# 1、初识Quartz

Quartz可以用来做什么？

Quartz是一个任务调度框架。比如你遇到这样的问题

想每月25号，信用卡自动还款

想每年4月1日自己给当年暗恋女神发一封匿名贺卡

想每隔1小时，备份一下自己的爱情片 学习笔记到云盘

这些问题总结起来就是：在某一个有规律的时间点干某件事。并且时间的触发的条件可以非常复杂（比如每月最后一个工作日的17:50），复杂到需要一个专门的框架来干这个事。 Quartz就是来干这样的事，你给它一个触发条件的定义，它负责到了时间点，触发相应的Job起来干活。

体验quartz:

<dependencies>

<dependency>

<groupId>org.quartz-scheduler</groupId>

<artifactId>quartz</artifactId>

<version>2.2.1</version>

</dependency>

</dependencies>

import static org.quartz.DateBuilder.newDate;

import static org.quartz.JobBuilder.newJob;

import static org.quartz.SimpleScheduleBuilder.simpleSchedule;

import static org.quartz.TriggerBuilder.newTrigger;

import java.util.GregorianCalendar;

import org.quartz.JobDetail;

import org.quartz.Scheduler;

import org.quartz.Trigger;

import org.quartz.impl.StdSchedulerFactory;

import org.quartz.impl.calendar.AnnualCalendar;

public class QuartzTest {

public static void main(String[] args) {

try {

//创建scheduler

Scheduler scheduler = StdSchedulerFactory.getDefaultScheduler();

//定义一个Trigger

Trigger trigger = newTrigger().withIdentity("trigger1", "group1") //定义name/group

.startNow()//一旦加入scheduler，立即生效

.withSchedule(simpleSchedule() //使用SimpleTrigger

.withIntervalInSeconds(1) //每隔一秒执行一次

.repeatForever()) //一直执行，奔腾到老不停歇

.build();

//定义一个JobDetail

JobDetail job = newJob(HelloQuartz.class) //定义Job类为HelloQuartz类，这是真正的执行逻辑所在

.withIdentity("job1", "group1") //定义name/group

.usingJobData("name", "quartz") //定义属性

.build();

//加入这个调度

scheduler.scheduleJob(job, trigger);

//启动之

scheduler.start();

//运行一段时间后关闭

Thread.sleep(10000);

scheduler.shutdown(true);

} catch (Exception e) {

e.printStackTrace();

}

}

}

import java.util.Date;

import org.quartz.DisallowConcurrentExecution;

import org.quartz.Job;

import org.quartz.JobDetail;

import org.quartz.JobExecutionContext;

import org.quartz.JobExecutionException;

public class HelloQuartz implements Job {

public void execute(JobExecutionContext context) throws JobExecutionException {

JobDetail detail = context.getJobDetail();

String name = detail.getJobDataMap().getString("name");

System.out.println("say hello to " + name + " at " + new Date());

}

}

# 2、Trigger讲解

## 01、SimpleTrigger

import org.quartz.\*;

import java.text.SimpleDateFormat;

import java.util.Date;

public class SimpleTriggerMain {

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

// 获取一个调度工厂

SchedulerFactory schedFact = new org.quartz.impl.StdSchedulerFactory();

// 获取一个调度器

Scheduler sched = schedFact.getScheduler();

SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd hh:mm:ss");

JobDetail job = JobBuilder.newJob(HelloJob.class).withIdentity("job1", "group1").build();

// 在当前时间15秒后运行

Date startTime = DateBuilder.nextGivenSecondDate(new Date( ),15);

// 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.当前时间15秒后运行，每10秒运行一次，共运行5次

SimpleTrigger trigger = (SimpleTrigger) TriggerBuilder.newTrigger().withIdentity("trigger1", "group1")

.startAt(startTime).withSchedule(SimpleScheduleBuilder.simpleSchedule()

.withIntervalInSeconds(10)

.withRepeatCount(5)

)

.build();

sched.scheduleJob(job, trigger);

// 调度启动

sched.start();

}

}

## 02、CronTriger-Cron触发器

Crontab讲解

1、contab

linux 系统由 cron (crond) 这个系统服务来控制的，crond 是linux下用来周期性的执行某种任务或等待处理某些事件的一个守护进程，Linux 系统也提供了使用者控制计划任务的命令 :crontab 命令。

crontab文件的含义：

每一行都代表一项任务，每行的每个字段代表一项设置，它的格式共分为六个字段，前五段是时间设定段，第六段是要执行的命令段，格式如下：

minute hour day month week command

其中：

minute： 表示分钟，可以是从0到59之间的任何整数。

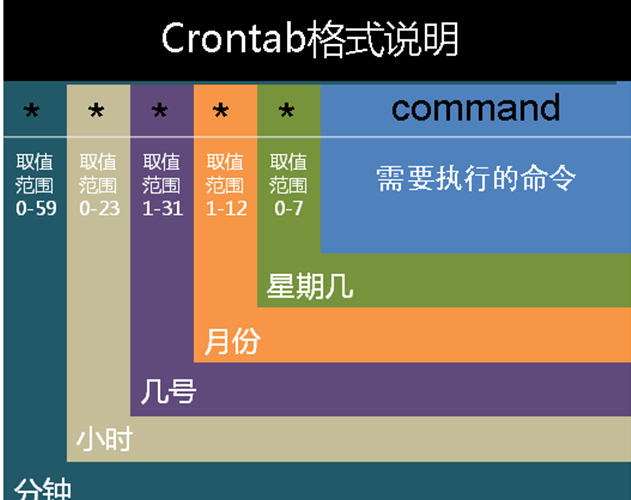
hour：表示小时，可以是从0到23之间的任何整数。

day：表示日期，可以是从1到31之间的任何整数。

month：表示月份，可以是从1到12之间的任何整数。

week：表示星期几，可以是从0到7之间的任何整数，这里的0或7代表星期日。

command：要执行的命令，可以是系统命令，也可以是自己编写的脚本文件。



在以上各个字段中，还可以使用以下特殊字符：

星号（\*）：代表所有可能的值，例如month字段如果是星号，则表示在满足其它字段的制约条件后每月都执行该命令操作。

逗号（,）：可以用逗号隔开的值指定一个列表范围，例如，“1,2,5,7,8,9”

中杠（-）：可以用整数之间的中杠表示一个整数范围，例如“2-6”表示“2,3,4,5,6”

正斜线（/）：可以用正斜线指定时间的间隔频率，例如“0-15/2”表示每两小时执行一次。同时正斜线可以和星号一起使用，例如\*/10，如果用在minute字段，表示每十分钟执行一次。

2、使用命令：

crontab -l

crontab -e

3、查找错误

查看下已经设置好的计划任务

cat /etc/crontab

重启下计划任务服务

/etc/init.d/crond restart

上面一步如果OK就不用这步

查看下日志找到错误原因

cat /var/log/cron

使用实例

实例1：每1分钟执行一次command

命令：

\* \* \* \* \* command

实例2：每小时的第3和第15分钟执行

命令：

3,15 \* \* \* \* command

实例3：在上午8点到11点的第3和第15分钟执行

命令：

3,15 8-11 \* \* \* command

实例4：每隔两天的上午8点到11点的第3和第15分钟执行

命令：

3,15 8-11 \*/2 \* \* command

实例5：每个星期一的上午8点到11点的第3和第15分钟执行

命令：

3,15 8-11 \* \* 1 command

实例6：每晚的21:30重启smb

命令：

30 21 \* \* \* /etc/init.d/smb restart

实例7：每月1、10、22日的4 : 45重启smb

命令：

45 4 1,10,22 \* \* /etc/init.d/smb restart

实例8：每周六、周日的1 : 10重启smb

命令：

10 1 \* \* 6,0 /etc/init.d/smb restart

实例9：每天18 : 00至23 : 00之间每隔30分钟重启smb

命令：

0,30 18-23 \* \* \* /etc/init.d/smb restart

实例10：每星期六的晚上11 : 00 pm重启smb

命令：

0 23 \* \* 6 /etc/init.d/smb restart

实例11：每一小时重启smb

命令：

\* \*/1 \* \* \* /etc/init.d/smb restart

实例12：晚上11点到早上7点之间，每隔一小时重启smb

命令：

\* 23-7/1 \* \* \* /etc/init.d/smb restart

实例13：每月的4号与每周一到周三的11点重启smb

命令：

0 11 4 \* mon-wed /etc/init.d/smb restart

实例14：一月一号的4点重启smb

命令：

0 4 1 jan \* /etc/init.d/smb restart

实例15：每小时执行/etc/cron.hourly目录内的脚本

命令：

01 \* \* \* \* root run-parts /etc/cron.hourly

import org.quartz.\*;

import java.text.SimpleDateFormat;

public class CronTriggerMain {

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

SchedulerFactory schedFact = new org.quartz.impl.StdSchedulerFactory();

// 获取一个调度器

Scheduler sched = schedFact.getScheduler();

JobDetail job = JobBuilder.newJob(HelloJob.class).withIdentity("job1", "group1").build();

// 每两秒执行

CronTrigger trigger = TriggerBuilder.newTrigger().withIdentity("trigger1", "group1").withSchedule(

CronScheduleBuilder.cronSchedule("/2 \* \* \* \* ?")

).build();

sched.scheduleJob(job, trigger);

// 调度启动

sched.start();

}

}

适合于更复杂的任务，它支持类型于Linux Cron的语法（并且更强大）。基本上它覆盖了以上三个Trigger的绝大部分能力（但不是全部）—— 当然，也更难理解。

它适合的任务类似于：每天0:00,9:00,18:00各执行一次。

它的属性只有:

* Cron表达式。但这个表示式本身就够复杂了。下面会有说明。

例子：

cronSchedule**(**"0 0/2 8-17 \* \* ?"**)** *// 每天8:00-17:00，每隔2分钟执行一次*

**.**build**();**

cronSchedule**(**"0 30 9 ? \* MON"**)** *// 每周一，9:30执行一次*

**.**build**();**

weeklyOnDayAndHourAndMinute**(**MONDAY**,**9**,** 30**)** *//等同于 0 30 9 ? \* MON*

**.**build**();**

#### Cron表达式

| **位置** | **时间域** | **允许值** | **特殊值** |
| --- | --- | --- | --- |
| 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 | 年份（可选） | 1-31 | , - \* / |

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。

一些例子：

| **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。 |

## 03、DailyTimeIntervalTrigger-日期触发

import org.quartz.\*;

public class DailyTimeIntervalTriggerMain {

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

SchedulerFactory schedFact = new org.quartz.impl.StdSchedulerFactory();

// 获取一个调度器

Scheduler sched = schedFact.getScheduler();

JobDetail job = JobBuilder.newJob(HelloJob.class).withIdentity("job1", "group1").build();

// 每两秒执行

DailyTimeIntervalTrigger trigger = TriggerBuilder.newTrigger().withIdentity("trigger1", "group1").withSchedule(

DailyTimeIntervalScheduleBuilder.dailyTimeIntervalSchedule().withInterval(2, DateBuilder.IntervalUnit.SECOND)

).build();

sched.scheduleJob(job, trigger);

// 调度启动

sched.start();

