHandlingInstructionFireAndProceed())
    .forJob("myJob", "group1")
    .build();
````

~~~~~
````
~~~~~

第七章 TriggerListeners 和 JobListeners

监听器是在调度器中基于事件机制执行操作的对象。你大概可以猜到，触发监听器接收响应跟触发器有关的事件，作业任务监听器接收响应跟作业任务有关的事件。

跟触发器有关的事件包括：触发器被触发，触发器触发失败（在触发器一章讨论过），以及触发器触发完成（触发器完成后作业任务开始运行）。

org.quartz.TriggerListener 接口

````
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);
}
````

跟作业任务相关的事件包括：job 即将被执行的通知和 job 执行完成的通知事件。

org.quartz.JobListener 接口

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

7.1 自定义 Listeners

要创建一个监听器，可以简单地创建一个实现 org.quartz.TriggerListener 或 org.quartz.JobListener 接口的对象即可。监听器会在运行期间注册到调度器中，并且必须要给定监听器名（或者更确切地说，监听器会调用 getName 方法获取自己的名字）。

为了方便使用，监听器除了实现这些接口，你还可以继承 JobListenerSupport 和 TriggerListenerSupport 类，可以只重写你感兴趣的事件方法。

监听器注册到调度器中的监听器管理类时还携带着一个匹配器，这个匹配器描述了作业任务和触发器的监听器想接收的事件。

监听器在运行期间注册到调度中，但是不会把作业任务和触发器存储到 JobStore 中。那是因为监听器在你的应用中通常是一些点的集合。因此，每次应用运行时，监听器都需要重新在调度器中注册。

添加 JobListener 到部分 job 上：
````
scheduler.getListenerManager().addJobListener(myJobListener,
KeyMatcher.jobKeyEquals(new JobKey("myJobName", "myJobGroup")));
````
　　你可以静态导入 matcher 类和其他关键类，让你定义 matchers 时更简洁：
````
import static org.quartz.JobKey.*;
import static org.quartz.impl.matchers.KeyMatcher.*;
import static org.quartz.impl.matchers.GroupMatcher.*;
import static org.quartz.impl.matchers.AndMatcher.*;
import static org.quartz.impl.matchers.OrMatcher.*;
import static org.quartz.impl.matchers.EverythingMatcher.*;
...etc.
````
　　上面的例子将变成这样：
````
scheduler.getListenerManager().addJobListener(myJobListener,
jobKeyEquals(jobKey("myJobName", "myJobGroup")));
````
　　为一个特定的 group 中所有 jobs 添加 JobListener：
````
scheduler.getListenerManager().addJobListener(myJobListener,
jobGroupEquals("myJobGroup"));
````
　　为两个特定的 group 中所有的 jobs 添加 JobListener：
````
scheduler.getListenerManager().addJobListener(myJobListener,
or(jobGroupEquals("myJobGroup"), jobGroupEquals("yourGroup")));
````
　　在所有的 jobs 中添加 JobListener：
````
scheduler.getListenerManager().addJobListener(myJobListener, allJobs());
````
　　注册 Trigger 监听器也是同样的方法。

　　大部分的 Quartz 用户都不会用到监听器，但是当应用要求创建需要的事件通知时，而没有 Job 实例去通知应用时，使用监听器非常方便。

第八章 SchedulerListeners

　　调度监听器和触发监听器和触发监听器、作业任务监听器非常相似，只是调度监听器在调度器内接收通知事件，而不需要关联具体的触发器或作业任务事件。

　　跟调度监听器相关的事件，添加作业任务/触发器，移除作业任务/触发器，调度器发生严重错误，调度器关闭等。

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 schedulerStarted();
    public void schedulerInStandbyMode();
    public void schedulerShutdown();
    public void schedulingDataCleared();
}
````
调度监听器注册到调度器的监听管理器中，调度监听器实际上可以是实现 org.quartz.SchedulerListener 接口的任何对象。

添加 SchedulerListener：
````
scheduler.getListenerManager().addSchedulerListener(mySchedListener);
````
移除 SchedulerListener:
````
scheduler.getListenerManager().removeSchedulerListener(mySchedListener);
````

第九章 JobStores

JobStore 是负责跟踪调度器中所有的工作数据:作业任务、触发器、日历等。为你的 Quartz 调度器实例选择一个适当的 JobStore 是非常重要的一步。幸运的是，一旦你理解了这些 JobStore 之间的区别，选择它们是非常容易的事。你可以在配置文件（或是类对象）中定义调度器使用哪种 JobStore，这个 JobStore 将会提供给 SchedulerFactory，用来创建你的调度器实例。

不要在代码中直接使用 JobStore 实例，因为一些原因可能很多开发者会尝试这样做。JobStore 是给 Quartz 在幕后使用的。你只需要通过配置信息告知 Quartz 该用哪个 JobStore，然后在代码里只需要使用调度器接口即可。

9.1 RAMJobStore

RAMJobStore 是最容易使用的 JobStore，它也是最高效的（从 CPU 时间计算）。从 RAMJobStore 的名字可以明显地发现：它将所有数据存储在 RAM 中。这就是为什么它速度

快并且配置简单的原因。缺点是当你的应用终止（或是崩溃）时，所有的调度信息都会丢失——这意味着 RAMJobStore 会导致作业任务和触发器的 non-volatility 设置不起作用。对某些应用来说，这种缺点可以接受，甚至需要这样的特性，但对另外一些应用来说，这种缺点可能会成为一个灾难。

假定你正在使用 StdSchedulerFactory，想要使用 RAMJobStore 时，只需要在 Quartz 配置文件中将 JobStore class 参数的值指定为 org.quartz.simpl.RAMJobStore 的类全名即可：

Quartz 配置文件中使用 RAMJobStore：

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

就再没有其他配置了。

9.2 JDBCJobStore

JDBCJobStore 同样人如其名——它通过 JDBC 将所有的数据保存在数据库中。正因为如此配置 JDBCJobStore 要比 RAMJobStore 复杂许多，并且也没 RAMJobStore 速度快。然而，后台的执行效率也不是非常糟糕，特别是如果你在数据库的主键中创建索引，就可以提高效率。比较好的局域网环境内的现代标准配置的机器，恢复和更新一个触发器所需要花费的时间一般在 10 毫秒以内。

JDBCJobStore 几乎可以适用于任何数据库，它已经在 Oracle,PostgreSQL, MySQL, MS SQLServer, HSQLDB 和 DB2 数据库中广泛使用。为了使用 JDBCJobStore，你必须先创建一套数据库表供 Quartz 使用。你可以在 Quartz 发布包的“docs/dbTables”目录找到建表 SQL 脚本。如果现成的脚本不适合你的数据库类型，找到其中一个脚本，想尽一切必要的方法修改成适合你的数据库。需要注意一点是在这些脚本中，所有表的前缀都是“QRTZ_”（例如表“QRTZ_TRIGGERS”和“QRTZ_JOB_DETAIL”）。这个前缀实际上可以是任何你想要的。只要你告诉 JDBCJobStore 前缀是什么（在 Quartz 配置文件里）。使用不同的前缀可能用来在同一个数据库中创建多套数据表，供多个调度器实例使用。

一旦创建了数据表，在配置和触发 JDBCJobStore 之前你需要作多个重要决定。你需要决定你的应用需要什么类型的事务。如果你不需要把调度命令（例如添加和移除触发器）和其他事务捆绑在一起，那么你就让 Quartz 使用 JobStoreTx 作为 JobStore 管理事务（这个是最常用的选择）。

如果你需要 Quartz 关联其他事务（例如在 J2EE 应用服务器中），然后你应该使用 JobStoreCMT——这种情况下 Quartz 会让应用服务容器管理事务。

最后一件难题是从 JDBCJobStore 中设置 DataSource，用来获得数据库的连接。数据源在 Quartz 配置中定义，从几种不同方式中选择其中一种。一种方式是 Quartz 自己创建和管理数据源——通过提供所有的数据库连接信息。另一种方式是 Quartz 使用由 Quartz 运行所在的应用服务器中的数据源——将数据源的 JNDI 名字提供给 JDBCJobStore。想了解更多参数的详细信息，请查阅“docs/config”文件夹的示例配置文件。

假定你正在使用 StdSchedulerFactory，想要使用 JDBCJobStore 时，只需要在 Quartz 配置文件中将 JobStore class 参数的值指定为 org.quartz.impl.jdbcjobstore.JobStoreTX 或 org.quartz.impl.jdbcjobstore.JobStoreCMT 的类全名即可——你可以根据上面几段文字中的解释决定你的选择。

Quartz 配置文件中使用 JobStoreTx

````
````

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

接下来，你需要选择 DriverDelegate 给 JobStore 使用。DriverDelegate 负责完成任何 JDBC 的工作，它需要和指定的数据库类型对应。StdJDBCDelegate 是使用“vanilla”JDBC 代码和 SQL 语句完成工作的一种 delegate。如果没有专门与你的数据库对应的 delegate，尝试使用这个 delegate——我们只是为数据库开发对应的 delegates，并且已经发现使用 StdJDBCDelegate 的问题（好像是问题最多的）。其他的 delegates 可以在 org.quartz.impl.jdbcjobstore 包或子包中找到。其他的 delegates 包括 DB2v6Delegate(DB2 6 及更早版本), HSQLDBDelegate (HSQLDB 数据库), MSSQLDelegate(微软 SQLServer 数据库),PostgreSQLDelegate (PostgreSQL 数据库), WeblogicDelegate (使用 Weblogic 的 JDBC 的数据库), OracleDelegate(Oracle 数据库)和其他。

一旦你选择了 delegates，设置 delegate 的类全名给 JDBCJobStore 使用。

使用 DriverDelegate 配置 JDBCJobStore

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

接下来，你需要通知 JobStore 你使用的表前缀（前面讨论过）配置 JDBCJobStore 的表前缀

````
使用表前缀配置 JDBCJobStore
````

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

最后，你需要设置 JobStore 使用哪个数据源。数据源的名字也必须在 Quartz 配置中定义。既然这样，我们指定 Quartz 的数据源的名字为“myDS”（即在配置文件另外的参数中定义）。

配置 JDBCJobStore 的数据源名字：

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

如果调度器繁忙（例如几乎总是执行跟线程池一样多的作业任务），你可能需要设置数据源的连接数，大约比线程池数多 2 个。

org.quartz.jobStore.useProperties 配置参数可以设置为“true”（默认为 false），为了通知 JDBCJobStore 所有在 JobDataMaps 的值都会为 String 类型，因此可以作为键值对存储，而不是在 BLOB 列中存储序列化的对象。这从长远看来更安全，例如你可以避免将非 String 类对象序列化到 BLOB 中导致的类版本问题。

9.3 TerracottaJobStore

TerracottaJobStorer 提供了一种不需要使用数据的可伸缩，健壮的方案。这意味着数据库在 Quartz 方面可以保持空载，而是将所有的资源保存在应用的其他部分中。

TerracottaJobStore 可以在集群或非集群环境中运行，在任何一种环境下，应用服务器重启期间都提供一个永久存储 job 数据的介质，因为这些数据存储在 Terracotta 服务器中。这个比通过 JDBCJobStore 使用数据库的方式高效(大约高一个数量级)，但比 RAMJobStore 慢。

假定你正在使用 StdSchedulerFactory，想要使用 TerracottaJobStore 时，只需要在 Quartz 配置文件中将类名 JobStore class 参数的值指定为 org.quartz.jobStore.class= org.terracotta.quartz.TerracottaJobStore，并且额外加一行配置指定 Terracotta 服务器地址。

使用 Terracotta 配置 Quartz

