Quartz 调度器开发指南

V 2.2.1

目录

使用 Quartz API

实例化 Scheduler

在你使用 Scheduler 之前，首先需要实例化。你必须使用 SchedulerFactory 来完成。

很多 Quartz 的使用者在 JNDI 中存储一个工厂的实例，这样非常容易直接使用一个工厂实例。

一旦一个 scheduler 被实例化，它就能被启动、设置为备用模式、停止。请注意，一旦你关闭了 scheduler，不重新实例化就无法重启它。Scheduler 不启动 trigger 也不会启动（因此，job 也不会执行）。Scheduler 处于暂停状态它们都不会工作。

下面的代码段实例化并启动一个 scheduler，安排一个 job 执行。

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

关键接口

Quartz API 的关键接口：

●··Scheduler——Scheduler 进行交互的主要 API。
●··Job——你想让 Scheduler 执行时需要实现的接口组件。
●··JobDetail——用于定义 Job 的实例。
●··Trigger——一个定义在将要执行 Job 的 scheduler 上的组件。

●··JobBuilder——已定义的 Job 实例，用于定义/构建的 JobDetail 实例。
●··TriggerBuilder——用于定义/构建 Trigger 实例。

一个 scheduler 的生命周期是有限的，从 SchedulerFactory 开始直到调用它的 shutdown()方法。

Scheduler 接口一旦创建，就能添加、移除、和列出 job 和 trigger，并执行另一些与 scheduling 相关的操作(例如暂停 trigger)，然而，在 scheduler 被 start()方法启动前，它不会作用于任何 trigger（执行 job），见第 3 页“实例化 Scheduler”。

Quartz 提供”builder”类定义领域特定语言(或 DSL，有时也被称为“连贯接口”)。这里有一个例子：

```
// define the job and tie it to our HelloJob class
JobDetail job = new Job(HelloJob.class)
          .withIdentity("myJob", "group1") // name "myJob", group "group1"
          .build();
// Trigger the job to run now, and then every 40 seconds
Trigger trigger = new Trigger()
          .withIdentity("myTrigger", "group1")
          .startNow()
          .withSchedule(simpleSchedule().withIntervalInSeconds(40).repeatForever())
          .build();
// Tell quartz to schedule the job using our trigger
sched.scheduleJob(job, trigger);
```

使用 JobBuilder 类的静态方法创建 job 的代码块。同样，构建 trigger 的代码块使用 TriggerBuilder 类的静态方法创建，SimpleScheduleBulder 类也一样。

DSL 的静态导入可以接受这样的导入语句：

```
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.*;
```

不同的 SchedulerBuilder 类拥有不同定义 schedules 类型的方法。

DateBuilder 类包含有不同类型的方法去简单的构造 java.util.Date 实例的特定的时间点（例如如果当前时间为 9:43:27，下一小时执行的时间，或者换句话说 10:00）。

Job 和 Trigger

一个 job 就是实现了 Job 接口的类。如下所示，这个接口有一个简单的方法：

```
package org.quartz;
publicinterface Job {
        publicvoid execute(JobExecutionContext context) throws JobExecutionException;
}
```

当一个 job 的触发器触发时，execute(..)方法被 Scheduler 的一个工作线程调用。JobExecutionContext 对象就是通过这个方法提供 job 实例的信息，如运行环境，包括

scheduler 执行的句柄，trigger 触发执行的句柄，job 的 JobDetail 对象，和一些其它信息。

JobDetail 对象由 Quartz 客户端（你的程序）在 job 添加进 scheduler 的时候创建。这个对象包含了很多 job 的属性设置，和 JobDataMap 一样，它能够存储你的 job 实例的状态信息。JobDetail 对象本质上是定义的 job 实例。

Trigger 对象常常用于触发 job 的执行（或者触发），当你希望安排一个 job，你可以实例化一个 trigger，并“调整”其属性来提供你想要的调度。

Trigger 可能有一个与它们关联的 JobDataMap 对象。JobDataMap 用于传递特定于触发器触发的工作参数。Quartz 附带一些不同的触发类型，但是最常用的就是 SimpleTrigger 和 CronTrigger。

●‥SimpleTrigger 很方便，如果你需要“一次性”执行(只是在一个给定时刻执行 job),或者如果你需要一个 job 在一个给定的时间,并让它重复 N 次,并在执行之间延迟 T。

●‥CronTrigger 是有用的，如果你想拥有引发基于当前日历时间表，如“每个星期五,中午”或“在每个月的第十天 10:15。”

为什么会同时有 job 和 trigger？许多工作调度器没有单独的 job 和 trigger 的概念。一些定义一个 job 只是一个执行时间（或计划）以及一些小的工作标识符。其它的大部分就像 Quartz 的 job 和 triiger 对象的结合体。Quartz 旨在创建一个分离的 scheduler 和 scheduler 的执行。这个设计有很多好处。

例如，你可以创建独立于 trigger 的 job 并将他们存储在 job scheduler。这使你能在同一个 job 上关联多个 trigger。另一个解耦的好处是，能够在 scheduler 关联的 trigger 已过期时仍能配置 job。这使你在重新安排它们后，不用重新定义它们。它还允许您修改或替换一个 trigger，而不必重新定义相关的 job。

特性

当 Job 和 trigger 注册进 Quartz scheduler 时给出了 key。Job 和 trigger（JobKey 和 TriggerKey）的 key 允许它们放被放置到 group 中，group 可用于分类组织你得 job 和 trigger，例如“报表 job”和“维护 job”。Job 或者 trigger 的 key 值在 group 中必须是唯一的。换句话说，job 或 trigger 的完整的 key(或标识)是由 key 和 group 组成。

有关 job 和 trigger 的详细信息,请参见第 6 页“Job 和 JobDetails”和第 10 页“使用 trigger”。

使用 Job 和 JobDetail

Job 和 JobDetail

Job 很容易实现，接口中只有一个“execute”方法。Job 有更多的东西你需要了解，就是关于 Job 接口的 execute(..)方法，和 JobDetails。

当你实现一个 job 类时，代码知道如何实现 job 的特定类型的实际工作，你或许希望了解 job 实例拥有的 Quartz 的各种属性。

使用 JobBuilder 类构建 JobDetail 实例。你通常会想要使用静态导入导入所有的方法,为了你的代码能够 DSL-feel。

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

下面的代码片段定义了一个 job 并安排它的执行：

```
        // define the job and tie it to our HelloJob class
        JobDetail job = new Job(HelloJob.class)
                  .withIdentity("myJob", "group1") // name "myJob", group "group1"
                  .build();
        // Trigger the job to run now, and then every 40 seconds
        Trigger trigger = new Trigger()
                  .withIdentity("myTrigger", "group1")
                  .startNow()
                  .withSchedule(simpleSchedule().withIntervalInSeconds(40).repeatForever())
                  .build();
        // Tell quartz to schedule the job using our trigger
        sched.scheduleJob(job, trigger);
```

