

基于 Quartz 开发企业级任务调度应用

Quartz 是 OpenSymphony 开源组织在任务调度领域的一个开源项目，完全基于 Java 实现。作为一个优秀的开源调度框架，Quartz 具有功能强大，应用灵活，易于集成的特点。本文剖析了 Quartz 框架内部的基本实现原理，通过一些具体实例描述了应用 Quartz 开发应用程序的基本方法，并对企业应用中常见的问题及解决方案进行了讨论。

0



评论

张 晓宁, 软件工程师, IBM
2013 年 5 月 15 日

expand

- 内容



在 IBM Bluemix 云平台上开发并部署您的下一个应用。

Quartz 基本概念及原理

Quartz Scheduler 开源框架

Quartz 是 OpenSymphony 开源组织在任务调度领域的一个开源项目，完全基于 Java 实现。该项目于 2009 年被 Terracotta 收购，目前是 Terracotta 旗下的一个项目。读者可以到 <http://www.quartz-scheduler.org/> 站点下载 Quartz 的发布版本及其源代码。笔者在产品开发中使用的是版本 1.8.4，因此本文内容基于该版本。本文不仅介绍如何应用 Quartz 进行开发，也对其内部实现原理作一定讲解。

作为一个优秀的开源调度框架，Quartz 具有以下特点：

1. 强大的调度功能，例如支持丰富多样的调度方法，可以满足各种常规及特殊需求；
2. 灵活的应用方式，例如支持任务和调度的多种组合方式，支持调度数据的多种存储方式；
3. 分布式和集群能力，Terracotta 收购后在原来功能基础上作了进一步提升。本文暂不讨论该部分内容

另外，作为 Spring 默认的调度框架，Quartz 很容易与 Spring 集成实现灵活可配置的调度功能。

下面是本文中用到的一些专用词汇，在此声明：

scheduler:

任务调度器

trigger:

触发器，用于定义任务调度时间规则

job:

任务，即被调度的任务

misfire:

错过的，指本来应该被执行但实际没有被执行的任务调度

Quartz 任务调度的基本实现原理

核心元素

Quartz 任务调度的核心元素是 scheduler, trigger 和 job，其中 trigger 和 job 是任务调度的元数据，scheduler 是实际执行调度的控制器。

在 Quartz 中，trigger 是用于定义调度时间的元素，即按照什么时间规则去执行任务。

Quartz 中主要提供了四种类型的 trigger：SimpleTrigger，CronTrigger，DateIntervalTrigger，和 NthIncludedDayTrigger。这四种 trigger 可以满足企业应用中的绝大部分需求。我们将在企业应用一节中进一步讨论四种 trigger 的功能。

在 Quartz 中，job 用于表示被调度的任务。主要有两种类型的 job：无状态的

(stateless) 和有状态的 (stateful)。对于同一个 trigger 来说，有状态的 job 不能被并行执行，只有上一次触发的任务被执行完之后，才能触发下一次执行。Job 主要有两种属性：volatility 和 durability，其中 volatility 表示任务是否被持久化到数据库存储，而 durability 表示在没有 trigger 关联的时候任务是否被保留。两者都是在值为 true 的时候任务被持久化或保留。一个 job 可以被多个 trigger 关联，但是一个 trigger 只能关联一个 job。

