**Timer**

## 一、Timer简介

**1 在java中一个完整定时任务需要由Timer、TimerTask两个类来配合完成。**

API中是这样定义他们的，Timer：一种工具，线程用其安排以后在后台线程中执行的任务。可安排任务执行一次，或者定期重复执行。由TimerTask：Timer安排为一次执行或重复执行的任务。我们可以这样理解Timer是一种定时器工具，用来在一个后台线程计划执行指定任务，而TimerTask一个抽象类，它的子类代表一个可以被Timer计划的任务。

**2 Timer类**

在工具类Timer中，提供了四个构造方法，每个构造方法都启动了计时器线程，同时Timer类可以保证多个线程可以共享单个Timer对象而无需进行外部同步，所以Timer类是线程安全的。但是由于每一个Timer对象对应的是单个后台线程，用于顺序执行所有的计时器任务，一般情况下我们的线程任务执行所消耗的时间应该非常短，但是由于特殊情况导致某个定时器任务执行的时间太长，那么他就会“独占”计时器的任务执行线程，其后的所有线程都必须等待它执行完，这就会延迟后续任务的执行，使这些任务堆积在一起，具体情况我们后面分析。

当程序初始化完成Timer后，定时任务就会按照我们设定的时间去执行，Timer提供了schedule方法，该方法有多中重载方式来适应不同的情况，如下：

schedule(TimerTask task, Date time)：安排在指定的时间执行指定的任务。

schedule(TimerTask task, Date firstTime, long period) ：安排指定的任务在指定的时间开始进行重复的固定延迟执行。

schedule(TimerTask task, long delay) ：安排在指定延迟后执行指定的任务。

schedule(TimerTask task, long delay, long period) ：安排指定的任务从指定的延迟后开始进行重复的固定延迟执行。

同时也重载了scheduleAtFixedRate方法，scheduleAtFixedRate方法与schedule相同，只不过他们的侧重点不同，区别后面分析。

scheduleAtFixedRate(TimerTask task, Date firstTime, long period)：安排指定的任务在指定的时间开始进行重复的固定速率执行。

scheduleAtFixedRate(TimerTask task, long delay, long period)：安排指定的任务在指定的延迟后开始进行重复的固定速率执行。

**3 TimerTask**

TimerTask类是一个抽象类，由Timer 安排为一次执行或重复执行的任务。它有一个抽象方法run()方法，该方法用于执行相应计时器任务要执行的操作。

因此每一个具体的任务类都必须继承TimerTask，然后重写run()方法。

另外它还有两个非抽象的方法：

boolean cancel()：取消此计时器任务。

long scheduledExecutionTime()：返回此任务最近实际执行的安排执行时间。

## 二、Timer实例

**1、指定延迟时间执行定时任务**

public class TimerTest01

{

Timer timer;

public TimerTest01(int time)

{

timer = new Timer();

timer.schedule(new TimerTaskTest01(), time \* 1000);

}

public static void main(String[] args)

{

System.out.println("timer begin....");

new TimerTest01(3);

}

}

public class TimerTaskTest01 extends TimerTask

{

public void run()

{

System.out.println("Time's up!!!!");

}

}

运行结果：

首先打印：timer begin....3秒后打印：Time's up!!!!

**2、在指定时间执行定时任务**

import java.util.Calendar;

import java.util.Date;

import java.util.Timer;

import java.util.TimerTask;

public class TimerTest02

{

Timer timer;

public TimerTest02()

{

Date time = getTime();

System.out.println("指定时间time=" + time);

timer = new Timer();

timer.schedule(new TimerTaskTest02(), time);

}

public Date getTime()

{

Calendar calendar = Calendar.getInstance();

calendar.set(Calendar.HOUR\_OF\_DAY, 11);

calendar.set(Calendar.MINUTE, 39);

calendar.set(Calendar.SECOND, 00);

Date time = calendar.getTime();

return time;

}

public static void main(String[] args)

{

new TimerTest02();

}

}

public class TimerTaskTest02 extends TimerTask

{

@Override

public void run()

{

System.out.println("指定时间执行线程任务...");

}

}

当时间到达11:39:00时就会执行该线程任务，当然大于该时间也会执行！！执行结果为：

指定时间time=Tue Jun 10 11:39:00 CST 2014指定时间执行线程任务...

**3、在延迟指定时间后以指定的间隔时间循环执行定时任务**

import java.util.Date;

import java.util.Timer;

import java.util.TimerTask;

public class TimerTest03

{

Timer timer;

public TimerTest03()

{

timer = new Timer();

timer.schedule(new TimerTaskTest03(), 1000, 2000);

}

public static void main(String[] args)

{

new TimerTest03();

}

}

public class TimerTaskTest03 extends TimerTask

{

@Override

public void run()

{

Date date = new Date(this.scheduledExecutionTime());

System.out.println("本次执行该线程的时间为：" + date);

}

}

运行结果:

本次执行该线程的时间为：Tue Jun 10 21:19:47 CST 2014本次执行该线程的时间为：Tue Jun 10 21:19:49 CST 2014本次执行该线程的时间为：Tue Jun 10 21:19:51 CST 2014本次执行该线程的时间为：Tue Jun 10 21:19:53 CST 2014本次执行该线程的时间为：Tue Jun 10 21:19:55 CST 2014本次执行该线程的时间为：Tue Jun 10 21:19:57 CST 2014.................

