**几种任务调度的 Java 实现方法与比较**

综观目前的 Web 应用，多数应用都具备任务调度的功能。本文由浅入深介绍了几种任务调度的 Java 实现方法，包括 Timer，Scheduler, Quartz 以及 JCron Tab，并对其优缺点进行比较，目的在于给需要开发任务调度的程序员提供有价值的参考。

**前言**

任务调度是指基于给定时间点，给定时间间隔或者给定执行次数自动执行任务。本文由浅入深介绍四种任务调度的 Java 实现：

* Timer
* ScheduledExecutor
* 开源工具包 Quartz
* 开源工具包 JCronTab

此外，为结合实现复杂的任务调度，本文还将介绍 Calendar 的一些使用方法。

[**回页首**](https://www.ibm.com/developerworks/cn/java/j-lo-taskschedule/#ibm-pcon)

**Timer**

相信大家都已经非常熟悉 java.util.Timer 了，它是最简单的一种实现任务调度的方法，下面给出一个具体的例子：

**清单 1. 使用 Timer 进行任务调度**

package com.ibm.scheduler;

import java.util.Timer;

import java.util.TimerTask;

public class TimerTest extends TimerTask {

private String jobName = "";

public TimerTest(String jobName) {

super();

this.jobName = jobName;

}

@Override

public void run() {

System.out.println("execute " + jobName);

}

public static void main(String[] args) {

Timer timer = new Timer();

long delay1 = 1 \* 1000;

long period1 = 1000;

// 从现在开始 1 秒钟之后，每隔 1 秒钟执行一次 job1

timer.schedule(new TimerTest("job1"), delay1, period1);

long delay2 = 2 \* 1000;

long period2 = 2000;

// 从现在开始 2 秒钟之后，每隔 2 秒钟执行一次 job2

timer.schedule(new TimerTest("job2"), delay2, period2);

}

}

Output:

execute job1

execute job1

execute job2

execute job1

execute job1

execute job2

使用 Timer 实现任务调度的核心类是 Timer 和 TimerTask。其中 Timer 负责设定 TimerTask 的起始与间隔执行时间。使用者只需要创建一个 TimerTask 的继承类，实现自己的 run 方法，然后将其丢给 Timer 去执行即可。

Timer 的设计核心是一个 TaskList 和一个 TaskThread。Timer 将接收到的任务丢到自己的 TaskList 中，TaskList 按照 Task 的最初执行时间进行排序。TimerThread 在创建 Timer 时会启动成为一个守护线程。这个线程会轮询所有任务，找到一个最近要执行的任务，然后休眠，当到达最近要执行任务的开始时间点，TimerThread 被唤醒并执行该任务。之后 TimerThread 更新最近一个要执行的任务，继续休眠。

Timer 的优点在于简单易用，但由于所有任务都是由同一个线程来调度，因此所有任务都是串行执行的，同一时间只能有一个任务在执行，前一个任务的延迟或异常都将会影响到之后的任务。

[**回页首**](https://www.ibm.com/developerworks/cn/java/j-lo-taskschedule/#ibm-pcon)

**ScheduledExecutor**

鉴于 Timer 的上述缺陷，Java 5 推出了基于线程池设计的 ScheduledExecutor。其设计思想是，每一个被调度的任务都会由线程池中一个线程去执行，因此任务是并发执行的，相互之间不会受到干扰。需要注意的是，只有当任务的执行时间到来时，ScheduedExecutor 才会真正启动一个线程，其余时间 ScheduledExecutor 都是在轮询任务的状态。

**清单 2. 使用 ScheduledExecutor 进行任务调度**

package com.ibm.scheduler;

import java.util.concurrent.Executors;

import java.util.concurrent.ScheduledExecutorService;

import java.util.concurrent.TimeUnit;

public class ScheduledExecutorTest implements Runnable {

private String jobName = "";

public ScheduledExecutorTest(String jobName) {

super();

this.jobName = jobName;

}

@Override

public void run() {

System.out.println("execute " + jobName);

}

public static void main(String[] args) {

ScheduledExecutorService service = Executors.newScheduledThreadPool(10);

long initialDelay1 = 1;

long period1 = 1;

// 从现在开始1秒钟之后，每隔1秒钟执行一次job1

service.scheduleAtFixedRate(

new ScheduledExecutorTest("job1"), initialDelay1,

period1, TimeUnit.SECONDS);

long initialDelay2 = 1;

long delay2 = 1;

// 从现在开始2秒钟之后，每隔2秒钟执行一次job2

service.scheduleWithFixedDelay(

new ScheduledExecutorTest("job2"), initialDelay2,

delay2, TimeUnit.SECONDS);