在 Quartz 中，scheduler 由 scheduler 工厂创建：DirectSchedulerFactory 或者

StdSchedulerFactory。第二种工厂 StdSchedulerFactory 使用较多，因为

DirectSchedulerFactory 使用起来不够方便，需要作许多详细的手工编码设置。

Scheduler 主要有三种：RemoteMBeanScheduler，RemoteScheduler 和

StdScheduler。本文以最常用的 StdScheduler 为例讲解。这也是笔者在项目中所使用的 scheduler 类。

Quartz 核心元素之间的关系如下图所示：

图 1. Quartz 核心元素关系图

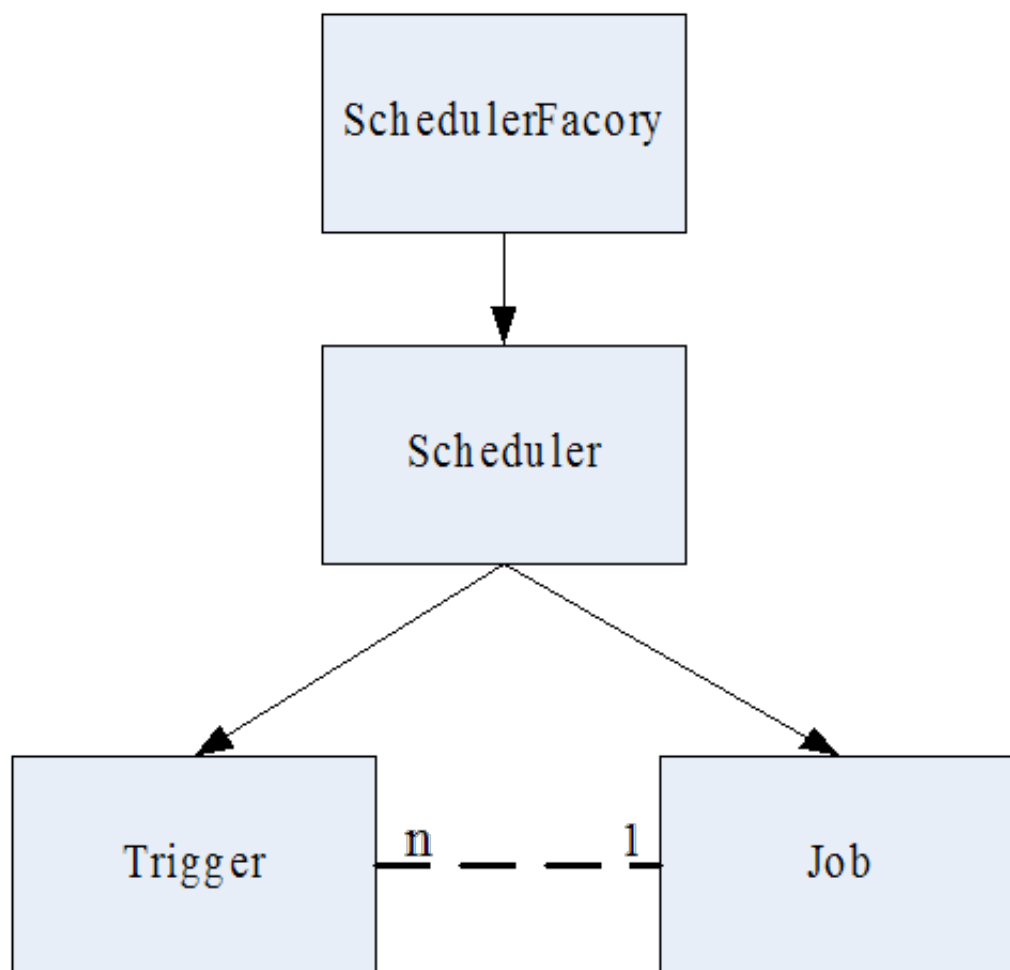


图 1. Quartz 核心元素关系图

线程视图

在 Quartz 中，有两类线程，Scheduler 调度线程和任务执行线程，其中任务执行线程通常使用一个线程池维护一组线程。

图 2. Quartz 线程视图

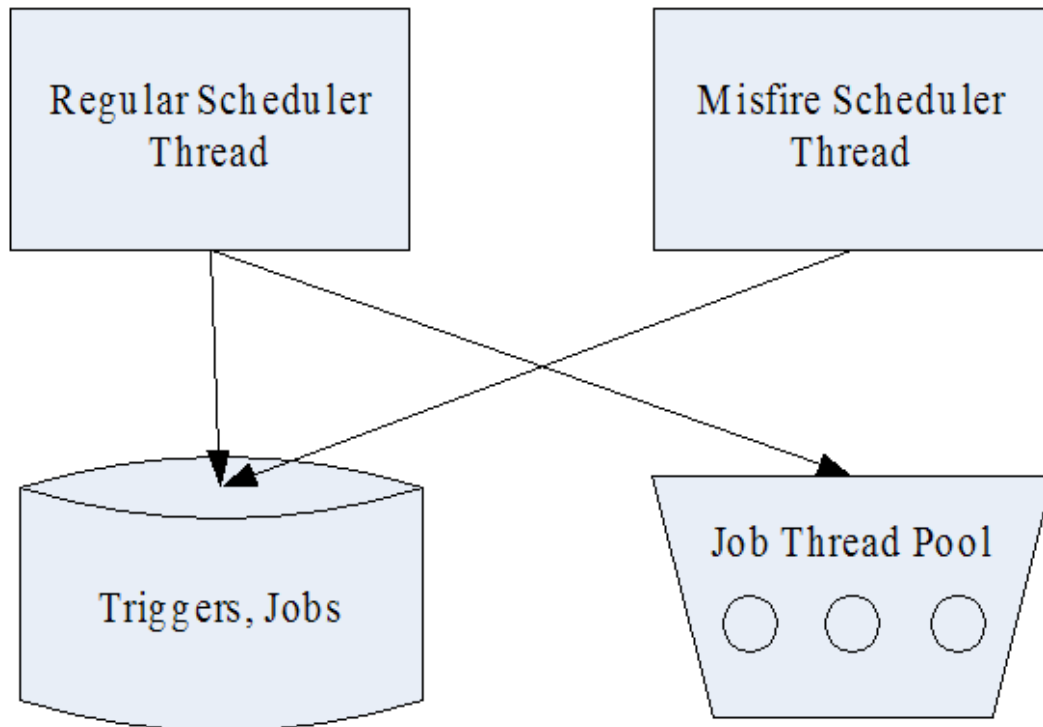


图 2. Quartz 线程视图

Scheduler 调度线程主要有两个：执行常规调度的线程，和执行 misfired trigger 的线程。常规调度线程轮询存储的所有 trigger，如果有需要触发的 trigger，即到达了下一次触发的时间，则从任务执行线程池获取一个空闲线程，执行与该 trigger 关联的任务。Misfire 线程是扫描所有的 trigger，查看是否有 misfired trigger，如果有的话根据 misfire 的策略分别处理。下图描述了这两个线程的基本流程：

