版本：

Quartz 2.2.3

官网链接

#一、 Quartz简介

1.1 简介

Quartz是一个任务调度框架。比如你遇到这样的问题
想每月29号，信用卡自动还款
想每年4月1日自己给当年暗恋女神发一封匿名贺卡
想每隔1小时，备份一下自己的学习笔记

这些问题总结起来就是：在某一个有规律的时间点干某件事。并且时间的触发的条件可以非常复杂（比如每月最后一个工作日的17:50），复杂到需要一个专门的框架来干这个事。 Quartz就是来干这样的事，你给它一个触发条件的定义，它负责到了时间点，触发相应的Job起来干活
如果应用程序需要在给定时间执行任务，或者如果系统有连续维护作业，那么Quartz是理想的解决方案。

1.2 特点

1.2.1 作业调度

作业被安排在一个给定的触发时运行。触发器可以使用以下指令的组合来创建：

在一天中的某个时间（到毫秒）
在一周的某几天
在每月的某一天
在一年中的某些日期
不在注册的日历中列出的特定日期（如商业节假日除外）
重复特定次数
重复进行，直到一个特定的时间/日期
无限重复
重复的延迟时间间隔

1.2.2 作业持久性

Quartz的设计包括一个作业存储接口，有多种实现。
通过使用包含的JDBCJobStore，所有的作业和触发器配置为“非挥发性”都存储在通过JDBC关系数据库。
通过使用包含的RAMJobStore，所有的作业和触发器存储在RAM，因此不计划执行仍然存在 - 但这是无需使用外部数据库的优势。

二、 Quartz使用

##2.1 导入依赖

```
<dependencies>
    <!--Quartz任务调度-->
    <!-- https://mvnrepository.com/artifact/org.quartz-scheduler/quartz -->
    <dependency>
        <groupId>org.quartz-scheduler</groupId>
        <artifactId>quartz</artifactId>
        <version>2.2.3</version>
    </dependency>
</dependencies>
```

2.2 定义Job

```
/**
 * 工作类的具体实现，即需要定时执行的“某件事”
 * */
public class HelloQuartz implements Job {
    //执行
    public void execute(JobExecutionContext context) throws JobExecutionException {
         //创建工作详情
```

```
        JobDetail jobDetail=context.getJobDetail();
        //获取工作的名称
        String name = jobDetail.getKey().getName();//任务名
        String group = jobDetail.getKey().getGroup();//任务group
        String job=jobDetail.getJobDataMap().getString("data04");//任务中的数据
        System.out.println("job执行，job名：  "+name+" group:"+group+" data:"+job+new Date());
    }
}
```

2.3 API测试

```
public static void main(String[] args) {
        try{
            //创建scheduler，调度器
            Scheduler scheduler = StdSchedulerFactory.getDefaultScheduler();
            //定义一个Trigger,触发条件类
            TriggerBuilder triggerBuilder = TriggerBuilder.newTrigger();
            triggerBuilder.withIdentity("trigger1", "group1") //定义name/group
                          .startNow()//一旦加入scheduler，立即生效，即开始时间
                          .withSchedule(SimpleScheduleBuilder.simpleSchedule()
                                                             .withIntervalInSeconds(1) //每隔一秒执行一次
                                                             .repeatForever()) //一直执行，直到结束时间
                      //可以设置结束时间,如果不设置，则一直执行
                    .endAt(new GregorianCalendar(2019,7,15,16,7,0).getTime());
            Trigger trigger =triggerBuilder.build();
            //定义一个JobDetail
            //定义Job类为HelloQuartz类，这是真正的执行逻辑所在
            JobDetail job = JobBuilder.newJob(HelloQuartz.class)
                    .withIdentity("测试任务1","test") //定义name/group
                    .usingJobData("data04","jobData_zhj") //定义属性，存储数据
                    .build();
            //调度器 中加入 任务和触发器
            scheduler.scheduleJob(job, trigger);
            //启动任务调度
            scheduler.start();
        }catch (Exception ex){
            ex.printStackTrace();
        }
}
```

2.4 配置

```
# 名为：quartz.properties，放置在classpath下，如果没有此配置则按默认配置启动
# 指定调度器名称，非实现类
org.quartz.scheduler.instanceName = DefaultQuartzScheduler04
# 指定线程池实现类
org.quartz.threadPool.class = org.quartz.simpl.SimpleThreadPool
# 线程池线程数量
org.quartz.threadPool.threadCount = 10
# 优先级，默认5
org.quartz.threadPool.threadPriority = 5
# 非持久化job
org.quartz.jobStore.class = org.quartz.simpl.RAMJobStore
```

