# 概念

Quartz是由Java开发的一个开源的任务日程管理系统，任务进度管理器就是一个在预先确实的时间到达时，负责执行（或者通知）其他软件组件的系统。Quartz用一个小Java库发布文件（.jar文件），这个库文件包含了所有Quartz核心功能。这些功能的主要接口是Scheduler接口。它提供了简单的操作，例如：将任务纳入日程或者从日程中取消，开始/停止/暂停日程进度。

# 特点

（1）强大的调度功能

（2）灵活的应用方式

（3）分布式和集群能力

# 适用场景

1、定时发送信息

2、定时生成报表

3、自动更新静态数据

4、自动结账等。

# 主要用到的设计模式

（1）Builder模式

（2）组件模式

（3）Factory模式

（4）链式写法

# 三个核心概念

（1）调度器（Scheduler）

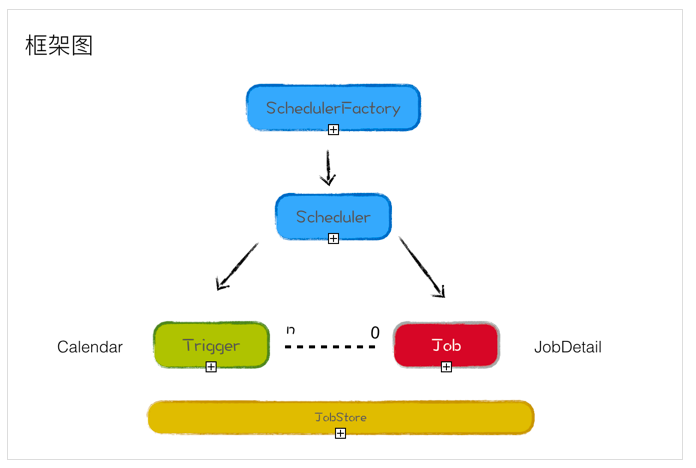
（2）任务（Job）

（3）触发器（Trigger）

JobDetail：描述任务的具体信息。

Trigger:分为SimpleTrigger和CronTrigger

# 架构



# 重要组成

（1）Job

（2）JobDetail

（3）JobBuilder

（4）JobStore

（5）Trigger

（6）TriggerBuilder

（7）ThreadPool

（8）Scheduler

（9）Calendar

一个Trigger可以和多个Calendar关联，以排除或包含某些时间点。

（10）JobListener

（11）TriggerListener

（12）SchedulerListener

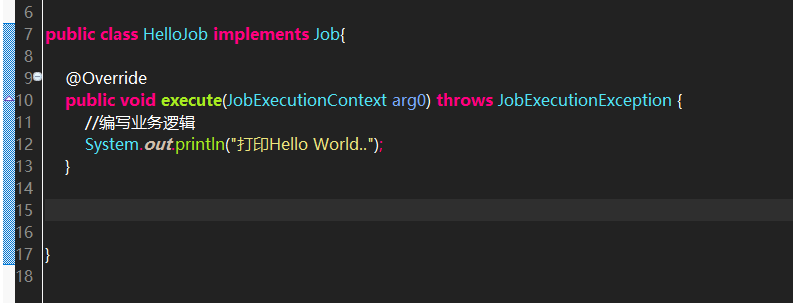
# 入门示例

说明：每2秒打印一个“Hello World”

