幾種任務調度的 Java 實現方法與比較

ibm.com/developerworks/cn/java/j-lo-taskschedule/

前言

任務調度是指基於給定時間點，給定時間間隔或者給定執行次數自動執行任務。本文由淺入深介紹四種任務調度的 Java 實現：

- Timer
- ScheduledExecutor
- 開源工具包 Quartz
- 開源工具包 JCronTab

此外，為結合實現複雜的任務調度，本文還將介紹 Calendar 的一些使用方法。

回頁首

Timer

相信大家都已經非常熟悉 java.util.Timer 了，它是最簡單的一種實現任務調度的方法，下面給出一個具體的例子：

清單 1. 使用 Timer 進行任務調度

```
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;
// 從現在開始 1 秒鐘之後，每隔 1 秒鐘執行一次 job1
timer.schedule(new TimerTest("job1"), delay1, period1);
```

```
long delay2 = 2 * 1000;
long period2 = 2000;
// 從現在開始 2 秒鐘之後，每隔 2 秒鐘執行一次 job2
timer.schedule(new TimerTest("job2"), delay2, period2);
}
}
Output:
execute job1
execute job1
execute job2
execute job1
execute job1
execute job2
```

使用 Timer 實現任務調度的核心類是 Timer 和 TimerTask。其中 Timer 負責設定 TimerTask 的起始與間隔執行時間。使用者只需要創建一個 TimerTask 的繼承類，實現自己的 run 方法，然後將其丟給 Timer 去執行即可。

Timer 的設計核心是一個 TaskList 和一個 TaskThread。Timer 將接收到的任務丟到自己的 TaskList 中，TaskList 按照 Task 的最初執行時間進行排序。TimerThread 在創建 Timer 時會啟動成為一個守護線程。這個線程會輪詢所有任務，找到一個最近要執行的任務，然後休眠，當到達最近要執行任務的開始時間點，TimerThread 被喚醒並執行該任務。之後 TimerThread 更新最近一個要執行的任務，繼續休眠。

Timer 的優點在於簡單易用，但由於所有任務都是由同一個線程來調度，因此所有任務都是串行執行的，同一時間只能有一個任務在執行，前一個任務的延遲或異常都將會影響到之後的任務。

回頁首

ScheduledExecutor

鑑於 Timer 的上述缺陷，Java 5 推出了基於線程池設計的 ScheduledExecutor。其設計思想是，每一個被調度的任務都會由線程池中一個線程去執行，因此任務是並發執行的，相互之間不會受到干擾。需要注意的是，只有當任務的執行時間到來時，ScheduedExecutor 才會真正啟動一個線程，其餘時間 ScheduledExecutor 都是在輪詢任務的狀態。

清單 2. 使用 ScheduledExecutor 進行任務調度

```
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;
        // 從現在開始1秒鐘之後，每隔1秒鐘執行一次job1
  service.scheduleAtFixedRate(
          new ScheduledExecutorTest("job1"), initialDelay1,
    period1, TimeUnit.SECONDS);

  long initialDelay2 = 1;
  long delay2 = 1;
        // 從現在開始2秒鐘之後，每隔2秒鐘執行一次job2
  service.scheduleWithFixedDelay(
          new ScheduledExecutorTest("job2"), initialDelay2,
    delay2, TimeUnit.SECONDS);
 }
}
Output:
execute job1
execute job1
execute job2
execute job1
execute job1
execute job2
```

清單 2 展示了 ScheduledExecutorService 中兩種最常用的調度方法 ScheduleAtFixedRate 和 ScheduleWithFixedDelay。ScheduleAtFixedRate 每次執行時間為上一次任務開始起向後推一個時間間隔，即每次執行時間為 :initialDelay, initialDelay+period, initialDelay+2*period, …；ScheduleWithFixedDelay 每次執行時間為上一次任務結束起向後推一個時間間隔，即每次執行時間為：initialDelay, initialDelay+executeTime+delay, initialDelay+2*executeTime+2*delay。由此可見，ScheduleAtFixedRate 是基於固定時間間隔進行任務調度，ScheduleWithFixedDelay 取決於每次任務執行的時間長短，是基於不固定時間間隔進行任務調度。

回頁首

用 ScheduledExecutor 和 Calendar 實現複雜任務調度

Timer 和 ScheduledExecutor 都僅能提供基於開始時間與重複間隔的任務調度，不能勝任更加複雜的調度需求。比如，設置每星期二的 16:38:10 執行任務。該功能使用 Timer 和 ScheduledExecutor 都不能直接實現，但我們可以借助 Calendar 間接實現該功能。