````
org.quartz.jobStore.class = org.terracotta.quartz.TerracottaJobStore
org.quartz.jobStore.tcConfigUrl = localhost:9510
````

更多的有关 JobStore 和 Terracotta 的信息能在 http://www.terracotta.org/quartz 中找到。

第十章 配置、资源使用和 SchedulerFactory

Quartz 框架的结构是模块化的，因此它运行的几个组件需要耦合在一起。幸运的是，一些帮助类可以完成这些工作。

Quartz 主要的组件如下：

ThreadPool

JobStore

DataSources (如果需要)

Scheduler 自身

ThreadPool 提供一组线程给 Quartz 执行作业任务时使用。池里有越多的线程，能并发执行的作业任务数量就越多。然而，太多的线程可能会导致系统挂死。大多数 Quartz 用户发现大约 5 个线程是比较合适的——在任何时候作业任务数都小于 100，所有的作业任务一般不会在同一个时间被调度，并且作业任务生命周期短（很快就能完成）。其他人认为他们需要 10，15，50 甚至 100 个线程——他们有成千上万个触发器用于各种各样的调度——最后在任何时候都是平均在 10 到 100 个作业任务同时执行。寻求调度线程池正确的大小完全依赖于你使用什么样的调度。这里没有固定的规则，而是保留尽可能少的线程浸透（为了你的计算资源）——但是要确保有足够的线程让作业任务准时起动。注意如果触发器要触发的时间到了，但是没有可用的线程，Quartz 会让该触发器阻塞直到有可用的线程，然后作业任务才会执行——这样会比本应该执行的时间延迟几毫秒。这里有可能会导致触发失败——如果在 Quartz 配置的“misfire threshold”期间一直没有可用的线程。

ThreadPool 接口在包 org.quartz.spi 内定义，你可以用任何方式创建 ThreadPool 的实现类。Quartz 附带了一个简单的线程池名为 org.quartz.simpl.SimpleThreadPool（但是也能满足需要）。这个线程池简单包含固定数量的线程数——不会增加，也不会减少。但是它在其他方面特别健壮，并且经过很好的测试——几乎每个使用 Quartz 的人都用这个线程池。

JobStores 和数据源在教程中的第九章讨论过。这里需要注意，事实上所有的 JobStores 都实现 org.quartz.spi.JobStore 接口，如果这些 JobStores 没有一个适合你的需要，你可以自行开发一个。

最后，你需要创建一个调度器实例。该实例需要指定一个名字，告知它 RIM 的设置和持有的 JobStore 和 ThreadPool 实例。RMI 设置包括调度器是否作为 RMI 服务对象（设置允许远程连接），并设置连接的主机名和端口号等。StdSchedulerFactory（下面将讨论）也可以产生调度器实例，实际上是代理 RMI 端在远程进程中创建调度器实例。

10.1 StdSchedulerFactory

StdSchedulerFactory 是 org.quartz.SchedulerFactory 接口的实现类。它使用一组参数（Java.util.Properties）来创建和初始化 Quartz 调度器。这些参数一般存储在文件中，并且从文件中加载，也可以直接在程序中创建直接赋值给工厂类。在工厂类中简单地调用 getScheduler 方法就能创建和初始化调度器对象（包括它的 ThreadPool，JobStore 和数据源），并返回一个引用（也叫句柄）到它的公共接口。

在 Quartz 发布包的“docs/config”目录里有一些示例配置（包含参数说明）。你可以在 Quartz 文档的参考板块下的配置手册里找到完整的文档。

10.2 DirectSchedulerFactory

DirectSchedulerFactory 是另一个 SchedulerFactory 实现类，它适用于希望用更多的编程方式创建调度器实例。一般不鼓励使用它主要有以下原因：（1）它要求使用都对他们想做的事情有比较深的理解，（2）它不允许声明式配置，换句话说，你必须编写代码来完成调度器的所有参数设置。

10.3 Logging

Quartz 选用 SLF4J 框架来满足所有的日志需求。为了“调整”日志设置（例如日志输出量，日志输出位置），你需要去了解 SLF4J 框架，这个框架不在此文档内。

如果你想了解触发器触发和作业任务执行的额外信息，你可能会对 org.quartz.plugins.history.LoggingJobHistoryPlugin 和 / 或 org.quartz.plugins.history.LoggingTriggerHistoryPlugin 感兴趣。

第十一章 高级功能（企业级）

11.1 集群

使用 JDBC-JobStore（JobStoreTx 或 JobStoreCMT）和 TerracottaJobStore 可以集群并发工作。特性包含负载均衡和作业任务故障切换（当 JobDetails 的“请求恢复”标记设置为 true）。

设置“org.quartz.jobStore.isClustered”参数为 true 可以开启集群功能。集群中的每个实例都应该使用相同的 quartz.properties 文件。使用相同的属性文件后可以有不同的属性，下面的几条允许例外：不同的线程池大小，“org.quartz.scheduler.instanceId”属性值可以不相同。集群的每一个节点必须有独一无二的 instanceId，把该属性值设置为“AUTO”就可以很容易达到这个要求。（不需要不同的属性文件）

不要在不同的计算机上运行 Quartz 集群，除非它们的时钟是使用时间同步服务（后台服务）同步过的，运行非常有规律（每台计算机的时间误差在 1 秒内）。如果你不熟悉怎么同步可以看这里：http://www.boulder.nist.gov/timefreq/service/its.htm。

不要在集群中任何节点运行的一套表中存在非集群实例的运行。这可能会导致严重的数据问题，并且肯定会遇到不稳定的行为。

每次只会有一个节点的作业任务被触发。我的意思是，如果一个作业任务有重复的触发器显示每 10 秒触发一次，然后在 12:00:00 准点时集群中的一个节点会执行作业任务，在 12:00:10 时也是一个节点执行作业任务。它不一定每次都是同一个节点——它或多或少会随机选择一个节点去执行。负载均衡机制是就近随机的忙碌调度（大量的触发器）而倾向于选

取活跃而不繁忙调度（例如只有一两个触发器）的节点。

使用 TerracottaJobStore 简单地配置调度器（在第九章：JobStores 讲过），你的调度器将全部设置为集群。

你可能还需要考虑如何设置您的 Terracotta 服务器的影响，尤其是打开功能的配置选项如持久化，运行 Terracotta 服务器的高可用性集群。

企业版本的 TerracottaJobStore 提供高级 Quartz 功能，即允许作业任务智能定位到合适的集群节点上。

更多关于 JobStore 和 Terracotta 的信息可以参照：http://www.terracotta.org/quartz。

11.2 JTA 事务

从第九章：JobStores 介绍可知，JobStoreCMT 允许 Quartz 调度操作在大型 JTA 事务中执行。

当"org.quartz.scheduler.wrapJobExecutionInUserTransaction"属性设置为 true 时，作业任务也可以在一个 JTA 事务(用户事务)中执行。使用此选项设置,JTA 事务将在作业任务 execute 方法调用前开始，并且 execute 结束之后立即提交。这适用于所有作业任务。

如果你想标明每个作业任务的执行是否嵌套一个 JTA 事务，你可以在 job 类中使用注解 @ExecuteInJTATransaction。

除了 Quartz 自动地将作业任务执行嵌套进 JTA 事务，当你的调度器接口使用 JobStoreCMT 时，也可以加入事务。只是要确保你在调度器调用方法前要启动一个事务。你可以通过使用 UserTransaction 直接执行此操作，或者把你的代码放到有管理事务容器的 SessionBean 中的调度器里。

第十二章 Quartz 的其他功能

12.1 插件

Quartz 提供 org.quartz.spi.SchedulerPlugin 接口用作插件附加功能开发。

Quartz 附带的插件提供了多种实用的功能，可以发现记录在 org.quartz.plugins 包中。他们提供的功能例如调度器启动时自动调用作业任务，记录作业任务和触发器事件的历史记录，当 JVM 退出时确保调度器完全关闭。

12.2 JobFactory

当触发器被触发时，通过在调度器配置的 JobFactory 会将关联的 Job 类实例化。默认的 JobFactory 只是在 Job 类中调用 newInstance 方法。你可能需要创建自己的 JobFactory 实现类去完成诸如应用服务的 IOC 或 DI 容器创建/初始化 job 实例。

可以查阅 org.quartz.spi.JobFactory 接口和关联的 Scheduler.setJobFactory(fact)方法。

12.3 ‘Factory-Shipped’ Jobs

Quartz 也提供一些通用的作业任务，你可以在你的应用中使用它们来完成一些功能如发送邮件和调用 EJB。这些常用的作业任务可以在 org.quartz.jobs 包中找到。

Quartz Cookbook

http://www.quartz-scheduler.org/documentation/quartz-2.2.x/cookbook/MonthlyTrigger.html

### Quartz Job Scheduler Cookbook
The Quartz cookbook is a collection of succinct code examples of doing specific things with Quartz.

The examples assume you have used static imports of Quartz’s DSL classes such as these:
````
import static org.quartz.JobBuilder.*;
import static org.quartz.TriggerBuilder.*;
import static org.quartz.SimpleScheduleBuilder.*;
import static org.quartz.CronScheduleBuilder.*;
import static org.quartz.CalendarIntervalScheduleBuilder.*;
import static org.quartz.JobKey.*;
import static org.quartz.TriggerKey.*;
import static org.quartz.DateBuilder.*;
import static org.quartz.impl.matchers.KeyMatcher.*;
import static org.quartz.impl.matchers.GroupMatcher.*;
import static org.quartz.impl.matchers.AndMatcher.*;
import static org.quartz.impl.matchers.OrMatcher.*;
import static org.quartz.impl.matchers.EverythingMatcher.*;
````

### How-To: Instantiating a Scheduler
#### Instantiating the Default Scheduler
````
// the ’default’ scheduler is defined in ″quartz.properties″ found
// in the current working directory, in the classpath, or
// resorts to a fall-back default that is in the quartz.jar

SchedulerFactory sf = new StdSchedulerFactory();
Scheduler scheduler = sf.getScheduler();

// Scheduler will not execute jobs until it has been started (though they can be scheduled before start())
scheduler.start();
````

#### Instantiating A Specific Scheduler From Specific Properties
````
StdSchedulerFactory sf = new StdSchedulerFactory();
````

sf.initialize(schedulerProperties);

Scheduler scheduler = sf.getScheduler();

// Scheduler will not execute jobs until it has been started (though they can be scheduled before start())
scheduler.start();
````

Instantiating A Specific Scheduler From A Specific Property File
````
StdSchedulerFactory sf = new StdSchedulerFactory();

sf.initialize(fileName);

Scheduler scheduler = sf.getScheduler();

// Scheduler will not execute jobs until it has been started (though they can be scheduled before start())
scheduler.start();
````
点评：scheduler工厂能不加配置启动，也能从xml配置文件启动，也能配置普通文件启动。

How-To: Placing a Scheduler in Stand-by Mode
Placing a Scheduler in Stand-by Mode
````
// start() was previously invoked on the scheduler

scheduler.standby();

// now the scheduler will not fire triggers / execute jobs

// ...

scheduler.start();

// now the scheduler will fire triggers and execute jobs
````
点评：使用.standby()方法能够在scheduler启动前做一些事情。

How-To: Shutting Down a Scheduler
To shutdown / destroy a scheduler, simply call one of the shutdown(..) methods.

