Quartz2.2 版本开发手册

Java 版

2014 年 5 月 13 日星期二

What is Quartz?
Job Scheduling Library

Quartz is a richly featured, open source job scheduling library that can be integrated within virtually any Java application - from the smallest stand-alone application to the largest e-commerce system. Quartz can be used to create simple or complex schedules for executing tens, hundreds, or even tens-of-thousands of jobs; jobs whose tasks are defined as standard Java components that may execute virtually anything you may program them to do. The Quartz Scheduler includes many enterprise-class features, such as support for JTA transactions and clustering.

Quartz is freely usable, licensed under the Apache 2.0 license.

什么是 Quartz？
作业调度图书馆

Quartz 是一个功能丰富的，开源的作业调度库，可以在几乎任何 Java 应用程序集成 - 从最小的独立的应用程序，以最大的电子商务系统。 Quartz 可以用来创建简单或复杂的调度执行几十，几百，甚至是数万名成千上万的就业机会; job 被定义为标准的 Java 组件可以执行几乎任何你可以编程他们做。 Quartz 调度包括了许多企业级特性，如 JTA 事务和集群支持。

Quartz 是可自由使用的，基于 Apache2.0 许可协议发布。

目录

1.1 Quartz 手册 java 版-(一)使用 Quartz 3
1.2 Quartz 手册 java 版-(二)Jobs And Triggers 4
1.3 Quartz 手册 java 版-(三)更多关于 Jobs 和 JobDetails 6
1.4 Quartz 手册 java 版-(四)关于 Triggers 更多内容.....10
1.5 Quartz 手册 java 版-(五) SimpleTrigger.....13
1.6 Quartz 手册 java 版-(六)CronTrigger 15
1.7 Quartz 手册 java 版-(七)TriggerListeners 和 JobListeners.....17
1.8 Quartz 手册 java 版-(八)SchedulerListeners.....19
1.9 Quartz 手册 java 版-(九)JobStores.....20
1.10 Quartz 手册 java 版-(十)配置、资源使用以及 SchedulerFactory 23
1.11 Quartz 手册 java 版-(十一)高级（企业级）属性.....25
1.12 Quartz 手册 java 版-(十二)Quartz 的其他特性.....26

1.1 Quartz 手册 java 版-(一)使用 Quartz

使用 scheduler 之前应首先实例化它。使用 SchedulerFactory 可以完成 scheduler 的实例化。
用户可直接地实例化这个工厂类并且直接使用工厂的实例（例如下面的例子）。
一旦一个 scheduler 被实例化，它就可以被启动(start),并且处于驻留模式，直到被关闭(shutdown)。
注意，一旦 scheduler 被关闭（shutdown）,则它不能再重新启动，除非重新实例化它。
除非 scheduler 被启动或者不处于暂停状态，否则触发器不会被触发(任务也不能被执行)。
下面是一个代码片断，这个代码片断实例化并且启动了一个 scheduler，接着将一个要执行的任务纳入了进程。

```
SchedulerFactory schedFact = new org.quartz.impl.StdSchedulerFactory();

Scheduler sched = schedFact.getScheduler();

sched.start();

JobDetail jobDetail = new JobDetail("myJob",

        null,

        DumbJob.class);

Trigger trigger = TriggerUtils.makeHourlyTrigger(); // fire every hour
trigger.setStartTime(TriggerUtils.getEvenHourDate(new Date())); //
start on the next even hour
trigger.setName("myTrigger");

sched.scheduleJob(jobDetail, trigger);
```

