目录

[定时任务 2](#_Toc27121)

[什么是定时任务 2](#_Toc26073)

[定时任务应用场景 2](#_Toc6822)

[定时任务技术实现 3](#_Toc19493)

[单机 3](#_Toc11175)

[分布/集群 3](#_Toc17937)

[Quartz、elastic-job和xxl-job 框架功能对比 4](#_Toc19767)

[Quartz 分布式调度框架 5](#_Toc28936)

[Quartz简介 5](#_Toc25722)

[Quartz 知识体系结构 6](#_Toc3422)

[Quartz 之 任务 6](#_Toc26931)

[Quartz 之 触发器 7](#_Toc29089)

[Quartz 之 调度器 9](#_Toc26177)

[Quartz 持久化 9](#_Toc31124)

[Quartz定时任务存储方式RAMJobStore和JDBCJobStore 9](#_Toc23349)

[Quartz 设置JDBCJobStore 10](#_Toc22005)

[Quartz 集群化 13](#_Toc28542)

[Quartz 设置集群 14](#_Toc2937)

[Quartz 监听器 14](#_Toc11253)

[SchedulerListener 任务调度监听器 14](#_Toc8870)

[TriggerListener任务触发监听器 17](#_Toc21973)

[JobListener任务执行监听器 18](#_Toc22383)

[Quartz 线程池 18](#_Toc7777)

[Quartz 线程池配置属性 19](#_Toc20157)

[Quartz 线程池默认实现 19](#_Toc13507)

[Quartz 任务并行 20](#_Toc18771)

[任务并行与串行区别： 20](#_Toc29679)

[@DisallowConcurrentExecution标签 21](#_Toc18086)

[SpringBoot 集成Quartz集群 22](#_Toc29836)

[Quartz 依赖Jar包 22](#_Toc24484)

[Quartz 配置文件quartz.properties 22](#_Toc19249)

[注册 Quartz 任务工厂 23](#_Toc8316)

[注册 Quartz 调度工厂 24](#_Toc15733)

[设置 Quartz 数据连接池 25](#_Toc8680)

[Quartz 服务接口定义和默认实现 29](#_Toc19626)

[Quartz 请求参数Dto定义 34](#_Toc25992)

[Quartz 之控制层 35](#_Toc11060)

[Quartz UI管理界面 38](#_Toc22091)

# 定时任务

## **什么是定时任务**

定时任务是指调度程序在指定的时间或周期触发执行的任务

## **定时任务应用场景**

使用场景:发送邮件、统计、状态修改、消息推送、活动开启

## 定时任务技术实现

### **单机**

* Java 自带的 java.util.Timer类，这个类允许你调度一个 java.util.TimerTask 任务。使用这种方式可以让你的程序按照某一个频度执行，但不能在指定时间运行。使用较少。
* Spring3.0 以后自主开发的定时任务工具 spring task ，使用简单，支持线程池，可以高效处理许多不同的定时任务，除 spring 相关的包外不需要额外的包，支持注解和配置文件两种形式。 不能处理过于复杂的任务。
* 专业的定时框架 quartz ，功能强大，可以让你的程序在指定时间执行，也可以按照某一个频度执行，支持数据库、监听器、插件、集群。

### 分布/集群

* Quartz：Java事实上的定时任务标准。但Quartz关注点在于定时任务而非数据，并无一套根据数据处理而定制化的流程。虽然Quartz可以基于数据库实现作业的高可用，缺少分布式并行调度的功能
* TBSchedule：阿里早期开源的分布式任务调度系统。代码略陈旧，使用timer而非线程池执行任务调度。众所周知，timer在处理异常状况时是有缺陷的。而且TBSchedule作业类型较为单一，只能是获取/处理数据一种模式。还有就是文档缺失比较严重
* elastic-job：当当开发的弹性分布式任务调度系统，功能丰富强大，采用zookeeper实现分布式协调，实现任务高可用以及分片。版本已经停止维护更新
* Saturn：是唯品会自主研发的分布式的定时任务的调度平台，基于当当的elastic-job 版本1开发，并且可以很好的部署到docker容器上。
* xxl-job: 是大众点评员工徐雪里于2015年发布的分布式任务调度平台，是一个轻量级分布式任务调度框架，其核心设计目标是开发迅速、学习简单、轻量级、易扩展。上手门槛高

## Quartz、elastic-job和xxl-job 框架功能对比

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Quartz** | **Elastic-job** | **Xxl-job** |
| 相关依赖 | 数据库 | 数据库、zookeeper 注册中心 | 数据库 |
| HA | 多节点部署，通过竞争数据库锁来保证只有一个节点执行任务 | 通过zookeeper的注册与发现，可以动态的添加服务器。 支持水平扩容 | 集群部署 |
| 任务分片 | 不支持 | 支持 | 支持 |
| 文档 | 完善 | 完善 | 完善 |
| 管理界面 | 支持 | 支持 | 支持 |
| 难易程度 | 简单 | 较复杂 | 较复杂 |
| 开源公司/个人 | OpenSymphony | 当当网 | 个人 |
| 高级功能 | 不支持 | 弹性扩容，多种作业模式，失效转移，运行状态收集，多线程处理数据，幂等性，容错处理，spring命名空间支持 | 弹性扩容，分片广播，故障转移，Rolling实时日志，GLUE（支持在线编辑代码，免发布）,任务进度监控，任务依赖，数据加密，邮件报警，运行报表，国际化 |
| 缺点 | 不适用于分布式场景 | 引入zookeeper , 增加系统复杂度, 上手成本较高 | 调度中心通过获取 DB锁来保证集群中执行任务的唯一性， 会造成数据库性能下降。 |

综合上述开源任务调度框架对比并结合项目开发的实际功能需求，推荐使用Quartz分布式调度框架。

# Quartz 分布式调度框架

## Quartz简介

Quartz是OpenSymphony开源组织在Job scheduling领域又一个开源项目，完全由Java开发，可以用来执行定时任务，类似于java.util.Timer。但是相较于Timer， Quartz增加了很多功能：

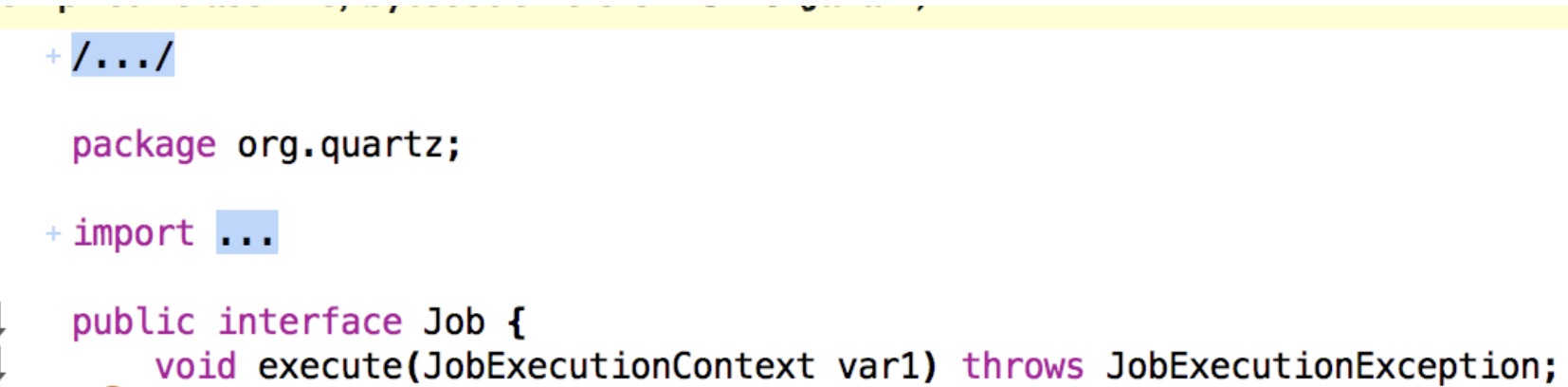
* 持久性作业 - 就是保持调度定时的状态;
* 作业管理 - 对调度作业进行有效的管理;