现在考虑这个 job 类 HelloJob，如下所示:

```
publicclass HelloJob implements Job {
     public HelloJob() {
     }

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

请注意我们给 scheduler 一个 JobDetail 实例，而且它知道要执行的 job 的类型，只需提供我们构建 JobDetail 的 job 类。每次 scheduler 执行 job,它会在调用它的 execute(..)方法之前创建这个类的新实例。当执行完成，job 实例的引用被删除,然后垃圾收集器回收实例。

这种行为的后果之一是 job 必须有一个无参数的构造函数(当使用默认 JobFactory 实现)。另一个是 job 类上定义的没有意义的数据字段的状态,它们的值在 Job 执行期间不会被保存。

你能够使用 JobDataMap 为 job 实例提供属性/配置或跟踪 job 执行间的状态,,它是 JobDetail 对象的一部分。

JobDataMap

JobDataMap 可以用来保存大量那些在 job 实例执行时你希望得到的 (序列化)数据对象。JobDataMap 是 Java Map 接口的一个实现,并且添加一些便利的方法来存储和检索数据的原始类型。

当定义/构建 JobDetail，这里有一些将 job 添加进 scheduler 之前放到 JobDataMap 的数据。

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

这是一个例子，在 job 执行期间，从 JobDataMap 获取数据:

```
publicclass DumbJob implements Job {
```

```
    public DumbJob() {
    }

    publicvoid execute(JobExecutionContext context) throws JobExecutionException {
        JobKey key = context.getJobDetail().getKey();
        JobDataMap dataMap = context.getJobDetail().getJobDataMap();
        String jobSays = dataMap.getString("jobSays");
        floatmyFloatValue = dataMap.getFloat("myFloatValue");
        System.err.println("Instance " + key + " of DumbJob says: " + jobSays + ", and val
is: " + myFloatValue);
    }
}
```

如果你使用一个持久化的 JobStore(在本教程的 JobStore 节讨论)你应该注意在决定你在 JobDataMap 中安排什么,因为里面的对象将序列化,他们因此会有类版本问题。显然标准 Java 类型应该很安全,但除此之外,任何时候有人改变一个类的定义的序列化实例,必须小心不要打破兼容性。可选地,您可以将 JDBC-JobStore 和 JobDataMap 放入 model 中,只允许基本类型和字符串存储在 map 中,从而消除任何后来序列化问题的可能性。

如果你为 job 类添加 setter 方法,相当于在 JobDataMap 中对应的 key (如上面的示例中 setJobSays(String val)方法，然后 quartz 的默认 JobFactory 实现将自动调用这些 setter 当 Job 被实例化,从而防止需要显式地在你的执行方法获得 map 的值。

trigger 也可以有与他们有关的 JobDataMaps。如果你有一个 job 存储在 scheduler 由多个 trigger 常规/重复使用,然而每个独立的触发,你想工作提供不同的数据输入，这是有用的。

job 执行期间在 JobExecutionContext 上建立 JobDataMap 十分方便。它是基于 JobDetail 的 JobDataMap 和基于 Trigger 上的一个合并，后者任何相同名字的值将覆盖前者。

这是一个简单的例子在 job 执行期间从 JobExecutionContext 合并的 JobDataMap 获取数据:

```
publicclass DumbJob implements Job {
    public DumbJob() {
    }

    publicvoid 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");
        floatmyFloatValue = 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);
    }
    }
```

或者如果你想依靠 JobFactory 注入数据映射到类的值,它可能看起来像这样:

```
publicclass DumbJob implements Job {
    String jobSays;
    floatmyFloatValue;
    ArrayList state;

    public DumbJob() {
    }

    publicvoid 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);
    }

    publicvoid setJobSays(String jobSays) {
        this.jobSays = jobSays;
    }

    publicvoid setMyFloatValue(floatmyFloatValue) {
        myFloatValue = myFloatValue;
    }