}

}

指定每天的某个时间段内，以一定的时间间隔执行任务。并且它可以支持指定星期。

它适合的任务类似于：指定每天9:00 至 18:00 ，每隔70秒执行一次，并且只要周一至周五执行。

它的属性有:

* startTimeOfDay 每天开始时间
* endTimeOfDay 每天结束时间
* daysOfWeek 需要执行的星期
* interval 执行间隔
* intervalUnit 执行间隔的单位（秒，分钟，小时，天，月，年，星期）
* repeatCount 重复次数

例子:

dailyTimeIntervalSchedule**()**

**.**startingDailyAt**(**TimeOfDay**.**hourAndMinuteOfDay**(**9**,** 0**))** *//第天9：00开始*

**.**endingDailyAt**(**TimeOfDay**.**hourAndMinuteOfDay**(**16**,** 0**))** *//16：00 结束*

**.**onDaysOfTheWeek**(**MONDAY**,**TUESDAY**,**WEDNESDAY**,**THURSDAY**,**FRIDAY**)** *//周一至周五执行*

**.**withIntervalInHours**(**1**)** *//每间隔1小时执行一次*

**.**withRepeatCount**(**100**)** *//最多重复100次（实际执行100+1次）*

**.**build**();**

dailyTimeIntervalSchedule**()**

**.**startingDailyAt**(**TimeOfDay**.**hourAndMinuteOfDay**(**9**,** 0**))** *//第天9：00开始*

**.**endingDailyAfterCount**(**10**)** *//每天执行10次，这个方法实际上根据 startTimeOfDay+interval\*count 算出 endTimeOfDay*

**.**onDaysOfTheWeek**(**MONDAY**,**TUESDAY**,**WEDNESDAY**,**THURSDAY**,**FRIDAY**)** *//周一至周五执行*

**.**withIntervalInHours**(**1**)** *//每间隔1小时执行一次*

**.**build**();**

## 04、CalendarIntervalTrigger-日历触发

import org.quartz.\*;

import org.quartz.impl.StdSchedulerFactory;

public class CalendarIntervalMain {

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

SchedulerFactory schedFact = new StdSchedulerFactory();

// 获取一个调度器

Scheduler sched = schedFact.getScheduler();

JobDetail job = JobBuilder.newJob(HelloJob.class).withIdentity("job1", "group1").build();

// 每两秒执行

CalendarIntervalTrigger trigger = TriggerBuilder.newTrigger().withIdentity("trigger1", "group1").withSchedule(

CalendarIntervalScheduleBuilder.calendarIntervalSchedule().withInterval(2, DateBuilder.IntervalUnit.SECOND)

).build();

sched.scheduleJob(job, trigger);

// 调度启动

sched.start();

}

}

类似于SimpleTrigger，指定从某一个时间开始，以一定的时间间隔执行的任务。 但是不同的是SimpleTrigger指定的时间间隔为**毫秒**，没办法指定每隔一个月执行一次（每月的时间间隔不是固定值），而CalendarIntervalTrigger支持的间隔单位有**秒，分钟，小时，天，月，年，星期**。

相较于SimpleTrigger有两个优势：1、更方便，比如每隔1小时执行，你不用自己去计算1小时等于多少毫秒。 2、支持不是固定长度的间隔，比如间隔为月和年。但劣势是精度只能到秒。

它适合的任务类似于：9:00 开始执行，并且以后每周 9:00 执行一次

它的属性有:

* interval 执行间隔
* intervalUnit 执行间隔的单位（秒，分钟，小时，天，月，年，星期）

例子:

calendarIntervalSchedule**()**

**.**withIntervalInDays**(**1**)** *//每天执行一次*

**.**build**();**

calendarIntervalSchedule**()**

**.**withIntervalInWeeks**(**1**)** *//每周执行一次*

**.**build**();**

# 3、job和jobDetail讲解

JobDetail & Job

JobDetail是任务的定义，而Job是任务的执行逻辑。在JobDetail里会引用一个Job Class定义。一个最简单的例子

public class JobTest {

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

JobDetail job=newJob()

.ofType(DoNothingJob.class) //引用Job Class

.withIdentity("job1", "group1") //设置name/group

.withDescription("this is a test job") //设置描述

.usingJobData("age", 18) //加入属性到ageJobDataMap

.build();

job.getJobDataMap().put("name", "quertz"); //加入属性name到JobDataMap

//定义一个每秒执行一次的SimpleTrigger

Trigger trigger=newTrigger()

.startNow()

.withIdentity("trigger1")

.withSchedule(simpleSchedule()

.withIntervalInSeconds(1)

.repeatForever())

.build();

Scheduler sche=StdSchedulerFactory.getDefaultScheduler();

sche.scheduleJob(job, trigger);

sche.start();

System.in.read();

sche.shutdown();

}

}

public class DoNothingJob implements Job {

public void execute(JobExecutionContext context) throws JobExecutionException {

System.out.println("do nothing");

}

}

从上例我们可以看出，要定义一个任务，需要干几件事：

创建一个org.quartz.Job的实现类，并实现实现自己的业务逻辑。比如上面的DoNothingJob。

定义一个JobDetail，引用这个实现类

加入scheduleJob

Quartz调度一次任务，会干如下的事：

JobClass jobClass=JobDetail.getJobClass()

Job jobInstance=jobClass.newInstance()。所以Job实现类，必须有一个public的无参构建方法。

jobInstance.execute(JobExecutionContext context)。JobExecutionContext是Job运行的上下文，可以获得Trigger、Scheduler、JobDetail的信息。

也就是说，每次调度都会创建一个新的Job实例，这样的好处是有些任务并发执行的时候，不存在对临界资源的访问问题——当然，如果需要共享JobDataMap的时候，还是存在临界资源的并发访问的问题。

JobDataMap

Job都次都是newInstance的实例，那我怎么传值给它？ 比如我现在有两个发送邮件的任务，一个是发给"liLei",一个发给"hanmeimei",不能说我要写两个Job实现类LiLeiSendEmailJob和HanMeiMeiSendEmailJob。实现的办法是通过JobDataMap。

每一个JobDetail都会有一个JobDataMap。JobDataMap本质就是一个Map的扩展类，只是提供了一些更便捷的方法，比如getString()之类的。

我们可以在定义JobDetail，加入属性值，方式有二：

newJob().usingJobData("age", 18) //加入属性到ageJobDataMap

or

job.getJobDataMap().put("name", "quertz"); //加入属性name到JobDataMap

然后在Job中可以获取这个JobDataMap的值，方式同样有二：

public class HelloQuartz implements Job {

private String name;

public void execute(JobExecutionContext context) throws JobExecutionException {

JobDetail detail = context.getJobDetail();

JobDataMap map = detail.getJobDataMap(); //方法一：获得JobDataMap

System.out.println("say hello to " + name + "[" + map.getInt("age") + "]" + " at "

+ new Date());

}

//方法二：属性的setter方法，会将JobDataMap的属性自动注入

public void setName(String name) {

this.name = name;

}

}

对于同一个JobDetail实例，执行的多个Job实例，是共享同样的JobDataMap，也就是说，如果你在任务里修改了里面的值，会对其他Job实例（并发的或者后续的）造成影响。

除了JobDetail，Trigger同样有一个JobDataMap，共享范围是所有使用这个Trigger的Job实例。

Job并发

Job是有可能并发执行的，比如一个任务要执行10秒中，而调度算法是每秒中触发1次，那么就有可能多个任务被并发执行。

有时候我们并不想任务并发执行，比如这个任务要去”获得数据库中所有未发送邮件的名单“，如果是并发执行，就需要一个数据库锁去避免一个数据被多次处理。这个时候一个@DisallowConcurrentExecution解决这个问题。

就是这样

public class DoNothingJob implements Job {

@DisallowConcurrentExecution

public void execute(JobExecutionContext context) throws JobExecutionException {

System.out.println("do nothing");

}

}

@PersistJobDataAfterExecution： 将这一次Job执行的结果保存在JobDetail中，下一次Job执行的时候可以访问。

注意，@DisallowConcurrentExecution是对JobDetail实例生效，也就是如果你定义两个JobDetail，引用同一个Job类，是可以并发执行的。

JobExecutionException

Job.execute()方法是不允许抛出除JobExecutionException之外的所有异常的（包括RuntimeException)，所以编码的时候，最好是try-catch住所有的Throwable，小心处理。

例子：

1. **public** **class** DumbJob **implements** Job {
3. **public** DumbJob() {
4. }
6. **public** **void** execute(JobExecutionContext context)
7. **throws** JobExecutionException
8. {
9. JobKey key = context.getJobDetail().getKey();
11. JobDataMap dataMap = context.getJobDetail().getJobDataMap();
13. String jobSays = dataMap.getString("jobSays");
14. **float** myFloatValue = dataMap.getFloat("myFloatValue");
16. System.err.println("Instance " + key + " of DumbJob says: " + jobSays + ", and val is: " + myFloatValue);
17. }
18. }
19. **public** **class** DumbJob **implements** Job {
21. **public** DumbJob() {
22. }
24. **public** **void** execute(JobExecutionContext context)
25. **throws** JobExecutionException
26. {
27. JobKey key = context.getJobDetail().getKey();
29. JobDataMap dataMap = context.getMergedJobDataMap();  // Note the difference from the previous example
31. String jobSays = dataMap.getString("jobSays");
32. **float** myFloatValue = dataMap.getFloat("myFloatValue");
33. ArrayList state = (ArrayList)dataMap.get("myStateData");
34. state.add(**new** Date());
36. System.err.println("Instance " + key + " of DumbJob says: " + jobSays + ", and val is: " + myFloatValue);
37. }
38. }
39. **public** **class** DumbJob **implements** Job {

42. String jobSays;
43. **float** myFloatValue;
44. ArrayList state;
46. **public** DumbJob() {
47. }
49. **public** **void** execute(JobExecutionContext context)
50. **throws** JobExecutionException
51. {
52. JobKey key = context.getJobDetail().getKey();
54. JobDataMap dataMap = context.getMergedJobDataMap();  // Note the difference from the previous example
56. state.add(**new** Date());
58. System.err.println("Instance " + key + " of DumbJob says: " + jobSays + ", and val is: " + myFloatValue);
59. }
61. **public** **void** setJobSays(String jobSays) {
62. **this**.jobSays = jobSays;
63. }
65. **public** **void** setMyFloatValue(**float** myFloatValue) {
66. myFloatValue = myFloatValue;
67. }
69. **public** **void** setState(ArrayList state) {
70. state = state;
71. }
73. }

其他属性

Durability(耐久性？)

如果一个任务不是durable，那么当没有Trigger关联它的时候，它就会被自动删除。

RequestsRecovery

如果一个任务是"requests recovery"，那么当任务运行过程非正常退出时（比如进程崩溃，机器断电，但不包括抛出异常这种情况），Quartz再次启动时，会重新运行一次这个任务实例。

可以通过JobExecutionContext.isRecovering()查询任务是否是被恢复的。

<https://blog.csdn.net/chen3749102/article/details/53502710>

# 4、Scheduler讲解

Scheduler

Scheduler就是Quartz的大脑，所有任务都是由它来设施。

Schduelr包含一个两个重要组件: JobStore和ThreadPool。

JobStore是会来存储运行时信息的，包括Trigger,Schduler,JobDetail，业务锁等。它有多种实现RAMJob(内存实现)，JobStoreTX(JDBC，事务由Quartz管理），JobStoreCMT(JDBC，使用容器事务)，ClusteredJobStore(集群实现)。

ThreadPool就是线程池，Quartz有自己的线程池实现。所有任务的都会由线程池执行。

SchedulerFactory

SchdulerFactory，顾名思义就是来用创建Schduler了，有两个实现：DirectSchedulerFactory和 StdSchdulerFactory。前者可以用来在代码里定制你自己的Schduler参数。后者是直接读取classpath下的quartz.properties（不存在就都使用默认值）配置来实例化Schduler。通常来讲，我们使用StdSchdulerFactory也就足够了。

SchdulerFactory本身是支持创建RMI stub的，可以用来管理远程的Scheduler，功能与本地一样，可以远程提交个Job什么的。

DirectSchedulerFactory的创建接口

/\*\*

\* Same as

\* {@link DirectSchedulerFactory#createScheduler(ThreadPool threadPool, JobStore jobStore)},

\* with the addition of specifying the scheduler name and instance ID. This

\* scheduler can only be retrieved via

\* {@link DirectSchedulerFactory#getScheduler(String)}

\*

\* @param schedulerName

\* The name for the scheduler.

