链滴

定时任务系列之 Quartz 框架

作者：BigBigBigPeach

原文链接：https://ld246.com/article/1567607466715

来源网站：链滴

前言

我们前面提到了很多定时任务的简单实现了，大家可以点击下面的链接查看相关内容。

- 定时任务系列之crontab&Timer实现
- 定时任务系列之ScheduledExecutorService
- 定时任务系列之Spring Scheduler

这一篇我们看一个比较成熟的框架——Quartz。

使用

首先需要引入相关依赖

```
<!-- 引入依赖 -->
<dependency>
   <groupId>org.quartz-scheduler</groupId>
   <artifactId>quartz</artifactId>
   <version>2.2.1</version>
</dependency>
<dependency>
   <groupId>org.quartz-scheduler</groupId>
   <artifactId>quartz-jobs</artifactId>
   <version>2.2.1</version>
</dependency>
```

在xml文件中配置触发器、调度器等参数。

```
<!-- 调度器 -->
<bean id="quartzScheduler" class="org.springframework.scheduling.quartz.SchedulerFactory
ean">
      <!--数据库连接池使用自己的连接池-->
      <property name="dataSource" ref="multipleDataSource"/>
      <property name="quartzProperties">
         <props>
            <!--实例名称-->
            <prop key="org.quartz.scheduler.instanceName">quartz-cluster</prop>
            <!--自动生成实例id-->
            <prop key="org.quartz.scheduler.instanceId">AUTO</prop>
            <!-- 线程池配置 -->
            <prop key="org.quartz.threadPool.class">org.quartz.simpl.SimpleThreadPool</pro
>
            <!--线程池的线程数量-->
            <prop key="org.quartz.threadPool.threadCount">10</prop>
            <!--线程优先级，10最大，1最小-->
            <prop key="org.quartz.threadPool.threadPriority">5</prop>
            <!-- JobStore 配置 -->
            <prop key="org.quartz.jobStore.class">org.quartz.impl.jdbcjobstore.JobStoreTX</p
op>
            <!-- 集群配置！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！
-->
            <prop key="org.quartz.jobStore.isClustered">true</prop>
```

```
            <!--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-->
            <prop key="org.quartz.jobStore.clusterCheckinInterval">10000</prop>
            <!--jobStore处理未按时触发的Job的数量-->
            <prop key="org.quartz.jobStore.maxMisfiresToHandleAtATime">2</prop>
            <!--超时时间10min，如果超过则认为“失误”,忽略过这个任务-->
            <prop key="org.quartz.jobStore.misfireThreshold">600000</prop>
            <!--不检查更新-->
            <prop key="org.quartz.scheduler.skipUpdateCheck">true</prop>
            <!--table名称的前缀-->
            <prop key="org.quartz.jobStore.tablePrefix">QRTZ_</prop>
        </props>
    </property>
    <!--调度器的name-->
    <property name="schedulerName" value="clusterScheduler"/>
    <!--启动延时-->
    <property name="startupDelay" value="10"/>
    <!-- 通过applicationContextSchedulerContextKey属性配置spring上下文 -->
    <property name="applicationContextSchedulerContextKey" value="applicationContext"
>
    <!--是否重写数据库已存在的job，如果这个覆盖配置为false，quratz启动以后将以数据库的数为准，配置文件的修改不起作用-->
    <property name="overwriteExistingJobs" value="true"/>
    <!--线程池-->
    <property name="taskExecutor" ref="taskExecutor"/>
    <property name="autoStartup" value="true"/>
    <!--<property name="jobDetails" ref="jobDetail"/>-->
    <property name="triggers">
        <list>
            <ref bean="testRemindJobTrigger"/>
        </list>
    </property>
</bean>

<!--触发器-->
<bean id="testJobTrigger" class="org.springframework.scheduling.quartz.CronTriggerFactryBean">
    <!--corn表达式，表示每天的8点到20点，在整点的时候执行一次-->
    <property name="cronExpression" value="0 0 8,9,10,11,12,13,14,15,16,17,18,19,20 * * ?"
>
    <!--<property name="cronExpression" value="0/2 * * * * ? " />-->
    <property name="jobDetail" ref="testRemindJob"/>
</bean>

<!-- Job类 -->
<bean id="testRemindJob" class="org.springframework.scheduling.quartz.JobDetailFactorBean">
    <!--job类-->
    <property name="jobClass" value="com.test.timetask.job.testRemindJob"/>
    <!--程序中断，重启会重新执行-->
```

```
        <property name="requestsRecovery" value="true"/>
        <property name="group" value="remind"/>
        <!--设置job持久化-->
        <property name="durability" value="true"/>
    </bean>
```

