|  |
| --- |
| **任务调度框架Quartz** |
| [建立高效可控的任务调度系统] |
|  |
| [各种企业应用几乎都会碰到任务调度的需求,e.g.:定时清理系统垃圾文件,定时导入导出数据,定时发送邮件等等业务场景,Quartz提供了这样的功能,并且高度可控,本文将带您一起领略Quartz的风采] |
|  |
| **注:资料来源于网络,版权归原作者所有,仅供学习之用** |
| **[2010-11-30][村长]** |
|  |

目录

[Quartz任务调度 3](#_Toc278876151)

[quartz简介 3](#_Toc278876152)

[Quartz体系结构 3](#_Toc278876153)

[Quartz实例入门 6](#_Toc278876154)

[任务调度信息存储 16](#_Toc278876155)

[小结 20](#_Toc278876156)

[附录1 21](#_Toc278876157)

Quartz任务调度

# quartz简介

各种企业应用几乎都会碰到任务调度的需求，就拿论坛来说：每隔半个小时生成精华文章的RSS文件，每天凌晨统计论坛用户的积分排名，每隔30分钟执行锁定用户解锁任务。

对于一个典型的MIS系统来说，在每月1号凌晨统计上个月各部门的业务数据生成月报表，每半个小时查询用户是否已经有快到期的待处理业务……，这样的例子俯拾皆是，不胜枚举。

任务调度本身涉及到多线程并发、运行时间规则制定和解析、场景保持与恢复、线程池维护等诸多方面的工作。如果直接使用自定义线程这种刀耕火种的原始办法，开发任务调度程序是一项颇具挑战性的工作。Java开源的好处就是：领域问题都能找到现成的解决方案。

OpenSymphony所提供的Quartz自2001年发布版本以来已经被众多项目作为任务调度的解决方案，Quartz在提供巨大灵活性的同时并未牺牲其简单性，它所提供的强大功能使你可以应付绝大多数的调度需求。

Quartz 在开源任务调度框架中的翘首，它提供了强大任务调度机制，难能可贵的是它同时保持了使用的简单性。Quartz 允许开发人员灵活地定义触发器的调度时间表，并可以对触发器和任务进行关联映射。

此外，Quartz提供了调度运行环境的持久化机制，可以保存并恢复调度现场，即使系统因故障关闭，任务调度现场数据并不会丢失。此外，Quartz还提供了组件式的侦听器、各种插件、线程池等功能。

# Quartz体系结构

Quartz对任务调度的领域问题进行了高度的抽象，提出了调度器、任务和触发器这3个核心的概念，并在org.quartz通过接口和类对重要的这些核心概念进行描述：

●Job：是一个接口，只有一个方法void execute(JobExecutionContext context)，开发者实现该接口定义运行任务，JobExecutionContext类提供了调度上下文的各种信息。Job运行时的信息保存在JobDataMap实例中；

●JobDetail：Quartz在每次执行Job时，都重新创建一个Job实例，所以它不直接接受一个Job的实例，相反它接收一个Job实现类，以便运行时通过newInstance()的反射机制实例化Job。因此需要通过一个类来描述Job的实现类及其它相关的静态信息，如Job名字、描述、关联监听器等信息，JobDetail承担了这一角色。

通过该类的构造函数可以更具体地了解它的功用：JobDetail(java.lang.String name, java.lang.String group, java.lang.Class jobClass)，该构造函数要求指定Job的实现类，以及任务在Scheduler中的组名和Job名称；

●Trigger：是一个类，描述触发Job执行的时间触发规则。主要有SimpleTrigger和CronTrigger这两个子类。当仅需触发一次或者以固定时间间隔周期执行，SimpleTrigger是最适合的选择；而CronTrigger则可以通过Cron表达式定义出各种复杂时间规则的调度方案：如每早晨9:00执行，周一、周三、周五下午5:00执行等；

●Calendar：org.quartz.Calendar和java.util.Calendar不同，它是一些日历特定时间点的集合（可以简单地将org.quartz.Calendar看作java.util.Calendar的集合——java.util.Calendar代表一个日历时间点，无特殊说明后面的Calendar即指org.quartz.Calendar）。一个Trigger可以和多个Calendar关联，以便排除或包含某些时间点。