Once you have shutdown a scheduler, it cannot be restarted (as threads and other
````

resources are permanently destroyed). Also see the suspend method if you wish to simply pause the scheduler for a while.
#### Wait for Executing Jobs to Finish
````
//shutdown() does not return until executing Jobs complete execution
scheduler.shutdown(true);
````

#### Do Not Wait for Executing Jobs to Finish
````
//shutdown() returns immediately, but executing Jobs continue running to completion
scheduler.shutdown();
//or
scheduler.shutdown(false);
````

If you are using the org.quartz.ee.servlet.QuartzInitializerListener to fire up a scheduler in your servlet container, its contextDestroyed() method will shutdown the scheduler when your application is undeployed or the application server shuts down (unless its shutdown-on-unload property has been explicitly set to false).
点评：使用.shutdown(x)关闭scheduler。

### How-To: Initializing a scheduler within a servlet container
There are two approaches for this which are shown below.

For both cases, make sure to look at the JavaDOC for the related classes to see all possible configuration parameters, as a complete set is not show below.
#### Adding A Context/Container Listener To web.xml
````
...
     <context-param>
         <param-name>quartz:config-file</param-name>
         <param-value>/some/path/my_quartz.properties</param-value>
     </context-param>
     <context-param>
         <param-name>quartz:shutdown-on-unload</param-name>
         <param-value>true</param-value>
     </context-param>
     <context-param>
         <param-name>quartz:wait-on-shutdown</param-name>
         <param-value>false</param-value>
     </context-param>
     <context-param>
         <param-name>quartz:start-scheduler-on-load</param-name>
````

```
        <param-value>true</param-value>
    </context-param>
...
    <listener>
        <listener-class>
            org.quartz.ee.servlet.QuartzInitializerListener
        </listener-class>
    </listener>
...
```

#### Adding A Start-up Servlet To web.xml
```

...
        <servlet>
          <servlet-name>QuartzInitializer</servlet-name>
          <servlet-class>org.quartz.ee.servlet.QuartzInitializerServlet</servlet-
class>
          <init-param>

            <param-name>shutdown-on-unload</param-name>
            <param-value>true</param-value>
          </init-param>
          <load-on-startup>2</load-on-startup>

        </servlet>
...
```

### How-To: Using Multiple (Non-Clustered) Schedulers
Reasons you may want to do this:

For managing resources - e.g. if you have a mix of light-weight and heavy-weight jobs, then you may wish to have a scheduler with many threads to service the lightweight jobs and one with few threads to service the heavy-weight jobs, in order to keep your machines resources from being overwhelmed by running to many heavy-weight jobs concurrently.
To schedule jobs in one application, but have them execute within another (when using JDBC-JobStore).

Note that you can create as many schedulers as you like within any application, but they must have unique scheduler names (typically defined in the quartz.properties file). This means that you'll need to have multiple properties files, which means that you'll need to specify them as you initialize the StdSchedulerFactory (as it only defaults to finding “quartz.properties”).

If you run multiple schedulers they can of course all have distinct characteristics – e.g. one may use RAMJobStore and have 100 worker threads, and another may use JDBC-JobStore and have 20 worker threads.

Never start (scheduler.start()) a non-clustered instance against the same set of database tables that any other instance with the same scheduler name is running (start()ed) against. You may get serious data corruption, and will definitely experience erratic behavior.

#### Example/Discussion Relating To Scheduling Jobs From One Application To Be Executed In Another Application

This description/usage applies to JDBC-JobStore. You may also want to look at RMI or JMX features to control a Scheduler in a remote process - which works for any JobStore. You may also be interested in the Terracotta Quartz Where features.

Currently, If you want to have particular jobs run in a particular scheduler, then it needs to be a distinct scheduler - unless you use the Terracotta Quartz Where features.

Suppose you have an application “App A” that needs to schedule jobs (based on user input) that need to run either on the local process/machine “Machine A” (for simple jobs) or on a remote machine “Machine B” (for complex jobs).

It is possible within an application to instantiate two (or more) schedulers, and schedule jobs into both (or more) schedulers, and have only the jobs placed into one scheduler run on the local machine. This is achieved by calling scheduler.start() on the scheduler(s) within the process where you want the jobs to execute. Scheduler.start() causes the scheduler instance to start processing the jobs (i.e. start waiting for trigger fire times to arrive, and then executing the jobs). However a non-started scheduler instance can still be used to schedule (and retrieve) jobs.

For example:

In “App A” create “Scheduler A” (with config that points it at database tables prefixed with “A”), and invoke start() on “Scheduler A”. Now “Scheduler A” in “App A” will execute jobs scheduled by “Scheduler A” in “App A”
In “App A” create “Scheduler B” (with config that points it at database tables prefixed with “B”), and DO NOT invoke start() on “Scheduler B”. Now “Scheduler B” in “App A” can schedule jobs to be ran where “Scheduler B” is started.
In “App B” create “Scheduler B” (with config that points it at database tables

prefixed with “B”), and invoke start() on “Scheduler B”. Now “Scheduler B” in “App B” will execute jobs scheduled by “Scheduler B” in “App A”.

### How-To: Defining a Job (with input data)
#### A Job Class
````
public class PrintPropsJob implements Job {

	public PrintPropsJob() {
		// Instances of Job must have a public no-argument constructor.
	}

	public void execute(JobExecutionContext context)
			throws JobExecutionException {

		JobDataMap data = context.getMergedJobDataMap();
		System.out.println("someProp = " + data.getString("someProp"));
	}

}
````

#### Defining a Job Instance
````
// Define job instance
JobDetail job1 = newJob(MyJobClass.class)
    .withIdentity("job1", "group1")
    .usingJobData("someProp", "someValue")
    .build();
````

Also note that if your Job class contains setter methods that match your JobDataMap keys (e.g. “setSomeProp” for the data in the above example), and you use the default JobFactory implementation, then Quartz will automatically call the setter method with the JobDataMap value, and there is no need to have code in the Job’s execute method that retrieves the value from the JobDataMap.

### How-To: Scheduling a Job
````

// Define job instance
JobDetail job1 = newJob(ColorJob.class)
    .withIdentity("job1", "group1")
    .build();
````

// Define a Trigger that will fire "now", and not repeat
Trigger trigger = newTrigger()
    .withIdentity("trigger1", "group1")
    .startNow()
    .build();

// Schedule the job with the trigger
sched.scheduleJob(job, trigger);
````

How-To: Unscheduling a Job
Unscheduling a Particular Trigger of Job
````
// Unschedule a particular trigger from the job (a job may have more than one trigger)
scheduler.unscheduleJob(triggerKey("trigger1", "group1"));
````

Deleting a Job and Unscheduling All of Its Triggers
````
// Schedule the job with the trigger
scheduler.deleteJob(jobKey("job1", "group1"));
````

How-To: Storing a Job for Later Use
Storing a Job
````
// Define a durable job instance (durable jobs can exist without triggers)
JobDetail job1 = newJob(MyJobClass.class)
    .withIdentity("job1", "group1")
    .storeDurably()
    .build();

// Add the the job to the scheduler's store
sched.addJob(job, false);
````

How-To: Scheduling an already stored job
Scheduling an already stored job
````
// Define a Trigger that will fire "now" and associate it with the existing job
Trigger trigger = newTrigger()
    .withIdentity("trigger1", "group1")

~~~~~
    .startNow()
    .forJob(jobKey("job1", "group1"))
    .build();

// Schedule the trigger
sched.scheduleJob(trigger);
````


How-To: Update an existing job
Update an existing job
````
// Add the new job to the scheduler, instructing it to "replace"
//  the existing job with the given name and group (if any)
JobDetail job1 = newJob(MyJobClass.class)
    .withIdentity("job1", "group1")
    .build();

// store, and set overwrite flag to 'true'
scheduler.addJob(job1, true);
````


How-To: Updating a trigger
Replacing a trigger
````
// Define a new Trigger
Trigger trigger = newTrigger()
    .withIdentity("newTrigger", "group1")
    .startNow()
    .build();

// tell the scheduler to remove the old trigger with the given key, and put the new
one in its place
sched.rescheduleJob(triggerKey("oldTrigger", "group1"), trigger);
````


Updating an existing trigger
````
// retrieve the trigger
Trigger oldTrigger = sched.getTrigger(triggerKey("oldTrigger", "group1");

// obtain a builder that would produce the trigger
TriggerBuilder tb = oldTrigger.getTriggerBuilder();

// update the schedule associated with the builder, and build the new trigger
~~~~~

```
// (other builder methods could be called, to change the trigger in any desired
way)
Trigger newTrigger = tb.withSchedule(simpleSchedule()
    .withIntervalInSeconds(10)
    .withRepeatCount(10)
    .build();

sched.rescheduleJob(oldTrigger.getKey(), newTrigger);
```

### How-To: Initializing Job Data With Scheduler Initialization

You can initialize the scheduler with predefined jobs and triggers using the XMLSchedulingDataProcessorPlugin (which, with the 1.8 release, replaced the older JobInitializationPlugin). An example is provided in the Quartz distribution in the directory examples/example10. However, following is a short description of how the plugin works.

First of all, we need to explicitly specify in the scheduler properties that we want to use the XMLSchedulingDataProcessorPlugin. This is an excerpt from an example quartz.properties:

```
#===================================================
# Configure the Job Initialization Plugin
#===================================================

org.quartz.plugin.jobInitializer.class =
org.quartz.plugins.xml.XMLSchedulingDataProcessorPlugin
org.quartz.plugin.jobInitializer.fileNames = jobs.xml
org.quartz.plugin.jobInitializer.failOnFileNotFound = true
org.quartz.plugin.jobInitializer.scanInterval = 10
org.quartz.plugin.jobInitializer.wrapInUserTransaction = false
```

Let's see what each property does:
1)fileNames: a comma separated list of filenames (with paths). These files contain the xml definition of jobs and associated triggers. We'll see an example jobs.xml definition shortly.
2)failOnFileNotFound: if the xml definition files are not found, should the plugin throw an exception, thus preventing itself (the plugin) from initializing?
3)scanInterval: the xml definition files can be reloaded if a file change is detected. This is the interval (in seconds) the files are looked at. Set to 0 to disable scanning.
4)wrapInUserTransaction: if using the XMLSchedulingDataProcessorPlugin with JobStoreCMT, be sure to set the value of this property to true, otherwise you might

experience unexpected behavior.

The jobs.xml file (or any other name you use for it in the fileNames property) declaratively defines jobs and triggers. It can also contain directive to delete existing data. Here's a self-explanatory example:
````
<?xml version='1.0' encoding='utf-8'?>
<job-scheduling-data xmlns="http://www.quartz-scheduler.org/xml/JobSchedulingData"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.quartz-scheduler.org/xml/JobSchedulingData
http://www.quartz-scheduler.org/xml/job_scheduling_data_1_8.xsd"
  version="1.8">

    <schedule>
        <job>
            <name>my-very-clever-job</name>
            <group>MYJOB_GROUP</group>

            <description>The job description</description>
            <job-class>com.acme.scheduler.job.CleverJob</job-class>
            <job-data-map allows-transient-data="false">

                <entry>
                    <key>burger-type</key>
                    <value>hotdog</value>
                </entry>
                <entry>

                    <key>dressing-list</key>
                    <value>ketchup,mayo</value>
                </entry>
            </job-data-map>
        </job>

        <trigger>
            <cron>
                <name>my-trigger</name>
                <group>MYTRIGGER_GROUP</group>
                <job-name>my-very-clever-job</job-name>

