课程目标

1、了解任务调度的应用场景和 Quartz 的基本特性

2、掌握 Quartz Java 编程和 Spring 集成的使用

3、掌握 Quartz 动态调度和集群部署的实现

4、理解 Quartz 原理与线程模型

内容定位

适合没有用过 Quartz 或者只会 Quartz 基本配置的同学

说明：基于最新稳定版本 2.3.0

1 漫谈任务调度

1.1 什么时候需要任务调度？

1.1.1 任务调度的背景

在业务系统中有很多这样的场景：

1、账单日或者还款日上午 10 点，给每个信用卡客户发送账单通知，还款通知。如何判断客户的账单日、还款日，完成通知的发送？

2、银行业务系统，夜间要完成跑批的一系列流程，清理数据，下载文件，解析文件，对账清算、切换结算日期等等。如何触发一系列流程的执行？

3、金融机构跟人民银行二代支付系统对接，人民银行要求低于 5W 的金额（小额支

付）半个小时打一次包发送，以缓解并发压力。所以，银行的跨行转账分成了多个流程：录入、复核、发送。如何把半个小时以内的所有数据一次性发送？

类似于这种 1、基于准确的时刻或者固定的时间间隔触发的任务，或者 2、有批量数据需要处理，或者 3、要实现两个动作解耦的场景，我们都可以用任务调度来实现。

1.2 任务调度需求分析

任务调度的实现方式有很多，如果要实现我们的调度需求，我们对这个工具有什么样的基本要求呢？

1.2.1 基本需求

1）可以定义触发的规则，比如基于时刻、时间间隔、表达式。

2）可以定义需要执行的任务。比如执行一个脚本或者一段代码。任务和规则是分开的。

3）集中管理配置，持久配置。不用把规则写在代码里面，可以看到所有的任务配置，方便维护。重启之后任务可以再次调度——配置文件或者配置中心。

4）支持任务的串行执行，例如执行 A 任务后再执行 B 任务再执行 C 任务。

5）支持多个任务并发执行，互不干扰（例如 ScheduledThreadPoolExecutor）。

6）有自己的调度器，可以启动、中断、停止任务。

7）容易集成到 Spring。

1.3 任务调度工具对比

层次	举例	特点
操作系统	Linux crontab Windows 计划任务	只能执行简单脚本或者命令

数据库	MySQL、Oracle	可以操作数据。不能执行 Java 代码
工具	Kettle	可以操作数据，执行脚本。没有集中配置
开发语言	JDK Timer、ScheduledThreadPool	Timer：单线程 JDK1.5 之后：ScheduledThreadPool（Cache、Fiexed、Single）:没有集中配置，日程管理不够灵活
容器	Spring Task、@Scheduled	不支持集群
分布式框架	XXL-JOB，Elastic-Job	

@Scheduled 也是用 JUC 的 ScheduledExecutorService 实现的

Scheduled(cron = “0 15 10 15 * ?”)

1、 ScheduledAnnotationBeanPostProcessor 的 postProcessAfterInitialization 方法将@Scheduled 的方法包装为指定的 task 添加到 ScheduledTaskRegistrar 中
2、 ScheduledAnnotationBeanPostProcessor 会监听 Spring 的容器初始化事件，在 Spring 容器初始化完成后进行 TaskScheduler 实现类实例的查找，若发现有 SchedulingConfigurer 的实现类实例，则跳过 3
3、 查找 TaskScheduler 的实现类实例默认是通过类型查找，若有多个实现则会查找名字为"taskScheduler"的实现 Bean，若没有找到则在 ScheduledTaskRegistrar 调度任务的时候会创建一个 newSingleThreadScheduledExecutor，将 TaskScheduler 的实现类实例设置到 ScheduledTaskRegistrar 属性中
4、 ScheduledTaskRegistrar 的 scheduleTasks 方法触发任务调度
5、 真正调度任务的类是 TaskScheduler 实现类中的 ScheduledExecutorService，由 J.U.C 提供

2 Quartz 基本介绍

官网：http://www.quartz-scheduler.org/

Quartz 的意思是石英，像石英表一样精确。

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.

Quatz 是一个特性丰富的，开源的任务调度库，它几乎可以嵌入所有的 Java 程序，从很小的独立应用程序到大型商业系统。Quartz 可以用来创建成百上千的简单的或者复杂的任务，这些任务可以用来执行任何程序可以做的事情。Quartz 拥有很多企业级的特性，包括支持 JTA 事务和集群。

Quartz 是一个老牌的任务调度系统，98 年构思，01 年发布到 sourceforge。现在更新比较慢，因为已经非常成熟了。

https://github.com/quartz-scheduler/quartz

Quartz 的目的就是让任务调度更加简单，开发人员只需要关注业务即可。他是用 Java 语言编写的（也有.NET 的版本）。Java 代码能做的任何事情，Quartz 都可以调度。

特点：

精确到毫秒级别的调度

可以独立运行，也可以集成到容器中

支持事务（JobStoreCMT ）

支持集群

支持持久化

3 Quartz Java 编程

http://www.quartz-scheduler.org/documentation/quartz-2.3.0/
http://www.quartz-scheduler.org/documentation/quartz-2.3.0/quick-start.html

3.1 引入依赖

```
<dependency>
  <groupId>org.quartz-scheduler</groupId>
  <artifactId>quartz</artifactId>
  <version>2.3.0</version>
</dependency>
```

3.2 默认配置文件

org.quartz.core 包下，有一个默认的配置文件，quartz.properties。当我们没有定义一个同名的配置文件的时候，就会使用默认配置文件里面的配置。

```
org.quartz.scheduler.instanceName: DefaultQuartzScheduler
org.quartz.scheduler.rmi.export: false
org.quartz.scheduler.rmi.proxy: false
org.quartz.scheduler.wrapJobExecutionInUserTransaction: false
org.quartz.threadPool.class: org.quartz.simpl.SimpleThreadPool
org.quartz.threadPool.threadCount: 10
org.quartz.threadPool.threadPriority: 5
org.quartz.threadPool.threadsInheritContextClassLoaderOfInitializingThread: true
org.quartz.jobStore.misfireThreshold: 60000
org.quartz.jobStore.class: org.quartz.simpl.RAMJobStore
```

3.3 创建 Job

实现唯一的方法 execute() ,方法中的代码就是任务执行的内容。此处仅输出字符串。

```
public class MyJob implements Job {
  public void execute(JobExecutionContext context) throws JobExecutionException {
    System.out.println("假发在哪里买的");
  }
}
```

在测试类 main()方法中，把 Job 进一步包装成 JobDetail。

必须要指定 JobName 和 groupName，两个合起来是唯一标识符。

可以携带 KV 的数据(JobDataMap),用于扩展属性 ,在运行的时候可以从 context 获取到。

```
JobDetail jobDetail = JobBuilder.newJob(MyJob1.class)
      .withIdentity("job1", "group1")
      .usingJobData("gupao","2673")
      .usingJobData("moon",5.21F)
      .build();
```

3.4 创建 Trigger

在测试类 main()方法中，基于 SimpleTrigger 定义了一个每 2 秒钟运行一次、不断重复的 Trigger：

```
Trigger trigger = TriggerBuilder.newTrigger()
    .withIdentity("trigger1", "group1")
    .startNow()
    .withSchedule(SimpleScheduleBuilder.simpleSchedule()
            .withIntervalInSeconds(2)
            .repeatForever())
    .build();
```