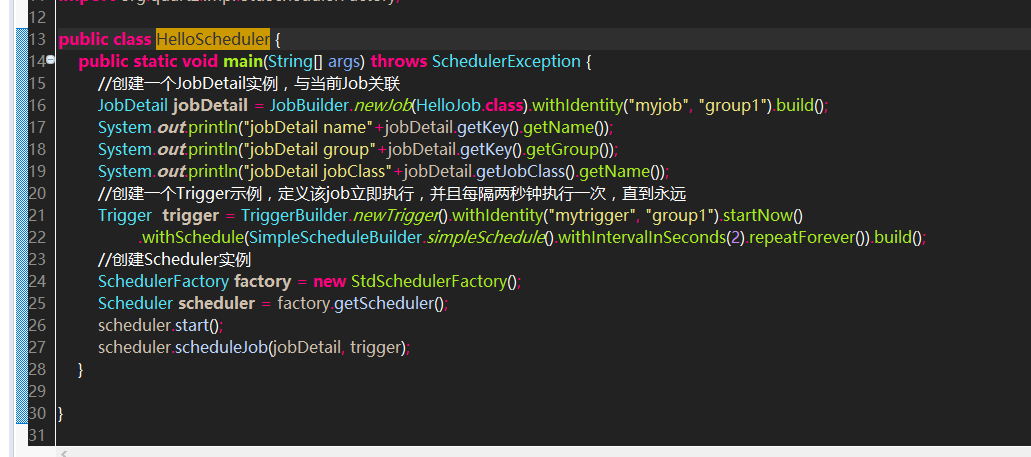
我们需要引入quartz的jar包。

## （1）创建一个Job类

创建一个HelloJob的类，实现Job接口，然后重写execute()方法。

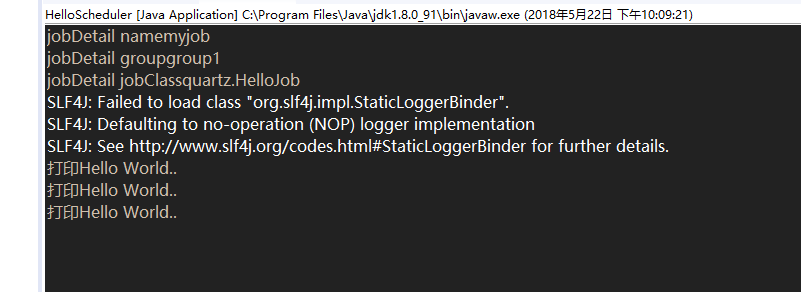


## （2）创建HelloScheduler类



说明：代码思路是先创建JobDetial实例，然后创建Trigger，再创建Scheduler实例，把JobDetail与Trigger关联起来。

## （3）运行

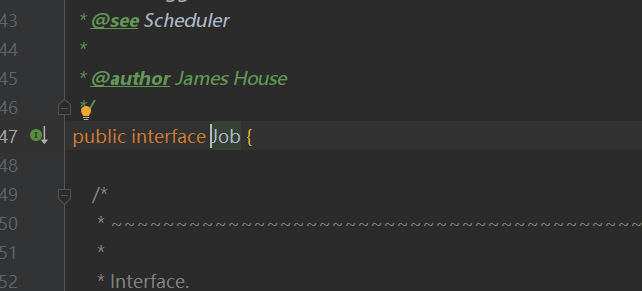


# Job接口

## 基本概念

实现业务逻辑的任务接口，Job接口非常容易实现，只有一个方法execute方法，在里面编写业务逻辑。

## 源代码



## 方法

### void execute(JobExecutionContext context)

## Job实例在Quartz中的生命周期

每次调度器执行job时，它在调用execute方法前会创建一个新的job实例，当调用完成后，关联的job对象实例会被释放，释放的实例会被垃圾回收机制回收。

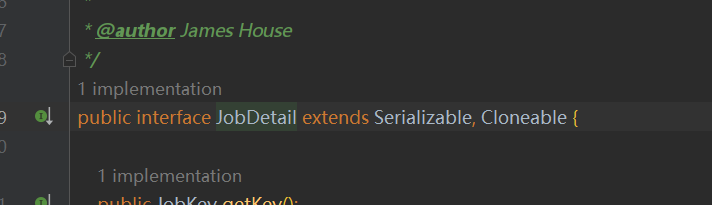
# JobDetail接口

## 基本概念

JobDetail为Job实例提供了许多设置属性，以及JobDataMap成员变量属性，它用来存储特定Job实例的状态信息，调度器需要借助JobDetail对象来添加Job实例，代码如下：

JobDetail jobDetail = JobBuilder.newJob(HelloJob.class).build();

## 源码



## 方法

### （1）JobKey getKey();

### （2）Class<? extends Job> getJobClass();

### （3）JobDataMap getJobDataMap();

### （4）JobBuilder getJobBuilder();

## 重要属性

### name

任务的名称

### group

任务所在的组

### jobClass

任务的实现类

### jobDataMap

作为参数传入

说明：这几个属性很重要，在记录日志的时候会使用。

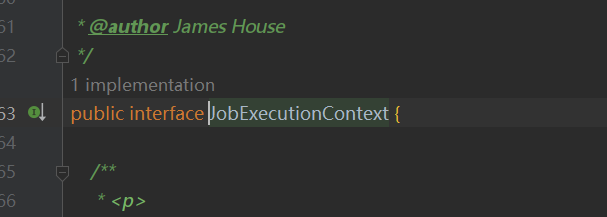
# JobBuilder类

# JobExecutionContext接口

## 基本概念

这是一个接口，当Scheduler调用一个Job，就会将JobExecutionContext传递给Job的execute()方法，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。

## 源码



## 方法

### （1）Scheduler getScheduler();

### （2）Trigger getTrigger();

### （3）Calendar getCalendar()

### （4）JobDetail getJobDetail()

### （5）void put(Object key, Object value);

### （6）Object get(Object key);

### （7）JobDataMap getMergedJobDataMap();

# JobDataMap类

## 基本概念

在进行任务调度时JobDataMap存储在JobExecutionContext接口中，非常方便获取，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。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储基本数据类型。

