定时任务

# Timer

## Timer

### **schedule方法，四个**

1.schedule(TimerTask task, Date time)：定时在time的时间时执行。

2.schedule(TimerTask task, Date firstTime, long period)：定时在time的时间时执行，并每隔period时间执行一次。

3.schedule(TimerTask task, long delay)：延时delay的时间执行

4.schedule(TimerTask task, long delay, long period)：延时delay的时间执行，并每隔period时间执行一次。

### scheduleAtFixedRate方法

scheduleAtFixedRate(TimerTask task, Date firstTime, long period):时间等于或超过time时首次执行task，之后每隔period毫秒重复执行一次task。

scheduleAtFixedRate(TimerTask task, long delay, long period)

### **schedule与**scheduleAtFixedRate的区别

#### 两种情况：

##### 1.首次计划执行的时间早于当前的时间

###### schedule方法

“fixed-delay”；如果第一次执行时间被delay了，随后的执行时间按照上一次实际执行完成的时间点进行计算

###### scheduleAtFixedRate方法

“fixed-rate”；如果第一次执行时间被delay了，随后的执行时间按照上一次开始的时间点进行计算，并且为了赶上进度会多次执行任务，因此TImerTask中的执行体需要考虑同步。

##### 2.任务执行所需时间超出任务的执行周期间隔

###### schedule方法

下一次执行时间相对于上一次实际执行完成的时间点，因此执行时间会不断延后。

###### scheduleAtFixedRate方法

下一次执行时间相对于上一次开始的时间点，因此执行时间一般不会延后，因此存在并发性。

### 其他函数

cancel()：终止此计时器，丢弃所有当前已安排的任务。

purge()：作用，从此计时器的任务队列中移除所有已取消的任务

返回值，从队列中移除的任务数

## TimerTask

### cancel方法

作用：取消当前TimerTask里的任务。

### scheduledExecutionTime方法

作用：返回此任务最近实际执行的已安排执行的时间

返回值：最近发生此任务执行安排的时间，为long型

## Timer的缺陷

### 管理并发任务的缺陷

Timer有且仅有一个线程去执行定时任务，如果存在多个任务，且任务时间过长，会导致执行效果与预期不符

### 当任务抛出异常时的缺陷

如果TimerTask抛出RuntimeException，Timer会停止所有任务的执行。

## Timer的使用禁区

### 对时效性要求较高的多任务并发作业

### 对复杂的任务的调度

# Quartz

## **简介**

### **特点**

强大的调度功能，灵活的应用方式，分布式和集群能力。

### **主要用到的设计模式**

Builder模式，Factory模式，组件模式，链式写法

### **三个核心概念**

调度器scheduler，任务jobDetail，触发器trigger

### 重要组成

Job，JobDetail，JobBuilder，JobStore，Trigger，TriggerBuilder，ThreadPool，Scheduler，

Calendar：一个Trigger可以和多个Calendar关联，以排除或包含某些时间点。

监听器：JobListener，TriggerListener，SchedulerLIstener

## 浅谈Job&JobDetail

### **重要属性**

name，group，jobClass，jobDataMap

### 例子

**public class** HelloJob **implements** Job {  
  
 **private** String **message**;  
 **private** Float **floatJobValue**;  
 **private** Double **doubleTriggerValue**;  
  
 **public** String getMessage() {  
 **return message**;  
 }  
  
 **public void** setMessage(String message) {  
 **this**.**message** = message;  
 }  
  
 **public** Float getFloatJobValue() {  
 **return floatJobValue**;  
 }  
  
 **public void** setFloatJobValue(Float floatJobValue) {  
 **this**.**floatJobValue** = floatJobValue;  
 }  
  
 **public** Double getDoubleTriggerValue() {  
 **return doubleTriggerValue**;  
 }  
  
 **public void** setDoubleTriggerValue(Double doubleTriggerValue) {  
 **this**.**doubleTriggerValue** = doubleTriggerValue;  
 }  
  