}

}

Output:

execute job1

execute job1

execute job2

execute job1

execute job1

execute job2

清单 2 展示了 ScheduledExecutorService 中两种最常用的调度方法 ScheduleAtFixedRate 和 ScheduleWithFixedDelay。ScheduleAtFixedRate 每次执行时间为上一次任务开始起向后推一个时间间隔，即每次执行时间为 :initialDelay, initialDelay+period, initialDelay+2\*period, …；ScheduleWithFixedDelay 每次执行时间为上一次任务结束起向后推一个时间间隔，即每次执行时间为：initialDelay, initialDelay+executeTime+delay, initialDelay+2\*executeTime+2\*delay。由此可见，ScheduleAtFixedRate 是基于固定时间间隔进行任务调度，ScheduleWithFixedDelay 取决于每次任务执行的时间长短，是基于不固定时间间隔进行任务调度。

[**回页首**](https://www.ibm.com/developerworks/cn/java/j-lo-taskschedule/#ibm-pcon)

**用 ScheduledExecutor 和 Calendar 实现复杂任务调度**

Timer 和 ScheduledExecutor 都仅能提供基于开始时间与重复间隔的任务调度，不能胜任更加复杂的调度需求。比如，设置每星期二的 16:38:10 执行任务。该功能使用 Timer 和 ScheduledExecutor 都不能直接实现，但我们可以借助 Calendar 间接实现该功能。

**清单 3. 使用 ScheduledExcetuor 和 Calendar 进行任务调度**

package com.ibm.scheduler;

import java.util.Calendar;

import java.util.Date;

import java.util.TimerTask;

import java.util.concurrent.Executors;

import java.util.concurrent.ScheduledExecutorService;

import java.util.concurrent.TimeUnit;

public class ScheduledExceutorTest2 extends TimerTask {

private String jobName = "";

public ScheduledExceutorTest2(String jobName) {

super();

this.jobName = jobName;

}

@Override

public void run() {

System.out.println("Date = "+new Date()+", execute " + jobName);

}

/\*\*

\* 计算从当前时间currentDate开始，满足条件dayOfWeek, hourOfDay,

\* minuteOfHour, secondOfMinite的最近时间

\* @return

\*/

public Calendar getEarliestDate(Calendar currentDate, int dayOfWeek,

int hourOfDay, int minuteOfHour, int secondOfMinite) {

//计算当前时间的WEEK\_OF\_YEAR,DAY\_OF\_WEEK, HOUR\_OF\_DAY, MINUTE,SECOND等各个字段值

int currentWeekOfYear = currentDate.get(Calendar.WEEK\_OF\_YEAR);

int currentDayOfWeek = currentDate.get(Calendar.DAY\_OF\_WEEK);

int currentHour = currentDate.get(Calendar.HOUR\_OF\_DAY);

int currentMinute = currentDate.get(Calendar.MINUTE);

int currentSecond = currentDate.get(Calendar.SECOND);

//如果输入条件中的dayOfWeek小于当前日期的dayOfWeek,则WEEK\_OF\_YEAR需要推迟一周

boolean weekLater = false;

if (dayOfWeek < currentDayOfWeek) {

weekLater = true;

} else if (dayOfWeek == currentDayOfWeek) {

//当输入条件与当前日期的dayOfWeek相等时，如果输入条件中的

//hourOfDay小于当前日期的

//currentHour，则WEEK\_OF\_YEAR需要推迟一周

if (hourOfDay < currentHour) {

weekLater = true;

} else if (hourOfDay == currentHour) {

//当输入条件与当前日期的dayOfWeek, hourOfDay相等时，

//如果输入条件中的minuteOfHour小于当前日期的

//currentMinute，则WEEK\_OF\_YEAR需要推迟一周

if (minuteOfHour < currentMinute) {

weekLater = true;

} else if (minuteOfHour == currentSecond) {

//当输入条件与当前日期的dayOfWeek, hourOfDay，

//minuteOfHour相等时，如果输入条件中的

//secondOfMinite小于当前日期的currentSecond，

//则WEEK\_OF\_YEAR需要推迟一周

if (secondOfMinite < currentSecond) {

weekLater = true;

}

}

}

}

if (weekLater) {

//设置当前日期中的WEEK\_OF\_YEAR为当前周推迟一周

currentDate.set(Calendar.WEEK\_OF\_YEAR, currentWeekOfYear + 1);

}

// 设置当前日期中的DAY\_OF\_WEEK,HOUR\_OF\_DAY,MINUTE,SECOND为输入条件中的值。

currentDate.set(Calendar.DAY\_OF\_WEEK, dayOfWeek);

currentDate.set(Calendar.HOUR\_OF\_DAY, hourOfDay);

currentDate.set(Calendar.MINUTE, minuteOfHour);

currentDate.set(Calendar.SECOND, secondOfMinite);

return currentDate;

}

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

ScheduledExceutorTest2 test = new ScheduledExceutorTest2("job1");

//获取当前时间

Calendar currentDate = Calendar.getInstance();

long currentDateLong = currentDate.getTime().getTime();

System.out.println("Current Date = " + currentDate.getTime().toString());

//计算满足条件的最近一次执行时间

Calendar earliestDate = test

.getEarliestDate(currentDate, 3, 16, 38, 10);

long earliestDateLong = earliestDate.getTime().getTime();

System.out.println("Earliest Date = "

+ earliestDate.getTime().toString());

//计算从当前时间到最近一次执行时间的时间间隔

long delay = earliestDateLong - currentDateLong;