## 源码



## 获取JobDataMap的方法

### （1）从Map中直接获取

代码示例：

这是第一部分代码：

//通过获取JobDetail对象来获取JobDataMap对象

JobDataMap dataMap = context.getJobDetail().getJobDataMap();

String str = dataMap.getString("message");

System.out.println(str);

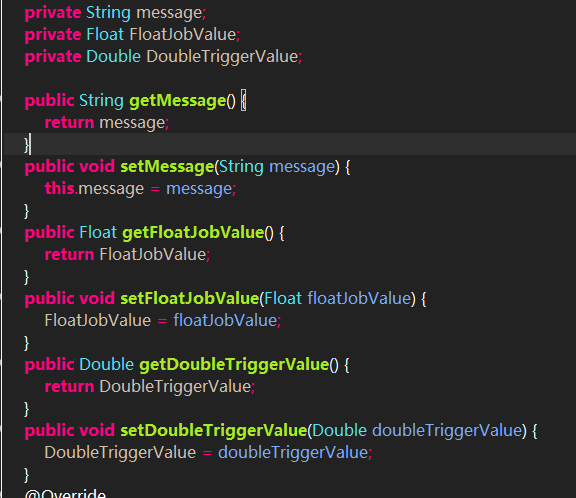
这是第二部分代码：

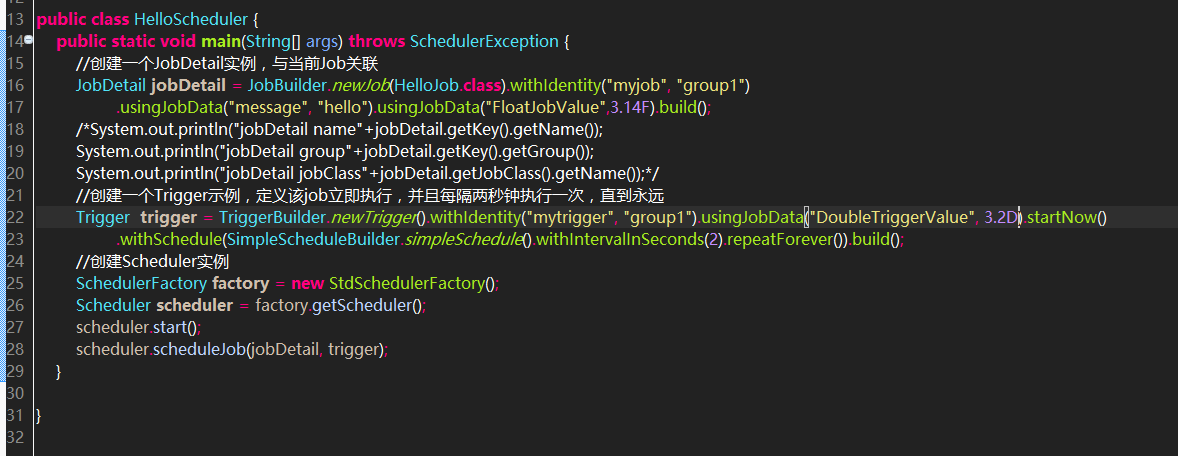


### （2）用成员变量去获取

定义同名的成员变量，然后生成get、set方法去获取

代码示例：



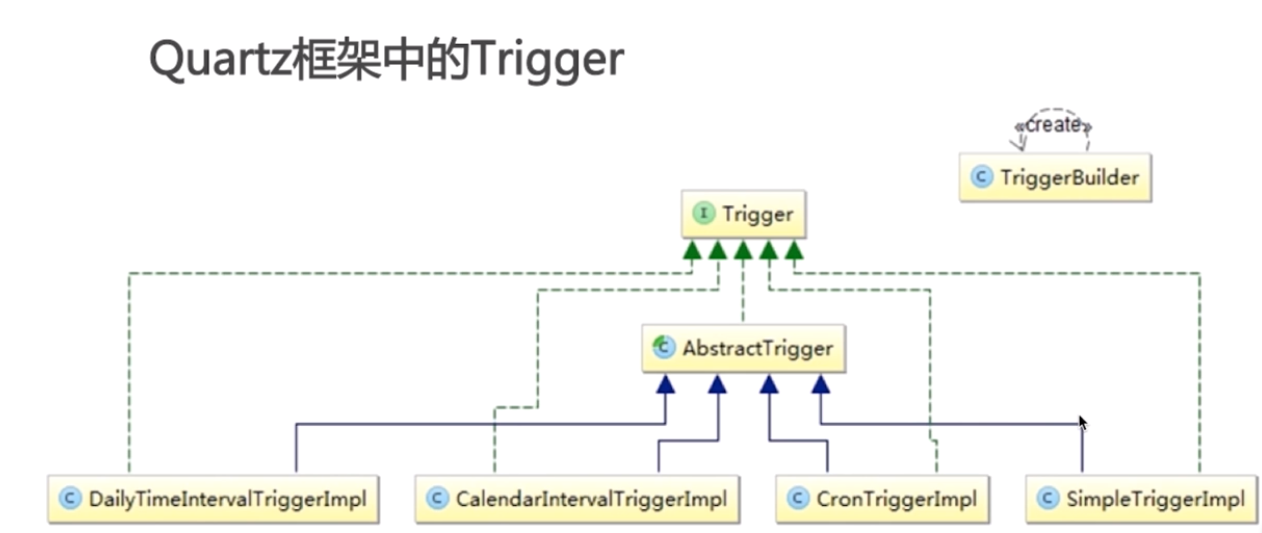