假设，我们安排每周星期一早上10:00执行任务，但是如果碰到法定的节日，任务则不执行，这时就需要在Trigger触发机制的基础上使用Calendar进行定点排除。针对不同时间段类型，Quartz在org.quartz.impl.calendar包下提供了若干个Calendar的实现类，如AnnualCalendar、MonthlyCalendar、WeeklyCalendar分别针对每年、每月和每周进行定义；

●Scheduler：代表一个Quartz的独立运行容器，Trigger和JobDetail可以注册到Scheduler中，两者在Scheduler中拥有各自的组及名称，组及名称是Scheduler查找定位容器中某一对象的依据，Trigger的组及名称必须唯一，JobDetail的组和名称也必须唯一（但可以和Trigger的组和名称相同，因为它们是不同类型的）。Scheduler定义了多个接口方法，允许外部通过组及名称访问和控制容器中Trigger和JobDetail。

Scheduler可以将Trigger绑定到某一JobDetail中，这样当Trigger触发时，对应的Job就被执行。一个Job可以对应多个Trigger，但一个Trigger只能对应一个Job。可以通过SchedulerFactory创建一个Scheduler实例。Scheduler拥有一个SchedulerContext，它类似于ServletContext，保存着Scheduler上下文信息，Job和Trigger都可以访问SchedulerContext内的信息。SchedulerContext内部通过一个Map，以键值对的方式维护这些上下文数据，SchedulerContext为保存和获取数据提供了多个put()和getXxx()的方法。可以通过Scheduler# getContext()获取对应的SchedulerContext实例；

●ThreadPool：Scheduler使用一个线程池作为任务运行的基础设施，任务通过共享线程池中的线程提高运行效率。

Job有一个StatefulJob子接口，代表有状态的任务，该接口是一个没有方法的标签接口，其目的是让Quartz知道任务的类型，以便采用不同的执行方案。无状态任务在执行时拥有自己的JobDataMap拷贝，对JobDataMap的更改不会影响下次的执行。而有状态任务共享共享同一个JobDataMap实例，每次任务执行对JobDataMap所做的更改会保存下来，后面的执行可以看到这个更改，也即每次执行任务后都会对后面的执行发生影响。

正因为这个原因，无状态的Job可以并发执行，而有状态的StatefulJob不能并发执行，这意味着如果前次的StatefulJob还没有执行完毕，下一次的任务将阻塞等待，直到前次任务执行完毕。有状态任务比无状态任务需要考虑更多的因素，程序往往拥有更高的复杂度，因此除非必要，应该尽量使用无状态的Job。

如果Quartz使用了数据库持久化任务调度信息，无状态的JobDataMap仅会在Scheduler注册任务时保持一次，而有状态任务对应的JobDataMap在每次执行任务后都会进行保存。

Trigger自身也可以拥有一个JobDataMap，其关联的Job可以通过JobExecutionContext#getTrigger().getJobDataMap()获取Trigger中的JobDataMap。不管是有状态还是无状态的任务，在任务执行期间对Trigger的JobDataMap所做的更改都不会进行持久，也即不会对下次的执行产生影响。

Quartz拥有完善的事件和监听体系，大部分组件都拥有事件，如任务执行前事件、任务执行后事件、触发器触发前事件、触发后事件、调度器开始事件、关闭事件等等，可以注册相应的监听器处理感兴趣的事件。

图1描述了Scheduler的内部组件结构，SchedulerContext提供Scheduler全局可见的上下文信息，每一个任务都对应一个JobDataMap，虚线表达的JobDataMap表示对应有状态的任务：

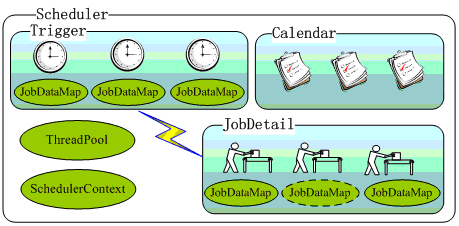


图1 Scheduler结构图

一个Scheduler可以拥有多个Triger组和多个JobDetail组，注册Trigger和JobDetail时，如果不显式指定所属的组，Scheduler将放入到默认组中，默认组的组名为Scheduler.DEFAULT\_GROUP。组名和名称组成了对象的全名，同一类型对象的全名不能相同。

