

作者：洞庭散人

出处：<http://phinecos.cnblogs.com/>

本博客遵从 [Creative Commons Attribution 3.0 License](#)，若用于非商业目的，您可以自由转载，但请保留原作者信息和文章链接 URL。

Quartz 框架快速入门（一）

创建一个 Java 工程，引入几个 JAR 到工程中才能成功构建它们。首先，你需要 Quartz 的二进制版本，包的名字是 `quartz-<version>.jar`。Quartz 还需要几个第三方库；这依赖于你要用到框架的什么功能而定，Commons Digester 库可以在 `<QUARTZ_HOME>/lib/core` 和 `<QUARTZ_HOME>/lib/optional` 目录中找到。如果出现 `java.lang.NoClassDefFoundError: javax/transaction/UserTransaction` 的错误，解决办法是：引入 `jta.jar` 包，这个包在 `quartz-1.6.0/lib/build` 下。

·创建一个 Quartz Job 类

每一个 Quartz Job 必须有一个实现了 `org.quartz.Job` 接口的具体类。这个接口仅有一个要你在 Job 中实现的方法，`execute()`，方法 `execute()` 的原型如下：

```
public void execute(JobExecutionContext context) throws JobExecutionException;
```

当 Quartz 调度器确定到时间要激发一个 Job 的时候，它就会生成一个 Job 实例，并调用这个实例的 `execute()` 方法。调度器只管调用 `execute()` 方法，而不关心执行的结果，除了在作业执行中出问题抛出的 `org.quartz.JobExecutionException` 异常。

下面是我们的第一个 Quartz job，它被设计来扫描一个目录中的文并显示文件的详细信息。



```
package com.vista.quartz;

import java.io.File;
import java.io.FileFilter;
import java.util.Date;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.quartz.Job;
import org.quartz.JobDataMap;
import org.quartz.JobDetail;
import org.quartz.JobExecutionContext;
import org.quartz.JobExecutionException;

public class ScanDirectoryJob implements Job
{
    static Log logger = LogFactory.getLog(ScanDirectoryJob.class); //日志记录器

    public void execute(JobExecutionContext context) throws JobExecutionException
    {
        //Every job has its own job detail
        JobDetail jobDetail = context.getJobDetail();
        // The name is defined in the job definition
        String jobName = jobDetail.getName(); //任务名称
    }
}
```

```

// Log the time the job started
logger.info(jobName + " fired at " + new Date()); //记录任务开始执行的时间
// The directory to scan is stored in the job map
JobDataMap dataMap = jobDetail.getJobDataMap(); //任务所配置的数据映射表
String dirName = dataMap.getString("SCAN_DIR"); //获取要扫描的目录
// Validate the required input
if (dirName == null)
{
    //所需要的扫描目录没有提供
    throw new JobExecutionException( "Directory not configured" );
}
// Make sure the directory exists
File dir = new File(dirName);
if (!dir.exists())
{
    //提供的是错误目录
    throw new JobExecutionException( "Invalid Dir "+ dirName);
}
// Use FileFilter to get only XML files
FileFilter filter = new FileExtensionFileFilter(".xml");
//只统计 xml 文件
File[] files = dir.listFiles(filter);
if (files == null || files.length <= 0)
{
    //目录下没有 xml 文件
    logger.info("No XML files found in " + dir);
    // Return since there were no files
    return;
}
// The number of XML files
int size = files.length;
// Iterate through the files found
for (int i = 0; i < size; i++)
{
    File file = files[i];
    // Log something interesting about each file.
    File aFile = file.getAbsoluteFile();
    long fileSize = file.length();
    String msg = aFile + " - Size: " + fileSize;
    logger.info(msg); //记录下文件的路径和大小
}
}
}

```



当 Quartz 调用 `execute()` 方法，会传递一个 `org.quartz.JobExecutionContext` 上下文变量，里面封装有 Quartz 的运行时环境和当前正执行的 Job。通过 `JobExecutionContext`，你可以访问到调度器的信息，作业和作业上的触发器的信息，还有更多更多的信息。在代码中，`JobExecutionContext` 被用来访问 `org.quartz.JobDetail` 类，`JobDetail` 类持有 Job 的详细信息，包括为 Job 实例指定的名称，Job 所属组，Job 是否被持久化(易失性)，和许多其他感兴趣的属性。

`JobDetail` 又持有一个指向 `org.quartz.JobDataMap` 的引用。`JobDataMap` 中有为指定 `Job` 配置的自定义属性。例如，在代码中我们从 `JobDataMap` 中获得欲扫描的目录名，我们可以在 `ScanDirectoryJob` 中硬编码这个目录名，但是这样的话我们难以重用这个 `Job` 来扫描别的目录了。在后面你将会看到目录是如何配置到 `JobDataMap` 的。

