

-

整理自互联网，仅供学习交流。



2018-10-30

[公司名称]

[公司地址]

目录

[一、 123 1](#_Toc3539499)

[1.1 1 1](#_Toc3539500)

[1.1.1 3 1](#_Toc3539501)

# Quartz

## Quartz简介

Quartz是Java领域最著名的开源任务调度工具，是一个任务调度框架，通过触发器设置作业的定时运行规则，来执行定时任务。其中quartz集群通过故障切换和负载平衡的功能，能给调度器带来高可用性和伸缩性。

Quartz提供了极为广泛的特性如持久化任务，集群和分布式任务等。

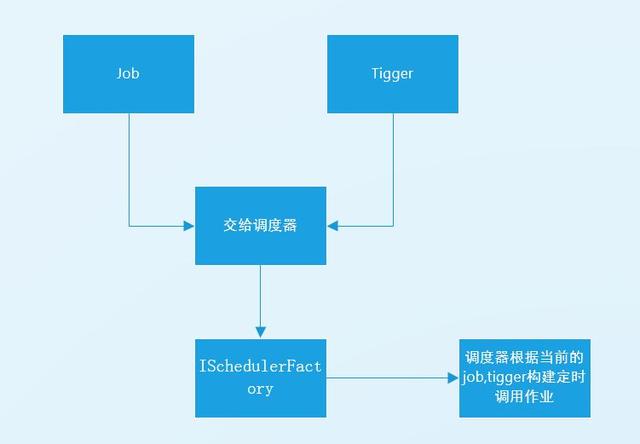
**其特点如下**：

* 完全由Java写成，方便集成(Spring)
* 伸缩性
* 负载均衡
* 高可用性

**典型的使用场景，**主要用来执行定时任务，例如：

* 定时发送信息
* 定时生成报表
* 自动更新静态数据
* 自动结账等等

## Quartz架构简介



**Quartz框架主要核心组件包括：**

**1.Scheduler任务调度**

是最核心的概念，需要把JobDetail和Trigger注册到scheduler中，才可以执行。

工厂模式，组装各个组件<JOB，Trigger> sched.scheduleJob(job, trigger);

**2.Job任务**

其实Job是接口，其中只有一个execute方法,我们只需要 implements 此接口，重写 execute(\*) 方法。

**3.Trigger触发器**

执行任务的规则；比如每天，每小时等。

一般情况使用SimpleTrigger，和CronTrigger，这些触发器实现了Trigger接口。或者 ScheduleBuilder 子类 SimpleScheduleBuilder和CronScheduleBuilder。

对于简单的时间来说，比如每天执行几次，使用SimpleTrigger。

对于复杂的时间表达式来说，比如每个月15日上午几点几分，使用CronTrigger以及CromExpression 类。

**4.JobDetail任务细节**

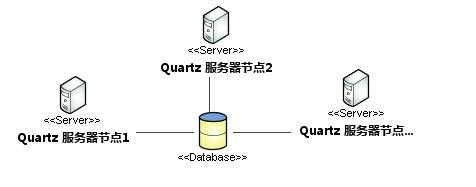
任务细节，Quartz执行Job时，需要新建个Job实例，但是不能直接操作Job类，所以通过JobDetail来获取Job的名称、描述信息。

**调度器作为作业的总指挥，触发器作为作业的操作者，作业为应用的功能模块。**

## Quartz集群部署实践

Quartz与Spring结合使用，Spring通过提供org.springframework.scheduling.quartz下的封装类对Quartz支持。

**1.Quartz集群部署：**



Quartz集群中的每个节点是一个独立的Quartz应用，它又管理着其他的节点。该集群需要分别对每个节点分别启动或停止，不像应用服务器的集群，独立的Quartz节点并不与另一个节点或是管理节点通信。Quartz应用是通过数据库表来感知到另一应用。只有使用持久的JobStore才能完成Quqrtz集群。

**基于Spring的集群配置：**

<!-- 调度工厂 -->

<bean id="quartzScheduler"

class="org.springframework.scheduling.quartz.SchedulerFactoryBean">

<property name="dataSource" ref="dataSource" />

<property name="quartzProperties">

<props>

<prop key="org.quartz.scheduler.instanceName">CRMscheduler</prop>

<prop key="org.quartz.scheduler.instanceId">AUTO</prop>

<!-- 线程池配置 -->

<prop key="org.quartz.threadPool.class">org.quartz.simpl.SimpleThreadPool</prop>

<prop key="org.quartz.threadPool.threadCount">20</prop>

<prop key="org.quartz.threadPool.threadPriority">5</prop>

<!-- JobStore 配置 -->

<prop key="org.quartz.jobStore.class">org.quartz.impl.jdbcjobstore.JobStoreTX</prop>

<!-- 集群配置 -->

<prop key="org.quartz.jobStore.isClustered">true</prop>

<prop key="org.quartz.jobStore.clusterCheckinInterval">15000</prop>

<prop key="org.quartz.jobStore.maxMisfiresToHandleAtATime">1</prop>

<prop key="org.quartz.jobStore.misfireThreshold">120000</prop>

<prop key="org.quartz.jobStore.tablePrefix">QRTZ\_</prop>

</props>

</property>

<property name="schedulerName" value="CRMscheduler" />

<!--必须的，QuartzScheduler 延时启动，应用启动完后 QuartzScheduler 再启动 -->

<property name="startupDelay" value="30" />

<property name="applicationContextSchedulerContextKey" value="applicationContextKey" />

<!--可选，QuartzScheduler 启动时更新己存在的Job，这样就不用每次修改targetObject后删除qrtz\_job\_details表对应记录了 -->

<property name="overwriteExistingJobs" value="true" />

<!-- 设置自动启动 -->

<property name="autoStartup" value="true" />

<!-- 注册触发器 -->

<property name="triggers">

<list>

<ref bean="userSyncScannerTrigger" />

......