\* @param schedulerInstanceId

\* The instance ID for the scheduler.

\* @param threadPool

\* The thread pool for executing jobs

\* @param jobStore

\* The type of job store

\* @throws SchedulerException

\* if initialization failed

\*/

public void createScheduler(String schedulerName,

String schedulerInstanceId, ThreadPool threadPool, JobStore jobStore)

throws SchedulerException;

StdSchdulerFactory的配置例子， 更多配置，参考Quartz配置指南：

org.quartz.scheduler.instanceName = DefaultQuartzScheduler

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

org.quartz.threadPool.threadCount = 10

org.quartz.threadPool.threadPriority = 5

org.quartz.threadPool.threadsInheritContextClassLoaderOfInitializingThread = true

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

# 5、Quartz系统参数配置详解

https://blog.csdn.net/zixiao217/article/details/53091812

我们通常是通过quartz.properties属性配置文件(默认情况下均使用该文件)结合**StdSchedulerFactory**来使用Quartz的。StdSchedulerFactory 会加载属性配置文件并实例化一个Scheduler。

默认情况下，Quartz会加载classpath下的”quartz.properties”文件作为配置属性，如果找不到则会使用quartz框架自己jar下**org/quartz**包底下的”quartz.properties”文件。当然你也可以指定”org.quartz.properties”属性指向你自定义的属性配置文件。或者，你也可以在调用StdSchedulerFactory的 getScheduler()方法之前调用 initialize(xx)初始化factory配置。

**主调度程序Scheduler的配置**

| **参数名** | **是否必须** | **类型** | **默认值** |
| --- | --- | --- | --- |
| org.quartz.scheduler.instanceName | N | string | ‘QuartzScheduler’ |
| org.quartz.scheduler.instanceId | N | string | ‘NON\_CLUSTERED’ |
| org.quartz.scheduler.instanceIdGenerator.class | N | string (class name) | org.quartz.simpl.SimpleInstanceIdGenerator |
| org.quartz.scheduler.threadName | N | string | instanceName+’\_QuartzSchedulerThread’ |
| org.quartz.scheduler.makeSchedulerThreadDaemon | N | boolean | false |
| org.quartz.scheduler.threadsInheritContextClassLoaderOfInitializer | N | boolean | false |
| org.quartz.scheduler.idleWaitTime | N | string | 30000 |
| org.quartz.scheduler.dbFailureRetryInterval | N | long | 15000 |
| org.quartz.scheduler.classLoadHelper.class | N | string (class name) | org.quartz.simpl.CascadingClassLoadHelper |
| org.quartz.scheduler.jobFactory.class | N | string (class name) | org.quartz.simpl.PropertySettingJobFactory |
| org.quartz.context.key.SOME\_KEY | N | string | none |
| org.quartz.scheduler.userTransactionURL | N | string (url) | ‘java:comp/UserTransaction’ |
| org.quartz.scheduler.wrapJobExecutionInUserTransaction | N | boolean | false |
| org.quartz.scheduler.skipUpdateCheck | N | boolean | false |
| org.quartz.scheduler.batchTriggerAcquisitionMaxCount | N | int | 1 |
| org.quartz.scheduler.batchTriggerAcquisitionFireAheadTimeWindow | N | long | 0 |

**org.quartz.scheduler.instanceName**

使用StdSchedulerFactory的 getScheduler()方法创建的scheduler实例名称，在同一个程序中可以根据该名称来区分scheduler。如果实在集群环境中使用，你必须使用同一个名称——集群环境下”逻辑”相同的scheduler。

**org.quartz.scheduler.instanceId**

scheduler实例的标志id，必须是全局唯一的，即使在集群环境中”逻辑”相同的scheduler。 或者可以使用“SYS\_PROP”通过系统属性设置id。

**org.quartz.scheduler.instanceIdGenerator.class**

只有在”org.quartz.scheduler.instanceId”设置为”AUTO”的时候才使用该属性设置。默认情况下，“org.quartz.simpl.SimpleInstanceIdGenerator”是基于instanceId和时间戳来自动生成的。其他的id生成器的实现包括 **SystemPropertyInstanceIdGenerator** 从系统属性获取 “org.quartz.scheduler.instanceId”, 和 **HostnameInstanceIdGenerator** 使用主机名 (InetAddress.getLocalHost().getHostName())。你也可以自定义生成方式哦。

**org.quartz.scheduler.threadName**

指定线程名，如果不指定的话，会自动使用org.quartz.scheduler.instanceName属性值加上后缀字符串”\_QuartzSchedulerThread”.

**org.quartz.scheduler.makeSchedulerThreadDaemon**

指定scheduler的主线程是否为后台线程。

**org.quartz.scheduler.threadsInheritContextClassLoaderOfInitializer**

指定Quartz生成的线程是否继承初始化线程的上下文类加载器。这会影响Quartz的主调度线程、JDBCJobStore的”熄火”处理线程、集群回复线程和线程池里的线程。 将该值设置为“true”可以帮助类加载，JNDI查找，并在应用程序服务器上使用Quartz等相关问题。

**org.quartz.scheduler.idleWaitTime**

在调度程序空闲的时候，重复查询是否有可用触发器的等待时间。通常并不会设置为true，除非你是用XA事务，并且延迟触发会导致问题的场景。 5000ms以下是不推荐的，因为它会导致过的的数据库查询。1000ms以下是非法的。

**org.quartz.scheduler.dbFailureRetryInterval**

连接超时重试连接的间隔。使用 RamJobStore时，该参数并没什么用。

**org.quartz.scheduler.classLoadHelper.class**

默认最可靠的方式就是指定”org.quartz.simpl.CascadingClassLoadHelper”，没必要指定其他类。

**org.quartz.scheduler.jobFactory.class**

指定JobFactory的类(接口)名称。负责实例化jobClass。默认是”org.quartz.simpl.PropertySettingJobFactory”,只是在job被执行的时候简单调用newInstance()实例化一个job类。PropertySettingJobFactory 会使用反射机制通过SchedulerContext、 Job、Trigger和 JobDataMaps设置job bean的属性。

使用JTA事务时，可以设置事务相关的属性

**org.quartz.scheduler.userTransactionURL**

设置Quartz能够加载UserTransaction换利器的JNDI的 URL。默认值是”java:comp/UserTransaction”。Websphere 的用户可能会设置为“jta/usertransaction。只有在Quartz使用JobStoreCMT的时候，才会使用该属性，并且**org.quartz.scheduler.wrapJobExecutionInUserTransaction**也会设置为true。

**org.quartz.scheduler.wrapJobExecutionInUserTransaction**

如果想使用Quartz在执行一个job前使用UserTransaction，则应该设置该属性为true。job执行完、在JobDataMap改变之后事务会提交。默认值是false。 可以在你的job类中使用 @ExecuteInJTATransaction注解, 可以控制job是否使用事务。

**org.quartz.scheduler.skipUpdateCheck**

建议设置为“org.terracotta.quartz.skipUpdateCheck=true”不会在程序运行中还去检查quartz是否有版本更新。

**org.quartz.scheduler.batchTriggerAcquisitionMaxCount**

允许调度程序一次性触发的触发器数量。.默认值是1。值越大一次性触发的任务就可以越多，但是在集群环境下，不建议设置为很大值。如果值 > 1, 并且使用了 JDBC JobStore的话, **org.quartz.jobStore.acquireTriggersWithinLock**属性必须设置为true，以避免”弄脏”数据。

**org.quartz.scheduler.batchTriggerAcquisitionFireAheadTimeWindow**

允许触发器被获取并在其预定的触发时间之前触发的数量。默认值0。

**Quartz线程池的配置**

| **参数名** | **是否必须** | **类型** | **默认值** |
| --- | --- | --- | --- |
| org.quartz.threadPool.class | Y | string (class name) | null |
| org.quartz.threadPool.threadCount | Y | string | -1 |
| org.quartz.threadPool.threadPriority | N | int | Thread.NORM\_PRIORITY (5) |

**org.quartz.threadPool.class**

线程池的名字。可以使用后Quartz 的 “org.quartz.simpl.SimpleThreadPool”。

**org.quartz.threadPool.threadCount**

指定线程数量。一般1-100足以满足你的应用需求了。

**org.quartz.threadPool.threadPriority**

线程优先级，Thread.MIN\_PRIORITY (1) and Thread.MAX\_PRIORITY (10)之间，默认Thread.NORM\_PRIORITY (5)。

**监听器的配置**

全局触发器的监听器配置:

org.quartz.triggerListener.NAME.class = com.foo.MyListenerClass

org.quartz.triggerListener.NAME.propName = propValue

org.quartz.triggerListener.NAME.prop2Name = prop2Value

1

2

3

全局job的监听器配置:

org.quartz.jobListener.NAME.class = com.foo.MyListenerClass

org.quartz.jobListener.NAME.propName = propValue

org.quartz.jobListener.NAME.prop2Name = prop2Value

1

2

3

**jobStore的配置**

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

或者

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

或者

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

1

2

3

4

5

**数据源的配置**

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

org.quartz.dataSource.myDS.URL = jdbc:oracle:thin:@10.0.1.23:1521:demodb

org.quartz.dataSource.myDS.user = myUser

org.quartz.dataSource.myDS.password = myPassword

org.quartz.dataSource.myDS.maxConnections = 30

1

2

3

4

5

从一个应用服务中获取数据源配置:

org.quartz.dataSource.myOtherDS.jndiURL=jdbc/myDataSource

org.quartz.dataSource.myOtherDS.java.naming.factory.initial=com.evermind.server.rmi.RMIInitialContextFactory

org.quartz.dataSource.myOtherDS.java.naming.provider.url=ormi://localhost

org.quartz.dataSource.myOtherDS.java.naming.security.principal=admin

org.quartz.dataSource.myOtherDS.java.naming.security.credentials=123

1

2

3

4

5

**集群配置**

#============================================================================

# Configure Main Scheduler Properties

#============================================================================

org.quartz.scheduler.instanceName = MyClusteredScheduler

org.quartz.scheduler.instanceId = AUTO

#============================================================================

# Configure ThreadPool

#============================================================================

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

org.quartz.threadPool.threadCount = 25

org.quartz.threadPool.threadPriority = 5

#============================================================================

# Configure JobStore

#============================================================================

org.quartz.jobStore.misfireThreshold = 60000

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

org.quartz.jobStore.driverDelegateClass = org.quartz.impl.jdbcjobstore.oracle.OracleDelegate

org.quartz.jobStore.useProperties = false

org.quartz.jobStore.dataSource = myDS

org.quartz.jobStore.tablePrefix = QRTZ\_

org.quartz.jobStore.isClustered = true

org.quartz.jobStore.clusterCheckinInterval = 20000

#============================================================================

# Configure Datasources

#============================================================================

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

org.quartz.dataSource.myDS.URL = jdbc:oracle:thin:@polarbear:1521:dev

org.quartz.dataSource.myDS.user = quartz

org.quartz.dataSource.myDS.password = quartz

org.quartz.dataSource.myDS.maxConnections = 5

org.quartz.dataSource.myDS.validationQuery=select 0 from dual

# 6、源码解析

## 1、Trigger源码解析

### 0.建造者模式讲解

1. 介绍

1.1 定义

将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示

1.2 主要作用

在用户不知道对象的建造过程和细节的情况下就可以直接创建复杂的对象。

用户只需要给出指定复杂对象的类型和内容；

建造者模式负责按顺序创建复杂对象（把内部的建造过程和细节隐藏起来)