    publicvoid setState(ArrayList state) {
        state = state;
    }
    }
```

你会注意到整个代码的类是长,但是 execute()方法中的代码更干净。当然你也可以说,虽然代码比较长,它实际上花了更少的编码,如果程序员用 IDE 自动生成 setter 方法,而不必手工从 JobDataMap 检索值。这是你的选择。

Job 实例

你可以创建一个单一的工作类,在 scheduler 中存储被 JobDetails 创建的多个实例定义——每个都有自己的一组属性和 JobDataMap——并将它们添加到调度器。

例如,您可以创建一个类称为 SalesReportJob,实现了 job 接口。这个 job 也许被编码成预计的参数(通过 JobDataMap)发送给指定的基于销售报告的销售人员。然后,他们可能会创建多个 Job 定义(JobDetails),比如 SalesReportForJoe 和 SalesReportForMike 中指定相应的 JobDataMaps 作为“Joe”和“Mike”各自的输入工作。

当 trigger 触发,JobDetail(实例定义)被关联加载,Job 类通过 scheduler 中的配置 JobFactory 实例化。默认 JobFactory 简单地调用 Job 类的 newInstance(),然后试图调用类中与 JobDataMap 键的名称相匹配 setter 方法。您可能想要创建自己的 JobFactory 实现，诸如应用程序的 IoC 或 DI 容器生产/实例化 job 类。

每个存储的 JobDetail 作为 job 定义或 JobDetail 实例被引用,每个执行的 job 都是一个 job 实例或 job 定义的实例。一般来说,使用术语“工作”时,指的是定义的名称或 JobDetail。类实现的工作接口,称为“job 类”。

Job 状态和并发

下面是 job 状态数据和并发的一些注意点。你可以添加一些注解在你的 job 类上，在一些方面影响 Quartz 的行为。

@DisallowConcurrentExecution 注释,可以添加到 job 类上，通知 Quartz 不要同时执行给定的 job 定义（引用给定的 job 类）的多个实例。在上一个例子中，如果 SalesReportJob 用了这个注解，在给定时间将只能执行 SalesReportJob 的一个实例，但是它可以同时执行“SalesReportForMike”的一个实例。约束是基于实例的定义（JobDetail），而不是 job 类的实例。然而，它决定了注解参与到类本身中（在 Quartz 设计时），因为它对类如何编码产生影响。

@PersistJobDataAfterExecution 注释可以添加到 job 类，通知 Quartz 在 execute()方法成功执行后更新 JobDetail 的 JobDataMap 的拷贝，这样相同的 job 下一次执行时接收到更新的值，而不是原来存储的值。和@DisallowConcurrentExecution 注解一样，应用到 job 定义的实例上，而不是 job 类的实例，虽然它给 job 类带来了属性，因为它对类如何编码产生影响。（例如，“有状态性”需要明确理解执行方法的代码）。

如果你使用@PersistJobDataAfterExecution 注解,您还应该考虑使用 @DisallowConcurrentExecution 注解,以避免可能的混乱(竞争条件)的数据被存储在相同的工作的两个实例(JobDetail)并发执行。

Job 的其他属性

这里有你可以通过 JobDetail 对象定义 job 实例的另一些属性的快速总结：

持久性——如果一个 job 不是持久的, 一旦不再有任何活动的与之关联的 trigger 它将自动从 scheduler 删除。换句话说,不持久的 job 的生命周期依赖于它的 trigger 的存在。

RequestsRecovery——如果一份工作“请求恢复”,在 scheduler 强制关闭期间执行(即进程是运行崩溃,或机器关闭)，在 scheduler 再次启动时会重新执行。在这种情况下,JobExecutionContext.isRecovering()方法将返回 true。

JobExecutionException

最后,我们需要告诉你 Job.execute(..)一些细节。唯一类型的异常(包括 RuntimException),你可以从执行方法抛出 JobExecutionException。因此,你通常应该使用“rycatch”块包裹执行方法。你也应该花些时间看着 JobExecutionException 的文档,因为你的 job 可以使用它提供 scheduler 作为你想如何处理异常的各种指令。

使用 trigger

Trigger

和 job 一样,trigger 容易处理,但是他们有不同的定制项,在你充分使用 Quartz 之前需要了解它们。

不同类型的 trigger 用来满足不同的调度需求。常用的有两种，SimpleTrigger 和 CronTrigger。关于这些特定触发器的细节在第 10 页“使用 trigger”。

通用的 trigger 属性

除了所有的触发器都有 TriggerKey 属性作为追踪它们的标识符外，它们还有很多通用的其它属性。当你创建 trigger 定义时你可以使用 TriggerBuilder 设置这些通用的属性（如下示例）。

这是通用的触发器类型：

●‥属性 jobKey 是 trigger 触发将要执行的 job 的标识。

●‥startTime 是 trigger 的计划第一次生效的标识。值是一个 java.util.Date 对象，定义了一个给定的日期时间。对于许多类型的 trigger 来说，trigger 应该在开始时间准确的启动。对于其它类型来说标志着 scheduler 开始被追踪。这意味着你可以这样存储一个 scheduler 的 trigger，例如一月“每月的第 5 天执行”，如果 startTime 属性设置为 4 月 1 日，在首次触发前，还要过几个月。

●‥endTime 属性表示 trigger 的计划不再有效。换句话说，一个“每月 5 号”的 trigger，endTime 为 7 月 1 号，它上次执行的时间就是 6 月 5 号。

其它属性，还有更多的解析，将在下面讨论。

优先级

有时，当你有很多触发器（或者你得 Quartz 线程池有很多工作线程），Quartz 也许没有足够的资源在同一时间触发所有的 trigger。这种情况下，你也许想控制你的哪些触发器可以优先获得 Quartz 的工作线程。为此，你需要设置 trigger 的优先级属性。如果 N 个触发器同一时间触发，但是当前仅有 Z 和可用的工作线程，前 Z 个具有高优先级的触发器将首先执行。

任何整数，正数或者负数都允许作为优先级。如果你的 trigger 没有设定优先级它将使用默认的优先级为 5。

注意：优先级仅仅在同一时间触发的 trigger 中才有效。一个在 10:59 触发的 trigger 将永远比在 11:00 触发的 trigger 先执行。

注意：当触发器被监测到需要恢复时，它的优先级将与原始的优先级相同。

触发失败说明

Trigger 另一个重要的属性就是“过时触发指令”。如果因为当前 scheduler 被关闭或者 Quartz 的线程池没有可用的线程用来执行 job 导致当前的触发器错过了触发时间，就是出现错误。

不同类型的 trigger 有不同的失败处理方式。默认情况下，它们使用“智能策略”，其拥有基于 trigger 类型和配置的动态行为。Scheduler 启动时，它将搜索所有失败的持久化的 trigger，然后基于它们的失败配置来更新它们。

当你在你的项目中使用 Quartz，一定要熟悉它们 javaDoc 中给定 trigger 的失败处理策略。更多关于特殊 trigger 类型的失败策略在第 10 页 “使用 trigger” 也有提供。

日历

Quartz 日历对象（不是 java.util.Calendar 对象）能够与在 scheduler 中定义和存储的 trigger 时间相关联。

日历可用于从 trigger 的触发计划中排除时间块。例如，你可以创建一个 trigger 在工作日的 9:30 触发，但是排除法定假日。

日历可以使任何实现了 Calendar 接口的序列化对象（如下所示）：

```
package org.quartz;

publicinterface Calendar {

    publicboolean isTimeIncluded(long timeStamp);

    publiclong getNextIncludedTime(long timeStamp);

}
```

注意这些方法的参数类型是 long。这是毫秒的时间戳格式。这意味着日历可以“阻挡”精确至一毫秒的时间。最有可能的是你会对“屏蔽”一整天感兴趣。Quartz 包括一个 org.quartz.impl.HolidayCalendar 就是干这个的，非常方便。

日历必须通过 addCalendar(..)方法实例化并注册到 scheduler 中。如果你使用 HolidayCalendar，实例化之后，你应该使用它的 addExcludedDate(Date date)方法将你想排除在调度器之外的日期添加进去。同一个日历实例可以被多个 trigger 使用，如下所示：