`execute()` 方法中剩下的就是标准 Java 代码了：获得目录名并创建一个 `java.io.File` 对象。它还对目录名作为简单的校验，确保是一个有效且存在的目录。接着调用 `File` 对象的 `listFiles()` 方法得到目录下的文件。还创建了一个 `java.io.FileFilter` 对象作为参数传递给 `listFiles()` 方法。`org.quartzbook.cavaness.FileExtensionFileFilter` 实现了 `java.io.FileFilter` 接口，它的作用是过滤掉目录仅返回 XML 文件。默认情况下，`listFiles()` 方法是返回目录中所有内容，不管是文件还是子目录，所以我们必须过滤一下，因为我们只对 XML 文件感兴趣。

`FileExtensionFileFilter` 被用来屏蔽名称中不含字符串 `".xml"` 的文件。它还屏蔽了子目录——这些子目录原本会让 `listFiles()` 方法正常返回。过滤器提供了一种很便利的方式选择性的向你的 Quartz 作业提供它能接受的作为输入的文件



```
package com.vista.quartz;

import java.io.File;
import java.io.FileFilter;

public class FileExtensionFileFilter implements FileFilter
{
    private String extension;//文件后缀

    public FileExtensionFileFilter(String extension)
    {
        this.extension = extension;
    }

    public boolean accept(File file)
    { //只接受指定后缀的文件
        // Lowercase the filename for easier comparison
        String lCaseFilename = file.getName().toLowerCase();//小写化
        return (file.isFile() && (lCaseFilename.indexOf(extension) > 0 )) ? true : false ;
    }
}
```



到目前为止，我们已经创建了一个 Quartz job，但还没有决定怎么处置它——明显地，我们需以某种方式对这个 Job 设置一个运行时间表。时间表可以是一次性的事件，或者我们可能会安装它在除周日之外的每个午夜执行。你即刻将会看到，Quartz Scheduler 是框架的心脏与灵魂。所有的 Job 都通过 Scheduler 注册；必要时，Scheduler 也会创建 Job 类的实例，并执行实例的 `execute()` 方法。

•编程式安排一个 Quartz Job

所有的要 Quartz 来执行的作业必须通过调度器来注册。大多情况下，这会在调度器启动前做好。正如前面说过，这一操作也提供了声明式与编程式两种实现途径的选择。

因为每一个 Job 都必须用 Scheduler 来注册，所以先定义一个 `JobDetail`，并关联到这个 Scheduler 实例。

下面的程序提供了一个理解如何编程式安排一个 Job 很好的例子。代码首先调用 `createScheduler()` 方法从 Scheduler 工厂获取一个 Scheduler 的实例。得到 Scheduler 实例之后，把它传递给 `schedulerJob()` 方法，由它把 Job 同 Scheduler 进行关联。

首先，创建了我们想要运行的 Job 的 `JobDetail` 对象。`JobDetail` 构造器的参数中包含指派给 Job 的名称，逻辑组名，和实现 `org.quartz.Job` 接口的全限定类名称。我们可以使用 `JobDetail` 的别的构造器。

在前面有说过，`JobDetail` 扮演着某一 Job 定义的角色。它带有 Job 实例的属性，能在运行时被所关联的 Job 访问到。其中在使用 `JobDetail` 时，的一个最重要的东西就是 `JobDataMap`，它被用来存放 Job 实例的状态和参数。在代码中，待扫描的目录名称就是通过 `scheduleJob()` 方法存入到 `JobDataMap` 中的。

Job 只是一个部分而已。注意我们没有在 `JobDetail` 对象中为 Job 设定执行日期和次数。这是 Quartz Trigger 该做的事。顾名思义，Trigger 的责任就是触发一个 Job 去执行。当用 Scheduler 注册一个 Job 的时候要创建一个 Trigger 与这个 Job 相关联。Quartz 提供了四种类型的 Trigger，但其中两种是最为常用的，它们就是在下面要用到的 `SimpleTrigger` 和 `CronTrigger`。

`SimpleTrigger` 是两个之中简单的那个，它主要用来激发单事件的 Job，Trigger 在指定时间激发，并重复 n 次--两次激发时间之间的延时为 m，然后结束作业。`CronTrigger` 非常复杂且强大。它是基于通用的公历，当需要用一种较复杂的时间表去执行一个 Job 时用到。例如，四月至九月的每个星期一、星期三、或星期五的午夜。

为更简单的使用 Trigger，Quartz 包含了一个工具类，叫做 `org.quartz.TriggerUtils`。`TriggerUtils` 提供了许多便捷的方法简化了构造和配置 trigger。本文的例子中有用的就是 `TriggerUtils` 类；`SimpleTrigger` 和 `CronTrigger` 会在后面用到。