图 3. Quartz 调度线程流程图

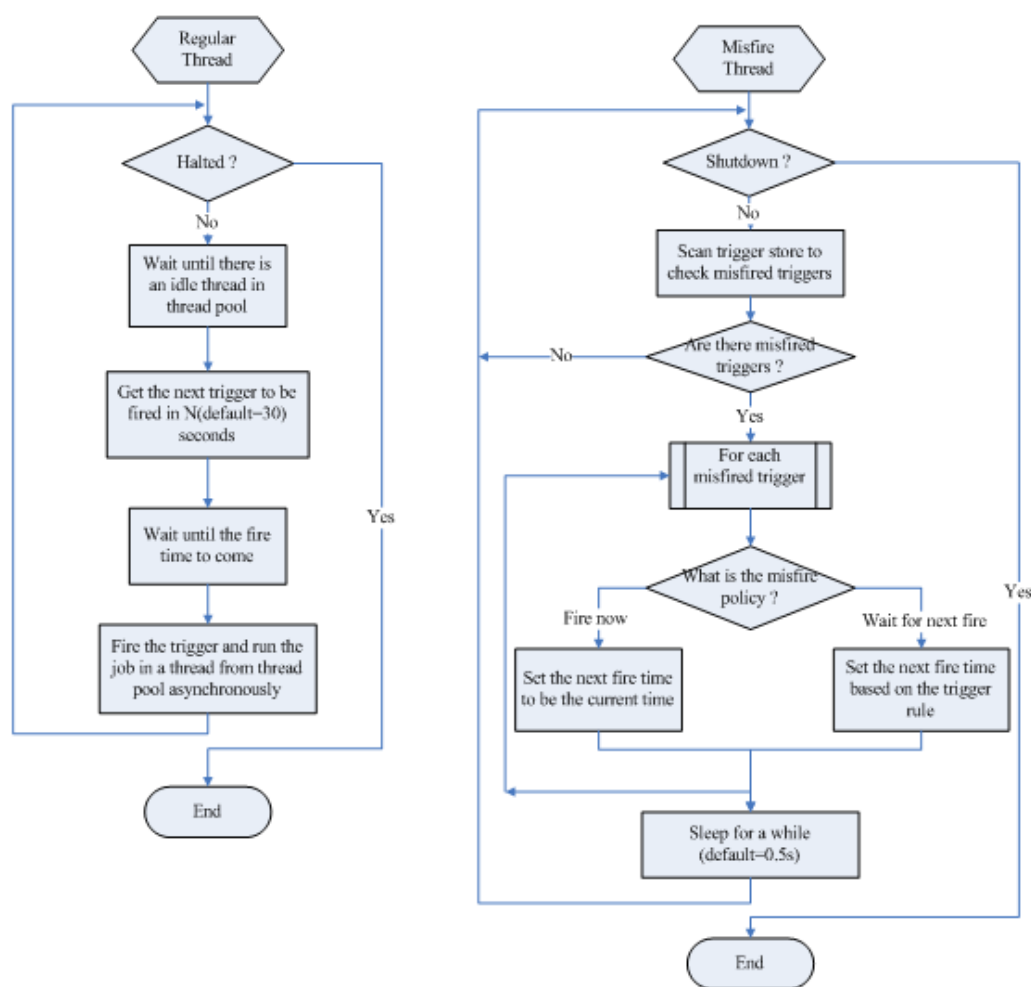


图 3. Quartz 调度线程流程图

关于 misfired trigger，我们在企业应用一节中将进一步描述。

数据存储

Quartz 中的 trigger 和 job 需要存储下来才能被使用。Quartz 中有两种存储方式：RAMJobStore, JobStoreSupport，其中 RAMJobStore 是将 trigger 和 job 存储在内存中，而 JobStoreSupport 是基于 jdbc 将 trigger 和 job 存储到数据库中。RAMJobStore 的存取速度非常快，但是由于其在系统被停止后所有的数据都会丢失，所以在通常应用中，都是使用 JobStoreSupport。

在 Quartz 中，JobStoreSupport 使用一个驱动代理来操作 trigger 和 job 的数据存储：StdJDBCDelegate。StdJDBCDelegate 实现了大部分基于标准 JDBC 的功能接口，但是对于各种数据库来说，需要根据其具体实现的特点做某些特殊处理，因此各种数据库需要扩展 StdJDBCDelegate 以实现这些特殊处理。Quartz 已经自带了一些数据库的扩展实现，可以直接使用，如下图所示：

图 4. Quartz 数据库驱动代理

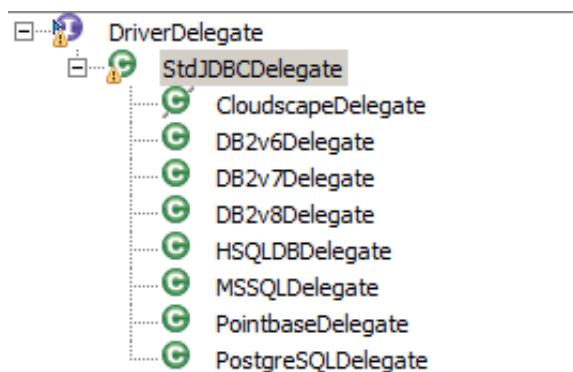


图 4. Quartz 数据库驱动代理

作为嵌入式数据库的代表，Derby 近来非常流行。如果使用 Derby 数据库，可以使用上图中的 CloudscapeDelegate 作为 trigger 和 job 数据存储的代理类。

[回页首](#)

基本开发流程及简单实例

搭建开发环境

利用 Quartz 进行开发相当简单，只需要将下载开发包中的 quartz-all-1.8.4.jar 加入到 classpath 即可。根据笔者的经验，对于任务调度功能比较复杂的企业级应用来说，最好在开发阶段将 Quartz 的源代码导入到开发环境中来。一方面可以通过阅读源码了解 Quartz 的实现机理，另一方面可以通过扩展或修改 Quartz 的一些类来实现某些 Quartz 尚不提供的功能。