```
HolidayCalendar cal = new HolidayCalendar();
cal.addExcludedDate(someDate);
cal.addExcludedDate(someOtherDate);
sched.addCalendar("myHolidays", cal, false);
Trigger t = new Trigger()
          .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 = new Trigger()
          .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
```

上面的代码创建了两个触发器，每天都会执行。然而，任何发生在排除日历的触发将被跳过。

org.quartz.impl.calendar 包提供了许多 Calendar 的实现，也许可以满足你的需求。

SimpleTrigger

如果你需要一个 job 在特定的时间执行一次或者在特定的时间，以特定的间隔执行多次 SimpleTrigger 能够满足你的需求。例如：你想让一个 trigger 在 2015 年 1 月 13 号 11:23:54 分触发，或者你想在那个时间执行，执行 5 次，每 10 秒一次。

这样描述，你也许不难发现 SimpleTrigger 的属性：开始时间，结束时间，重复次数，重复间隔。所有这些属性正是你期望的，只有一组与结束时间关联的特殊属性。

重复的次数可以是 0，正数或者常数 SimpleTrigger.REPEAT_INDEFINITELY。重复的间隔属性必须是 0 或一个正的长整型，代表毫秒值。注意，间隔为 0 将导致触发器同时触发“重复次数”这么多次（尽可能接近 scheduler 的并发管理数）。

如果你还不熟悉 Quartz 的 DateBuilder 类，你可能发现依赖于你的开始时间（或结束时间）计算 trigger 的触发次数很有帮助。

endTIme 属性重写了重复次数的属性。如果你想创建一个每 10 秒触发一次的 trigger 直到一个给定的时刻，这也许会有用。而不必计算开始时间和结束时间间的重复次数，你可以简单的指出结束时间然后使用 REPEAT_INDEFINITELY 表明重复次数（你甚至可以指定较大的重复次数超过在结束时间到达之前 trigger 触发的次数）。SimpleTrigger 实例

通过使用 TriggerBuilder（创建 trigger 的主属性）和 SimpleSchedulerBuilder（创建 SimpleTrigger 特殊属性）创建。使用静态导入 DSL-style 使用这些构造工具。

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

使用不同的 Scheduler 定义 trigger 的例子

下面是使用简单的 scheduler 定义触发器的一些例子。仔细观察，它们每个至少展示了一个新的/不同的点。

同时，花费一些时间查看 TriggerBuilder 和 SimpleSchedulerBuilder 定义的所有可用方法以便熟悉这里的示例没展示但是你可能会使用到的选项。

注意：注意如果你不显示的设置属性 TriggerBuilder（和其它的 quartz 创建者）会选取一个合理的值。例如，如果你不调用 withIdentity(..)的某个方法，TriggerBuilder 将为你的 trigger 生成一个随机的名字，同样，如果你没有调用 startAt(..)，则设定为当前的时间（立即执行）。

创建一个指定时间执行的 trigger，没有重复次数

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

在指定的时间创建一个 trigger，然后每 10 秒触发一次，共触发 10 次

```
trigger = new Trigger()
          .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();
```

创建一个触发一次的 trigger，5 分钟后触发

```
trigger = (SimpleTrigger) new Trigger()
          .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();
```

创建一个立即触发的 trigger，每 5 分钟触发一次，直到 22:00 结束

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

建立一个 trigger，每小时开始的时候触发，每 2 小时出发一次，直到永远

```
trigger = new Trigger()
```

```
                .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);
```

SimpleTrigger 失败说明

SimpleTrigger 有几个指令，用来通知 quartz 当触发失败该怎么做。（关于 trigger 触发失败的信息，参见 trigger 章节）这些指令被 SimpleTrigger 自己定义（它们的行为在 Javadoc 中有描述）。这些常量包括：

```
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
```

所有的 trigger 都有 MISFIRE_INSTRUCTION_SMART_POLICY 属性可以使用，这个指定对所有的触发器类型都是可用的。

如果使用“智能策略”，SimpleTrigger 会基于给定 SimpleTrigger 实例的配置或语句之间动态的选择触发失败指令。SimpleTrigger.updateAfterMisfire()方法的 Javadoc 解释了这些动态行为的细节。

在创建 SimpleTrigger 时，你可以指定失败指令作为简单 scheduler 的一部分（通过 SimpleSchedulerBuilder）：