//计算执行周期为一星期

long period = 7 \* 24 \* 60 \* 60 \* 1000;

ScheduledExecutorService service = Executors.newScheduledThreadPool(10);

//从现在开始delay毫秒之后，每隔一星期执行一次job1

service.scheduleAtFixedRate(test, delay, period,

TimeUnit.MILLISECONDS);

}

}

Output:

Current Date = Wed Feb 02 17:32:01 CST 2011

Earliest Date = Tue Feb 8 16:38:10 CST 2011

Date = Tue Feb 8 16:38:10 CST 2011, execute job1

Date = Tue Feb 15 16:38:10 CST 2011, execute job1

清单 3 实现了每星期二 16:38:10 调度任务的功能。其核心在于根据当前时间推算出最近一个星期二 16:38:10 的绝对时间，然后计算与当前时间的时间差，作为调用 ScheduledExceutor 函数的参数。计算最近时间要用到 java.util.calendar 的功能。首先需要解释 calendar 的一些设计思想。Calendar 有以下几种唯一标识一个日期的组合方式：

YEAR + MONTH + DAY\_OF\_MONTH

YEAR + MONTH + WEEK\_OF\_MONTH + DAY\_OF\_WEEK

YEAR + MONTH + DAY\_OF\_WEEK\_IN\_MONTH + DAY\_OF\_WEEK

YEAR + DAY\_OF\_YEAR

YEAR + DAY\_OF\_WEEK + WEEK\_OF\_YEAR

上述组合分别加上 HOUR\_OF\_DAY + MINUTE + SECOND 即为一个完整的时间标识。本例采用了最后一种组合方式。输入为 DAY\_OF\_WEEK, HOUR\_OF\_DAY, MINUTE, SECOND 以及当前日期 , 输出为一个满足 DAY\_OF\_WEEK, HOUR\_OF\_DAY, MINUTE, SECOND 并且距离当前日期最近的未来日期。计算的原则是从输入的 DAY\_OF\_WEEK 开始比较，如果小于当前日期的 DAY\_OF\_WEEK，则需要向 WEEK\_OF\_YEAR 进一， 即将当前日期中的 WEEK\_OF\_YEAR 加一并覆盖旧值；如果等于当前的 DAY\_OF\_WEEK, 则继续比较 HOUR\_OF\_DAY；如果大于当前的 DAY\_OF\_WEEK，则直接调用 java.util.calenda 的 calendar.set(field, value) 函数将当前日期的 DAY\_OF\_WEEK, HOUR\_OF\_DAY, MINUTE, SECOND 赋值为输入值，依次类推，直到比较至 SECOND。读者可以根据输入需求选择不同的组合方式来计算最近执行时间。

可以看出，用上述方法实现该任务调度比较麻烦，这就需要一个更加完善的任务调度框架来解决这些复杂的调度问题。幸运的是，开源工具包 Quartz 与 JCronTab 提供了这方面强大的支持。

[**回页首**](https://www.ibm.com/developerworks/cn/java/j-lo-taskschedule/#ibm-pcon)

**Quartz**

Quartz 可以满足更多更复杂的调度需求，首先让我们看看如何用 Quartz 实现每星期二 16:38 的调度安排：

**清单 4. 使用 Quartz 进行任务调度**

package com.ibm.scheduler;

import java.util.Date;

import org.quartz.Job;

import org.quartz.JobDetail;

import org.quartz.JobExecutionContext;

import org.quartz.JobExecutionException;

import org.quartz.Scheduler;

import org.quartz.SchedulerFactory;

import org.quartz.Trigger;

import org.quartz.helpers.TriggerUtils;

public class QuartzTest implements Job {

@Override

//该方法实现需要执行的任务

public void execute(JobExecutionContext arg0) throws JobExecutionException {

System.out.println("Generating report - "

+ arg0.getJobDetail().getFullName() + ", type ="

+ arg0.getJobDetail().getJobDataMap().get("type"));

System.out.println(new Date().toString());

}

public static void main(String[] args) {

try {

// 创建一个Scheduler

SchedulerFactory schedFact =

new org.quartz.impl.StdSchedulerFactory();

Scheduler sched = schedFact.getScheduler();

sched.start();

// 创建一个JobDetail，指明name，groupname，以及具体的Job类名，

//该Job负责定义需要执行任务

JobDetail jobDetail = new JobDetail("myJob", "myJobGroup",

QuartzTest.class);

jobDetail.getJobDataMap().put("type", "FULL");

// 创建一个每周触发的Trigger，指明星期几几点几分执行

Trigger trigger = TriggerUtils.makeWeeklyTrigger(3, 16, 38);

trigger.setGroup("myTriggerGroup");

// 从当前时间的下一秒开始执行

trigger.setStartTime(TriggerUtils.getEvenSecondDate(new Date()));

// 指明trigger的name

trigger.setName("myTrigger");

// 用scheduler将JobDetail与Trigger关联在一起，开始调度任务

sched.scheduleJob(jobDetail, trigger);

} catch (Exception e) {

e.printStackTrace();

}

}

}

Output:

Generating report - myJobGroup.myJob, type =FULL

Tue Feb 8 16:38:00 CST 2011

Generating report - myJobGroup.myJob, type =FULL

Tue Feb 15 16:38:00 CST 2011

清单 4 非常简洁地实现了一个上述复杂的任务调度。Quartz 设计的核心类包括 Scheduler, Job 以及 Trigger。其中，Job 负责定义需要执行的任务，Trigger 负责设置调度策略，Scheduler 将二者组装在一起，并触发任务开始执行。

**Job**

使用者只需要创建一个 Job 的继承类，实现 execute 方法。JobDetail 负责封装 Job 以及 Job 的属性，并将其提供给 Scheduler 作为参数。每次 Scheduler 执行任务时，首先会创建一个 Job 的实例，然后再调用 execute 方法执行。Quartz 没有为 Job 设计带参数的构造函数，因此需要通过额外的 JobDataMap 来存储 Job 的属性。JobDataMap 可以存储任意数量的 Key，Value 对，例如：

**清单 5. 为 JobDataMap 赋值**

jobDetail.getJobDataMap().put("myDescription", "my job description");

jobDetail.getJobDataMap().put("myValue", 1998);

ArrayList<String> list = new ArrayList<String>();

list.add("item1");

jobDetail.getJobDataMap().put("myArray", list);

JobDataMap 中的数据可以通过下面的方式获取：

**清单 6. 获取 JobDataMap 的值**

public class JobDataMapTest implements Job {

@Override

public void execute(JobExecutionContext context)

throws JobExecutionException {

//从context中获取instName，groupName以及dataMap

String instName = context.getJobDetail().getName();

String groupName = context.getJobDetail().getGroup();

JobDataMap dataMap = context.getJobDetail().getJobDataMap();

//从dataMap中获取myDescription，myValue以及myArray

String myDescription = dataMap.getString("myDescription");

int myValue = dataMap.getInt("myValue");

ArrayList<String> myArray = (ArrayListlt;Strin>) dataMap.get("myArray");

System.out.println("

Instance =" + instName + ", group = " + groupName

+ ", description = " + myDescription + ", value =" + myValue

+ ", array item0 = " + myArray.get(0));

}

}

Output：

Instance = myJob, group = myJobGroup,

description = my job description,

value =1998, array item0 = item1

**Trigger**

Trigger 的作用是设置调度策略。Quartz 设计了多种类型的 Trigger，其中最常用的是 SimpleTrigger 和 CronTrigger。

SimpleTrigger 适用于在某一特定的时间执行一次，或者在某一特定的时间以某一特定时间间隔执行多次。上述功能决定了 SimpleTrigger 的参数包括 start-time, end-time, repeat count, 以及 repeat interval。

Repeat count 取值为大于或等于零的整数，或者常量 SimpleTrigger.REPEAT\_INDEFINITELY。

Repeat interval 取值为大于或等于零的长整型。当 Repeat interval 取值为零并且 Repeat count 取值大于零时，将会触发任务的并发执行。

Start-time 与 dnd-time 取值为 java.util.Date。当同时指定 end-time 与 repeat count 时，优先考虑 end-time。一般地，可以指定 end-time，并设定 repeat count 为 REPEAT\_INDEFINITELY。

以下是 SimpleTrigger 的构造方法：

public SimpleTrigger(String name,

String group,

Date startTime,

Date endTime,

int repeatCount,

long repeatInterval)

举例如下：

创建一个立即执行且仅执行一次的 SimpleTrigger：

SimpleTrigger trigger=

new SimpleTrigger("myTrigger", "myGroup", new Date(), null, 0, 0L);

创建一个半分钟后开始执行，且每隔一分钟重复执行一次的 SimpleTrigger：

SimpleTrigger trigger=

new SimpleTrigger("myTrigger", "myGroup",

new Date(System.currentTimeMillis()+30\*1000), null, 0, 60\*1000);

创建一个 2011 年 6 月 1 日 8:30 开始执行，每隔一小时执行一次，一共执行一百次，一天之后截止的 SimpleTrigger：

Calendar calendar = Calendar.getInstance();

calendar.set(Calendar.YEAR, 2011);

calendar.set(Calendar.MONTH, Calendar.JUNE);

calendar.set(Calendar.DAY\_OF\_MONTH, 1);

calendar.set(Calendar.HOUR, 8);

calendar.set(Calendar.MINUTE, 30);

calendar.set(Calendar.SECOND, 0);

calendar.set(Calendar.MILLISECOND, 0);

Date startTime = calendar.getTime();

Date endTime = new Date (calendar.getTimeInMillis() +24\*60\*60\*1000);

SimpleTrigger trigger=new SimpleTrigger("myTrigger",

"myGroup", startTime, endTime, 100, 60\*60\*1000);

上述最后一个例子中，同时设置了 end-time 与 repeat count，则优先考虑 end-time，总共可以执行二十四次。

CronTrigger 的用途更广，相比基于特定时间间隔进行调度安排的 SimpleTrigger，CronTrigger 主要适用于基于日历的调度安排。例如：每星期二的 16:38:10 执行，每月一号执行，以及更复杂的调度安排等。

CronTrigger 同样需要指定 start-time 和 end-time，其核心在于 Cron 表达式，由七个字段组成：

Seconds

Minutes

Hours

Day-of-Month

Month

Day-of-Week

Year (Optional field)

举例如下：

创建一个每三小时执行的 CronTrigger，且从每小时的整点开始执行：

0 0 0/3 \* \* ?