2.5 核心类说明

```
Scheduler：调度器。所有的调度都是由它控制
  Scheduler就是Quartz的大脑，所有任务都是由它来设施
  Schduelr包含一个两个重要组件：JobStore和ThreadPool
    JobStore是会来存储运行时信息的，包括Trigger,Schduler,JobDetail，业务锁等
    ThreadPool就是线程池，Quartz有自己的线程池实现。所有任务的都会由线程池执行
```

```
SchdulerFactory，顾名思义就是来用创建Schduler了，有两个实现：DirectSchedulerFactory和 StdSchdulerFactory。前者可以用来在代码里定制你自己的Schduler参数。后者是直接读取classpath下的quartz.properties（不存在就都使用默认值）配置来实例化Schduler。通常来讲，我们使用StdSchdulerFactory也就足够了。
SchdulerFactory本身是支持创建RMI stub的，可以用来管理远程的Scheduler，功能与本地一样
```

三、 Trigger （重点）

3.1 SimpleTrigger

指定从某一个时间开始，以一定的时间间隔（单位是毫秒）执行的任务。
它适合的任务类似于：9:00 开始，每隔1小时，执行一次。
它的属性有：
repeatInterval 重复间隔
repeatCount 重复次数。实际执行次数是 repeatCount+1。因为在startTime的时候一定会执行一次。

示例：

```
SimpleScheduleBuilder.simpleSchedule().
                withIntervalInSeconds(10).//每隔10秒执行一次
                repeatForever().//永远执行
                build();
```

```
SimpleScheduleBuilder.simpleSchedule().
                withIntervalInMinutes(3).//每隔3分钟执行一次
                withRepeatCount(3).//执行3次
                build();
```

3.2 CalendarIntervalTrigger

类似于SimpleTrigger，指定从某一个时间开始，以一定的时间间隔执行的任务。 但是不同的是SimpleTrigger指定的时间间隔为毫秒，没办法指定每隔一个月执行一次（每月的时间间隔不是固定值），而CalendarIntervalTrigger支持的间隔单位有秒，分钟，小时，天，月，年，星期。

示例：

```
CalendarIntervalScheduleBuilder.calendarIntervalSchedule()
                .withIntervalInDays(2) //每2天执行一次
                .build();
```

```
CalendarIntervalScheduleBuilder.calendarIntervalSchedule()
                .withIntervalInWeeks(1) //每周执行一次
                .build();
```

3.3 DailyTimeIntervalTrigger

指定每天的某个时间段内，以一定的时间间隔执行任务。并且它可以支持指定星期。
它适合的任务类似于：指定每天9:00 至 18:00 ，每隔70秒执行一次，并且只要周一至周五执行。
它的属性有：
startTimeOfDay 每天开始时间
endTimeOfDay 每天结束时间
daysOfWeek 需要执行的星期
interval 执行间隔
intervalUnit 执行间隔的单位（秒，分钟，小时，天，月，年，星期）
repeatCount 重复次数

示例：

```
DailyTimeIntervalScheduleBuilder.dailyTimeIntervalSchedule()
                .startingDailyAt(TimeOfDay.hourAndMinuteOfDay(9, 0)) //每天9：00开始
                .endingDailyAt(TimeOfDay.hourAndMinuteOfDay(18, 0)) //18：00 结束
.onDaysOfTheWeek(MONDAY,TUESDAY,WEDNESDAY,THURSDAY,FRIDAY) //周一至周五执行
                .withIntervalInHours(1) //每间隔1小时执行一次
                .withRepeatCount(100) //最多重复100次（实际执行100+1次）
                .build();
```

```
DailyTimeIntervalScheduleBuilder.dailyTimeIntervalSchedule()
                    .startingDailyAt(TimeOfDay.hourAndMinuteOfDay(10, 0)) //每天10：00开始
                    .endingDailyAfterCount(10) //每天执行10次，这个方法实际上根据 startTimeOfDay+interval*count
算出 endTimeOfDay
                    .onDaysOfTheWeek(MONDAY,TUESDAY,WEDNESDAY,THURSDAY,FRIDAY) //周一至周五执行
                    .withIntervalInHours(1) //每间隔1小时执行一次
                    .build();
```