## Quartz 知识体系结构

Quartz的核心主要包括三部分：任务（Job）、触发器（Trigger）和调度器（Scheduler）

### Quartz 之 任务

**Job**:Quartz中的一个接口，接口下只有execute方法，在这个方法中编写具体业务逻辑。



**JobDetail**:用来绑定Job，为Job实例提供许多属性：

* name
* group
* jobClass
* jobDataMap

JobDetail绑定指定的Job，每次Scheduler调度执行一个Job的时候，首先会拿到对应的Job，然后创建该Job实例，再去执行Job中的execute()的内容，任务执行结束后，关联的Job对象实例会被释放，且会被JVM GC清除。

**JobExecutionContext:**

JobExecutionContext中包含了Quartz运行时的环境以及Job本身的详细数据信息。

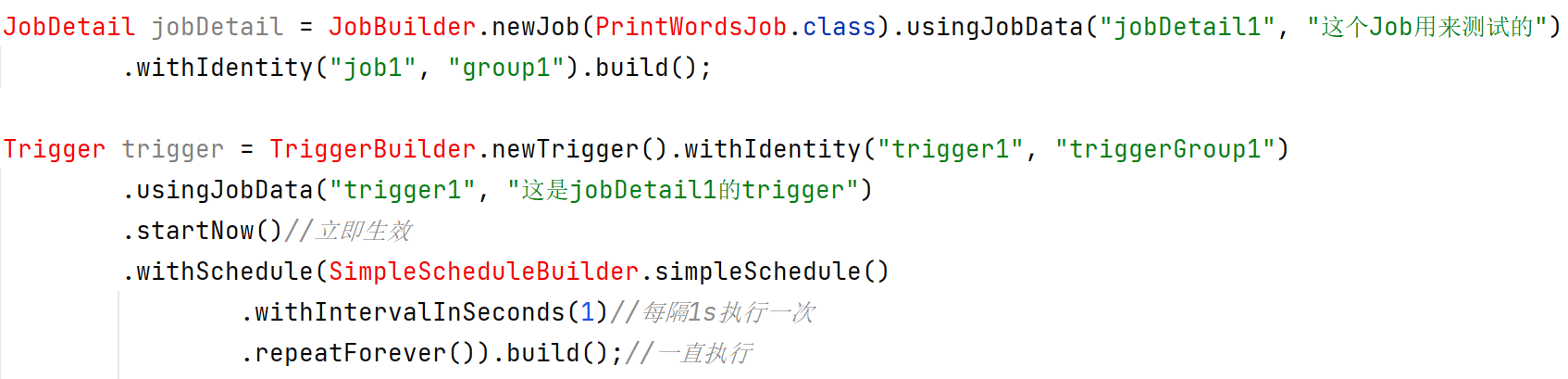
当Schedule调度执行一个Job的时候，就会将JobExecutionContext传递给该Job的execute()中，Job就可以通过JobExecutionContext对象获取信息**。**



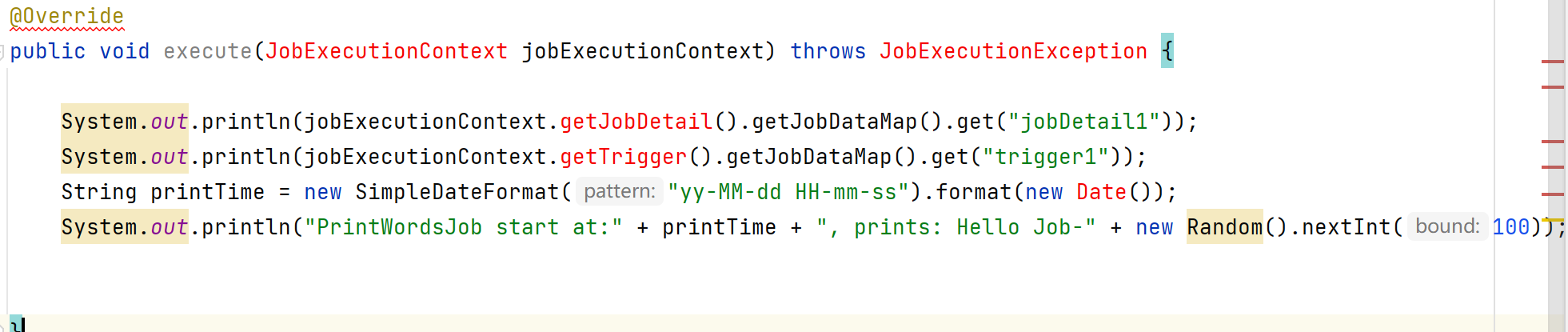
**JobDataMap:**

JobDataMap实现了JDK的Map接口，可以以Key-Value的形式存储数据。

JobDetail、Trigger都可以使用JobDataMap来设置一些参数或信息。



Job执行execute()方法的时候，JobExecutionContext可以获取到JobDataMap中的信息：



### Quartz 之 触发器

**Trigger:**

Trigger是Quartz的触发器，会去通知Scheduler何时去执行对应Job。

new Trigger().startAt():表示触发器首次被触发的时间;  
new Trigger().endAt():表示触发器结束触发的时间;

**SimpleTrigger**

SimpleTrigger可以实现在一个指定时间段内执行一次作业任务或一个时间段内多次执行作业任务。

Date startDate = new Date();  
startDate.setTime(startDate.getTime() + 5000);  
  
Date endDate = new Date();  
endDate.setTime(startDate.getTime() + 5000);  
  
Trigger trigger = TriggerBuilder.*newTrigger*().withIdentity("trigger1", "triggerGroup1")  
 .usingJobData("trigger1", "这是jobDetail1的trigger")  
 .startNow()*//立即生效* .startAt(startDate)  
 .endAt(endDate)  
 .withSchedule(SimpleScheduleBuilder.*simpleSchedule*()  
 .withIntervalInSeconds(1)*//每隔1s执行一次* .repeatForever()).build();*//一直执行*

**CronTrigger：**

CronTrigger功能非常强大，是基于日历的作业调度，而SimpleTrigger是精准指定间隔，所以相比SimpleTrigger，CroTrigger更加常用。

CroTrigger是基于Cron表达式的，先了解下Cron表达式。

Cron语法格式：由7个子表达式组成字符串

[秒] [分] [小时] [日] [月] [周] [年]

常见Cron 表达式

|  |  |
| --- | --- |
| 表达式 | 功能说明 |
| 0 0 12 \* \* ？ | 每天12点运行 |
| 0 15 10 ? \* \* | 每天10:15运行 |
| 0 15 10 ? \* \* 2021 | 在2021年的每天10:15运行 |
| 0 15 10 L \* ? | 每月最后一天10：15运行 |

