**Quartz体系结构：**

明白Quartz怎么用，首先要了解Scheduler(调度器)、Job(任务)和Trigger(触发器)这3个核心的概念。

**1. Job：**是一个接口，只定义一个方法execute(JobExecutionContext context)，在实现接口的execute方法中编写所需要定时执行的Job(任务)， JobExecutionContext类提供了调度应用的一些信息。Job运行时的信息保存在JobDataMap实例中；

**2. JobDetail：**Quartz每次调度Job时， 都重新创建一个Job实例， 所以它不直接接受一个Job的实例，相反它接收一个Job实现类(JobDetail:描述Job的实现类及其它相关的静态信息，如Job名字、描述、关联监听器等信息)，以便运行时通过newInstance()的反射机制实例化Job。

**3. Trigger：**是一个类，描述触发Job执行的时间触发规则。主要有SimpleTrigger和CronTrigger这两个子类。当且仅当需调度一次或者以固定时间间隔周期执行调度，SimpleTrigger是最适合的选择；而CronTrigger则可以通过Cron表达式定义出各种复杂时间规则的调度方案：如工作日周一到周五的15：00~16：00执行调度等；

  Cron表达式的格式：秒 分 时 日 月 周 年(可选)。  
               字段名                 允许的值                        允许的特殊字符    
               秒                         0-59                               , - \* /    
               分                         0-59                               , - \* /    
               小时                   0-23                                 , - \* /    
               日                         1-31                               , - \* ? / L W C    
               月                         1-12 or JAN-DEC           , - \* /    
               周几                     1-7 or SUN-SAT             , - \* ? / L C #      SUN, MON, TUE, WED, THU, FRI and SAT  
               年 (可选字段)     empty, 1970-2099            , - \* /  
  
               “?”字符：表示不确定的值  
               “,”字符：指定数个值  
               “-”字符：指定一个值的范围  
               “/”字符：指定一个值的增加幅度。n/m表示从n开始，每次增加m  
               “L”字符：用在日表示一个月中的最后一天，用在周表示该月最后一个星期X  
               “W”字符：指定离给定日期最近的工作日(周一到周五)  
               “#”字符：表示该月第几个周X。6#3表示该月第3个周五

         Cron表达式范例：  
                 每隔5秒执行一次：\*/5 \* \* \* \* ?  
                 每隔1分钟执行一次：0 \*/1 \* \* \* ?  
                 每天23点执行一次：0 0 23 \* \* ?  
                 每天凌晨1点执行一次：0 0 1 \* \* ?  
                 每月1号凌晨1点执行一次：0 0 1 1 \* ?  
                 每月最后一天23点执行一次：0 0 23 L \* ?  
                 每周星期天凌晨1点实行一次：0 0 1 ? \* L  
                 在26分、29分、33分执行一次：0 26,29,33 \* \* \* ?  
                 每天的0点、13点、18点、21点都执行一次：0 0 0,13,18,21 \* \* ?

**4. Calendar：**org.quartz.Calendar和java.util.Calendar不同， 它是一些日历特定时间点的集合（可以简单地将org.quartz.Calendar看作java.util.Calendar的集合——java.util.Calendar代表一个日历时间点，无特殊说明后面的Calendar即指org.quartz.Calendar）。 一个Trigger可以和多个Calendar关联， 以便排除或包含某些时间点。

假设，我们安排每周星期一早上10:00执行任务，但是如果碰到法定的节日，任务则不执行，这时就需要在Trigger触发机制的基础上使用Calendar进行定点排除。针对不同时间段类型，Quartz在org.quartz.impl.calendar包下提供了若干个Calendar的实现类，如AnnualCalendar、MonthlyCalendar、WeeklyCalendar分别针对每年、每月和每周进行定义；

**5. Scheduler：**代表一个Quartz的独立运行容器， Trigger和JobDetail可以注册到Scheduler中， 两者在Scheduler中拥有各自的组及名称， 组及名称是Scheduler查找定位容器中某一对象的依据， Trigger的组及名称必须唯一， JobDetail的组和名称也必须唯一（但可以和Trigger的组和名称相同，因为它们是不同类型的）。Scheduler定义了多个接口方法， 允许外部通过组及名称访问和控制容器中Trigger和JobDetail。