正如你看到的那样，调用了 `TriggerUtils` 的方法 `makeSecondlyTrigger()` 来创建一个每 10 秒种激发一次的 trigger(实际是由 `TriggerUtils` 生成了一个 `SimpleTrigger` 实例，但是我们的代码并不想知道这些)。我们同样要给这个 trigger 实例一个名称并告诉它何时激发相应的 Job；与之关联的 Job 会立即启动，因为由方法 `setStartTime()` 设定的是当前时间



```
package com.vista.quartz;

import java.util.Date;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.quartz.JobDetail;
import org.quartz.Scheduler;
import org.quartz.SchedulerException;
import org.quartz.Trigger;
import org.quartz.TriggerUtils;
import org.quartz.impl.StdSchedulerFactory;

public class SimpleScheduler
{
    static Log logger = LogFactory.getLog(SimpleScheduler.class);
    public static void main(String[] args)
    {
        SimpleScheduler simple = new SimpleScheduler();
        try
        {
            // Create a Scheduler and schedule the Job
            Scheduler scheduler = simple.createScheduler();
            simple.scheduleJob(scheduler);

            // Start the Scheduler running
        }
    }
}
```

```

        scheduler.start();

        logger.info( "Scheduler started at " + new Date());

    } catch (SchedulerException ex) {
        logger.error(ex);
    }

}

public Scheduler createScheduler() throws SchedulerException
{//创建调度器
    return StdSchedulerFactory.getDefaultScheduler();
}

//Create and Schedule a ScanDirectoryJob with the Scheduler
private void scheduleJob(Scheduler scheduler) throws SchedulerException
{
    // Create a JobDetail for the Job
    JobDetail jobDetail = new JobDetail("ScanDirectory", Scheduler.DEFAULT_GROUP, ScanDirectoryJob.class);

    // Configure the directory to scan
    jobDetail.getJobDataMap().put("SCAN_DIR", "D:\\Tomcat\\conf"); //set the JobDataMap that
    is associated with the Job.

    // Create a trigger that fires every 10 seconds, forever
    Trigger trigger = TriggerUtils.makeSecondlyTrigger(10); //每 10 秒触发一次
    trigger.setName("scanTrigger");

    // Start the trigger firing from now
    trigger.setStartTime(new Date()); //设置第一次触发时间

    // Associate the trigger with the job in the scheduler
    scheduler.scheduleJob(jobDetail, trigger);
}
}

```



假如你有不只一个个 **Job** (你也许就是), 你将需要为每一个 **Job** 创建各自的 **JobDetail**。每一个 **JobDetail** 必须通过 **scheduleJob()** 方法一一注册到 **Scheduler** 上。而如果你想重用了一个 **Job** 类, 让它产生多个实例运行, 那么你需要为每个实例都创建一个 **JobDetail**。例如, 假如你想重用 **ScanDirectoryJob** 让它检查两个不同的目录, 你需要创建并注册两个 **JobDetail** 实例



```

package com.vista.quartz;

import java.util.Date;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.quartz.JobDetail;
import org.quartz.Scheduler;
import org.quartz.SchedulerException;

```

```

import org.quartz.Trigger;
import org.quartz.TriggerUtils;
import org.quartz.impl.StdSchedulerFactory;

public class SimpleScheduler
{
    static Log logger = LogFactory.getLog(SimpleScheduler.class);
    public static void main(String[] args)
    {
        SimpleScheduler simple = new SimpleScheduler();
        try
        {
            // Create a Scheduler and schedule the Job
            Scheduler scheduler = simple.createScheduler();
            // Jobs can be scheduled after Scheduler is running
            scheduler.start();
            logger.info("Scheduler started at " + new Date());
            // Schedule the first Job
            simple.scheduleJob(scheduler, "ScanDirectory1", ScanDirectoryJob.class, "D:\\conf1",
10);

            // Schedule the second Job
            simple.scheduleJob(scheduler, "ScanDirectory2", ScanDirectoryJob.class, "D:\\conf2 ",
15);
        }
        catch (SchedulerException ex)
        {
            logger.error(ex);
        }
    }

    public Scheduler createScheduler() throws SchedulerException
    { //创建调度器
        return StdSchedulerFactory.getDefaultScheduler();
    }

    private void scheduleJob(Scheduler scheduler, String jobName, Class jobClass, String scanDir,
int scanInterval) throws SchedulerException
    {
        // Create a JobDetail for the Job
        JobDetail jobDetail = new JobDetail(jobName, Scheduler.DEFAULT_GROUP, jobClass);
        // Configure the directory to scan
        jobDetail.getJobDataMap().put("SCAN_DIR", scanDir);
        // Trigger that repeats every "scanInterval" secs forever
        Trigger trigger = TriggerUtils.makeSecondlyTrigger(scanInterval);
        trigger.setName(jobName + "-Trigger");
        // Start the trigger firing from now
        trigger.setStartTime(new Date());
        // Associate the trigger with the job in the scheduler
        scheduler.scheduleJob(jobDetail, trigger);
    }
}

```



[Quartz 框架快速入门（二）](#)

尽可能的用声明式处理软件配置，其次才考虑编程式的方式。在上一篇《[Quartz 框架快速入门（一）](#)》中，如果我们要在 **Job** 启动之后改变它的执行时间和频度，必须去修改源代码重新编译。这种方式只适用于小的例子程序，但是对于一个大且复杂的系统，这就成了一个问题了。因此，假如能以声明式部署 **Quartz Job** 时，并且也是需求允许的情况下，你应该每次都选择这种方式