在线生成Cron表达式的工具：http://cron.qqe2.com/

### Quartz 之 调度器

**Scheduler：**

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。

## Quartz 持久化

### Quartz定时任务存储方式RAMJobStore和JDBCJobStore

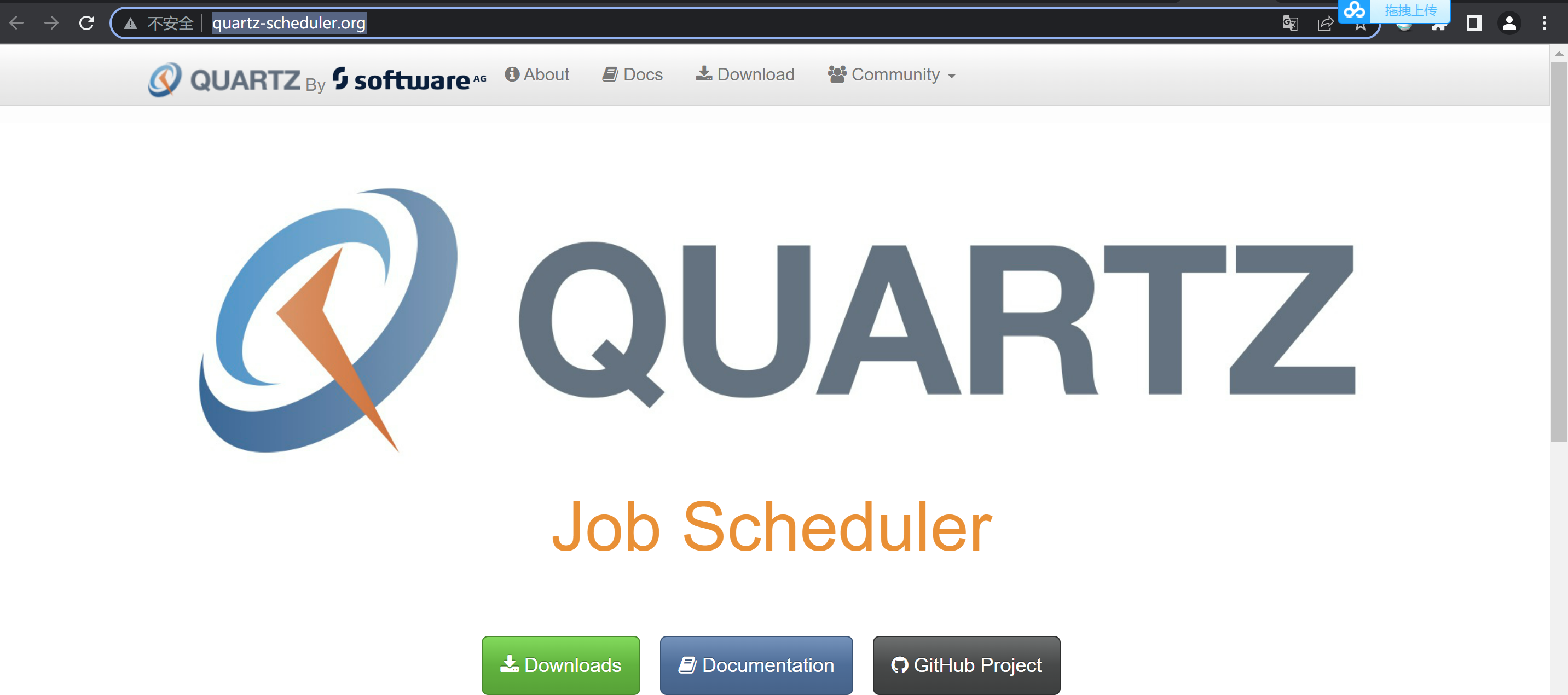
RAMJobStore（默认） 不要外部数据库，配置容易，运行速度快 因为调度程序信息是存储在被分配给JVM的内存里面，所以，当应用程序停止运行时，所有调度信息将被丢失。另外因为存储到JVM内存里面，所以可以存储多少个Job和Trigger将会受到限制 JDBCJobStore 支持集群，因为所有的任务信息都会保存到数据库中，可以控制事物，还有就是如果应用服务器关闭或者重启，任务信息都不会丢失，并且可以恢复因服务器关闭或者重启而导致执行失败的任务 运行速度的快慢取决与连接数据库的快慢