Scheduler本身就是一个容器，它维护着Quartz的各种组件并实施调度的规则。Scheduler还拥有一个线程池，线程池为任务提供执行线程——这比执行任务时简单地创建一个新线程要拥有更高的效率，同时通过共享节约资源的占用。通过线程池组件的支持，对于繁忙度高、压力大的任务调度，Quartz将可以提供良好的伸缩性。

**提示：** Quartz完整下载包examples目录下拥有10多个实例，它们是快速掌握Quartz应用很好的实例。

# Quartz实例入门

#### 使用SimpleTrigger

SimpleTrigger拥有多个重载的构造函数，用以在不同场合下构造出对应的实例：

●SimpleTrigger(String name, String group)：通过该构造函数指定Trigger所属组和名称；

●SimpleTrigger(String name, String group, Date startTime)：除指定Trigger所属组和名称外，还可以指定触发的开发时间；

●SimpleTrigger(String name, String group, Date startTime, Date endTime, int repeatCount, long repeatInterval)：除指定以上信息外，还可以指定结束时间、重复执行次数、时间间隔等参数；

●SimpleTrigger(String name, String group, String jobName, String jobGroup, Date startTime, Date endTime, int repeatCount, long repeatInterval)：这是最复杂的一个构造函数，在指定触发参数的同时，还通过jobGroup和jobName，让该Trigger和Scheduler中的某个任务关联起来。

通过实现 org.quartz..Job 接口，可以使 Java 类化身为可调度的任务。代码清单1提供了 Quartz 任务的一个示例：

代码清单1 SimpleJob：简单的Job实现类

package com.baobaotao.basic.quartz;

import java.util.Date;

import org.quartz.Job;

import org.quartz.JobExecutionContext;

import org.quartz.JobExecutionException;

public class SimpleJob implements Job {

①实例Job接口方法

public void execute(JobExecutionContext jobCtx)throws JobExecutionException {

System.out.println(jobCtx.getTrigger().getName()+ " triggered. time is:" + (new Date()));

}

}

这个类用一条非常简单的输出语句实现了Job接口的execute(JobExecutionContext context) 方法，这个方法可以包含想要执行的任何代码。下面，我们通过SimpleTrigger对SimpleJob进行调度：

代码清单2 SimpleTriggerRunner：使用SimpleTrigger进行调度

package com.baobaotao.basic.quartz;

import java.util.Date;

import org.quartz.JobDetail;

import org.quartz.Scheduler;

import org.quartz.SchedulerFactory;

import org.quartz.SimpleTrigger;

import org.quartz.impl.StdSchedulerFactory;

public class SimpleTriggerRunner {

public static void main(String args[]) {

try {

**①创建一个JobDetail实例，指定SimpleJob**

JobDetail jobDetail = new JobDetail("job1\_1","jGroup1", SimpleJob.class);

**②通过SimpleTrigger定义调度规则：马上启动，每2秒运行一次，共运行100次**

SimpleTrigger simpleTrigger = new SimpleTrigger("trigger1\_1","tgroup1");

simpleTrigger.setStartTime(new Date());

simpleTrigger.setRepeatInterval(2000);

simpleTrigger.setRepeatCount(100);

**③通过SchedulerFactory获取一个调度器实例**

SchedulerFactory schedulerFactory = new StdSchedulerFactory();

Scheduler scheduler = schedulerFactory.getScheduler();

scheduler.scheduleJob(jobDetail, simpleTrigger);**④ 注册并进行调度**

scheduler.start();**⑤调度启动**

} catch (Exception e) {

e.printStackTrace();

}

}

}

首先在①处通过JobDetail封装SimpleJob，同时指定Job在Scheduler中所属组及名称，这里，组名为jGroup1，而名称为job1\_1。

在②处创建一个SimpleTrigger实例，指定该Trigger在Scheduler中所属组及名称。接着设置调度的时间规则。

最后，需要创建Scheduler实例，并将JobDetail和Trigger实例注册到Scheduler中。这里，我们通过StdSchedulerFactory获取一个Scheduler实例，并通过scheduleJob(JobDetail jobDetail, Trigger trigger)完成两件事：