Scheduler可以将Trigger绑定到某一JobDetail中， 这样当Trigger触发时， 对应的Job就被执行。一个Job可以对应多个Trigger， 但一个Trigger只能对应一个Job。可以通过SchedulerFactory创建一个Scheduler实例。Scheduler拥有一个SchedulerContext，它类似于ServletContext，保存着Scheduler上下文信息，Job和Trigger都可以访问SchedulerContext内的信息。SchedulerContext内部通过一个Map，以键值对的方式维护这些上下文数据，SchedulerContext为保存和获取数据提供了多个put()和getXxx()的方法。可以通过Scheduler# getContext()获取对应的SchedulerContext实例；

**6. ThreadPool：**Scheduler使用一个线程池作为任务运行的基础设施，任务通过共享线程池中的线程提高运行效率。  
Job有一个StatefulJob子接口，代表有状态的任务，该接口是一个没有方法的标签接口，其目的是让Quartz知道任务的类型，以便采用不同的执行方案。无状态任务在执行时拥有自己的JobDataMap拷贝，对JobDataMap的更改不会影响下次的执行。而有状态任务共享共享同一个JobDataMap实例，每次任务执行对JobDataMap所做的更改会保存下来，后面的执行可以看到这个更改，也即每次执行任务后都会对后面的执行发生影响。  
正因为这个原因，无状态的Job可以并发执行，而有状态的StatefulJob不能并发执行，这意味着如果前次的StatefulJob还没有执行完毕，下一次的任务将阻塞等待，直到前次任务执行完毕。有状态任务比无状态任务需要考虑更多的因素，程序往往拥有更高的复杂度，因此除非必要，应该尽量使用无状态的Job。  
如果Quartz使用了数据库持久化任务调度信息，无状态的JobDataMap仅会在Scheduler注册任务时保持一次，而有状态任务对应的JobDataMap在每次执行任务后都会进行保存。  
Trigger自身也可以拥有一个JobDataMap，其关联的Job可以通过JobExecutionContext#getTrigger().getJobDataMap()获取Trigger中的JobDataMap。不管是有状态还是无状态的任务，在任务执行期间对Trigger的JobDataMap所做的更改都不会进行持久，也即不会对下次的执行产生影响。

Quartz拥有完善的事件和监听体系，大部分组件都拥有事件，如任务执行前事件、任务执行后事件、触发器触发前事件、触发后事件、调度器开始事件、关闭事件等等，可以注册相应的监听器处理感兴趣的事件。

**Quartz编程API几个重要接口**

* **Scheduler** - 用于与调度程序交互的主程序接口。
* **Job** - 我们预先定义的希望在未来时间能被调度程序执行的任务类，如上一节的HelloJob类。
* **JobDetail** - 使用**JobDetail**来定义定时任务的实例。
* **Trigger** - 触发器，表明任务在什么时候会执行。定义了一个已经被安排的任务将会在什么时候执行的时间条件
* **JobBuilder** -用于声明一个任务实例，也可以定义关于该任务的详情比如任务名、组名等，这个声明的实例将会作为一个实际执行的任务。
* **TriggerBuilder** - 触发器创建器，用于创建触发器trigger实例。

# SchedulerFactoryBean 配置

配置SchedulerFactoryBean  
Quartz的SchedulerFactory是标准的工厂类，不太适合在Spring环境下使用。此外，为了保证Scheduler能够感知Spring容器的生命周期，完成自动启动和关闭的操作，必须让Scheduler和Spring容器的生命周期相关联。以便在Spring容器启动后，Scheduler自动开始工作，而在Spring容器关闭前，自动关闭Scheduler。为此，Spring提供SchedulerFactoryBean，这个FactoryBean大致拥有以下的功能：  
1)以更具Bean风格的方式为Scheduler提供配置信息；  
2)让Scheduler和Spring容器的生命周期建立关联，相生相息；  
3)通过属性配置部分或全部代替Quartz自身的配置文件。  
spring容器中的bean只能放到SchedulerContext里面传入job中。