对于这个线程任务,如果我们不将该任务停止,他会一直运行下去。

**4、分析schedule和scheduleAtFixedRate**

**（1）schedule(TimerTask task, Date time)、schedule(TimerTask task, long delay)**

对于这两个方法而言，如果指定的计划执行时间scheduledExecutionTime<= systemCurrentTime，则task会被立即执行。scheduledExecutionTime不会因为某一个task的过度执行而改变。

1. **schedule(TimerTask task, Date firstTime, long period)、schedule(TimerTask task, long delay, long period)**

这两个方法与上面两个就有点儿不同的，前面提过Timer的计时器任务会因为前一个任务执行时间较长而延时。在这两个方法中，每一次执行的task的计划时间会随着前一个task的实际时间而发生改变，也就是scheduledExecutionTime(n+1)=realExecutionTime(n)+periodTime。也就是说如果第n个task由于某种情况导致这次的执行时间过程，最后导致systemCurrentTime>= scheduledExecutionTime(n+1)，这是第n+1个task并不会因为到时了而执行，他会等待第n个task执行完之后再执行，那么这样势必会导致n+2个的执行实现scheduledExecutionTime放生改变即scheduledExecutionTime(n+2) = realExecutionTime(n+1)+periodTime。所以这两个方法更加注重保存间隔时间的稳定。

**（3）scheduleAtFixedRate(TimerTask task, Date firstTime, long period)、scheduleAtFixedRate(TimerTask task, long delay, long period)**

在前面也提过scheduleAtFixedRate与schedule方法的侧重点不同，schedule方法侧重保存间隔时间的稳定，而scheduleAtFixedRate方法更加侧重于保持执行频率的稳定。为什么这么说，原因如下。在schedule方法中会因为前一个任务的延迟而导致其后面的定时任务延时，而scheduleAtFixedRate方法则不会，如果第n个task执行时间过长导致systemCurrentTime>= scheduledExecutionTime(n+1)，则不会做任何等待他会立即执行第n+1个task，所以scheduleAtFixedRate方法执行时间的计算方法不同于schedule，而是scheduledExecutionTime(n)=firstExecuteTime +n\*periodTime，该计算方法永远保持不变。所以scheduleAtFixedRate更加侧重于保持执行频率的稳定。

## 三、Timer的缺陷

**1、Timer的缺陷**

Timer计时器可以定时（指定时间执行任务）、延迟（延迟5秒执行任务）、周期性地执行任务（每隔个1秒执行任务），但是，Timer存在一些缺陷。首先Timer对调度的支持是基于绝对时间的，而不是相对时间，所以它对系统时间的改变非常敏感。其次Timer线程是不会捕获异常的，如果TimerTask抛出的了未检查异常则会导致Timer线程终止，同时Timer也不会重新恢复线程的执行，他会错误的认为整个Timer线程都会取消。同时，已经被安排单尚未执行的TimerTask也不会再执行了，新的任务也不能被调度。故如果TimerTask抛出未检查的异常，Timer将会产生无法预料的行为。

**2、Timer管理时间延迟缺陷**

前面Timer在执行定时任务时只会创建一个线程任务，如果存在多个线程，若其中某个线程因为某种原因而导致线程任务执行时间过长，超过了两个任务的间隔时间，会发生一些缺陷：

import java.util.Timer;

import java.util.TimerTask;

public class TimerTest04

{

private Timer timer;

public long start;

public TimerTest04()

{

this.timer = new Timer();

start = System.currentTimeMillis();

}

public void timerOne()

{

timer.schedule(new TimerTask()

{

public void run()

{

System.out.println("timerOne invoked ,the time:" + (System.currentTimeMillis() - start));

try

{

Thread.sleep(4000);

} catch (InterruptedException e)

{

e.printStackTrace();

}

}

}, 1000);

}

public void timerTwo()

{

timer.schedule(new TimerTask()

{

public void run()

{

System.out.println("timerOne invoked ,the time:" + (System.currentTimeMillis() - start));

}

}, 3000);

}

public static void main(String[] args) throws Exception

{

TimerTest04 test = new TimerTest04();

test.timerOne();

test.timerTwo();

}

}

按照我们正常思路，timerTwo应该是在3s后执行，其结果应该是：

timerOne invoked ,the time:1001timerOne invoked ,the time:3001

但是事与愿违，timerOne由于sleep(4000)，休眠了4S，同时Timer内部是一个线程，导致timeOne所需的时间超过了间隔时间，结果：

timerOne invoked ,the time:1000timerOne invoked ,the time:5000

**3、Timer抛出异常缺陷**

如果TimerTask抛出RuntimeException，Timer会终止所有任务的运行。如下：

import java.util.Timer;

import java.util.TimerTask;

public class TimerTest04

{

private Timer timer;

public TimerTest04()

{

this.timer = new Timer();

}

public void timerOne()

{

timer.schedule(new TimerTask()

{

public void run()

{

throw new RuntimeException();

}

}, 1000);

}

public void timerTwo()

{

timer.schedule(new TimerTask()

{

public void run()

{

System.out.println("我会不会执行呢？？");

}

}, 1000);

}

public static void main(String[] args)

{

TimerTest04 test = new TimerTest04();

test.timerOne();

test.timerTwo();

}

}

运行结果：timerOne抛出异常，导致timerTwo任务终止。

Exception in thread "Timer-0" java.lang.RuntimeException at com.chenssy.timer.TimerTest04$1.run(TimerTest04.java:25) at java.util.TimerThread.mainLoop(Timer.java:555) at java.util.TimerThread.run(Timer.java:505)

对于Timer的缺陷，我们可以考虑 ScheduledThreadPoolExecutor 来替代。Timer是基于绝对时间的，对系统时间比较敏感，而ScheduledThreadPoolExecutor 则是基于相对时间；Timer是内部是单一线程，而ScheduledThreadPoolExecutor内部是个线程池，所以可以支持多个任务并发执行。

**4、用ScheduledExecutorService替代Timer**

1、解决问题一：

import java.util.concurrent.Executors;

import java.util.concurrent.ScheduledExecutorService;

import java.util.concurrent.TimeUnit;

public class ScheduledExecutorTest

{

private ScheduledExecutorService scheduExec;

public long start;

ScheduledExecutorTest()

{

this.scheduExec = Executors.newScheduledThreadPool(2);

this.start = System.currentTimeMillis();