•配置 quartz.properties 文件

文件 **quartz.properties** 定义了 **Quartz** 应用运行时行为，还包含了许多能控制 **Quartz** 运转的属性。这个文件应该放在 **classpath** 所指的路径下，比如我们这个 **java** 工程，就将它和下面将介绍的 **jobs.xml** 一起放在项目根目录下就是。如果不清楚就查看 **.classpath** 文件，它里面就配置了你的项目的 **classpath**。

我们来看看最基础的 **quartz.properties** 文件，并讨论其中一些设置。下面是一个修剪版的 **quartz.properties** 文件



```
#=====
# Configure Main Scheduler Properties
#=====
org.quartz.scheduler.instanceName = TestScheduler
org.quartz.scheduler.instanceId = AUTO
#=====
# Configure ThreadPool
#=====
org.quartz.threadPool.class = org.quartz.simpl.SimpleThreadPool
org.quartz.threadPool.threadCount = 3
org.quartz.threadPool.threadPriority = 5
#=====
# Configure JobStore
#=====
org.quartz.jobStore.misfireThreshold = 60000
org.quartz.jobStore.class = org.quartz.simpl.RAMJobStore
#=====
# Configure Plugins
#=====
org.quartz.plugin.triggHistory.class = org.quartz.plugins.history.LoggingJobHistoryPlugin
org.quartz.plugin.jobInitializer.class = org.quartz.plugins.xml.JobInitializationPlugin
org.quartz.plugin.jobInitializer.fileNames = jobs.xml
org.quartz.plugin.jobInitializer.overWriteExistingJobs = true
org.quartz.plugin.jobInitializer.failOnFileNotFound = true
org.quartz.plugin.jobInitializer.scanInterval = 10
org.quartz.plugin.jobInitializer.wrapInUserTransaction = false
```



•调度器属性

第一部分有两行，分别设置调度器的实例名(instanceName) 和实例 ID (instanceId)。属性 `org.quartz.scheduler.instanceName` 可以是你喜欢的任何字符串。它用来在用到多个调度器区分特定的调度器实例。多个调度器通常用在集群环境中。(Quartz 集群将会在第十一章，“Quartz 集群”中讨论)。现在的话，设置如下的一个字符串就行：`org.quartz.scheduler.instanceName = QuartzScheduler` 实际上，这也是当你没有该属性配置时的默认值。

调度器的第二个属性是 `org.quartz.scheduler.instanceId`。和 `instanceName` 属性一样，`instanceId` 属性也允许任何字符串。这个值必须是在所有调度器实例中是唯一的，尤其是在一个集群当中。假如你想 Quartz 帮你生成这个值的话，可以设置为 `AUTO`。如果 Quartz 框架是运行在非集群环境中，那么自动产生的值将会是 `NON_CLUSTERED`。假如是在集群环境下使用 Quartz，这个值将会是主机名加上当前的日期和时间。大多情况下，设置为 `AUTO` 即可。

•线程池属性

接下来的部分是设置有关线程必要的属性值，这些线程在 Quartz 中是运行在后台担当重任的。`threadCount` 属性控制了多少个工作者线程被创建用来处理 Job。原则上是，要处理的 Job 越多，那么需要的工作者线程也就越多。`threadCount` 的数值至少为 1。Quartz 没有限定你设置工作者线程的最大值，但是在多数机器上设置该值超过 100 的话就会显得相当不实用了，特别是在你的 Job 执行时间较长的情况下。这项没有默认值，所以你必须为这个属性设定一个值。

`threadPriority` 属性设置工作者线程的优先级。优先级别高的线程比级别低的线程更优先得到执行。`threadPriority` 属性的最大值是常量 `java.lang.Thread.MAX_PRIORITY`，等于 10。最小值为常量 `java.lang.Thread.MIN_PRIORITY`，为 1。这个属性的正常值是 `Thread.NORM_PRIORITY`，为 5。大多情况下，把它设置为 5，这也是没指定该属性的默认值。

最后一个要设置的线程池属性是 `org.quartz.threadPool.class`。这个值是一个实现了 `org.quartz.spi.ThreadPool` 接口的类的全限名称。Quartz 自带的线程池实现类是 `org.quartz.simpl.SimpleThreadPool`，它能够满足大多数用户的需求。这个线程池实现具备简单的行为，并经过很好的测试过。它在调度器的生命周期中提供固定大小的线程池。你能根据需求创建自己的线程池实现，如果你想要一个随需可伸缩的线程池时也许需要这么做。这个属性没有默认值，你必须为其指定值。

•作业存储设置

作业存储部分的设置描述了在调度器实例的生命周期中，Job 和 Trigger 信息是如何被存储的。我们还没有讨论到作业存储和它的目的；因为对当前例子是非必需的，所以我们留待以后说明。现在的话，你所要了解的就是我们存储调度器信息在内存中而不是在关系型数据库中就行了。