SchedulerFactoryBean属性介绍：  
●triggers：triggers属性为Trigger[]类型，可以通过该属性注册多个Trigger  
●calendars：类型为Map，通过该属性向Scheduler注册Calendar；  
●jobDetails：类型为JobDetail[]，通过该属性向Scheduler注册JobDetail；  
●autoStartup：SchedulerFactoryBean在初始化后是否马上启动Scheduler，默认为true。如果设置为false，需要手工启动Scheduler；  
●startupDelay：在SchedulerFactoryBean初始化完成后，延迟多少秒启动Scheduler，默认为0，表示马上启动。如果并非马上拥有需要执行的任务，可通过startupDelay属性让Scheduler延迟一小段时间后启动，以便让Spring能够更快初始化容器中剩余的Bean；  
  
SchedulerFactoryBean的一个重要功能是允许你将Quartz配置文件中的信息转移到Spring配置文件中，带来的好处是，配置信息的集中化管理，同时我们不必熟悉多种框架的配置文件结构。回忆一个Spring集成JPA、Hibernate框架，就知道这是Spring在集成第三方框架经常采用的招数之一。SchedulerFactoryBean通过以下属性代替框架的自身配置文件：  
●dataSource：当需要使用数据库来持久化任务调度数据时，你可以在Quartz中配置数据源，也可以直接在Spring中通过dataSource指定一个Spring管理的数据源。如果指定了该属性，即使quartz.properties中已经定义了数据源，也会被此dataSource覆盖；  
●transactionManager：可以通过该属性设置一个Spring事务管理器。在设置dataSource时，Spring强烈推荐你使用一个事务管理器，否则数据表锁定可能不能正常工作；  
●nonTransactionalDataSource：在全局事务的情况下，如果你不希望Scheduler执行化数据操作参与到全局事务中，则可以通过该属性指定数据源。在Spring本地事务的情况下，使用dataSource属性就足够了；  
●quartzProperties：类型为Properties，允许你在Spring中定义Quartz的属性。其值将覆盖quartz.properties配置文件中的设置，这些属性必须是Quartz能够识别的合法属性，在配置时，你可以需要查看Quartz的相关文档。

# 任务调度框架Quartz

## 任务调度框架Quartz实例详解深入理解Scheduler,Job,Trigger,JobDetail

**Scheduler 调度程序-任务执行计划表，只有安排进执行计划的任务Job（通过scheduler.scheduleJob方法安排进执行计划），当它预先定义的执行时间到了的时候（任务触发trigger），该任务才会执行。**

**Quartz编程API几个重要接口**

* **Scheduler** - 用于与调度程序交互的主程序接口。
* **Job** - 我们预先定义的希望在未来时间能被调度程序执行的任务类，如上一节的HelloJob类。
* **JobDetail** - 使用**JobDetail**来定义定时任务的实例。
* **Trigger** - 触发器，表明任务在什么时候会执行。定义了一个已经被安排的任务将会在什么时候执行的时间条件。
* **JobBuilder** -用于声明一个任务实例，也可以定义关于该任务的详情比如任务名、组名等，这个声明的实例将会作为一个实际执行的任务。
* **TriggerBuilder** - 触发器创建器，用于创建触发器trigger实例。

**Scheduler调度程序、SchedulerFactory调度程序工厂**

**org.quartz.Scheduler**这是Quartz 调度程序的主要接口。   
**Scheduler**维护了一个**JobDetails** 和**Triggers**的注册表。一旦在Scheduler注册过了，当定时任务触发时间一到，调度程序就会负责执行预先定义的**Job**。

**调度程序Scheduler实例**是通过**SchedulerFactory工厂**来创建的。一个已经创建的scheduler ，可以通过同一个工厂实例来获取它。 调度程序创建之后，它只是出于”待机”状态，必须在任务执行前调用scheduler的**start()**方法启用调度程序。你还可以使用shutdown()方法关闭调度程序，使用isShutdown()方法判断该调度程序是否已经处于关闭状态。通过Scheduler的scheduleJob(…)方法的几个重载方法将任务纳入调度程序中。使用是scheduleJob(JobDetail jobDetail, Trigger trigger)方法将我们预先定义的定时任务安排进调度计划中。任务安排之后，你就可以调用start()方法启动调度程序了，当任务触发时间到了的时候，该任务将被执行。