}

public void timerOne()

{

scheduExec.schedule(new Runnable()

{

public void run()

{

System.out.println("timerOne,the time:" + (System.currentTimeMillis() - start));

try

{

Thread.sleep(4000);

} catch (InterruptedException e)

{

e.printStackTrace();

}

}

}, 1000, TimeUnit.MILLISECONDS);

}

public void timerTwo()

{

scheduExec.schedule(new Runnable()

{

public void run()

{

System.out.println("timerTwo,the time:" + (System.currentTimeMillis() - start));

}

}, 2000, TimeUnit.MILLISECONDS);

}

public static void main(String[] args)

{

ScheduledExecutorTest test = new ScheduledExecutorTest();

test.timerOne();

test.timerTwo();

}

}

运行结果：

timerOne,the time:1003timerTwo,the time:2005

2、解决问题二

import java.util.concurrent.Executors;

import java.util.concurrent.ScheduledExecutorService;

import java.util.concurrent.TimeUnit;

public class ScheduledExecutorTest

{

private ScheduledExecutorService scheduExec;

public long start;

ScheduledExecutorTest()

{

this.scheduExec = Executors.newScheduledThreadPool(2);

this.start = System.currentTimeMillis();

}

public void timerOne()

{

scheduExec.schedule(new Runnable()

{

public void run()

{

throw new RuntimeException();

}

}, 1000, TimeUnit.MILLISECONDS);

}

public void timerTwo()

{

scheduExec.scheduleAtFixedRate(new Runnable()

{

public void run()

{

System.out.println("timerTwo invoked .....");

}

}, 2000, 500, TimeUnit.MILLISECONDS);

}

public static void main(String[] args)

{

ScheduledExecutorTest test = new ScheduledExecutorTest();

test.timerOne();

test.timerTwo();

}

}

运行结果：

timerTwo invoked .....timerTwo invoked .....timerTwo invoked .....timerTwo invoked .....timerTwo invoked .....timerTwo invoked .....timerTwo invoked .....timerTwo invoked .....timerTwo invoked .............................

**ScheduledExecutor**

## 一：简单说明

**1 ScheduleExecutorService接口中有四个重要的方法**，其中scheduleAtFixedRate和scheduleWithFixedDelay在实现定时程序时比较方便。下面是该接口的原型定义:

java.util.concurrent.ScheduleExecutorService extends ExecutorService extends Executor

**2 接口scheduleAtFixedRate原型定义及参数说明**

public ScheduledFuture<?> scheduleAtFixedRate(Runnable command,

long initialDelay,

long period,

TimeUnit unit);

command：执行线程

initialDelay：初始化延时

period：两次开始执行最小间隔时间

unit：计时单位

**3 接口scheduleWithFixedDelay原型定义及参数说明**

public ScheduledFuture<?> scheduleWithFixedDelay(Runnable command,

long initialDelay,

long delay,

TimeUnit unit);

command：执行线程

initialDelay：初始化延时

period：前一次执行结束到下一次执行开始的间隔时间（间隔执行延迟时间）

unit：计时单位

## 二：功能示例

**1.按指定频率周期执行某个任务。**

初始化延迟0ms开始执行，每隔100ms重新执行一次任务。

/\*\*

\* 以固定周期频率执行任务

\*/

public static void executeFixedRate() {

ScheduledExecutorService executor = Executors.newScheduledThreadPool(1);

executor.scheduleAtFixedRate(

new EchoServer(),

0,

100,

TimeUnit.MILLISECONDS);

}

间隔指的是连续两次任务开始执行的间隔。

**2.按指定频率间隔执行某个任务。**

初始化时延时0ms开始执行，本次执行结束后延迟100ms开始下次执行。

/\*\*

\* 以固定延迟时间进行执行

\* 本次任务执行完成后，需要延迟设定的延迟时间，才会执行新的任务

\*/

public static void executeFixedDelay() {

ScheduledExecutorService executor = Executors.newScheduledThreadPool(1);

executor.scheduleWithFixedDelay(

new EchoServer(),

0,

100,

TimeUnit.MILLISECONDS);

}

**3.周期定时执行某个任务。**

有时候我们希望一个任务被安排在凌晨3点（访问较少时）周期性的执行一个比较耗费资源的任务，可以使用下面方法设定每天在固定时间执行一次任务。

/\*\*

\* 每天晚上8点执行一次

\* 每天定时安排任务进行执行

\*/

public static void executeEightAtNightPerDay() {

ScheduledExecutorService executor = Executors.newScheduledThreadPool(1);

long oneDay = 24 \* 60 \* 60 \* 1000;

long initDelay = getTimeMillis("20:00:00") - System.currentTimeMillis();

initDelay = initDelay > 0 ? initDelay : oneDay + initDelay;

executor.scheduleAtFixedRate(

new EchoServer(),

initDelay,

oneDay,

TimeUnit.MILLISECONDS);

}

/\*\*

\* 获取指定时间对应的毫秒数

\* @param time "HH:mm:ss"

\* @return

\*/

private static long getTimeMillis(String time) {

try {

DateFormat dateFormat = new SimpleDateFormat("yy-MM-dd HH:mm:ss");

DateFormat dayFormat = new SimpleDateFormat("yy-MM-dd");

Date curDate = dateFormat.parse(dayFormat.format(new Date()) + " " + time);

return curDate.getTime();

} catch (ParseException e) {

e.printStackTrace();

}

return 0;

}

**4.辅助代码**

class EchoServer implements Runnable {

@Override

public void run() {

try {

Thread.sleep(50);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("This is a echo server. The current time is " +

System.currentTimeMillis() + ".");

}

}

## 三、一些问题

上面写的内容有不严谨的地方，比如对于scheduleAtFixedRate方法，当我们要执行的任务大于我们指定的执行间隔时会怎么样呢？对于中文API中的注释，我们可能会被忽悠，认为无论怎么样，它都会按照我们指定的间隔进行执行，其实当执行任务的时间大于我们指定的间隔时间时，它并不会在指定间隔时开辟一个新的线程并发执行这个任务。而是等待该线程执行完毕。

源码注释如下：

\* Creates and executes a periodic action that becomes enabled first

\* after the given initial delay, and subsequently with the given

\* period; that is executions will commence after

\* <tt>initialDelay</tt> then <tt>initialDelay+period</tt>, then

\* <tt>initialDelay + 2 \* period</tt>, and so on.