1)将JobDetail和Trigger注册到Scheduler中；

2)将Trigger指派给JobDetail，将两者关联起来。

当Scheduler启动后，Trigger将定期触发并执行SimpleJob的execute(JobExecutionContext jobCtx)方法，然后每 10 秒重复一次，直到任务被执行 100 次后停止。

还可以通过SimpleTrigger的setStartTime(java.util.Date startTime)和setEndTime(java.util.Date endTime)指定运行的时间范围，当运行次数和时间范围冲突时，超过时间范围的任务运行不被执行。如可以通过simpleTrigger.setStartTime(new Date(System.currentTimeMillis() + 60000L))指定60秒钟以后开始。

除了通过scheduleJob(jobDetail, simpleTrigger)建立Trigger和JobDetail的关联，还有另外一种关联Trigger和JobDetail的方式：

JobDetail jobDetail = new JobDetail("job1\_1","jGroup1", SimpleJob.class);

SimpleTrigger simpleTrigger = new SimpleTrigger("trigger1\_1","tgroup1");

…

simpleTrigger.setJobGroup("jGroup1");**①-1：指定关联的Job组名**

simpleTrigger.setJobName("job1\_1");**①-2：指定关联的Job名称**

scheduler.addJob(jobDetail, true);**② 注册JobDetail**

scheduler.scheduleJob(simpleTrigger);**③ 注册指定了关联JobDetail的Trigger**

在这种方式中，Trigger通过指定Job所属组及Job名称，然后使用Scheduler的scheduleJob(Trigger trigger)方法注册Trigger。有两个值得注意的地方：

通过这种方式注册的Trigger实例必须已经指定Job组和Job名称，否则调用注册Trigger的方法将抛出异常；

引用的JobDetail对象必须已经存在于Scheduler中。也即，代码中①、②和③的先后顺序不能互换。

在构造Trigger实例时，可以考虑使用org.quartz.TriggerUtils工具类，该工具类不但提供了众多获取特定时间的方法，还拥有众多获取常见Trigger的方法，如makeSecondlyTrigger(String trigName)方法将创建一个每秒执行一次的Trigger，而makeWeeklyTrigger(String trigName, int dayOfWeek, int hour, int minute)将创建一个每星期某一特定时间点执行一次的Trigger。而getEvenMinuteDate(Date date)方法将返回某一时间点一分钟以后的时间。

1. **使用CronTrigger**

CronTrigger 能够提供比 SimpleTrigger 更有具体实际意义的调度方案，调度规则基于 Cron 表达式，CronTrigger 支持日历相关的重复时间间隔（比如每月第一个周一执行），而不是简单的周期时间间隔。因此，相对于SimpleTrigger而言，CronTrigger在使用上也要复杂一些。

**Cron表达式**

Quartz使用类似于Linux下的Cron表达式定义时间规则，Cron表达式由6或7个由空格分隔的时间字段组成，如表1所示：

表1 Cron表达式时间字段

|  |  |  |  |
| --- | --- | --- | --- |
| 位置 | 时间域名 | 允许值 | 允许的特殊字符 |
| 1 | 秒 | 0-59 | , - \* / |
| 2 | 分钟 | 0-59 | , - \* / |
| 3 | 小时 | 0-23 | , - \* / |
| 4 | 日期 | 1-31 | , - \* ? / L W C |
| 5 | 月份 | 1-12 | , - \* / |
| 6 | 星期 | 1-7 | , - \* ? / L C # |
| 7 | 年(可选) | 空值1970-2099 | , - \* / |

Cron表达式的时间字段除允许设置数值外，还可使用一些特殊的字符，提供列表、范围、通配符等功能，细说如下：

●星号(\*)：可用在所有字段中，表示对应时间域的每一个时刻，例如，\*在分钟字段时，表示“每分钟”；

●问号（?）：该字符只在日期和星期字段中使用，它通常指定为“无意义的值”，相当于点位符；

●减号(-)：表达一个范围，如在小时字段中使用“10-12”，则表示从10到12点，即10,11,12；

●逗号(,)：表达一个列表值，如在星期字段中使用“MON,WED,FRI”，则表示星期一，星期三和星期五；

●斜杠(/)：x/y表达一个等步长序列，x为起始值，y为增量步长值。如在分钟字段中使用0/15，则表示为0,15,30和45秒，而5/15在分钟字段中表示5,20,35,50，你也可以使用\*/y，它等同于0/y；