把调度器信息存储在内存中非常的快也易于配置。当调度器进程一旦被终止，所有的 Job 和 Trigger 的状态就丢失了。要使 Job 存储在内存中需通过设置 `org.quartz.jobStore.class` 属性为 `org.quartz.simpl.RAMJobStore`。假如我们不希望在 JVM 退出之后丢失调度器的状态信息的话，我们可以使用关系型数据库来存储这些信息。这需要另一个作业存储(JobStore) 实现，我们在后面将会讨论到。第五章“Cron Trigger 和其他”和第六章“作业存储和持久化”会提到你需要用到的不同类型的作业存储实现。

•插件配置

在这个简单的 `quartz.properties` 文件中最后一部分是你要用到的 Quartz 插件的配置。插件常常在别的开源框架上使用到，比如 Apache 的 Struts 框架(见 <http://struts.apache.org/>)。


一个声明式扩框架的方法就是通过新加实现了 `org.quartz.spi.SchedulerPlugin` 接口的类。`SchedulerPlugin` 接口中有给调度器调用的三个方法。

要在我们的例子中声明式配置调度器信息，我们会用到一个 Quartz 自带的叫做 `org.quartz.plugins.xml.JobInitializationPlugin` 的插件。

默认时，这个插件会在 classpath 中搜索名为 `quartz_jobs.xml` 的文件并从中加载 Job 和 Trigger 信息。在下面中讨论 `quartz_jobs.xml` 文件，这是我们所参考的非正式的 Job 定义文件。

•为插件修改 quartz.properties 配置

`JobInitializationPlugin` 找寻 `quartz_jobs.xml` 来获得声明的 `Job` 信息。假如你想改变这个文件名，你需要修改 `quartz.properties` 来告诉插件去加载那个文件。例如，假如你想要 `Quartz` 从名为 `my_quartz_jobs.xml` 的 XML 文件中加载 `Job` 信息，你不得不为插件指定这一文件



```
org.quartz.plugin.triggHistory.class = org.quartz.plugins.history.LoggingJobHistoryPlugin
org.quartz.plugin.jobInitializer.class = org.quartz.plugins.xml.JobInitializationPlugin
org.quartz.plugin.jobInitializer.fileNames = jobs.xml
org.quartz.plugin.jobInitializer.overWriteExistingJobs = true
org.quartz.plugin.jobInitializer.failOnFileNotFound = true
org.quartz.plugin.jobInitializer.scanInterval = 10
org.quartz.plugin.jobInitializer.wrapInUserTransaction = false
```



我们添加了属性 `org.quartz.plugin.jobInitializer.fileName` 并设置该属性值为我们想要的文件名。这个文件名要对 `classloader` 可见，也就是说要在 `classpath` 下。

当 `Quartz` 启动后读取 `quartz.properties` 文件，然后初始化插件。它会传递上面配置的所有属性给插件，这时候插件也就得到通知去搜寻不同的文件。

下面就是目录扫描例子的 `Job` 定义的 XML 文件。正如上一篇所示例子那样，这里我们用的是声明式途径来配置 `Job` 和 `Trigger` 信息的



```
<?xml version='1.0' encoding='utf-8'?>
<quartz xmlns="http://www.opensymphony.com/quartz/JobSchedulingData"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.opensymphony.com/quartz/JobSchedulingData
http://www.opensymphony.com/quartz/xml/job_scheduling_data_1_5.xsd"
  version="1.5">
  <job>
    <job-detail>
      <name>ScanDirectory</name>
      <group>DEFAULT</group>
      <description>
        A job that scans a directory for files
      </description>
      <job-class>
        com.vista.quartz.ScanDirectoryJob
      </job-class>
      <volatility>false</volatility>
      <durability>false</durability>
      <recover>false</recover>
      <job-data-map allows-transient-data="true">
        <entry>
          <key>SCAN_DIR</key>
          <value>D:\conf1</value>
        </entry>
      </job-data-map>
    </job-detail>

    <trigger>
```

```

    <simple>
      <name>scanTrigger</name>
      <group>DEFAULT</group>
      <job-name>ScanDirectory</job-name>
      <job-group>DEFAULT</job-group>
      <start-time>2008-09-03T14:43:00</start-time>
      <!-- repeat indefinitely every 10 seconds -->
      <repeat-count>-1</repeat-count>
      <repeat-interval>10000</repeat-interval>
    </simple>
  </trigger>
</job>
</quartz>

```



在 jobs.xml 中 <start-time> 的格式是:

```
<start-time>2008-09-03T14:43:00</start-time>
```