那么我们真正的实现类应该怎么写呢?

```
@Component(value = "RemindJob")
// 继承QuartzJobBean
public class RemindJob extends QuartzJobBean {

    // 实现如下方法
    @Override
    protected final void executeInternal(JobExecutionContext jobExecutionContext) throws Job
xecutionException {
        doRemind();
    }
}
```

优点

- 相比较于shell + cron，quartz的方式不需要其他角色人员介入。
- 可以配置集群。
- 可以任务持久化。任务数据存在数据库中，如果服务器挂了，重启的时候重新加载数据库的任务数，并判断执行状态如果是未执行且已超过了执行的时间就立即执行一下。

缺点

- 调度逻辑是在每个服务器上执行的，如果定时任务多，那么这个调度占用的资源也不容小觑。
- 每个节点都需要进行数据库资源的竞争（必须同一个实例），才能得到执行机会。
- 上任务简单，下任务难。quartz的xml配置中如果设置了持久化，那么只是删除配置的话，定时任还是会执行。这个时候还是需要删除相关数据库记录。而且数据库表中有外键...所以删除的时候需要顺序删除。
- 对于各个节点的时间同步依赖。

源码分析

```
// org.springframework.scheduling.quartz.SchedulerFactoryBean spring初始化事件
@Override
public void afterPropertiesSet() throws Exception {
    if (this.dataSource == null && this.nonTransactionalDataSource != null) {
        this.dataSource = this.nonTransactionalDataSource;
    }

    if (this.applicationContext != null && this.resourceLoader == null) {
        this.resourceLoader = this.applicationContext;
    }

    // Initialize the Scheduler instance...
```

```
    this.scheduler = prepareScheduler(prepareSchedulerFactory());
    try {
        // 注册监听器 & 触发器 & 任务
        registerListeners();
        registerJobsAndTriggers();
    }
    catch (Exception ex) {
        try {
            this.scheduler.shutdown(true);
        }
        catch (Exception ex2) {
            logger.debug("Scheduler shutdown exception after registration failure", ex2);
        }
        throw ex;
    }
}


/**
 * Register jobs and triggers (within a transaction, if possible).----------------------------------
-------------------------------------
 */
protected void registerJobsAndTriggers() throws SchedulerException {
    TransactionStatus transactionStatus = null;
    if (this.transactionManager != null) {
        transactionStatus = this.transactionManager.getTransaction(new DefaultTransactionDefin
tion());
    }

    try {
        // 查看本地是不是有配置任务
        if (this.jobSchedulingDataLocations != null) {
            ClassLoadHelper clh = new ResourceLoaderClassLoadHelper(this.resourceLoader);
            clh.initialize();
            XMLSchedulingDataProcessor dataProcessor = new XMLSchedulingDataProcessor(clh)

            for (String location : this.jobSchedulingDataLocations) {
                // 处理本地的任务 为其绑定触发器等操作
                dataProcessor.processFileAndScheduleJobs(location, getScheduler());
            }
        }

        // Register JobDetails.
        if (this.jobDetails != null) {
            for (JobDetail jobDetail : this.jobDetails) {
                addJobToScheduler(jobDetail);
            }
        }
        else {
            // Create empty list for easier checks when registering triggers.
            this.jobDetails = new LinkedList<>();
        }

        // Register Calendars.
```

```
        if (this.calendars != null) {
            // 时间
            for (String calendarName : this.calendars.keySet()) {
                Calendar calendar = this.calendars.get(calendarName);
                getScheduler().addCalendar(calendarName, calendar, true, true);
            }
        }

        // Register Triggers.
        if (this.triggers != null) {
            for (Trigger trigger : this.triggers) {
                addTriggerToScheduler(trigger);
            }
        }
    }

    catch (Throwable ex) {
        if (transactionStatus != null) {
            try {
                this.transactionManager.rollback(transactionStatus);
            }
            catch (TransactionException tex) {
                logger.error("Job registration exception overridden by rollback exception", ex);
                throw tex;
            }
        }
        if (ex instanceof SchedulerException) {
            throw (SchedulerException) ex;
        }
        if (ex instanceof Exception) {
            throw new SchedulerException("Registration of jobs and triggers failed: " + ex.getMes
age(), ex);
        }
        throw new SchedulerException("Registration of jobs and triggers failed: " + ex.getMessa
e());
    }

    if (transactionStatus != null) {
        this.transactionManager.commit(transactionStatus);