### Quartz 设置JDBCJobStore

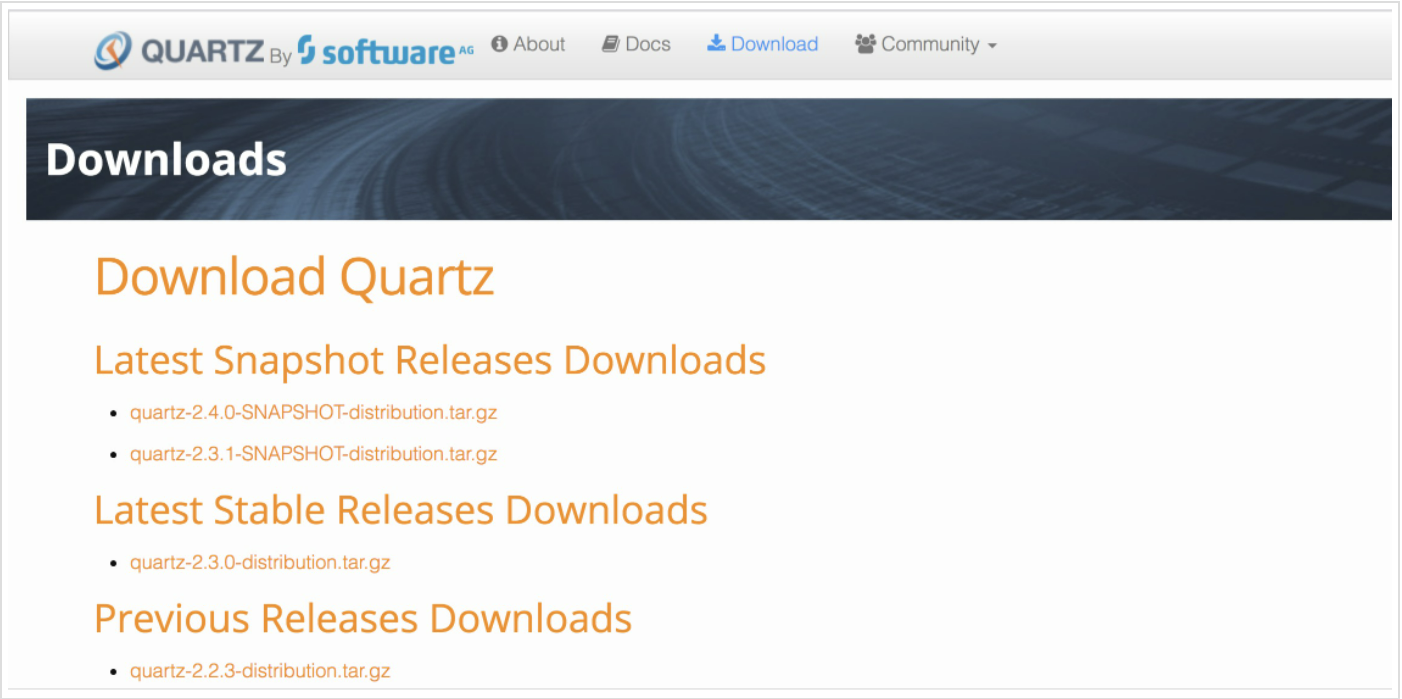
集成Quartz 数据库持久化总体分为两步

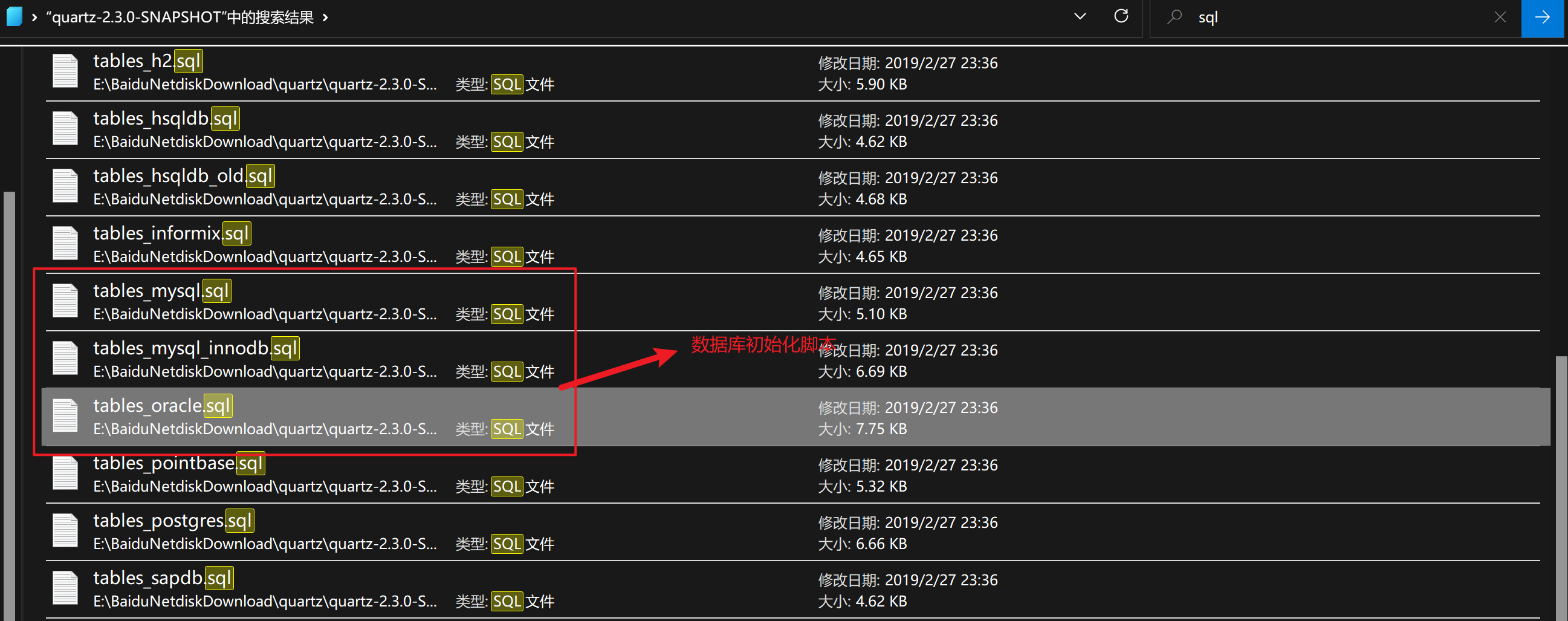
第一步: 创建Quartz 数据库表

Quartz数据库表结构官网已经提供，我们可以直接访问Quartz对应的官方网站，找到对应的版本，然后将其下载！ Quartz官网地址：<http://www.quartz-scheduler.org/>



我选择的是quartz-2.3.0-distribution.tar.gz，下载完成之后将其解压，在文件中搜索sql，在里面选择适合当前环境的数据库脚本文件，然后将其初始化到数据库中即可！





**Quartz表结构说明：**

| 表名 | 描述 |
| --- | --- |
| QRTZ\_BLOG\_TRIGGERS | Trigger作为Blob类型存储 |
| QRTZ\_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\_JOB\_DETAILS | 存储每一个已配置的Job的详细信息 |
| QRTZ\_LOCKS | 存储程序的悲观锁的信息 |
| QRTZ\_PAUSED\_TRIGGER\_GRPS | 存储已暂停的Trigger组的信息 |
| QRTZ\_SCHEDULER\_STATE | 存储少量的有关Scheduler的状态信息，和别的Scheduler实例 |
| QRTZ\_SIMPLE\_TRIGGERS | 存储简单的Trigger，包括重复次数、间隔、以及已触的次数 |
| QRTZ\_SIMPROP\_TRIGGERS | 存储CalendarIntervalTrigger和DailyTimeIntervalTrigger两种类型的触发器 |
| QRTZ\_TRIGGERS | 存储已配置的Trigger的信息 |

第二步：在application.properties 配置文件中指定JDBCJobStore属性配置。

# 连接池配置

org.quartz.threadPool.class = org.quartz.simpl.SimpleThreadPool

org.quartz.threadPool.threadCount = 10

org.quartz.threadPool.threadPriority = 5

org.quartz.threadPool.threadsInheritContextClassLoaderOfInitializingThread = true

# Using JobStoreTX

## Be sure to run the appropriate script(under docs/dbTables) first to create tables

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

# Configuring JDBCJobStore with the Table Prefix

org.quartz.jobStore.tablePrefix = QRTZ\_

# Using DriverDelegate

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

# Using datasource

org.quartz.jobStore.dataSource = qzDS

# Define the datasource to use

org.quartz.dataSource.qzDS.driver = com.mysql.jdbc.Driver

org.quartz.dataSource.qzDS.URL = jdbc:mysql://localhost:3306/quartz

org.quartz.dataSource.qzDS.user = root

org.quartz.dataSource.qzDS.password = root

org.quartz.dataSource.qzDS.maxConnections = 30

## Quartz 集群化

单个Quartz实例能给予项目应用很好的Job调度能力，但它不能满足典型的企业需求，如可伸缩性、高可靠性满足。假如你需要故障转移的能力并能运行日益增多的 Job，Quartz集群势必成为你应用的一部分了。使用 Quartz 的集群能力可以更好的支持你的业务需求，并且即使是其中一台机器在最糟的时间崩溃了也能确保所有的 Job 得到执行。

一个 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。意思是你必须对每个节点分别启动或停止。不像许多应用服务器的集群，独立的 Quartz 节点并不与另一其的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的

### Quartz 设置集群

在application.properties 配置文件中添加指定Quartz集群属性配置。

# 集群实例配置

org.quartz.scheduler.instanceName=集群名称

org.quartz.scheduler.instanceId=集群节点

org.quartz.scheduler.skipUpdateCheck=true是否检查节点(默认为:false)

#使用 JobStoreTX 配置是否集群

org.quartz.jobStore.isClustered = true 是否集群

## Quartz 监听器

Quartz的监听器用于当任务调度执行时，你所关注事件发生时，能够及时获取这一事件的通知。类似于任务执行过程中的邮件，短信的提醒。

Quartz监听器主要有JobListener、TriggerListener、SchedulerListener三种，顾名思义，分别表示任务，触发器，调度器对应的监听器。