其中 T 隔开日期和时间, 默认时区

或者:

```
<start-time>2008-09-03T14:43:00+08:00</start-time>
```

其中+08:00 表示东八区

<job> 元素描述了一个要注册到调度器上的 Job, 相当于我们在前面章节中使用 `scheduleJob()` 方法那样。你所看到的<job-detail>和 <trigger> 这两个元素就是我们在代码中以程式化传递给方法 `schedulerJob()` 的参数。前面本质上是与这里一样的, 只是现在用的是一种较流行声明的方式。<trigger>元素也是非常直观的: 它使用前面同样的属性, 但更简单的建立一个 `SimpleTrigger`。因此仅仅是一种不同的(可论证的且更好的)方式做了上一篇代码 中同样的事情。显然, 你也可以支持多个 Job。在上一篇代码 中我们编程的方式那么做的, 也能用声明的方式来支持



```

<?xml version='1.0' encoding='utf-8'?>
<quartz xmlns="http://www.opensymphony.com/quartz/JobSchedulingData"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.opensymphony.com/quartz/JobSchedulingData
    http://www.opensymphony.com/quartz/xml/job_scheduling_data_1_5.xsd"
  version="1.5">
  <job>
    <job-detail>
      <name>ScanDirectory1</name>
      <group>DEFAULT</group>
      <description>
        A job that scans a directory for files
      </description>
      <job-class>
        com.vista.quartz.ScanDirectoryJob
      </job-class>
      <volatility>false</volatility>
      <durability>false</durability>
      <recover>false</recover>
    </job-detail>
  </job>
</quartz>

```

```
<job-data-map allows-transient-data="true">
  <entry>
    <key>SCAN_DIR</key>
    <value>D:\dyk\Java\Tomcat\conf</value>
  </entry>
</job-data-map>
</job-detail>
<trigger>
  <simple>
    <name>scanTrigger1</name>
    <group>DEFAULT</group>
    <job-name>ScanDirectory1</job-name>
    <job-group>DEFAULT</job-group>
    <start-time>2008-09-03T15:00:10</start-time>
    <!-- repeat indefinitely every 10 seconds -->
    <repeat-count>-1</repeat-count>
    <repeat-interval>10000</repeat-interval>
  </simple>
</trigger>
</job>
<job>
  <job-detail>
    <name>ScanDirectory2</name>
    <group>DEFAULT</group>
    <description>
      A job that scans a directory for files
    </description>
    <job-class>
      com.vista.quartz.ScanDirectoryJob
    </job-class>
    <volatility>false</volatility>
    <durability>false</durability>
    <recover>false</recover>
    <job-data-map allows-transient-data="true">
      <entry>
        <key>SCAN_DIR</key>
        <value>D:\dyk\Java\Tomcat\webapps\MyTest\WEB-INF</value>
      </entry>
    </job-data-map>
  </job-detail>
  <trigger>
    <simple>
      <name>scanTrigger2</name>
      <group>DEFAULT</group>
      <job-name>ScanDirectory2</job-name>
      <job-group>DEFAULT</job-group>
      <start-time>2008-09-03T15:00:20</start-time>
      <!-- repeat indefinitely every 15 seconds -->
```

```
<repeat-count>-1</repeat-count>
<repeat-interval>15000</repeat-interval>
</simple>
</trigger>
</job>
</quartz>
```



最后我们来看看原来的代码简化成如何了：



```
package com.vista.quartz;

import java.util.Date;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.quartz.JobDetail;
import org.quartz.Scheduler;
import org.quartz.SchedulerException;
import org.quartz.Trigger;
import org.quartz.TriggerUtils;
import org.quartz.impl.StdSchedulerFactory;

public class SimpleScheduler
{
    static Log logger = LogFactory.getLog(SimpleScheduler.class);
    public static void main(String[] args)
    {
        SimpleScheduler simple = new SimpleScheduler();
        try
        {
            // Create a Scheduler and schedule the Job
            Scheduler scheduler = simple.createScheduler();
            // Jobs can be scheduled after Scheduler is running
            scheduler.start();
            logger.info("Scheduler started at " + new Date());
        }
        catch (SchedulerException ex)
        {
            logger.error(ex);
        }
    }
    public Scheduler createScheduler() throws SchedulerException
    { //创建调度器
        return StdSchedulerFactory.getDefaultScheduler();
    }
}
```

Quartz 框架快速入门（三）

在前面两篇文章中简单介绍了在 java 应用程序中如何使用 Quartz 框架，这一篇中我们将看到如何在 web 环境下通过配置文件来完成 Quartz 的后台作业调度，而不必手工去创建 Trigger 和 Scheduler，其步骤如下：

首先创建一个 Web 项目，将 quartz-1.6.0.jar,以及 lib 目录下面 core 下所有 jar，optional 目录下的所有 commons-beanutils.jar 和 commons-digester-1.7.jar，build 目录下的 jta.jar 都放入 Web 项目的 WEB-INF\lib 目录下。

创建一个简单的 job 类 HelloWorld，它的功能很简单，就是输出当前的时间，代码如下：

```
package com.vista.quartz;

import java.util.Date;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.quartz.Job;
import org.quartz.JobExecutionContext;
import org.quartz.JobExecutionException;

public class Helloworld implements Job
{
    private static Log logger = LogFactory.getLog(Helloworld.class); //日志记录器
    public Helloworld()
    {
    }
    public void execute(JobExecutionContext context) throws JobExecutionException
    {
        logger.info("Hello World! - " + new Date());
    }
}
```