3.5 创建 Scheduler

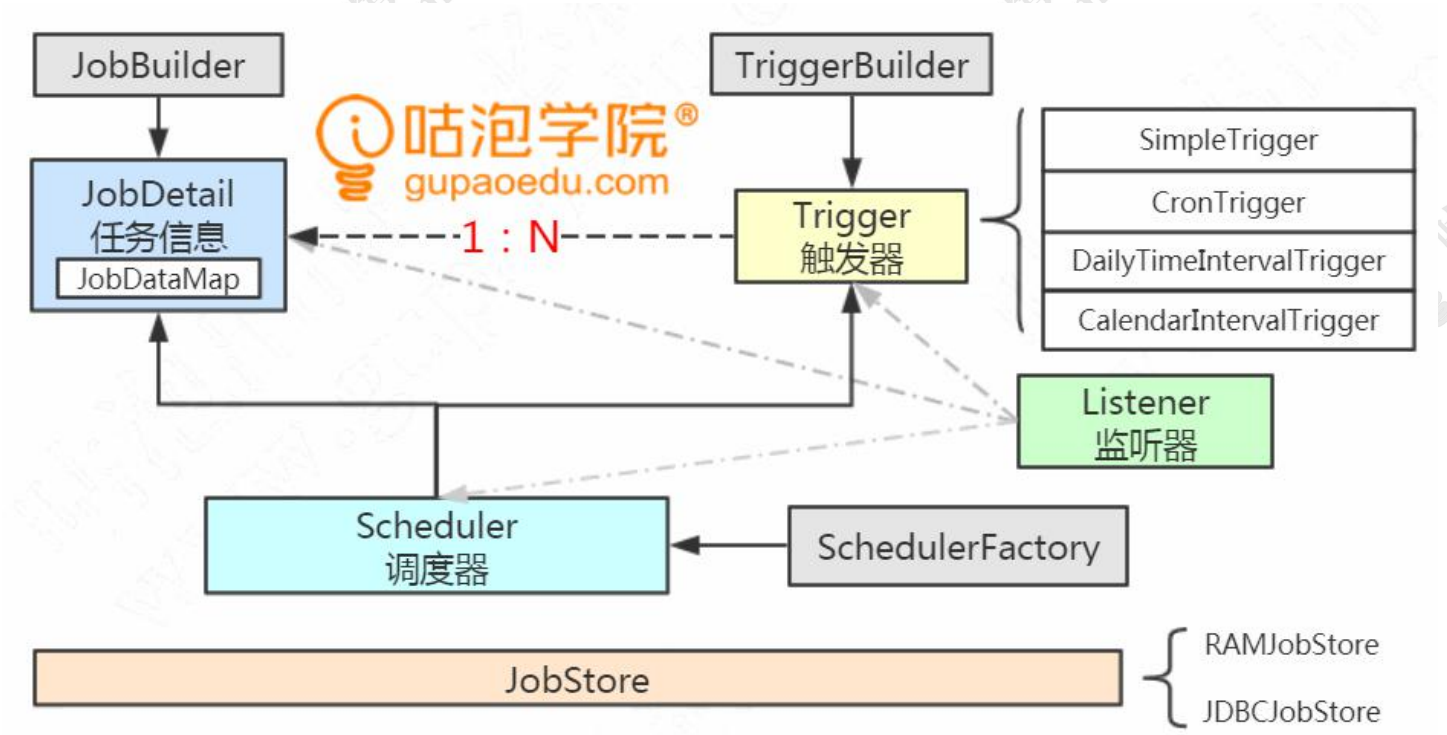
在测试类 main()方法中，通过 Factory 获取调度器的实例，把 JobDetail 和 Trigger 绑定，注册到容器中。

Scheduler 先启动后启动无所谓，只要有 Trigger 到达触发条件，就会执行任务。

```
SchedulerFactory   factory = new StdSchedulerFactory();
Scheduler scheduler = factory.getScheduler();
scheduler.scheduleJob(jobDetail, trigger);
scheduler.start();
```

注意这里，调度器一定是单例的。

3.6 体系结构总结



总体结构

3.6.1 JobDetail

我们创建一个实现 Job 接口的类，使用 JobBuilder 包装成 JobDetail，它可以携带 KV 的数据。

3.6.2 Trigger

定义任务的触发规律，Trigger，使用 TriggerBuilder 来构建。

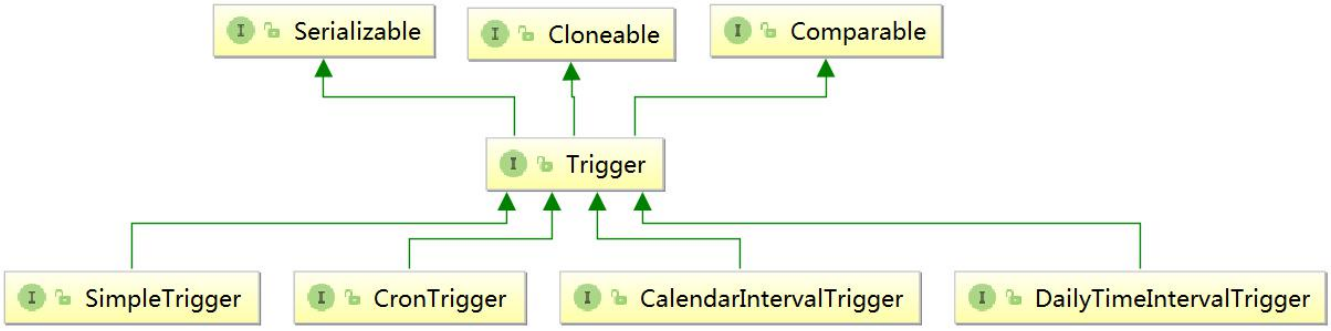
JobDetail 跟 Trigger 是 1:N 的关系。

思考：为什么要解耦？

Trigger 接口在 Quartz 有 4 个继承的子接口：

子接口	描述	特点
SimpleTrigger	简单触发器	固定时刻或时间间隔，毫秒
CalendarIntervalTrigger	基于日历的触发器	比简单触发器更多时间单位，支持非固定时间的触发，例如一年可能 365/366，一个月可能 28/29/30/31
DailyTimeIntervalTrigger	基于日期的触发器	每天的某个时间段
CronTrigger	基于 Cron 表达式的触发器	

MutableTrigger 和 CoreTrigger 最终也是用到以上四个类的实现类。



代码：standalone com.gupaoedu.trigger.TriggerDefine

SimpleTrigger

SimpleTrigger 可以定义固定时刻或者固定时间间隔的调度规则（精确到毫秒）。

例如：每天 9 点钟运行；每隔 30 分钟运行一次。

CalendarIntervalTrigger

CalendarIntervalTrigger 可以定义更多时间单位的调度需求，精确到秒。

好处是不需要去计算时间间隔，比如 1 个小时等于多少毫秒。

例如每年、每个月、每周、每天、每小时、每分钟、每秒。

每年的月数和每个月的天数不是固定的，这种情况也适用。

DailyTimeIntervalTrigger

每天的某个时间段内，以一定的时间间隔执行任务。

例如：每天早上 9 点到晚上 9 点，每隔半个小时执行一次，并且只在周一到周六执行。

CronTrigger

CronTirgger 可以定义基于 Cron 表达式的调度规则，是最常用的触发器类型。

Cron 表达式

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

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“无意义的值”，相当于点位符；

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 X 天”，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；

井号(#)：该字符只能在星期字段中使用，表示当月某个工作日。如 6#3 表示当月的第三个星期五(6 表示星期五，#3 表示当前的第三个)，而 4#5 表示当月的第五个星期三，假设当月没有第五个星期三，忽略不触发；