创建一个每十分钟执行的 CronTrigger，且从每小时的第三分钟开始执行：

0 3/10 \* \* \* ?

创建一个每周一，周二，周三，周六的晚上 20:00 到 23:00，每半小时执行一次的 CronTrigger：

0 0/30 20-23 ? \* MON-WED,SAT

创建一个每月最后一个周四，中午 11:30-14:30，每小时执行一次的 trigger：

0 30 11-14/1 ? \* 5L

解释一下上述例子中各符号的含义：

首先所有字段都有自己特定的取值，例如，Seconds 和 Minutes 取值为 0 到 59，Hours 取值为 0 到 23，Day-of-Month 取值为 0-31, Month 取值为 0-11，或者 JAN，FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC，Days-of-Week 取值为 1-7 或者 SUN, MON, TUE, WED, THU, FRI, SAT。每个字段可以取单个值，多个值，或一个范围，例如 Day-of-Week 可取值为“MON，TUE，SAT”,“MON-FRI”或者“TUE-THU，SUN”。

通配符 \* 表示该字段可接受任何可能取值。例如 Month 字段赋值 \* 表示每个月，Day-of-Week 字段赋值 \* 表示一周的每天。

/ 表示开始时刻与间隔时段。例如 Minutes 字段赋值 2/10 表示在一个小时内每 20 分钟执行一次，从第 2 分钟开始。

? 仅适用于 Day-of-Month 和 Day-of-Week。? 表示对该字段不指定特定值。适用于需要对这两个字段中的其中一个指定值，而对另一个不指定值的情况。一般情况下，这两个字段只需对一个赋值。

L 仅适用于 Day-of-Month 和 Day-of-Week。L 用于 Day-of-Month 表示该月最后一天。L 单独用于 Day-of-Week 表示周六，否则表示一个月最后一个星期几，例如 5L 或者 THUL 表示该月最后一个星期四。

W 仅适用于 Day-of-Month，表示离指定日期最近的一个工作日，例如 Day-of-Month 赋值为 10W 表示该月离 10 号最近的一个工作日。

# 仅适用于 Day-of-Week，表示该月第 XXX 个星期几。例如 Day-of-Week 赋值为 5#2 或者 THU#2，表示该月第二个星期四。

CronTrigger 的使用如下：

CronTrigger cronTrigger = new CronTrigger("myTrigger", "myGroup");

try {

cronTrigger.setCronExpression("0 0/30 20-13 ? \* MON-WED,SAT");

} catch (Exception e) {

e.printStackTrace();

}

Job 与 Trigger 的松耦合设计是 Quartz 的一大特点，其优点在于同一个 Job 可以绑定多个不同的 Trigger，同一个 Trigger 也可以调度多个 Job，灵活性很强。

**Listener**

除了上述基本的调度功能，Quartz 还提供了 listener 的功能。主要包含三种 listener：JobListener，TriggerListener 以及 SchedulerListener。当系统发生故障，相关人员需要被通知时，Listener 便能发挥它的作用。最常见的情况是，当任务被执行时，系统发生故障，Listener 监听到错误，立即发送邮件给管理员。下面给出 JobListener 的实例：

**清单 7. JobListener 的实现**

import org.quartz.JobExecutionContext;

import org.quartz.JobExecutionException;

import org.quartz.JobListener;

import org.quartz.SchedulerException;

public class MyListener implements JobListener{

@Override

public String getName() {

return "My Listener";

}

@Override

public void jobWasExecuted(JobExecutionContext context,

JobExecutionException jobException) {

if(jobException != null){

try {

//停止Scheduler

context.getScheduler().shutdown();

System.out.println("

Error occurs when executing jobs, shut down the scheduler ");

// 给管理员发送邮件…

} catch (SchedulerException e) {

e.printStackTrace();

}

}

}

}

从清单 7 可以看出，使用者只需要创建一个 JobListener 的继承类，重载需要触发的方法即可。当然，需要将 listener 的实现类注册到 Scheduler 和 JobDetail 中：

sched.addJobListener(new MyListener());

jobDetail.addJobListener("My Listener"); // listener 的名字

使用者也可以将 listener 注册为全局 listener，这样便可以监听 scheduler 中注册的所有任务 :

sched.addGlobalJobListener(new MyListener());

为了测试 listener 的功能，可以在 job 的 execute 方法中强制抛出异常。清单 7 中，listener 接收到异常，将 job 所在的 scheduler 停掉，阻止后续的 job 继续执行。scheduler、jobDetail 等信息都可以从 listener 的参数 context 中检索到。

清单 7 的输出结果为：

Generating report - myJob.myJob, type =FULL

Tue Feb 15 18:57:35 CST 2011

2011-2-15 18:57:35 org.quartz.core.JobRunShell run

信息 : Job myJob.myJob threw a JobExecutionException:

org.quartz.JobExecutionException

at com.ibm.scheduler.QuartzListenerTest.execute(QuartzListenerTest.java:22)

at org.quartz.core.JobRunShell.run(JobRunShell.java:191)

at org.quartz.simpl.SimpleThreadPool$WorkerThread.run(SimpleThreadPool.java:516)

2011-2-15 18:57:35 org.quartz.core.QuartzScheduler shutdown

信息 : Scheduler DefaultQuartzScheduler\_$\_NON\_CLUSTERED shutting down.