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

1.2 Quartz 手册 java 版-(二)Jobs And Triggers

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

```
package org.quartz;
public interface Job {

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

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

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

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

SimpleTrigger 用来触发只需执行一次或者在给定时间触发并且重复 N 次且每次执行延迟一定时间的任务。
CronTrigger 按照日历触发，例如“每个周五”，每个月 10 日中午或者 10：15 分。
为什么要分为 Jobs 和 Triggers?很多任务日程管理器没有将 Jobs 和 Triggers 进行区分。
一些产品中只是将“job”简单地定义为一个带有一些小任务标识的执行时间。
其他产品则更像 Quartz 中 job 和 trigger 的联合。
而开发 Quartz 的时候，我们决定对日程和按照日程执行的工作进行分离。（从我们的观点来看）这有很多好处。
例如：jobs 可以被创建并且存储在 job scheduler 中，而不依赖于 trigger,而且，很多 triggers 可以关联一个 job.
另外的好处就是这种“松耦合”能使与日程中的 Job 相关的 trigger 过期后重新配置这些 Job，
这样以后就能够重新将这些 Job 纳入日程而不必重新定义它们。这样就可以更改或者替换 trigger 而不必重新定义与之相连的 job 标识符。

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

1.3 Quartz 手册 java 版-(三)更多关于 Jobs 和 JobDetails

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

在所实现的类成为真正的“Job”时，期望任务所具有的各种属性需要通知给 Quartz。
通过 JobDetail 类可以完成这个工作，这个类在前面的章节中曾简短提及过。
现在，我们花一些时间来讨论 Quartz 中 Jobs 的本质和 Job 实例的生命周期。首先让我们回顾一下第一课中所看到的代码片断

```
JobDetail jobDetail = new
JobDetail("myJob",                                  // job name

        sched.DEFAULT_GROUP,      // job group (you can also specify
'null' to use the default group)

        DumbJob.class);                 // the java class to execute

Trigger trigger = TriggerUtils.makeDailyTrigger(8, 30);
trigger.setStartTime(new Date());
trigger.setName("myTrigger");

sched.scheduleJob(jobDetail, trigger);
```

现在定义一个 DumbJob 类

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

        public void execute(JobExecutionContext context)
            throws JobExecutionException
        {
            System.err.println("DumbJob is executing.");
        }
}
```

注意我们传递给 scheduler 一个 JobDetail 实例,JobDetail 关联一个 job,提供 job 的 class,每次 scheduler 执行 job 时,在执行 execute(...)
这前会创建一个实例. job 必须有一个无参构造方法.

你可能想问如何提供配置 job 实例,或者保存 job 状态在执行过程中.答案是 JobDataMap.它是 JobDetail 的一部分.

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

的便捷方法。

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

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

```
public class DumbJob implements Job {

        public DumbJob() {
        }

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

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

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

            System.err.println("Instance " + instName + " of DumbJob says:
" + jobSays);
        }
}
```

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

Triggers 也可以有 JobDataMaps 与之相关联。当 scheduler 中的 Job 被多个有规律或者重复触发的 Triggers 所使用时非常有用。
对于每次独立的触发，你可为 Job 提供不同的输入数据。
从 Job 执行时的 JobExecutionContext 中取得 JobDataMap 是惯用手段，它融合了从 JobDetail 和从 Trigger 中获的 JobDataMap，
当有相同名字的键时，它用后者的值覆盖前者值。

下面给一个例子取数据从 JobExecutionContext 关联 JobDatMap 在 job 执行中

```
public class DumbJob implements Job {

        public DumbJob() {
        }

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

            JobDataMap dataMap = context.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 " + instName + " of DumbJob says:
" + jobSays);
        }
}
```

StatefulJob——有状态任务

现在，一些关于 Job 状态数据的附加论题：一个 Job 实例可以被定义为“有状态的”或者“无状态的”。

“无状态的”任务只拥有它们被加入到 scheduler 时所存储的 JobDataMap。这意味着，

在执行任务过程中任何对 Job Data Map 所作的更改都将丢失而且任务下次执行时也无法看到。

你可能会猜想出，有状态的任务恰好相反，它在任务的每次执行之后重新存储 JobDataMap。

有状态任务的一个副作用就是它不能并发执行。换句话说，如果任务有状态，那么当触发器在这个任务已经在执行的时候试图触发它，这个触发器就会被阻塞（等待），直到前面的执行完成。