</list>

</property>

<!-- 注册jobDetail -->

<property name="jobDetails">

<list>

</list>

</property>

<property name="schedulerListeners">

<list>

<ref bean="quartzExceptionSchedulerListener" />

</list>

</property>

</bean>

org.quartz.jobStore.class属性为JobStoreTX，将任务持久化到数据中。因为集群中节点依赖于数据库来传播Scheduler实例的状态，你只能在使用JDBC JobStore时应用Quartz集群。

org.quartz.jobStore.isClustered属性为true，通知Scheduler实例要它参与到一个集群当中。

org.quartz.jobStore.clusterCheckinInterval属性定义了Scheduler实例检入到数据库中的频率(单位：毫秒)。Scheduler检查是否其他的实例到了它们应当检入的时候未检入；这能指出一个失败的Scheduler实例，且当前 Scheduler会以此来接管任何执行失败并可恢复的Job。通过检入操作，Scheduler 也会更新自身的状态记录。clusterChedkinInterval越小，Scheduler节点检查失败的Scheduler实例就越频繁。默认值是 15000 (即15 秒)。

其余参数在后文将会详细介绍。

## Quartz监控

Quartz实例的监控、操作以及动态部署Trigger.

**1.Triggers监控：**



**2.JobDetails监控：**



## Quartz集群原理分析

**1. Quartz集群数据库表**

Quartz的集群部署方案在架构上是分布式的，没有负责集中管理的节点，而是利用数据库锁的方式来实现集群环境下进行并发控制。BTW，分布式部署时需要保证各个节点的系统时间一致。

Quartz数据库核心表如下：

* Table NameDescriptionQRTZ\_CALENDARS存储Quartz的Calendar信息
* QRTZ\_CRON\_TRIGGERS存储CronTrigger，包括Cron表达式和时区信息
* QRTZ\_FIRED\_TRIGGERS存储与已触发的Trigger相关的状态信息，以及相联Job的执行信息
* QRTZ\_PAUSED\_TRIGGER\_GRPS存储已暂停的Trigger组的信息QRTZ\_SCHEDULER\_STATE

**2. Quartz线程模型**

**在Quartz中有两类线程:**

* Scheduler调度线程
* 任务执行线程

**任务执行线程**：Quartz不会在主线程(QuartzSchedulerThread)中处理用户的Job。

Quartz把线程管理的职责委托给ThreadPool，一般的设置使用SimpleThreadPool。SimpleThreadPool创建了一定数量的WorkerThread实例来使得Job能够在线程中进行处理。WorkerThread是定义在SimpleThreadPool类中的内部类，它实质上就是一个线程。例如，CRM中配置如下：

<!-- 线程池配置 -->

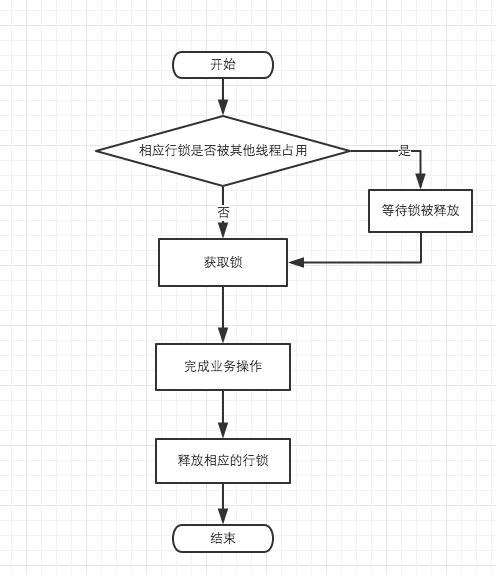
<prop key="org.quartz.threadPool.class">org.quartz.simpl.SimpleThreadPool</prop>

<prop key="org.quartz.threadPool.threadCount">20</prop>

<prop key="org.quartz.threadPool.threadPriority">5</prop>

**QuartzSchedulerThread调度主线程**：QuartzScheduler被创建时创建一个QuartzSchedulerThread实例。

**3.Quartz集群基于数据库锁的同步操作流程如下图所示**：



一个调度器实例在执行涉及到分布式问题的数据库操作前，首先要获取QUARTZ\_LOCKS表中对应的行级锁，获取锁后即可执行其他表中的数据库操作，随着操作事务的提交，行级锁被释放，供其他调度实例获取。集群中的每一个调度器实例都遵循这样一种严格的操作规程。

总结一下Quartz集群同步机制：每当要进行与某种业务相关的数据库操作时，先去QRTZ\_LOCKS表中查询操作相关的业务对象所需要的锁，在select语句之后加for update来实现。例如，TRIGGER\_ACCESS表示对任务触发器相关的信息进行修改、删除操作时所需要获得的锁。这时，执行查询这个表数据的SQL形如：

select \* from QRTZ\_LOCKS t where t.lock\_name='TRIGGER\_ACCESS' for update

当一个线程使用上述的SQL对表中的数据执行查询操作时，若查询结果中包含相关的行，数据库就对该行进行ROW LOCK；若此时，另外一个线程使用相同的SQL对表的数据进行查询，由于查询出的数据行已经被数据库锁住了，此时这个线程就只能等待，直到拥有该行锁的线程完成了相关的业务操作，执行了commit动作后，数据库才会释放了相关行的锁，这个线程才能继续执行。

通过这样的机制，在集群环境下，结合悲观锁的机制就可以防止一个线程对数据库数据的操作的结果被另外一个线程所覆盖，从而可以避免一些难以觉察的错误发生。当然，达到这种效果的前提是需要把Connection设置为手动提交，即autoCommit为false。