Error occurs when executing jobs, shut down the scheduler

TriggerListener、SchedulerListener 与 JobListener 有类似的功能，只是各自触发的事件不同，如 JobListener 触发的事件为：

Job to be executed, Job has completed execution 等

TriggerListener 触发的事件为：

Trigger firings, trigger mis-firings, trigger completions 等

SchedulerListener 触发的事件为：

add a job/trigger, remove a job/trigger, shutdown a scheduler 等

读者可以根据自己的需求重载相应的事件。

**JobStores**

Quartz 的另一显著优点在于持久化，即将任务调度的相关数据保存下来。这样，当系统重启后，任务被调度的状态依然存在于系统中，不会丢失。默认情况下，Quartz 采用的是 org.quartz.simpl.RAMJobStore，在这种情况下，数据仅能保存在内存中，系统重启后会全部丢失。若想持久化数据，需要采用 org.quartz.simpl.JDBCJobStoreTX。

实现持久化的第一步，是要创建 Quartz 持久化所需要的表格。在 Quartz 的发布包 docs/dbTables 中可以找到相应的表格创建脚本。Quartz 支持目前大部分流行的数据库。本文以 DB2 为例，所需要的脚本为 tables\_db2.sql。首先需要对脚本做一点小的修改，即在开头指明 Schema：

SET CURRENT SCHEMA quartz;

为了方便重复使用 , 创建表格前首先删除之前的表格：  
drop table qrtz\_job\_details;

drop table qrtz\_job\_listeners;

…

然后创建数据库 sched，执行 tables\_db2.sql 创建持久化所需要的表格。

第二步，配置数据源。数据源与其它所有配置，例如 ThreadPool，均放在 quartz.properties 里：

**清单 8. Quartz 配置文件**

# Configure ThreadPool

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

org.quartz.threadPool.threadCount = 5

org.quartz.threadPool.threadPriority = 4

# Configure Datasources

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

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

org.quartz.jobStore.dataSource = db2DS

org.quartz.jobStore.tablePrefix = QRTZ\_

org.quartz.dataSource.db2DS.driver = com.ibm.db2.jcc.DB2Driver

org.quartz.dataSource.db2DS.URL = jdbc:db2://localhost:50001/sched

org.quartz.dataSource.db2DS.user = quartz

org.quartz.dataSource.db2DS.password = passw0rd

org.quartz.dataSource.db2DS.maxConnections = 5

使用时只需要将 quatz.properties 放在 classpath 下面，不用更改一行代码，再次运行之前的任务调度实例，trigger、job 等信息便会被记录在数据库中。

将清单 4 中的 makeWeeklyTrigger 改成 makeSecondlyTrigger，重新运行 main 函数，在 sched 数据库中查询表 qrtz\_simple\_triggers 中的数据。其查询语句为“db2 ‘ select repeat\_interval, times\_triggered from qrtz\_simple\_triggers ’”。结果 repeat\_interval 为 1000，与程序中设置的 makeSecondlyTrigger 相吻合，times\_triggered 值为 21。

停掉程序，将数据库中记录的任务调度数据重新导入程序运行：

**清单 9. 从数据库中导入任务调度数据重新运行**

package com.ibm.scheduler;

import org.quartz.Scheduler;

import org.quartz.SchedulerException;

import org.quartz.SchedulerFactory;

import org.quartz.Trigger;

import org.quartz.impl.StdSchedulerFactory;

public class QuartzReschedulerTest {

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

// 初始化一个 Schedule Factory

SchedulerFactory schedulerFactory = new StdSchedulerFactory();

// 从 schedule factory 中获取 scheduler

Scheduler scheduler = schedulerFactory.getScheduler();

// 从 schedule factory 中获取 trigger

Trigger trigger = scheduler.getTrigger("myTrigger", "myTriggerGroup");

// 重新开启调度任务

scheduler.rescheduleJob("myTrigger", "myTriggerGroup", trigger);

scheduler.start();

}

}

清单 9 中，schedulerFactory.getScheduler() 将 quartz.properties 的内容加载到内存，然后根据数据源的属性初始化数据库的链接，并将数据库中存储的数据加载到内存。之后，便可以在内存中查询某一具体的 trigger，并将其重新启动。这时候重新查询 qrtz\_simple\_triggers 中的数据，发现 times\_triggered 值比原来增长了。

[**回页首**](https://www.ibm.com/developerworks/cn/java/j-lo-taskschedule/#ibm-pcon)