                <job-group>MYJOB_GROUP</job-group>
                <!-- trigger every night at 4:30 am -->
                <!-- do not forget to light the kitchen's light -->
                <cron-expression>0 30 4 * * ?</cron-expression>
            </cron>
        </trigger>
    </schedule>
</job-scheduling-data>
````

A further jobs.xml example is in the examples/example10 directory of the Quartz distribution.

Checkout the XML schema for full details of what is possible.

### How-To: Listing Jobs in the Scheduler
#### Listing all Jobs in the scheduler
````
// enumerate each job group
for(String group: sched.getJobGroupNames()) {
    // enumerate each job in group
    for(JobKey jobKey : sched.getJobKeys(groupEquals(group))) {
        System.out.println("Found job identified by: " + jobKey);
    }
}
````

### How-To: Listing Triggers In Scheduler
#### Listing all Triggers in the scheduler
````
// enumerate each trigger group
for(String group: sched.getTriggerGroupNames()) {
    // enumerate each trigger in group
    for(TriggerKey triggerKey : sched.getTriggerKeys(groupEquals(group))) {
        System.out.println("Found trigger identified by: " + triggerKey);
    }
}
````

### How-To: Finding Triggers of a Job
#### Finding Triggers of a Job
````
List<Trigger> jobTriggers = sched.getTriggersOfJob(jobKey("jobName", "jobGroup"));
````

###How-To: Using Job Listeners
####Creating a JobListener

Implement the JobListener interface.
````
package foo;

import org.quartz.JobExecutionContext;
import org.quartz.JobExecutionException;
import org.quartz.JobListener;

public class MyJobListener implements JobListener {

    private String name;

    public MyJobListener(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void jobToBeExecuted(JobExecutionContext context) {
        // do something with the event
    }

    public void jobWasExecuted(JobExecutionContext context,
            JobExecutionException jobException) {
        // do something with the event
    }

    public void jobExecutionVetoed(JobExecutionContext context) {
        // do something with the event
    }
}
````
OR -

Extend JobListenerSupport.
````
package foo;

import org.quartz.JobExecutionContext;
import org.quartz.JobExecutionException;
import org.quartz.listeners.JobListenerSupport;
````

````
public class MyOtherJobListener extends JobListenerSupport {

    private String name;

    public MyOtherJobListener(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
        @Override
    public void jobWasExecuted(JobExecutionContext context,
            JobExecutionException jobException) {
        // do something with the event
    }
}
````

Registering A JobListener With The Scheduler To Listen To All Jobs
````
scheduler.getListenerManager().addJobListener(myJobListener, allJobs());
````

Registering A JobListener With The Scheduler To Listen To A Specific Job
````
scheduler.getListenerManager().addJobListener(myJobListener,
jobKeyEquals(jobKey("myJobName", "myJobGroup")));
````

Registering A JobListener With The Scheduler To Listen To All Jobs In a Group
````
scheduler.getListenerManager().addJobListener(myJobListener,
jobGroupEquals("myJobGroup"));
````

### How-To: Using Trigger Listeners
#### Creating a TriggerListener

Implement the TriggerListener interface.
````
package foo;

import org.quartz.JobExecutionContext;
import org.quartz.Trigger;
import org.quartz.TriggerListener;
import org.quartz.Trigger.CompletedExecutionInstruction;

public class MyTriggerListener implements TriggerListener {

    private String name;

    public MyTriggerListener(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void triggerComplete(Trigger trigger, JobExecutionContext context,
            CompletedExecutionInstruction triggerInstructionCode) {
        // do something with the event

    }

    public void triggerFired(Trigger trigger, JobExecutionContext context) {
        // do something with the event
    }

    public void triggerMisfired(Trigger trigger) {
        // do something with the event
    }

    public boolean vetoJobExecution(Trigger trigger, JobExecutionContext context) {
        // do something with the event
        return false;
    }

}
````

OR -

Extend TriggerListenerSupport.

````
package foo;

import org.quartz.JobExecutionContext;
import org.quartz.Trigger;
import org.quartz.listeners.TriggerListenerSupport;

public class MyOtherTriggerListener extends TriggerListenerSupport {

    private String name;

    public MyOtherTriggerListener(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    @Override
    public void triggerFired(Trigger trigger, JobExecutionContext context) {
        // do something with the event
    }
}
````

Registering A TriggerListener With The Scheduler To Listen To All Triggers
````
scheduler.getListenerManager().addTriggerListener(myTriggerListener,
allTriggers());
````

Registering A TriggerListener With The Scheduler To Listen To A Specific Trigger
````
scheduler.getListenerManager().addTriggerListener(myTriggerListener,
triggerKeyEquals(triggerKey("myTriggerName", "myTriggerGroup")));
````

Registering A TriggerListener With The Scheduler To Listen To All Triggers In a Group
````
scheduler.getListenerManager().addTriggerListener(myTriggerListener,
triggerGroupEquals("myTriggerGroup"));
````

### How-To: Using Scheduler Listeners
#### Creating a SchedulerListener

Extend TriggerListenerSupport and override methods for events you're interested in.

````
package foo;

import org.quartz.Trigger;
import org.quartz.listeners.SchedulerListenerSupport;

public class MyOtherSchedulerListener extends SchedulerListenerSupport {

    @Override
    public void schedulerStarted() {
        // do something with the event
    }

    @Override
    public void schedulerShutdown() {
        // do something with the event
    }

    @Override
    public void jobScheduled(Trigger trigger) {
        // do something with the event
    }

}
````

Registering A SchedulerListener With The Scheduler
````
scheduler.getListenerManager().addSchedulerListener(mySchedListener);
````

### How-To: Trigger That Executes Every Ten Seconds
#### Using SimpleTrigger
````
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startNow()
    .withSchedule(simpleSchedule()
            .withIntervalInSeconds(10)
            .repeatForever())
    .build();
````

### How-To: Trigger That Executes Every 90 minutes

#### Using SimpleTrigger
````

trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startNow()
    .withSchedule(simpleSchedule()
            .withIntervalInMinutes(90)
            .repeatForever())
    .build();
````

#### Using CalendarIntervalTrigger
````

trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startNow()
    .withSchedule(calendarIntervalSchedule()
            .withIntervalInMinutes(90))
    .build();
````

### How-To: Trigger That Executes Every Day
If you want a trigger that always fires at a certain time of day, use CronTrigger or CalendarIntervalTrigger because they can preserve the fire time's time of day across daylight savings time changes.

#### Create a CronTrigger. that executes every day at 3:00PM:
````

   trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startNow()
    .withSchedule(dailyAtHourAndMinute(15, 0)) // fire every day at 15:00
    .build();
````

#### Using SimpleTrigger
Create a SimpleTrigger that executes 3:00PM tomorrow, and then every 24 hours (which may not always be at 3:00 PM - because adding 24 hours on days where daylight savings time shifts may result in 2:00 PM or 4:00 PM depending upon whether the 3:00 PM time was started during DST or standard time):

````

trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startAt(tomorrowAt(15, 0, 0) // first fire time 15:00:00 tomorrow
    .withSchedule(simpleSchedule()
            .withIntervalInHours(24) // interval is actually set at 24 hours' worth of milliseconds
            .repeatForever())
    .build();
````

#### Using CalendarIntervalTrigger
````
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startAt(tomorrowAt(15, 0, 0) // 15:00:00 tomorrow
    .withSchedule(calendarIntervalSchedule()
            .withIntervalInDays(1)) // interval is set in calendar days
    .build();
````

### How-To: Trigger That Executes Every 2 Days
At first glance, you may be tempted to use a CronTrigger. However, if this is truly to be every two days, CronTrigger won't work. To illustrate this, simply think of how many days are in a typical month (28-31). A cron expression like "0 0 5 2/2 * ?" would give us a trigger that would restart its count at the beginning of every month. This means that we would would get subsequent firings on July 30 and August 2, which is an interval of three days, not two.

Likewise, an expression like "0 0 5 1/2 * ?" would end up firing on July 31 and August 1, just one day apart.

Therefore, for this schedule, using SimpleTrigger or CalendarIntervalTrigger makes sense:
#### Using SimpleTrigger
Create a SimpleTrigger that executes 3:00PM tomorrow, and then every 48 hours (which may not always be at 3:00 PM - because adding 24 hours on days where daylight savings time shifts may result in 2:00 PM or 4:00 PM depending upon whether the 3:00 PM time was started during DST or standard time):
````
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startAt(tomorrowAt(15, 0, 0) // first fire time 15:00:00 tomorrow
    .withSchedule(simpleSchedule()
            .withIntervalInHours(2 * 24) // interval is actually set at 48 hours' worth of milliseconds
            .repeatForever())
    .build();
````

Using CalendarIntervalTrigger

````
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startAt(tomorrowAt(15, 0, 0)  // 15:00:00 tomorrow
    .withSchedule(calendarIntervalSchedule()
            .withIntervalInDays(2)) // interval is set in calendar days
    .build();
````

### How-To: Trigger That Executes Every Week
If you want a trigger that always fires at a certain time of day, use CronTrigger or CalendarIntervalTrigger because they can preserve the fire time's time of day across daylight savings time changes.
#### Using CronTrigger
Create a CronTrigger. that executes every Wednesday at 3:00PM:

````
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startNow()
    .withSchedule(weeklyOnDayAndHourAndMinute(DateBuilder.WEDNESDAY, 15, 0)) //
fire every wednesday at 15:00
    .build();
````

OR -

````
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startNow()
    .withSchedule(cronSchedule("0 0 15 ? * WED")) // fire every wednesday at 15:00
    .build();
````

Using SimpleTrigger
Create a SimpleTrigger that executes 3:00PM tomorrow, and then every 7 * 24 hours (which may not always be at 3:00 PM - because adding 24 hours on days where daylight savings time shifts may result in 2:00 PM or 4:00 PM depending upon whether the 3:00 PM time was started during DST or standard time):

````
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startAt(tomorrowAt(15, 0, 0)  // first fire time 15:00:00 tomorrow
    .withSchedule(simpleSchedule()
            .withIntervalInHours(7 * 24) // interval is actually set at 7 * 24
hours’ worth of milliseconds
            .repeatForever())
    .build();
````

Using CalendarIntervalTrigger

````
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startAt(tomorrowAt(15, 0, 0)  // 15:00:00 tomorrow
    .withSchedule(calendarIntervalSchedule()
            .withIntervalInWeeks(1)) // interval is set in calendar weeks
    .build();
````

### How-To: Trigger That Executes Every 2 Weeks
As with a trigger meant to fire every two days, CronTrigger won’t work for this schedule. For more details, see Trigger That Fires Every 2 Days. We’ll need to use a SimpleTrigger or CalendarIntervalTrigger:
#### Using SimpleTrigger
Create a SimpleTrigger that executes 3:00PM tomorrow, and then every 48 hours (which may not always be at 3:00 PM - because adding 24 hours on days where daylight savings time shifts may result in 2:00 PM or 4:00 PM depending upon whether the 3:00 PM time was started during DST or standard time):

````
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startAt(tomorrowAt(15, 0, 0)  // first fire time 15:00:00 tomorrow
    .withSchedule(simpleSchedule()
            .withIntervalInHours(14 * 24) // interval is actually set at 14 * 24
hours’ worth of milliseconds
            .repeatForever())
    .build();
````

#### Using CalendarIntervalTrigger

````
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startAt(tomorrowAt(15, 0, 0)  // 15:00:00 tomorrow
    .withSchedule(calendarIntervalSchedule()
````

~~~
            .withIntervalInWeeks(2)) // interval is set in calendar weeks
    .build();
````


### How-To: Trigger That Executes Every Month
####  Using CronTrigger
Create a CronTrigger. that executes every Wednesday at 3:00PM:
````

trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startNow()
    .withSchedule(monthlyOnDayAndHourAndMinute(5, 15, 0)) // fire on the 5th day of
every month at 15:00
    .build();
````

OR -
````

trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startNow()
    .withSchedule(cronSchedule("0 0 15 5 * ?")) // fire on the 5th day of every
month at 15:00
    .build();
````

OR -
````

trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startNow()
    .withSchedule(cronSchedule("0 0 15 L * ?")) // fire on the last day of every
month at 15:00
    .build();
````

OR -
````

trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startNow()
    .withSchedule(cronSchedule("0 0 15 LW * ?")) // fire on the last weekday day of
every month at 15:00
    .build();
````

OR -
````
~~~

```
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startNow()
    .withSchedule(cronSchedule("0 0 15 L-3 * ?")) // fire on the third to last day of every month at 15:00
    .build();
```

There are other possible combinations as well, which are more fully covered in the API documentation. All of these options were made by simply changing the day-of-month field. Imagine what you can do if you leverage the other fields as well!

#### Using CalendarIntervalTrigger