C：该字符只在日期和星期字段中使用，代表“Calendar”的意思。它的意思是计划所关联的日期，如果日期没有被关联，则相当于日历中所有日期。例如 5C 在日期字段中就相当于日历 5 日以后的第一天。1C 在星期字段中相当于星期日后的第一天。

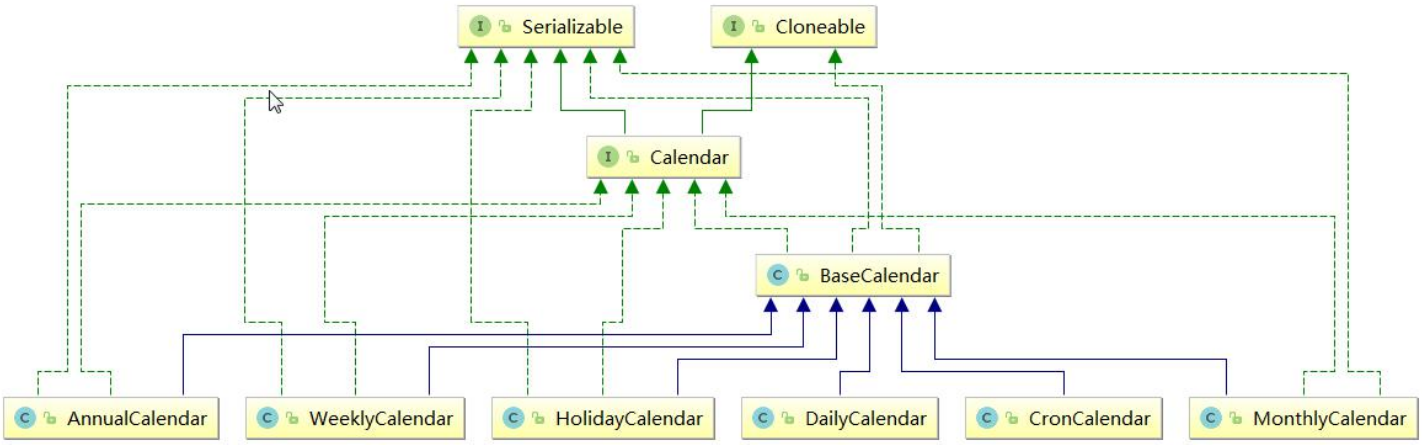
Cron 表达式对特殊字符的大小写不敏感，对代表星期的缩写英文大小写也不敏感。

上面我们定义的都是在什么时间执行，但是我们有一些在什么时间不执行的需求，比如：理财周末和法定假日购买不计息；证券公司周末和法定假日休市。

是不是要把日期写在数据库中，然后读取基于当前时间判断呢？

基于 Calendar 的排除规则

如果要在触发器的基础上，排除一些时间区间不执行任务，就要用到 Quartz 的 Calendar 类（注意不是 JDK 的 Calendar）。可以按年、月、周、日、特定日期、Cron 表达式排除。



调用 Trigger 的 modifiedByCalendar()添加到触发器中，并且调用调度器的 addCalendar()方法注册排除规则。

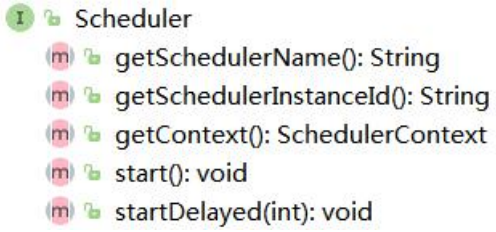
代码示例：standalone 工程：com.gupaoedu.calendar.CalendarDemo

Calendar 名称	用法
BaseCalendar	为高级的 Calendar 实现了基本的功能，实现了 org.quartz.Calendar 接口
AnnualCalendar	排除年中一天或多天
CronCalendar	日历的这种实现排除了由给定的 CronExpression 表达的时间集合。 例如，您可以使用此日历使用表达式“* * 0-7,18-23？* *”每天排除所有营业时间（上午 8 点至下午 5 点）。 如果 CronTrigger 具有给定的 cron 表达式并且与具有相同表达式的 CronCalendar 相关联，则日历将排除触发器包含的所有时间，并且它们将彼此抵消。
DailyCalendar	您可以使用此日历来排除营业时间（上午 8 点 - 5 点）每天。 每个 DailyCalendar 仅允许指定单个时间范围，并且该时间范围可能不会跨越每日边界（即，您不能指定从上午 8 点至凌晨 5 点的时间范围）。 如果属性 invertTimeRange 为 false（默认），则时间范围定义触发器不允许触发的时间范围。 如果 invertTimeRange 为 true，则时间范围被反转 - 也就是排除在定义的时间范围之外的所有时间。
HolidayCalendar	特别的用于从 Trigger 中排除节假日
MonthlyCalendar	排除月份中的指定数天，例如，可用于排除每月的最后一天
WeeklyCalendar	排除星期中的任意周几，例如，可用于排除周末，默认周六和周日

3.6.3 Scheduler

调度器，是 Quartz 的指挥官，由 StdSchedulerFactory 产生。它是单例的。

并且是 Quartz 中最重要的 API，默认是实现类是 StdScheduler，里面包含了一个 QuartzScheduler。QuartzScheduler 里面又包含了一个 QuartzSchedulerThread。



Scheduler 中的方法主要分为三大类：

1）操作调度器本身，例如调度器的启动 start()、调度器的关闭 shutdown()。

2）操作 Trigger，例如 pauseTriggers()、resumeTrigger()。

3）操作 Job，例如 scheduleJob()、unscheduleJob()、rescheduleJob()

这些方法非常重要，可以实现任务的动态调度。

3.6.4 Listener

我们有这么一种需求，在每个任务运行结束之后发送通知给运维管理员。那是不是要在每个任务的最后添加一行代码呢？这种方式对原来的代码造成了入侵，不利于维护。如果代码不是写在任务代码的最后一行，怎么知道任务执行完了呢？或者说，怎么监测到任务的生命周期呢？

观察者模式：定义对象间一种一对多的依赖关系，使得每当一个对象改变状态，则所有依赖它的对象都会得到通知并自动更新。

Quartz 中提供了三种 Listener，监听 Scheduler 的，监听 Trigger 的，监听 Job 的。

只需要创建类实现相应的接口，并在 Scheduler 上注册 Listener，便可实现对核心

对象的监听。

standalone 工程：com.gupaoedu.listener

MyJobListenerTest

MySchedulerListenerTest

MyTriggerListenerTest

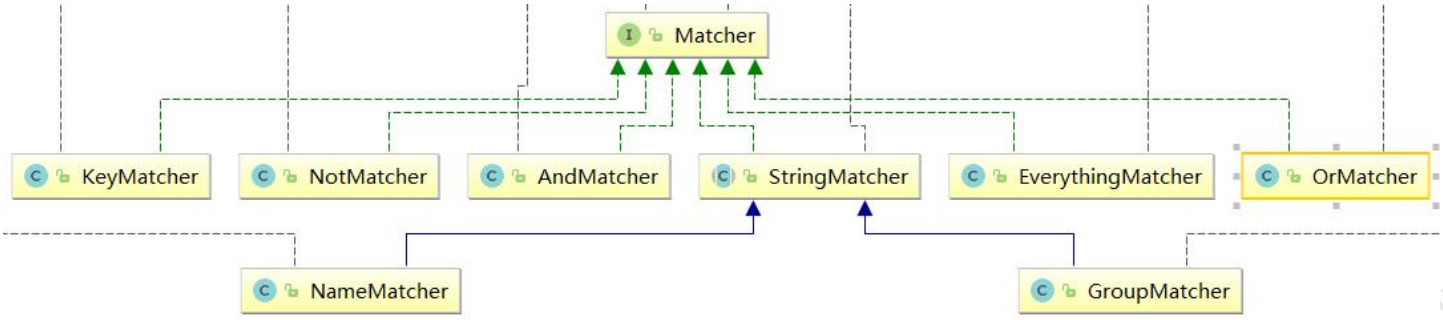
JobListener

四个方法：

方法	作用或执行实际
getName()	返回 JobListener 的名称
jobToBeExecuted()	Scheduler 在 JobDetail 将要被执行时调用这个方法
jobExecutionVetoed()	Scheduler 在 JobDetail 即将被执行，但又被 TriggerListener 否决了时调用这个方法
jobWasExecuted()	Scheduler 在 JobDetail 被执行之后调用这个方法