    }
}


/**
 * Process the xml file in the given location, and schedule all of the
 * jobs defined within it.
 *
 * @param fileName meta data file name.
 */
public void processFileAndScheduleJobs(String fileName, String systemId, Scheduler sched) t
rows Exception {
    // 处理xml
    processFile(fileName, systemId);
    // 处理分组等，检查是否要执行任务 还是 直接删除任务
```

```
    executePreProcessCommands(sched);
    // 添加调度任务
    scheduleJobs(sched);
}


/**
 * Schedules the given sets of jobs and triggers.
 *
 * @param sched 负责注册后在trigger触发时，调用相关jobDetail
 *              job scheduler.
 * @throws SchedulerException if the Job or Trigger cannot be added to the Scheduler, or
 *                            there is an internal Scheduler error.
 */
@SuppressWarnings("ConstantConditions")
protected void scheduleJobs(Scheduler sched)
        throws SchedulerException {

    // 拿到加载的JobDetails & Triggers
    List<JobDetail> jobs = new LinkedList<JobDetail>(getLoadedJobs());
    List<MutableTrigger> triggers = new LinkedList<MutableTrigger>(getLoadedTriggers());

    log.info("Adding " + jobs.size() + " jobs, " + triggers.size() + " triggers.");

    // 构建Job对应的triggers的映射map
    Map<JobKey, List<MutableTrigger>> triggersByFQJobName = buildTriggersByFQJobNam
Map(triggers);

    // add each job, and it's associated triggers
    Iterator<JobDetail> itr = jobs.iterator();
    while (itr.hasNext()) {
        JobDetail detail = itr.next();
        // 拿出来先删掉...
        itr.remove(); // remove jobs as we handle them...

        JobDetail dupeJ = null;
        try {
            // The existing job could have been deleted, and Quartz API doesn't allow us to query
his without
            // loading the job class, so use try/catch to handle it.
            // 从调度器拿到相关的JobDetail【src = 数据库】
            dupeJ = sched.getJobDetail(detail.getKey());
        } catch (JobPersistenceException e) {
            if (e.getCause() instanceof ClassNotFoundException && isOverWriteExistingData()) {
                // We are going to replace jobDetail anyway, so just delete it first.
                log.info("Removing job: " + detail.getKey());
                sched.deleteJob(detail.getKey());
            } else {
                throw e;
            }
        }

        // 检查条件是否正常 是否进行覆盖 & 是否忽略重复
        if ((dupeJ != null)) {
```

```
            if (!isOverWriteExistingData() && isIgnoreDuplicates()) {
                log.info("Not overwriting existing job: " + dupeJ.getKey());
                continue; // just ignore the entry
            }
            if (!isOverWriteExistingData() && !isIgnoreDuplicates()) {
                throw new ObjectAlreadyExistsException(detail);
            }
        }

        // 这两个日志就说明了要干啥
        if (dupeJ != null) {
            // 替换?
            log.info("Replacing job: " + detail.getKey());
        } else {
            // 添加
            log.info("Adding job: " + detail.getKey());
        }

        // 拿到job相关触发器
        List<MutableTrigger> triggersOfJob = triggersByFQJobName.get(detail.getKey());

        // 如果不是持久化的任务 并且触发器是空的
        if (!detail.isDurable() && (triggersOfJob == null || triggersOfJob.size() == 0)) {
            // 哦，连dupeJ也是null，那就没办法了，只能报错了
            if (dupeJ == null) {
                throw new SchedulerException(
                        "A new job defined without any triggers must be durable: " +
                                detail.getKey());
            }

            // 如果dupeJ是持久化的，从调度器中能不到trigger，也会报错
            if ((dupeJ.isDurable() &&
                    (sched.getTriggersOfJob(
                            detail.getKey()).size() == 0))) {
                throw new SchedulerException(
                        "Can't change existing durable job without triggers to non-durable: " +
                                detail.getKey());
            }
        }

        // 这个判断 决定了 在调度前是否需要存储非持久化任务。
        if (dupeJ != null || detail.isDurable()) {
            if (triggersOfJob != null && triggersOfJob.size() > 0)
                // add the job regardless is durable or not b/c we have trigger to add
                sched.addJob(detail, true, true);