```
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startAt(tomorrowAt(15, 0, 0)  // 15:00:00 tomorrow
    .withSchedule(calendarIntervalSchedule()
            .withIntervalInMonths(1)) // interval is set in calendar months
    .build();
```

Quartz Configuration Reference

### Configure Main Scheduler Settings

These properties configure the identification of the scheduler, and various other ‘top level’ settings.

Property Name	Req'd	Type	Default Value
org.quartz.scheduler.instanceName	no	string	'QuartzScheduler'
org.quartz.scheduler.instanceId	no	string	'NON_CLUSTERED'
org.quartz.scheduler.instanceIdGenerator.class	no	string (class name)	org.quartz.simpl .SimpleInstanceIdGenerator
org.quartz.scheduler.threadName	no	string	instanceName + '_QuartzSchedulerThread'
org.quartz.scheduler .makeSchedulerThreadDaemon	no	boolean	false
org.quartz.scheduler .threadsInheritContextClassLoaderOfInitializer	no	boolean	false
org.quartz.scheduler.idleWaitTime	no	long	30000
org.quartz.scheduler.dbFailureRetryInterval	no	long	15000
org.quartz.scheduler.classLoadHelper.class	no	string (class name)	org.quartz.simpl

			.CascadingClassLoadHelper
org.quartz.scheduler.jobFactory.class	no	string (class name)	org.quartz.simpl.PropertySettingJobFactory
org.quartz.context.key.SOME_KEY	no	string	none
org.quartz.scheduler.userTransactionURL	no	string (url)	'java:comp/UserTransaction'
org.quartz.scheduler.wrapJobExecutionInUserTransaction	no	boolean	false
org.quartz.scheduler.skipUpdateCheck	no	boolean	false
org.quartz.scheduler.batchTriggerAcquisitionMaxCount	no	int	1
org.quartz.scheduler.batchTriggerAcquisitionFireAheadTimeWindow	no	long	0

org.quartz.scheduler.instanceName

Can be any string, and the value has no meaning to the scheduler itself - but rather serves as a mechanism for client code to distinguish schedulers when multiple instances are used within the same program. If you are using the clustering features, you must use the same name for every instance in the cluster that is 'logically' the same Scheduler.

org.quartz.scheduler.instanceId

Can be any string, but must be unique for all schedulers working as if they are the same 'logical' Scheduler within a cluster. You may use the value "AUTO" as the instanceId if you wish the Id to be generated for you. Or the value "SYS_PROP" if you want the value to come from the system property "org.quartz.scheduler.instanceId".

org.quartz.scheduler.instanceIdGenerator.class

Only used if org.quartz.scheduler.instanceId is set to "AUTO". Defaults to "org.quartz.simpl.SimpleInstanceIdGenerator", which generates an instance id based upon host name and time stamp. Other IntanceIdGenerator implementations include SystemPropertyInstanceIdGenerator (which gets the instance id from the system property "org.quartz.scheduler.instanceId", and HostnameInstanceIdGenerator which uses the local host name (InetAddress.getLocalHost().getHostName()). You can also implement the InstanceIdGenerator interface your self.

org.quartz.scheduler.threadName

Can be any String that is a valid name for a java thread. If this property is not specified, the thread will receive the scheduler's name (“org.quartz.scheduler.instanceName”) plus an the appended string ‘_QuartzSchedulerThread’.

org.quartz.scheduler.makeSchedulerThreadDaemon

A boolean value (‘true’ or ‘false’) that specifies whether the main thread of the scheduler should be a daemon thread or not. See also the org.quartz.scheduler.makeSchedulerThreadDaemon property for tuning the SimpleThreadPool if that is the thread pool implementation you are using (which is most likely the case).

org.quartz.scheduler.threadsInheritContextClassLoaderOfInitializer

A boolean value (‘true’ or ‘false’) that specifies whether the threads spawned by Quartz will inherit the context ClassLoader of the initializing thread (thread that initializes the Quartz instance). This will affect Quartz main scheduling thread, JDBCJobStore’s misfire handling thread (if JDBCJobStore is used), cluster recovery thread (if clustering is used), and threads in SimpleThreadPool (if SimpleThreadPool is used). Setting this value to ‘true’ may help with class loading, JNDI look-ups, and other issues related to using Quartz within an application server.

org.quartz.scheduler.idleWaitTime

Is the amount of time in milliseconds that the scheduler will wait before re-queries for available triggers when the scheduler is otherwise idle. Normally you should not have to ‘tune’ this parameter, unless you’re using XA transactions, and are having problems with delayed firings of triggers that should fire immediately. Values less than 5000 ms are not recommended as it will cause excessive database querying. Values less than 1000 are not legal.

org.quartz.scheduler.dbFailureRetryInterval

Is the amount of time in milliseconds that the scheduler will wait between re-tries when it has detected a loss of connectivity within the JobStore (e.g. to the database). This parameter is obviously not very meaningful when using RamJobStore.

org.quartz.scheduler.classLoadHelper.class

Defaults to the most robust approach, which is to use the “org.quartz.simpl.CascadingClassLoadHelper” class - which in turn uses every other ClassLoadHelper class until one works. You should probably not find the need to specify any other class for this property, though strange things seem to happen within application servers. All of the current possible ClassLoadHelper implementation can be found in the org.quartz.simpl package.

org.quartz.scheduler.jobFactory.class

The class name of the JobFactory to use. A JobFatcory is responsible for producing instances of JobClasses. The default is 'org.quartz.simpl.PropertySettingJobFactory', which simply calls newInstance() on the class to produce a new instance each time execution is about to occur. PropertySettingJobFactory also reflectively sets the job's bean properties using the contents of the SchedulerContext and Job and Trigger JobDataMaps.

org.quartz.context.key.SOME_KEY

Represent a name-value pair that will be placed into the "scheduler context" as strings. (see Scheduler.getContext()). So for example, the setting "org.quartz.context.key.MyKey = MyValue" would perform the equivalent of scheduler.getContext().put("MyKey", "MyValue").

The Transaction-Related properties should be left out of the config file unless you are using JTA transactions.

org.quartz.scheduler.userTransactionURL

Should be set to the JNDI URL at which Quartz can locate the Application Server's UserTransaction manager. The default value (if not specified) is "java:comp/UserTransaction" - which works for almost all Application Servers. Websphere users may need to set this property to "jta/usertransaction". This is only used if Quartz is configured to use JobStoreCMT, and org.quartz.scheduler.wrapJobExecutionInUserTransaction is set to true.

org.quartz.scheduler.wrapJobExecutionInUserTransaction

Should be set to "true" if you want Quartz to start a UserTransaction before calling execute on your job. The Tx will commit after the job's execute method completes, and after the JobDataMap is updated (if it is a StatefulJob). The default value is "false". You may also be interested in using the @ExecuteInJTATransaction annotation on your job class, which lets you control for an individual job whether Quartz should start a JTA transaction - whereas this property causes it to occur for all jobs.

org.quartz.scheduler.skipUpdateCheck

Whether or not to skip running a quick web request to determine if there is an updated version of Quartz available for download. If the check runs, and an update is found, it will be reported as available in Quartz's logs. You can also disable the update check with the system property "org.terracotta.quartz.skipUpdateCheck=true" (which you can set in your system environment or as a -D on the java command line). It is recommended that you disable the update check for production deployments.

org.quartz.scheduler.batchTriggerAcquisitionMaxCount

The maximum number of triggers that a scheduler node is allowed to acquire (for firing) at once. Default value is 1. The larger the number, the more efficient firing is (in situations where there are very many triggers needing to be fired all at once) - but at the cost of possible imbalanced load between cluster nodes. If the value of this property is set to > 1, and JDBC JobStore is used, then the property "org.quartz.jobStore.acquireTriggersWithinLock" must be set to "true" to avoid data corruption.

org.quartz.scheduler.batchTriggerAcquisitionFireAheadTimeWindow

The amount of time in milliseconds that a trigger is allowed to be acquired and fired ahead of its scheduled fire time.
Defaults to 0. The larger the number, the more likely batch acquisition of triggers to fire will be able to select and fire more than 1 trigger at a time - at the cost of trigger schedule not being honored precisely (triggers may fire this amount early). This may be useful (for performance's sake) in situations where the scheduler has very large numbers of triggers that need to be fired at or near the same time.

### Configure ThreadPool Settings

Property Name	Required	Type	Default Value
org.quartz.threadPool.class	yes	string (class name)	null
org.quartz.threadPool.threadCount	yes	int	-1
org.quartz.threadPool.threadPriority	no	int	Thread.NORM_PRIORITY (5)

org.quartz.threadPool.class

Is the name of the ThreadPool implementation you wish to use. The threadpool that ships with Quartz is "org.quartz.simpl.SimpleThreadPool", and should meet the needs of nearly every user. It has very simple behavior and is very well tested. It provides a fixed-size pool of threads that 'live' the lifetime of the Scheduler.

org.quartz.threadPool.threadCount

Can be any positive integer, although you should realize that only numbers between 1 and 100 are very practical. This is the number of threads that are available for concurrent execution of jobs. If you only have a few jobs that fire a few times a day, then 1 thread is plenty! If you have tens of thousands of jobs, with many firing every minute, then you probably want a thread count more like 50 or 100 (this highly depends on the nature of the work that your jobs perform, and your systems resources!).

org.quartz.threadPool.threadPriority

Can be any int between Thread.MIN_PRIORITY (which is 1) and Thread.MAX_PRIORITY (which is 10). The default is Thread.NORM_PRIORITY (5).

#### SimpleThreadPool-Specific Properties

Property Name	Required	Type	Default Value
org.quartz.threadPool.makeThreadsDaemons	no	boolean	false
org.quartz.threadPool.threadsInheritGroupOfInitializingThread	no	boolean	true
org.quartz.threadPool.threadsInheritContextClassLoaderOfInitializingThread	no	boolean	false
org.quartz.threadPool.threadNamePrefix	no	string	[Scheduler Name]_Worker

org.quartz.threadPool.makeThreadsDaemons

Can be set to “true” to have the threads in the pool created as daemon threads. Default is “false”. See also the org.quartz.scheduler.makeSchedulerThreadDaemon property.

org.quartz.threadPool.threadsInheritGroupOfInitializingThread

Can be “true” or “false”, and defaults to true.

org.quartz.threadPool.threadsInheritContextClassLoaderOfInitializingThread

Can be “true” or “false”, and defaults to false.

org.quartz.threadPool.threadNamePrefix

The prefix for thread names in the worker pool - will be postpended with a number.

#### Custom ThreadPools

If you use your own implementation of a thread pool, you can have properties set on it reflectively simply by naming the property as thus:

Setting Properties on a Custom ThreadPool

````
````

````
org.quartz.threadPool.class = com.mycompany.goo.FooThreadPool
org.quartz.threadPool.somePropOfFooThreadPool = someValue
````

### Configure Global Listeners
Global listeners can be instantiated and configured by StdSchedulerFactory, or your application can do it itself at runtime, and then register the listeners with the scheduler. “Global” listeners listen to the events of every job/trigger rather than just the jobs/triggers that directly reference them.

Configuring listeners through the configuration file consists of giving then a name, and then specifying the class name, and any other properties to be set on the instance. The class must have a no-arg constructor, and the properties are set reflectively. Only primitive data type values (including Strings) are supported.

Thus, the general pattern for defining a “global” TriggerListener is:

Configuring a Global TriggerListener
````

org.quartz.triggerListener.NAME.class = com.foo.MyListenerClass
org.quartz.triggerListener.NAME.propName = propValue
org.quartz.triggerListener.NAME.prop2Name = prop2Value
````

And the general pattern for defining a “global” JobListener is:

Configuring a Global JobListener
````

org.quartz.jobListener.NAME.class = com.foo.MyListenerClass
org.quartz.jobListener.NAME.propName = propValue
org.quartz.jobListener.NAME.prop2Name = prop2Value
````

### Configure Scheduler Plugins
Like listeners configuring plugins through the configuration file consists of giving then a name, and then specifying the class name, and any other properties to be set on the instance. The class must have a no-arg constructor, and the properties are set reflectively. Only primitive data type values (including Strings) are supported.

Thus, the general pattern for defining a plug-in is:

Configuring a Plugin
````
````