\* If any execution of the task

\* encounters an exception, subsequent executions are suppressed.

\* Otherwise, the task will only terminate via cancellation or

\* termination of the executor. If any execution of this task

\* takes longer than its period, then subsequent executions

\* may start late, but will not concurrently execute.

根据注释中的内容，我们需要注意的时，我们需要捕获最上层的异常，防止出现异常中止执行，导致周期性的任务不再执行。

## 四：Spring管理ScheduleExecutorService

Spring自动定时任务配置方法（我们要执行任务的类名为com.study.MyTimedTask）

<bean id="myTimedTask" class="com.study.MyTimedTask"/>

<bean id="doMyTimedTask" class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">

<property name="targetObject" ref="myTimedTask"/>

<property name="targetMethod" value="execute"/>

<property name="concurrent" value="false"/>

</bean>

<bean id="myTimedTaskTrigger" class="org.springframework.scheduling.quartz.CronTriggerBean">

<property name="jobDetail" ref="doMyTimedTask"/>

<property name="cronExpression" value="0 0 2 \* ?"/>

</bean>

<bean id="doScheduler" class="org.springframework.scheduling.quartz.SchedulerFactoryBean">

<property name="triggers">

<list>

<ref local="myTimedTaskTrigger"/>

</list>

</property>

</bean>

<bean id="doScheduler" class="org.springframework.scheduling.quartz.SchedulerFactoryBean">

<property name="triggers">

<list>

<bean class="org.springframework.scheduling.quartz.CronTriggerBean">

<property name="jobDetail"/>

<bean id="doMyTimedTask" class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">

<property name="targetObject">

<bean class="com.study.MyTimedTask"/>

</property>

<property name="targetMethod" value="execute"/>

<property name="concurrent" value="false"/>

</bean>

</property>

<property name="cronExpression" value="0 0 2 \* ?"/>

</bean>

</list>

</property>

</bean>

**开源工具包 Quartz**

## 一、Quartz简介

1 Quartz是一个任务调度框架。比如你遇到这样的问题

@想每月25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，备份一下自己的爱情动作片 学习笔记到云盘

2 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。

## 二、简单代码开发

**1 一个简单的示例**

这里面的所有例子都是基于Quartz 2.2.1

package com.test.quartz;

import static org.quartz.DateBuilder.newDate;

import static org.quartz.JobBuilder.newJob;

import static org.quartz.SimpleScheduleBuilder.simpleSchedule;

import static org.quartz.TriggerBuilder.newTrigger;

import java.util.GregorianCalendar;

import org.quartz.JobDetail;

import org.quartz.Scheduler;

import org.quartz.Trigger;

import org.quartz.impl.StdSchedulerFactory;

import org.quartz.impl.calendar.AnnualCalendar;

public class QuartzTest {

public static void main(String[] args) {

try {

//创建scheduler

Scheduler scheduler = StdSchedulerFactory.getDefaultScheduler();

//定义一个Trigger

Trigger trigger = newTrigger().withIdentity("trigger1", "group1") //定义name/group

.startNow()//一旦加入scheduler，立即生效

.withSchedule(simpleSchedule() //使用SimpleTrigger

.withIntervalInSeconds(1) //每隔一秒执行一次

.repeatForever()) //一直执行，奔腾到老不停歇

.build();

//定义一个JobDetail

JobDetail job = newJob(HelloQuartz.class) //定义Job类为HelloQuartz类，这是真正的执行逻辑所在

.withIdentity("job1", "group1") //定义name/group

.usingJobData("name", "quartz") //定义属性

.build();

//加入这个调度

scheduler.scheduleJob(job, trigger);

//启动之

scheduler.start();

//运行一段时间后关闭

Thread.sleep(10000);

scheduler.shutdown(true);

} catch (Exception e) {

e.printStackTrace();

}

}

}

package com.test.quartz;

import java.util.Date;

import org.quartz.DisallowConcurrentExecution;

import org.quartz.Job;

import org.quartz.JobDetail;

import org.quartz.JobExecutionContext;

import org.quartz.JobExecutionException;

public class HelloQuartz implements Job {

public void execute(JobExecutionContext context) throws JobExecutionException {

JobDetail detail = context.getJobDetail();

String name = detail.getJobDataMap().getString("name");

System.out.println("say hello to " + name + " at " + new Date());

}

}

**2 这个例子很好的覆盖了Quartz最重要的3个基本要素**

**（1）Scheduler：**调度器。所有的调度都是由它控制。

**（2）Trigger：** 定义触发的条件。例子中，它的类型是SimpleTrigger，每隔1秒中执行一次（什么是SimpleTrigger下面会有详述）。

**（3）JobDetail & Job：** JobDetail 定义的是任务数据，而真正的执行逻辑是在Job中，例子中是HelloQuartz。 为什么设计成JobDetail + Job，不直接使用Job？这是因为任务是有可能并发执行，如果Scheduler直接使用Job，就会存在对同一个Job实例并发访问的问题。而JobDetail & Job 方式，sheduler每次执行，都会根据JobDetail创建一个新的Job实例，这样就可以规避并发访问的问题。