图 5. Quartz 实例工程及源码导入

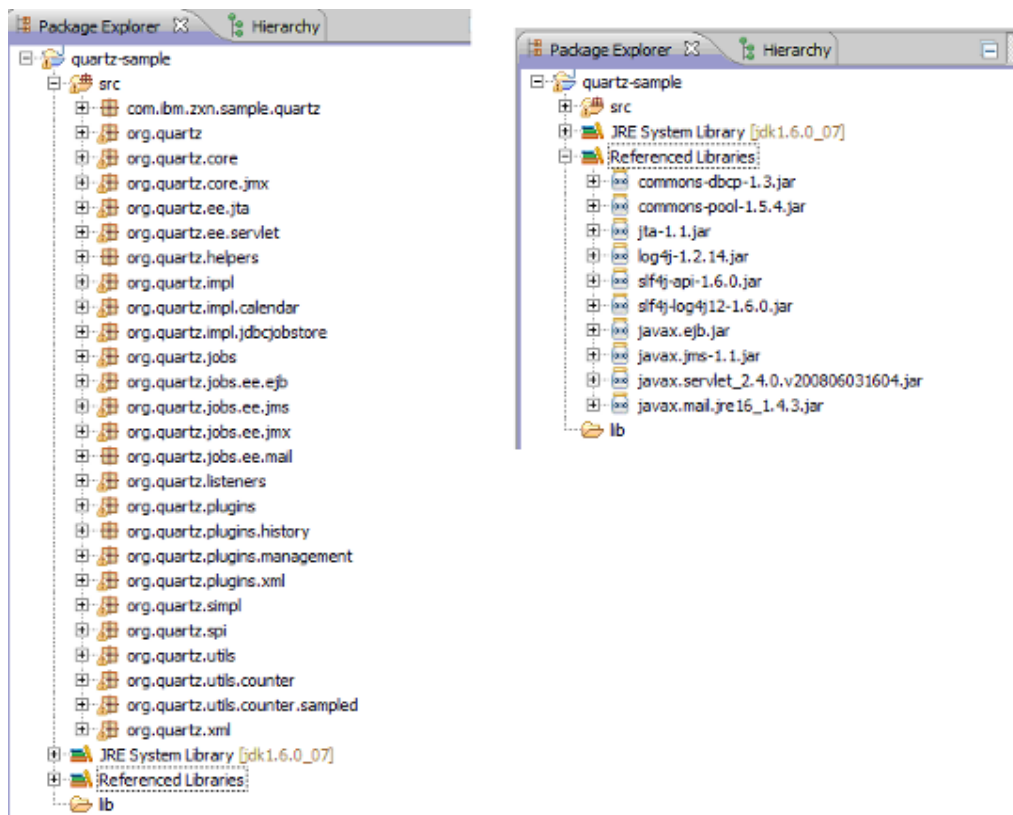


图 5. Quartz 实例工程及源码导入

上图中左边是源码导入后的截图，其中 org.quartz.* 即为 quartz 的源码。导入源码后可

能会有一些编译错误，通常出现在 `org.quartz.ee.*` 和 `org.quartz.jobs.ee.*` 包中。下载开发包中有一个 `lib` 目录，读者可以将该目录下的 `jar` 文件加入到编译环境。如果还有编译错误，读者可以参考上图中右侧的 `jar` 列表，到网上去搜索下载。

项目中 `com.ibm.zxn.sample.quartz` 是我们自己的类包，下面的实例中我们会用到它。

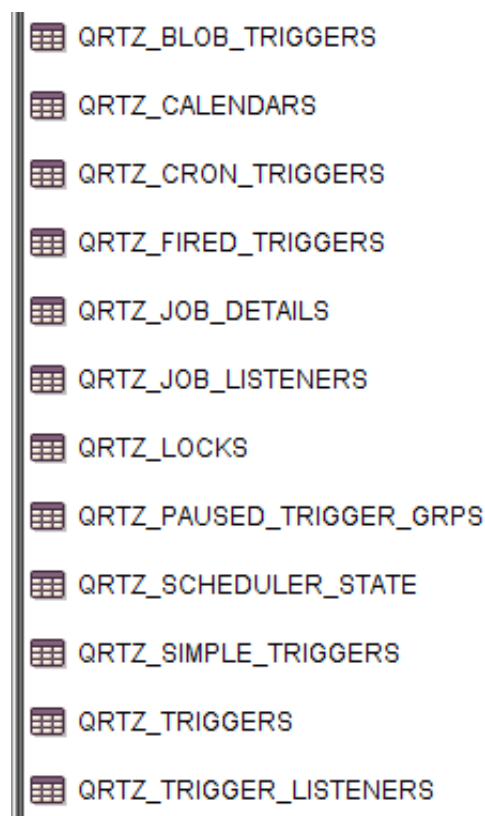
一个简单实例

Quartz 开发包中有一个 `examples` 目录，其中有 15 个基本实例。建议读者阅读并实践这些例子。本文这里只列举一个小的实例，介绍基本的开发方法。

1. 准备数据库和 Quartz 用的数据表

- 本文使用 IBM DB2 数据库：将 `jdbc` 驱动程序 `db2jcc.jar` 加入到项目中；
- 在数据库中创建一个新库 `QUARTZDB`；
- 执行 `/quartz-1.8.4/docs/dbTables/tables_db2_v8.sql`，创建数据表；表建好后如下所示：

图 6. Quartz 数据表



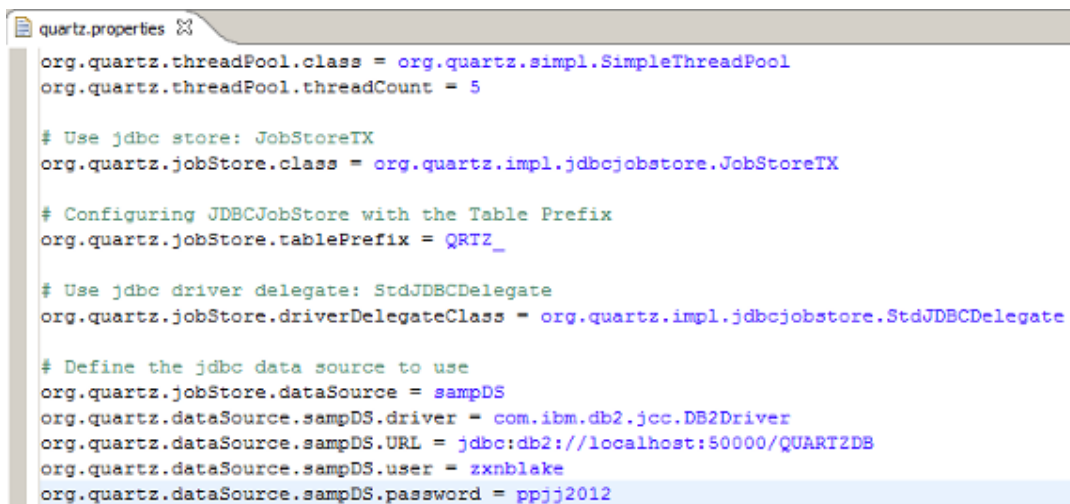
The image shows a vertical list of database tables, each preceded by a small icon representing a table grid. The tables are listed in the following order from top to bottom:

QRTZ_BLOB_TRIGGERS
QRTZ_CALENDARS
QRTZ_CRON_TRIGGERS
QRTZ_FIRED_TRIGGERS
QRTZ_JOB_DETAILS
QRTZ_JOB_LISTENERS
QRTZ_LOCKS
QRTZ_PAUSED_TRIGGER_GRP
QRTZ_SCHEDULER_STATE
QRTZ_SIMPLE_TRIGGERS
QRTZ_TRIGGERS
QRTZ_TRIGGER_LISTENERS

图 6. Quartz 数据表

2. 准备配置文件，加入到项目中

图 7. 实例配置文件



```

quartz.properties
org.quartz.threadPool.class = org.quartz.simpl.SimpleThreadPool
org.quartz.threadPool.threadCount = 5

# Use jdbc store: JobStoreTX
org.quartz.jobStore.class = org.quartz.impl.jdbcjobstore.JobStoreTX

# Configuring JDBCJobStore with the Table Prefix
org.quartz.jobStore.tablePrefix = QRTZ_

# Use jdbc driver delegate: StdJDBCDelegate
org.quartz.jobStore.driverDelegateClass = org.quartz.impl.jdbcjobstore.StdJDBCDelegate

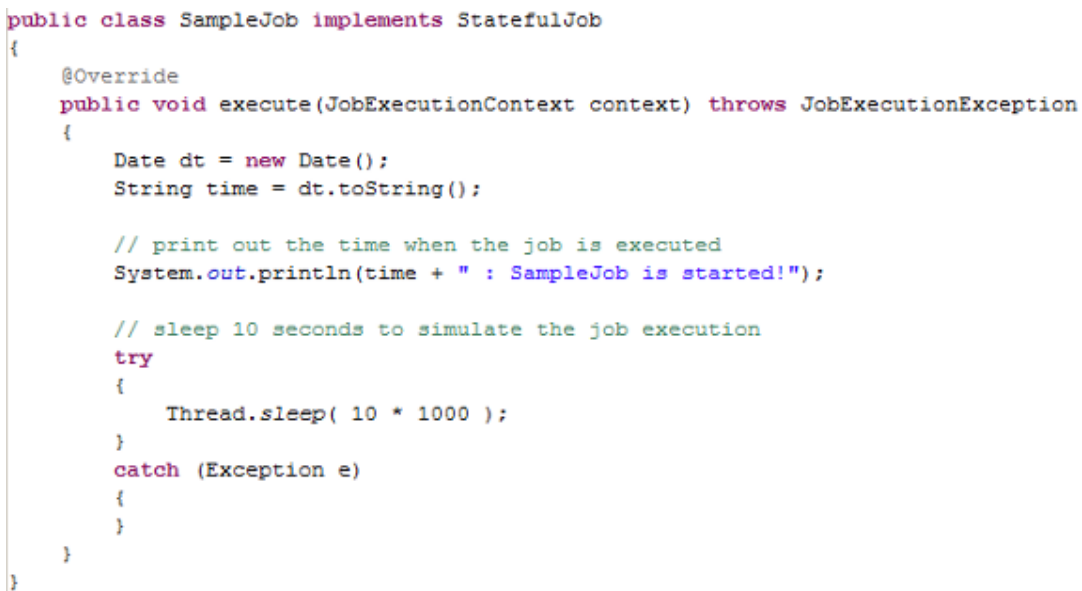
# Define the jdbc data source to use
org.quartz.jobStore.dataSource = sampDS
org.quartz.dataSource.sampDS.driver = com.ibm.db2.jcc.DB2Driver
org.quartz.dataSource.sampDS.URL = jdbc:db2://localhost:50000/QUARTZDB
org.quartz.dataSource.sampDS.user = zxnblake
org.quartz.dataSource.sampDS.password = ppjj2012

```

图 7. 实例配置文件

3. 通过实现 job 接口定义我们自己的任务类，如下所示：

图 8. 定义任务类



```

public class SampleJob implements StatefulJob
{
    @Override
    public void execute(JobExecutionContext context) throws JobExecutionException
    {
        Date dt = new Date();
        String time = dt.toString();

        // print out the time when the job is executed
        System.out.println(time + " : SampleJob is started!");

        // sleep 10 seconds to simulate the job execution
        try
        {
            Thread.sleep( 10 * 1000 );
        }
        catch (Exception e)
        {
        }
    }
}