工具类：ListenerManager，用于添加、获取、移除监听器

工具类：Matcher，主要是基于 groupName 和 keyName 进行匹配。



TriggerListener

方法	作用或执行实际
getName()	返回监听器的名称

triggerFired()	Trigger 被触发，Job 上的 execute() 方法将要被执行时，Scheduler 就调用这个方法
vetoJobExecution()	在 Trigger 触发后，Job 将要被执行时由 Scheduler 调用这个方法。TriggerListener 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 触发而得到执行
triggerMisfired()	Trigger 错过触发时调用
triggerComplete()	Trigger 被触发并且完成了 Job 的执行时，Scheduler 调用这个方法

SchedulerListener

方法比较多，省略。

3.6.5 JobStore

问题：最多可以运行多少个任务（磁盘、内存、线程数）

Jobstore 用来存储任务和触发器相关的信息，例如所有任务的名称、数量、状态等等。Quartz 中有两种存储任务的方式，一种在在内存，一种是在数据库。

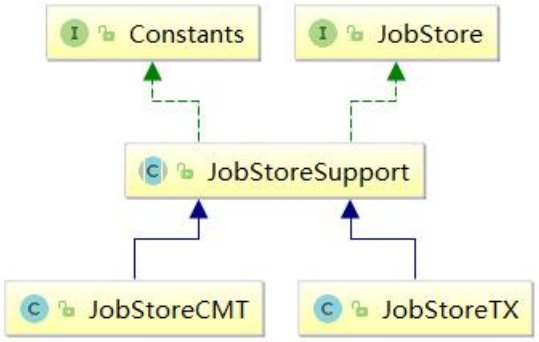
RAMJobStore

Quartz 默认的 JobStore 是 RAMJobstore ,也就是把任务和触发器信息运行的信息存储在内存中，用到了 HashMap、TreeSet、HashSet 等等数据结构。

如果程序崩溃或重启，所有存储在内存中的数据都会丢失。所以我们需要把这些数据持久化到磁盘。

JDBCJobStore

JDBCJobStore 可以通过 JDBC 接口，将任务运行数据保存在数据库中。



JDBC 的实现方式有两种，JobStoreSupport 类的两个子类：

JobStoreTX：在独立的程序中使用，自己管理事务，不参与外部事务。

JobStoreCMT：(Container Managed Transactions (CMT)，如果需要容器管理事务时，使用它。

使用 JDBCJobSotre 时，需要配置数据库信息：

```
org.quartz.jobStore.class:org.quartz.impl.jdbcjobstore.JobStoreTX
org.quartz.jobStore.driverDelegateClass:org.quartz.impl.jdbcjobstore.StdJDBCDelegate
# 使用 quartz.properties，不使用默认配置
org.quartz.jobStore.useProperties:true
#数据库中 quartz 表的表名前缀
org.quartz.jobStore.tablePrefix:QRTZ_
org.quartz.jobStore.dataSource:myDS

#配置数据源
org.quartz.dataSource.myDS.driver:com.mysql.jdbc.Driver
org.quartz.dataSource.myDS.URL:jdbc:mysql://localhost:3306/gupao?useUnicode=true&characterEncoding=utf8
org.quartz.dataSource.myDS.user:root
org.quartz.dataSource.myDS.password:123456
org.quartz.dataSource.myDS.validationQuery=select 0 from dual
```

问题来了？需要建什么表？表里面有什么字段？字段类型和长度是什么？

在官网的 Downloads 链接中，提供了 11 张表的建表语句：

quartz-2.2.3-distribution\quartz-2.2.3\docs\dbTables

2.3 的版本在这个路径下：src\org\quartz\impl\jdbcjobstore

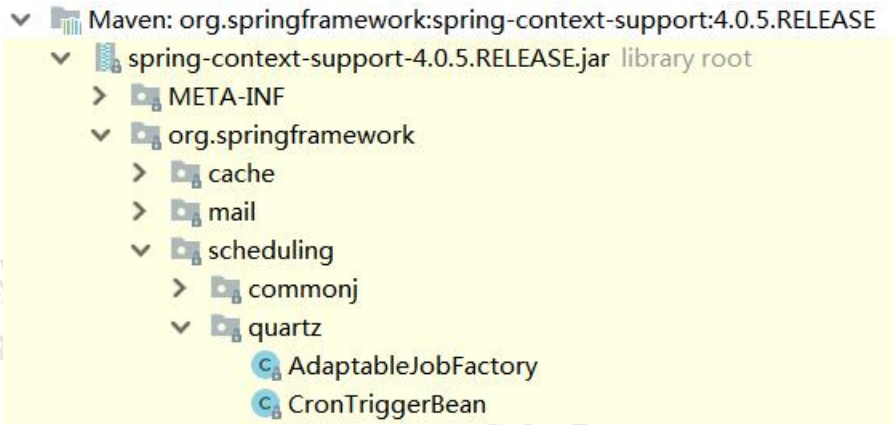
表名与作用：

表名	作用
QRTZ_BLOB_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，以及相关 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 SimpleTrigger 的信息，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

4 Quartz 集成到 Spring

Spring-quartz 工程

Spring 在 spring-context-support.jar 中直接提供了对 Quartz 的支持。



可以在配置文件中把 JobDetail、Trigger、Scheduler 定义成 Bean。

4.1 定义 Job

```
<bean name="myJob1" class="org.springframework.scheduling.quartz.JobDetailFactoryBean">
  <property name="name" value="my_job_1"/>
```

```
    <property name="group" value="my_group"/>
    <property name="jobClass" value="com.gupaoedu.quartz.MyJob1"/>
    <property name="durability" value="true"/>
</bean>
```

4.2 定义 Trigger

```
<bean name="simpleTrigger" class="org.springframework.scheduling.quartz.SimpleTriggerFactoryBean">
    <property name="name" value="my_trigger_1"/>
    <property name="group" value="my_group"/>
    <property name="jobDetail" ref="myJob1"/>
    <property name="startDelay" value="1000"/>
    <property name="repeatInterval" value="5000"/>
    <property name="repeatCount" value="2"/>
</bean>
```

4.3 定义 Scheduler

```
<bean name="scheduler" class="org.springframework.scheduling.quartz.SchedulerFactoryBean">
    <property name="triggers">
        <list>
            <ref bean="simpleTrigger"/>
            <ref bean="cronTrigger"/>
        </list>
    </property>
</bean>
```