 @Override  
 **public void** execute(JobExecutionContext jobExecutionContext) **throws** JobExecutionException {  
 SimpleDateFormat simpleDateFormat = **new** SimpleDateFormat(**"yyyy-MM-dd HH:mm:ss"**);  
 System.***out***.println(**"current exec time is: "** + simpleDateFormat.format(**new** Date()));  
*// JobKey key = jobExecutionContext.getJobDetail().getKey();  
// TriggerKey triggerKey = jobExecutionContext.getTrigger().getKey();  
// JobDataMap jobDataMap = jobExecutionContext.getJobDetail().getJobDataMap();  
// JobDataMap triggerJobDataMap = jobExecutionContext.getTrigger().getJobDataMap();  
//  
// System.out.println("my job name is: " + key.getName()  
// + "; my job group is: " + key.getGroup());  
// System.out.println("my trigger name is: " + triggerKey.getName()  
// + "; my trigger group is: " + triggerKey.getGroup());  
// System.out.println("job message is: " + jobDataMap.get("message")  
// + "; and job floatJobValue is: " + jobDataMap.get("floatJobValue"));  
// System.out.println("trigger message is: " + triggerJobDataMap.get("message")  
// + "; and job doubleTriggerValue is: " + triggerJobDataMap.get("doubleTriggerValue"));  
//  
// //job和trigger实例合并之后的map，key相同trigger会覆盖job的数据  
// JobDataMap mergedJobDataMap = jobExecutionContext.getMergedJobDataMap();  
// System.out.println("message is: " + message);  
// System.out.println("floatJobValue is: " + floatJobValue);  
// System.out.println("doubleTriggerValue is: " + doubleTriggerValue);* }  
}

## 浅谈JobExecutionContext

### jobExecutionContext是什么

1.当Scheduler调用一个Job，就会讲JobExecutionContext传递给Job的execute()方法；

2.Job能通过JobExecutionContext对象访问到Quartz运行时候的环境以及Job本身的明细数据。

## 浅谈JobDataMap

### JobDataMap是什么

1. 在进行任务调度时JobDataMap存储在JobExecutionContext中，非常方便获取
2. JobDataMap可以用来装载任何可序列化的数据对象，当job实例对象被执行时这些参数对象会传递给它。
3. JobDataMap实现了JDK的Map接口，并且添加了一些非常方便的方法用来存取基本数据类型。

### 获取JobDataMap的两种方式

1. 从Map中直接获取
2. Job实现类中添加setter方法对应JobDataMap的键值(Quartz框架默认的JobFactory实现类在初始化job实例对象时会自动地调用这些setter方法)

## 浅谈Trigger

### Quartz框架中的Trigger

接口Trigger <- 抽象类AbstractTrigger <- 四个实现类(DailyTimeIntervalTriggerImpl, CalendarIntervalTriggerImpl, CronTriggerImpl, SimpleTriggerImpl)

### 触发器通用属性

JobKey：表示job实例的标识，触发器被触发时，该制定的job实例会执行

StartTime：表示触发器的时间表首次被触发的时间，它的值的类型是java.util.Date

EndTime：指定触发器的不再被触发的时间，它的值的类型是java.util.Date

### SimpleTrigger

#### 作用

在一个指定时间段内执行一次作业任务；

或是在指定的时间间隔内多次执行作业任务

#### 需要注意的点

重复次数可以为0，正整数或是SimpleTrigger.REPEAT\_INDEFINITELY常量值

重复执行间隔必须为0或长整数

一旦被指定了endTime参数，那么他会覆盖重复次数参数的效果

#### 例子

**public class** HelloScheduler {  
  
 **public static void** main(String[] args) **throws** SchedulerException {  
 JobDetail jobDetail = JobBuilder.*newJob*(HelloJob.**class**)  
 .withIdentity(**"myJob"**, **"group1"**)  
*// .usingJobData("message", "hello myJob")  
// .usingJobData("floatJobValue", 3.14f)* .build();  
  
 Date now = **new** Date();  
 Date startTime = **new** Date(now.getTime() + 2000);  
 Date endTime = **new** Date(now.getTime() + 8000);  
 SimpleTrigger trigger = TriggerBuilder.*newTrigger*()  
 .withIdentity(**"myTrigger"**, **"group1"**)  
*// .usingJobData("message", "hello myTrigger")  
// .usingJobData("doubleTriggerValue", 2.0)* .startAt(startTime)  
 .endAt(endTime)  
 .withSchedule(SimpleScheduleBuilder  
 .*simpleSchedule*()  
 .withIntervalInSeconds(2)  
 .withRepeatCount(2))  
 .build();  
*// System.out.println("jobDetail's name: " + jobDetail.getKey().getName());  
// System.out.println("jobDetail's group: " + jobDetail.getKey().getGroup());  
// System.out.println("jobDetail's jobClass: " + jobDetail.getJobClass().getName());* StdSchedulerFactory schedulerFactory = **new** StdSchedulerFactory();  
 Scheduler scheduler = schedulerFactory.getScheduler();  
 scheduler.start();  
  
 SimpleDateFormat simpleDateFormat = **new** SimpleDateFormat(**"yyyy-MM-dd HH:mm:ss"**);  
 System.***out***.println(**"current time is: "** + simpleDateFormat.format(**new** Date()));  
 scheduler.scheduleJob(jobDetail, trigger);  
 }  
  