3.4 CronTrigger （重点）

适合于更复杂的任务，它支持类型于Linux Cron的语法（并且更强大）。基本上它覆盖了以上三个Trigger的绝大部分能力（但不是全部）——当然，也更难理解。
它适合的任务类似于：每天0:00,9:00,18:00各执行一次。
它的属性只有：
Cron表达式。但这个表示式本身就够复杂了

示例：

```
 CronScheduleBuilder.cronSchedule("0 0/2 10-12 * * ?") // 每天10:00-12:00，每隔2分钟执行一次
 .build();
```

```
cronSchedule("0 30 9 ? * MON") // 每周一，9:30执行一次
.build();
```

```
 CronScheduleBuilder.weeklyOnDayAndHourAndMinute(MONDAY,9, 30) //等同于 0 30 9 ? * MON
 .build();
```

3.4.1 Cron表达式

位置	时间域	允许值	特殊值
1	秒	0-59	, - * /
2	分钟	0-59	, - * /
3	小时	0-23	, - * /
4	日期	1-31	, - * ? / L W
5	月份	1-12	, - * /
6	星期	1-7	, - * ? / L #
7	年份（可选）	1-31	, - * /

> [1] [2] [3] ？ [5] 3#5

星号(*)：可用在所有字段中，表示对应时间域的每一个时刻，例如， 在分钟字段时，表示“每分钟”；

问号（?）：该字符只在日期和星期字段中使用，它通常指定为“不确定值”

减号(-)：表达一个范围，如在小时字段中使用“10-12”，则表示从10到12点，即10,11,12;

逗号(,)：表达一个列表值，如在星期字段中使用“MON,WED,FRI”，则表示星期一，星期三和星期五；

斜杠(/)：x/y表达一个等步长序列，x为起始值，y为增量步长值。如在分钟字段中使用0/15，则表示为0,15,30和45秒，而5/15在分钟字段中表示5,20,35,50，你也可以使用*/y，它等同于0/y；

L：该字符只在日期和星期字段中使用，代表“Last”的意思，但它在两个字段中意思不同。L在日期字段中，表示这个月份的最后一天，如一月的31号，非闰年二月的28号；如果L用在星期中，则表示星期六，等同于7。但是，如果L出现在星期字段里，而且在前面有一个数值X，则表示“这个月的最后一个周x”，例如，6L表示该月的最后星期五；

W：该字符只能出现在日期字段里，是对前导日期的修饰，表示离该日期最近的工作日。例如15W表示离该月15号最近的工作日，如果该月15号是星期六，则匹配14号星期五；如果15日是星期日，则匹配16号星期一；如果15号是星期二，那结果就是15号星期二。但必须注意关联的匹配日期不能够跨月，如你指定1W，如果1号是星期六，结果匹配的是3号星期一，而非上个月最后的那天。W字符串只能指定单一日期，而不能指定日期范围；

LW组合：在日期字段可以组合使用LW，它的意思是当月的最后一个工作日；

表达式示例：

表示式	说明
秒 分 时 日 月 周	
0 0 12 * * ?	每天12点运行
0 15 10 * * ?	每天10:15运行
0 15 10 * * ? 2008	在2008年的每天10：15运行
0 * 14 * * ?	每天14点到15点之间每分钟运行一次，开始于14:00，结束于14:59。
0 0/5 14 * * ?	每天14点到15点每5分钟运行一次，开始于14:00，结束于14:55。
0 0/5 14,18 * * ?	每天14点到15点每5分钟运行一次，此外每天18点到19点每5钟也运行一次。
0 0-5 14 * * ?	每天14:00点到14:05，每分钟运行一次。
0 0-5/2 14 * * ?	每天14:00点到14:05，每2分钟运行一次。
0 10,44 14 ? 3 WED	3月每周三的14:10分和14:44，每分钟运行一次。
0 15 10 ? * MON-FRI	每周一，二，三，四，五的10:15分运行。
0 15 10 15 * ?	每月15日10:15分运行。
0 15 10 L * ?	每月最后一天10:15分运行。
0 15 10 ? * 6L	每月最后一个星期五10:15分运行。【此时天必须是"?"】
0 15 10 ? * 6L 2007-2009	在2007,2008,2009年每个月的最后一个星期五的10:15分运行。