既然可以在配置文件配置，当然也可以用@Bean 注解配置。在配置类上加上@Configuration 让 Spring 读取到。

```
public class QuartzConfig {
    @Bean
    public JobDetail printTimeJobDetail(){
        return JobBuilder.newJob(MyJob1.class)
                .withIdentity("gupaoJob")
                .usingJobData("gupao", "职位更好的你")
                .storeDurably()
                .build();
    }
    @Bean
```

```
    public Trigger printTimeJobTrigger() {
        CronScheduleBuilder cronScheduleBuilder = CronScheduleBuilder.cronSchedule("0/5 * * * * ?");
        return TriggerBuilder.newTrigger()
                .forJob(printTimeJobDetail())
                .withIdentity("quartzTaskService")
                .withSchedule(cronScheduleBuilder)
                .build();
    }
}
```

运行 spring-quartz 工程的 com.gupaoedu.quartz.QuartzTest

5 动态调度的实现

springboot-quartz 工程

传统的 Spring 方式集成，由于任务信息全部配置在 xml 文件中，如果需要操作任务或者修改任务运行频率，只能重新编译、打包、部署、重启，如果有紧急问题需要处理，会浪费很多的时间。

有没有可以动态调度任务的方法？比如停止一个 Job？启动一个 Job？修改 Job 的触发频率？

读取配置文件、写入配置文件、重启 Scheduler 或重启应用明显是不可取的。

对于这种频繁变更并且需要实时生效的配置信息，我们可以放到哪里？

ZK、Redis、DB tables。

并且，我们可以提供一个界面，实现对数据表的轻松操作。

5.1 配置管理

这里我们用最简单的数据库的实现。

问题 1：建一张什么样的表？参考 JobDetail 的属性。

```
CREATE TABLE `sys_job` (
```

```
  `id` int(11) NOT NULL AUTO_INCREMENT COMMENT 'ID',
  `job_name` varchar(512) NOT NULL COMMENT '任务名称',
  `job_group` varchar(512) NOT NULL COMMENT '任务组名',
  `job_cron` varchar(512) NOT NULL COMMENT '时间表达式',
  `job_class_path` varchar(1024) NOT NULL COMMENT '类路径,全类型',
  `job_data_map` varchar(1024) DEFAULT NULL COMMENT '传递 map 参数',
  `job_status` int(2) NOT NULL COMMENT '状态:1 启用  0 停用',
  `job_describe` varchar(1024) DEFAULT NULL COMMENT '任务功能描述',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=25 DEFAULT CHARSET=utf8;
```

5.2 数据操作与任务调度

操作数据表非常简单，SSM 增删改查。

但是在修改了表的数据之后，怎么让调度器知道呢？

调度器的接口：Scheduler

在我们的需求中，我们需要做的事情：

1、 新增一个任务

2、 删除一个任务

3、 启动、停止一个任务

4、 修改任务的信息（包括调度规律）

因此可以把相关的操作封装到一个工具类中。com.gupaoedu.demo.util.SchedulerUtil

5.3 前端界面

任务名称（英文）	任务组别	任务表达式	类路径	任务描述	任务状态	操作
test	test	*/1 * * * * ?	com.gupaoedu.demo.task.TestTask1	长沙	已停止	查看 编辑 删除 运行 停止
test2	test	*/3 * * * * ?	com.gupaoedu.demo.task.TestTask2	输出	已停止	查看 编辑 删除 运行 停止
test3	mail	*/15 * * * * ?	com.gupaoedu.demo.task.TestTask3	发送邮件	已停止	查看 编辑 删除 运行 停止

1 到第 1 页 确定 共 3 条 10 条/页

接下来我们有两个问题要解决：

5.4 容器启动与 Service 注入

5.4.1 容器启动

因为任务没有定义在 ApplicationContext.xml 中，而是放到了数据库中，Spring Boot 启动时，怎么读取任务信息？

或者，怎么在 Spring 启动完成的时候做一些事情？

创建一个类，实现 CommandLineRunner 接口，实现 run 方法。

从表中查出状态是 1 的任务，然后构建。

5.4.2 Service 类注入到 Job 中

Spring Bean 如何注入到实现了 Job 接口的类中？

例如在 TestTask3 中，需要注入 ISysJobService，查询数据库发送邮件。

如果没有任何配置，注入会报空指针异常。

原因：

因为定时任务 Job 对象的实例化过程是在 Quartz 中进行的，而 Service Bean 是由 Spring 容器管理的，Quartz 察觉不到 Service Bean 的存在，所以无法将 Service Bean 装配到 Job 对象中。

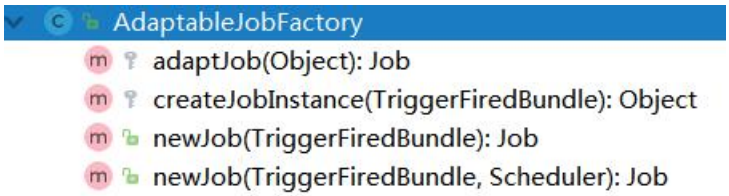
分析：

Quartz 集成到 Spring 中，用到 SchedulerFactoryBean，其实现了 InitializingBean 方法，在唯一的方法 afterPropertiesSet()在 Bean 的属性初始化后调用。

调度器用 AdaptableJobFactory 对 Job 对象进行实例化。所以，如果我们可以把这个 JobFactory 指定为我们自定义的工厂的话，就可以在 Job 实例化完成之后，把 Job 纳入到 Spring 容器中管理。

解决这个问题的步骤：

1、定义一个 AdaptableJobFactory，实现 JobFactory 接口，实现接口定义的 newJob 方法，在这里面返回 Job 实例



2、定义一个 MyJobFactory，继承 AdaptableJobFactory。

使用 Spring 的 AutowireCapableBeanFactory，把 Job 实例注入到容器中。

```
@Component
public class MyJobFactory extends AdaptableJobFactory {
    @Autowired
    private AutowireCapableBeanFactory capableBeanFactory;

    protected Object createJobInstance(TriggerFiredBundle bundle) throws Exception {
        Object jobInstance = super.createJobInstance(bundle);
        capableBeanFactory.autowireBean(jobInstance);

        return jobInstance;
    }
}
```

3、指定 Scheduler 的 JobFactory 为自定义的 JobFactory。

com.gupaoedu.demo.config.InitStartSchedule 中：

```
scheduler.setJobFactory(myJobFactory);
```

考虑这么一种情况：

正在运行的 Quartz 节点挂了，而所有人完全不知情……

6 Quartz 集群部署

springboot-quartz 工程

6.1 为什么需要集群？

1、防止单点故障，减少对业务的影响

2、减少节点的压力，例如在 10 点要触发 1000 个任务，如果有 10 个节点，则每个节点之需要执行 100 个任务

6.2 集群需要解决的问题？

1、任务重跑，因为节点部署的内容是一样的，到 10 点的时候，每个节点都会执行相同的操作，引起数据混乱。比如跑批，绝对不能执行多次。

2、任务漏跑，假如任务是平均分配的，本来应该在某个节点上执行的任务，因为节点故障，一直没有得到执行。

3、水平集群需要注意时间同步问题

4、Quartz 使用的是随机的负载均衡算法，不能指定节点执行

所以必须要有一种共享数据或者通信的机制。在分布式系统的不同节点中，我们可以采用什么样的方式，实现数据共享？

两两通信，或者基于分布式的服务，实现数据共享。

例如：ZK、Redis、DB。

在 Quartz 中，提供了一种简单的方式，基于数据库共享任务执行信息。也就是说，一个节点执行任务的时候，会操作数据库，其他的节点查询数据库，便可以感知到了。

同样的问题：建什么表？哪些字段？依旧使用系统自带的 11 张表。

6.3 集群配置与验证

quartz.properties 配置。

四个配置：集群实例 ID、集群开关、数据库持久化、数据源信息

注意先清空 quartz 所有表、改端口、两个任务频率改成一样

验证 1：先后启动 2 个节点，任务是否重跑

验证 2：停掉一个节点，任务是否漏跑

7 Quartz 调度原理

问题：

1、Job 没有继承 Thread 和实现 Runnable，是怎么被调用的？通过反射还是什么？

2、任务是什么时候被调度的？是谁在监视任务还是监视 Trigger？

3、任务是怎么被调用的？谁执行了任务？

4、任务本身有状态吗？还是触发器有状态？

看源码的入口

```
Scheduler scheduler = factory.getScheduler();
scheduler.scheduleJob(jobDetail, trigger);
scheduler.start();
```