清單 3. 使用 ScheduledExcetuor 和 Calendar 進行任務調度

```
 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);
 }

 /**
  * 計算從當前時間currentDate開始，滿足條件dayOfWeek, hourOfDay,
  * minuteOfHour, secondOfMinite的最近時間
  * @return
  */
 public Calendar getEarliestDate(Calendar currentDate, int dayOfWeek,
   int hourOfDay, int minuteOfHour, int secondOfMinite) {
  //計算當前時間的WEEK_OF_YEAR,DAY_OF_WEEK, HOUR_OF_DAY, MINUTE,SECOND等各個字段值
  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);

  //如果輸入條件中的dayOfWeek小於當前日期的dayOfWeek,則WEEK_OF_YEAR需要推遲一週
  boolean weekLater = false;
  if (dayOfWeek < currentDayOfWeek) {
   weekLater = true;
  } else if (dayOfWeek == currentDayOfWeek) {
   //當輸入條件與當前日期的dayOfWeek相等時，如果輸入條件中的
   //hourOfDay小於當前日期的
   //currentHour，則WEEK_OF_YEAR需要推遲一週
   if (hourOfDay < currentHour) {
    weekLater = true;
   } else if (hourOfDay == currentHour) {
                 //當輸入條件與當前日期的dayOfWeek, hourOfDay相等時,
                 //如果輸入條件中的minuteOfHour小於當前日期的
    //currentMinute，則WEEK_OF_YEAR需要推遲一週
    if (minuteOfHour < currentMinute) {
     weekLater = true;
    } else if (minuteOfHour == currentSecond) {
```

```
                        //當輸入條件與當前日期的dayOfWeek, hourOfDay,
                        //minuteOfHour相等時, 如果輸入條件中的
                       //secondOfMinite小於當前日期的currentSecond,
                       //則WEEK_OF_YEAR需要推遲一週
     if (secondOfMinite < currentSecond) {
      weekLater = true;
     }
    }
   }
  }
  if (weekLater) {
   //設置當前日期中的WEEK_OF_YEAR為當前周推遲一週
   currentDate.set(Calendar.WEEK_OF_YEAR, currentWeekOfYear + 1);
  }
  // 設置當前日期中的DAY_OF_WEEK,HOUR_OF_DAY,MINUTE,SECOND為輸入條件中的值。
  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");
  //獲取當前時間
  Calendar currentDate = Calendar.getInstance();
  long currentDateLong = currentDate.getTime().getTime();
  System.out.println("Current Date = " + currentDate.getTime().toString());
  //計算滿足條件的最近一次執行時間
  Calendar earliestDate = test
    .getEarliestDate(currentDate, 3, 16, 38, 10);
  long earliestDateLong = earliestDate.getTime().getTime();
  System.out.println("Earliest Date = "
    + earliestDate.getTime().toString());
  //計算從當前時間到最近一次執行時間的時間間隔
  long delay = earliestDateLong - currentDateLong;
  //計算執行週期為一星期
  long period = 7 * 24 * 60 * 60 * 1000;
  ScheduledExecutorService service = Executors.newScheduledThreadPool(10);
  //從現在開始delay毫秒之後, 每隔一星期執行一次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
```

清單 3 實現了每星期二 16:38:10 調度任務的功能。其核心在於根據當前時間推算出最近一個星期二 16:38:10 的絕對時間，然後計算與當前時間的時間差，作為調用 ScheduledExceutor 函數的參數。計算最近時間要用到 java.util.calendar 的功能。首先需要解釋 calendar 的一些設計思想。Calendar 有以下幾種唯一標識一個日期的組合方式：

```
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
```

上述組合分別加上 HOUR_OF_DAY + MINUTE + SECOND 即為一個完整的時間標識。本例採用了最後一種組合方式。輸入為 DAY_OF_WEEK, HOUR_OF_DAY, MINUTE, SECOND 以及當前日期，輸出為一個滿足 DAY_OF_WEEK, HOUR_OF_DAY, MINUTE, SECOND 並且距離當前日期最近的未來日期。計算的原則是從輸入的 DAY_OF_WEEK 開始比較，如果小於當前日期的 DAY_OF_WEEK，則需要向 WEEK_OF_YEAR 進一， 即將當前日期中的 WEEK_OF_YEAR 加一併覆蓋舊值；如果等於當前的 DAY_OF_WEEK, 則繼續比較 HOUR_OF_DAY；如果大於當前的 DAY_OF_WEEK，則直接調用 java.util.calenda 的 calendar.set(field, value) 函數將當前日期的 DAY_OF_WEEK, HOUR_OF_DAY, MINUTE, SECOND 賦值為輸入值，依次類推，直到比較至 SECOND。讀者可以根據輸入需求選擇不同的組合方式來計算最近執行時間。