```
        trigger = new Trigger()
                .withIdentity("trigger7", "group1")
            .withSchedule(simpleSchedule().withIntervalInMinutes(5).repeatForever().withMisf
ireHandlingInstructionNextWithExistingCount())
                .build();
```

CronTriggers

Cron 是一个 UNIX 工具,已经存在了很长时间,所以它的调度功能强大且得到证实。CronTrigger 类是基于 cron 的调度功能。

CronTrigger 使用“cron 表达式”，其能够像这样创建触发的计划：“每周一至周五早上 8:00”或者“每月最后一个周五早上 1:30”。

如果你需要基于类似日历的观念反复触发一个 job，而不是使用 SimpleTrigger 指定一个固定的时间间隔，相对于 SimpleTrigger，CronTrigger 往往更有用。

使用 CronTrigger，你可以指定一个触发计划例如“每周五的中午”，或者“每个工作日上午 9:00”，甚至“1 月的周一，周三和周五早上 9:00 至 10:00 每五分钟”。

即便如此，和 SimpleTrigger 一样， CronTrigger 也有一个 startTime 指定计划的生效时间也有一个（可选的）endTime 指定计划的停止时间。

Cron 表达式

Cron 表达式用于 CronTrigger 实例的配置。Cron 表达式实际上是由 7 个子表达式组成，用来描述计划安排的细节。这些子表达式使用空格分割：

●··秒
●··分
●··时
●··每月的第几天
●··月
●··每周的第几天
●··年（可选字段）

一个完整的 cron 表达式是一个像这样的字符串“0 0 12 ? WED *”——这代表着“每周三的 12:00”。

每个子表达式都能包含范围/或列表，例如前面每周的第几天（“WED”）可以替换成“MON-FRI”，“MON,WED,FRI”，甚至“MON-WED,SAT”。

可使用通配符（“*”）代表这个字段上的每一个可能的值。因此“月”字段上使用“*”代表着“每个月”。“*”在“每周的第几天”代表“一周的每一天”。

所有的字段都可以指定一个有效值。这些值应该相当明显，例如数字 0-59 代表秒数和分钟数，0-23 代表小时。每月中的第几天可能是 1-31，但是你应该注意一个指定的月份应该有多少天！月份可以指定 0-11，或者使用字符串 JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV 和 DEC。每周的第几天可以使用 1-7（1=周日）或者使用字符串 SUN, MON, TUE, WED, THU, FRI 和 SAT。

“/”符号被用来指定增量值。例如，如果你在分钟字段设置“0/15”，意味着“从 0 分开始，每 15 分钟”。如果你在分钟字段使用“3/20”，意味着“从第 3 分钟开始，每 20 分钟”——换句话说它和在分钟字段上指定“3,23,43”是一样的。注意“/35”并不是“每 35 分钟”——它是指“从 0 开始，每 35 分钟——换句话说就和“0,35”一样。

“?”允许在每月中的第几天和每周的第几天上设置。它用于指定“无特殊值”。当你需要在这两个字段之一指定一些值，但不能同时设置。具体参见下面示例（和 CronTrigger Javadoc）。

“L”允许在每月的第几天和每周的第几天上设置。这个符号是“last”的简写，但是它在这两个字段上的意思是不同的。例如，“L”在每月的第几天字段上指的是“每月的最后一天”——1 月 31 号，和非闰年的 2 月 28 号。如果用在每周的第几天，就是“7”或者“SAT”的意思。但是如果用在每周第几天的一个值后面，就意味着“每月最后一个星期的第几天”——例如“6L”或者“FRIL”都表示“每月最后一个周五”。你还可以为每月的最后一天指定一个偏移量，例如“L-3”意味着每月倒数第三天。当使用“L”选项时，重要的是不要指定列表或范围值，不然你将得到混乱/不期望的结果。

“W”用于指定距离给定日期最近的一个工作日（周一至周五）。例如，如果你在每月的第几天字段上指定“15W”，意味着：“离本月 15 号最近的工作日”。

“#”用于指定每月中的“第几个”周几。例如，在每周的第几天设置“6#3”或者“FRI#3”表示“本月第三个周五”。

格式

字段如下：

字段名称	必填项	允许的值	允许的特殊字符
Seconds	YES	0-59	, - *
Minutes	YES	0-59	, - *
Hours	YES	0-23	, - *

Day of month	YES	1-31	, - * ? / L W
Month	YES	1-12 或 JAN-DEC	, - *
Day of week	YES	1-7 或 SUN-SAT	, - * ? / L #
Year	NO	空值,1970-2099	, - *

最简单的 cron 表达式可以写成：* * * * ? *

或者比较复杂的：0/5 14,18,3-39,52 * ? JAN,MAR,SEP MON-FRI 2002-2010

特殊字符

●‥*（所有值）——用来选择一个字段上的所有值。例如：分钟字段上的“*”表示“每分钟”。

●‥?（无特定值）——当你需要在允许的两个字段中的一个指定一些值的时候使用。例如，你想让你得 trigger 在每月的某一天触发（假如，10 号），但是不关系在周几发生，你应该在每月的第几天字段上设置“10”，在每周的第几天字段上设置“?”。具体参见下面实例。

●‥-——用于指定范围。例如，小时字段上设置“10-12”表示“10 点，11 点，12 点”。

●‥,——用于指定额外的值。例如，在每周的第几天设置“MON,WED,FRI”表示“周一，周三，周五”。

●‥/——用于指定增量。例如，秒字段上“0/15”表示“0 秒，15 秒，30 秒，45 秒”。秒字段上“5/15”表示“5 秒，20 秒，35 秒，50 秒”，你还可以在“character - in this case”后面指定“/”，相当于“/”前有一个“0”，每月中的第几天设置“1/3”表示“从每月的第一天开始每 3 天触发一次”。

●‥L（“last”简写）——可用在两个字段每个子段的含义不同。例如，“L”在每月的第几天字段上指的是“每月的最后一天”——1 月 31 号，和非闰年的 2 月 28 号。如果用在每周的第几天，就是“7”或者“SAT”的意思。但是如果用在每周第几天的一个值后面，就意味着“每月最后一个星期的第几天”——例如“6L”表示“每月最后一个周五”。你还可以为每月的最后一天指定一个偏移量，例如“L-3”意味着每月倒数第三天。当使用“L”选项时，重要的是不要指定列表或范围值，不然你将得到混乱/不期望的结果。

●‥W（工作日）——用来指定距离指定日期最近的工作日（周一至周五）。例如，如果你指定了“15W”在每月第几天的字段上，表示：“离本月 15 号最近的工作日”。所以，如果 15 号是周六，trigger 将在 14 号周五触发，如果 15 号是周日，trigger 将在 16 号周一触发，如果 15 号是周二，trigger 将在 15 号周二触发。然而如果在每月的第几天字段指定“1W”，而且 1 号是周六，trigger 将会在 3 号周一触发，它不会跳过月的边界。“W”只能指定每月第几天字段中的一天，不能指定一个范围或列表。

●‥“L”和“W”字符也可以在每月的第几天字段组合成“LW”，表示“每月的最后一个工作日”。

●‥#——用于指定每月中的第几个周几。例如，每周第几天字段上的“6#3”表示“每月的第 3 个周五”（”6“=周五，”#3“=每月的第 3 个）。另一个例子：”2#1“=每月的第 1 个周一，”4#5“每月的第 5 个周三。注意，如果你指定”#5”，但是该月没有第 5 个这一周的天数，触发器将不会执行。

注意：月份的英文名和周几的英文是不区分大小写的。MON 与 mon 都是合法的字符。

示例

这里有一些完整的示例：

表达式	含义
0 0 12 * * ?	每天 12:00 触发
0 15 10 ? * *	每天 10:15 触发

0 15 10 * * ?	每天 10:15 触发
0 15 10 * * ? *	每天 10:15 触发
0 15 10 * * ? 2005	2005 年的每天 10:15 触发
0 * 14 * * ?	每天 14:00 至 14:59 每分钟触发
0 0/5 14 * * ?	每天 14:00 开始至 14:55 每 5 分钟触发
0 0/5 14,18 * * ?	每天 14:00 开始至 14:55 每 5 分钟触发，18:00 开始至 18:55 每 5 分钟触发
0 0-5 14 * * ?	每天 14:00 开始至 14:05 每分钟触发
0 10,44 14 ? 3 WED	三月份的每个周三 14:10 和 14:44 触发
0 15 10 ? * MON-FRI	每个周一至周五上午 10:15 触发
0 15 10 15 * ?	每月 15 号 10:15 触发
0 15 10 L * ?	每月最后一天 10:15 触发
0 15 10 L-2 * ?	每月倒数第 2 天 10:15 触发
0 15 10 ? * 6L	每月最后一个周五 10:15 触发
0 15 10 ? * 6L 2002-2005	2002 年至 2005 年每月最后一个周五 10:15 触发
0 15 10 ? * 6#3	每月第 3 个周五 10:15 触发
0 0 12 1/5 * ?	每月的第一天开始，每个第 5 天 12:00 触发
0 11 11 11 11 ?	每年 11 月 11 号 11:11 触发

注意”?”和”*”在每周第几天和每月第几天字段上的影响。

注意

●‥支持不需要完全指定一周中的第几天和一月中的第几天（你必须在这两个字段之一使用”?”号）。

●‥当设置触发时间在凌晨时注意那些实行夏令时的地区（在美国，时间通常会提前或推后到凌晨 2 点——因为依赖于时间向前或向后改变会跳过或重复执行触发器。你可以从维基百科查看你的使用环境：https://secure.wikimedia.org/wikipedia/en/wiki/Daylight_saving_time_around_the_world）

创建 CronTrigger

使用 TriggerBuilder（创建 trigger 的主属性）和 CronScheduleBuilder（创建 CronTrigger 特殊属性）创建 CronTrigger 实例。用 DSL-style 使用这些构造类，使用静态导入：

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

创建一个每天 8:00 至 17:00，每分钟执行一次的触发器

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

创建一个每天 10:42 触发的触发器

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

或

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

创建一个触发器将在周三 10:42 触发，基于系统默认的时区

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

或

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

CronTrigger 失败说明

下面的说明可以在 CronTrigger 失败时用来介绍 Quartz 信息。（更多关于 trigger 失败的情况介绍在教程中）。这些介绍是 CronTrigger 自己定义的常量（包括 Javadoc 描述的行为）。包括：

MISFIRE_INSTRUCTION_IGNORE_MISFIRE_POLICY

MISFIRE_INSTRUCTION_DO_NOTHING

MISFIRE_INSTRUCTION_FIRE_NOW

所有的 trigger 都有 MISFIRE_INSTRUCTION_SMART_POLICY 说明可供使用，这个说明也是所有 trigger 类型默认的。”智能策略 CronTrigger 解读为“MISFIRE_INSTRUCTION_FIRE_NOW。CronTrigger.updateAfterMisfire()方法的 Javadoc 解析了这些行为的细节。

在构建 CronTrigger 时，你可以（通过 CronSchedulerBuilder）指定简单 schedule 的失败说明。

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