●L：该字符只在日期和星期字段中使用，代表“Last”的意思，但它在两个字段中意思不同。L在日期字段中，表示这个月份的最后一天，如一月的31号，非闰年二月的28号；如果L用在星期中，则表示星期六，等同于7。但是，如果L出现在星期字段里，而且在前面有一个数值X，则表示“这个月的最后X天”，例如，6L表示该月的最后星期五；

●W：该字符只能出现在日期字段里，是对前导日期的修饰，表示离该日期最近的工作日。例如15W表示离该月15号最近的工作日，如果该月15号是星期六，则匹配14号星期五；如果15日是星期日，则匹配16号星期一；如果15号是星期二，那结果就是15号星期二。但必须注意关联的匹配日期不能够跨月，如你指定1W，如果1号是星期六，结果匹配的是3号星期一，而非上个月最后的那天。W字符串只能指定单一日期，而不能指定日期范围；

●LW组合：在日期字段可以组合使用LW，它的意思是当月的最后一个工作日；

●井号(#)：该字符只能在星期字段中使用，表示当月某个工作日。如6#3表示当月的第三个星期五(6表示星期五，#3表示当前的第三个)，而4#5表示当月的第五个星期三，假设当月没有第五个星期三，忽略不触发；

● C：该字符只在日期和星期字段中使用，代表“Calendar”的意思。它的意思是计划所关联的日期，如果日期没有被关联，则相当于日历中所有日期。例如5C在日期字段中就相当于日历5日以后的第一天。1C在星期字段中相当于星期日后的第一天。

Cron表达式对特殊字符的大小写不敏感，对代表星期的缩写英文大小写也不敏感。

表2下面给出一些完整的Cron表示式的实例：

表2 Cron表示式示例

|  |  |
| --- | --- |
| 表示式 | 说明 |
| "0 0 12 \* \* ? " | 每天12点运行 |
| "0 15 10 ? \* \*" | 每天10:15运行 |
| "0 15 10 \* \* ?" | 每天10:15运行 |
| "0 15 10 \* \* ? \*" | 每天10:15运行 |
| "0 15 10 \* \* ? 2008" | 在2008年的每天10：15运行 |
| "0 \* 14 \* \* ?" | 每天14点到15点之间每分钟运行一次，开始于14:00，结束于14:59。 |
| "0 0/5 14 \* \* ?" | 每天14点到15点每5分钟运行一次，开始于14:00，结束于14:55。 |
| "0 0/5 14,18 \* \* ?" | 每天14点到15点每5分钟运行一次，此外每天18点到19点每5钟也运行一次。 |
| "0 0-5 14 \* \* ?" | 每天14:00点到14:05，每分钟运行一次。 |
| "0 10,44 14 ? 3 WED" | 3月每周三的14:10分到14:44，每分钟运行一次。 |
| "0 15 10 ? \* MON-FRI" | 每周一，二，三，四，五的10:15分运行。 |
| "0 15 10 15 \* ?" | 每月15日10:15分运行。 |
| "0 15 10 L \* ?" | 每月最后一天10:15分运行。 |
| "0 15 10 ? \* 6L" | 每月最后一个星期五10:15分运行。 |
| "0 15 10 ? \* 6L 2007-2009" | 在2007,2008,2009年每个月的最后一个星期五的10:15分运行。 |
| "0 15 10 ? \* 6#3" | 每月第三个星期五的10:15分运行。 |

**CronTrigger实例**

下面，我们使用CronTrigger对SimpleJob进行调度，通过Cron表达式制定调度规则，让它每5秒钟运行一次：

代码清单3 CronTriggerRunner：使用CronTrigger进行调度

package com.baobaotao.basic.quartz;

import org.quartz.CronExpression;

import org.quartz.CronTrigger;

import org.quartz.JobDetail;

import org.quartz.Scheduler;

import org.quartz.SchedulerFactory;

import org.quartz.impl.StdSchedulerFactory;

public class CronTriggerRunner {

public static void main(String args[]) {

try {

JobDetail jobDetail = new JobDetail("job1\_2", "jGroup1",SimpleJob.class);