1.3 解决的问题

方便用户创建复杂的对象（不需要知道实现过程）

代码复用性 & 封装性（将对象构建过程和细节进行封装 & 复用）

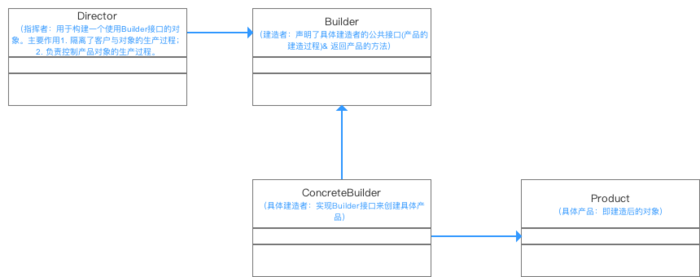
例子：造汽车 & 买汽车。

工厂（建造者模式）：负责制造汽车（组装过程和细节在工厂内）

汽车购买者（用户）：你只需要说出你需要的型号（对象的类型和内容），然后直接购买就可以使用了  
（不需要知道汽车是怎么组装的（车轮、车门、发动机、方向盘等等））

2. 模式原理

2.1 UML类图 & 组成



UML类图

模式讲解：

指挥者（Director）直接和客户（Client）进行需求沟通；

沟通后指挥者将客户创建产品的需求划分为各个部件的建造请求（Builder）；

将各个部件的建造请求委派到具体的建造者（ConcreteBuilder）；

各个具体建造者负责进行产品部件的构建；

最终构建成具体产品（Product）。

2.2 实例讲解

接下来我用一个实例来对建造者模式进行更深一步的介绍。  
**a. 实例概况**

背景：小成希望去电脑城买一台组装的台式主机

过程：

电脑城老板（Diretor）和小成（Client）进行需求沟通（买来打游戏？学习？看片？）

了解需求后，电脑城老板将小成需要的主机划分为各个部件（Builder）的建造请求（CPU、主板blabla）

指挥装机人员（ConcreteBuilder）去构建组件；

将组件组装起来成小成需要的电脑（Product）

**b. 使用步骤**  
**步骤1：** 定义组装的过程（Builder）：组装电脑的过程

public abstract class Builder {

//第一步：装CPU

//声明为抽象方法，具体由子类实现

public abstract void BuildCPU()；

//第二步：装主板

//声明为抽象方法，具体由子类实现

public abstract void BuildMainboard（）；

//第三步：装硬盘

//声明为抽象方法，具体由子类实现

public abstract void BuildHD（）；

//返回产品的方法：获得组装好的电脑

public abstract Computer GetComputer（）；

}

**步骤2：** 电脑城老板委派任务给装机人员（Director）

public class Director{

//指挥装机人员组装电脑

public void Construct(Builder builder){

builder. BuildCPU();

builder.BuildMainboard（）;

builder. BuildHD（）;

}

}

\*\*步骤3： \*\*创建具体的建造者（ConcreteBuilder）:装机人员

//装机人员1

public class ConcreteBuilder extend Builder{

//创建产品实例

Computer computer = new Computer();

//组装产品

@Override

public void BuildCPU(){

computer.Add("组装CPU")

}

@Override

public void BuildMainboard（）{

computer.Add("组装主板")

}

@Override

public void BuildHD（）{

computer.Add("组装主板")

}

//返回组装成功的电脑

@Override

public Computer GetComputer（）{

return computer

}

}

\*\*步骤4： \*\*定义具体产品类（Product）：电脑

public class Computer{

//电脑组件的集合

private List<String> parts = new ArrayList<String>()；

//用于将组件组装到电脑里

public void Add(String part){

parts.add(part);

}

public void Show(){

for (int i = 0;i<parts.size();i++){

System.out.println(“组件”+parts.get(i)+“装好了”);

}

System.out.println(“电脑组装完成，请验收”);

}

}

\*\*步骤5： \*\*客户端调用-小成到电脑城找老板买电脑

public class Builder Pattern{

public static void main(String[] args){

//逛了很久终于发现一家合适的电脑店

//找到该店的老板和装机人员

Director director = new Director();

Builder builder = new ConcreteBuilder();

//沟通需求后，老板叫装机人员去装电脑

director.Construct(builder);

//装完后，组装人员搬来组装好的电脑

Computer computer = builder.GetComputer();

//组装人员展示电脑给小成看

computer.Show()；

}

}

结果输出

组件CUP装好了

组件主板装好了

组件硬盘装好了

电脑组装完成，请验收

### 1.触发器接口基本介绍

Trigger触发器方法介绍

// 触发器状态

TriggerState

|-public enum TriggerState { NONE, NORMAL, PAUSED, COMPLETE, ERROR, BLOCKED }

|-NONE 无

|-NORMAL 正常状态

|-PAUSED 暂停状态

|-COMPLETE 完成

|-ERROR 错误

|-BLOCKED 堵塞

// 执行完成时状态

CompletedExecutionInstruction

|- public enum CompletedExecutionInstruction {

NOOP, RE\_EXECUTE\_JOB, SET\_TRIGGER\_COMPLETE, DELETE\_TRIGGER,

SET\_ALL\_JOB\_TRIGGERS\_COMPLETE, SET\_TRIGGER\_ERROR, SET\_ALL\_JOB\_TRIGGERS\_ERROR }

|-NOOP 无

|-RE\_EXECUTE\_JOB 重复执行

|-SET\_TRIGGER\_COMPLETE 触发器执行完成

|-DELETE\_TRIGGER 删除触发器

|-SET\_ALL\_JOB\_TRIGGERS\_COMPLETE 所有作业和触发器执行完成

|-SET\_TRIGGER\_ERROR 触发器执行错误

|-SET\_ALL\_JOB\_TRIGGERS\_ERROR 设置所有都是错误的

TriggerTimeComparator

getKey 获取触发器key值

getJobKey 获取作业key

getDescription 获取面熟

getCalendarName 获取日历名称

getJobDataMap 获取作业数据map

getPriority 获取优先级

mayFireAgain 是否重复执行

getStartTime 开始时间

getEndTime 结束时间

getNextFireTime 下一次执行时间

getPreviousFireTime 上一执行时间

getFireTimeAfter(Date afterTime) 获取某个时间后的运行时间

getFinalFireTime 获取最后执行时间

getMisfireInstruction 获取失败策略

getTriggerBuilder 获取触发器建造者

getScheduleBuilder 获取调度类建造者

equals

compareTo

// 失败策略

MISFIRE\_INSTRUCTION\_SMART\_POLICY

MISFIRE\_INSTRUCTION\_IGNORE\_MISFIRE\_POLICY

DEFAULT\_PRIORITY

### 2.触发器实现类

Trigger (org.quartz)

|-CalendarIntervalTrigger (org.quartz) 日期触发器

| |-CalendarIntervalTriggerImpl (org.quartz.impl.triggers)

|-MutableTrigger (org.quartz.spi)

| |-OperableTrigger (org.quartz.spi)

| |-AbstractTrigger (org.quartz.impl.triggers)

| |-CalendarIntervalTriggerImpl (org.quartz.impl.triggers)

| |-SimpleTriggerImpl (org.quartz.impl.triggers)

| |-DailyTimeIntervalTriggerImpl (org.quartz.impl.triggers)

| |-CronTriggerImpl (org.quartz.impl.triggers)

|-SimpleTrigger (org.quartz) 简单触发器

| |-SimpleTriggerImpl (org.quartz.impl.triggers)

|-CoreTrigger (org.quartz.impl.triggers)

| |-CalendarIntervalTriggerImpl (org.quartz.impl.triggers)

| |-SimpleTriggerImpl (org.quartz.impl.triggers)

| |-DailyTimeIntervalTriggerImpl (org.quartz.impl.triggers)

| |-CronTriggerImpl (org.quartz.impl.triggers)

|-CronTrigger (org.quartz) cron表达式

| |-CronTriggerImpl (org.quartz.impl.triggers)

|-DailyTimeIntervalTrigger (org.quartz)日期触发类(日)

|-DailyTimeIntervalTriggerImpl (org.quartz.impl.triggers)

### 3.调度器建造者

// 用于创建各个调度器

ScheduleBuilder (org.quartz)

|-CalendarIntervalScheduleBuilder (org.quartz)

|-DailyTimeIntervalScheduleBuilder (org.quartz)

|-SimpleScheduleBuilder (org.quartz)

|-CronScheduleBuilder (org.quartz)

### 4.触发器建造者

简单触发器器源码分析

利用了建造者模式

private TriggerBuilder() 构造函数私有

public static TriggerBuilder<Trigger> newTrigger() 创建一个建造者

build() 创建触发器

// 根据name和默认的group(即"DEFAULT\_GROUP")创建trigger的key

public TriggerBuilder<T> withIdentity(String name)

public TriggerBuilder<T> withIdentity(String name, String group)

public TriggerBuilder<T> withIdentity(TriggerKey triggerKey)

public TriggerBuilder<T> withDescription(String triggerDescription) 描述

public TriggerBuilder<T> withPriority(int triggerPriority) 优先级

public TriggerBuilder<T> modifiedByCalendar(String calName) 日期

public TriggerBuilder<T> startAt(Date triggerStartTime) 开始时间

public TriggerBuilder<T> startNow() 立即执行

public TriggerBuilder<T> endAt(Date triggerEndTime) 结束时间

public <SBT extends T> TriggerBuilder<SBT> withSchedule(ScheduleBuilder<SBT> schedBuilder) 调度器

public TriggerBuilder<T> forJob(JobKey keyOfJobToFire) 设置作业

public TriggerBuilder<T> forJob(String jobName)

public TriggerBuilder<T> forJob(String jobName, String jobGroup)

public TriggerBuilder<T> forJob(JobDetail jobDetail)

usingJobData(----,----) 设置作业内容

key

description

startTime

endTime

priority Trigger.DEFAULT\_PRIORITY

calendarName

jobKey

jobDataMap

scheduleBuilder

### 5.简单调度器源码分析

利用了建造者模式

//构造函数私有化

protected SimpleScheduleBuilder()