```

图 8. 定义任务类

4. 然后，实现任务调度的主程序，如下所示：

本实例中，我们利用 DateIntervalTrigger 实现一个每两分钟执行一次的任务调度。

图 9. 实现主程序

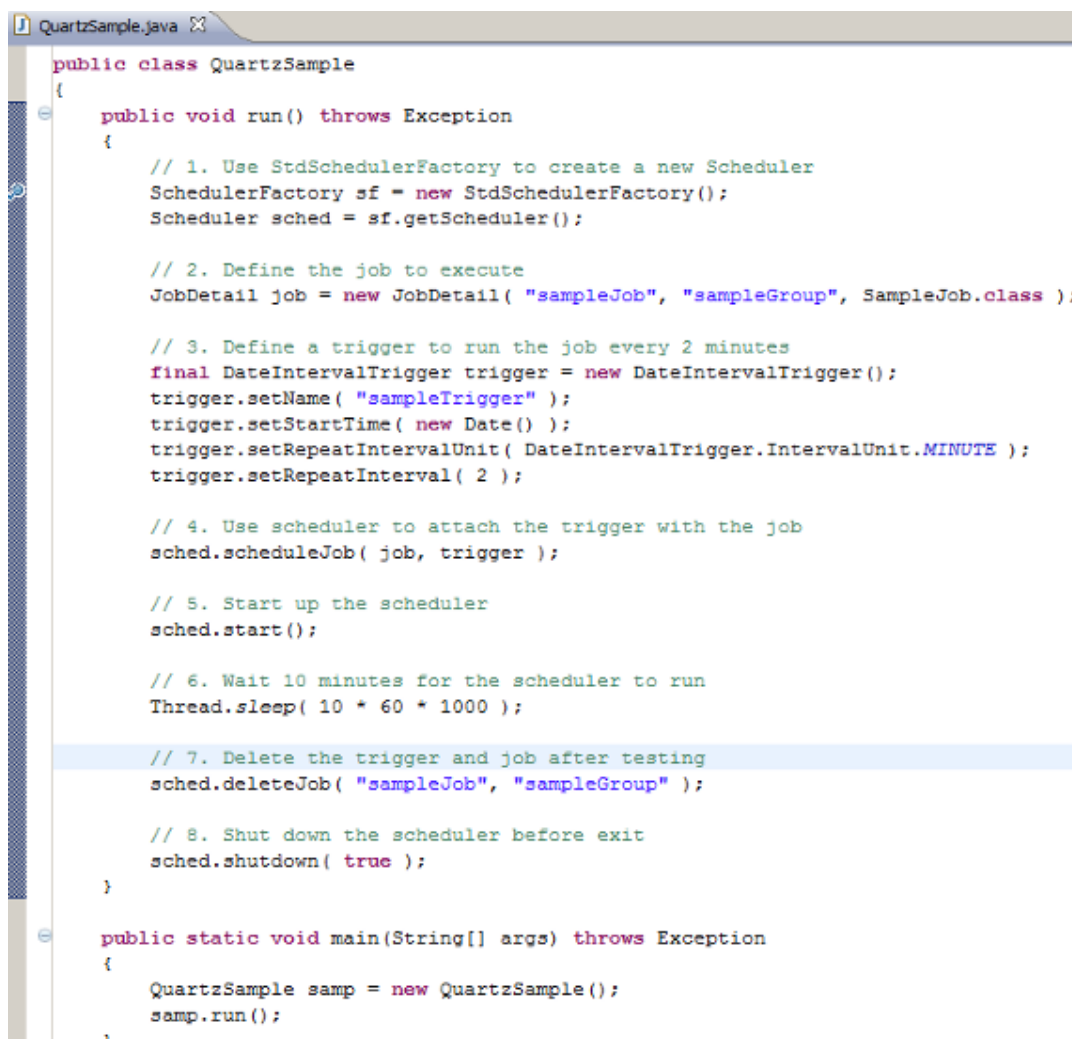


图 9. 实现主程序

5. 完成后项目结构如下所示：

图 10. 实例项目结构图

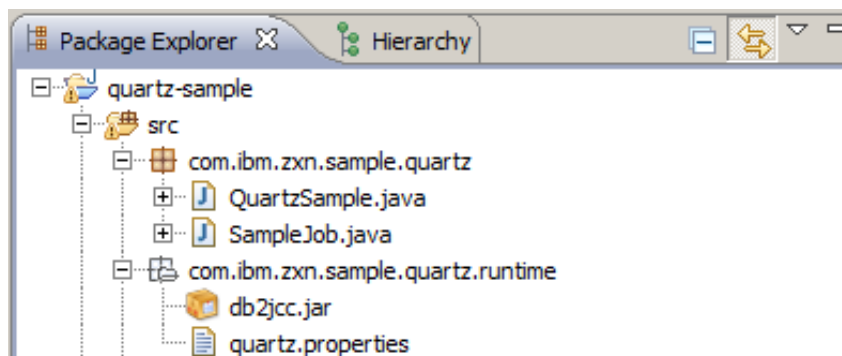


图 10. 实例项目结构图

6. 运行程序，查看数据库表和运行结果

数据库中，QRTZ_TRIGGERS 表中添加了一条 trigger 记录，如下所示：

图 11. QRTZ_TRIGGERS 表中的记录

TRIGGER_NAME	TRIGGER_GROUP	JOB_NAME	JOB_GROUP	IS_VOLATILE	DESCRIPTION	NEXT_FIRE_TIME
sampleTrigger	DEFAULT	sampleJob	sampleGroup	0		1349336524420

图 11. QRTZ_TRIGGERS 表中的记录

QRTZ_JOB_DETAILS 表中添加了一条 job 记录，如下所示：

图 12. QRTZ_JOB_DETAILS 表中的记录

JOB_NAME	JOB_GROUP	DESCRIPTION	JOB_CLASS_NAME	IS_DURABLE	IS_VOLATILE
sampleJob	sampleGroup		com.ibm.zxn.sample.quartz.SampleJob	0	0

图 12. QRTZ_JOB_DETAILS 表中的记录

从运行结果来看，任务每两分钟被执行一次：

图 13. 运行结果

```
Thu Oct 04 15:52:36 CST 2012 : SampleJob is started!
Thu Oct 04 15:54:36 CST 2012 : SampleJob is started!
Thu Oct 04 15:56:36 CST 2012 : SampleJob is started!
Thu Oct 04 15:58:36 CST 2012 : SampleJob is started!
```

图 13. 运行结果

[回页首](#)