可以看出，用上述方法實現該任務調度比較麻煩，這就需要一個更加完善的任務調度框架來解決這些複雜的調度問題。幸運的是，開源工具包 Quartz 與 JCronTab 提供了這方面強大的支持。

回頁首

Quartz

Quartz 可以滿足更多更複雜的調度需求，首先讓我們看看如何用 Quartz 實現每星期二 16:38 的調度安排：

清單 4. 使用 Quartz 進行任務調度

```
 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
 //該方法實現需要執行的任務
 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 {
   // 創建一個Scheduler
   SchedulerFactory schedFact =
   new org.quartz.impl.StdSchedulerFactory();
   Scheduler sched = schedFact.getScheduler();
   sched.start();
   // 創建一個JobDetail，指明name，groupname，以及具體的Job類名，
   //該Job負責定義需要執行任務
   JobDetail jobDetail = new JobDetail("myJob", "myJobGroup",
     QuartzTest.class);
   jobDetail.getJobDataMap().put("type", "FULL");
            // 創建一個每週觸發的Trigger，指明星期幾幾點幾分執行
   Trigger trigger = TriggerUtils.makeWeeklyTrigger(3, 16, 38);
   trigger.setGroup("myTriggerGroup");
   // 從當前時間的下一秒開始執行
   trigger.setStartTime(TriggerUtils.getEvenSecondDate(new Date()));
   // 指明trigger的name
   trigger.setName("myTrigger");
   // 用scheduler將JobDetail與Trigger關聯在一起，開始調度任務
   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
```

清單 4 非常簡潔地實現了一個上述複雜的任務調度。Quartz 設計的核心類包括 Scheduler, Job 以及 Trigger。其中，Job 負責定義需要執行的任務，Trigger 負責設置調度策略，Scheduler 將二者組裝在一起，並觸發任務開始執行。

Job

使用者只需要創建一個 Job 的繼承類，實現 execute 方法。JobDetail 負責封裝 Job 以及 Job 的屬性，並將其提供給 Scheduler 作為參數。每次 Scheduler 執行任務時，首先會創建一個 Job 的實例，然後再調用 execute 方法執行。Quartz 沒有為 Job 設計帶參數的構造函數，因此需要通過額外的 JobDataMap 來存儲 Job 的屬性。JobDataMap 可以存儲任意數量的 Key，Value 對，例如：

清單 5. 為 JobDataMap 賦值

```
 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 中的數據可以通過下面的方式獲取：

清單 6. 獲取 JobDataMap 的值

```
 public class JobDataMapTest implements Job {

 @Override
 public void execute(JobExecutionContext context)
   throws JobExecutionException {
  //從context中獲取instName, groupName以及dataMap
  String instName = context.getJobDetail().getName();
  String groupName = context.getJobDetail().getGroup();
  JobDataMap dataMap = context.getJobDetail().getJobDataMap();
  //從dataMap中獲取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 的作用是設置調度策略。Quartz 設計了多種類型的 Trigger，其中最常用的是 SimpleTrigger 和 CronTrigger。

SimpleTrigger 適用於在某一特定的時間執行一次，或者在某一特定的時間以某一特定時間間隔執行多次。上述功能決定了 SimpleTrigger 的參數包括 start-time, end-time, repeat count, 以及 repeat interval。

Repeat count 取值為大於或等於零的整數，或者常量 SimpleTrigger.REPEAT_INDEFINITELY。

Repeat interval 取值為大於或等於零的長整型。當 Repeat interval 取值為零並且 Repeat count 取值大於零時，將會觸發任務的並發執行。

Start-time 與 dnd-time 取值為 java.util.Date。當同時指定 end-time 與 repeat count 時，優先考慮 end-time。一般地，可以指定 end-time，並設定 repeat count 為 REPEAT_INDEFINITELY。

以下是 SimpleTrigger 的構造方法：

```
public SimpleTrigger(String name,
                     String group,
                     Date startTime,
                     Date endTime,
                     int repeatCount,
                     long repeatInterval)
```