> Calendar不是jdk的java.util.Calendar，不是为了计算日期的。它的作用是在于补充Trigger的时间。可以排除或加入某一些特定的时间点。
> 以”每月29日零点自动还信用卡“为例，我们想排除掉每年的2月29号零点这个时间点（因为平年和润年2月不一样）。这个时间，就可以用Calendar来实现

> Quartz提供以下几种Calendar，注意，所有的Calendar既可以是排除，也可以是包含，取决于：
> HolidayCalendar。指定特定的日期，比如20140613。精度到天。
> DailyCalendar。指定每天的时间段（rangeStartingTime, rangeEndingTime），格式是HH:MM[:SS[:mmm]]。也就是最大精度可以到毫秒。
> WeeklyCalendar。指定每星期的星期几，可选值比如为java.util.Calendar.SUNDAY。精度是天。
> MonthlyCalendar。指定每月的几号。可选值为1-31。精度是天
> AnnualCalendar。 指定每年的哪一天。使用方式如上例。精度是天。
> CronCalendar。指定Cron表达式。精度取决于Cron表达式，也就是最大精度可以到秒。

> 当scheduler比较繁忙的时候，可能在同一个时刻，有多个Trigger被触发了，但资源不足（比如线程池不足）。那么这个时候比剪刀石头布更好的方式，就是设置优先级。优先级高的先执行。
> 需要注意的是，优先级只有在同一时刻执行的Trigger之间才会起作用，如果一个Trigger是9:00，另一个Trigger是9:30。那么无论后一个优先级多高，前一个都是先执行。
> 优先级的值默认是5，当为负数时使用默认值。最大值似乎没有指定，但建议遵循Java的标准，使用1-10，不然鬼才知道看到【优先级为10】是时，上头还有没有更大的值。

四、Job并发（重点）

> job是有可能并发执行的，比如一个任务要执行10秒中，而调度算法是每秒中触发1次，那么就有可能多个任务被并发执行。
>
> 有时候我们并不想任务并发执行，比如这个任务要去”获得数据库中所有未发送邮件的名单“，如果是并发执行，就需要一个数据库锁去避免一个数据被多次处理。这个时候一个@DisallowConcurrentExecution解决这个问题

```
@DisallowConcurrentExecution
public class DoNothingJob implements Job {
    public void execute(JobExecutionContext context) throws JobExecutionException {
        System.out.println("操作");
    }
}
```

> 注意，@DisallowConcurrentExecution是对JobDetail实例生效，也就是如果你定义两个JobDetail，引用同一个Job类，是可以并发执行的

代码示例：

```
@DisallowConcurrentExecution //会不允许并发执行，（如果每1s触发一次，但每个job要执行3秒）
public class MyJob implements Job{
    @Override
    public void execute(JobExecutionContext context) throws JobExecutionException {
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("任务调度：组："+group+",工作名："+name+" "+data+new Date());
    }
}
```

五、Spring整合Quartz （重点）

5.1 依赖

```
  <properties>
    <springframework.version>4.3.11.RELEASE</springframework.version>
    <quartz.version>2.2.3</quartz.version>
  </properties>

  <dependencies>

    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context-support</artifactId>
      <version>${springframework.version}</version>
    </dependency>

    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-tx</artifactId>
      <version>${springframework.version}</version>
    </dependency>

    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-web</artifactId>
      <version>${springframework.version}</version>
    </dependency>

    <dependency>
      <groupId>org.quartz-scheduler</groupId>
      <artifactId>quartz</artifactId>
      <version>${quartz.version}</version>
    </dependency>

  </dependencies>
</project>
```

5.2 配置

> 调度器 `SchedulerFactoryBean`
>
> 触发器 `CronTriggerFactoryBean`
>
> JobDetail `JobDetailFactoryBean`

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!--
        Spring整合Quartz进行配置遵循下面的步骤:
        1: 定义工作任务的Job
        2: 定义触发器Trigger, 并将触发器与工作任务绑定
        3: 定义调度器, 并将Trigger注册到Scheduler
     -->