//获取简单调度器

public static SimpleScheduleBuilder simpleSchedule()

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// 1分钟执行(一直执行)

public static SimpleScheduleBuilder repeatMinutelyForever()

//每隔几分钟执行(一直执行)

public static SimpleScheduleBuilder repeatMinutelyForever(int minutes)

// 1秒执行(一直执行)

public static SimpleScheduleBuilder repeatSecondlyForever()

//每隔几秒钟执行(一直执行)

public static SimpleScheduleBuilder repeatSecondlyForever(int seconds)

// 1小时执行(一直执行)

public static SimpleScheduleBuilder repeatHourlyForever()

//每隔几小时钟执行(一直执行)

public static SimpleScheduleBuilder repeatHourlyForever(int hours)

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

)//间隔时间为1分钟，总的执行次数为count

public static SimpleScheduleBuilder repeatMinutelyForTotalCount(int count

//间隔时间为几分钟，总的执行次数为count .............

public static SimpleScheduleBuilder repeatMinutelyForTotalCount(int count, int minutes)

public static SimpleScheduleBuilder repeatSecondlyForTotalCount(int count)

public static SimpleScheduleBuilder repeatSecondlyForTotalCount(int count, int seconds)

public static SimpleScheduleBuilder repeatHourlyForTotalCount(int count)

public static SimpleScheduleBuilder repeatHourlyForTotalCount(int count, int hours)

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public MutableTrigger build() 创建一个Trigger

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// 几秒钟重复执行

public SimpleScheduleBuilder withIntervalInMilliseconds(long intervalInMillis)

public SimpleScheduleBuilder withIntervalInSeconds(int intervalInSeconds)

public SimpleScheduleBuilder withIntervalInMinutes(int intervalInMinutes)

public SimpleScheduleBuilder withIntervalInHours(int intervalInHours)

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// 重复执行册数

public SimpleScheduleBuilder withRepeatCount(int triggerRepeatCount)

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

参考地址

http://blog.sina.com.cn/s/blog\_56d8ea900101eu45.html

//以错过的第一个频率时间立刻开始执行

//重做错过的所有频率周期后

//当下一次触发频率发生时间大于当前时间后，再按照正常的Cron频率依次执行

public SimpleScheduleBuilder withMisfireHandlingInstructionIgnoreMisfires()

//以当前时间为触发频率立即触发执行

//执行至FinalTIme的剩余周期次数

//以调度或恢复调度的时刻为基准的周期频率，FinalTime根据剩余次数和当前时间计算得到

//调整后的FinalTime会略大于根据starttime计算的到的FinalTime值

public SimpleScheduleBuilder withMisfireHandlingInstructionFireNow()

//不触发立即执行

//等待下次触发频率周期时刻，执行至FinalTime的剩余周期次数

//以startTime为基准计算周期频率，并得到FinalTime

//即使中间出现pause，resume以后保持FinalTime时间不变

public SimpleScheduleBuilder withMisfireHandlingInstructionNextWithExistingCount()

//不触发立即执行

//等待下次触发频率周期时刻，执行至FinalTime的剩余周期次数

//以startTime为基准计算周期频率，并得到FinalTime

//即使中间出现pause，resume以后保持FinalTime时间不变

public SimpleScheduleBuilder withMisfireHandlingInstructionNextWithRemainingCount()

//以当前时间为触发频率立即触发执行

//执行至FinalTIme的剩余周期次数

//以调度或恢复调度的时刻为基准的周期频率，FinalTime根据剩余次数和当前时间计算得到

//调整后的FinalTime会略大于根据starttime计算的到的FinalTime值

public SimpleScheduleBuilder withMisfireHandlingInstructionNowWithExistingCount()

//以当前时间为触发频率立即触发执行

//执行至FinalTIme的剩余周期次数

//以调度或恢复调度的时刻为基准的周期频率，FinalTime根据剩余次数和当前时间计算得到

public SimpleScheduleBuilder withMisfireHandlingInstructionNowWithRemainingCount()

interval 时间间隔

repeatCount 重复时间

misfireInstruction

### 6.Cron调度器建造者源码分析

// 构造函数私有化

protected CronScheduleBuilder(CronExpression cronExpression)

public MutableTrigger build()

// 根据cron表达式建造

public static CronScheduleBuilder cronSchedule(String cronExpression)

// 核查表达式是否正确

public static CronScheduleBuilder cronScheduleNonvalidatedExpression(

String cronExpression) throws ParseException

//表达式异常

cronScheduleNoParseException

// 利用CronExpression建造

public static CronScheduleBuilder cronSchedule(CronExpression cronExpression)

//每天在指定的时间执行，根据这个调度创建一个cron表达式

public static CronScheduleBuilder dailyAtHourAndMinute(int hour, int minute)

// 通过`分钟`、`小时`、`周`创建一个CronScheduleBuilder实例，即在某一天的给定时刻

// (通过`分钟`、`小时`指定)执行，，而天数由`周`确定，如果“周二、周四的10:05“等；

public static CronScheduleBuilder atHourAndMinuteOnGivenDaysOfWeek(

int hour, int minute, Integer... daysOfWeek)

//调度计划：每周的某一天，在指定的时间（小时和分钟）执行

public static CronScheduleBuilder weeklyOnDayAndHourAndMinute(

int dayOfWeek, int hour, int minute)

//调度计划：每月的某一天，在指定的时间（小时和分钟）执行

public static CronScheduleBuilder monthlyOnDayAndHourAndMinute(

int dayOfMonth, int hour, int minute)

//设置时区

public CronScheduleBuilder inTimeZone(TimeZone timezone)

// 设置处理办法

public CronScheduleBuilder withMisfireHandlingInstructionIgnoreMisfires()

public CronScheduleBuilder withMisfireHandlingInstructionDoNothing()

public CronScheduleBuilder withMisfireHandlingInstructionFireAndProceed()

cronExpression

misfireInstruction

### 7.DailyTime调度器源码分析

// 建造者模式

DailyTimeIntervalScheduleBuilder

dailyTimeIntervalSchedule

build

//执行时间间隔触发执行,unit时间单位

public DailyTimeIntervalScheduleBuilder withInterval(int timeInterval, IntervalUnit unit)

//秒

withIntervalInSeconds

//分钟

withIntervalInMinutes

//小时

withIntervalInHours

// 周几执行

public DailyTimeIntervalScheduleBuilder onDaysOfTheWeek(Integer … onDaysOfWeek)

onDaysOfTheWeek

onMondayThroughFriday

onSaturdayAndSunday

onEveryDay

// 开始触发时间

startingDailyAt

//结束时间

endingDailyAt

//支持次数

endingDailyAfterCount

withMisfireHandlingInstructionIgnoreMisfires

withMisfireHandlingInstructionDoNothing

withMisfireHandlingInstructionFireAndProceed

//重复次数

withRepeatCount

validateInterval

// 常量等

interval

intervalUnit

daysOfWeek

startTimeOfDay

endTimeOfDay

repeatCount

misfireInstruction

ALL\_DAYS\_OF\_THE\_WEEK

MONDAY\_THROUGH\_FRIDAY

SATURDAY\_AND\_SUNDAY

### 8.Calendar调度器建造者源码分析

CalendarIntervalScheduleBuilder

calendarIntervalSchedule

build

// 和DailyTimeIntervalScheduleBuilder差不多

public CalendarIntervalScheduleBuilder withInterval(int timeInterval, IntervalUnit unit)

withIntervalInSeconds

withIntervalInMinutes

withIntervalInHours

withIntervalInDays

withIntervalInWeeks

withIntervalInMonths

withIntervalInYears

withMisfireHandlingInstructionIgnoreMisfires

withMisfireHandlingInstructionDoNothing

withMisfireHandlingInstructionFireAndProceed

inTimeZone

preserveHourOfDayAcrossDaylightSavings

skipDayIfHourDoesNotExist

validateInterval

interval

intervalUnit

misfireInstruction

timeZone

preserveHourOfDayAcrossDaylightSavings

skipDayIfHourDoesNotExist

## 2、job和jobDetail源码分析

## 3、scheduler源码分析

# 7、spring+quartz的分布式定时任务框架

## quartz.properties：

#============================================================================

# Configure JobStore

# Using Spring datasource in quartzJobsConfig.xml

# Spring uses LocalDataSourceJobStore extension of JobStoreCMT

#============================================================================

org.quartz.jobStore.useProperties=true

org.quartz.jobStore.tablePrefix = QRTZ\_

org.quartz.jobStore.isClustered = true

org.quartz.jobStore.clusterCheckinInterval = 5000

org.quartz.jobStore.misfireThreshold = 60000

org.quartz.jobStore.txIsolationLevelReadCommitted = true

# Change this to match your DB vendor

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

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

#============================================================================

# Configure Main Scheduler Properties

# Needed to manage cluster instances

#============================================================================

org.quartz.scheduler.instanceId=AUTO

org.quartz.scheduler.instanceName=MY\_CLUSTERED\_JOB\_SCHEDULER

org.quartz.scheduler.rmi.export = false

org.quartz.scheduler.rmi.proxy = false

#============================================================================

# Configure ThreadPool

#============================================================================

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

org.quartz.threadPool.threadCount = 10

org.quartz.threadPool.threadPriority = 5

org.quartz.threadPool.threadsInheritContextClassLoaderOfInitializingThread = true

## maven依赖 ：

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>4.2.6.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>4.2.6.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aop</artifactId>

<version>4.2.6.RELEASE</version>

</dependency>

<dependency>

<groupId>org.quartz-scheduler</groupId>

<artifactId>quartz</artifactId>

<version>2.2.1</version>

</dependency>

<dependency>

<groupId>org.quartz-scheduler</groupId>

<artifactId>quartz-jobs</artifactId>

<version>2.2.1</version>

</dependency>

<dependency>

<groupId>commons-logging</groupId>

<artifactId>commons-logging</artifactId>

<version>1.2</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-web</artifactId>

<version>4.2.6.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context-support</artifactId>

<version>4.2.6.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-tx</artifactId>

<version>4.2.6.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jdbc</artifactId>

<version>4.2.6.RELEASE</version>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>servlet-api</artifactId>

<version>LATEST</version>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.2</version>

</dependency>

<dependency>

<groupId>commons-dbcp</groupId>

<artifactId>commons-dbcp</artifactId>

<version>1.4</version>

</dependency>

## spring.xml：

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns:context=*"http://www.springframework.org/schema/context"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-4.0.xsd http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd"*>

<context:component-scan base-package=*"com.lidahai.job"*/>

<bean id=*"dataSource"* class=*"org.apache.commons.dbcp.BasicDataSource"*>

<property name=*"driverClassName"* value=*"com.mysql.jdbc.Driver"*></property>

<property name=*"url"* value=*"jdbc:mysql://localhost:3306/test2"*></property>

<property name=*"username"* value=*"root"*></property>

<property name=*"password"* value=*"123456"*></property>

</bean>

<!-- 分布式事务配置 start -->

<!-- 配置线程池-->

<bean name=*"executor"* class=*"org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor"*>