````
org.quartz.plugin.NAME.class = com.foo.MyPluginClass
org.quartz.plugin.NAME.propName = propValue
org.quartz.plugin.NAME.prop2Name = prop2Value
````

There are several Plugins that come with Quartz, that can be found in the org.quartz.plugins package (and subpackages). Example of configuring a few of them are as follows:
Sample configuration of Logging Trigger History Plugin

The logging trigger history plugin catches trigger events (it is also a trigger listener) and logs then with Jakarta Commons-Logging. See the class's JavaDoc for a list of all the possible parameters.

#### Sample configuration of Logging Trigger History Plugin
````
org.quartz.plugin.triggHistory.class = \
  org.quartz.plugins.history.LoggingTriggerHistoryPlugin
org.quartz.plugin.triggHistory.triggerFiredMessage = \
  Trigger \{1\}.\{0\} fired job \{6\}.\{5\} at: \{4, date, HH:mm:ss MM/dd/yyyy}
org.quartz.plugin.triggHistory.triggerCompleteMessage = \
  Trigger \{1\}.\{0\} completed firing job \{6\}.\{5\} at \{4, date, HH:mm:ss
MM/dd/yyyy\}.
````

#### Sample configuration of XML Scheduling Data Processor Plugin

Job initialization plugin reads a set of jobs and triggers from an XML file, and adds them to the scheduler during initialization. It can also delete exiting data. See the class's JavaDoc for more details.

Sample configuration of JobInitializationPlugin
````
org.quartz.plugin.jobInitializer.class = \
  org.quartz.plugins.xml.XMLSchedulingDataProcessorPlugin
org.quartz.plugin.jobInitializer.fileNames = \
  data/my_job_data.xml
org.quartz.plugin.jobInitializer.failOnFileNotFound = true
````

The XML schema definition for the file can be found here:

http://www.quartz-scheduler.org/xml/job_scheduling_data_1_8.xsd

#### Sample configuration of Shutdown Hook Plugin

The shutdown-hook plugin catches the event of the JVM terminating, and calls shutdown on the scheduler.

Sample configuration of ShutdownHookPlugin
````
org.quartz.plugin.shutdownhook.class = \
  org.quartz.plugins.management.ShutdownHookPlugin
org.quartz.plugin.shutdownhook.cleanShutdown = true
````

### Configure RMI Settings
None of the primary properties are required, and all have ‘reasonable’ defaults. When using Quartz via RMI, you need to start an instance of Quartz with it configured to “export” its services via RMI. You then create clients to the server by configuring a Quartz scheduler to “proxy” its work to the server.

Some users experience problems with class availability (namely Job classes) between the client and server. To work through these problems you'll need an understanding of RMI's ″codebase″ and RMI security managers. You may find these resources to be useful:

An excellent description of RMI and codebase: http://www.kedwards.com/jini/codebase.html . One of the important points is to realize that “codebase” is used by the client!

Quick info about security managers:
http://gethelp.devx.com/techtips/java_pro/10MinuteSolutions/10min0500.asp

And finally from the Java API docs, read the docs for the RMISecurityManager.

Property Name	Required	Default Value
org.quartz.scheduler.rmi.export	no	false
org.quartz.scheduler.rmi.registryHost	no	'localhost'
org.quartz.scheduler.rmi.registryPort	no	1099
org.quartz.scheduler.rmi.createRegistry	no	'never'
org.quartz.scheduler.rmi.serverPort	no	random
org.quartz.scheduler.rmi.proxy	no	false

org.quartz.scheduler.rmi.export

If you want the Quartz Scheduler to export itself via RMI as a server then set the ‘rmi.export’ flag to true.

org.quartz.scheduler.rmi.registryHost

The host at which the RMI Registry can be found (often ‘localhost’).

org.quartz.scheduler.rmi.registryPort

The port on which the RMI Registry is listening (usually 1099).

org.quartz.scheduler.rmi.createRegistry

Set the ‘rmi.createRegistry’ flag according to how you want Quartz to cause the creation of an RMI Registry. Use “false” or “never” if you don’t want Quartz to create a registry (e.g. if you already have an external registry running). Use “true” or “as_needed” if you want Quartz to first attempt to use an existing registry, and then fall back to creating one. Use “always” if you want Quartz to attempt creating a Registry, and then fall back to using an existing one. If a registry is created, it will be bound to port number in the given ‘org.quartz.scheduler.rmi.registryPort’ property, and ‘org.quartz.rmi.registryHost’ should be “localhost”.

org.quartz.scheduler.rmi.serverPort

The port on which the the Quartz Scheduler service will bind and listen for connections. By default, the RMI service will ‘randomly’ select a port as the scheduler is bound to the RMI Registry.

org.quartz.scheduler.rmi.proxy

If you want to connect to (use) a remotely served scheduler, then set the ‘org.quartz.scheduler.rmi.proxy’ flag to true. You must also then specify a host and port for the RMI Registry process - which is typically ‘localhost’ port 1099.

It does not make sense to specify a 'true' value for both 'org.quartz.scheduler.rmi.export' and 'org.quartz.scheduler.rmi.proxy' in the same config file - if you do, the 'export' option will be ignored. A value of 'false' for both 'export' and 'proxy' properties is of course valid, if you're not using Quartz via RMI.

### Configure RAMJobStore
RAMJobStore is used to store scheduling information (job, triggers and calendars) within memory. RAMJobStore is fast and lightweight, but all scheduling information is lost when the process terminates.

RAMJobStore is selected by setting the ‘org.quartz.jobStore.class’ property as such:

Setting The Scheduler's JobStore to RAMJobStore
````
org.quartz.jobStore.class = org.quartz.simpl.RAMJobStore
````

RAMJobStore can be tuned with the following properties:

Property Name	Required	Type	Default Value
org.quartz.jobStore.misfireThreshold	no	int	60000

org.quartz.jobStore.misfireThreshold

The the number of milliseconds the scheduler will 'tolerate' a trigger to pass its next-fire-time by, before being considered "misfired". The default value (if you don't make an entry of this property in your configuration) is 60000 (60 seconds).

### Configure JDBC-JobStoreTX
JDBCJobStore is used to store scheduling information (job, triggers and calendars) within a relational database. There are actually two seperate JDBCJobStore classes that you can select between, depending on the transactional behaviour you need.

JobStoreTX manages all transactions itself by calling commit() (or rollback()) on the database connection after every action (such as the addition of a job). JDBCJobStore is appropriate if you are using Quartz in a stand-alone application, or within a servlet container if the application is not using JTA transactions.

The JobStoreTX is selected by setting the 'org.quartz.jobStore.class' property as such:

Setting The Scheduler's JobStore to JobStoreTX
````
org.quartz.jobStore.class = org.quartz.impl.jdbcjobstore.JobStoreTX
````

JobStoreTX can be tuned with the following properties:

Property Name	Required	Type	Default Value
org.quartz.jobStore.driverDelegateClass	yes	string	null
org.quartz.jobStore.dataSource	yes	string	null
org.quartz.jobStore.tablePrefix	no	string	"QRTZ_"
org.quartz.jobStore.useProperties	no	boolean	false
org.quartz.jobStore.misfireThreshold	no	int	60000
org.quartz.jobStore.isClustered	no	boolean	false
org.quartz.jobStore.clusterCheckinInterval	no	long	15000
org.quartz.jobStore.maxMisfiresToHandleAtATime	no	int	20

org.quartz.jobStore.dontSetAutoCommitFalse	no	boolean	false
org.quartz.jobStore.selectWithLockSQL	no	string	″SELECT * FROM {0}LOCKS WHERE SCHED_NAME = {1} AND LOCK_NAME = ? FOR UPDATE″
org.quartz.jobStore.txIsolationLevelSerializable	no	boolean	false
org.quartz.jobStore.acquireTriggersWithinLock	no	boolean	false (or true - see doc below)
org.quartz.jobStore.lockHandler.class	no	string	null
org.quartz.jobStore.driverDelegateInitString	no	string	null

org.quartz.jobStore.driverDelegateClass

Driver delegates understand the particular ‘dialects’ of varies database systems. Possible choices include:

org.quartz.impl.jdbcjobstore.StdJDBCDelegate (for fully JDBC-compliant drivers)

org.quartz.impl.jdbcjobstore.MSSQLDelegate (for Microsoft SQL Server, and Sybase)

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 (for Oracle drivers used within Weblogic)

org.quartz.impl.jdbcjobstore.oracle.weblogic.WebLogicOracleDelegate (for Oracle drivers used within Weblogic)

org.quartz.impl.jdbcjobstore.CloudscapeDelegate

org.quartz.impl.jdbcjobstore.DB2v6Delegate

org.quartz.impl.jdbcjobstore.DB2v7Delegate

org.quartz.impl.jdbcjobstore.DB2v8Delegate

org.quartz.impl.jdbcjobstore.HSQLDBDelegate

org.quartz.impl.jdbcjobstore.PointbaseDelegate

org.quartz.impl.jdbcjobstore.SybaseDelegate

Note that many databases are known to work with the StdJDBCDelegate, while others are known to work with delegates for other databases, for example Derby works well with the Cloudscape delegate (no surprise there).

org.quartz.jobStore.dataSource

The value of this property must be the name of one the DataSources defined in the configuration properties file. See the configuration docs for DataSources for more information.

org.quartz.jobStore.tablePrefix

JDBCJobStore’s “table prefix” property is a string equal to the prefix given to Quartz’s tables that were created in your database. You can have multiple sets of Quartz’s tables within the same database if they use different table prefixes.

org.quartz.jobStore.useProperties

The “use properties” flag instructs JDBCJobStore that all values in JobDataMaps will be Strings, and therefore can be stored as name-value pairs, rather than storing more complex objects in their serialized form in the BLOB column. This is can be handy, as you avoid the class versioning issues that can arise from serializing your non-String classes into a BLOB.

org.quartz.jobStore.misfireThreshold

The the number of milliseconds the scheduler will ‘tolerate’ a trigger to pass its next-fire-time by, before being considered “misfired”. The default value (if you don’t make an entry of this property in your configuration) is 60000 (60 seconds).

org.quartz.jobStore.isClustered

Set to “true” in order to turn on clustering features. This property must be set to “true” if you are having multiple instances of Quartz use the same set of database tables… otherwise you will experience havoc. See the configuration docs for clustering for more information.

org.quartz.jobStore.clusterCheckinInterval

Set the frequency (in milliseconds) at which this instance “checks-in”* with the other instances of the cluster. Affects the quickness of detecting failed instances.

org.quartz.jobStore.maxMisfiresToHandleAtATime

The maximum number of misfired triggers the jobstore will handle in a given pass. Handling many (more than a couple dozen) at once can cause the database tables to be locked long enough that the performance of firing other (not yet misfired) triggers may be hampered.

org.quartz.jobStore.dontSetAutoCommitFalse

Setting this parameter to “true” tells Quartz not to call setAutoCommit(false) on connections obtained from the DataSource(s). This can be helpful in a few situations, such as if you have a driver that complains if it is called when it is already off. This property defaults to false, because most drivers require that setAutoCommit(false) is called.

org.quartz.jobStore.selectWithLockSQL

Must be a SQL string that selects a row in the “LOCKS” table and places a lock on the row. If not set, the default is “SELECT * FROM {0}LOCKS WHERE SCHED_NAME = {1} AND LOCK_NAME = ? FOR UPDATE”, which works for most databases. The “{0}” is replaced during run-time with the TABLE_PREFIX that you configured above. The “{1}” is replaced with the scheduler’s name.

org.quartz.jobStore.txIsolationLevelSerializable

A value of “true” tells Quartz (when using JobStoreTX or CMT) to call setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE) on JDBC connections. This can be helpful to prevent lock timeouts with some databases under high load, and “long-lasting” transactions.

org.quartz.jobStore.acquireTriggersWithinLock

Whether or not the acquisition of next triggers to fire should occur within an explicit database lock. This was once necessary (in previous versions of Quartz) to avoid dead-locks with particular databases, but is no longer considered necessary, hence the default value is “false”.

If “org.quartz.scheduler.batchTriggerAcquisitionMaxCount” is set to > 1, and JDBC JobStore is used, then this property must be set to “true” to avoid data corruption (as of Quartz 2.1.1 “true” is now the default if batchTriggerAcquisitionMaxCount is set > 1).

org.quartz.jobStore.lockHandler.class

The class name to be used to produce an instance of a org.quartz.impl.jdbcjobstore.Semaphore to be used for locking control on the job store data. This is an advanced configuration feature, which should not be used by most users. By default, Quartz will select the most appropriate (pre-bundled) Semaphore implementation to use.
“org.quartz.impl.jdbcjobstore.UpdateLockRowSemaphore” QUARTZ-497 may be of interest to MS SQL Server users. See QUARTZ-441.

org.quartz.jobStore.driverDelegateInitString

A pipe-delimited list of properties (and their values) that can be passed to the DriverDelegate during initialization time.

The format of the string is as such:
````
"settingName=settingValue|otherSettingName=otherSettingValue|..."
````

The StdJDBCDelegate and all of its descendants (all delegates that ship with Quartz) support a property called 'triggerPersistenceDelegateClasses' which can be set to a comma-separated list of classes that implement the TriggerPersistenceDelegate interface for storing custom trigger types. See the Java classes SimplePropertiesTriggerPersistenceDelegateSupport and SimplePropertiesTriggerPersistenceDelegateSupport for examples of writing a persistence delegate for a custom trigger.

### Configure JDBC-JobStoreCMT
JDBCJobStore is used to store scheduling information (job, triggers and calendars) within a relational database. There are actually two seperate JDBCJobStore classes that you can select between, depending on the transactional behaviour you need.

JobStoreCMT relies upon transactions being managed by the application which is using Quartz. A JTA transaction must be in progress before attempt to schedule (or unschedule) jobs/triggers. This allows the "work" of scheduling to be part of the applications "larger" transaction. JobStoreCMT actually requires the use of two datasources - one that has it's connection's transactions managed by the application server (via JTA) and one datasource that has connections that do not participate in global (JTA) transactions. JobStoreCMT is appropriate when applications are using JTA transactions (such as via EJB Session Beans) to perform their work.

The JobStore is selected by setting the 'org.quartz.jobStore.class' property as such:

Setting The Scheduler's JobStore to JobStoreCMT

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

JobStoreCMT can be tuned with the following properties:

Property Name	Required	Type	Default Value
org.quartz.jobStore.driverDelegateClass	yes	string	null
org.quartz.jobStore.dataSource	yes	string	null
org.quartz.jobStore.nonManagedTXDataSource	yes	string	null
org.quartz.jobStore.tablePrefix	no	string	"QRTZ_"
org.quartz.jobStore.useProperties	no	boolean	false
org.quartz.jobStore.misfireThreshold	no	int	60000
org.quartz.jobStore.isClustered	no	boolean	false
org.quartz.jobStore.clusterCheckinInterval	no	long	15000

org.quartz.jobStore.maxMisfiresToHandleAtATime	no	int	20
org.quartz.jobStore.dontSetAutoCommitFalse	no	boolean	false
org.quartz.jobStore.dontSetNonManagedTXConnectionAutoCommitFalse	no	boolean	false
org.quartz.jobStore.selectWithLockSQL	no	string	"SELECT * FROM {0}LOCKS WHERE SCHED_NAME = {1} AND LOCK_NAME = ? FOR UPDATE"
org.quartz.jobStore.txIsolationLevelSerializable	no	boolean	false
org.quartz.jobStore.txIsolationLevelReadCommitted	no	boolean	false
org.quartz.jobStore.acquireTriggersWithinLock	no	boolean	false (or true - see doc below)
org.quartz.jobStore.lockHandler.class	no	string	null
org.quartz.jobStore.driverDelegateInitString	no	string	null

org.quartz.jobStore.driverDelegateClass

Driver delegates understand the particular 'dialects' of varies database systems. Possible choices include:

org.quartz.impl.jdbcjobstore.StdJDBCDelegate (for fully JDBC-compliant drivers)
org.quartz.impl.jdbcjobstore.MSSQLDelegate (for Microsoft SQL Server, and Sybase)