    <!-- 1: 定义任务的bean , 这里使用JobDetailFactoryBean,也可以使用MethodInvokingJobDetailFactoryBean , 配置类
似-->
    <bean name="lxJob" class="org.springframework.scheduling.quartz.JobDetailFactoryBean">
        <!-- 指定job的名称 -->
        <property name="name" value="job1"/>
        <!-- 指定job的分组 -->
        <property name="group" value="group1"/>
        <!-- 指定具体的job类 -->
        <property name="jobClass" value="com.zhj.quartz0.MyJob"/>
        <!-- 如果为false, 当没有活动的触发器与之关联时会在调度器中会删除该任务 (可选) -->
        <property name="durability" value="true"/>
        <!--  (可选)
             指定spring容器的key, 如果不设定在job中的jobmap中是获取不到spring容器的
             其实现了ApplicationContextWare,则其中的setApplicationContext方法会得到
             当前的工厂对象, 且将工厂对象存在了类中的一个属性"applicationContext"中, 源码如下

            getJobDataMap().put(this.applicationContextJobDataKey, this.applicationContext);
            则在Job的jobmap中可以获得工厂对象, 如果需要可以使用
        (ApplicationContext) jobDataMap.get("applicationContext04");
        jobDataMap.get("data04");

        .usingJobData("data04", "hello world~~")
        .usingJobData("applicationContext04",spring工厂对象)
         -->
        <property name="applicationContextJobDataKey" value="applicationContext04"/>
    </bean>

    <bean name="lxJob2" class="org.springframework.scheduling.quartz.JobDetailFactoryBean">
        <!-- 指定job的名称 -->
        <property name="name" value="job2"/>
        <!-- 指定job的分组 -->
        <property name="group" value="group1"/>
        <!-- 指定具体的job类 -->
        <property name="jobClass" value="com.zhj.quartz0.MyJob"/>
        <!-- 如果为false, 当没有活动的触发器与之关联时会在调度器中会删除该任务  -->
        <property name="durability" value="true"/>
        <!-- 指定spring容器的key, 如果不设定在job中的jobmap中是获取不到spring容器的
             其实现了ApplicationContextWare,则其中的setApplicationContext方法会得到
             当前的工厂对象, 且将工厂对象存在了类中的一个属性"applicationContext"中, 源码如下

            getJobDataMap().put(this.applicationContextJobDataKey, this.applicationContext);
            则在Job的jobmap中可以获得工厂对象, 如果需要可以使用
        -->
        <property name="applicationContextJobDataKey" value="applicationContext"/>
    </bean>


    <!-- 2.2: 定义触发器的bean, 定义一个Cron的Trigger, 一个触发器只能和一个任务进行绑定 -->
    <bean id="cronTrigger" class="org.springframework.scheduling.quartz.CronTriggerFactoryBean">
        <!-- 指定Trigger的名称 -->
        <property name="name" value="hw_trigger"/>
        <!-- 指定Trigger的名称 -->
        <property name="group" value="hw_trigger_group"/>
        <!-- 指定Tirgger绑定的Job -->
        <property name="jobDetail" ref="lxJob"/>
        <!-- 指定Cron 的表达式 , 当前是每隔5s运行一次 -->
        <property name="cronExpression" value="*/5 * * * * ?" />
    </bean>

    <!-- 2.2: 定义触发器的bean, 定义一个Cron的Trigger, 一个触发器只能和一个任务进行绑定 -->
```

```
    <bean id="cronTrigger2" class="org.springframework.scheduling.quartz.CronTriggerFactoryBean">
        <!-- 指定Trigger的名称 -->
        <property name="name" value="hw_trigger2"/>
        <!-- 指定Trigger的名称 -->
        <property name="group" value="hw_trigger_group"/>
        <!-- 指定Tirgger绑定的Job -->
        <property name="jobDetail" ref="lxJob2"/>
        <!-- 指定Cron 的表达式 ，当前是每隔5s运行一次 -->
        <property name="cronExpression" value="*/5 * * * * ?" />
    </bean>

    <!-- 3.定义调度器，并将Trigger注册到调度器中 -->
    <bean id="scheduler" class="org.springframework.scheduling.quartz.SchedulerFactoryBean">
        <property name="triggers">
            <list>
                <ref bean="cronTrigger"/>
                <ref bean="cronTrigger2"/>
            </list>
        </property>
        <!-- 添加 quartz 配置，如下两种方式均可 -->
        <!--<property name="configLocation" value="classpath:quartz.properties"></property>-->
        <property name="quartzProperties">
            <value>
                # 指定调度器名称，实际类型为：QuartzScheduler
                org.quartz.scheduler.instanceName = MyScheduler
                # 指定连接池
                org.quartz.threadPool.class = org.quartz.simpl.SimpleThreadPool
                # 连接池线程数量
                org.quartz.threadPool.threadCount = 11
                # 优先级
                org.quartz.threadPool.threadPriority = 5
                # 不持久化job
                org.quartz.jobStore.class = org.quartz.simpl.RAMJobStore
            </value>
        </property>
    </bean>
</beans>
```

web.xml 项目的全局配置