## Job定时任务实例类

一个任务是一个实现***org.quartz.Job***接口的类，**任务类必须含有空构造器**，它只有一个简单的方法：

void execute(JobExecutionContext context) throws JobExecutionException;

**Job实例化的过程**

你可以创建一个Job类，在调度程序(任务计划表)中创建很多JobDetai可以存储很多初始化定义信息——每一个都可以设置自己的属性和JobDataMap——将他们全部添加到调度程序中去。

你可以创建一个任务类实现Job接口，不妨称之为”SalesReportJob”，我们用它做销售报表使用。我们可以通过**JobDataMap**指定销售员的名称和销售报表的依据等等。这就会创建多个**JobDetails**了，例如”SalesReportForJoe”，”SalesReportForMike”分别对应在JobDataMap中指定的名字”joe”和”mike”。

重要：**当触发器的执行时间到了的时候，会加载与之关联的JobDetail，并在调度程序Scheduler中通过JobFactory的配置实例化它引用的Job。JobFactory 调用newInstance()创建一个任务实例，然后调用setter 方法设置在JobDataMap定义好的名字。你可以实现JobFactory，比如使用IOC或DI机制初始化的任务实例。**

**Job的声明和并发**

关于Job的声明和并发需要说明一下，以下一对注解使用在你的Job类中，可以影响Quartz的行为：   
***@DisallowConcurrentExecution*** : 可以添加到你的任务类中，它会告诉Quartz不要执行多个任务实例。   
注意措辞，在上面的”SalesReportJob”类添加该注解，将会只有一个”SalesReportForJoe”实例在给定的时间执行，但是”SalesReportForMike”是可以执行的。**这个约束是基于JobDetail的，而不是基于任务类的**。

***@PersistJobDataAfterExecution*** : 告诉Quartz在任务执行成功完毕之后（没有抛出异常），修改JobDetail的JobDataMap备份，以供下一个任务使用。

如果你使用了**@PersistJobDataAfterExecution**注解的话，强烈建议同时使用**@DisallowConcurrentExecution**注解，以避免当两个同样的job并发执行的时候产生的存储数据迷惑。

**Job的其他一些属性**

* **持久化** - 如果一个任务不是持久化的，则当没有触发器关联它的时候，Quartz会从scheduler中删除它。
* **请求恢复** - 如果一个任务请求恢复，一般是该任务执行期间发生了系统崩溃或者其他关闭进程的操作，当服务再次启动的时候，会再次执行该任务。这种情况下，JobExecutionContext.isRecovering()会返回true。

## JobDetail定义任务实例的一些属性特征

org.quartz.JobDetail接口负责传输给定的任务实例的属性到Scheduler。**JobDetail**是通过**JobBuilder**创建的。

Quartz不会存储一个真实的Job类实例，但是允许你通过**JobDetail**定义一个任务实例——**JobDetail**是用来定义任务实例的。

任务Job有一个名称name 和组group 来关联。在一个Scheduler中这二者的组合必须是唯一的。

触发器任务计划执行表的执行”机制”。多个触发器可以指向同一个工作，但一个触发器只能指向一个工作。

**JobDataMap任务数据映射**

**JobDataMap**用来保存任务实例的状态信息。   
当一个Job被添加到调度程序(任务执行计划表)scheduler的时候，JobDataMap实例就会存储一次关于该任务的状态信息数据。也可以使用**@PersistJobDataAfterExecution**注解标明在一个任务执行完毕之后就存储一次。