<property name=*"corePoolSize"* value=*"15"*/>

<property name=*"maxPoolSize"* value=*"25"*/>

<property name=*"queueCapacity"* value=*"100"*/>

</bean>

<bean name=*"transactionManager"* class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>

<property name=*"dataSource"* ref=*"dataSource"*/>

</bean>

<!-- 配置调度任务-->

<bean name=*"quartzScheduler"* class=*"org.springframework.scheduling.quartz.SchedulerFactoryBean"*>

<property name=*"configLocation"* value=*"classpath:quartz.properties"*/>

<property name=*"dataSource"* ref=*"dataSource"*/>

<property name=*"transactionManager"* ref=*"transactionManager"*/>

<!-- 任务唯一的名称，将会持久化到数据库-->

<property name=*"schedulerName"* value=*"baseScheduler"*/>

<!-- 每台集群机器部署应用的时候会更新触发器-->

<property name=*"overwriteExistingJobs"* value=*"true"*/>

<property name=*"applicationContextSchedulerContextKey"* value=*"appli22"*/>

<property name=*"jobFactory"*>

<bean class=*"com.lidahai.autowired.AutowiringSpringBeanJobFactory"*/>

</property>

<property name=*"triggers"*>

<list>

<ref bean=*"printCurrentTimeScheduler"*/>

</list>

</property>

<property name=*"jobDetails"*>

<list>

<ref bean=*"printCurrentTimeJobs"*/>

</list>

</property>

<property name=*"taskExecutor"* ref=*"executor"*/>

</bean>

<!-- 配置Job详情 -->

<bean name=*"printCurrentTimeJobs"* class=*"org.springframework.scheduling.quartz.JobDetailFactoryBean"*>

<property name=*"jobClass"* value=*"com.lidahai.job.PrintCurrentTimeJobs"*/>

<!--<property name="jobDataAsMap">

<map>

<entry key="clusterQuartz" value="com.aaron.framework.clusterquartz.job.ClusterQuartz"/>

</map>

</property>-->

<property name=*"durability"* value=*"true"*/>

<property name=*"requestsRecovery"* value=*"false"*/>

</bean>

<!-- 配置触发时间 -->

<bean name=*"printCurrentTimeScheduler"* class=*"org.springframework.scheduling.quartz.CronTriggerFactoryBean"*>

<property name=*"jobDetail"* ref=*"printCurrentTimeJobs"*/>

<property name=*"cronExpression"*>

<value>0/10 \* \* \* \* ?</value>

</property>

<property name=*"timeZone"*>

<value>GMT+8:00</value>

</property>

</bean>

<!-- 分布式事务配置 end -->

</beans>

## 建立表：

#

# Quartz seems to work best **with** the driver mysql-connector-java-5.1.34-bin.jar

#

# PLEASE consider **using** mysql **with** innodb tables to avoid locking issues

#

# In your Quartz properties file, you'll need to set

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

#

DROP TABLE IF EXISTS QRTZ\_FIRED\_TRIGGERS;

DROP TABLE IF EXISTS QRTZ\_PAUSED\_TRIGGER\_GRPS;

DROP TABLE IF EXISTS QRTZ\_SCHEDULER\_STATE;

DROP TABLE IF EXISTS QRTZ\_LOCKS;

DROP TABLE IF EXISTS QRTZ\_SIMPLE\_TRIGGERS;

DROP TABLE IF EXISTS QRTZ\_SIMPROP\_TRIGGERS;

DROP TABLE IF EXISTS QRTZ\_CRON\_TRIGGERS;

DROP TABLE IF EXISTS QRTZ\_BLOB\_TRIGGERS;

DROP TABLE IF EXISTS QRTZ\_TRIGGERS;

DROP TABLE IF EXISTS QRTZ\_JOB\_DETAILS;

DROP TABLE IF EXISTS QRTZ\_CALENDARS;

CREATE TABLE QRTZ\_JOB\_DETAILS

(

SCHED\_NAME VARCHAR(120) NOT NULL,

JOB\_NAME VARCHAR(200) NOT NULL,

JOB\_GROUP VARCHAR(200) NOT NULL,

DESCRIPTION VARCHAR(250) NULL,

JOB\_CLASS\_NAME VARCHAR(250) NOT NULL,

IS\_DURABLE VARCHAR(1) NOT NULL,

IS\_NONCONCURRENT VARCHAR(1) NOT NULL,

IS\_UPDATE\_DATA VARCHAR(1) NOT NULL,

REQUESTS\_RECOVERY VARCHAR(1) NOT NULL,

JOB\_DATA BLOB NULL,

PRIMARY KEY (SCHED\_NAME,JOB\_NAME,JOB\_GROUP)

);

CREATE TABLE QRTZ\_TRIGGERS

(

SCHED\_NAME VARCHAR(120) NOT NULL,

TRIGGER\_NAME VARCHAR(200) NOT NULL,

TRIGGER\_GROUP VARCHAR(200) NOT NULL,

JOB\_NAME VARCHAR(200) NOT NULL,

JOB\_GROUP VARCHAR(200) NOT NULL,

DESCRIPTION VARCHAR(250) NULL,

NEXT\_FIRE\_TIME BIGINT(13) NULL,

PREV\_FIRE\_TIME BIGINT(13) NULL,

PRIORITY INTEGER NULL,

TRIGGER\_STATE VARCHAR(16) NOT NULL,

TRIGGER\_TYPE VARCHAR(8) NOT NULL,

START\_TIME BIGINT(13) NOT NULL,

END\_TIME BIGINT(13) NULL,

CALENDAR\_NAME VARCHAR(200) NULL,

MISFIRE\_INSTR SMALLINT(2) NULL,

JOB\_DATA BLOB NULL,

PRIMARY KEY (SCHED\_NAME,TRIGGER\_NAME,TRIGGER\_GROUP),

FOREIGN KEY (SCHED\_NAME,JOB\_NAME,JOB\_GROUP)

REFERENCES QRTZ\_JOB\_DETAILS(SCHED\_NAME,JOB\_NAME,JOB\_GROUP)

);

CREATE TABLE QRTZ\_SIMPLE\_TRIGGERS

(

SCHED\_NAME VARCHAR(120) NOT NULL,

TRIGGER\_NAME VARCHAR(200) NOT NULL,

TRIGGER\_GROUP VARCHAR(200) NOT NULL,

REPEAT\_COUNT BIGINT(7) NOT NULL,

REPEAT\_INTERVAL BIGINT(12) NOT NULL,

TIMES\_TRIGGERED BIGINT(10) NOT NULL,

PRIMARY KEY (SCHED\_NAME,TRIGGER\_NAME,TRIGGER\_GROUP),

FOREIGN KEY (SCHED\_NAME,TRIGGER\_NAME,TRIGGER\_GROUP)

REFERENCES QRTZ\_TRIGGERS(SCHED\_NAME,TRIGGER\_NAME,TRIGGER\_GROUP)

);

CREATE TABLE QRTZ\_CRON\_TRIGGERS

(

SCHED\_NAME VARCHAR(120) NOT NULL,

TRIGGER\_NAME VARCHAR(200) NOT NULL,

TRIGGER\_GROUP VARCHAR(200) NOT NULL,

CRON\_EXPRESSION VARCHAR(200) NOT NULL,

TIME\_ZONE\_ID VARCHAR(80),

PRIMARY KEY (SCHED\_NAME,TRIGGER\_NAME,TRIGGER\_GROUP),

FOREIGN KEY (SCHED\_NAME,TRIGGER\_NAME,TRIGGER\_GROUP)

REFERENCES QRTZ\_TRIGGERS(SCHED\_NAME,TRIGGER\_NAME,TRIGGER\_GROUP)

);

CREATE TABLE QRTZ\_SIMPROP\_TRIGGERS

(

SCHED\_NAME VARCHAR(120) NOT NULL,

TRIGGER\_NAME VARCHAR(200) NOT NULL,

TRIGGER\_GROUP VARCHAR(200) NOT NULL,

STR\_PROP\_1 VARCHAR(512) NULL,

STR\_PROP\_2 VARCHAR(512) NULL,

STR\_PROP\_3 VARCHAR(512) NULL,

INT\_PROP\_1 INT NULL,

INT\_PROP\_2 INT NULL,

LONG\_PROP\_1 BIGINT NULL,

LONG\_PROP\_2 BIGINT NULL,

DEC\_PROP\_1 NUMERIC(13,4) NULL,

DEC\_PROP\_2 NUMERIC(13,4) NULL,

BOOL\_PROP\_1 VARCHAR(1) NULL,

BOOL\_PROP\_2 VARCHAR(1) NULL,

PRIMARY KEY (SCHED\_NAME,TRIGGER\_NAME,TRIGGER\_GROUP),

FOREIGN KEY (SCHED\_NAME,TRIGGER\_NAME,TRIGGER\_GROUP)

REFERENCES QRTZ\_TRIGGERS(SCHED\_NAME,TRIGGER\_NAME,TRIGGER\_GROUP)

);

CREATE TABLE QRTZ\_BLOB\_TRIGGERS

(

SCHED\_NAME VARCHAR(120) NOT NULL,

TRIGGER\_NAME VARCHAR(200) NOT NULL,

TRIGGER\_GROUP VARCHAR(200) NOT NULL,

BLOB\_DATA BLOB NULL,

PRIMARY KEY (SCHED\_NAME,TRIGGER\_NAME,TRIGGER\_GROUP),

FOREIGN KEY (SCHED\_NAME,TRIGGER\_NAME,TRIGGER\_GROUP)

REFERENCES QRTZ\_TRIGGERS(SCHED\_NAME,TRIGGER\_NAME,TRIGGER\_GROUP)

);

CREATE TABLE QRTZ\_CALENDARS

(

SCHED\_NAME VARCHAR(120) NOT NULL,

CALENDAR\_NAME VARCHAR(200) NOT NULL,

CALENDAR BLOB NOT NULL,

PRIMARY KEY (SCHED\_NAME,CALENDAR\_NAME)

);

CREATE TABLE QRTZ\_PAUSED\_TRIGGER\_GRPS

(

SCHED\_NAME VARCHAR(120) NOT NULL,

TRIGGER\_GROUP VARCHAR(200) NOT NULL,

PRIMARY KEY (SCHED\_NAME,TRIGGER\_GROUP)

);

CREATE TABLE QRTZ\_FIRED\_TRIGGERS

(

SCHED\_NAME VARCHAR(120) NOT NULL,

ENTRY\_ID VARCHAR(95) NOT NULL,

TRIGGER\_NAME VARCHAR(200) NOT NULL,

TRIGGER\_GROUP VARCHAR(200) NOT NULL,

INSTANCE\_NAME VARCHAR(200) NOT NULL,

FIRED\_TIME BIGINT(13) NOT NULL,

SCHED\_TIME BIGINT(13) NOT NULL,

PRIORITY INTEGER NOT NULL,

STATE VARCHAR(16) NOT NULL,

JOB\_NAME VARCHAR(200) NULL,

JOB\_GROUP VARCHAR(200) NULL,

IS\_NONCONCURRENT VARCHAR(1) NULL,

REQUESTS\_RECOVERY VARCHAR(1) NULL,

PRIMARY KEY (SCHED\_NAME,ENTRY\_ID)

);