企业级开发中的常见应用

在应用 Quartz 进行企业级的开发时，有一些问题会经常遇到。本节笔者根据自己在项目开发中的经验，介绍企业开发中常见的一些问题以及通常的解决办法。

应用一：如何使用不同类型的 Trigger

前面我们提到 Quartz 中四种类型的 Trigger：SimpleTrigger，CronTrigger，DateIntervalTrigger，和 NthIncludedDayTrigger。

SimpleTrigger 一般用于实现每隔一定时间执行任务，以及重复多少次，如每 2 小时执行一次，重复执行 5 次。SimpleTrigger 内部实现机制是通过计算间隔时间来计算下次的执行时间，这就导致其不适合调度定时的任务。例如我们想每天的 1:00AM 执行任务，如果使用 SimpleTrigger 的话间隔时间就是一天。注意这里就会有一个问题，即当有 misfired 的任务并且恢复执行时，该执行时间是随机的（取决于何时执行 misfired 的任务，例如某天的 3:00PM）。这会导致之后每天的执行时间都会变成 3:00PM，而不是我们原来期望的 1:00AM。

CronTrigger 类似于 LINUX 上的任务调度命令 crontab，即利用一个包含 7 个字段的表达式来表示时间调度方式。例如，"0 15 10 * * ? *" 表示每天的 10:15AM 执行任务。对于涉及到星期和月份的调度，CronTrigger 是最适合的，甚至某些情况下是唯一选择。例如，"0 10 14 ? 3 WED" 表示三月份的每个星期三的下午 14:10PM 执行任务。读者可以在具体用到该 trigger 时再详细了解每个字段的含义。

DateIntervalTrigger 是 Quartz 1.7 之后的版本加入的，其最适合调度类似每 N（1, 2, 3...）小时，每 N 天，每 N 周等的任务。虽然 SimpleTrigger 也能实现类似的任务，但是 DateIntervalTrigger 不会受到我们上面说到的 misfired 任务的影响。另外，DateIntervalTrigger 也不会受到 DST（Daylight Saving Time，即中国的夏令时）调整的影响。笔者就曾经因为该原因将项目中的 SimpleTrigger 改为了 DateIntervalTrigger，因为如果使用 SimpleTrigger，本来设定的调度时间就会由于 DST 的调整而提前或延迟一个小时，而 DateIntervalTrigger 不会受此影响。

NthIncludedDayTrigger 的用途比较简单明确，即用于每隔一个周期的第几天调度任务，例如，每个月的第 3 天执行指定的任务。

除了上面提到的 4 种 Trigger，Quartz 中还定义了一个 Calendar 类（注意，是 org.quartz.Calendar）。这个 Calendar 与 Trigger 一起使用，但是它们的作用相反，它是用于排除任务不被执行的情况。例如，按照 Trigger 的规则在 10 月 1 号需要执行任务，但是 Calendar 指定了 10 月 1 号是节日（国庆），所以任务在这一天将不会被执行。通常来说，Calendar 用于排除节假日的任务调度，从而使任务只在工作日执行。

应用二：使用有状态（StatefulJob）还是无状态的任务（Job）

在 Quartz 中，Job 是一个接口，企业应用需要实现这个接口以定义自己的任务。基本来说，任务分为有状态和无状态两种。实现 Job 接口的任务缺省为无状态的。Quartz 中还有另外一个接口 StatefulJob。实现 StatefulJob 接口的任务为有状态的，上一节的简单实例中，我们定义的 SampleJob 就是实现了 StatefulJob 接口的有状态任务。下图列出了 Quartz 中 Job 接口的定义以及一些自带的实现类：

图 14. Quartz 中 Job 接口定义

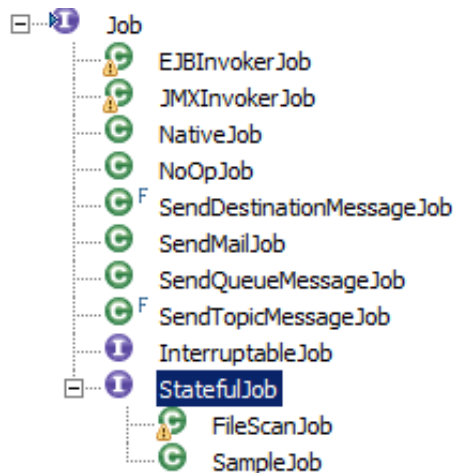


图 14. Quartz 中 Job 接口定义

无状态任务一般指可以并发的任务，即任务之间是独立的，不会互相干扰。例如我们定义一个 trigger，每 2 分钟执行一次，但是某些情况下一个任务可能需要 3 分钟才能执行完，这样，在上一个任务还处在执行状态时，下一次触发时间已经到了。对于无状态任务，只要触发时间到了就会被执行，因为几个相同任务可以并发执行。但是对有状态任务来说，是不能并发执行的，同一时间只能有一个任务在执行。

在笔者项目中，某些任务需要对数据库中的数据进行增删改处理。这些任务不能并发执行，否则会造成数据混乱。因此我们使用 StatefulJob 接口。现在回到上面的例子，任务每 2 分钟执行一次，若某次任务执行了 5 分钟才完成，Quartz 会怎么处理呢？按照 trigger 的规则，第 2 分钟和第 4 分钟分别会有一次预定的触发执行，但是由于是有状态任务，因此实际不会被触发。在第 5 分钟第一次任务执行完毕时，Quartz 会把第 2 和第 4 分钟的两次触发作为 misfired job 进行处理。对于 misfired job，Quartz 会查看其 misfire 策略是如何设定的，如果是立刻执行，则会马上启动一次执行，如果是等待下次执行，则会忽略错过的任务，而等待下次（即第 6 分钟）触发执行。