7.1 获取调度器实例

7.1.1 读取配置文件

```
public Scheduler getScheduler() throws SchedulerException {
  if (cfg == null) {
      // 读取 quartz.properties 配置文件
      initialize();
}
  // 这个类是一个 HashMap，用来基于调度器的名称保证调度器的唯一性
  SchedulerRepository schedRep = SchedulerRepository.getInstance();

  Scheduler sched = schedRep.lookup(getSchedulerName());
  // 如果调度器已经存在了
  if (sched != null) {
      // 调度器关闭了，移除
      if (sched.isShutdown()) {
          schedRep.remove(getSchedulerName());
    } else {
          // 返回调度器
          return sched;
    }
}

  // 调度器不存在，初始化
  sched = instantiate();

  return sched;
}
```

instantiate()方法中做了初始化的所有工作：

```
// 存储任务信息的 JobStore
JobStore js = null;
// 创建线程池，默认是 SimpleThreadPool
ThreadPool tp = null;
// 创建调度器
QuartzScheduler qs = null;
// 连接数据库的连接管理器
DBConnectionManager dbMgr = null;
// 自动生成 ID
```

```
// 创建线程执行器，默认为 DefaultThreadExecutor
ThreadExecutor threadExecutor;
```

7.1.2 创建线程池（包工头）

830 行和 839 行 ,创建了一个线程池 默认是配置文件中指定的 SimpleThreadPool。

```
String tpClass = cfg.getStringProperty(PROP_THREAD_POOL_CLASS, SimpleThreadPool.class.getName());
tp = (ThreadPool) loadHelper.loadClass(tpClass).newInstance();
```

SimpleThreadPool 里面维护了三个 list，分别存放所有的工作线程、空闲的工作线程和忙碌的工作线程。我们可以把 SimpleThreadPool 理解为包工头。

```
private List<WorkerThread> workers;
private LinkedList<WorkerThread> availWorkers = new LinkedList<WorkerThread>();
private LinkedList<WorkerThread> busyWorkers = new LinkedList<WorkerThread>();
```

tp 的 runInThread()方法是线程池运行线程的接口方法。参数 Runnable 是执行的任务内容。

取出 WorkerThread 去执行参数里面的 runnable（JobRunShell）。

```
WorkerThread wt = (WorkerThread)availWorkers.removeFirst();
busyWorkers.add(wt);
wt.run(runnable);
```

7.1.3 WorkerThread（工人）

WorkerThread 是 SimpleThreadPool 的内部类，用来执行任务。我们把 WorkerThread 理解为工人。在 WorkerThread 的 run 方法中 执行传入的参数 runnable 任务：

```
runnable.run();
```

7.1.4 创建调度线程（项目经理）

1321 行，创建了调度器 QuartzScheduler：

```
qs = new QuartzScheduler(rsrcs, idleWaitTime, dbFailureRetry);
```

在 QuartzScheduler 的构造函数中，创建了 QuartzSchedulerThread，我们把它理解为项目经理，它会调用包工头的工人资源，给他们安排任务。

并且创建了线程执行器 schedThreadExecutor，执行了这个 QuartzSchedulerThread，也就是调用了它的 run 方法。

```
// 创建一个线程，resouces 里面有线程名称
this.schedThread = new QuartzSchedulerThread(this, resources);
// 线程执行器
ThreadExecutor schedThreadExecutor = resources.getThreadExecutor();
//执行这个线程，也就是调用了线程的 run 方法
schedThreadExecutor.execute(this.schedThread);
```

点开 QuartzSchedulerThread 类，找到 run 方法，这个是 Quartz 任务调度的核心方法：

```
public void run() {
    int acquiresFailed = 0;
    // 检查 scheuler 是否为停止状态
    while (!halted.get()) {
        try {
            // check if we're supposed to pause...
            synchronized (sigLock) {
                // 检查是否为暂停状态
                while (paused && !halted.get()) {
                    try {
                        // wait until togglePause(false) is called...
                        // 暂停的话会尝试去获得信号锁，并 wait 一会
                        sigLock.wait(1000L);
                    } catch (InterruptedException ignore) {
                    }
```

```
                    // reset failure counter when paused, so that we don't
                    // wait again after unpausing
                    acquiresFailed = 0;
                }

                if (halted.get()) {
                    break;
                }
            }

            // wait a bit, if reading from job store is consistently
            // failing (e.g. DB is down or restarting)..
            //  从 JobStore 获取 Job 持续失败，sleep 一下
            if (acquiresFailed > 1) {
                try {
                    long delay = computeDelayForRepeatedErrors(qsRsrcs.getJobStore(), acquiresFailed);
                    Thread.sleep(delay);
                } catch (Exception ignore) {
                }
            }
            //  从线程池获取可用的线程
            int availThreadCount = qsRsrcs.getThreadPool().blockForAvailableThreads();
            if(availThreadCount > 0) { // will always be true, due to semantics of blockForAvailableThreads...

                List<OperableTrigger> triggers;

                long now = System.currentTimeMillis();

                clearSignaledSchedulingChange();
                try {
                    //  获取需要下次执行的 triggers
                    // idleWaitTime：  默认 30s
                    // availThreadCount：获取可用（空闲）的工作线程数量，总会大于 1，因为该方法会一直阻塞，直到有工作线程空闲下来。
                    // maxBatchSize：一次拉取 trigger 的最大数量，默认是 1
                    // batchTimeWindow：时间窗口调节参数，默认是 0
                    // misfireThreshold：  超过这个时间还未触发的 trigger，被认为发生了 misfire，默认 60s
                    //  调度线程一次会拉取 NEXT_FIRETIME 小于（now + idleWaitTime +batchTimeWindow）,大于（now - misfireThreshold）的，min(availThreadCount,maxBatchSize)个 triggers，默认情况下，会拉取未来 30s、过去 60s 之间还未 fire 的 1 个 trigger
                    triggers = qsRsrcs.getJobStore().acquireNextTriggers(
                            now + idleWaitTime, Math.min(availThreadCount, qsRsrcs.getMaxBatchSize()), qsRsrcs.getBatchTimeWindow());
                    //  省略……
```

```
                // set triggers to 'executing'
                List<TriggerFiredResult> bndles = new ArrayList<TriggerFiredResult>();

                boolean goAhead = true;
                synchronized(sigLock) {
                    goAhead = !halted.get();
                }
                if(goAhead) {
                    try {
                        // 触发 Trigger，把 ACQUIRED 状态改成 EXECUTING
                        // 如果这个 trigger 的 NEXTFIRETIME 为空，也就是未来不再触发，就将其状态改为 COMPLETE
                        // 如果 trigger 不允许并发执行(即 Job 的实现类标注了@DisallowConcurrentExecution)，则将状态变为 BLOCKED，否则就将状态改为 WAITING
                        List<TriggerFiredResult> res = qsRsrcs.getJobStore().triggersFired(triggers);
                        // 省略.....
                        continue;
                    }

                }

                // 循环处理 Trigger
                for (int i = 0; i < bndles.size(); i++) {
                    TriggerFiredResult result =  bndles.get(i);
                    TriggerFiredBundle bndle =  result.getTriggerFiredBundle();
                    Exception exception = result.getException();

                    // 省略.....

                    JobRunShell shell = null;
                    try {
                        // 根据 trigger 信息实例化 JobRunShell（implements Runnable），同时依据 JOB_CLASS_NAME 实例化 Job，随后我们将 JobRunShell 实例丢入工作线。
                        shell = qsRsrcs.getJobRunShellFactory().createJobRunShell(bndle);
                        shell.initialize(qs);
                    } catch (SchedulerException se) {
                        qsRsrcs.getJobStore().triggeredJobComplete(triggers.get(i), bndle.getJobDetail(), CompletedExecutionInstruction.SET_ALL_JOB_TRIGGERS_ERROR);
                        continue;
                    }
                    // 执行 JobRunShell 的 run 方法
                    if (qsRsrcs.getThreadPool().runInThread(shell) == false) {
                        // 省略.....
```