### SchedulerListener 任务调度监听器

public interface SchedulerListener {

/\*\*

\* 用于部署Job Detail时调用

\*/

void jobScheduled(Trigger trigger);

/\*\*

\* 用于卸载JobDetail时调用

\*/

void jobUnscheduled(TriggerKey triggerKey);

/\*\*

\* 当一个Trigger来到了再也不会触发的状态时调用这个方法。除非这个Job已设置成了持久性，否则它就会中scheduler中移除

\*/

void triggerFinalized(Trigger trigger);

/\*\*

\* scheduler调用这个方法时发生在一个Trigger被暂停时调用。

\*/

void triggerPaused(TriggerKey triggerKey);

/\*\*

\* scheduler调用这个方法时发生在一个Trigger组被暂停时调用。

\*/

void triggersPaused(String triggerGroup);

/\*\*

\* scheduler调用这个方法发生在为一个Trigger从暂停中恢复时

\*/

void triggerResumed(TriggerKey triggerKey);

/\*\*

\* scheduler调用这个方法发生在为一个Trigger组从暂停中恢复时

\*/

void triggersResumed(String triggerGroup);

/\*\*

\* 添加任务时被调用

\*/

void jobAdded(JobDetail jobDetail);

/\*\*

\* 删除任务时被调用

\*/

void jobDeleted(JobKey jobKey);

/\*\*

\* 暂停任务时被调

\*/

void jobPaused(JobKey jobKey);

/\*\*

\* 暂停任务组时被调用

\*/

void jobsPaused(String jobGroup);

/\*\*

\* 恢复任务时被调用

\*/

void jobResumed(JobKey jobKey);

/\*\*

\* 恢复任务组时被调用

\*/

void jobsResumed(String jobGroup);

/\*\*

\* 在scheduler的政策运行期间发生严重错误时被调用

\*/

void schedulerError(String msg, SchedulerException cause);

/\*\*

\* 在scheduler处于挂载状态时被调用

\*/

void schedulerInStandbyMode();

/\*\*

\* 在scheduler开启时被调用

\*/

void schedulerStarted();

/\*\*

\* 当调度程序启动时调用该方法

\*/

void schedulerStarting();

/\*\*

\* 当scheduler关闭时调用该方法

\*/

void schedulerShutdown();

/\*\*

\* 当调度程序关闭时调用

\*/

void schedulerShuttingdown();

/\*\*

\* 当scheduler中的数据被清除时，调用该方法

\*/

void schedulingDataCleared();

}

### TriggerListener任务触发监听器

public interface TriggerListener {

/\*\*

\* 用于触发器的名称

\*/

String getName();

/\*\*

\* 当与监听器相关联的Trigger被触发，Job上的execute()方法将被执行时，Scheduler就调用该方法【job执行前】

\*/

void triggerFired(Trigger trigger, JobExecutionContext context);

/\*\*

\* 在Trigger触发后，Job将要被执行由Scheduler调用这个方法。Trigger Listener给了一个选择去否决Job的执行。假如这个方法返回true，这个Job将不会为此次Trigger而得到触发【job执行前】

\*/

boolean vetoJobExecution(Trigger trigger, JobExecutionContext context);

/\*\*

\* scheduler调用这个方法是在Trigger错过触发时，你应该关注此方法中持续时间长的逻辑；在出现许多错过触发的Trigger时，长逻辑会导致骨牌效应，你应当保持这个方法尽量的少使用

\*/

void triggerMisfired(Trigger trigger);

/\*\*

\* Trigger被触发并完成了Job的执行，scheduler调用这个方法

\*/

void triggerComplete(Trigger trigger, JobExecutionContext context,

CompletedExecutionInstruction triggerInstructionCode);

}

### JobListener任务执行监听器

public interface JobListener {

/\*\*

\* 用于获取该JobListener的名称

\*/

String getName();

/\*\*

\* scheduler在JobDetail将要被执行时调用这个方法【JobDetail执行前】

\* @see #jobExecutionVetoed(JobExecutionContext)

\*/

void jobToBeExecuted(JobExecutionContext context);

/\*\*

\* scheduler在JobDetail即将被执行，但又被TriggerListener否决时会调用该方法

\* @see #jobToBeExecuted(JobExecutionContext)

\*/

void jobExecutionVetoed(JobExecutionContext context);

/\*\*

\* scheduler在JobDetail被执行后调用这个方法

\*/

void jobWasExecuted(JobExecutionContext context,JobExecutionException jobException);

}

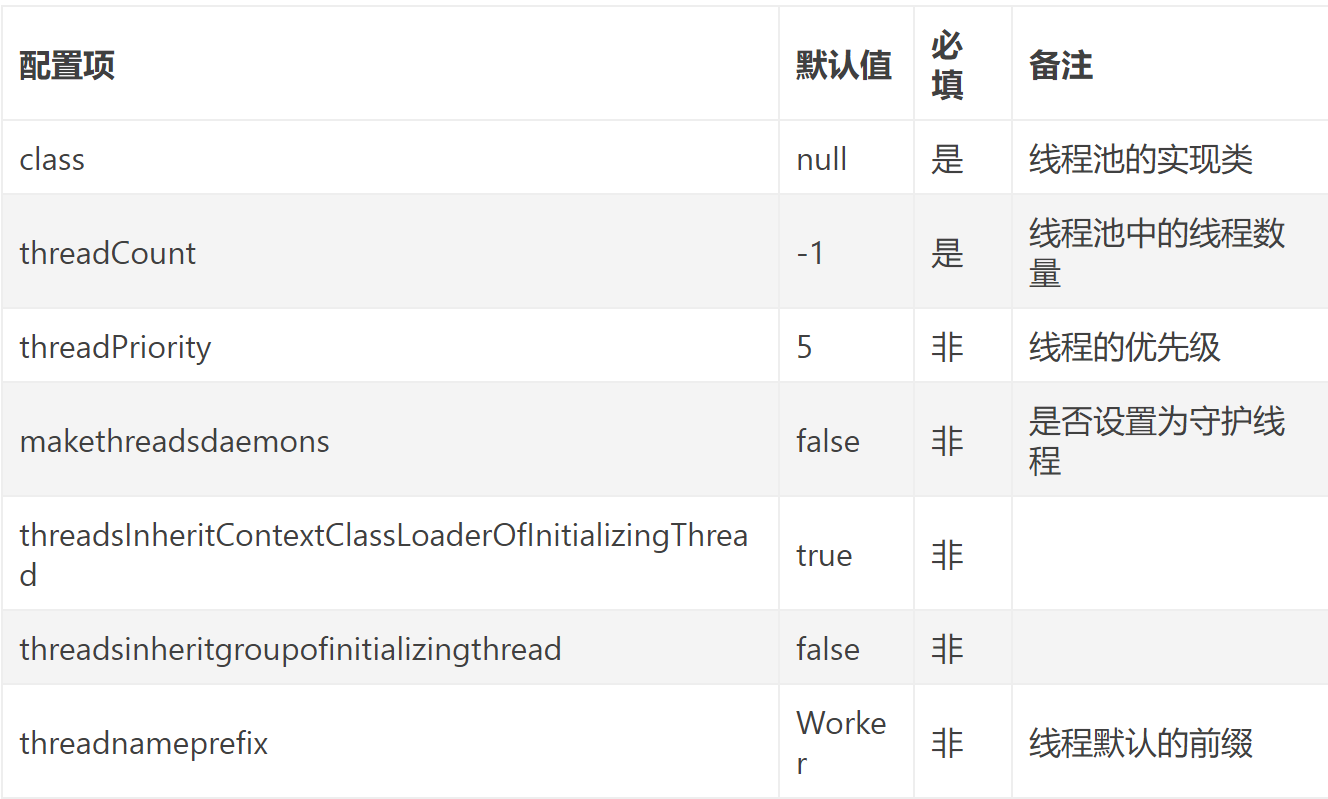
## Quartz 线程池

Quartz 任务调度框架使用线程池主要解决：

* 降低资源消耗。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。
* 提高响应速度。当任务到达时，任务可以不需要的等到线程创建就能立即执行。