舉例如下：

創建一個立即執行且僅執行一次的 SimpleTrigger：

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

創建一個半分鐘後開始執行，且每隔一分鐘重複執行一次的 SimpleTrigger：

```
SimpleTrigger trigger=
new SimpleTrigger("myTrigger", "myGroup",
   new Date(System.currentTimeMillis()+30*1000), null, 0, 60*1000);
```

創建一個 2011 年 6 月 1 日 8:30 開始執行，每隔一小時執行一次，一共執行一百次，一天之後截止的 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);
```

上述最後一個例子中，同時設置了 end-time 與 repeat count，則優先考慮 end-time，總共可以執行二十四次。

CronTrigger 的用途更廣，相比基於特定時間間隔進行調度安排的 SimpleTrigger，CronTrigger 主要適用於基於日曆的調度安排。例如：每星期二的 16:38:10 執行，每月一號執行，以及更複雜的調度安排等。

CronTrigger 同樣需要指定 start-time 和 end-time，其核心在於 Cron 表達式，由七個字段組成：

```
Seconds
Minutes
Hours
Day-of-Month
Month
Day-of-Week
Year (Optional field)
```

舉例如下：

創建一個每三小時執行的 CronTrigger，且從每小時的整點開始執行：

```
0 0 0/3  * * ?
```

創建一個每十分鐘執行的 CronTrigger，且從每小時的第三分鐘開始執行：

```
0 3/10 * * * ?
```

創建一個每週一，週二，週三，週六的晚上 20:00 到 23:00，每半小時執行一次的 CronTrigger：

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

創建一個每月最後一個週四，中午 11:30-14:30，每小時執行一次的 trigger：

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

解釋一下上述例子中各符號的含義：

首先所有字段都有自己特定的取值，例如，Seconds 和 Minutes 取值為 0 到 59，Hours 取值為 0 到 23，Day-of-Month 取值為 0-31, Month 取值為 0-11，或者 JAN，FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC，Days-of-Week 取值為 1-7 或者 SUN, MON, TUE, WED, THU, FRI, SAT。每個字段可以取單個值，多個值，或一個範圍，例如 Day-of-Week 可取值為「MON，TUE，SAT」，「MON-FRI」或者「TUE-THU，SUN」。

通配符 * 表示該字段可接受任何可能取值。例如 Month 字段賦值 * 表示每個月，Day-of-Week 字段賦值 * 表示一週的每天。

/ 表示開始時刻與間隔時段。例如 Minutes 字段賦值 2/10 表示在一個小時內每 20 分鐘執行一次，從第 2 分鐘開始。

? 僅適用於 Day-of-Month 和 Day-of-Week。? 表示對該字段不指定特定值。適用於需要對這兩個字段中的其中一個指定值，而對另一個不指定值的情況。一般情況下，這兩個字段只需對一個賦值。

L 僅適用於 Day-of-Month 和 Day-of-Week。L 用於 Day-of-Month 表示該月最後一天。L 單獨用於 Day-of-Week 表示週六，否則表示一個月最後一個星期幾，例如 5L 或者 THUL 表示該月最後一個星期四。

W 僅適用於 Day-of-Month，表示離指定日期最近的一個工作日，例如 Day-of-Month 賦值為 10W 表示該月離 10 號最近的一個工作日。

僅適用於 Day-of-Week，表示該月第 XXX 個星期幾。例如 Day-of-Week 賦值為 5#2 或者 THU#2，表示該月第二個星期四。

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 與 Trigger 的松耦合設計是 Quartz 的一大特點，其優點在於同一個 Job 可以綁定多個不同的 Trigger，同一個 Trigger 也可以調度多個 Job，靈活性很強。

Listener

除了上述基本的調度功能，Quartz 還提供了 listener 的功能。主要包含三種 listener：JobListener，TriggerListener 以及 SchedulerListener。當系統發生故障，相關人員需要被通知時，Listener 便能發揮它的作用。最常見的情況是，當任務被執行時，系統發生故障，Listener 監聽到錯誤，立即發送郵件給管理員。下面給出 JobListener 的實例：

清單 7. JobListener 的實現

```
 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 ");
                // 給管理員發送郵件…
   } catch (SchedulerException e) {
    e.printStackTrace();
   }
  }
 }
}
```

從清單 7 可以看出，使用者只需要創建一個 JobListener 的繼承類，重載需要觸發的方法即可。當然，需要將 listener 的實現類註冊到 Scheduler 和 JobDetail 中：