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

Job ’Instances’ 任务“实例”

这个课程的最终观点或许已经很明确，可以创建一个单独的 Job 类，

并且通过创建多个 JobDetails 实例来将它的多个实例存储在 scheduler 中，

这样每个 JobDetails 对象都有它自己的一套属性和 JobDataMap，而且将它们都加入到 scheduler 中。

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

jobs 的其它属性
这里简短地总结一下通过 JobDetail 对象可以定义 Job 的其它属性。
Durability（持久性）-如果一个 Job 是不持久的， 一旦没有触发器与之关联，它就会被从 scheduler 中自动删除。
Volatility（无常性）-如果一个 Job 是无常的,在重新启动 Quartz i scheduler 时它不能被保持。
RequestsRecovery（请求恢复能力） -如果一个 Job 具备“请求恢复”能力，当它在执行时遇到 scheduler “硬性的关闭”
（例如：执行的过程崩溃，或者计算机被关机），那么当 scheduler 重新启动时，这个任务会被重新执行。
这种情况下，JobExecutionContext.isRecovering() 属性将是 true。
JobListeners（任务监听器） -一个 Job 如果有 0 个或者多个 JobListeners 监听器与之相关联，当这个 Job 执行时，监听器被会被通知。
更多有关 JobListeners 的讨论见 TriggerListeners & JobListeners 章节。

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

1.4 Quartz 手册 java 版-(四)关于 Triggers 更多内容

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

Calendars——日历

Quartz Calendar 对象在 trigger 被存储到 scheduler 时与 trigger 相关联。Calendar 对于在 trigger 触发日程中的采用批量世间非常有用。

例如：你想要创建一个在每个工作日上午 9：30 触发一个触发器，那么就添加一个排除所有节假日的日历。

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

```
package org.quartz;

public interface Calendar {

public boolean isTimeIncluded(long timeStamp);

public long getNextIncludedTime(long timeStamp);

}
```

注意，这些方法的参数都是 DateTime 型，你可以猜想出，它们的时间戳是毫秒的格式。这意味日历能够排除毫秒精度的时间。

最可能的是，你可能对排除整天的时间感兴趣。为了提供方便，Quartz 提供了一个 Quartz.Impl.Calendar.HolidayCalendar，

这个类可以排除整天的时间。

Calendars 必须被实例化，然后通过 addCalendar (..)方法注册到 scheduler 中。如果使用 HolidayCalendar，在实例化之后，

你可以使用它的 AddExcludedDate (DateTime excludedDate))方法来定义你想要从日程表中排除的时间。

同一个 calendar 实例可以被用于多个 trigger 中，如下：

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

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

Trigger trigger = TriggerUtils.makeHourlyTrigger(); // fire every one hour interval
trigger.setStartTime(TriggerUtils.getEvenHourDate(new Date())); // start on the next even hour
trigger.setName("myTrigger1");

trigger.setCalendarName("myHolidays");

// .. schedule job with trigger
```

```
Trigger trigger2 = TriggerUtils.makeDailyTrigger(8, 0); // fire every day
at 08:00
trigger.setStartTime(new Date()); // begin immediately
trigger2.setName("myTrigger2");

trigger2.setCalendarName("myHolidays");

// .. schedule job with trigger2
```

传入 SimpleTrigger 构造函数的参数的细节将在下章中介绍。但是，任何在日历中被排除的时间所要进行的触发都被取消。

Priority

有时，当有多个 Triggers 时，Quartz 没有足够的资源来同时立即处理 scheduled 的 trigger.所以你可能要控制哪个先执行.

在这种情况下，你要设置 Trigger 的 priority，如果 N 个 triggers 不会同时触发.但此时只有有限的几个线程可用.这时会先处

理级别高的 trigger.如果你没有设置级别，默认为 5，下面为一个例子:

```
Calendar cal = Calendar.getInstance();
cal.add(Calendar.MINUTE, 5);

Trigger trig1 = new SimpleTrigger("T1", "MyGroup", cal.getTime());
Trigger trig2 = new SimpleTrigger("T2", "MyGroup", cal.getTime());
Trigger trig3 = new SimpleTrigger("T3", "MyGroup", cal.getTime());

JobDetail jobDetail = new JobDetail("MyJob", "MyGroup", NoOpJob.class);

// Trigger1 does not have its priority set, so it defaults to 5
sched.scheduleJob(jobDetail, trig1);

// Trigger2 has its priority set to 10
trig2.setJobName(jobDetail.getName());
trig2.setPriority(10);
sched.scheduleJob(trig2);

// Trigger2 has its priority set to 1
trig3.setJobName(jobDetail.getName());
trig2.setPriority(1);
sched.scheduleJob(trig3);

// Five minutes from now, when the scheduler invokes these three triggers
// they will be allocated worker threads in decreasing order of their
// priority: Trigger2(10), Trigger1(5), Trigger3(1)
```

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

TriggerUtils - Triggers Made Easy（TriggerUtils——使 Triggers 变得容易）
TriggerUtils 类包含了创建触发器以及日期的便捷方法。使用这个类可以轻松地使触发器在每分钟，小时，日，星期，月等触发。
使用这个类也可以产生距离触发最近的妙、分或者小时，这对设定触发开始时间非常有用。

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

1.5 Quartz 手册 java 版-(五) SimpleTrigger

如果需要让任务只在某个时刻执行一次，或者，在某个时刻开始，然后按照某个时间间隔重复执行，
简单地说，如果你想让触发器在 2007 年 8 月 20 日上午 11：23：54 秒执行，然后每个隔 10 秒钟重复执行一次，并且这样重复 5 次。
那么 SimpleTrigger 就可以满足你的要求。

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

重复次数可能是 0，正数或者一个常量值 SimpleTrigger.REPEAT_INDEFINITELY。
重复间隔时间属性可能是 0，
正的 long 型，这个数字以毫秒为单位。注意：如果指定的重复间隔时间是 0，那么会导致触发器按照‘重复数量’定义的次数并发触发（或者接近并发）。
org.quartz.helpers.TriggerUtils 类对处理这样的循环也提供了很多支持。

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

SimpleTrigger 有几个不同的构造函数，下面我们来看看这结果构造函数：

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

```
SimpleTrigger Example 1 - Create a trigger that fires exactly once, ten seconds from now
long startTime = System.currentTimeMillis() + 10000L;
SimpleTrigger trigger = new SimpleTrigger("myTrigger", null,new Date(startTime),null,0,0L);
```

```
SimpleTrigger Example 2 - Create a trigger that fires immediately, then repeats every 60 seconds, forever
SimpleTrigger trigger = new SimpleTrigger("myTrigger", null, new Date(),null,SimpleTrigger.REPEAT_INDEFINITELY,60L * 1000L);
```