读者可以在自己的项目中体会两种任务的区别以及 Quartz 的处理方法，根据具体情况选择不同类型的任务。

应用三：如何设置 Quartz 的线程池和并发任务

Quartz 中自带了一个线程池的实现：SimpleThreadPool。类如其名，这只是线程池的一个简单实现，没有提供动态自发调整等高级特性。Quartz 提供了一个配置参数：

org.quartz.threadPool.threadCount，可以在初始化时设定线程池的线程数量，但是一旦设定后不能再修改。假定这个数目是 10，则在并发任务达到 10 个以后，再有触发的任务就无法被执行了，只能等待有空闲线程的时候才能得到执行。因此有些 trigger 就可能被 misfire。但是必须指出一点，这个初始线程数并不是越大越好。当并发线程太多时，系统整体性能反而会下降，因为系统把很多时间花在了线程调度上。根据一般经验，这个值在 10 -- 50 比较合适。

对于一些注重性能的线程池来说，会根据实际线程使用情况进行动态调整，例如初始线程数，最大线程数，空闲线程数等。读者在应用中，如果有更好的线程池，则可以在配置文件中通过下面参数替换 SimpleThreadPool：org.quartz.threadPool.class = myapp.GreatThreadPool。

应用四：如何处理 Misfired 任务

在 Quartz 应用中，misfired job 是经常遇到的情况。一般来说，下面这些原因可能造成

misfired job:

- 1) 系统因为某些原因被重启。在系统关闭到重新启动之间的一段时间里，可能有些任务会被 misfire；
- 2) Trigger 被暂停 (suspend) 的一段时间里，有些任务可能会被 misfire；
- 3) 线程池中所有线程都被占用，导致任务无法被触发执行，造成 misfire；
- 4) 有状态任务在下次触发时间到达时，上次执行还没有结束；

为了处理 misfired job，Quartz 中为 trigger 定义了处理策略，主要有下面两种：

MISFIRE_INSTRUCTION_FIRE_ONCE_NOW：针对 misfired job 马上执行一次；

MISFIRE_INSTRUCTION_DO_NOTHING：忽略 misfired job，等待下次触发；

建议读者在应用开发中，将该设置作为可配置选项，使得用户可以在使用过程中，针对已经添加的 trigger 动态配置该选项。

应用五：如何保留已经结束的 Trigger

在 Quartz 中，一个 trigger 在最后一次触发完成之后，会被自动删除。Quartz 默认不会保留已经结束的 trigger，如下面 Quartz 源代码所示：

图 15. executionComplete() 源码

```
public int executionComplete(JobExecutionContext context,
    JobExecutionException result) {
    if (result != null && result.refireImmediately()) {
        return INSTRUCTION_RE_EXECUTE_JOB;
    }

    if (result != null && result.unscheduleFiringTrigger()) {
        return INSTRUCTION_SET_TRIGGER_COMPLETE;
    }

    if (result != null && result.unscheduleAllTriggers()) {
        return INSTRUCTION_SET_ALL_JOB_TRIGGERS_COMPLETE;
    }

    if (!mayFireAgain()) {
        return INSTRUCTION_DELETE_TRIGGER;
    }

    return INSTRUCTION_NOOP;
}
```

图 15. executionComplete() 源码

但是在实际应用中，有些用户需要保留以前的 trigger，作为历史记录，或者作为以后创建其他 trigger 的依据。如何保留结束的 trigger 呢？

一个办法是应用开发者自己维护一份数据备份记录，并且与 Quartz 原表的记录保持一定的同步。这个办法实际操作起来比较繁琐，而且容易出错，不推荐使用。

另外一个办法是通过修改并重新编译 Quartz 的 trigger 类，修改其默认的行为。我们以 org.quartz.SimpleTrigger 为例，修改上面代码中 if (!mayFireAgain()) 部分的代码如下：

图 16. 修改 executionComplete() 源码

```

if (!mayFireAgain())
{
    if ( !isNeedRetain() )
    {
        return INSTRUCTION_DELETE_TRIGGER;
    }
}

```

图 16. 修改 executionComplete() 源码

另外我们需要在 SimpleTrigger 中定义一个新的类属性：needRetain，如下所示：

图 17. 定义新属性 needRetain

```

private boolean needRetain = false;
public boolean isNeedRetain()
{
    return needRetain;
}
public void setNeedRetain(boolean needRetain)
{
    this.needRetain = needRetain;
}

```

图 17. 定义新属性 needRetain

在定义自己的 trigger 时，设置该属性，就可以选择是否在 trigger 结束时删除 trigger。如下代码所示：

图 18. 使用修改后的 SimpleTrigger

```

SimpleTrigger trigger = new SimpleTrigger();
trigger.setName( "retainedTrigger" );
trigger.setStartTime( new Date() );
trigger.setRepeatInterval( 60 * 1000 );
trigger.setRepeatCount( 5 );

// We want the trigger to be retained after completion
trigger.setNeedRetain( true );

```

图 18. 使用修改后的 SimpleTrigger

有人可能会考虑通过定义一个新的类，然后继承 org.quartz.SimpleTrigger 类并覆盖 executionComplete() 方法来实现。但是这种方法是行不通的，因为 Quartz 内部在处理时会根据 trigger 的类型重新生成 SimpleTrigger 类的实例，而不是使用我们自己定义类创建的实例。这一点应该是 Quartz 的一个小小的不足之处，因为它把扩展 trigger 的能力堵死了。好在 Quartz 是开源的，我们可以根据需要进行修改。

[回页首](#)

小结

作为当前颇具生命力的开源框架，Quartz 已经得到了广泛的应用。Quartz 的强大功能和应用灵活性，在企业应用中发挥了巨大的作用。本文描述了如何应用 Quartz 开发应用程序，并对企业应用中常见的问题及解决方案进行了讨论。