JobRunShell 的作用

JobRunShell instances are responsible for providing the 'safe' environment for Job s to run in, and for performing all of the work of executing the Job, catching ANY thrown exceptions, updating the Trigger with the Job's completion code, etc.

A JobRunShell instance is created by a JobRunShellFactory on behalf of the QuartzSchedulerThread which then runs the shell in a thread from the configured ThreadPool when the scheduler determines that a Job has been triggered.

JobRunShell 用来为 Job 提供安全的运行环境的，执行 Job 中所有的作业，捕获运行中的异常，在任务执行完毕的时候更新 Trigger 状态，等等。

JobRunShell 实例是用 JobRunShellFactory 为 QuartzSchedulerThread 创建的，在调度器决定一个 Job 被触发的时候，它从线程池中取出一个线程来执行任务。

7.1.5 线程模型总结

SimpleThreadPool：包工头，管理所有 WorkerThread

WorkerThread：工人，把 Job 包装成 JobRunShell，执行

QuartSchedulerThread：项目经理，获取即将触发的 Trigger，从包工头出拿到 worker，执行 Trigger 绑定的任务

7.2 绑定 JobDetail 和 Trigger

```
// 存储 JobDetail 和 Trigger
resources.getJobStore().storeJobAndTrigger(jobDetail, trig);
// 通知相关的 Listener
notifySchedulerListenersJobAdded(jobDetail);
notifySchedulerThread(trigger.getNextFireTime().getTime());
notifySchedulerListenersSchduled(trigger);
```

7.3 启动调度器

```
// 通知监听器
notifySchedulerListenersStarting();
if (initialStart == null) {
    initialStart = new Date();
    this.resources.getJobStore().schedulerStarted();
    startPlugins();
```

```
} else {
    resources.getJobStore().schedulerResumed();
}
// 通知 QuartzSchedulerThread 不再等待，开始干活
schedThread.togglePause(false);
// 通知监听器
notifySchedulerListenersStarted();
```

7.4 源码总结

getScheduler 方法创建线程池 ThreadPool，创建调度器 QuartzScheduler，创建调度线程 QuartzSchedulerThread，调度线程初始处于暂停状态。

scheduleJob 将任务添加到 JobStore 中。

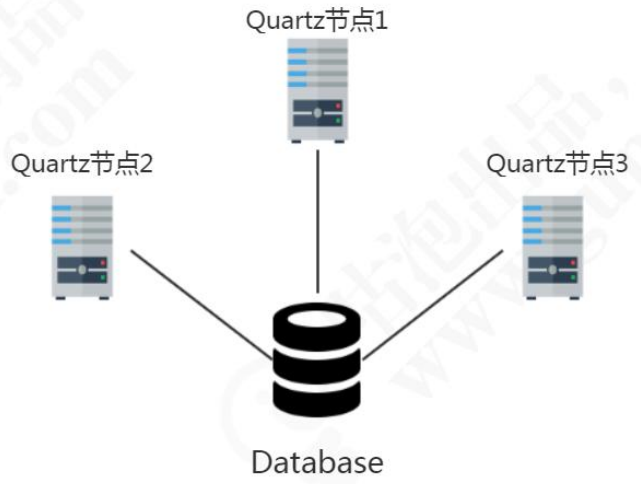
scheduler.start()方法激活调度器，QuartzSchedulerThread 从 timeTrriger 取出待触发的任务，并包装成 TriggerFiredBundle，然后由 JobRunShellFactory 创建 TriggerFiredBundle 的执行线程 JobRunShell，调度执行通过线程池 SimpleThreadPool 去执行 JobRunShell，而 JobRunShell 执行的就是任务类的 execute 方法：job.execute(JobExecutionContext context)。

7.5 集群原理

基于数据库，如何实现任务的不重跑不漏跑？

问题 1：如果任务执行中的资源是“下一个即将触发的任务”，怎么基于数据库实现这个资源的竞争？

问题 2：怎么对数据的行加锁？



QuartzSchedulerThread 第 287 行，获取下一个即将触发的 Trigger

```
triggers = qsRsrcs.getJobStore().acquireNextTriggers(
```

调用 JobStoreSupport 的 acquireNextTriggers()方法，2793 行

调用 JobStoreSupport.executeInNonManagedTXLock()方法，3829 行：

```
return executeInNonManagedTXLock(lockName,
```

尝试获取锁，3843 行：

```
transOwner = getLockHandler().obtainLock(conn, lockName);
```

下面有回滚和释放锁的语句，即使发生异常，锁同样能释放。

调用 DBSemaphore 的 obtainLock()方法，103 行

```
public boolean obtainLock(Connection conn, String lockName)
    throws LockException {
    if (!isLockOwner(lockName)) {
        executeSQL(conn, lockName, expandedSQL, expandedInsertSQL);
```

调用 StdRowLockSemaphore 的 executeSQL()方法，78 行。

最终用 JDBC 执行 SQL，语句内容是 expandedSQL 和 expandedInsertSQL。

```
ps = conn.prepareStatement(expandedSQL);
```

问题：expandedSQL 和 expandedInsertSQL 是一条什么 SQL 语句？似乎我们没有赋值？

在 StdRowLockSemaphore 的构造函数中，把定义的两条 SQL 传进去：

```
public StdRowLockSemaphore() {
  super(DEFAULT_TABLE_PREFIX, null, SELECT_FOR_LOCK, INSERT_LOCK);
}
```

```
public static final String SELECT_FOR_LOCK = "SELECT * FROM "
        + TABLE_PREFIX_SUBST + TABLE_LOCKS + " WHERE " + COL_SCHEDULER_NAME + " = " +
SCHED_NAME_SUBST
        + " AND " + COL_LOCK_NAME + " = ? FOR UPDATE";

  public static final String INSERT_LOCK = "INSERT INTO "
      + TABLE_PREFIX_SUBST + TABLE_LOCKS + "(" + COL_SCHEDULER_NAME + ", " +
COL_LOCK_NAME + ") VALUES ("
      + SCHED_NAME_SUBST + ", ?)";
```