}

### CronTrigger

#### 作用

基于日历的作业调度器，而不是像SimpleTrigger那样精确指定间隔时间，比SimpleTrigger更常用。

#### Cron表达式

用于配置CronTrigger实例

是由7个子表达式组成的字符串，描述了时间表的详细信息。格式：[秒][分][时][日][月][周][年]

#### Cron表达式举例

0 15 10 ? \* \*:每天10点15分触发

0 0/5 14 \* \* ?:每天下午的2点到2点59分(整点开始，每隔5分中触发)

0 15 10 ? \* MON-FRI:从周一到周五每天上午的10点15分触发

0 15 10 ? \* 6#3:每月的第三周的星期五开始触发

0 15 10 ? \* 6L 2016-2017:从2016年到2017年每月最后一周的星期五的10点15分触发

|  |  |
| --- | --- |
| 字符 | 含义 |
| \* | 表示所有值。例如：在分的字段上设置\*，表示每一分钟都会触发 |
| ? | 表示不指定值。使用场景为不需要关心当前设置这个字段的值 |
| - | 表示区间。例如在小时上设置“10-12”，表示10，11，12点都会触发 |
| , | 表示指定多个值。例如，MON,WED,FRI |
| / | 用于递增触发。例如，在秒上5/15，表示从5秒开始，每增15秒触发一次 |
| L | 表示最后的意思 |
| W | 表示离指定日期的最近那个工作日 |
| # | 序号(表示每月的第几个周几) |

具体参见：<https://blog.csdn.net/li295214001/article/details/52065634>

Cron表达式小提示：L和W可以同时使用，可以使用在线Cron表达式生成器

#### 代码例子

**public class** HelloScheduler {  
  
 **public static void** main(String[] args) **throws** SchedulerException {  
 JobDetail jobDetail = JobBuilder.*newJob*(HelloJob.**class**)  
 .withIdentity(**"myJob"**, **"group1"**)  
 .build();  
  
 CronTrigger trigger = TriggerBuilder.*newTrigger*()  
 .withIdentity(**"myTrigger"**, **"group1"**)  
 .withSchedule(CronScheduleBuilder.*cronSchedule*(**""**))  
 .build();  
 *//1.2017年内每天10点15分触发一次 -> 0 15 10 ? \* \* 2017  
 //2.每天的14点整至14点59分55秒，以及18点至18点59分55秒，每5秒触发一次 -> 0/5 \* 14,18 \* \* ?* StdSchedulerFactory schedulerFactory = **new** StdSchedulerFactory();  
 Scheduler scheduler = schedulerFactory.getScheduler();  
 scheduler.start();  
  
 SimpleDateFormat simpleDateFormat = **new** SimpleDateFormat(**"yyyy-MM-dd HH:mm:ss"**);  
 System.***out***.println(**"current time is: "** + simpleDateFormat.format(**new** Date()));  
 scheduler.scheduleJob(jobDetail, trigger);  
 }  
}

## 浅谈Scheduler

Scheduler的创建方式：

SchedulerFactory factory = new StdSchedulerFactory();

Scheduler scheduler = factory.getScheduler();

DirectSchedulerFactory factory = DirectSchedulerFactory.getInstance();

Scheduler scheduler = factory.getScheduler();

### StdSchedulerFactory

使用一组参数(java.util.Properties)来创建和初始化Quartz调度器。

配置参数一般存储在quartz.properties中。

调用getScheduler方法就能创建和初始化调度器对象。

### Scheduler的主要函数

Date schedulerJob(JobDetail jobDetail, Trigger trigger)

void start()

void standby() //scheduler挂起

void shutdown() // 关闭scheduler,不能重启scheduler

shutdown(true) 表示等待所有正在执行的job执行完毕之后,再关闭scheduler

shutdown(false)表示直接关闭scheduler

### quartz.properties

文档的位置和加载顺序

组成部分:调度器属性,线程池属性,作业存储设置,插件配置

#### 调度器属性

org.quartz.scheduler.instanceName属性用来区分特定的调度器实例,可以按照功能用途来给调度器起名.

org.quartz.scheduler.instanceId属性和前者一样,也允许任何字符串,但这个值必须是在所有调度器实例中是唯一的,尤其是在一个集群当中,作为集群的唯一key.加入你想Quartz帮你生成这个值的话,可以设置为AUTO.

#### 线程池属性

threadCount

threadPriority

org.quartz.threadPool.class

#### 作业存储设置

描述了在调度器实例的生命周期中,Job和Trigger信息是如何被存储的.

#### 插件配置

满足特定需求用到的Quartz插件的配置.

## 使用Quartz配置作业(spring + Quartz)

### 两种方式

MethodInvokingJobDetailFactoryBean

JobDetailFactoryBean