①-1：创建CronTrigger，指定组及名称

CronTrigger cronTrigger = new CronTrigger("trigger1\_2", "tgroup1");

CronExpression cexp = new CronExpression("0/5 \* \* \* \* ?");**①-2：定义Cron表达式**

cronTrigger.setCronExpression(cexp);**①-3：设置Cron表达式**

SchedulerFactory schedulerFactory = new StdSchedulerFactory();

Scheduler scheduler = schedulerFactory.getScheduler();

scheduler.scheduleJob(jobDetail, cronTrigger);

scheduler.start();

//**②**

} catch (Exception e) {

e.printStackTrace();

}

}

}

运行CronTriggerRunner，每5秒钟将触发运行SimpleJob一次。默认情况下Cron表达式对应当前的时区，可以通过CronTriggerRunner的setTimeZone(java.util.TimeZone timeZone)方法显式指定时区。你还也可以通过setStartTime(java.util.Date startTime)和setEndTime(java.util.Date endTime)指定开始和结束的时间。

在代码清单3的②处需要通过Thread.currentThread.sleep()的方式让主线程睡眠，以便调度器可以继续工作执行任务调度。否则在调度器启动后，因为主线程马上退出，也将同时引起调度器关闭，调度器中的任务都将相应销毁，这将导致看不到实际的运行效果。在单元测试的时候，让主线程睡眠经常使用的办法。对于某些长周期任务调度的测试，你可以简单地调整操作系统时间进行模拟。

1. **使用Calendar**

在实际任务调度中，我们不可能一成不变地按照某个周期性的调度规则运行任务，必须考虑到实现生活中日历上特定日期，就象习惯了大男人作风的人在2月14号也会有不同表现一样。

下面，我们安排一个任务，每小时运行一次，并将五一节和国际节排除在外，其代码如代码清单4所示：

代码清单4 CalendarExample：使用Calendar

package com.baobaotao.basic.quartz;

import java.util.Calendar;

import java.util.Date;

import java.util.GregorianCalendar;

import org.quartz.impl.calendar.AnnualCalendar;

import org.quartz.TriggerUtils;

…

public class CalendarExample {

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

SchedulerFactory sf = new StdSchedulerFactory();

Scheduler scheduler = sf.getScheduler();

①法定节日是以每年为周期的，所以使用AnnualCalendar

AnnualCalendar holidays = new AnnualCalendar();

**②五一劳动节**

Calendar laborDay = new GregorianCalendar();

laborDay.add(Calendar.MONTH,5);

laborDay.add(Calendar.DATE,1);

holidays.setDayExcluded(laborDay, true); **②-1：排除的日期，如果设置为false则为包含**

**③国庆节**

Calendar nationalDay = new GregorianCalendar();

nationalDay.add(Calendar.MONTH,10);

nationalDay.add(Calendar.DATE,1);

holidays.setDayExcluded(nationalDay, true);**③-1：排除该日期**

scheduler.addCalendar("holidays", holidays, false, false);**④向Scheduler注册日历**

Date runDate = TriggerUtils.getDateOf(0,0, 10, 1, 4);**⑤4月1号 上午10点**

JobDetail job = new JobDetail("job1", "group1", SimpleJob.class);

SimpleTrigger trigger = new SimpleTrigger("trigger1", "group1",

runDate,

null,

SimpleTrigger.REPEAT\_INDEFINITELY,

60L \* 60L \* 1000L);

trigger.setCalendarName("holidays");**⑥让Trigger应用指定的日历规则**

scheduler.scheduleJob(job, trigger);

scheduler.start();

//实际应用中主线程不能停止，否则Scheduler得不到执行，此处从略

}

}

由于节日是每年重复的，所以使用org.quartz.Calendar的AnnualCalendar实现类，通过②、③的代码，指定五一和国庆两个节日并通过AnnualCalendar#setDayExcluded(Calendar day, boolean exclude)方法添加这两个日期。exclude为true时表示排除指定的日期，如果为false时表示包含指定的日期。

在定制好org.quartz.Calendar后，还需要通过Scheduler#addCalendar(String calName, Calendar calendar, boolean replace, boolean updateTriggers)进行注册，如果updateTriggers为true，Scheduler中已引用Calendar的Trigger将得到更新，如④所示。