它调用了父类 DBSemaphore 的构造函数：

```
public DBSemaphore(String tablePrefix, String schedName, String defaultSQL, String defaultInsertSQL) {
      this.tablePrefix = tablePrefix;
      this.schedName = schedName;
      setSQL(defaultSQL);
      setInsertSQL(defaultInsertSQL);
}
```

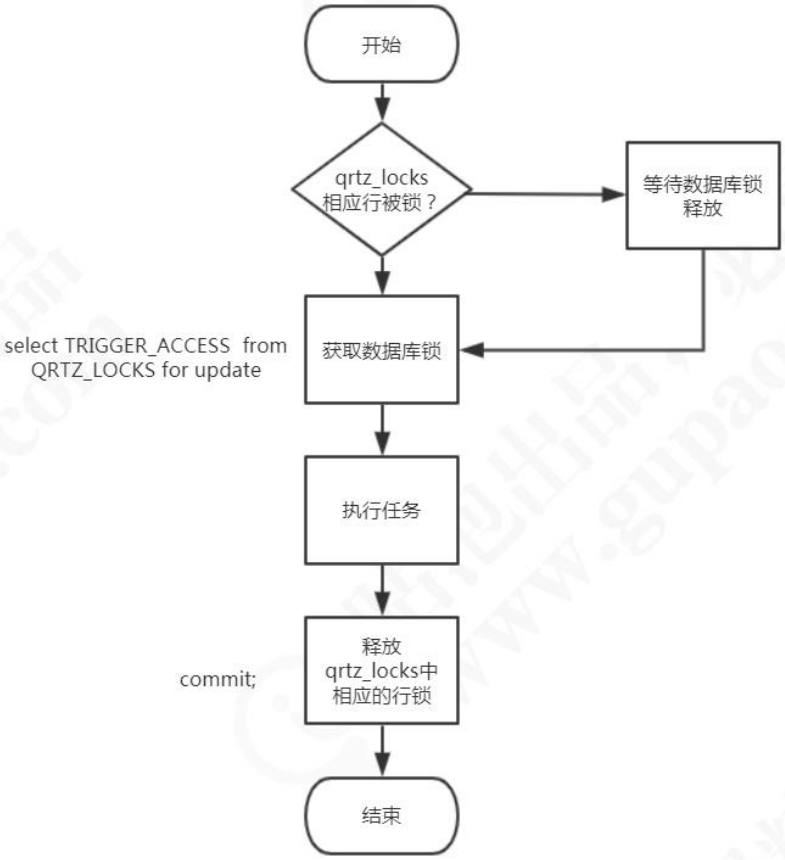
在 setSQL()和 setInsertSQL()中为 expandedSQL 和 expandedInsertSQL 赋值。

执行的 SQL 语句：

```
select * from QRTZ_LOCKS t where t.lock_name='TRIGGER_ACCESS' for update
```

在我们执行官方的建表脚本的时候，QRTZ_LOCKS 表，它会为每个调度器创建两行数据，获取 Trigger 和触发 Trigger 是两把锁：

SCHED_NAME	LOCK_NAME
MyScheduler	STATE_ACCESS
MyScheduler	TRIGGER_ACCESS



0.调度器线程run()

1.获取待触发trigger

1.1数据库LOCKS表TRIGGER_ACCESS行加锁

1.2读取JobDetail信息

1.3读取trigger表中触发器信息并标记为"已获取"

1.4commit事务,释放锁

2.触发trigger

2.1数据库LOCKS表STATE_ACCESS行加锁

2.2确认trigger的状态

2.3读取trigger的JobDetail信息

2.4读取trigger的Calendar信息

2.5更新trigger信息

2.6commit事务,释放锁

3实例化并执行Job

3.1从线程池获取线程执行JobRunShell的run方法

7.6 任务为什么重复执行

在我们的演示过程中，有多个调度器，任务没有重复执行，也就是默认会加锁，什么情况下不会上锁呢？

JobStoreSupport 的 executeInNonManagedTXLock()方法

如果 lockName 为空，则不上锁

```
if (lockName != null) {
            // If we aren't using db locks, then delay getting DB connection
             // until after acquiring the lock since it isn't needed.
            if (getLockHandler().requiresConnection()) {
                conn = getNonManagedTXConnection();
            }

            transOwner = getLockHandler().obtainLock(conn, lockName);
        }

        if (conn == null) {
            conn = getNonManagedTXConnection();
        }
```

而上一步 JobStoreSupport 的 acquireNextTriggers()方法，

1）如果 acquireTriggersWithinLock=true 或者 batchTriggerAcquisitionMaxCount>1 时，locaName 赋值为 LOCK_TRIGGER_ACCESS，此时获取 Trigger 会加锁。

2)否则 ,如果 isAcquireTriggersWithinLock()值是 false 并且 maxCount=1 的话，lockName 赋值为 null，这种情况获取 Trigger 下不加锁。

```
public List<OperableTrigger> acquireNextTriggers(final long noLaterThan, final int maxCount, final long timeWindow)
    throws JobPersistenceException {

    String lockName;
    if(isAcquireTriggersWithinLock() || maxCount > 1) {
        lockName = LOCK_TRIGGER_ACCESS;
    } else {
        lockName = null;
}
```

acquireTriggersWithinLock 变量默认是 false：

```
private boolean acquireTriggersWithinLock = false;
```

maxCount 来自 QuartzSchedulerThread：

```
triggers = qsRsrcs.getJobStore().acquireNextTriggers(
                        now + idleWaitTime, Math.min(availThreadCount, qsRsrcs.getMaxBatchSize()),
qsRsrcs.getBatchTimeWindow());
```

getMaxBatchSize()来自 QuartzSchedulerResources，代表 Scheduler 一次拉取 trigger 的最大数量，默认是 1：

```
private int maxBatchSize = 1;
```

这个值可以通过参数修改，代表允许调度程序节点一次获取（用于触发）的触发器的最大数量，默认值是 1。

```
org.quartz.scheduler.batchTriggerAcquisitionMaxCount=1
```

根据以上两个默认值，理论上在获取 Trigger 的时候不会上锁，但是实际上为什么没有出现频繁的重复执行问题？因为每个调度器的线程持有锁的时间太短了，单机的测试无法体现，而在高并发的情况下，有可能会出现这个问题。

QuartzSchedulerThread 的 triggersFired()方法：

```
List<TriggerFiredResult> res = qsRsrcs.getJobStore().triggersFired(triggers);
```

调用了 JobStoreSupport 的 triggersFired()方法，接着又调用了一个 triggerFired triggerFired(Connection conn, OperableTrigger trigger)方法：

如果 Trigger 的状态不是 ACQUIRED，也就是说被其他的线程 fire 了，返回空。但是这种乐观锁的检查在高并发下难免会出现 ABA 的问题，比如线程 A 拿到的时候还是

ACQUIRED 状态，但是刚准备执行的时候已经变成了 EXECUTING 状态，这个时候就会出现重复执行的问题。

```
if (!state.equals(STATE_ACQUIRED)) {
    return null;
}
```

总结，如果：

如果设置的数量为 1（默认值），并且使用 JDBC JobStore（RAMJobStore 不支持分布式，只有一个调度器实例，所以不加锁），则属性 org.quartz.jobStore.acquireTriggersWithinLock 应设置为 true。否则不加锁可能会导致任务重复执行。

```
org.quartz.scheduler.batchTriggerAcquisitionMaxCount=1
org.quartz.jobStore.acquireTriggersWithinLock=true
```

作者：咕泡学院-青山

最后更新时间：2019 年 10 月 27 日