```
<!DOCTYPE web-app PUBLIC
 "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
 "http://java.sun.com/dtd/web-app_2_3.dtd" >

<web-app>
  <display-name>Archetype Created Web Application</display-name>
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:applicationContext.xml</param-value>
  </context-param>

  <listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
  </listener>
</web-app>
```

5.3 代码

MyJob 自定义任务类

```
public class MyJob implements Job {
    public void execute(JobExecutionContext jobExecutionContext) throws JobExecutionException {
        System.err.println("job 执行"+new Date());
    }
}
```

```
public static void main(String[] args) throws InterruptedException, SchedulerException {
    ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
        System.out.println("==============");
        StdScheduler scheduler = (StdScheduler) context.getBean("scheduler");
        System.out.println(scheduler.getClass());
```

```
            Thread.sleep(3000);
        // 如下api可以设计几个Controller，用来做job的暂停和删除
            scheduler.pauseTrigger(TriggerKey.triggerKey("hw_trigger", "hw_trigger_group"));
            scheduler.unscheduleJob(TriggerKey.triggerKey("hw_trigger", "hw_trigger_group"));
            scheduler.deleteJob(JobKey.jobKey("job1","group1"));//移除trigger后，删除工作
            /*
            scheduler.pauseJob(new JobKey("job2","group1"));// 暂停工作
            Thread.sleep(3000);
            scheduler.resumeJob(new JobKey("job2","group1"));// 恢复工作
            */
            /*
            Thread.sleep(2000);
            GroupMatcher<JobKey> group1 = GroupMatcher.groupEquals("group1");
            scheduler.pauseJobs(group1); //暂停工作组中所有工作
            Thread.sleep(2000);
            scheduler.resumeJobs(group1);
            //删除工作，没有group的操作
            */
        /*
        CronTrigger Trigger = TriggerBuilder.newTrigger().withIdentity(jt.getJobName(), jt.getJobGroup())
                .withSchedule(CronScheduleBuilder.cronSchedule(jt.getCronExpression())).build();
            scheduler.rescheduleJob(TriggerKey.triggerKey(jt.getJobName(),jt.getJobGroup()),cronTrigger);
            **/

    }
```

六、持久化

6.1 建表

> quartz官方提供了完整的持久化job的支持，并给出了一套库表

```
create table QRTZ_CALENDARS
(
  SCHED_NAME    varchar(120) not null,
  CALENDAR_NAME varchar(200) not null,
  CALENDAR      blob         not null,
  primary key (SCHED_NAME, CALENDAR_NAME)
);

create table QRTZ_FIRED_TRIGGERS
(
  SCHED_NAME        varchar(120) not null,
  ENTRY_ID          varchar(95)  not null,
  TRIGGER_NAME      varchar(200) not null,
  TRIGGER_GROUP     varchar(200) not null,
  INSTANCE_NAME     varchar(200) not null,
  FIRED_TIME        bigint(13)   not null,
  SCHED_TIME        bigint(13)   not null,
  PRIORITY          int          not null,
  STATE             varchar(16)  not null,
  JOB_NAME          varchar(200) null,
  JOB_GROUP         varchar(200) null,
  IS_NONCONCURRENT  varchar(1)   null,
  REQUESTS_RECOVERY varchar(1)   null,
  primary key (SCHED_NAME, ENTRY_ID)
);

create table QRTZ_JOB_DETAILS
(
  SCHED_NAME        varchar(120) not null,
  JOB_NAME          varchar(200) not null,
  JOB_GROUP         varchar(200) not null,
  DESCRIPTION       varchar(250) null,
  JOB_CLASS_NAME    varchar(250) not null,
  IS_DURABLE        varchar(1)   not null,
  IS_NONCONCURRENT  varchar(1)   not null,
  IS_UPDATE_DATA    varchar(1)   not null,
  REQUESTS_RECOVERY varchar(1)   not null,
```