### Quartz 线程池配置属性

prefix = org.quartz.threadPool



### Quartz 线程池默认实现

Quartz 线程池接口定义：ThreadPool

package org.quartz.spi;

import org.quartz.SchedulerConfigException;

public interface ThreadPool {

// 运行指定线程

boolean runInThread(Runnable var1);

// 可用线程数量

int blockForAvailableThreads();

// 初始化

void initialize() throws SchedulerConfigException;

// 关闭

void shutdown(boolean var1);

// 线程池大小

int getPoolSize();

// 设置线程Id

void setInstanceId(String var1);

// 设置线程池名称

void setInstanceName(String var1);

}

默认实现：org.quartz.simpl.SimpleThreadPool

## Quartz 任务并行

### 任务并行与串行区别：

* 并行是指：一个定时任务，当执行时间到了的时候，立刻执行此任务，不管当前这个任务是否在执行中；
* 串行是指：一个定时任务，当执行时间到了的时候，需要等待当前任务执行完毕，再去执行下一个任务。

Quartz定时任务默认都是[并发](https://so.csdn.net/so/search?q=%E5%B9%B6%E5%8F%91&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/bbj12345678/article/details/_blank)执行的，不会等待上一次任务执行完毕，只要间隔时间到就会执行, 如果定时任执行太长，会长时间占用资源，导致其它任务堵塞。

### @DisallowConcurrentExecution标签

@DisallowConcurrentExecution 禁止并发执行多个相同定义的JobDetail, 这个注解是加在Job类上的, 但意思并不是不能同时执行多个Job, 而是不能并发执行同一个Job Definition(由JobDetail定义), 但是可以同时执行多个不同的JobDetail。

推荐用法：

* @PersistJobDataAfterExecution标签, 也是仅支持加在Job上,表示当正常执行完Job后, JobDataMap中的数据应该被改动, 以被下一次调用时用。当使用@PersistJobDataAfterExecution 注解时, 为了避免并发时, 存储数据造成混乱, 强烈建议把@DisallowConcurrentExecution注解也加上。

@DisallowConcurrentExecution  
@PersistJobDataAfterExecution  
public class SimpleJob implements Job{  
 @Override  
 public void execute(JobExecutionContext jobExecutionContext) throws JobExecutionException {  
 }  
}

## SpringBoot 集成Quartz集群

### Quartz 依赖Jar包

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-quartz</artifactId>

</dependency>

### Quartz 配置文件quartz.properties

#调度配置

#调度器实例名称

org.quartz.scheduler.instanceName=SsmScheduler

#调度器实例编号自动生成

org.quartz.scheduler.instanceId=AUTO

#是否在Quartz执行一个job前使用UserTransaction

org.quartz.scheduler.wrapJobExecutionInUserTransaction=false

#线程池配置

#线程池的实现类

org.quartz.threadPool.class=org.quartz.simpl.SimpleThreadPool

#线程池中的线程数量

org.quartz.threadPool.threadCount=10

#线程优先级

org.quartz.threadPool.threadPriority=5

#配置是否启动自动加载数据库内的定时任务，默认true

org.quartz.threadPool.threadsInheritContextClassLoaderOfInitializingThread=true

#是否设置为守护线程，设置后任务将不会执行

#org.quartz.threadPool.makeThreadsDaemons=true

#持久化方式配置

#JobDataMaps是否都为String类型

org.quartz.jobStore.useProperties=true

#数据表的前缀，默认QRTZ\_

org.quartz.jobStore.tablePrefix=QRTZ\_

#最大能忍受的触发超时时间

org.quartz.jobStore.misfireThreshold=60000

#是否以集群方式运行

org.quartz.jobStore.isClustered=true

#调度实例失效的检查时间间隔，单位毫秒

org.quartz.jobStore.clusterCheckinInterval=2000

#数据保存方式为数据库持久化

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

#数据库代理类，一般org.quartz.impl.jdbcjobstore.StdJDBCDelegate可以满足大部分数据库

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

#数据库别名 随便取

org.quartz.jobStore.dataSource=qzDS

#数据库连接池，将其设置为druid

org.quartz.dataSource.qzDS.connectionProvider.class=com.example.cluster.quartz.config.DruidConnectionProvider

#数据库引擎

org.quartz.dataSource.qzDS.driver=com.mysql.cj.jdbc.Driver

#数据库连接

org.quartz.dataSource.qzDS.URL=jdbc:mysql://127.0.0.1:3306/test-quartz?serverTimezone=UTC&useUnicode=true&characterEncoding=utf-8&useSSL=true

#数据库用户

org.quartz.dataSource.qzDS.user=root

#数据库密码

org.quartz.dataSource.qzDS.password=123456

#允许最大连接

org.quartz.dataSource.qzDS.maxConnection=5

#验证查询sql,可以不设置

org.quartz.dataSource.qzDS.validationQuery=select 0 from dual

### 注册 Quartz 任务工厂

@Component

public class QuartzJobFactory extends AdaptableJobFactory {

@Autowired

private AutowireCapableBeanFactory capableBeanFactory;

@Override

protected Object createJobInstance(TriggerFiredBundle bundle) throws Exception {

//调用父类的方法

Object jobInstance = super.createJobInstance(bundle);

//进行注入

capableBeanFactory.autowireBean(jobInstance);

return jobInstance;

}}

### 注册 Quartz 调度工厂

@Configuration

public class QuartzConfig {

@Autowired

private QuartzJobFactory jobFactory;

@Bean

public SchedulerFactoryBean schedulerFactoryBean() throws IOException {

//获取配置属性

PropertiesFactoryBean propertiesFactoryBean = new PropertiesFactoryBean();

propertiesFactoryBean.setLocation(new ClassPathResource("/quartz.properties"));

//在quartz.properties中的属性被读取并注入后再初始化对象

propertiesFactoryBean.afterPropertiesSet();

//创建SchedulerFactoryBean

SchedulerFactoryBean factory = new SchedulerFactoryBean();

factory.setQuartzProperties(propertiesFactoryBean.getObject());

factory.setJobFactory(jobFactory);//支持在JOB实例中注入其他的业务对象

factory.setApplicationContextSchedulerContextKey("applicationContextKey");

factory.setWaitForJobsToCompleteOnShutdown(true);//这样当spring关闭时，会等待所有已经启动的quartz job结束后spring才能完全shutdown。

factory.setOverwriteExistingJobs(false);//是否覆盖己存在的Job

factory.setStartupDelay(10);//QuartzScheduler 延时启动，应用启动完后 QuartzScheduler 再启动

return factory;