**JobDataMap实例也可以村粗一个触发器trigger。这是非常有用的，特别是当你的任务被多个触发器引用的时候，根据不同的触发时机，你可以提供不同的输入条件。**

**JobExecutionContext**也可以再执行时包含一个方便的**JobDataMap**，它合并了触发器的 JobDataMap (如果有的话)和Job的 JobDataMap (如果有的话)。

## Trigger触发器

Trigger触发器，可以理解为安排了一个任务，这个任务是在每年9月10日早上9点向你敬爱的老师发送一天祝福短信，触发器就是指**每年9月10日早上9点**触发执行这个任务。

触发器使用TriggerBuilder来实例化。   
触发器有一个TriggerKey关联，这在一个Scheduler中必须是唯一的。   
触发器任务计划执行表的执行”机制”。多个触发器可以指向同一个工作，但一个触发器只能指向一个工作。   
触发器可以传送数据给job——通过将数据放进触发器的JobDataMap。

**触发器常用属性**

触发器也有很多属性，这些属性都是在使用TriggerBuilder 定义触发器时设置的。

* TriggerKey - 唯一标识触发器，这在一个Scheduler中必须是唯一的
* “startTime” - 开始时间，通常使用startAt(java.util.Date)
* “endTime” - 结束时间，设置了结束时间则在这之后，不再触发

**触发器的优先级**

有时候，你会安排很多任务，但是Quartz并没有更多的资源去处理它。这种情况下，你必须需要很好地控制哪个任务先执行了。这时候你可以使用设置**priority**属性(使用方法withPriority(int))来控制触发器的优先级。

**注意**：优先级只有触发器出发时间一样的时候才有意义。   
**注意**：当一个任务请求恢复执行时，它的优先级和原始优先级是一样的。

## JobBuilder用于创建JobDetail;TriggerBuilder 用于创建触发器Trigger

**JobBuilder用于创建JobDetail**。总是把保持在有效状态，合理的使用默认设置在你调用build() 方法的时候。如果你没有调用**withIdentity(..)**指定job的名字，它会自动给你生成一个。

**TriggerBuilder 用于创建触发器Trigger**。如果你没有调用**withSchedule(..)**方法，会使用默认的schedule 。如果没有使用**withIdentity(..)** 会自动生成一个触发器名称给你。

## 简单触发器SimpleTrigger

**SimpleTrigger**是接口Trigger的一个具体实现，它可以触发一个已经安排进调度程序（任务执行计划）的任务，并可以指定时间间隔重复执行该任务。

SimpleTrigger 包含几个特点：开始时间、结束时间、重复次数以及重复执行的时间间隔。

重复的次数可以是零，一个正整数，或常量SimpleTrigger.REPEAT\_INDEFINITELY。   
重复执行的时间间隔可以是零，或者long类型的数值表示毫秒。值得注意的是，零重复间隔会造成触发器同时发生（或接近同时作为调度管理）。   
“结束时间”的属性会重写重复的次数，这可能是有用的，如果你想创建一个触发器，如每10秒触发一次，直到一个给定的时刻，而不是要计算的次数，它会在开始时间和结束时间重复执行。结束时间一到，就算你指定了重复次数很多次(比如执行10W次)，但是时间一到它将不再执行。

**SimpleTrigger**实例创建依赖于TriggerBuilder 和SimpleScheduleBuilder ，使用Quartz提供的DSL风格创建触发器实例，   
首先静态引入builder，引入时间builder——**DateBuilder:**

**简单触发器SimpleTrigger**

**SimpleTrigger**是接口Trigger的一个具体实现，它可以触发一个已经安排进调度程序（任务执行计划）的任务，并可以指定时间间隔重复执行该任务。

SimpleTrigger 包含几个特点：开始时间、结束时间、重复次数以及重复执行的时间间隔。

重复的次数可以是零，一个正整数，或常量SimpleTrigger.REPEAT\_INDEFINITELY。   
重复执行的时间间隔可以是零，或者long类型的数值表示毫秒。值得注意的是，零重复间隔会造成触发器同时发生（或接近同时作为调度管理）。   
“结束时间”的属性会重写重复的次数，这可能是有用的，如果你想创建一个触发器，如每10秒触发一次，直到一个给定的时刻，而不是要计算的次数，它会在开始时间和结束时间重复执行。结束时间一到，就算你指定了重复次数很多次(比如执行10W次)，但是时间一到它将不再执行。