**3 Quartz API**

（1）Quartz的API的风格在2.x以后，采用的是DSL风格（通常意味着fluent interface风格），就是示例中newTrigger()那一段东西。它是通过Builder实现的，就是以下几个。（\*\* 下面大部分代码都要引用这些Builder \*\* )

import static org.quartz.JobBuilder.\*;

import static org.quartz.TriggerBuilder.\*;

import static org.quartz.SimpleScheduleBuilder.\*;

import static org.quartz.CronScheduleBuilder.\*;

import static org.quartz.DailyTimeIntervalScheduleBuilder.\*;

import static org.quartz.CalendarIntervalScheduleBuilder.\*;

import static org.quartz.DateBuilder.\*;

（2）DSL风格写起来会更加连贯，畅快，而且由于不是使用setter的风格，语义上会更容易理解一些。对比一下：

JobDetail jobDetail=new JobDetailImpl("jobDetail1","group1",HelloQuartz.class);

jobDetail.getJobDataMap().put("name", "quartz");

SimpleTriggerImpl trigger=new SimpleTriggerImpl("trigger1","group1");

trigger.setStartTime(new Date());

trigger.setRepeatInterval(1);

trigger.setRepeatCount(-1);

**4 关于name和group**

JobDetail和Trigger都有name和group。

name是它们在这个sheduler里面的唯一标识。如果我们要更新一个JobDetail定义，只需要设置一个name相同的JobDetail实例即可。

group是一个组织单元，sheduler会提供一些对整组操作的API，比如 scheduler.resumeJobs()。

## 三、Trigger

在开始详解每一种Trigger之前，需要先了解一下Trigger的一些共性。

**1 StartTime & EndTime**

startTime和endTime指定的Trigger会被触发的时间区间。在这个区间之外，Trigger是不会被触发的。

\*\* 所有Trigger都会包含这两个属性 \*\*

**2 优先级（Priority）**

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。

**3 Misfire(错失触发）策略**

类似的Scheduler资源不足的时候，或者机器崩溃重启等，有可能某一些Trigger在应该触发的时间点没有被触发，也就是Miss Fire了。这个时候Trigger需要一个策略来处理这种情况。每种Trigger可选的策略各不相同。

这里有两个点需要重点注意：

MisFire的触发是有一个阀值，这个阀值是配置在JobStore的。比RAMJobStore是org.quartz.jobStore.misfireThreshold。只有超过这个阀值，才会算MisFire。小于这个阀值，Quartz是会全部重新触发。

所有MisFire的策略实际上都是解答两个问题：

已经MisFire的任务还要重新触发吗？

如果发生MisFire，要调整现有的调度时间吗？

SimpleTrigger的MisFire策略有：

**MISFIRE\_INSTRUCTION\_IGNORE\_MISFIRE\_POLICY**

这个不是忽略已经错失的触发的意思，而是说忽略MisFire策略。它会在资源合适的时候，重新触发所有的MisFire任务，并且不会影响现有的调度时间。比如，SimpleTrigger每15秒执行一次，而中间有5分钟时间它都MisFire了，一共错失了20个，5分钟后，假设资源充足了，并且任务允许并发，它会被一次性触发。这个属性是所有Trigger都适用。

**MISFIRE\_INSTRUCTION\_FIRE\_NOW**

忽略已经MisFire的任务，并且立即执行调度。这通常只适用于只执行一次的任务。

**MISFIRE\_INSTRUCTION\_RESCHEDULE\_NOW\_WITH\_EXISTING\_REPEAT\_COUNT**

将startTime设置当前时间，立即重新调度任务，包括的MisFire的

**MISFIRE\_INSTRUCTION\_RESCHEDULE\_NOW\_WITH\_REMAINING\_REPEAT\_COUNT**

类似WITH\_EXISTING\_REPEAT\_COUNT，区别在于会忽略已经MisFire的任务

**MISFIRE\_INSTRUCTION\_RESCHEDULE\_NEXT\_WITH\_EXISTING\_COUNT**

在下一次调度时间点，重新开始调度任务，包括的MisFire的

**MISFIRE\_INSTRUCTION\_RESCHEDULE\_NEXT\_WITH\_REMAINING\_COUNT**

类似于\_NEXT\_WITH\_EXISTING\_COUNT，区别在于会忽略已经MisFire的任务。

**MISFIRE\_INSTRUCTION\_SMART\_POLICY**

所有的Trigger的MisFire默认值都是这个，大致意思是“把处理逻辑交给聪明的Quartz去决定”。基本策略是，如果是只执行一次的调度，使用MISFIRE\_INSTRUCTION\_FIRE\_NOW如果是无限次的调度(repeatCount是无限的)，使用MISFIRE\_INSTRUCTION\_RESCHEDULE\_NEXT\_WITH\_REMAINING\_COUNT

否则，使用MISFIRE\_INSTRUCTION\_RESCHEDULE\_NOW\_WITH\_EXISTING\_REPEAT\_COUNT

**4 Calendar**

这里的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25日零点自动还卡债“为例，我们想排除掉每年的2月25号零点这个时间点（因为有2.14，所以2月一定会破产）。这个时间，就可以用Calendar来实现。

AnnualCalendar cal = new AnnualCalendar(); //定义一个每年执行Calendar，精度为天，即不能定义到2.25号下午2:00

java.util.Calendar excludeDay = new GregorianCalendar();

excludeDay.setTime(newDate().inMonthOnDay(2, 25).build());

cal.setDayExcluded(excludeDay, true); //设置排除2.25这个日期

scheduler.addCalendar("FebCal", cal, false, false); //scheduler加入这个Calendar

Trigger trigger = newTrigger().withIdentity("trigger1", "group1")

.startNow()//一旦加入scheduler，立即生效

.modifiedByCalendar("FebCal") //使用Calendar !!