```
  JOB_DATA          blob         null,
  primary key (SCHED_NAME, JOB_NAME, JOB_GROUP)
);

create table QRTZ_LOCKS
(
  SCHED_NAME varchar(120) not null,
  LOCK_NAME  varchar(40)  not null,
  primary key (SCHED_NAME, LOCK_NAME)
);

create table QRTZ_PAUSED_TRIGGER_GRPS
(
  SCHED_NAME    varchar(120) not null,
  TRIGGER_GROUP varchar(200) not null,
  primary key (SCHED_NAME, TRIGGER_GROUP)
);

create table QRTZ_SCHEDULER_STATE
(
  SCHED_NAME        varchar(120) not null,
  INSTANCE_NAME     varchar(200) not null,
  LAST_CHECKIN_TIME bigint(13)   not null,
  CHECKIN_INTERVAL  bigint(13)   not null,
  primary key (SCHED_NAME, INSTANCE_NAME)
);

create table QRTZ_TRIGGERS
(
  SCHED_NAME     varchar(120) not null,
  TRIGGER_NAME   varchar(200) not null,
  TRIGGER_GROUP  varchar(200) not null,
  JOB_NAME       varchar(200) not null,
  JOB_GROUP      varchar(200) not null,
  DESCRIPTION    varchar(250) null,
  NEXT_FIRE_TIME bigint(13)   null,
  PREV_FIRE_TIME bigint(13)   null,
  PRIORITY       int          null,
  TRIGGER_STATE  varchar(16)  not null,
  TRIGGER_TYPE   varchar(8)   not null,
  START_TIME     bigint(13)   not null,
  END_TIME       bigint(13)   null,
  CALENDAR_NAME  varchar(200) null,
  MISFIRE_INSTR  smallint(2)  null,
  JOB_DATA       blob         null,
  primary key (SCHED_NAME, TRIGGER_NAME, TRIGGER_GROUP),
  constraint QRTZ_TRIGGERS_ibfk_1
  foreign key (SCHED_NAME, JOB_NAME, JOB_GROUP) references QRTZ_JOB_DETAILS (SCHED_NAME, JOB_NAME,
JOB_GROUP)
);

create table QRTZ_BLOB_TRIGGERS
(
  SCHED_NAME    varchar(120) not null,
  TRIGGER_NAME  varchar(200) not null,
  TRIGGER_GROUP varchar(200) not null,
  BLOB_DATA     blob         null,
  primary key (SCHED_NAME, TRIGGER_NAME, TRIGGER_GROUP),
  constraint QRTZ_BLOB_TRIGGERS_ibfk_1
  foreign key (SCHED_NAME, TRIGGER_NAME, TRIGGER_GROUP) references QRTZ_TRIGGERS (SCHED_NAME,
TRIGGER_NAME, TRIGGER_GROUP)
);

create table QRTZ_CRON_TRIGGERS
(
  SCHED_NAME      varchar(120) not null,
  TRIGGER_NAME    varchar(200) not null,
  TRIGGER_GROUP   varchar(200) not null,
  CRON_EXPRESSION varchar(200) not null,
  TIME_ZONE_ID    varchar(80)  null,
  primary key (SCHED_NAME, TRIGGER_NAME, TRIGGER_GROUP),
  constraint QRTZ_CRON_TRIGGERS_ibfk_1
```

```
  foreign key (SCHED_NAME, TRIGGER_NAME, TRIGGER_GROUP) references QRTZ_TRIGGERS (SCHED_NAME,
TRIGGER_NAME, TRIGGER_GROUP)
);

create table QRTZ_SIMPLE_TRIGGERS
(
  SCHED_NAME      varchar(120) not null,
  TRIGGER_NAME    varchar(200) not null,
  TRIGGER_GROUP   varchar(200) not null,
  REPEAT_COUNT    bigint(7)    not null,
  REPEAT_INTERVAL bigint(12)   not null,
  TIMES_TRIGGERED bigint(10)   not null,
  primary key (SCHED_NAME, TRIGGER_NAME, TRIGGER_GROUP),
  constraint QRTZ_SIMPLE_TRIGGERS_ibfk_1
  foreign key (SCHED_NAME, TRIGGER_NAME, TRIGGER_GROUP) references QRTZ_TRIGGERS (SCHED_NAME,
TRIGGER_NAME, TRIGGER_GROUP)
);