```
        .build();
```

使用 TriggerListener 和 JobListener

TriggerListener 和 JobListener

监听器是你基于 scheduler 中事件的活动创建的对象。就像它们名字所示，TriggerListener 接收与 trigger 有关的事件，JobListener 接收与 Job 有关的事件。

与 trigger 有关的事件包括 trigger 触发，trigger 触发失败（在第 10 页“使用 trigger”中讨论），trigger 执行完成（trigger 完成 job 执行完毕）。

org.quartz.TriggerListener 接口

```
publicinterface TriggerListener {
    public String getName();

    publicvoid triggerFired(Trigger trigger, JobExecutionContext context);

    publicboolean vetoJobExecution(Trigger trigger, JobExecutionContext context);

    publicvoid triggerMisfired(Trigger trigger);

    publicvoid triggerComplete(Trigger trigger, JobExecutionContext context, inttri
ggerInstructionCode);
}
```

与 job 有关的时间包括：job 即将进行的通知，和 job 已经完成的通知。

org.quartz.JobListener 接口

```
publicinterface JobListener {
    public String getName();

    publicvoid jobToBeExecuted(JobExecutionContext context);

    publicvoid jobExecutionVetoed(JobExecutionContext context);

    publicvoid jobWasExecuted(JobExecutionContext context,
JobExecutionException jobException);
}
```

创建你自己的监听器

要创建一个监听器，需要简单创建一个实现 org.quartz.TriggerListener 接口或/和 org.quartz.JobListener 接口的对象。

然后在运行时监听器被注册到 scheduler 中，并且必须给定一个 name（或者确切的说，它们必须能够通过它们的 getName()方法给出它们的名字）。

为了你的方便，而不需要实现这些接口，你的类还可以继承 JobListenerSupport 或 TriggerListenerSupport 并简单的重写你感兴趣的方法。

监听器被 scheduler 的 ListenerManager 注册与 Matcher 一起描述 Job/Trigger 的监听器想要接受的事件。

注意：监听器在运行时被注册到 scheduler，而不是一直随 job 和 trigger 存储在 JobStore 中。这是因为监听器通常作为你的应用程序的一个集成点。因此，你的应用每次运行，监听器都需要重新注册进 scheduler。

添加监听器的代码示例

下面的示例演示了将感兴趣的 JobListener 添加进不同类型的 job 中。以同样的方式添加可用的 TriggerListener。

添加一个感兴趣的 JobListener 到特定的 job

```
scheduler.getListenerManager().addJobListener(myJobListener,KeyMatcher.jobKeyEquals(newJobKey("myJobName", "myJobGroup")));
```

你可能想使用静态导入导入 matcher 和 key 类，这将使你清晰的匹配 matcher：

```
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：

```
scheduler.getListenerManager().addJobListener(myJobListener,jobKeyEquals(jobKey("myJobName", "myJobGroup")));
```

添加一个感兴趣的 JobListener 到一个指定 group 的所有 job

```
scheduler.getListenerManager().addJobListener(myJobListener,
jobGroupEquals("myJobGroup"));
```

添加一个感兴趣的 JobListener 到两个指定的 group 的所有 job

```
scheduler.getListenerManager().addJobListener(myJobListener,
or(jobGroupEquals("myJobGroup"), jobGroupEquals("yourGroup")));
```

添加一个感兴趣的 JobListener 到所有 job

```
scheduler.getListenerManager().addJobListener(myJobListener, allJobs());
```

使用 SchedulerListener

SchedulerListener

SchedulerListener 和 TriggerListener 及 JobListener 很像，除了其接收 Scheduler 自身事件的通知，而不必与特定的 trigger 和 job 相关联。

在大量的其他事件中，Scheduler 关联的事件包括：

- 添加一个 job 或 trigger
- 移除一个 trigger 或 job

●··Schduler 中的一系列错误
●··Schduler 的关闭

SchedulerListener 被 scheduler 的 ListenerManager 注册。SchdulerListner 几乎可以是任何实现 org.quartz.SchedulerListener 接口的对象。

org.quartz.SchedulerListener 接口

```
publicinterface SchedulerListener {
    publicvoid jobScheduled(Trigger trigger);

    publicvoid jobUnscheduled(String triggerName, String triggerGroup);

    publicvoid triggerFinalized(Trigger trigger);

    publicvoid triggersPaused(String triggerName, String triggerGroup);

    publicvoid triggersResumed(String triggerName, String triggerGroup);

    publicvoid jobsPaused(String jobName, String jobGroup);

    publicvoid jobsResumed(String jobName, String jobGroup);

    publicvoid schedulerError(String msg, SchedulerException cause);

    publicvoid schedulerStarted();

    publicvoid schedulerInStandbyMode();

    publicvoid schedulerShutdown();

    publicvoid schedulingDataCleared();
    }
```

添加一个 SchedulerListener

```
scheduler.getListenerManager().addSchedulerListener(mySchedListener);
```

移除一个 SchedulerListener

```
scheduler.getListenerManager().removeSchedulerListener(mySchedListener);
```

使用 JobStore

关于 JobStores

JobStore 负责保存你给定的 schduler 所有的数据：job，trigger，calendar 等等。

为你的 Quartz scheduler 实例在一些重要的步骤选择合适的 JobStore。一旦你搞清楚它们之间的不同是很容易的。你声明哪种 JobStore 根据你的 scheduler 应该使用属性文件中你提供给你用来生成你的 scheduler 实例的 SchedulerFactory。

你在属性文件(或对象)中声明你的 scheduler 应该使用哪种 JobStore (和其配置设置)，你将其提供给你用来生成 scheduler 实例的 SchedulerFactory。

重点：不要再代码中直接使用你的 JobStore 实例。JobStore 是 Quartz 自己后台用的。你必须让 Quartz（通过配置）知道使用哪种 JobStore。之后，你只需在你的代码中使用 Scheduler 实例。

RAMJobStore

RAMJobStore 是使用 JobStore 最简单的方式。它也提供了最好的性能（在 CPU 时间上）。

RAMJobStore 就如它的名字所示，将它的数据保存在 RAM 中。这也是为什么它快速以及容易配置的原因。

使用 RAMJobStore 的缺点是，当你的应用程序结束（或者崩溃）后，Scheduler 的所有信息都会丢失。因此，RAMJobStore 不能担任 job 和 trigger 的”持久化“设置。对于某些应用来说，这是可以接受的，甚至是所需要的行为，但是对于另一些应用，可能就是灾难性的。