.withSchedule(simpleSchedule()

.withIntervalInSeconds(1)

.repeatForever())

.build();

Quartz体贴地为我们提供以下几种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。

**5 Trigger实现类**

**（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。\*\* 下面有关repeatCount 属性的都是同理。　\*\*

simpleSchedule()

.withIntervalInHours(1) //每小时执行一次

.repeatForever() //次数不限

.build();

simpleSchedule()

.withIntervalInMinutes(1) //每分钟执行一次

.withRepeatCount(10) //次数为10次

.build();

**（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。相较于SimpleTrigger有两个优势：1、更方便，比如每隔1小时执行，你不用自己去计算1小时等于多少毫秒。 2、支持不是固定长度的间隔，比如间隔为月和年。但劣势是精度只能到秒。它适合的任务类似于：9:00 开始执行，并且以后每周 9:00 执行一次它的属性有:

interval 执行间隔

intervalUnit 执行间隔的单位（秒，分钟，小时，天，月，年，星期）

calendarIntervalSchedule()

.withIntervalInDays(1) //每天执行一次

.build();

calendarIntervalSchedule()

.withIntervalInWeeks(1) //每周执行一次

.build();

**（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

dailyTimeIntervalSchedule()

.startingDailyAt(TimeOfDay.hourAndMinuteOfDay(9, 0)) //第天9：00开始

.endingDailyAt(TimeOfDay.hourAndMinuteOfDay(16, 0)) //16：00 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();

dailyTimeIntervalSchedule()

.startingDailyAt(TimeOfDay.hourAndMinuteOfDay(9, 0)) //第天9：00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();

**（4）CronTrigger**

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。下面会有说明。

cronSchedule("0 0/2 8-17 \* \* ?") // 每天8:00-17:00，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();

weeklyOnDayAndHourAndMinute(MONDAY,9, 30) //等同于 0 30 9 ? \* MON

.build();

**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 C**

**5 月份 1-12 , - \* /**

**6 星期 1-7 , - \* ? / L 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。

一些例子：

表示式 说明

0 0 12 \* \* ? 每天12点运行

0 15 10 ? \* \* 每天10:15运行

0 15 10 \* \* ? 每天10:15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 10,44 14 ? 3 WED 3月每周三的14:10分到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。

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。

0 15 10 ? \* 6#3 每月第三个星期五的10:15分运行。

**6 JobDetail & Job**

JobDetail是任务的定义，而Job是任务的执行逻辑。在JobDetail里会引用一个Job Class定义。一个最简单的例子

public class JobTest {

public static void main(String[] args) throws SchedulerException, IOException {

JobDetail job=newJob()

.ofType(DoNothingJob.class) //引用Job Class

.withIdentity("job1", "group1") //设置name/group

.withDescription("this is a test job") //设置描述

.usingJobData("age", 18) //加入属性到ageJobDataMap

.build();

job.getJobDataMap().put("name", "quertz"); //加入属性name到JobDataMap

Trigger trigger=newTrigger()

.startNow()

.withIdentity("trigger1")

.withSchedule(simpleSchedule()

.withIntervalInSeconds(1)

.repeatForever())

.build();

Scheduler sche=StdSchedulerFactory.getDefaultScheduler();

sche.scheduleJob(job, trigger);

sche.start();

System.in.read();

sche.shutdown();

}

}

public class DoNothingJob implements Job {

public void execute(JobExecutionContext context) throws JobExecutionException {

System.out.println("do nothing");

}

}

从上例我们可以看出，要定义一个任务，需要干几件事：

创建一个org.quartz.Job的实现类，并实现实现自己的业务逻辑。比如上面的DoNothingJob。

定义一个JobDetail，引用这个实现类

加入scheduleJob

Quartz调度一次任务，会干如下的事：

JobClass jobClass=JobDetail.getJobClass()

Job jobInstance=jobClass.newInstance()。所以Job实现类，必须有一个public的无参构建方法。

jobInstance.execute(JobExecutionContext context)。JobExecutionContext是Job运行的上下文，可以获得Trigger、Scheduler、JobDetail的信息。

也就是说，每次调度都会创建一个新的Job实例，这样的好处是有些任务并发执行的时候，不存在对临界资源的访问问题——当然，如果需要共享JobDataMap的时候，还是存在临界资源的并发访问的问题。

**7 JobDataMap**

Job都次都是newInstance的实例，那我怎么传值给它？ 比如我现在有两个发送邮件的任务，一个是发给"liLei",一个发给"hanmeimei",不能说我要写两个Job实现类LiLeiSendEmailJob和HanMeiMeiSendEmailJob。实现的办法是通过JobDataMap。每一个JobDetail都会有一个JobDataMap。JobDataMap本质就是一个Map的扩展类，只是提供了一些更便捷的方法，比如getString()之类的。我们可以在定义JobDetail，加入属性值，方式有二：

newJob().usingJobData("age", 18) //加入属性到ageJobDataMap

or

job.getJobDataMap().put("name", "quertz"); //加入属性name到JobDataMap

然后在Job中可以获取这个JobDataMap的值，方式同样有二：

public class HelloQuartz implements Job {

private String name;

public void execute(JobExecutionContext context) throws JobExecutionException {

JobDetail detail = context.getJobDetail();

JobDataMap map = detail.getJobDataMap(); //方法一：获得JobDataMap

System.out.println("say hello to " + name + "[" + map.getInt("age") + "]" + " at "

+ new Date());

}

public void setName(String name) {

this.name = name;

}

}

对于同一个JobDetail实例，执行的多个Job实例，是共享同样的JobDataMap，也就是说，如果你在任务里修改了里面的值，会对其他Job实例（并发的或者后续的）造成影响。

除了JobDetail，Trigger同样有一个JobDataMap，共享范围是所有使用这个Trigger的Job实例。

**8 Job并发**

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。

public class DoNothingJob implements Job {

@DisallowConcurrentExecution

public void execute(JobExecutionContext context) throws JobExecutionException {

System.out.println("do nothing");

}

}

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。

**9 JobExecutionException**

Job.execute()方法是不允许抛出除JobExecutionException之外的所有异常的（包括RuntimeException)，所以编码的时候，最好是try-catch住所有的Throwable，小心处理。