}

/\*\*

\* 通过SchedulerFactoryBean获取Scheduler的实例

\* @return

\* @throws IOException

\* @throws SchedulerException

\*/

@Bean(name = "scheduler")

public Scheduler scheduler() throws IOException, SchedulerException {

Scheduler scheduler = schedulerFactoryBean().getScheduler();

return scheduler;

}}

### **设置 Quartz 数据连接池**

默认 Quartz 的数据连接池是 c3p0，由于性能不太稳定，不推荐使用，因此我们将其改成driud数据连接池，配置如下：

public class DruidConnectionProvider implements ConnectionProvider {

/\*\*

\* 常量配置，与quartz.properties文件的key保持一致(去掉前缀)，同时提供set方法，Quartz框架自动注入值。

\* @return

\* @throws SQLException

\*/

//JDBC驱动

public String driver;

//JDBC连接串

public String URL;

//数据库用户名

public String user;

//数据库用户密码

public String password;

//数据库最大连接数

public int maxConnection;

//数据库SQL查询每次连接返回执行到连接池，以确保它仍然是有效的。

public String validationQuery;

private boolean validateOnCheckout;

private int idleConnectionValidationSeconds;

public String maxCachedStatementsPerConnection;

private String discardIdleConnectionsSeconds;

public static final int DEFAULT\_DB\_MAX\_CONNECTIONS = 10;

public static final int DEFAULT\_DB\_MAX\_CACHED\_STATEMENTS\_PER\_CONNECTION = 120;

//Druid连接池

private DruidDataSource datasource;

@Override

public Connection getConnection() throws SQLException {

return datasource.getConnection();

}

@Override

public void shutdown() throws SQLException {

datasource.close();

}

@Override

public void initialize() throws SQLException {

if (this.URL == null) {

throw new SQLException("DBPool could not be created: DB URL cannot be null");

}

if (this.driver == null) {

throw new SQLException("DBPool driver could not be created: DB driver class name cannot be null!");

}

if (this.maxConnection < 0) {

throw new SQLException("DBPool maxConnectins could not be created: Max connections must be greater than zero!");

}

datasource = new DruidDataSource();

try{

datasource.setDriverClassName(this.driver);

} catch (Exception e) {

try {

throw new SchedulerException("Problem setting driver class name on datasource: " + e.getMessage(), e);

} catch (SchedulerException e1) {

}

}

datasource.setUrl(this.URL);

datasource.setUsername(this.user);

datasource.setPassword(this.password);

datasource.setMaxActive(this.maxConnection);

datasource.setMinIdle(1);

datasource.setMaxWait(0);

datasource.setMaxPoolPreparedStatementPerConnectionSize(DEFAULT\_DB\_MAX\_CONNECTIONS);

if (this.validationQuery != null) {

datasource.setValidationQuery(this.validationQuery);

if(!this.validateOnCheckout)

datasource.setTestOnReturn(true);

else

datasource.setTestOnBorrow(true);

datasource.setValidationQueryTimeout(this.idleConnectionValidationSeconds);

}

}

public String getDriver() {

return driver;

}

public void setDriver(String driver) {

this.driver = driver;

}

public String getURL() {

return URL;

}

public void setURL(String URL) {

this.URL = URL;

}

public String getUser() {

return user;

}

public void setUser(String user) {

this.user = user;

}

public String getPassword() {

return password;

}

public void setPassword(String password) {

this.password = password;

}

public int getMaxConnection() {

return maxConnection;

}

public void setMaxConnection(int maxConnection) {

this.maxConnection = maxConnection;

}

public String getValidationQuery() {

return validationQuery;

}

public void setValidationQuery(String validationQuery) {

this.validationQuery = validationQuery;

}

public boolean isValidateOnCheckout() {

return validateOnCheckout;

}

public void setValidateOnCheckout(boolean validateOnCheckout) {

this.validateOnCheckout = validateOnCheckout;

}

public int getIdleConnectionValidationSeconds() {

return idleConnectionValidationSeconds;

}

public void setIdleConnectionValidationSeconds(int idleConnectionValidationSeconds) {

this.idleConnectionValidationSeconds = idleConnectionValidationSeconds;

}

public DruidDataSource getDatasource() {

return datasource;

}

public void setDatasource(DruidDataSource datasource) {

this.datasource = datasource;

}

public String getDiscardIdleConnectionsSeconds() {

return discardIdleConnectionsSeconds;

}

public void setDiscardIdleConnectionsSeconds(String discardIdleConnectionsSeconds) {

this.discardIdleConnectionsSeconds = discardIdleConnectionsSeconds;

}}

创建完成之后，还需要在quartz.properties配置文件中设置一下即可

#数据库连接池，将其设置为druid

org.quartz.dataSource.qzDS.connectionProvider.class=com.example.cluster.quartz.config.DruidConnectionProvider

### Quartz 服务接口定义和默认实现

public interface QuartzJobService {

/\*\*

\* 添加任务可以传参数

\* @param clazzName

\* @param jobName

\* @param groupName

\* @param cronExp

\* @param param

\*/

void addJob(String clazzName, String jobName, String groupName, String cronExp, Map<String, Object> param);

/\*\*

\* 暂停任务

\* @param jobName

\* @param groupName

\*/

void pauseJob(String jobName, String groupName);

/\*\*

\* 恢复任务

\* @param jobName

\* @param groupName

\*/

void resumeJob(String jobName, String groupName);

/\*\*

\* 立即运行一次定时任务

\* @param jobName

\* @param groupName

\*/

void runOnce(String jobName, String groupName);

/\*\*

\* 更新任务

\* @param jobName

\* @param groupName

\* @param cronExp

\* @param param

\*/

void updateJob(String jobName, String groupName, String cronExp, Map<String, Object> param);

/\*\*

\* 删除任务

\* @param jobName

\* @param groupName

\*/

void deleteJob(String jobName, String groupName);

/\*\*

\* 启动所有任务

\*/

void startAllJobs();

/\*\*

\* 暂停所有任务

\*/

void pauseAllJobs();

/\*\*

\* 恢复所有任务

\*/

void resumeAllJobs();

/\*\*

\* 关闭所有任务

\*/

void shutdownAllJobs();}

对应的实现类QuartzJobServiceImpl如下：

@Service

public class QuartzJobServiceImpl implements QuartzJobService {

private static final Logger log = LoggerFactory.getLogger(QuartzJobServiceImpl.class);

@Autowired

private Scheduler scheduler;

@Override

public void addJob(String clazzName, String jobName, String groupName, String cronExp, Map<String, Object> param) {

try {

// 启动调度器，默认初始化的时候已经启动// scheduler.start();

//构建job信息

Class<? extends Job> jobClass = (Class<? extends Job>) Class.forName(clazzName);

JobDetail jobDetail = JobBuilder.newJob(jobClass).withIdentity(jobName, groupName).build();

//表达式调度构建器(即任务执行的时间)

CronScheduleBuilder scheduleBuilder = CronScheduleBuilder.cronSchedule(cronExp);

//按新的cronExpression表达式构建一个新的trigger

CronTrigger trigger = TriggerBuilder.newTrigger().withIdentity(jobName, groupName).withSchedule(scheduleBuilder).build();

//获得JobDataMap，写入数据

if (param != null) {

trigger.getJobDataMap().putAll(param);