在⑥处，我们让一个Trigger指定使用Scheduler中代表节日的Calendar，这样Trigger就会避开五一和国庆这两个特殊日子了。

# 任务调度信息存储

在默认情况下Quartz将任务调度的运行信息保存在内存中，这种方法提供了最佳的性能，因为内存中数据访问最快。不足之处是缺乏数据的持久性，当程序路途停止或系统崩溃时，所有运行的信息都会丢失。

比如我们希望安排一个执行100次的任务，如果执行到50次时系统崩溃了，系统重启时任务的执行计数器将从0开始。在大多数实际的应用中，我们往往并不需要保存任务调度的现场数据，因为很少需要规划一个指定执行次数的任务。

对于仅执行一次的任务来说，其执行条件信息本身应该是已经持久化的业务数据（如锁定到期解锁任务，解锁的时间应该是业务数据），当执行完成后，条件信息也会相应改变。当然调度现场信息不仅仅是记录运行次数，还包括调度规则、JobDataMap中的数据等等。

如果确实需要持久化任务调度信息，Quartz允许你通过调整其属性文件，将这些信息保存到数据库中。使用数据库保存任务调度信息后，即使系统崩溃后重新启动，任务的调度信息将得到恢复。如前面所说的例子，执行50次崩溃后重新运行，计数器将从51开始计数。使用了数据库保存信息的任务称为持久化任务。

1. **通过配置文件调整任务调度信息的保存策略**

其实Quartz JAR文件的org.quartz包下就包含了一个quartz.properties属性配置文件并提供了默认设置。如果需要调整默认配置，可以在类路径下建立一个新的quartz.properties，它将自动被Quartz加载并覆盖默认的设置。

先来了解一下Quartz的默认属性配置文件：

代码清单5 quartz.properties：默认配置

①集群的配置，这里不使用集群

org.quartz.scheduler.instanceName = DefaultQuartzScheduler

org.quartz.scheduler.rmi.export = false

org.quartz.scheduler.rmi.proxy = false

org.quartz.scheduler.wrapJobExecutionInUserTransaction = false

②配置调度器的线程池

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

org.quartz.threadPool.threadCount = 10

org.quartz.threadPool.threadPriority = 5

org.quartz.threadPool.threadsInheritContextClassLoaderOfInitializingThread = true

③配置任务调度现场数据保存机制

org.quartz.jobStore.class = **org.quartz.simpl.RAMJobStore**

Quartz的属性配置文件主要包括三方面的信息：

1)集群信息；

2)调度器线程池；

3)任务调度现场数据的保存。

如果任务数目很大时，可以通过增大线程池的大小得到更好的性能。默认情况下，Quartz采用org.quartz.simpl.RAMJobStore保存任务的现场数据，顾名思义，信息保存在RAM内存中，我们可以通过以下设置将任务调度现场数据保存到数据库中：

代码清单6 quartz.properties：使用数据库保存任务调度现场数据

…

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

org.quartz.jobStore.tablePrefix = QRTZ\_**①数据表前缀**

org.quartz.jobStore.dataSource = qzDS**②数据源名称**

③定义数据源的具体属性

org.quartz.dataSource.qzDS.driver = oracle.jdbc.driver.OracleDriver

org.quartz.dataSource.qzDS.URL = jdbc:oracle:thin:@localhost:1521:ora9i

org.quartz.dataSource.qzDS.user = stamen

org.quartz.dataSource.qzDS.password = abc

org.quartz.dataSource.qzDS.maxConnections = 10

要将任务调度数据保存到数据库中，就必须使用org.quartz.impl.jdbcjobstore.JobStoreTX代替原来的org.quartz.simpl.RAMJobStore并提供相应的数据库配置信息。首先①处指定了Quartz数据库表的前缀，在②处定义了一个数据源，在③处具体定义这个数据源的连接信息。

你必须事先在相应的数据库中创建Quartz的数据表（共8张），在Quartz的完整发布包的docs/dbTables目录下拥有对应不同数据库的SQL脚本。

1. **查询数据库中的运行信息**

任务的现场保存对于上层的Quartz程序来说是完全透明的，我们在src目录下编写一个如代码清单6所示的quartz.properties文件后，重新运行代码清单2或代码清单3的程序，在数据库表中将可以看到对应的持久化信息。当调度程序运行过程中途停止后，任务调度的现场数据将记录在数据表中，在系统重启时就可以在此基础上继续进行任务的调度。