create table QRTZ_SIMPROP_TRIGGERS
(
  SCHED_NAME    varchar(120)   not null,
  TRIGGER_NAME  varchar(200)   not null,
  TRIGGER_GROUP varchar(200)   not null,
  STR_PROP_1    varchar(512)   null,
  STR_PROP_2    varchar(512)   null,
  STR_PROP_3    varchar(512)   null,
  INT_PROP_1    int            null,
  INT_PROP_2    int            null,
  LONG_PROP_1   bigint         null,
  LONG_PROP_2   bigint         null,
  DEC_PROP_1    decimal(13, 4) null,
  DEC_PROP_2    decimal(13, 4) null,
  BOOL_PROP_1   varchar(1)     null,
  BOOL_PROP_2   varchar(1)     null,
  primary key (SCHED_NAME, TRIGGER_NAME, TRIGGER_GROUP),
  constraint QRTZ_SIMPROP_TRIGGERS_ibfk_1
  foreign key (SCHED_NAME, TRIGGER_NAME, TRIGGER_GROUP) references QRTZ_TRIGGERS (SCHED_NAME,
TRIGGER_NAME, TRIGGER_GROUP)
);
```

6.2 配置

```
<!-- 如果没有固定的任务，可以不再定义 JobDetail 和 trigger，可以动态添加任务 -->
<!-- 定义调度器，并将Trigger注册到调度器中 -->
<bean id="scheduler" class="org.springframework.scheduling.quartz.SchedulerFactoryBean">
    <!-- 添加shiro配置，如下两种方式均可 -->
    <!--<property name="configLocation" value="classpath:quartz.properties"></property>-->
    <property name="quartzProperties">
        <value>
            # 指定调度器名称，实际类型为：QuartzScheduler
            org.quartz.scheduler.instanceName = MyScheduler78
            # 指定连接池
            org.quartz.threadPool.class = org.quartz.simpl.SimpleThreadPool
            # 连接池线程数量
            org.quartz.threadPool.threadCount = 11
            # 优先级
            org.quartz.threadPool.threadPriority = 5
            # 默认存储在内存中
            #org.quartz.jobStore.class = org.quartz.simpl.RAMJobStore
            #持久化
            org.quartz.jobStore.class = org.quartz.impl.jdbcjobstore.JobStoreTX
            #quartz表的前缀
            org.quartz.jobStore.tablePrefix = QRTZ_
        </value>
    </property>
    <!-- 连接池 -->
    <property name="dataSource" ref="druid_03"></property>
</bean>
```

6.3 增加任务

> 定义一个springMVC的 **Handler**

```
@Data
public class JobAndTrigger {
    private String jobName;
    private String jobGroup;
    private String jobClassName;
    private String triggerName;
    private String triggerGroup;
    //private BigInteger repeatInterval;
    //private BigInteger timesTriggered;
    private String cronExpression;
    private String timeZoneId;
}
```

```
@Controller
@RequestMapping("/quartz")
public class TestController {

    @Autowired //注入了工厂中 调度器
    private Scheduler scheduler;

    // 添加一个定时任务
    // 方法的参数：jt=新任务的相关数据
    @RequestMapping("add")
    public String addJob(JobAndTrigger jt) throws ClassNotFoundException, SchedulerException {
        // 创建JobDetail
        JobDetail jobDetail=null;
        jobDetail = JobBuilder.newJob((Class<? extends Job>)Class.forName(jt.getJobClassName()))
                .withIdentity(jt.getJobName(), jt.getJobGroup()).storeDurably(true).build();
        CronTrigger cronTrigger = null;
        cronTrigger = TriggerBuilder.newTrigger().withIdentity(jt.getJobName(),jt.getJobGroup())
                .withSchedule(CronScheduleBuilder.cronSchedule(jt.getCronExpression()))
                .build();
        scheduler.scheduleJob(jobDetail,cronTrigger);
        //scheduler.start();
        return "redirect:query";
    }
```

```
<form action="${pageContext.request.contextPath}/quartz/add" method="post">
    jobName: <input type="text" name="jobName"> <br>
    jobGroup: <input type="text" name="jobGroup"> <br>
    cronExp: <input type="text" name="cronExpression"> <br>
    jobClass: <input type="text" name="jobClassName"> <br>
    <input type="submit" value="增加">
</form>
```