}

scheduler.scheduleJob(jobDetail, trigger);

} catch (Exception e) {

log.error("创建任务失败", e);

}

}

@Override

public void pauseJob(String jobName, String groupName) {

try {

scheduler.pauseJob(JobKey.jobKey(jobName, groupName));

} catch (SchedulerException e) {

log.error("暂停任务失败", e);

}

}

@Override

public void resumeJob(String jobName, String groupName) {

try {

scheduler.resumeJob(JobKey.jobKey(jobName, groupName));

} catch (SchedulerException e) {

log.error("恢复任务失败", e);

}

}

@Override

public void runOnce(String jobName, String groupName) {

try {

scheduler.triggerJob(JobKey.jobKey(jobName, groupName));

} catch (SchedulerException e) {

log.error("立即运行一次定时任务失败", e);

}

}

@Override

public void updateJob(String jobName, String groupName, String cronExp, Map<String, Object> param) {

try {

TriggerKey triggerKey = TriggerKey.triggerKey(jobName, groupName);

CronTrigger trigger = (CronTrigger) scheduler.getTrigger(triggerKey);

if (cronExp != null) {

// 表达式调度构建器

CronScheduleBuilder scheduleBuilder = CronScheduleBuilder.cronSchedule(cronExp);

// 按新的cronExpression表达式重新构建trigger

trigger = trigger.getTriggerBuilder().withIdentity(triggerKey).withSchedule(scheduleBuilder).build();

}

//修改map

if (param != null) {

trigger.getJobDataMap().putAll(param);

}

// 按新的trigger重新设置job执行

scheduler.rescheduleJob(triggerKey, trigger);

} catch (Exception e) {

log.error("更新任务失败", e);

}

}

@Override

public void deleteJob(String jobName, String groupName) {

try {

//暂停、移除、删除

scheduler.pauseTrigger(TriggerKey.triggerKey(jobName, groupName));

scheduler.unscheduleJob(TriggerKey.triggerKey(jobName, groupName));

scheduler.deleteJob(JobKey.jobKey(jobName, groupName));

} catch (Exception e) {

log.error("删除任务失败", e);

}

}

@Override

public void startAllJobs() {

try {

scheduler.start();

} catch (Exception e) {

log.error("开启所有的任务失败", e);

}

}

@Override

public void pauseAllJobs() {

try {

scheduler.pauseAll();

} catch (Exception e) {

log.error("暂停所有任务失败", e);

}

}

@Override

public void resumeAllJobs() {

try {

scheduler.resumeAll();

} catch (Exception e) {

log.error("恢复所有任务失败", e);

}

}

@Override

public void shutdownAllJobs() {

try {

if (!scheduler.isShutdown()) {

// 需谨慎操作关闭scheduler容器

// scheduler生命周期结束，无法再 start() 启动scheduler

scheduler.shutdown(true);

}

} catch (Exception e) {

log.error("关闭所有的任务失败", e);

}

}}

### Quartz 请求参数Dto定义

public class QuartzConfigDTO implements Serializable {

private static final long serialVersionUID = 1L;

/\*\*

\* 任务名称

\*/

private String jobName;

/\*\*

\* 任务所属组

\*/

private String groupName;

/\*\*

\* 任务执行类

\*/

private String jobClass;

/\*\*

\* 任务调度时间表达式

\*/

private String cronExpression;

/\*\*

\* 附加参数

\*/

private Map<String, Object> param;

public String getJobName() {

return jobName;

}

public QuartzConfigDTO setJobName(String jobName) {

this.jobName = jobName;

return this;

}

public String getGroupName() {

return groupName;

}

public QuartzConfigDTO setGroupName(String groupName) {

this.groupName = groupName;

return this;

}

public String getJobClass() {

return jobClass;

}

public QuartzConfigDTO setJobClass(String jobClass) {

this.jobClass = jobClass;

return this;

}

public String getCronExpression() {

return cronExpression;

}

public QuartzConfigDTO setCronExpression(String cronExpression) {

this.cronExpression = cronExpression;

return this;

}

public Map<String, Object> getParam() {

return param;

}

public QuartzConfigDTO setParam(Map<String, Object> param) {

this.param = param;

return this;

}}

### Quartz 之控制层

@RestController

@RequestMapping("/quartz")

public class QuartzController {

private static final Logger log = LoggerFactory.getLogger(QuartzController .class);

@Autowired

private QuartzJobService quartzJobService;

/\*\*

\* 添加新任务

\* @param configDTO

\* @return

\*/

@RequestMapping("/addJob")

public Object addJob(@RequestBody QuartzConfigDTO configDTO) {

quartzJobService.addJob(configDTO.getJobClass(), configDTO.getJobName(), configDTO.getGroupName(), configDTO.getCronExpression(), configDTO.getParam());

return HttpStatus.OK;

}

/\*\*

\* 暂停任务

\* @param configDTO

\* @return

\*/

@RequestMapping("/pauseJob")

public Object pauseJob(@RequestBody QuartzConfigDTO configDTO) {

quartzJobService.pauseJob(configDTO.getJobName(), configDTO.getGroupName());

return HttpStatus.OK;

}

/\*\*

\* 恢复任务

\* @param configDTO

\* @return

\*/

@RequestMapping("/resumeJob")

public Object resumeJob(@RequestBody QuartzConfigDTO configDTO) {

quartzJobService.resumeJob(configDTO.getJobName(), configDTO.getGroupName());

return HttpStatus.OK;

}

/\*\*

\* 立即运行一次定时任务

\* @param configDTO

\* @return

\*/

@RequestMapping("/runOnce")

public Object runOnce(@RequestBody QuartzConfigDTO configDTO) {

quartzJobService.runOnce(configDTO.getJobName(), configDTO.getGroupName());

return HttpStatus.OK;

}

/\*\*

\* 更新任务

\* @param configDTO

\* @return

\*/

@RequestMapping("/updateJob")

public Object updateJob(@RequestBody QuartzConfigDTO configDTO) {

quartzJobService.updateJob(configDTO.getJobName(), configDTO.getGroupName(), configDTO.getCronExpression(), configDTO.getParam());

return HttpStatus.OK;

}

/\*\*

\* 删除任务

\* @param configDTO

\* @return

\*/

@RequestMapping("/deleteJob")

public Object deleteJob(@RequestBody QuartzConfigDTO configDTO) {

quartzJobService.deleteJob(configDTO.getJobName(), configDTO.getGroupName());

return HttpStatus.OK;

}

/\*\*

\* 启动所有任务

\* @return

\*/

@RequestMapping("/startAllJobs")

public Object startAllJobs() {

quartzJobService.startAllJobs();

return HttpStatus.OK;

}

/\*\*

\* 暂停所有任务

\* @return

\*/

@RequestMapping("/pauseAllJobs")

public Object pauseAllJobs() {

quartzJobService.pauseAllJobs();

return HttpStatus.OK;

}

/\*\*

\* 恢复所有任务

\* @return

\*/

@RequestMapping("/resumeAllJobs")

public Object resumeAllJobs() {

quartzJobService.resumeAllJobs();

return HttpStatus.OK;

}

/\*\*

\* 关闭所有任务

\* @return

\*/

@RequestMapping("/shutdownAllJobs")

public Object shutdownAllJobs() {

quartzJobService.shutdownAllJobs();

return HttpStatus.OK;

}

}

### Quartz UI管理界面