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 (for Oracle drivers used within Weblogic)
org.quartz.impl.jdbcjobstore.oracle.weblogic.WebLogicOracleDelegate (for Oracle drivers used within Weblogic)
org.quartz.impl.jdbcjobstore.CloudscapeDelegate
org.quartz.impl.jdbcjobstore.DB2v6Delegate
org.quartz.impl.jdbcjobstore.DB2v7Delegate
org.quartz.impl.jdbcjobstore.DB2v8Delegate
org.quartz.impl.jdbcjobstore.HSQLDBDelegate
org.quartz.impl.jdbcjobstore.PointbaseDelegate
org.quartz.impl.jdbcjobstore.SybaseDelegate

Note that many databases are known to work with the StdJDBCDelegate, while others are known to work with delegates for other databases, for example Derby works well with the Cloudscape delegate (no surprise there).

org.quartz.jobStore.dataSource

The value of this property must be the name of one the DataSources defined in the configuration properties file. For JobStoreCMT, it is required that this DataSource contains connections that are capable of participating in JTA (e.g. container-managed) transactions. This typically means that the DataSource will be configured and maintained within and by the application server, and Quartz will obtain a handle to it via JNDI. See the configuration docs for DataSources for more information.

org.quartz.jobStore.nonManagedTXDataSource

JobStoreCMT requires a (second) datasource that contains connections that will not be part of container-managed transactions. The value of this property must be the name of one the DataSources defined in the configuration properties file. This datasource must contain non-CMT connections, or in other words, connections for which it is legal for Quartz to directly call commit() and rollback() on.

org.quartz.jobStore.tablePrefix

JDBCJobStore's "table prefix" property is a string equal to the prefix given to Quartz's tables that were created in your database. You can have multiple sets of Quartz's tables within the same database if they use different table prefixes.

org.quartz.jobStore.useProperties

The "use properties" flag instructs JDBCJobStore that all values in JobDataMaps will be Strings, and therefore can be stored as name-value pairs, rather than storing more complex objects in their serialized form in the BLOB column. This is can be handy, as you avoid the class versioning issues that can arise from serializing your non-String classes into a BLOB.

org.quartz.jobStore.misfireThreshold

The the number of milliseconds the scheduler will 'tolerate' a trigger to pass its next-fire-time by, before being considered "misfired". The default value (if you don't make an entry of this property in your configuration) is 60000 (60 seconds).

org.quartz.jobStore.isClustered

Set to "true" in order to turn on clustering features. This property must be set to "true" if you are having multiple instances of Quartz use the same set of database tables… otherwise you will experience havoc. See the configuration docs for clustering for more information.

org.quartz.jobStore.clusterCheckinInterval

Set the frequency (in milliseconds) at which this instance "checks-in"* with the other instances of the cluster. Affects the quickness of detecting failed instances.

org.quartz.jobStore.maxMisfiresToHandleAtATime

The maximum number of misfired triggers the jobstore will handle in a given pass. Handling many (more than a couple dozen) at once can cause the database tables to be locked long enough that the performance of firing other (not yet misfired) triggers may be hampered.

org.quartz.jobStore.dontSetAutoCommitFalse

Setting this parameter to "true" tells Quartz not to call setAutoCommit(false) on connections obtained from the DataSource(s). This can be helpful in a few situations, such as if you have a driver that complains if it is called when it is already off. This property defaults to false, because most drivers require that setAutoCommit(false) is called.

org.quartz.jobStore.dontSetNonManagedTXConnectionAutoCommitFalse

The same as the property org.quartz.jobStore.dontSetAutoCommitFalse, except that it applies to the nonManagedTXDataSource.

org.quartz.jobStore.selectWithLockSQL

Must be a SQL string that selects a row in the "LOCKS" table and places a lock on the row. If not set, the default is "SELECT * FROM {0}LOCKS WHERE SCHED_NAME = {1} AND LOCK_NAME = ? FOR UPDATE", which works for most databases. The "{0}" is replaced during run-time with the TABLE_PREFIX that you configured above. The "{1}" is replaced with the scheduler's name.

org.quartz.jobStore.txIsolationLevelSerializable

A value of "true" tells Quartz to call setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE) on JDBC connections. This can be helpful to prevent lock timeouts with some databases under high load, and "long-lasting" transactions.

org.quartz.jobStore.txIsolationLevelReadCommitted

When set to "true", this property tells Quartz to call setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED) on the non-managed JDBC connections. This can be helpful to prevent lock timeouts with some databases (such as DB2) under high load, and "long-lasting" transactions.

org.quartz.jobStore.acquireTriggersWithinLock

Whether or not the acquisition of next triggers to fire should occur within an explicit database lock. This was once necessary (in previous versions of Quartz) to avoid dead-locks with particular databases, but is no longer considered necessary, hence the default value is “false”.

If “org.quartz.scheduler.batchTriggerAcquisitionMaxCount” is set to > 1, and JDBC JobStore is used, then this property must be set to “true” to avoid data corruption (as of Quartz 2.1.1 “true” is now the default if batchTriggerAcquisitionMaxCount is set > 1).

org.quartz.jobStore.lockHandler.class

The class name to be used to produce an instance of a org.quartz.impl.jdbcjobstore.Semaphore to be used for locking control on the job store data. This is an advanced configuration feature, which should not be used by most users. By default, Quartz will select the most appropriate (pre-bundled) Semaphore implementation to use.
“org.quartz.impl.jdbcjobstore.UpdateLockRowSemaphore” QUARTZ-497 may be of interest to MS SQL Server users. “JTANonClusteredSemaphore” which is bundled with Quartz may give improved performance when using JobStoreCMT, though it is an experimental implementation. See QUARTZ-441 and QUARTZ-442

org.quartz.jobStore.driverDelegateInitString

A pipe-delimited list of properties (and their values) that can be passed to the DriverDelegate during initialization time.

The format of the string is as such:
````
″settingName=settingValue|otherSettingName=otherSettingValue|...″
````
The StdJDBCDelegate and all of its descendants (all delegates that ship with Quartz) support a property called ‘triggerPersistenceDelegateClasses’ which can be set to a comma-separated list of classes that implement the TriggerPersistenceDelegate interface for storing custom trigger types. See the Java classes SimplePropertiesTriggerPersistenceDelegateSupport and SimplePropertiesTriggerPersistenceDelegateSupport for examples of writing a persistence delegate for a custom trigger.

### Configure DataSources
If you’re using JDBC-Jobstore, you’ll be needing a DataSource for its use (or two

DataSources, if you're using JobStoreCMT).

DataSources can be configured in three ways:

All pool properties specified in the quartz.properties file, so that Quartz can create the DataSource itself.
The JNDI location of an application server managed Datasource can be specified, so that Quartz can use it.
Custom defined org.quartz.utils.ConnectionProvider implementations.

It is recommended that your Datasource max connection size be configured to be at least the number of worker threads in the thread pool plus three. You may need additional connections if your application is also making frequent calls to the scheduler API. If you are using JobStoreCMT, the "non managed" datasource should have a max connection size of at least four.

Each DataSource you define (typically one or two) must be given a name, and the properties you define for each must contain that name, as shown below. The DataSource's "NAME" can be anything you want, and has no meaning other than being able to identify it when it is assigned to the JDBCJobStore.

#### Quartz-created DataSources are defined with the following properties:

Property Name	Required	Type	Default Value
org.quartz.dataSource.NAME.driver	yes	String	null
org.quartz.dataSource.NAME.URL	yes	String	null
org.quartz.dataSource.NAME.user	no	String	″″
org.quartz.dataSource.NAME.password	no	String	″″
org.quartz.dataSource.NAME.maxConnections	no	int	10
org.quartz.dataSource.NAME.validationQuery	no	String	null
org.quartz.dataSource.NAME.idleConnectionValidationSeconds	no	int	50
org.quartz.dataSource.NAME.validateOnCheckout	no	boolean	false
org.quartz.dataSource.NAME.discardIdleConnectionsSeconds	no	int	0 (disabled)

org.quartz.dataSource.NAME.driver

Must be the java class name of the JDBC driver for your database.

org.quartz.dataSource.NAME.URL

The connection URL (host, port, etc.) for connection to your database.

org.quartz.dataSource.NAME.user

The user name to use when connecting to your database.

org.quartz.dataSource.NAME.password

The password to use when connecting to your database.

org.quartz.dataSource.NAME.maxConnections

The maximum number of connections that the DataSource can create in it's pool of connections.

org.quartz.dataSource.NAME.validationQuery

Is an optional SQL query string that the DataSource can use to detect and replace failed/corrupt connections.
For example an oracle user might choose "select table_name from user_tables" - which is a query that should never fail - unless the connection is actually bad.

org.quartz.dataSource.NAME.idleConnectionValidationSeconds

The number of seconds between tests of idle connections - only enabled if the validation query property is set.
Default is 50 seconds.

org.quartz.dataSource.NAME.validateOnCheckout

Whether the database sql query to validate connections should be executed every time a connection is retrieved from the pool to ensure that it is still valid. If false, then validation will occur on check-in. Default is false.

org.quartz.dataSource.NAME.discardIdleConnectionsSeconds

Discard connections after they have been idle this many seconds. 0 disables the feature. Default is 0.

Example of a Quartz-defined DataSource
````
org.quartz.dataSource.myDS.driver = oracle.jdbc.driver.OracleDriver
org.quartz.dataSource.myDS.URL = jdbc:oracle:thin:@10.0.1.23:1521:demodb
org.quartz.dataSource.myDS.user = myUser
org.quartz.dataSource.myDS.password = myPassword
org.quartz.dataSource.myDS.maxConnections = 30
````

#### References to Application Server DataSources are defined with the following properties:

Property Name	Required	Type	Default Value
org.quartz.dataSource.NAME.jndiURL	yes	String	null
org.quartz.dataSource.NAME.java.naming.factory.initial	no	String	null
org.quartz.dataSource.NAME.java.naming.provider.url	no	String	null
org.quartz.dataSource.NAME.java.naming.security.principal	no	String	null
org.quartz.dataSource.NAME.java.naming.security.credentials	no	String	null

org.quartz.dataSource.NAME.jndiURL

The JNDI URL for a DataSource that is managed by your application server.

org.quartz.dataSource.NAME.java.naming.factory.initial

The (optional) class name of the JNDI InitialContextFactory that you wish to use.

org.quartz.dataSource.NAME.java.naming.provider.url

The (optional) URL for connecting to the JNDI context.

org.quartz.dataSource.NAME.java.naming.security.principal

The (optional) user principal for connecting to the JNDI context.

org.quartz.dataSource.NAME.java.naming.security.credentials

The (optional) user credentials for connecting to the JNDI context.

Example of a Datasource referenced from an Application Server
````
org.quartz.dataSource.myOtherDS.jndiURL=jdbc/myDataSource
org.quartz.dataSource.myOtherDS.java.naming.factory.initial=com.evermind.server.rmi.RMIInitialContextFactory
org.quartz.dataSource.myOtherDS.java.naming.provider.url=ormi://localhost
org.quartz.dataSource.myOtherDS.java.naming.security.principal=admin
org.quartz.dataSource.myOtherDS.java.naming.security.credentials=123
````
Custom ConnectionProvider Implementations

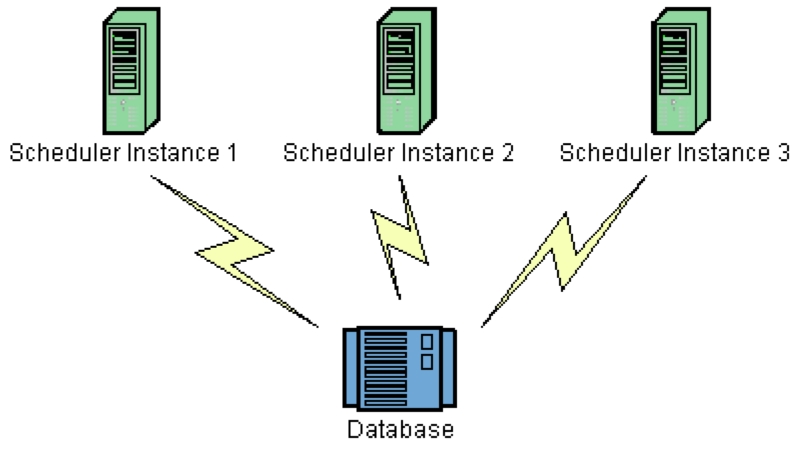
Property Name	Required	Type	Default Value
org.quartz.dataSource.NAME.connectionProvider.class	yes	String (class name)	null

org.quartz.dataSource.NAME.connectionProvider.class

The class name of the ConnectionProvider to use. After instantiating the class, Quartz can automatically set configuration properties on the instance, bean-style.

Example of Using a Custom ConnectionProvider Implementation
````
org.quartz.dataSource.myCustomDS.connectionProvider.class =
com.foo.FooConnectionProvider
org.quartz.dataSource.myCustomDS.someStringProperty = someValue
org.quartz.dataSource.myCustomDS.someIntProperty = 5
````

### Configure Clustering with JDBC-JobStore
Quartz's clustering features bring both high availability and scalability to your scheduler via fail-over and load balancing functionality.



Clustering currently only works with the JDBC-Jobstore (JobStoreTX or JobStoreCMT), and essentially works by having each node of the cluster share the same database.

Load-balancing occurs automatically, with each node of the cluster firing jobs as quickly as it can. When a trigger's firing time occurs, the first node to acquire it (by placing a lock on it) is the node that will fire it.

Only one node will fire the job for each firing. What I mean by that is, if the job has a repeating trigger that tells it to fire every 10 seconds, then at 12:00:00 exactly one node will run the job, and at 12:00:10 exactly one node will run the job, etc. It won't necessarily be the same node each time - it will more or less be random which node runs it. The load balancing mechanism is near-random for busy schedulers (lots of triggers) but favors the same node for non-busy (e.g. few triggers) schedulers.

Fail-over occurs when one of the nodes fails while in the midst of executing one or

more jobs. When a node fails, the other nodes detect the condition and identify the jobs in the database that were in progress within the failed node.
Any jobs marked for recovery (with the “requests recovery” property on the JobDetail) will be re-executed by the remaining nodes. Jobs not marked for recovery will simply be freed up for execution at the next time a related trigger fires.

The clustering feature works best for scaling out long-running and/or cpu-intensive jobs (distributing the work-load over multiple nodes). If you need to scale out to support thousands of short-running (e.g 1 second) jobs, consider partitioning the set of jobs by using multiple distinct schedulers (including multiple clustered schedulers for HA). The scheduler makes use of a cluster-wide lock, a pattern that degrades performance as you add more nodes (when going beyond about three nodes - depending upon your database’s capabilities, etc.).

Enable clustering by setting the “org.quartz.jobStore.isClustered” property to “true”. Each instance in the cluster should use the same copy of the quartz.properties file. Exceptions of this would be to use properties files that are identical, with the following allowable exceptions: Different thread pool size, and different value for the “org.quartz.scheduler.instanceId” property. Each node in the cluster MUST have a unique instanceId, which is easily done (without needing different properties files) by placing “AUTO” as the value of this property. See the info about the configuration properties of JDBC-JobStore for more information.

Never run clustering on separate machines, unless their clocks are synchronized using some form of time-sync service (daemon) that runs very regularly (the clocks must be within a second of each other). See http://www.boulder.nist.gov/timefreq/service/its.htm if you are unfamiliar with how to do this.

Never start (scheduler.start()) a non-clustered instance against the same set of database tables that any other instance is running (start()ed) against. You may get serious data corruption, and will definitely experience erratic behavior.

Example Properties For A Clustered Scheduler
````
#============================================================================
# Configure Main Scheduler Properties
#============================================================================