代码清单7 JDBCJobStoreRunner：从数据库中恢复任务的调度

package com.baobaotao.basic.quartz;

import org.quartz.Scheduler;

import org.quartz.SchedulerFactory;

import org.quartz.SimpleTrigger;

import org.quartz.Trigger;

import org.quartz.impl.StdSchedulerFactory;

public class JDBCJobStoreRunner {

public static void main(String args[]) {

try {

SchedulerFactory schedulerFactory = new StdSchedulerFactory();

Scheduler scheduler = schedulerFactory.getScheduler();

①获取调度器中所有的触发器组

String[] triggerGroups = scheduler.getTriggerGroupNames();

②重新恢复在tgroup1组中，名为trigger1\_1触发器的运行

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

String[] triggers = scheduler.getTriggerNames(triggerGroups[i]);

for (int j = 0; j < triggers.length; j++) {

Trigger tg = scheduler.getTrigger(triggers[j],triggerGroups[i]);

if (tg instanceof SimpleTrigger

&& tg.getFullName().equals("tgroup1.trigger1\_1")) {**②-1:根据名称判断**

**②-1:恢复运行**

scheduler.rescheduleJob(triggers[j], triggerGroups[i],tg);

}

}

}

scheduler.start();

} catch (Exception e) {

e.printStackTrace();

}

}

}

当代码清单2中的SimpleTriggerRunner执行到一段时间后非正常退出，我们就可以通过这个JDBCJobStoreRunner根据记录在数据库中的现场数据恢复任务的调度。Scheduler中的所有Trigger以及JobDetail的运行信息都会保存在数据库中，这里我们仅恢复tgroup1组中名称为trigger1\_1的触发器，这可以通过如②-1所示的代码进行过滤，触发器的采用GROUP.TRIGGER\_NAME的全名格式。通过Scheduler#rescheduleJob(String triggerName,String groupName,Trigger newTrigger)即可重新调度关联某个Trigger的任务。

下面我们来观察一下不同时期qrtz\_simple\_triggers表的数据：

1．运行代码清单2的SimpleTriggerRunner一小段时间后退出：

Quartz任务调度快速入门

REPEAT\_COUNT表示需要运行的总次数，而TIMES\_TRIGGER表示已经运行的次数。

2．运行代码清单7的JDBCJobStoreRunner恢复trigger1\_1的触发器，运行一段时间后退出，这时qrtz\_simple\_triggers中的数据如下：

Quartz任务调度快速入门

首先Quartz会将原REPEAT\_COUNT-TIMES\_TRIGGER得到新的REPEAT\_COUNT值，并记录已经运行的次数（重新从0开始计算）。

3．重新启动JDBCJobStoreRunner运行后，数据又将发生相应的变化：

Quartz任务调度快速入门

4．继续运行直至完成所有剩余的次数，再次查询qrtz\_simple\_triggers表：

Quartz任务调度快速入门

这时，该表中的记录已经变空。

值得注意的是，如果你使用JDBC保存任务调度数据时，当你运行代码清单2的SimpleTriggerRunner然后退出，当再次希望运行SimpleTriggerRunner时，系统将抛出JobDetail重名的异常：

Unable to store Job with name: 'job1\_1' and group: 'jGroup1', because one already exists with this identification.

因为每次调用Scheduler#scheduleJob()时，Quartz都会将JobDetail和Trigger的信息保存到数据库中，如果数据表中已经同名的JobDetail或Trigger，异常就产生了。

# 小结

Quartz提供了最为丰富的任务调度功能，不但可以制定周期性运行的任务调度方案，还可以让你按照日历相关的方式进行任务调度。Quartz框架的重要组件包括Job、JobDetail、Trigger、Scheduler以及辅助性的JobDataMap和SchedulerContext。

Quartz拥有一个线程池，通过线程池为任务提供执行线程，你可以通过配置文件对线程池进行参数定制。Quartz的另一个重要功能是可将任务调度信息持久化到数据库中，以便系统重启时能够恢复已经安排的任务。此外，Quartz还拥有完善的事件体系，允许你注册各种事件的监听器。

# 附录1

#### 

Quartz表结构图