使用 RAMJobStore（假如你使用 StdSchedulerFactory）简单的设置 org.quartz.jobStore.class 属性到 org.quartz.simpl.RAMJobStore 如下所示：

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

没有你需要关心的其他设置。

JDBCJobStore

JDBCJobStore 通过 JDBC 将所有的数据保存到数据库中，因为它使用数据库，它比 RAMJobStore 配置更复杂，而且不是那么快。然后，性能的缺点并不是很糟，特别是你建立了数据表上的主键之后。现代的机器配置在相当好的局域网中（在 Scheduler 和 database 之间）接收和更新一个 trigger 通常小于 10 毫秒。

JDBCJobStore 适用于几乎任何数据库,它已经广泛应用于 oracle、PostgreSQL、MySQL、MS SQL Serve、HSQLDB 和 DB2。

使用 JDBCJobStore，你必须首先创建一个 Quartz 使用的数据表。你可以在 Quartz 的发布包的“docs/dbTables”目录下找到创建表的 SQL 脚本。如果没有你的数据库的脚本，找一个已经存在的修改成你的数据库需要的形式。

注意，在脚本中所有的表都以前缀” QRTZ_ “开头（例如”QRTZ_TRIGGERS”, 和 “QRTZ_JOB_DETAIL”）。这个前缀可以是你想要的任何方式（在你的 Quartz 属性文件中）。创建不同的前缀可能用于在同一个数据库创建多个 scheduler 实例，创建多个表。

一旦你创建了表，你需要决定你的应用需要哪种类型的事务。如果你不需要把你的调度命令（如添加和删除 trigger）连接到其它的事务上，你可以使用 JobStoreTX 作为你的 JobStore 让 Quartz 管理这些事务（这是最常见的选择）。

如果你需要 Quartz 和其它事务一起工作（例如，使用 J2EE 应用服务），你应该使用 JobStoreCMT，这种情况下 Quartz 会让应用服务器管理事务。

最后，你必须设置一个数据源将 JDBCJobStore 连接到你的数据库。数据源可以在你的 Quartz 属性文件中使用多种方式之一设置。一种方法就是将数据库的所有连接信息提供

给 Quartz 由 Quartz 自己创建管理数据源。另一种就是 Quartz 使用 Quartz 运行时的应用服务器管理的数据源。你可以通过 JNDI 的名称提供数据源。关于属性更详细的信息，请参考“docs/config”目录下的配置文件。

使用 JDBCJobStore(假设你使用 StdSchedulerFactory)首先需要设置 Quartz 配置的 JobStore 类属性如下列之一：

- org.quartz.impl.jdbcjobstore.JobStoreTx
- org.quartz.impl.jdbcjobstore.JobStoreCMT

你的选择基于上面一段的解释。

下面的例子展示了如何配置 JobStoreTx:

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

接下来，你需要选择一个 DriverDelegate 供 JobStore 使用。驱动代理也许需要为你的特定的数据库负责做任何 JDBC 的工作。

StdJDBCDelegate 是一个使用” vanilla”JDBC 代码（和 SQL 语句）来完成工作的代理。如果没有一个特定于你的数据库的代理，就是用这个代理。Quartz 为那些使用 StdJDBCDelegate 不能够操作很好的数据库提供了一些特定的代理。那些代理可以在 org.quartz.impl.jdbcjobstore 包或其子包下找到。这些代理包括 DB2v6Delegate（为了 DB2 第 6 版或更早的版本），HSQLDBDelegate（为了 HSQLDB），MSSQLDelegate（为了微软 SQLServer），PostgreSQLDelegate（为了 PostgreSQL），WeblogicDelegate（为了使用 Weblogic 创建的 JDBC 驱动），OracleDelegate（为了使用 Oracle），等等。

一旦你选择了你的代理，设置它的类名作为代理供 JDBCJobStore 使用。

下面的实例展示了如何配置 JDBCJobStore 去使用 DriverDelegate:

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

接下来，你需要通知 JobStore 你使用的前缀（如上所示）。

下面的例子展示了如何配置 JDBCJobStore 的表前缀:

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

最后，你需要设置 JobStore 使用哪种数据源。已命名的数据源也要在你得 Quartz 属性中定义。这种情况下，我们指定 Quartz 应该使用的数据源名为“myDS”（即在配置属性中定义）。

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

提示：如果你的 scheduler 很繁忙，意味着其总是执行与线程池大小一样的 job 数，那么你应该将数据源的连接数设置为线程池的大小+2。

提示：org.quartz.jobStore.useProperties 的属性可以设置为 true（默认为 false），通知 JDBCJobStore 将 JobDataMaps 中的所有的值转成字符串，因此能够存成键值对的形式，而不是将更复杂的对象以序列化的形式存储在 BLOB 列中。长期来看这安全多了，因为你避免了从非字符串对象类序列化到 BLOB 的类版本问题。

TerracottaJobStore

TerracottaJobStore 为无需使用数据库进行扩展和鲁棒性提供了手段。这意味着你的数据库可以与 Quartz 保持自由的加载，反而可以将它的所有资源保存为你的应用的一部分。

TerracottaJobStore 能够在集群或者非集群上运行，这两种情况下为你的 job 数据持久化在应用重启时提供了存储介质，因为数据存储在 Terracotta server 上。它的性能比通

过 JDBCJobStore 使用一个数据库要慢（大概好一个数量级），但是比 RAMJobStore 要慢。

在你的 Quartz 配置中简单的设置 org.quartz.jobStore.class 属性为 org.terracotta.quartz.TerracottaJobStore 就可以使用 TerracottaJobStore，还需要添加一个额外的属性到 Terracotta server 上：

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

关于 TerracottaJobStore 的更多信息，参见 Terracotta Quartz 用户手册。

配置，Scheduler Factory 和日志

配置组件

Quartz 的体系结构是模块化的，因此想运行它需要把几个组建“结合”在一起。幸运的是，有很多已存在的助手完成这件事。

在 Quartz 能工作之前的主要需要配置的组件是：

- ThreadPool
- JobStore
- 数据源 (如果必须的话)
- Scheduler

ThreadPool 为 Quartz 执行 job 提供了一组线程。线程池中的线程越多，可以同时执行的 job 就越多。然而，太多的线程将会拖慢你的系统。大多数 Quartz 使用者发现大约 5 个线程就足够了。因为他们做任何时间给定的 job 数都少于 100，job 通常不会同时执行，而且 job 运行时间很短（完成很快）。

其他用户发现他们需要 15，50 或者 100 个线程,因为他们各种各样的 schduler 有成千上万的 trigger,最终在任何时刻平均有 10 到 100 个 job 在执行。