org.quartz.scheduler.instanceName = MyClusteredScheduler
org.quartz.scheduler.instanceId = AUTO

#============================================================================
# Configure ThreadPool
````

```
#============================================================================

org.quartz.threadPool.class = org.quartz.simpl.SimpleThreadPool
org.quartz.threadPool.threadCount = 25
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.oracle.OracleDelegate
org.quartz.jobStore.useProperties = false
org.quartz.jobStore.dataSource = myDS
org.quartz.jobStore.tablePrefix = QRTZ_

org.quartz.jobStore.isClustered = true
org.quartz.jobStore.clusterCheckinInterval = 20000

#============================================================================
# Configure Datasources
#============================================================================

org.quartz.dataSource.myDS.driver = oracle.jdbc.driver.OracleDriver
org.quartz.dataSource.myDS.URL = jdbc:oracle:thin:@polarbear:1521:dev
org.quartz.dataSource.myDS.user = quartz
org.quartz.dataSource.myDS.password = quartz
org.quartz.dataSource.myDS.maxConnections = 5
org.quartz.dataSource.myDS.validationQuery=select 0 from dual
```

### Configure TerracottaJobStore
TerracottaJobStore is used to store scheduling information (job, triggers and calendars) within a Terracotta server.
TerracottaJobStore is much more performant than utilizing a database for storing scheduling data (via JDBC-JobStore), and yet offers clustering features such as load-balancing and fail-over.

You may want to consider implications of how you setup your Terracotta server, particularly configuration options that turn on features such as storing data on disk, utilization of fsync, and running an array of Terracotta servers for HA.

The clustering feature works best for scaling out long-running and/or cpu-intensive jobs (distributing the work-load over multiple nodes). If you need to scale out to support thousands of short-running (e.g 1 second) jobs, consider partitioning the set of jobs by using multiple distinct schedulers. Using more than one scheduler currently forces the use of a cluster-wide lock, a pattern that degrades performance as you add more clients.

More information about this JobStore and Terracotta can be found at http://www.terracotta.org/quartz ›

TerracottaJobStore is selected by setting the 'org.quartz.jobStore.class' property as such:

Setting The Scheduler's JobStore to TerracottaJobStore

````
org.quartz.jobStore.class = org.terracotta.quartz.TerracottaJobStore
````

TerracottaJobStore can be tuned with the following properties:

Property Name	Required	Type	Default Value
org.quartz.jobStore.tcConfigUrl	yes	string	
org.quartz.jobStore.misfireThreshold	no	int	60000

org.quartz.jobStore.tcConfigUrl

The host and port identifying the location of the Terracotta server to connect to, such as "localhost:9510".

org.quartz.jobStore.misfireThreshold

The the number of milliseconds the scheduler will 'tolerate' a trigger to pass its next-fire-time by, before being considered "misfired". The default value (if you don't make an entry of this property in your configuration) is 60000 (60 seconds).