**SimpleTrigger**实例创建依赖于TriggerBuilder 和SimpleScheduleBuilder ，使用Quartz提供的DSL风格创建触发器实例，   
首先静态引入builder，引入时间builder——**DateBuilder:**

**创建一个指定时间开始执行，但是不重复的触发器**

使用startAt(java.util.Date)设置触发器的第一次执行时间:

使用**SimpleScheduleBuilder**的withIntervalInSeconds(N)方法可以指定间隔N秒就执行一次；withRepeatCount(M)可以指定执行次数M。

使用DateBuilder的futureDate方法可以指定在未来时间执行。

.startAt(futureDate(5, IntervalUnit.MINUTE))

使用TriggerBuilder的startNow()方法立即触发(scheduler调用start时算起，视优先级而定)；   
withIntervalInMinutes(5)每个5分钟执行一次；   
repeatForever()一直重复；   
endAt(dateOf(22, 0, 0))知道22:00终结触发器

## 基于Cron表达式的触发器CronTrigger

Cron表达式，实际上是由七个子表达式组成的字符串，它描述了不同的调度细节。这些子表达式是用空格分隔的，并表示：   
秒   
分   
时   
月中的天   
月   
周中的天   
年(可选项)

**例如: “0 0 12 ? \* WED” 表示 “个星期三的12点”**

"0 0 7 ? \* MON-FRI" 表示 "每个工作日的7点"

"0 0 19 ? \* MON,WED,FRI" 表示 "周一、周三和周五的19点"

"0 0 14 ? \* MON-WED,SAT" 表示 "周一到周三以及周六的14点"

**Cron表达式的规则说明**

所有字段都有一组可以指定的有效值。

* **数字 0 到 59 可以表示秒和分**
* **0到23可以表示小时**
* **月中的天可以使用1到31的数值, 但是你要注意该月的天数!**
* **月用0 到 11之间的数值表示, 或者使用JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV 和 DEC来表示1-12月**
* **一周中的天试用1到7表示 (1 表示 周日) 或者使用 SUN, MON, TUE, WED, THU, FRI 和 SAT**

## Quartz系统参数配置详解

**org.quartz.scheduler.instanceName**

使用StdSchedulerFactory的 getScheduler()方法创建的scheduler实例名称，在同一个程序中可以根据该名称来区分scheduler。如果实在集群环境中使用，你必须使用同一个名称——集群环境下”逻辑”相同的scheduler。

**org.quartz.scheduler.instanceId**

scheduler实例的标志id，必须是全局唯一的，即使在集群环境中”逻辑”相同的scheduler。 或者可以使用“SYS\_PROP”通过系统属性设置id。

**org.quartz.scheduler.instanceIdGenerator.class**

只有在”org.quartz.scheduler.instanceId”设置为”AUTO”的时候才使用该属性设置。默认情况下，“org.quartz.simpl.SimpleInstanceIdGenerator”是基于instanceId和时间戳来自动生成的。其他的id生成器的实现包括**SystemPropertyInstanceIdGenerator** 从系统属性获取 “org.quartz.scheduler.instanceId”, 和**HostnameInstanceIdGenerator** 使用主机名 (InetAddress.getLocalHost().getHostName())。你也可以自定义生成方式哦。

**org.quartz.scheduler.threadName**

指定线程名，如果不指定的话，会自动使用org.quartz.scheduler.instanceName属性值加上后缀字符串”\_QuartzSchedulerThread”.