            else
                // 非持久化任务调用此分支，则会报错  -- add the job only if a replacement or durable,
else exception will throw!
                sched.addJob(detail, true, false);
        } else {
            boolean addJobWithFirstSchedule = true;

            // Add triggers related to the job...
            for (MutableTrigger trigger : triggersOfJob) {
```

```
            // remove triggers as we handle them...
            triggers.remove(trigger);

            if (trigger.getStartTime() == null) {
                trigger.setStartTime(new Date());
            }

            // 跟处理任务一样，也是拿数据库的出来，看是不是要求持久化，是不是需要进行替换，
不是能忽略重复任务
            Trigger dupeT = sched.getTrigger(trigger.getKey());
            if (dupeT != null) {
                if (isOverWriteExistingData()) {
                    if (log.isDebugEnabled()) {
                        log.debug(
                                "Rescheduling job: " + trigger.getJobKey() + " with updated trigger: " +
rigger.getKey());
                    }
                } else if (isIgnoreDuplicates()) {
                    log.info("Not overwriting existing trigger: " + dupeT.getKey());
                    continue; // just ignore the trigger (and possibly job)
                } else {
                    throw new ObjectAlreadyExistsException(trigger);
                }

                if (!dupeT.getJobKey().equals(trigger.getJobKey())) {
                    log.warn("Possibly duplicately named ({}) triggers in jobs xml file! ", trigger.get
ey());
                }
                // 如果没有问题，可以覆盖的话，除旧迎新
                sched.rescheduleJob(trigger.getKey(), trigger);
            } else {
                if (log.isDebugEnabled()) {
                    log.debug(
                            "Scheduling job: " + trigger.getJobKey() + " with trigger: " + trigger.getKe
());
                }

                try {
                    // 添加调度任务，添加第一个触发器
                    if (addJobWithFirstSchedule) {
                        // add the job if it's not in yet...
                        // 会将之放入数据库
                        sched.scheduleJob(detail, trigger);
                        addJobWithFirstSchedule = false;
                    } else {
                        sched.scheduleJob(trigger);
                    }
                } catch (ObjectAlreadyExistsException e) {
                    if (log.isDebugEnabled()) {
                        log.debug(
                                "Adding trigger: " + trigger.getKey() + " for job: " + detail.getKey() +
                                        " failed because the trigger already existed.  " +
                                        "This is likely due to a race condition between multiple instances "
+
```

```
                                "in the cluster.  Will try to reschedule instead.");
                        }

                        // Let's try one more time as reschedule.
                        // 重试调度
                        sched.rescheduleJob(trigger.getKey(), trigger);
                    }
                }
            }
        }
    }

    // add triggers that weren't associated with a new job... (those we already handled were re
oved above)
    // 处理剩下的没有和任务关联的触发器 和上面套路一样
    for (MutableTrigger trigger : triggers) {

        if (trigger.getStartTime() == null) {
            trigger.setStartTime(new Date());
        }

        Trigger dupeT = sched.getTrigger(trigger.getKey());
        if (dupeT != null) {
            if (isOverWriteExistingData()) {
                if (log.isDebugEnabled()) {
                    log.debug(
                            "Rescheduling job: " + trigger.getJobKey() + " with updated trigger: " + trig
er.getKey());
                }
            } else if (isIgnoreDuplicates()) {
                log.info("Not overwriting existing trigger: " + dupeT.getKey());
                continue; // just ignore the trigger
            } else {
                throw new ObjectAlreadyExistsException(trigger);
            }

            if (!dupeT.getJobKey().equals(trigger.getJobKey())) {
                log.warn("Possibly duplicately named ({}) triggers in jobs xml file! ", trigger.getKey())