```
SimpleTrigger Example 3 - Create a trigger that fires immediately, then repeats every 10 seconds until 40 seconds from now
long endTime = System.currentTimeMillis() + 40000L;
SimpleTrigger trigger = new SimpleTrigger("myTrigger","myGroup", new
```

Date(),new Date(endTime), SimpleTrigger.REPEAT_INDEFINITELY,10L * 1000L);

SimpleTrigger Example 4 - Create a trigger that fires on March 17 of the year 2002 at precisely 10:30 am, and repeats 5 times (for a total of 6 firings) - with a 30 second delay between each firing
java.util.Calendar cal = new java.util.GregorianCalendar(2002,
cal.MARCH, 17);
cal.set(cal.HOUR, 10);
cal.set(cal.MINUTE, 30);
cal.set(cal.SECOND, 0);
cal.set(cal.MILLISECOND, 0);

Data startTime = cal.getTime()

SimpleTrigger trigger = new SimpleTrigger("myTrigger", null,
startTime,null,5,30L * 1000L);

SimpleTrigger Misfire Instructions——SimpleTrigger 的未触发指令
“未触发”发生时，SimpleTrigger 有几个指令可以用来通知 Quartz 进行相关处理。（“未触发”在上节课中介绍过了）。
这些指令以常量形式定义在 SimpleTrigger 本身，这些指令如下：
MISFIRE_INSTRUCTION_FIRE_NOW
MISFIRE_INSTRUCTION_RESCHEDULE_NOW_WITH_EXISTING_REPEAT_COUNT
MISFIRE_INSTRUCTION_RESCHEDULE_NOW_WITH_REMAINING_REPEAT_COUNT
MISFIRE_INSTRUCTION_RESCHEDULE_NEXT_WITH_REMAINING_COUNT
MISFIRE_INSTRUCTION_RESCHEDULE_NEXT_WITH_EXISTING_COUNT

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

如果使用 "smart policy" 指令,SimpleTriger 会基于配置和 SimpleTrigger 实例的状态动态的选择上面的指令.
SimpleTrigger.updateAfterMisfire() 会获取动态行为的详细信息.

1.6 Quartz 手册 java 版-(六)CronTrigger

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

使用 CronTrigger，你可以指定诸如“每个周五中午”，或者“每个工作日的 9:30”或者“从每个周一、周三、周五的上午
9：00 到上午 10：00 之间每隔五分钟”这样日程安排来触发。甚至，象 SimpleTrigger 一样，
CronTrigger 也有一个 StartTime 以指定日程从什么时候开始，也有一个（可选的）EndTime 以指定何时日程不再继续。
Cron 表达式被用来配置 CronTrigger 实例。Cron 表达式是一个由 7 个子表达式组成的字符串。

每个子表达式都描述了一个单独的日程细节。这些子表达式用空格分隔，分别表示：
1. Seconds 秒
2. Minutes 分钟
3. Hours 小时
4. Day-of-Month 月中的天
5. Month 月
6. Day-of-Week 周中的天
7. Year (optional field) 年（可选的域）

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

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

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

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

'/'字符用来表示值的增量，例如, 如果分钟域中放入'0/15'，它表示“每隔 15 分钟，从 0 开始”，

如果在份中域中使用'3/20'，则表示“小时中每隔 20 分钟，从第 3 分钟开始”或者另外相同的形式就是'3,23,43'。

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

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

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

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

例 1 - 一个简单的每隔 5 分钟触发一次的表达式
"0 0/5 * * * ?" CronTrigger
例 2 - 在每分钟的 10 秒后每隔 5 分钟触发一次的表达式(例如. 10:00:10 am, 10:05:10 等.)。
"10 0/5 * * * ?" CronTrigger
例 3 - 在每个周三和周五的 10：30，11：30，12：30 触发的表达式。
"0 30 10-13 ? * WED,FRI" CronTrigger
例 4 - 在每个月的 5 号，20 号的 8 点和 10 点之间每隔半个小时触发一次且不包括 10 点，只是 8：30，9：00 和 9：30 的表达式。
"0 0/30 8-9 5,20 * ?"
注意，对于单独触发器来说，有些日程需求可能过于复杂而不能用表达式表述，
例如：9：00 到 10：00 之间每隔 5 分钟触发一次，下午 1：00 到 10 点每隔 20 分钟触发一次。这个解决方案就是创建两个触发器，两个触发器都运行相同的任务。

1.7 Quartz 手册 java 版-(七)TriggerListeners 和 JobListeners

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

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

```
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);
}
```

任务相关的事件包括：即将被执行的任务的通知和任务已经执行完毕的通知。

```
public interface JobListener {

        public String getName();

        public void jobToBeExecuted(JobExecutionContext context);

        public void jobExecutionVetoed(JobExecutionContext context);

        public void jobWasExecuted(JobExecutionContext context,
                        JobExecutionException jobException);