**（1）Durability(耐久性？)**

如果一个任务不是durable，那么当没有Trigger关联它的时候，它就会被自动删除。

**（2）RequestsRecovery**

如果一个任务是"requests recovery"，那么当任务运行过程非正常退出时（比如进程崩溃，机器断电，但不包括抛出异常这种情况），Quartz再次启动时，会重新运行一次这个任务实例。可以通过JobExecutionContext.isRecovering()查询任务是否是被恢复的。

## 四、Scheduler

**1 Scheduler就是Quartz的大脑，所有任务都是由它来安排。**

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。它有多种实现RAMJob(内存实现)，JobStoreTX(JDBC，事务由Quartz管理），JobStoreCMT(JDBC，使用容器事务)，ClusteredJobStore(集群实现)、TerracottaJobStore(什么是Terractta)。**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。**

**2 SchedulerFactory**

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，可以远程提交个Job什么的。

**3 DirectSchedulerFactory的创建接口**

/\*\*

\* Same as

\* {@link DirectSchedulerFactory#createScheduler(ThreadPool threadPool, JobStore jobStore)},

\* with the addition of specifying the scheduler name and instance ID. This

\* scheduler can only be retrieved via

\* {@link DirectSchedulerFactory#getScheduler(String)}

\*

\* @param schedulerName

\* The name for the scheduler.

\* @param schedulerInstanceId

\* The instance ID for the scheduler.

\* @param threadPool

\* The thread pool for executing jobs

\* @param jobStore

\* The type of job store

\* @throws SchedulerException

\* if initialization failed

\*/

public void createScheduler(String schedulerName,

String schedulerInstanceId, ThreadPool threadPool, JobStore jobStore)

throws SchedulerException;

**4 StdSchdulerFactory的配置例子， 更多配置，参考Quartz配置指南：**

org.quartz.scheduler.instanceName = DefaultQuartzScheduler

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.class = org.quartz.simpl.RAMJobStore

## 五、spring+quartz配置

Spring被用在了越来越多的项目中, quartz也被公认为是比较好用的定时器设置工具, 在这里通过一个demo说明如何使用spring和quartz配置多个定时任务.

环境: eclipse + maven + spring 3.0.6.RELEASE版本 + quartz 1.8.6版本

**1.准备工作**

需要引入的jar包, 这里是采用maven的dependency依赖; 如果没有使用maven, 可以对照着上面的jar包截图一一进行下载. 这里唯一需要注意的就是, 如果使用spring 3.1以下的版本, quartz就需要相应的2.0以下版本, 否则会导致 org.springframework.beans.factory.CannotLoadBeanClassException: Error loading class [org.springframework.scheduling.quartz.CronTriggerBean] for bean with name 'cronTrigger\_1' defined in file [E:\workspace\.metadata\.plugins\org.eclipse.wst.server.core\tmp0\wtpwebapps\git-common\WEB-INF\classes\applicationContext.xml]: problem with class file or dependent class; nested exception isjava.lang.IncompatibleClassChangeError: class org.springframework.scheduling.quartz.CronTriggerBean has interface org.quartz.CronTrigger as super class 错误.

后续: 后来使用了spring 4.0.5.RELEASE版本+quartz 2.2.1版本, 还是报出了以上错误, 将quartz降低为1.8.6版本后, 问题消失...

<properties>

<springframework.version>3.0.6.RELEASE</springframework.version>

</properties>

<dependencies>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>servlet-api</artifactId>

<version>2.5</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>${springframework.version}</version>

</dependency>

<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>1.8.6</version>

</dependency>

</dependencies>

**2 程序代码及配置代码**

测试代码就简单的打印出一句话

package com.quartz.demo;

import java.util.Date;

public class Job1Demo {

public void sayHello() {

System.out.println(new Date() + " -> Hello, 我是任务 1");

}

}

package com.quartz.demo;

import java.util.Date;

public class Job2Demo {

public void sayHello() {

System.out.println(new Date() + " -> Hello, 我是任务 2");

}

}

**3 Spring配置文件applicationContext.xml中配置了要注入的bean, 定时任务, 任务执行周期时间等**

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

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:p="http://www.springframework.org/schema/p"

xmlns:context="http://www.springframework.org/schema/context"

xmlns:mvc="http://www.springframework.org/schema/mvc"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context.xsd

http://www.springframework.org/schema/mvc

http://www.springframework.org/schema/mvc/spring-mvc.xsd">

<!-- 任务1配置 -->

<bean name="job1" class="com.quartz.demo.Job1Demo"/>

<bean id="jobDetail\_1" class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">

<!-- 执行的类 -->

<property name="targetObject">

<ref bean="job1" />

</property>

<!-- 类中的方法 -->

<property name="targetMethod">

<value>sayHello</value>

</property>

</bean>

<bean id="cronTrigger\_1" class="org.springframework.scheduling.quartz.CronTriggerBean">

<property name="jobDetail">

<ref bean="jobDetail\_1" />

</property>

<!-- 每一秒钟执行一次 -->

<property name="cronExpression">

<value>0/1 \* \* \* \* ?</value>

</property>

</bean>

<!-- 任务2配置 -->

<bean name="job2" class="com.quartz.demo.Job2Demo"/>

<bean id="jobDetail\_2" class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">

<property name="targetObject">

<ref bean="job2" />

</property>

<property name="targetMethod">

<value>sayHello</value>

</property>

</bean>

<bean id="cronTrigger\_2" class="org.springframework.scheduling.quartz.CronTriggerBean">

<property name="jobDetail">

<ref bean="jobDetail\_2" />

</property>

<property name="cronExpression">

<value>0/1 \* \* \* \* ?</value>

</property>

</bean>

<!-- 总配置 -->

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

<!-- 添加触发器 -->

<property name="triggers">

<list>

<ref bean="cronTrigger\_1" />

<ref bean="cronTrigger\_2" />

</list>

</property>

</bean>

</beans>

<bean id=*"workFlowTaskScheduler"* class=*"com.yunnex.ops.erp.modules.workflow.flow.task.WorkFlowTaskScheduler"* />