            }

            sched.rescheduleJob(trigger.getKey(), trigger);
        } else {
            if (log.isDebugEnabled()) {
                log.debug(
                        "Scheduling job: " + trigger.getJobKey() + " with trigger: " + trigger.getKey());
            }

            try {
                sched.scheduleJob(trigger);
            } catch (ObjectAlreadyExistsException e) {
                if (log.isDebugEnabled()) {
                    log.debug(
                            "Adding trigger: " + trigger.getKey() + " for job: " + trigger.getJobKey() +
```

```
                        " failed because the trigger already existed.  " +
                        "This is likely due to a race condition between multiple instances " +
                        "in the cluster.  Will try to reschedule instead.");
                }

                // Let's rescheduleJob one more time.
                sched.rescheduleJob(trigger.getKey(), trigger);
            }
        }
    }
}


// -----------------------------------------------------------------------------------------开始调度----
-----------------------------------------------------------------------------------
/**
 * Start the Quartz Scheduler, respecting the "startupDelay" setting.
 * @param scheduler the Scheduler to start
 * @param startupDelay the number of seconds to wait before starting
 * the Scheduler asynchronously
 */
protected void startScheduler(final Scheduler scheduler, final int startupDelay) throws Schedu
erException {
    // 是否延迟启动 可配置参数
    if (startupDelay <= 0) {
        logger.info("Starting Quartz Scheduler now");
        scheduler.start();
    }
    else {
        if (logger.isInfoEnabled()) {
            logger.info("Will start Quartz Scheduler [" + scheduler.getSchedulerName() +
                    "] in " + startupDelay + " seconds");
        }
        // Not using the Quartz startDelayed method since we explicitly want a daemon
        // thread here, not keeping the JVM alive in case of all other threads ending.
        Thread schedulerThread = new Thread() {
            @Override
            public void run() {
                try {
                    TimeUnit.SECONDS.sleep(startupDelay);
                }
                catch (InterruptedException ex) {
                    Thread.currentThread().interrupt();
                    // simply proceed
                }
                if (logger.isInfoEnabled()) {
                    logger.info("Starting Quartz Scheduler now, after delay of " + startupDelay + " se
onds");
                }
                try {
                    // 重点----启动调度任务
                    scheduler.start();
                }
                catch (SchedulerException ex) {
```

```
                    throw new SchedulingException("Could not start Quartz Scheduler after delay", e
);
                }
            }
        };
        // 设置并启动调度线程
        schedulerThread.setName("Quartz Scheduler [" + scheduler.getSchedulerName() + "]");
        schedulerThread.setDaemon(true);
        schedulerThread.start();
    }
}

/**
 * <p>
 * Starts the <code>QuartzScheduler</code>'s threads that fire <code>{@link org.quartz.Tri
ger}s</code>.
 * </p>
 *
 * <p>
 * All <code>{@link org.quartz.Trigger}s</code> that have misfired will
 * be passed to the appropriate TriggerListener(s).
 * </p>
 */
public void start() throws SchedulerException {
    if (shuttingDown|| closed) {
        throw new SchedulerException(
                "The Scheduler cannot be restarted after shutdown() has been called.");
    }

    // QTZ-212 : calling new schedulerStarting() method on the listeners
    // right after entering start()
    notifySchedulerListenersStarting();

    if (initialStart == null) {
        initialStart = new Date();
        this.resources.getJobStore().schedulerStarted();
        startPlugins();
    } else {
        resources.getJobStore().schedulerResumed();
    }

    schedThread.togglePause(false);

    getLog().info(
            "Scheduler " + resources.getUniqueIdentifier() + " started.");

    notifySchedulerListenersStarted();
}