}
```

使用你自定义的监听器
创建监听器很简单，创建一个实现 org.quartz.TriggerListener and/or org.quartz.JobListener 的接口。
监听器然后在执行的时候注册到 scheduler 中，而且必须给定一个名字（或者，它们必须通过他们的 Name 属性来介绍自己）。
监听器可以被注册为“全局”的或者“非全局”。“全局”监听器接收所有 triggers/jobs 产生的事件，

而“非全局”监听器只接受那些通过 getTriggerListenerNames() 或 getJobListenerNames()方法显式指定监听器名的 triggers/jobs 所产生的事件。

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

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

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

1.8 Quartz 手册 java 版-(八)SchedulerListeners

SchedulerListeners 同 TriggerListeners 及 JobListeners 非常相似，SchedulerListeners 只接收与特定 trigger 或 job 无关的 Scheduler 自身事件通知。
Scheduler 相关的事件包括：增加 job 或者 trigger，移除 Job 或者 trigger，scheduler 内部发生的错误，scheduler 将被关闭的通知，以及其他。

```
public interface SchedulerListener {

        public void jobScheduled(Trigger trigger);

        public void jobUnscheduled(String triggerName, String
triggerGroup);

        public void triggerFinalized(Trigger trigger);

        public void triggersPaused(String triggerName, String
triggerGroup);

        public void triggersResumed(String triggerName, String
triggerGroup);

        public void jobsPaused(String jobName, String jobGroup);

        public void jobsResumed(String jobName, String jobGroup);

        public void schedulerError(String msg, SchedulerException
cause);

        public void schedulerShutdown();