```
 sched.addJobListener(new MyListener());
 jobDetail.addJobListener("My Listener"); // listener 的名字
```

使用者也可以將 listener 註冊為全局 listener，這樣便可以監聽 scheduler 中註冊的所有任務：

```
 sched.addGlobalJobListener(new MyListener());
```

為了測試 listener 的功能，可以在 job 的 execute 方法中強制拋出異常。清單 7 中，listener 接收到異常，將 job 所在的 scheduler 停掉，阻止後續的 job 繼續執行。scheduler、jobDetail 等信息都可以從 listener 的參數 context 中檢索到。

清單 7 的輸出結果為：

```
 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 與 JobListener 有類似的功能，只是各自觸發的事件不同，如 JobListener 觸發的事件為：

Job to be executed, Job has completed execution 等

TriggerListener 觸發的事件為：

Trigger firings, trigger mis-firings, trigger completions 等

SchedulerListener 觸發的事件為：

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

讀者可以根據自己的需求重載相應的事件。

JobStores

Quartz 的另一顯著優點在於持久化，即將任務調度的相關數據保存下來。這樣，當系統重啟後，任務被調度的狀態依然存在於系統中，不會丟失。默認情況下，Quartz 採用的是 org.quartz.simpl.RAMJobStore，在這種情況下，數據僅能保存在內存中，系統重啟後會全部丟失。若想持久化數據，需要採用 org.quartz.simpl.JDBCJobStoreTX。

實現持久化的第一步，是要創建 Quartz 持久化所需要的表格。在 Quartz 的發布包 docs/dbTables 中可以找到相應的表格創建腳本。Quartz 支持目前大部分流行的數據庫。本文以 DB2 為例，所需要的腳本為 tables_db2.sql。首先需要對腳本做一點小的修改，即在開頭指明 Schema：

```
 SET CURRENT SCHEMA quartz;
```

為了方便重複使用，創建表格前首先刪除之前的表格：

```
drop table qrtz_job_details;

 drop table qrtz_job_listeners;

…
```

然後創建數據庫 sched，執行 tables_db2.sql 創建持久化所需要的表格。

第二步，配置數據源。數據源與其它所有配置，例如 ThreadPool，均放在 quartz.properties 裡：

清單 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
```

使用時只需要將 quatz.properties 放在 classpath 下面，不用更改一行代碼，再次運行之前的任務調度實例，trigger、job 等信息便會被記錄在數據庫中。

將清單 4 中的 makeWeeklyTrigger 改成 makeSecondlyTrigger，重新運行 main 函數，在 sched 數據庫中查詢表 qrtz_simple_triggers 中的數據。其查詢語句為「db2 『 select repeat_interval, times_triggered from qrtz_simple_triggers 』」。結果 repeat_interval 為 1000，與程序中設置的 makeSecondlyTrigger 相吻合，times_triggered 值為 21。

停掉程序，將數據庫中記錄的任務調度數據重新導入程序運行：

清單 9. 從數據庫中導入任務調度數據重新運行

```
 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 {
 // 初始化一個 Schedule Factory
 SchedulerFactory schedulerFactory = new StdSchedulerFactory();
 // 從 schedule factory 中獲取 scheduler
 Scheduler scheduler = schedulerFactory.getScheduler();
 // 從 schedule factory 中獲取 trigger
 Trigger trigger = scheduler.getTrigger("myTrigger", "myTriggerGroup");
 // 重新開啟調度任務
 scheduler.rescheduleJob("myTrigger", "myTriggerGroup", trigger);
 scheduler.start();
 }
 }
```

清單 9 中，schedulerFactory.getScheduler() 將 quartz.properties 的內容加載到內存，然後根據數據源的屬性初始化數據庫的鏈接，並將數據庫中存儲的數據加載到內存。之後，便可以在內存中查詢某一具體的 trigger，並將其重新啟動。這時候重新查詢 qrtz_simple_triggers 中的數據，發現 times_triggered 值比原來增長了。

回頁首

JCronTab

習慣使用 unix/linux 的開發人員應該對 crontab 都不陌生。Crontab 是一個非常方便的用於 unix/linux 系統的任務調度命令。JCronTab 則是一款完全按照 crontab 語法編寫的 java 任務調度工具。

首先簡單介紹一下 crontab 的語法，與上面介紹的 Quartz 非常相似，但更加簡潔，集中了最常用的語法。主要由六個字段組成（括弧中標識了每個字段的取值範圍）：