// ----------------------------------------------------------------------------真·任务执行-
----------------------------------------------------------------------------
/**
 * <p>
```

```
 * The main processing loop of the <code>QuartzSchedulerThread</code>.
 * </p>
 */
@Override
public void run() {
    boolean lastAcquireFailed = false;

    while (!halted.get()) {
        try {
            // 检查一下是不是需要暂停
            synchronized (sigLock) {
                while (paused && !halted.get()) {
                    try {
                        // 那就暂停一下
                        sigLock.wait(1000L);
                    } catch (InterruptedException ignore) {
                    }
                }
                // 检查是否可以执行
                if (halted.get()) {
                    break;
                }
            }

            int availThreadCount = qsRsrcs.getThreadPool().blockForAvailableThreads();
            // will always be true, due to semantics of blockForAvailableThreads...
            if(availThreadCount > 0) {
                List<OperableTrigger> triggers = null;
                long now = System.currentTimeMillis();
                // 处理signaled、signaledNextFireTime等值
                clearSignaledSchedulingChange();
                try {
                    // 获取触发器
                    triggers = qsRsrcs.getJobStore().acquireNextTriggers(
                            now + idleWaitTime, Math.min(availThreadCount, qsRsrcs.getMaxBatchSize()
, qsRsrcs.getBatchTimeWindow());
                    lastAcquireFailed = false;
                    if (log.isDebugEnabled())
                        log.debug("batch acquisition of " + (triggers == null ? 0 : triggers.size()) + " tri
gers");
                } catch (JobPersistenceException jpe) {
                    if(!lastAcquireFailed) {
                        qs.notifySchedulerListenersError(
                            "An error occurred while scanning for the next triggers to fire.",
                            jpe);
                    }
                    lastAcquireFailed = true;
                    continue;
                } catch (RuntimeException e) {
                    if(!lastAcquireFailed) {
                        getLog().error("quartzSchedulerThreadLoop: RuntimeException "
                                +e.getMessage(), e);
                    }
                    lastAcquireFailed = true;
```

```
                continue;
            }

            if (triggers != null && !triggers.isEmpty()) {
                now = System.currentTimeMillis();
                long triggerTime = triggers.get(0).getNextFireTime().getTime();
                long timeUntilTrigger = triggerTime - now;
                while(timeUntilTrigger > 2) {
                    synchronized (sigLock) {
                        if (halted.get()) {
                            break;
                        }
                        // 看看是否需要一个新的触发器。里面对计划的变更做了判断处理。
                        if (!isCandidateNewTimeEarlierWithinReason(triggerTime, false)) {
                            try {
                                // 可能在同步上阻塞了很长时间，所以我们必须重新计算
                                now = System.currentTimeMillis();
                                timeUntilTrigger = triggerTime - now;
                                if(timeUntilTrigger >= 1)
                                    sigLock.wait(timeUntilTrigger);
                            } catch (InterruptedException ignore) {
                            }
                        }
                    }
                    // 如果调度变更 释放触发器，不再执行任务。
                    if(releaseIfScheduleChangedSignificantly(triggers, triggerTime)) {
                        break;
                    }
                    now = System.currentTimeMillis();
                    timeUntilTrigger = triggerTime - now;
                }

                // this happens if releaseIfScheduleChangedSignificantly decided to release trigg
rs
                if(triggers.isEmpty())
                    continue;

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

                boolean goAhead = true;
                synchronized(sigLock) {
                    goAhead = !halted.get();