**JCronTab**

习惯使用 unix/linux 的开发人员应该对 crontab 都不陌生。Crontab 是一个非常方便的用于 unix/linux 系统的任务调度命令。JCronTab 则是一款完全按照 crontab 语法编写的 java 任务调度工具。

首先简单介绍一下 crontab 的语法，与上面介绍的 Quartz 非常相似，但更加简洁 , 集中了最常用的语法。主要由六个字段组成（括弧中标识了每个字段的取值范围）：

Minutes （0-59）

Hours （0-23）

Day-of-Month （1-31）

Month （1-12/JAN-DEC）

Day-of-Week （0-6/SUN-SAT）

Command

与 Quartz 相比，省略了 Seconds 与 Year，多了一个 command 字段，即为将要被调度的命令。JCronTab 中也包含符号“\*”与“/”, 其含义与 Quartz 相同。

举例如下：

每天 12 点到 15 点 , 每隔 1 小时执行一次 Date 命令：

0 12-15/1 \* \* \* Date

每月 2 号凌晨 1 点发一封信给 zhjingbj@cn.ibm.com：

0 1 2 \* \* mail -s “good” zhjingbj@cn.ibm.com

每周一，周二，周三，周六的晚上 20:00 到 23:00，每半小时打印“normal”：

0/30 20-23 \* \* MON-WED,SAT echo “normal”

JCronTab 借鉴了 crontab 的语法，其区别在于 command 不再是 unix/linux 的命令，而是一个 Java 类。如果该类带参数，例如“com.ibm.scheduler.JCronTask2#run”，则定期执行 run 方法；如果该类不带参数，则默认执行 main 方法。此外，还可以传参数给 main 方法或者构造函数，例如“com.ibm.scheduler.JCronTask2#run Hello World“表示传两个参数 Hello 和 World 给构造函数。

JCronTab 与 Quartz 相比，其优点在于，第一，支持多种任务调度的持久化方法，包括普通文件、数据库以及 XML 文件进行持久化；第二，JCronTab 能够非常方便地与 Web 应用服务器相结合，任务调度可以随 Web 应用服务器的启动自动启动；第三，JCronTab 还内置了发邮件功能，可以将任务执行结果方便地发送给需要被通知的人。

JCronTab 与 Web 应用服务器的结合非常简单，只需要在 Web 应用程序的 web.xml 中添加如下行：

**清单 10. 在 web.xml 中配置 JCronTab 的属性**

<servlet>

<servlet-name>LoadOnStartupServlet</servlet-name>

<servlet-class>org.jcrontab.web.loadCrontabServlet</servlet-class>

<init-param>

<param-name>PROPERTIES\_FILE</param-name>

<param-value>D:/Scheduler/src/jcrontab.properties</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<!-- Mapping of the StartUp Servlet -->

<servlet-mapping>

<servlet-name>LoadOnStartupServlet</servlet-name>

<url-pattern>/Startup</url-pattern>

</servlet-mapping>

在清单 10 中，需要注意两点：第一，必须指定 servlet-class 为 org.jcrontab.web.loadCrontabServlet，因为它是整个任务调度的入口；第二，必须指定一个参数为 PROPERTIES\_FILE，才能被 loadCrontabServlet 识别。

接下来，需要撰写 D:/Scheduler/src/jcrontab.properties 的内容，其内容根据需求的不同而改变。

当采用普通文件持久化时，jcrontab.properties 的内容主要包括：

org.jcrontab.data.file = D:/Scheduler/src/crontab

org.jcrontab.data.datasource = org.jcrontab.data.FileSource

其中数据来源 org.jcrontab.data.datasource 被描述为普通文件，即 org.jcrontab.data.FileSource。具体的文件即 org.jcrontab.data.file 指明为 D:/Scheduler/src/crontab。

Crontab 描述了任务的调度安排：

\*/2 \* \* \* \* com.ibm.scheduler.JCronTask1

\* \* \* \* \* com.ibm.scheduler.JCronTask2#run Hello World

其中包含了两条任务的调度，分别是每两分钟执行一次 JCronTask1 的 main 方法，每一分钟执行一次 JCronTask2 的 run 方法。

**清单 11. JcronTask1 与 JCronTask2 的实现**

package com.ibm.scheduler;

import java.util.Date;

public class JCronTask1 {

private static int count = 0;

public static void main(String[] args) {

System.out.println("--------------Task1-----------------");

System.out.println("Current Time = " + new Date() + ", Count = "

+ count++);

}

}

package com.ibm.scheduler;

import java.util.Date;

public class JCronTask2 implements Runnable {

private static int count = 0;

private static String[] args;

public JCronTask2(String[] args) {

System.out.println("--------------Task2-----------------");

System.out.println("Current Time = " + new Date() + ", Count = "

+ count++);

JCronTask2.args = args;

}

@Override

public void run() {

System.out.println("enter into run method");

if (args != null && args.length > 0) {

for (int i = 0; i < args.length; i++) {

System.out.print("This is arg " + i + " " + args[i] + "\n");