<bean id=*"workFlowTaskJob"* class=*"org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean"*>

<property name=*"targetObject"* ref=*"workFlowTaskScheduler"* />

<property name=*"targetMethod"* value=*"changeState"* />

</bean>

<bean id=*"workFlowTaskTrigger"* class=*"org.springframework.scheduling.quartz.CronTriggerFactoryBean"*>

<property name=*"jobDetail"* ref=*"workFlowTaskJob"* />

<!-- 30分钟执行一次 -->

<property name=*"cronExpression"* value=*"${jyk\_flow\_scheduler}"* />

</bean>

<bean id=*"schedulerFactory"* class=*"org.springframework.scheduling.quartz.SchedulerFactoryBean"*>

<property name=*"triggers"*>

<array>

<ref bean=*"workFlowTaskTrigger"* />

</array>

</property>

</bean>

**4 web.xml, 在spring容器启动时读取加载applicationContext.xml配置文件**

<!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:\*.xml

</param-value>

</context-param>

<!-- Spring 容器启动监听器 -->

<listener>

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

</listener>

</web-app>

**5 执行结果**

将web项目部署到tomcat等容器中, 启动后, 就可以看到定时器执行效果, 如下图所示

## 六、SpringBoot+Quartz

**1 POM.XML文件**

<!-- 定时器任务 quartz需要导入的坐标 -->

<dependency>

<groupId>org.quartz-scheduler</groupId>

<artifactId>quartz</artifactId>

<version>1.8.5</version>

</dependency>

**2 类似于控制器代码:**

package com.xiaowu.quartz.demo;

import java.util.Date;

import org.springframework.scheduling.annotation.Scheduled;

import org.springframework.stereotype.Component;

@Component

public class QuartzDemo {

@Scheduled(cron = "0 0/1 \* \* \* ?") // 每分钟执行一次

public void work() throws Exception {

System.out.println("执行调度任务："+new Date());

}

@Scheduled(fixedRate = 5000)//每5秒执行一次

public void play() throws Exception {

System.out.println("执行Quartz定时器任务："+new Date());

}

@Scheduled(cron = "0/2 \* \* \* \* ?") //每2秒执行一次

public void doSomething() throws Exception {

System.out.println("每2秒执行一个的定时任务："+new Date());

}

@Scheduled(cron = "0 0 0/1 \* \* ? ") // 每一小时执行一次

public void goWork() throws Exception {

System.out.println("每一小时执行一次的定时任务："+new Date());

}

}

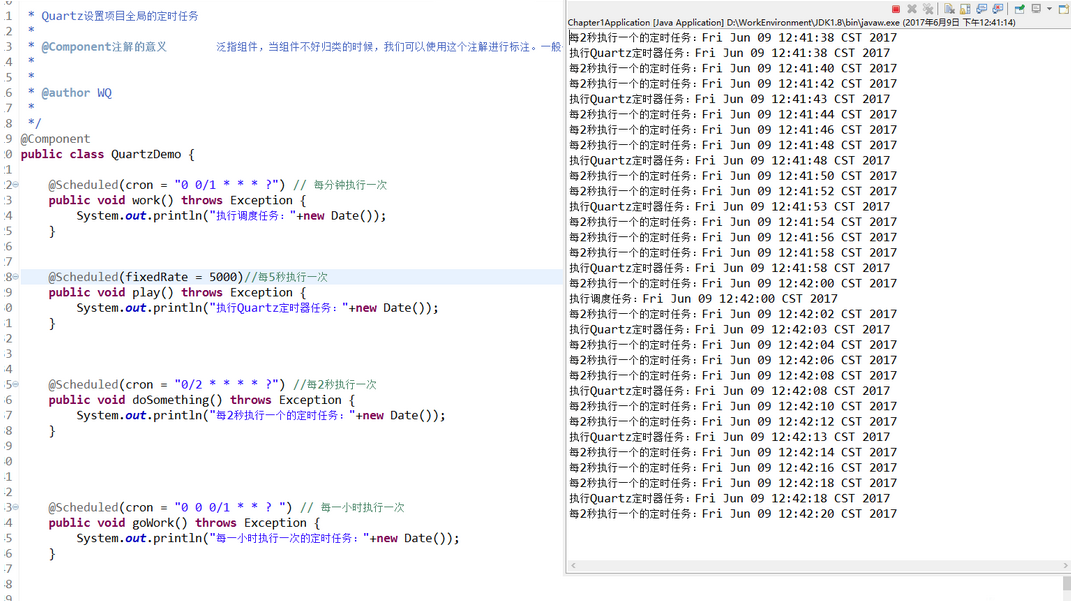
**3 启动SpringBoot项目**

public static void main(String[] args) {

SpringApplication.run(Chapter1Application.class, args);

}

**4 截图如下**



## 七、集群解决定时器重复执行

**1 问题描述**

Spring自带的Task虽然能很好使用定时任务，只需要做些简单的配置就可以了。不过如果部署在多台服务器上的时候，这样定时任务会在每台服务器都会执行，造成重复执行。

**如果宕机了怎么样，怎么高可用？**

**2全局锁机制**

对于只需要执行一次初始化的操作，在全局缓存中设置一个全局锁，拿到这个锁的节点执行相应的任务。考虑zookeeper。

**3.数据库/redis层面的状态标记**

对于有些定时任务，我们只希望执行一次，也就是说多个节点，只有一个节点在执行这个定时任务，可以在数据库对这个任务进行状态标记，根据状态来调节节点间定时任务的执，如果宕机了，可以考虑重新分配数据库锁

**4.代码分离**

如果有可能，将定时任务独立出来，成为一个单独的项目工程，部署单节点，避免集群部署