```
Minutes  (0-59)
Hours    (0-23)
Day-of-Month  (1-31)
Month     (1-12/JAN-DEC)
Day-of-Week   (0-6/SUN-SAT)
Command
```

與 Quartz 相比，省略了 Seconds 與 Year，多了一個 command 字段，即為將要被調度的命令。JCronTab 中也包含符號「*」與「/」，其含義與 Quartz 相同。

舉例如下：

每天 12 點到 15 點，每隔 1 小時執行一次 Date 命令：

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

每月 2 號凌晨 1 點發一封信給 zhjingbj@cn.ibm.com：

```
0 1 2 * * mail -s 「good」 zhjingbj@cn.ibm.com
```

每週一，週二，週三，週六的晚上 20:00 到 23:00，每半小時打印「normal」：

```
0/30 20-23 * * MON-WED,SAT echo 「normal」
```

JCronTab 借鑑了 crontab 的語法，其區別在於 command 不再是 unix/linux 的命令，而是一個 Java 類。如果該類帶參數，例如「com.ibm.scheduler.JCronTask2#run」，則定期執行 run 方法；如果該類不帶參數，則默認執行 main 方法。此外，還可以傳參數給 main 方法或者構造函數，例如「com.ibm.scheduler.JCronTask2#run Hello World「表示傳兩個參數 Hello 和 World 給構造函數。

JCronTab 與 Quartz 相比，其優點在於，第一，支持多種任務調度的持久化方法，包括普通文件、數據庫以及 XML 文件進行持久化；第二，JCronTab 能夠非常方便地與 Web 應用服務器相結合，任務調度可以隨 Web 應用服務器的啟動自動啟動；第三，JCronTab 還內置了發郵件功能，可以將任務執行結果方便地發送給需要被通知的人。

JCronTab 與 Web 應用服務器的結合非常簡單，只需要在 Web 應用程序的 web.xml 中添加如下行：

清單 10. 在 web.xml 中配置 JCronTab 的屬性

```
<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>
```

在清單 10 中，需要注意兩點：第一，必須指定 servlet-class 為 org.jcrontab.web.loadCrontabServlet，因為它是整個任務調度的入口；第二，必須指定一個參數為 PROPERTIES_FILE，才能被 loadCrontabServlet 識別。

接下來，需要撰寫 D:/Scheduler/src/jcrontab.properties 的內容，其內容根據需求的不同而改變。

當採用普通文件持久化時，jcrontab.properties 的內容主要包括：

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

其中數據來源 org.jcrontab.data.datasource 被描述為普通文件，即 org.jcrontab.data.FileSource。具體的文件即 org.jcrontab.data.file 指明為 D:/Scheduler/src/crontab。

Crontab 描述了任務的調度安排：

```
 */2 * * * * com.ibm.scheduler.JCronTask1
 * * * * * com.ibm.scheduler.JCronTask2#run Hello World
```

其中包含了兩條任務的調度，分別是每兩分鐘執行一次 JCronTask1 的 main 方法，每一分鐘執行一次 JCronTask2 的 run 方法。

清單 11. JcronTask1 與 JCronTask2 的實現

```
 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");
   }
  }
 }
}
```

到此為止，基於普通文件持久化的 JCronTab 的實例就全部配置好了。啟動 Web 應用服務器，便可以看到任務調度的輸出結果：

```
--------------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
```

通過修改 jcrontab.properties 中 datasource，可以選擇採用數據庫或 xml 文件持久化，感興趣的讀者可以參考 進階學習 JCronTab。

此外，JCronTab 還內置了發郵件功能，可以將任務執行結果方便地發送給需要被通知的人。其配置非常簡單，只需要在 jcontab.properties 中添加幾行配置即可：

```
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
```

回頁首

結束語

本文介紹了四種常用的對任務進行調度的 Java 實現方法，即 Timer，ScheduledExecutor, Quartz 以及 JCronTab。文本對每種方法都進行了實例解釋，並對其優缺點進行比較。對於簡單的基於起始時間點與時間間隔的任務調度，使用 Timer 就足夠了；如果需要同時調度多個任務，基於線程池的 ScheduledTimer 是更為合適的選擇；當任務調度的策略複雜到難以憑藉起始時間點與時間間隔來描述時，Quartz 與 JCronTab 則體現出它們的優勢。熟悉 Unix/Linux 的開發人員更傾向於 JCronTab，且 JCronTab 更適合與 Web 應用服務器相結合。Quartz 的 Trigger 與 Job 松耦合設計使其更適用於 Job 與 Trigger 的多對多應用場景。