找到正确的 scheduler 池的大小完全依赖于你使用 scheduler 做什么。没有真正的规则可循，除了尽可能小的保持线程数（参考你的机器资源）。不过，你要确保你要确保你的 job 能够准时触发。注意，如果一个 trigger 时间到了，但是没有足够的可用线程，Quartz 将会冻结（暂停）直到有一个可用的线程，那么 job 将会比它应该执行的时间晚一些。如果在 scheduler 配置了“失败阀值”期间没有可用线程，这甚至会导致线程失败。

ThreadPool 接口的定义在 org.quartz.spi 包中，你可以使用你喜欢的任何方式创建 ThreadPool。Quartz 附带了一个简单的 ThreadPool 叫 org.quartz.simpl.SimpleThreadPool。这个 org.quartz.simpl.SimpleThreadPool.简单的维护一组固定的线程池，不会增加也不会减少。但是它十分健壮和有很好的测试。几乎每个使用 Quartz 的用户都是用它。

JobStores 和 DataSources 的讨论在第 23 页“使用 JobStores”。值得注意的是，所有的 JobStores 都实现了 org.quartz.spi.JobStore 接口，如果内置的 JobStores 不符合你得需求，你可以创建自己的。

最后，你需要创建你自己的 scheduler 实例。Scheduler 实例需要给定一个名字，通知 RMI 设置，传入一个 JobStore 的实例和 ThreadPool。RMI 设置包括 scheduler 是否应该被创建成一个 RMI 的服务器对象（是其可用于远程连接），使用什么样的地址和端口等

等。StdSchedulerFactory（下面讨论）也可以远程创建一个 scheduler 实例，实际上是一个代理（RMI stubs）。更多关于 StdSchedulerFactory 的信息，参见第 26 页的 "Scheduler 工厂"

Scheduler 工厂

StdSchedulerFactory 是 org.quartz.SchedulerFactory 接口的一个实现。它使用一组属性（java.util.Properties）来创建 Quartz Scheduler 实例。这些属性通常从一个文件存储和加载，也可以由你的程序直接传给工厂。简单的调用工厂的 getScheduler()将产生一个 Scheduler，对其初始化（ThreadPool, JobStore 和数据源）并返回它的公共接口的句柄。

有一些简单的配置示例（包括属性说明）在 Quartz 发布包的“docs/config”目录下。你可以在 Quartz 的示例程序和示例代码中找到完整的文档。

DirectSchedulerFactory

DirectSchedulerFactory 是另一个 SchedulerFactory 实现。如果你想用更程序化的方法创建你自己的 Scheduler 实例，它会有帮助。基于下面两个原因一般不鼓励使用它：（1）它需要深入理解 Scheduler（2）它不允许使用配置（意味着你必须硬编码 Scheduler 的配置）。

日志

Quartz 使用 SLF4J 框架满足日志需求。为了“优化”框架设置（例如输出数量，输出的地方），你需要理解 SLF4J 框架，这超出了本文范围。

如果你想获得 trigger 触发和 job 执行的详细信息，你可能对 org.quartz.plugins.history.LoggingJobHistoryPlugin 和/或 org.quartz.plugins.history.LoggingTriggerHistoryPlugin 感兴趣。

Quartz 调度器的其他特性

插件

Quartz 提供 org.quartz.spi.SchedulerPlugin 接口为插件提供附加功能。

在 org.quartz.plugins 包中提供了 Quartz 附带插件各种实用功能的描述。

插件提供的功能，如 scheduler 启动后自动安排计划，记录 job 和 trigger 事件的历史记录，确保在 JVM 关闭时 scheduler 关闭干净。

JobFactory

当触发器触发时，在 scheduler 中相关的 job 通过 JobFactory 配置被实例化。Job 类上的 JobFactory 默认调用 newInstance()。你也许想创建你自己的 JobFactory 实例完成一些事诸如在你的 IoC 或 DI 容器中产生/初始化 job 实例。

观察 org.quartz.spi.JobFactory 接口，和相关的 Scheduler.setJobFactory(fact)方法。

Factory 自带 Job

Quartz 提供了大量实例的 job，你可以在你的应用中实现诸如发送短信执行 EJB 之类的事。

你可以在 org.quartz.jobs 包中找到它们的文档

高级功能

集群

集群当前可以使用 JDBC-Jobstore(JobStoreTX 或 JobStoreCMT)和 TerracottaJobStore 工作。包括负载平衡和故障转移功能（如果 JobDetail 的“请求复苏”标志设置为 true）。

使用 JobStoreTX 或 JobStoreCMT 集群

通过设置 org.quartz.jobStore.isClustered 为 true 来使用集群。集群中的每个实例应该使用相同的 quartz.properties 文件，以下情况可以例外：线程池的大小不同，org.quartz.scheduler.instanceId 的属性不同。集群中的每个节点必须有一个唯一的 instanceId，通过设置“AUTO”作为属性值这很容易实现（不需要不同的属性文件）。

重点：不要在单独的机器上运行集群，除非它们非常定期的使用某种时间同步服务或守护进程同步它们的时钟（每个时钟相差在一秒内）。如果你不清楚怎么做，参见 http://www.boulder.nist.gov/timefreq/service/its.htm。

重点：不要在非集群实例上将多个运行的实例设置为相同的表设置。你可能得到损坏的数据，并得到不可预测的行为。

每次执行只有一个节点触发 job。例如，如果 job 重复触发通知它每 10 秒触发一次，12:00:00 有个节点会运行 job，12:00:10 也会有一个节点将运行 job。每次运行的节点不一定相同——会或多或少的随机选择运行的节点。负载均衡机制就是为繁忙的 scheduler（很多 trigger）接近随机调度，仅仅在 scheduler 不繁忙的时候（一两个 trigger）容易使用同一个节点。

TerracottaJobStore 集群

使用 TerracottaJobStore 简单的配置 scheduler 的描述在第 25 页“TerracottaJobStore”中，你得 scheduler 将被设置为集群。

你也要考虑你设置 Terracotta server 的影响，特别是那些功能选项，例如持久化，为 HA 运行的 Terracotta server 阵列。

TerracottaJobStore 授权版提供了 Quartz 的高级特性，允许智能的为 job 目标提供合适的节点。

JTA 事务

JobStoreCMT 允许 Quartz 调度操作更大的 JTA 事务执行。

通过设置 org.quartz.scheduler.wrapJobExecutionInUserTransaction 为 true，job 也可以在 JTA 事务（用户事务）中执行。设置了这个选项，JTA 事务将在 job 执行方法调用前执行 begin()，在执行完成之后执行 commit()。这适用于所有的 job。

如果你想标明每个 job 是否应该在 JTA 事务中执行，你应该在 job 类上使用 @ExecuteInJTATransaction 注解。