CREATE TABLE QRTZ\_SCHEDULER\_STATE

(

SCHED\_NAME VARCHAR(120) NOT NULL,

INSTANCE\_NAME VARCHAR(200) NOT NULL,

LAST\_CHECKIN\_TIME BIGINT(13) NOT NULL,

CHECKIN\_INTERVAL BIGINT(13) NOT NULL,

PRIMARY KEY (SCHED\_NAME,INSTANCE\_NAME)

);

CREATE TABLE QRTZ\_LOCKS

(

SCHED\_NAME VARCHAR(120) NOT NULL,

LOCK\_NAME VARCHAR(40) NOT NULL,

PRIMARY KEY (SCHED\_NAME,LOCK\_NAME)

);

commit;

## 代码：

AutowiringSpringBeanJobFactory：

import org.quartz.spi.TriggerFiredBundle;

import org.springframework.beans.BeansException;

import org.springframework.beans.factory.config.AutowireCapableBeanFactory;

import org.springframework.context.ApplicationContext;

import org.springframework.context.ApplicationContextAware;

import org.springframework.scheduling.quartz.SpringBeanJobFactory;

/\*\*

使job类支持spring的自动注入

\*/

public class AutowiringSpringBeanJobFactory extends SpringBeanJobFactory implements ApplicationContextAware

{

private transient AutowireCapableBeanFactory beanFactory;

public void setApplicationContext(ApplicationContext applicationContext) throws BeansException

{

beanFactory = applicationContext.getAutowireCapableBeanFactory();

}

@Override

protected Object createJobInstance(TriggerFiredBundle bundle) throws Exception

{

Object job = super.createJobInstance(bundle);

beanFactory.autowireBean(job);

return job;

}

}

ClusterQuartz：

**import** org.springframework.stereotype.Controller;

**import** java.util.Date;

@Controller

**public** **class** ClusterQuartz

{

**public** **void** printUserInfo()

{

System.***out***.println("\*\*\* start " + DateUtils.*dateToString*(**new** Date(), "yyyy-MM-dd HH:mm:ss:SSS") + " \*\*\*\*\*\*\*\*\*\*\*\*\*");

System.***out***.println("\*");

System.***out***.println("\* current username is " + System.*getProperty*("user.name"));

System.***out***.println("\* current os name is " + System.*getProperty*("os.name"));

System.***out***.println("\*");

System.***out***.println("\*\*\*\*\*\*\*\*\*current user information end\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

}

}

PrintCurrentTimeJobs：

import org.apache.commons.logging.Log;

import org.apache.commons.logging.LogFactory;

import org.quartz.JobExecutionContext;

import org.quartz.JobExecutionException;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.scheduling.quartz.QuartzJobBean;

import java.util.Date;

public class PrintCurrentTimeJobs extends QuartzJobBean

{

private static final Log LOG\_RECORD = LogFactory.getLog(PrintCurrentTimeJobs.class);

@Autowired

private ClusterQuartz clusterQuartz;

protected void executeInternal(JobExecutionContext jobExecutionContext) throws JobExecutionException

{

LOG\_RECORD.info("begin to execute task," + DateUtils.dateToString(new Date()));

clusterQuartz.printUserInfo();

LOG\_RECORD.info("end to execute task," + DateUtils.dateToString(new Date()));

}

}

Test：

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

**public** **class** Test

{

**public** **static** **void** main(String[] args)

{

ClassPathXmlApplicationContext context = **new** ClassPathXmlApplicationContext("spring.xml");

ClusterQuartz clusterQuartz = (ClusterQuartz)context.getBean("clusterQuartz");

clusterQuartz.printUserInfo();

**while** (**true**)

{

}

}

}

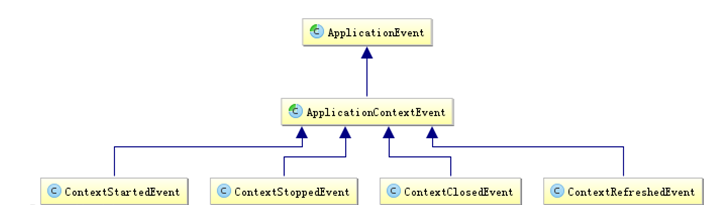
# 8、spring+quartz源码解析之事件驱动模型讲解

## 事件

具体代表者是：ApplicationEvent：

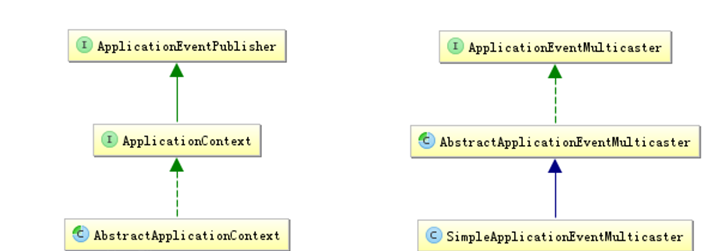
1、其继承自JDK的EventObject，JDK要求所有事件将继承它，并通过source得到事件源，比如我们的AWT事件体系也是继承自它；

2、系统默认提供了如下ApplicationEvent事件实现：

  
只有一个ApplicationContextEvent，表示ApplicationContext容器事件，且其又有如下实现：

* ContextStartedEvent：ApplicationContext启动后触发的事件；
* ContextStoppedEvent：ApplicationContext停止后触发的事件； ContextRefreshedEvent：ApplicationContext初始化或刷新完成后触发的事件；容器初始化完成后调用
* ContextClosedEvent：ApplicationContext关闭后触发的事件；

## 目标（发布事件者）

具体代表者是：ApplicationEventPublisher及ApplicationEventMulticaster，系统默认提供了如下实现：  


1、ApplicationContext接口继承了ApplicationEventPublisher，并在AbstractApplicationContext实现了具体代码，实际执行是委托给ApplicationEventMulticaster（可以认为是多播）：

**Java代码：**

**public void publishEvent(ApplicationEvent event) {**

**//省略部分代码**

**}**

**getApplicationEventMulticaster().multicastEvent(event);**

**if (this.parent != null) {**

**this.parent.publishEvent(event);**

**}**

**}**

我们常用的ApplicationContext都继承自AbstractApplicationContext，如ClassPathXmlApplicationContext、XmlWebApplicationContext等。所以自动拥有这个功能。

2、ApplicationContext自动到本地容器里找一个名字为”“的ApplicationEventMulticaster实现，如果没有自己new一个SimpleApplicationEventMulticaster。其中SimpleApplicationEventMulticaster发布事件的代码如下：

**public void multicastEvent(final ApplicationEvent event) {**

**for (final ApplicationListener listener : getApplicationListeners(event)) {**

**Executor executor = getTaskExecutor();**

**if (executor != null) {**

**executor.execute(new Runnable() {**

**public void run() {**

**listener.onApplicationEvent(event);**

**}**

**});**

**}**

**else {**

**listener.onApplicationEvent(event);**

**}**

**}**

**}**

## 监听器

具体代表者是：ApplicationListener

1、其继承自JDK的EventListener，JDK要求所有监听器将继承它，比如我们的AWT事件体系也是继承自它；

2、ApplicationListener接口：

**public interface ApplicationListener<E extends ApplicationEvent> extends EventListener {**

**void onApplicationEvent(E event);**

**}**

其只提供了onApplicationEvent方法，我们需要在该方法实现内部判断事件类型来处理，也没有提供按顺序触发监听器的语义，所以Spring提供了另一个接口，SmartApplicationListener：

**public interface SmartApplicationListener extends ApplicationListener<ApplicationEvent>, Ordered {**

**//如果实现支持该事件类型 那么返回true**

**boolean supportsEventType(Class<? extends ApplicationEvent> eventType);**

**//如果实现支持“目标”类型，那么返回true**

**boolean supportsSourceType(Class<?> sourceType);**

**//顺序，即监听器执行的顺序，值越小优先级越高**

**int getOrder();**

**}**

## 重构代码：

### 0、maven：

**<dependency>**

**<groupId>org.springframework</groupId>**

**<artifactId>spring-core</artifactId>**

**<version>4.2.6.RELEASE</version>**

**</dependency>**

**<dependency>**

**<groupId>org.springframework</groupId>**

**<artifactId>spring-context</artifactId>**

**<version>4.2.6.RELEASE</version>**

**</dependency>**

**<dependency>**

**<groupId>org.springframework</groupId>**

**<artifactId>spring-aop</artifactId>**

**<version>4.2.6.RELEASE</version>**

**</dependency>**

**<dependency>**

**<groupId>commons-logging</groupId>**

**<artifactId>commons-logging</artifactId>**

**<version>1.2</version>**

**</dependency>**

**<dependency>**

**<groupId>org.springframework</groupId>**

**<artifactId>spring-web</artifactId>**

**<version>4.2.6.RELEASE</version>**

**</dependency>**

**<dependency>**

**<groupId>org.springframework</groupId>**

**<artifactId>spring-context-support</artifactId>**

**<version>4.2.6.RELEASE</version>**

**</dependency>**

**<dependency>**

**<groupId>org.springframework</groupId>**

**<artifactId>spring-tx</artifactId>**

**<version>4.2.6.RELEASE</version>**

**</dependency>**

**<dependency>**

**<groupId>org.springframework</groupId>**

**<artifactId>spring-jdbc</artifactId>**

**<version>4.2.6.RELEASE</version>**

**</dependency>**

**<dependency>**

**<groupId>javax.servlet</groupId>**

**<artifactId>servlet-api</artifactId>**

**<version>LATEST</version>**

**</dependency>**

### 1、定义事件

**import org.springframework.context.ApplicationEvent;**

**public class RegisterEvent extends ApplicationEvent {**

**public RegisterEvent(final String userinfo) {**

**super(userinfo);**

**}**

**}**

### 2、定义无序监听器

**import org.springframework.context.ApplicationEvent;**

**import org.springframework.context.ApplicationListener;**

**import org.springframework.stereotype.Component;**

**@Component**

**public class PointListener implements ApplicationListener<ApplicationEvent> {**

**PointService pointService = new PointService();**

**@Override**

**public void onApplicationEvent(final ApplicationEvent event) {**

**if(event instanceof RegisterEvent) {**

**System.out.println("PointListener 接收到的注册用户是：" + event.getSource());**

**pointService.addPoint();**

**}**

**}**

**}**

**import org.springframework.context.ApplicationEvent;**

**import org.springframework.context.ApplicationListener;**

**import org.springframework.stereotype.Component;**

**@Component**

**public class EmailListener implements ApplicationListener<ApplicationEvent> {**

**EmailService emailService = new EmailService();**

**@Override**

**public void onApplicationEvent(final ApplicationEvent event) {**

**if(event instanceof RegisterEvent) {**

**System.out.println("EmailListener 接收到的注册用户是：" + event.getSource());**

**emailService.configemail();**

**}**

**}**

**}**

### 3、定义有序监听器

**import org.springframework.context.ApplicationEvent;**

**import org.springframework.context.event.SmartApplicationListener;**

**import org.springframework.stereotype.Component;**

**@Component**

**public class IndexListener implements SmartApplicationListener {**

**IndexService indexService = new IndexService();**

**@Override**

**public boolean supportsEventType(final Class<? extends ApplicationEvent> eventType) {**

**return eventType == RegisterEvent.class;**

**}**

**@Override**

**public boolean supportsSourceType(final Class<?> sourceType) {**

**return sourceType == String.class;**

**}**

**@Override**

**public void onApplicationEvent(final ApplicationEvent event) {**

**System.out.println("EmailListener 接收到的注册用户是：" + event.getSource());**

**indexService.index();**

**}**

**@Override**

**public int getOrder() {**

**return 1;**

**}**

**}**

**import org.springframework.context.ApplicationEvent;**

**import org.springframework.context.event.SmartApplicationListener;**

**import org.springframework.stereotype.Component;**

**@Component**

**public class PresentListener implements SmartApplicationListener {**

**PresentService presentService = new PresentService();**

**@Override**

**public boolean supportsEventType(final Class<? extends ApplicationEvent> eventType) {**

**return eventType == RegisterEvent.class;**

**}**

**@Override**

**public boolean supportsSourceType(final Class<?> sourceType) {**

**return sourceType == String.class;**

**}**

**@Override**

**public void onApplicationEvent(final ApplicationEvent event) {**

**System.out.println("EmailListener 接收到的注册用户是：" + event.getSource());**

**presentService.dalibao();**

**}**

**@Override**

**public int getOrder() {**

**return 1;**

**}**

**}**

### 4、spring.xml：

**<?xml version="1.0" encoding="UTF-8"?>**

**<beans xmlns="http://www.springframework.org/schema/beans"**

**xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:context="http://www.springframework.org/schema/context"**

**xsi:schemaLocation="http://www.springframework.org/schema/beans**

**http://www.springframework.org/schema/beans/spring-beans-4.0.xsd http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd">**