然后按照上一篇文章《**Quartz 框架快速入门（二）**》中所讲述的内容编写 quartz.properties 文件。如果启动项目的时候，Quartz 没有在工程中找到该文件，就会从自己的 jar 包下面读取其默认的 properties 文件，其内容如下

```
=====
# Configure Main Scheduler Properties
=====
org.quartz.scheduler.instanceName = QuartzScheduler
org.quartz.scheduler.instanceId = AUTO
=====
# Configure ThreadPool
=====
org.quartz.threadPool.class = org.quartz.simpl.SimpleThreadPool
org.quartz.threadPool.threadCount = 5
org.quartz.threadPool.threadPriority = 5
```

```

=====
# Configure JobStore
=====
org.quartz.jobStore.misfireThreshold = 60000
org.quartz.jobStore.class = org.quartz.simpl.RAMJobStore
=====
# Configure Plugins
=====
org.quartz.plugin.triggHistory.class = org.quartz.plugins.history.LoggingJobHistoryPlugin
org.quartz.plugin.jobInitializer.class = org.quartz.plugins.xml.JobInitializationPlugin
org.quartz.plugin.jobInitializer.fileNames = jobs.xml
org.quartz.plugin.jobInitializer.overWriteExistingJobs = true
org.quartz.plugin.jobInitializer.failOnFileNotFound = true
org.quartz.plugin.jobInitializer.scanInterval = 10
org.quartz.plugin.jobInitializer.wrapInUserTransaction = false

```

然后编写任务配置文件 **jobs.xml**，内容如下：

```

<?xml version='1.0' encoding='utf-8'?>
<quartz xmlns="http://www.opensymphony.com/quartz/JobSchedulingData"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.opensymphony.com/quartz/JobSchedulingData
    http://www.opensymphony.com/quartz/xml/job_scheduling_data_1_5.xsd"
  version="1.5">
  <job>
    <job-detail>
      <name>HelloWorld</name>
      <group>DEFAULT</group>
      <description>
        A job that just for test
      </description>
      <job-class>
        com.vista.quartz.Helloworld
      </job-class>
      <volatility>false</volatility>
      <durability>false</durability>
      <recover>false</recover>
    </job-detail>
    <trigger>
      <simple>
        <name>HelloTrigger1</name>
        <group>DEFAULT</group>
        <job-name>HelloWorld</job-name>
        <job-group>DEFAULT</job-group>
        <start-time>2008-09-03T15:56:30</start-time>
        <!-- repeat indefinitely every 10 seconds -->
        <repeat-count>-1</repeat-count>
        <repeat-interval>10000</repeat-interval>
      </simple>
    </trigger>
  </job>
</quartz>

```



```
</job>
</quartz>
```

可以看到，在配置文件中把 **jobdetail** 和 **trigger** 都作了完整的定义，并组合成一个 **job**。下面，我们把上面两个文件都放入 **/WEB-INF/classes** 目录下，然后按照 **api** 中的说明修改一下 **web.xml**，内容如下

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4"
  xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
  <servlet>
    <servlet-name>QuartzInitializer</servlet-name>
    <display-name>Quartz Initializer Servlet</display-name>
    <servlet-class>
      org.quartz.ee.servlet.QuartzInitializerServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
    <init-param>
      <param-name>config-file</param-name>
      <param-value>/quartz.properties</param-value>
    </init-param>
    <init-param>
      <param-name>shutdown-on-unload</param-name>
      <param-value>true</param-value>
    </init-param>
  </servlet>
  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>
</web-app>
```

这样，在启动 **Tomcat** 的时候，**QuartzInitializerServlet** 这个 **Servlet** 就会自动读取 **quartz.properties** 这个配置文件，并初始化调度信息，启动 **Scheduler**。

启动 **tomcat** 后，就可以看到输出的结果：



```
2008-9-3 15:58:50 com.vista.quartz.Helloworld execute
信息: Hello World! - Wed Sep 03 15:58:50 CST 2008
2008-9-3 15:58:50 org.quartz.plugins.history.LoggingJobHistoryPlugin jobWasExecu
ted
信息: Job DEFAULT.HelloWorld execution complete at 15:58:50 09/03/2008 and repo
rts: null
```

[Quartz 框架快速入门（四）](#)

Spring 的 `scheduling.quartz` 包中对 Quartz 框架进行了封装，使得开发时不用写任何 QuartzSpring 的代码就可以实现定时任务。Spring 通过 `JobDetailBean`，`MethodInvokingJobDetailFactoryBean` 实现 Job 的定义。后者更加实用，只需指定要运行的类，和该类中要运行的方法即可，Spring 将自动生成符合 Quartz 要求的 `JobDetail`。

在上一篇文章《[Quartz 框架快速入门（三）](#)》中我们将示例迁移到 Web 环境下了，但使用的是 Quartz 的启动机制，这一篇中我们将让 Web 服务器启动 Spring，通过 Spring 的配置文件来进行任务的调度

1, 创建一个 Web 项目，加入 `spring.jar`，`quartz-1.6.0.jar`，`commons-collections.jar`，`jta.jar`，`commons-logging.jar` 这几个包。

2, 创建一个类，在类中添加一个方法 `execute`，我们将对这个方法进行定时调度。