}
```

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

1.9 Quartz 手册 java 版-(九)JobStores

JobStore 负责保持对所有 scheduler“工作数据”追踪，这些工作数据包括：job（任务）,trigger（触发器）,calendar(日历)等。为你的 Quartz scheduler 选择合适的 JobStore 是非常重要的一步，幸运的是，如果你理解了不同的 JobStore 之间的差别，那么选择就变得非常简单。在提供产生 scheduler 实例的 SchedulerFactory 的属性文件中声明 scheduler 所使用的 JobStore（以及它的配置）。
注：不要在代码中直接使用 JobStore 实例，处于某些原因，很多人试图这么做。JobStore 是由 Quartz 自身在幕后使用。你必须告诉（通过配置）Quartz 使用哪个 JobStore，而你只是在你的代码中使用 Scheduler 接口完成工作。

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

JDBCJobStore
JDBCJobStore 通过 jdbc 保存所有数据到数据库,它的配置要比 RAMJobStore 稍微复杂，同时速度也没有那么快。
但是性能的缺陷不是非常差，尤其是如果你在数据库表的主键上建立索引。

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

一旦数据库表已经创建，在配置和启动 JDBCJobStore 之前，就需要作出一个更加重要的决策。你要决定在你的应用中需要什么类型的事务。

如果不想将 scheduling 命令绑到其他的事务上，那么你可以通过对 JobStore 使用 JobStoreTX 来让 Quartz 帮你管理事务（这是最普遍的选择）。

如是你需要 Quartz 使用其它事务(例如 j2ee 应用服务器),你需要使用 JobStoreStoreCMT(这时应用服务器容器会管理事务)

最后的疑问就是如何建立获得数据库联接的数据源（DataSource）。Quartz 属性中定义数据源是通过提供所有联接数据库的信息，
让 Quartz 自己创建和管理数据源。这里有很多种不同的方式,一种是 Quartz 提供所有数据库连接信息自己创建管理 DataSource.
另一种方式是 Quartz 使用数据源来管理应用服务器上运行的 Quartz.具体的信息例子在"docs/config"

要使用 JDBCJobStore（假定使用 StdSchedulerFactory），首先需要设置 Quartz 配置中的 quartz.jobStore.type 属性为
org.quartz.impl.jdbcjobstore.JobStoreTX or
org.quartz.impl.jdbcjobstore.JobStoreCMT
配置
org.quartz.jobStore.class = org.quartz.impl.jdbcjobstore.JobStoreTX

下一步，需要为 JobStore 选择一个 DriverDelegate ， DriverDelegate 负责做指定数据库的所有 JDBC 工作。
StdJDBCDelegate 是一个使用 vanilla" JDBC 代码（以及 SQL 语句）来完成工作的代理。
如果数据库没有其他指定的代理，那么就试用这个代理。只有当使用 StdJDBCDelegate 发生问题时，
我们才会使用数据库特定的代理（这看起来非常乐观。其他的代理可以在 org.quartz.impl.jdbcjobstore 包找到。）。
其他的代理包括 PostgreSQLDelegate （ 专为 PostgreSQL 7.x)。

一旦选择好了代理，就将它的名字设置给 JDBCJobStore 。
org.quartz.jobStore.driverDelegateClass =
org.quartz.impl.jdbcjobstore.StdJDBCDelegate
接下来，需要为 JobStore 指定所使用的数据库表前缀（前面讨论过）。
org.quartz.jobStore.tablePrefix = QRTZ_
最后，你需要设计数据源使用的 jobStore,这个数据源的名字必须在 Quartz.properties 中定义.
org.quartz.jobStore.dataSource = myDS

如果 Scheduler 非常忙（比如，执行的任务数量差不多和线程池的数量相同，那么你需要正确地配置 DataSource 的连接数量为线程池数量+1。
为了指示 JDBCJobStore 所有的 JobDataMaps 中的值都是字符串，
并且能以“名字-值”对的方式存储而不是以复杂对象的序列化形式存储在 BLOB 字段中，
应设置 org.quartz.jobStore.useProperties 配置参数的值为"true"(这是缺

省的方式)。
这样做，从长远来看非常安全，这样避免了对存储在 BLOB 中的非字符串的序列化对象的类型转换问题。

1.10 Quartz 手册 java 版-(十)配置、资源使用以及 SchedulerFactory

Quartz 以模块方式构架，因此，要使它运行，几个组件必须很好的咬合在一起。幸运的是，已经有了一些现存的助手可以完成这些工作。
在 Quartz 进行工作之前需要被配置的组件主要有：
ThreadPool 线程池
JobStore
DataSources （如果需要）
Scheduler 本身

ThreadPool（线程池）为 Quartz 运行任务时提供了一些线程。池中的线程越多，那么并发运行的任务数就越多。
但是，过多的线程会降低系统的运行速度。大多数用户发现 5 个或者相近的线程就已经足够了，