**<context:component-scan base-package="com.lidahai.spring"/>**

**</beans>**

### 5、测试：

**public class Test**

**{**

**public static void main(String[] args)**

**{**

**ClassPathXmlApplicationContext applicationContext = new ClassPathXmlApplicationContext("spring.xml");**

**applicationContext.publishEvent(new RegisterEvent("小明"));**

**}**

**}**

# 9、spring+quartz源码解析之Spring 之生命周期机制混合使用

## 1、spring 中bean的生命周期：

实例化

设置依赖

初始化

spring容器进行管理

销毁前处理

销毁处理

三种方式：

## 2、bean的初始化和消亡前主要由以下几种方式：

### 1．Spring的接口InitializingBean和DisposableBean。示例如下：

/\*\*

\* InitializingBean 接口从名字可以看出此方法在 bean的属性被spring赋值之后执行,但是和spring的接口耦合在了一起

\*/

@Override

public void afterPropertiesSet() throws Exception {

System.out.println(this+" i am afterPropertiesSet");

}

/\* \*

\* DisposableBean 接口定义的

\*/

@Override

public void destroy() throws Exception {

System.out.println(this+" i am destroy");

}

### 2．采用元数据配置，

即在bean的定义中配置，如init-method 和destroy-method属性，直接定义相关方法，

另外在根标签可以配置default-init-method 和default-destroy-method 配置默认的 方法，如果中有定义，则覆盖默认的定义：

<bean id="user0" class="com.test.service.UserServiceIml" init-method="testInit" destroy-method="testBeforeDesstroy">

<property name="userDao" ref="userDao"></property>

</bean>

/\*\*

\* @Description: 采用元数据配置，在xml中配置

\* @param

\* @return void

\*/

public void testInit(){

System.out.println(this+" i am testInit");

}

/\*\*

\* @Description: 采用元数据配置，在xml中配置

\* @param

\* @return void

\*/

public void testBeforeDesstroy(){

System.out.println(this+" i am testBeforeDesstroy");

}

### 3．采用JSR注解

/\*\*

\* @Description: 采用jsr注解

\* @param

\* @return void

\*/

@PostConstruct

public void testPostConstruct(){

System.out.println(this+" i am testPostConstruct");

}

/\*\*

\* @Description: 采用JSR的注解

\* @param

\* @return void

\*/

@PreDestroy

public void testPreDesstroy(){

System.out.println(this+" i am testPreDesstroy");

}

## **3、**混合机制：

他们可以单独使用，也可以混合使用，混合使用的时候，他们的顺序如下：

### 1、初始化函数：

1. @PostConstruct 注解的方法

2. InitializingBean接口定义的回调afterPropertiesSet()

3. Bean配置中自定义的初始化函数

### 2、析构

1. @PreDestroy注解的方法

2. DisposableBean接口定义的回调destroy()

3. Bean配置中自定义析构函数

### 3、startUp和shutDown回调

有时候我们可能需要一些组件的功能随着spring容器的启动而启动（如新开启一个线程来进行监听），容器的销毁而销毁。为此spring提供了一个接口，从方法名字很容易看到方法的作用：

public interface Lifecycle {

void start();

void stop();

boolean isRunning();

}

当启动的时候，由于组件之间的依赖，启动的顺序是相当重要的。

depends-on属性可以决定多个lifecycle的实现的顺序，但是有时候依赖是未知的。为此spring定义可一个新的接口，SmartLifecycle:

public interface Phased {

int getPhase();

}

public interface SmartLifecycle extends Lifecycle, Phased {

boolean isAutoStartup();

void stop(Runnable callback);

}

isAutoStartup决定是随着容器的启动自启动，Phased接口定义的方法则决定了启动顺序，其返回值越小启动越早，越大启动越晚。如果一个Lifecycle未实现Phased接口，则默认其值为0.

Stop方法可以异步的执行析构函数，spring默认会为每一个Phase等待30秒钟。这个时间是可以配置的。

在非web环境下，要使析构函数执行，需要执行AbstractApplicationContext的registerShutdownHook()方法。

### 4、spring扩展点

Bean的生命周期spring是采用beanpostProcessor(称之为spring扩展点)来进行管理和扩展的，spring定义了很多内置的beanpostProcessor，它们的一些回调在bean的实例化（这里指返回客户端调用前，而不是new）过程中被调用，且如果bean实现了一些aware接口（如ApplicationContextAware 和 beanNameAwre

），这些接口的回调也会进入到生命周期中

而且，spring广泛采用代理机制，需要注意的是 生命周期函数运行在代理创建之前

### 5、一个bean的管理过程

1. Call Constructor 调用构造函数.(如果没有配置constructor 属性，调用默认 无参数的构造函数(可以是private/protected/public))

2. 调用配置过的所有setter. (通过<protperty name="xxx" value=xxxx)

3. 调用Aware, 如果你实现了BeanNameAware/BeanFactoryAware/ApplicaitonContextAware接口的话

    3.1 调用setBeanName

    3.2 调用setBeanFactory

    3.3 调用setApplicationContext

4. 调用postProcessBeforeInitialization方法,如果你有一个实现了BeanPostProcessor接口的bean的话

5. 调用@PostConstruct， 如果你引用了CommonAnnotationBeanPostProcessor

6. 调用afterPropertiesSet, 如果你的bean实现了InitializingBean接口

7. 调用init-method，配置在spring配置文件中.

8. 调用postProcessAfterInitialization,如果你有一个实现了BeanPostProcessor接口的bean的话

9. 调用@PreDestroy， 如果你引用了CommonAnnotationBeanPostProcessor 并且注册的shutDownHook或者是一个被托管的容器

10.调用destory-method,配置在spring配置文件中.

### 6、代码：

public class LifecycleTest {  
   
 /\*\*  
  \* @param args  
  \*/  
 public static void main(String[] args) {  
  // TODO Auto-generated method stub  
  String[] beanFiles = new String[]{"lifecycle/beans.xml"};  
  ApplicationContext appCxt = new ClassPathXmlApplicationContext(beanFiles);  
  ((AbstractApplicationContext)appCxt).registerShutdownHook();  
  //(ClassPathXmlApplicationContext)appCxt)  
 }

}

public class TestBean implements BeanNameAware, BeanFactoryAware,  
  ApplicationContextAware, InitializingBean {

 private TestBean() {  
  System.out.println("Construct");  
 }

 @PostConstruct  
 public void postConstruct() {  
  System.out  
    .println("@PostConstruct annotation called, org.springframework.context.annotation.CommonAnnotationBeanPostProcessor registor requried.");  
 }

 @PreDestroy  
 public void preDestroy() {  
  System.out  
    .println("@PreDestroy annotation called, org.springframework.context.annotation.CommonAnnotationBeanPostProcessor registor requried.");  
 }

 public void initMethod() {  
  System.out.println("  configed init Method in spring file called");  
 }

 public void destroyMethod() {  
  System.out.println("  configed destory Method in spring file called");  
 }

 private String name;

 public String getName() {  
  System.out.println("getName - " + name);  
  return name;  
 }

 public void setName(String name) {  
  System.out.println("setName - " + name);  
  this.name = name;  
 }

 @Override  
 public void afterPropertiesSet() throws Exception {  
  System.out  
    .println("afterPropertiesSet called, bean need implements InitializingBean");  
 }

 ApplicationContext applicationContext;

 @Override  
 public void setApplicationContext(ApplicationContext applicationContext)  
   throws BeansException {  
  this.applicationContext = applicationContext;  
  System.out  
    .println(" setApplicationContext called, bean need implements ApplicationContextAware");  
 }

 BeanFactory beanFactory;

 @Override  
 public void setBeanFactory(BeanFactory beanFactory) throws BeansException {  
  this.beanFactory = beanFactory;  
  System.out  
    .println(" setBeanFactory called, bean need implements setBeanFactoryAware");  
 }

 String beanName;

 @Override  
 public void setBeanName(String beanName) {  
  this.beanName = beanName;  
  System.out  
    .println(" setBeanName called, bean need implements setBeanNameAware");

 }

}

public class TraceBeanPostProcessor implements BeanPostProcessor {

 @Override  
 public Object postProcessAfterInitialization(Object bean, String beanName)  
   throws BeansException {  
  System.out.println("  postProcessAfterInitialization --");  
  return bean;  
 }

 @Override  
 public Object postProcessBeforeInitialization(Object bean, String beanName)  
   throws BeansException {  
  System.out.println("  postProcessBeforeInitialization --" + "Bean '"  
    + beanName + "' created : " + bean.toString());  
  return bean;  
 }  
}

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>4.2.6.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>4.2.6.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aop</artifactId>

<version>4.2.6.RELEASE</version>

</dependency>

<dependency>

<groupId>commons-logging</groupId>

<artifactId>commons-logging</artifactId>

<version>1.2</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-web</artifactId>

<version>4.2.6.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context-support</artifactId>

<version>4.2.6.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-tx</artifactId>

<version>4.2.6.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jdbc</artifactId>

<version>4.2.6.RELEASE</version>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>servlet-api</artifactId>

<version>LATEST</version>

</dependency>

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns:context=*"http://www.springframework.org/schema/context"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-4.0.xsd http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd"*>

<bean id=*"CommonAnnotationBeanPostProcessor"*

class=*"org.springframework.context.annotation.CommonAnnotationBeanPostProcessor"* />

<bean id=*"testBean"* class=*"lifecycle.TestBean"* init-method=*"initMethod"*

destroy-method=*"destroyMethod"*>

<property name=*"name"* value=*"test"*></property>

</bean>

<bean id=*"beanPostProcessor"* class=*"lifecycle.TraceBeanPostProcessor"*>

</bean>

</beans>

# 10、Spring+quatz源码解析