```
package com.vista.quartz;

import java.util.Date;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.quartz.JobExecutionContext;
import org.quartz.JobExecutionException;

public class HelloWorld
{
    private static Log logger = LogFactory.getLog(HelloWorld.class); //日志记录器
    public HelloWorld()
    {
    }
    public void execute()
    {
        logger.info("Kick your ass and Fuck your mother! - " + new Date());
    }
}
```

2. Spring 配置文件 `applicationContext.xml` 修改如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

    <!-- 要调用的工作类 -->
    <bean id="quartzJob" class="com.vista.quartz.HelloWorld"></bean>
```

```

<!-- 定义调用对象和调用对象的方法 -->
<bean id="jobtask" class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">
    <!-- 调用的类 -->
    <property name="targetObject">
        <ref bean="quartzJob"/>
    </property>
    <!-- 调用类中的方法 -->
    <property name="targetMethod">
        <value>execute</value>
    </property>
</bean>
<!-- 定义触发时间 -->
<bean id="doTime" class="org.springframework.scheduling.quartz.CronTriggerBean">
    <property name="jobDetail">
        <ref bean="jobtask"/>
    </property>
    <!-- cron 表达式 -->
    <property name="cronExpression">
        <value>10,15,20,25,30,35,40,45,50,55 * * * * ?</value>
    </property>
</bean>
<!-- 总管理类 如果将 lazy-init='false' 那么容器启动就会执行调度程序 -->
<bean id="startQuartz" lazy-init="false" autowire="no" class="org.springframework.scheduling.quartz.SchedulerFactoryBean">
    <property name="triggers">
        <list>
            <ref bean="doTime"/>
        </list>
    </property>
</bean>
</beans>

```

3,先在控制台中对上面的代码进行测试，我们要做的只是加载 Spring 的配置文件就可以了，代码如下：

```

package com.vista.quartz;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Test
{
    public static void main(String[] args)
    {
        System.out.println("Test start.");
        ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");

        //如果配置文件中将 startQuartz bean 的 lazy-init 设置为 false 则不用实例化
        //context.getBean("startQuartz");
        System.out.print("Test end..");
    }
}

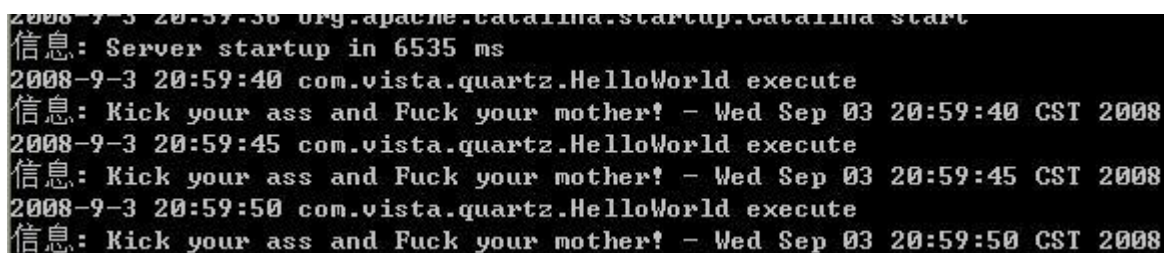
```

```
}  
}
```

4,然后将 Web.xml 修改如下, 让 tomcat 在启动时去初始化 Spring:

```
<?xml version="1.0" encoding="UTF-8"?>  
<web-app version="2.4"  
    xmlns="http://java.sun.com/xml/ns/j2ee"  
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee  
        http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">  
    <context-param>  
        <param-name>contextConfigLocation</param-name>  
        <param-value>  
            /WEB-INF/classes/applicationContext.xml  
        </param-value>  
    </context-param>  
  
    <servlet>  
        <servlet-name>SpringContextServlet</servlet-name>  
        <servlet-class>org.springframework.web.context.ContextLoaderServlet</servlet-class>  
        <load-on-startup>1</load-on-startup>  
    </servlet>  
  
    <welcome-file-list>  
        <welcome-file>index.jsp</welcome-file>  
    </welcome-file-list>  
</web-app>
```

5,最后启动 Tomcat,测试结果如下图所示:



```
2008-9-3 20:59:36 org.apache.catalina.startup.Catalina start  
信息: Server startup in 6535 ms  
2008-9-3 20:59:40 com.vista.quartz.HelloWorld execute  
信息: Kick your ass and Fuck your mother! - Wed Sep 03 20:59:40 CST 2008  
2008-9-3 20:59:45 com.vista.quartz.HelloWorld execute  
信息: Kick your ass and Fuck your mother! - Wed Sep 03 20:59:45 CST 2008  
2008-9-3 20:59:50 com.vista.quartz.HelloWorld execute  
信息: Kick your ass and Fuck your mother! - Wed Sep 03 20:59:50 CST 2008
```

作者: 洞庭散人

出处: <http://phinecos.cnblogs.com/>

本博客遵从 [Creative Commons Attribution 3.0 License](#), 若用于非商业目的, 您可以自由转载, 但请保留原作者信息和文章链接 URL。