                }
                if(goAhead) {
                    try {
                        List<TriggerFiredResult> res = qsRsrcs.getJobStore().triggersFired(triggers);
                        if(res != null)
                            bndles = res;
                    } catch (SchedulerException se) {
                        qs.notifySchedulerListenersError(
                                "An error occurred while firing triggers '"
                                        + triggers + "'", se);
                        //QTZ-179 : a problem occurred interacting with the triggers from the db
```

```
                    //we release them and loop again
                    for (int i = 0; i < triggers.size(); i++) {
                        qsRsrcs.getJobStore().releaseAcquiredTrigger(triggers.get(i));
                    }
                    continue;
                }

            }

            // 遍历触发器
            for (int i = 0; i < bndles.size(); i++) {
                TriggerFiredResult result =  bndles.get(i);
                TriggerFiredBundle bndle =  result.getTriggerFiredBundle();
                Exception exception = result.getException();
                // 出现异常
                if (exception instanceof RuntimeException) {
                    getLog().error("RuntimeException while firing trigger " + triggers.get(i), exce
tion);
                    qsRsrcs.getJobStore().releaseAcquiredTrigger(triggers.get(i));
                    continue;
                }

                // it's possible to get 'null' if the triggers was paused,
                // blocked, or other similar occurrences that prevent it being
                // fired at this time...  or if the scheduler was shutdown (halted)
                if (bndle == null) {
                    qsRsrcs.getJobStore().releaseAcquiredTrigger(triggers.get(i));
                    continue;
                }

                JobRunShell shell = null;
                try {
                    // 创建执行类
                    shell = qsRsrcs.getJobRunShellFactory().createJobRunShell(bndle);
                    shell.initialize(qs);
                } catch (SchedulerException se) {
                    qsRsrcs.getJobStore().triggeredJobComplete(triggers.get(i), bndle.getJobDet
il(), CompletedExecutionInstruction.SET_ALL_JOB_TRIGGERS_ERROR);
                    continue;
                }

                if (qsRsrcs.getThreadPool().runInThread(shell) == false) {
                    // this case should never happen, as it is indicative of the
                    // scheduler being shutdown or a bug in the thread pool or
                    // a thread pool being used concurrently - which the docs
                    // say not to do...
                    getLog().error("ThreadPool.runInThread() return false!");
                    // 触发执行
                    qsRsrcs.getJobStore().triggeredJobComplete(triggers.get(i), bndle.getJobDet
il(), CompletedExecutionInstruction.SET_ALL_JOB_TRIGGERS_ERROR);
                }

            }
```

```
                continue; // while (!halted)
            }
        } else { // if(availThreadCount > 0)
            // should never happen, if threadPool.blockForAvailableThreads() follows contract
            continue; // while (!halted)
        }

        long now = System.currentTimeMillis();
        long waitTime = now + getRandomizedIdleWaitTime();
        long timeUntilContinue = waitTime - now;
        synchronized(sigLock) {
            try {
              if(!halted.get()) {
                // 计算等待时间，进行等待。
                if (!isScheduleChanged()) {
                  sigLock.wait(timeUntilContinue);
                }
              }
            } catch (InterruptedException ignore) {
            }
        }

      } catch(RuntimeException re) {
          getLog().error("Runtime error occurred in main trigger firing loop.", re);
      }
    } // while (!halted)

    // drop references to scheduler stuff to aid garbage collection...
    // 为了GC
    qs = null;
    qsRsrcs = null;
}
```

小结

定时任务系列到此先告一段落吧。当然还有一个好的开源框架XXL-JOB（调度逻辑由quartz实现），后有机会再讲吧。大家可以看一下xxl-job文档

唉 看世界杯看的生气....