因为任何给定的时间段内都不超过 100 个任务要运行，而且这些任务不会在同一时刻运行，同时任务活动时间很短（很快就结束了）。
其他的用户发现需要 10，15，50，甚至 100 个线程，因为每个 schedules 都有成千上万的触发器，
并且在给定的时刻会有平均 10 到 100 个任务在运行。确定 schedule 的线程池中的线程数量的合理值取决于用 scheduler 来做什么。
除了尽可能少地设置线程数量，使得任务执行时线程够用外（由于计算机资源的有限性），没有其他实用的准则。
注意：如果触发器触发的时间到了，却没有可用的线程，那么 Quartz 将会让这个任务等待，直到有线程可用。
这样，任务的执行将比它因该执行的时间晚一些毫秒。如果 scheduler 的配置的“未触发极限”时限中仍然没有线程可用，
这甚至会导致“未触发(misfire)”。
ThreadPool 接口定义在 org.quartz.spi 中，你也可以创建一个自己的 ThreadPool（线程池）实现，
Quartz 打包了一个简单（但非常满意的）的线程池，名为：
org.quartz.simpl.SimpleThreadPool，
这个线程池只是简单地在它的池中保持固定数量的线程，不增长也不缩小。但是它非常健壮且经过良好的测试，
差不多每个 Quartz 用户都使用这个池。

JobStores 和 DataSrouces 在(九)中已经讨论过，值得注意的一个事实是所有的 JobStores 都实现了 org.quartz.spi.JobStore 接口，
如果捆绑的 JobStores 不能满足你的要求，你可以自己开发一个。

最后你需要创建自己的 Scheduler 实例。Scheduler 本身需要给定一个名字告诉 RMI 设置,处理的 JobStore 和 ThreadPool 实例

StdSchedulerFactory
StdSchedulerFactory 是对 org.quartz.SchedulerFactory 接口的一个实现。
是使用一套属性(java.util.Properties)来创建和初始化 Quartz Scheduler。

这些属性通常在文件中存储和加载。也可以通过编写程序来直接操作工厂。
简单地调用工厂的 getScheduler()就可以产生一个 scheduler，初始化（以及它的 ThreadPool、JobStore 和 DataSources），
并且返回一个公共的接口。

这里有许多例子在的配置在"docs/config"

DirectSchedulerFactory
DirectSchedulerFactory 是 SchedulerFactory 的另一个实现。它对于那些希望用更加程序化的方式创建 Scheduler 非常有用。
不鼓励使用它的原因如下：
(1) 它需要用户非常了解他们想要干什么。
(2) 它不允许声明式的配置。换句话说，它使用硬编码的方式设置 scheduler。

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

1.11 Quartz 手册 java 版-(十一)高级（企业级）属性

Clustering 集群
目前，集群只能用在使用 JDBC-Jobstore (JobStoreTX or JobStoreCMT)的情况。特新包括负载均衡和容错（如果 JobDetail 的"request recovery"标记被设置为 true）。
设置" org.quartz.jobStore.isClustered"属性为 true 才可以集群，集群中的每个实例都使用 quartz.properties 的相同拷贝。
集群所使用属性文件的例外是一致的，下面是允许的例外：不同的线程池数量，"org.quartz.scheduler.instanceId"的不同属性值。
集群中的每个节点必须有唯一的 instanceId，通过替换这个属性的值为"AUTO"就可以轻松做到（不要不同的属性文件）。
除非使用某些运行非常有整齐（时钟必须同步在一秒之内）的时间同步服务来同步不同计算机的时钟外，
不要将集群运行在不同的计算机三行。
不要在一套数据库表上运行未集群的实例。这会导致严重的数据冲突，及不可预知的行为。

JTA Transactions
像(九)中一样,JobStroeCMT 允许 Quartz 大量 JTA 事务
jobs 也能执行 JTA 事务通过设置
org.quartz.scheduler.wrapJobExecutionInUserTransaction=true,设置后
JTA 事务 begin()于 job execute()调用
commit()于 execute()结束

1.12 Quartz 手册 java 版-(十二)Quartz 的其他特性

Plug-Ins 插件
Quartz 提供了一个接口(org.quartz.spi.SchedulerPlugin)来插入附加的功能。
随 Quartz 打包儿来的插件有很多有用的功能，它们在 org.quartz.plugins 包中找到。他们提供了诸如自动安排任务的日程，
将任务和触发器事件的历史记入日志以及虚拟机退出时确保干净地关闭 scheduler 等的功能。

JobFactory
当触发器触发时，与之相关联的任务被 Scheduler 中配置的 JobFactory 所实例华。缺省的 JobFactory 只是简单调用 newInstance()地创建一个 Job 实例。
你也许想创建自己的 JobFactory 实现，以完成诸如让应用的 IoC 或者 DI 容器产生/初始化 job 实例的功能。
查看 org.quartz.spi.JobFactory 接口及与之相关的 Scheduler.setJobFactory(fact)方法。

'Factory-Shipped' Jobs
Quartz 也提供了一些可以在你的应用中使用的实用的 Jobs,比如，发邮件、调用远程对象。
这些外来的 Job 可以在 org.quartz.jobs 命名空间里中找到。