**org.quartz.scheduler.makeSchedulerThreadDaemon**

指定scheduler的主线程是否为后台线程。

**org.quartz.scheduler.threadsInheritContextClassLoaderOfInitializer**

指定Quartz生成的线程是否继承初始化线程的上下文类加载器。这会影响Quartz的主调度线程、JDBCJobStore的”熄火”处理线程、集群回复线程和线程池里的线程。 将该值设置为“true”可以帮助类加载，JNDI查找，并在应用程序服务器上使用Quartz等相关问题。

**org.quartz.scheduler.idleWaitTime**

在调度程序空闲的时候，重复查询是否有可用触发器的等待时间。通常并不会设置为true，除非你是用XA事务，并且延迟触发会导致问题的场景。 5000ms以下是不推荐的，因为它会导致过的的数据库查询。1000ms以下是非法的。

**org.quartz.scheduler.dbFailureRetryInterval**

连接超时重试连接的间隔。使用 RamJobStore时，该参数并没什么用。

**org.quartz.scheduler.classLoadHelper.class**

默认最可靠的方式就是指定”org.quartz.simpl.CascadingClassLoadHelper”，没必要指定其他类。

**org.quartz.scheduler.jobFactory.class**

指定JobFactory的类(接口)名称。负责实例化jobClass。默认是”org.quartz.simpl.PropertySettingJobFactory”,只是在job被执行的时候简单调用newInstance()实例化一个job类。PropertySettingJobFactory 会使用反射机制通过SchedulerContext、 Job、Trigger和 JobDataMaps设置job bean的属性。

使用JTA事务时，可以设置事务相关的属性

**org.quartz.scheduler.userTransactionURL**

设置Quartz能够加载UserTransaction换利器的JNDI的 URL。默认值是”java:comp/UserTransaction”。Websphere 的用户可能会设置为“jta/usertransaction。只有在Quartz使用JobStoreCMT的时候，才会使用该属性，并且**org.quartz.scheduler.wrapJobExecutionInUserTransaction**也会设置为true。

**org.quartz.scheduler.wrapJobExecutionInUserTransaction**

如果想使用Quartz在执行一个job前使用UserTransaction，则应该设置该属性为true。job执行完、在JobDataMap改变之后事务会提交。默认值是false。 可以在你的job类中使用 @ExecuteInJTATransaction注解, 可以控制job是否使用事务。

**org.quartz.scheduler.skipUpdateCheck**

建议设置为“org.terracotta.quartz.skipUpdateCheck=true”不会在程序运行中还去检查quartz是否有版本更新。

**org.quartz.scheduler.batchTriggerAcquisitionMaxCount**

允许调度程序一次性触发的触发器数量。.默认值是1。值越大一次性触发的任务就可以越多，但是在集群环境下，不建议设置为很大值。如果值 > 1, 并且使用了 JDBC JobStore的话, **org.quartz.jobStore.acquireTriggersWithinLock**属性必须设置为true，以避免”弄脏”数据。

**org.quartz.scheduler.batchTriggerAcquisitionFireAheadTimeWindow**

允许触发器被获取并在其预定的触发时间之前触发的数量。默认值0。

**Quartz线程池的配置**

| **参数名** | **是否必须** | **类型** | **默认值** |
| --- | --- | --- | --- |
| org.quartz.threadPool.class | Y | string (class name) | null |
| org.quartz.threadPool.threadCount | Y | string | -1 |
| org.quartz.threadPool.threadPriority | N | int | Thread.NORM\_PRIORITY (5) |

**org.quartz.threadPool.class**

线程池的名字。可以使用后Quartz 的 “org.quartz.simpl.SimpleThreadPool”。

**org.quartz.threadPool.threadCount**

指定线程数量。一般1-100足以满足你的应用需求了。

**org.quartz.threadPool.threadPriority**

线程优先级，Thread.MIN\_PRIORITY (1) and Thread.MAX\_PRIORITY (10)之间，默认Thread.NORM\_PRIORITY (5)。

**监听器的配置**

全局触发器的监听器配置:

org.quartz.triggerListener.NAME.class = com.foo.MyListenerClass

org.quartz.triggerListener.NAME.propName = propValue

org.quartz.triggerListener.NAME.prop2Name = prop2Value

全局job的监听器配置:

org.quartz.jobListener.NAME.class = com.foo.MyListenerClass

org.quartz.jobListener.NAME.propName = propValue

org.quartz.jobListener.NAME.prop2Name = prop2Value

**jobStore的配置**

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

或者

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

或者

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

**数据源的配置**

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

从一个应用服务中获取数据源配置:

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

**集群配置**

#============================================================================

# 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