}

}

}

}

到此为止，基于普通文件持久化的 JCronTab 的实例就全部配置好了。启动 Web 应用服务器，便可以看到任务调度的输出结果：

--------------Task2-----------------

Current Time = Tue Feb 15 09:22:00 CST 2011, Count = 0

enter into run method

This is arg 0 Hello

This is arg 1 World

--------------Task1-----------------

Current Time = Tue Feb 15 09:22:00 CST 2011, Count = 0

--------------Task2-----------------

Current Time = Tue Feb 15 09:23:00 CST 2011, Count = 1

enter into run method

This is arg 0 Hello

This is arg 1 World

--------------Task2-----------------

Current Time = Tue Feb 15 09:24:00 CST 2011, Count = 2

enter into run method

This is arg 0 Hello

This is arg 1 World

--------------Task1-----------------

Current Time = Tue Feb 15 09:24:00 CST 2011, Count = 1

通过修改 jcrontab.properties 中 datasource，可以选择采用数据库或 xml 文件持久化，感兴趣的读者可以参考 [进阶学习 JCronTab](http://blog.csdn.net/maskice/archive/2007/06/28/1670070.aspx)。

此外，JCronTab 还内置了发邮件功能，可以将任务执行结果方便地发送给需要被通知的人。其配置非常简单，只需要在 jcontab.properties 中添加几行配置即可：

org.jcrontab.sendMail.to= Ther email you want to send to

org.jcrontab.sendMail.from=The email you want to send from

org.jcrontab.sendMail.smtp.host=smtp server

org.jcrontab.sendMail.smtp.user=smtp username

org.jcrontab.sendMail.smtp.password=smtp password

[**回页首**](https://www.ibm.com/developerworks/cn/java/j-lo-taskschedule/#ibm-pcon)

**结束语**

本文介绍了四种常用的对任务进行调度的 Java 实现方法，即 Timer，ScheduledExecutor, Quartz 以及 JCronTab。文本对每种方法都进行了实例解释，并对其优缺点进行比较。对于简单的基于起始时间点与时间间隔的任务调度，使用 Timer 就足够了；如果需要同时调度多个任务，基于线程池的 ScheduledTimer 是更为合适的选择；当任务调度的策略复杂到难以凭借起始时间点与时间间隔来描述时，Quartz 与 JCronTab 则体现出它们的优势。熟悉 Unix/Linux 的开发人员更倾向于 JCronTab，且 JCronTab 更适合与 Web 应用服务器相结合。Quartz 的 Trigger 与 Job 松耦合设计使其更适用于 Job 与 Trigger 的多对多应用场景。

**参考资料**

**学习**

* [周期性任务调度](http://www.blogjava.net/xylz/archive/2011/01/10/342738.html)：ScheduledExecutor 周期性任务调度前世今生
* [用 Quartz 进行作业调度](http://www.ibm.com/developerworks/cn/java/j-quartz/#listing4): Quartz API 采用多面方式在 Java 应用程序中进行任务调度
* [Job Scheduling in Java](http://oreilly.com/pub/a/java/archive/quartz.html?page=2): Java 中的任务调度
* [Quartz Enterprise Job scheduler Tutorial](http://www.quartz-scheduler.org/docs/tutorial/index.html): Quartz 官方说明书
* [多种计时器的比较与分析](http://www.ibm.com/developerworks/cn/java/j-lo-timer/): java.util.Timer, Commonj Timer 和 IBM WAS Scheduler 的比较
* [Java 开源日程安排](http://man.lupaworld.com/content/develop/open-open/14.htm): 关于任务调度的 Java 开源工具总结
* [进阶学习 JCronTab](http://blog.csdn.net/maskice/archive/2007/06/28/1670070.aspx): 关于任务调度开源工具 JCronTab 的学习
* [JCronTab](http://jcrontab.sourceforge.net/index.shtml):JCronTab 官网
* [developerWorks Web development 专区](http://www.ibm.com/developerworks/cn/web/)：通过专门关于 Web 技术的文章和教程，扩展您在网站开发方面的技能。
* [developerWorks Ajax 资源中心](http://www.ibm.com/developerworks/cn/ajax/)：这是有关 Ajax 编程模型信息的一站式中心，包括很多文档、教程、论坛、blog、wiki 和新闻。任何 Ajax 的新信息都能在这里找到。
* [developerWorks Web 2.0 资源中心](http://www.ibm.com/developerworks/cn/web20/)，这是有关 Web 2.0 相关信息的一站式中心，包括大量 Web 2.0 技术文章、教程、下载和相关技术资源。您还可以通过 [Web 2.0 新手入门](http://www.ibm.com/developerworks/cn/web20/newto/) 栏目，迅速了解 Web 2.0 的相关概念。
* 查看 [HTML5 专题](http://www.ibm.com/developerworks/cn/web/lp/html5/)，了解更多和 HTML5 相关的知识和动向。

**讨论**

* 加入 [developerWorks 中文社区](http://www.ibm.com/developerworks/cn/community/)。查看开发人员推动的博客、论坛、组和维基，并与其他 developerWorks 用户交流。