# QuartzJobBean类

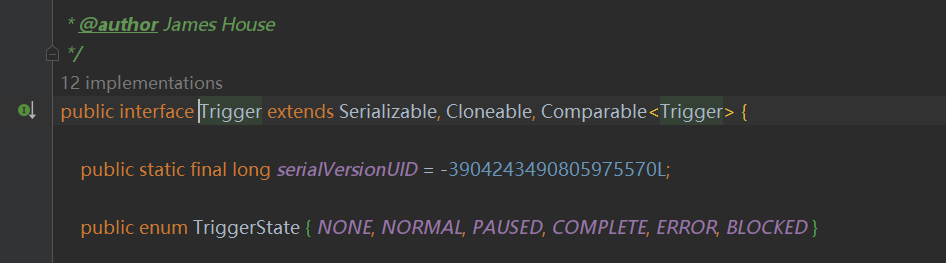
# Trigger接口

## 基本概念

Trigger是一个接口，它的作用是在规定的时间触发Job。Trigger是使用Builder模式去创建的。Trigger常见的实现类有SimpleTriggerImpl、CronTriggerImpl。结构如下图：



## 源码



## 触发器的通用属性

### （1）JobKey

表示Job实例的标识，触发器被触发时，该指定的job实例会实现。

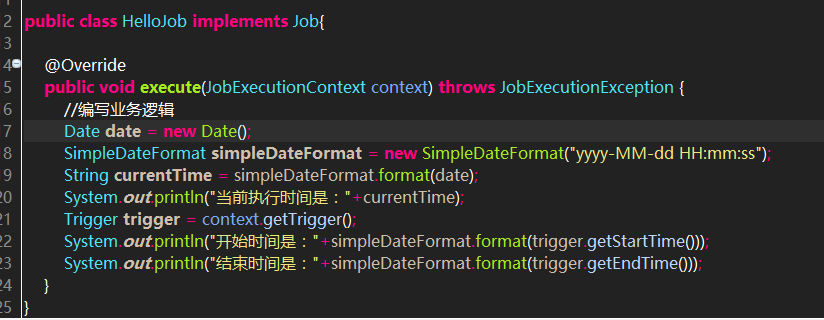
### （2）StartTime

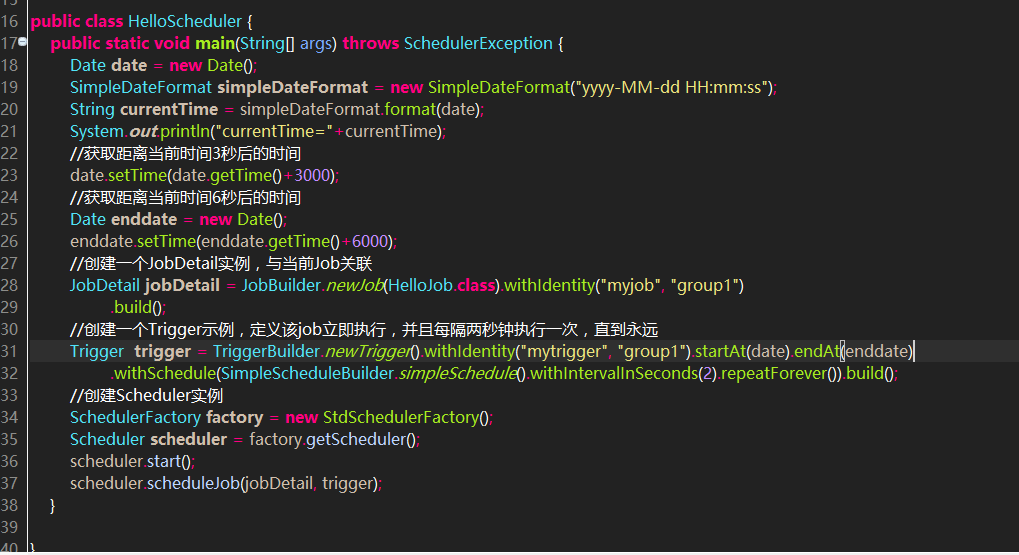
表示触发器的时间，首次被触发的时间。它的值的类型是java.util.Date。

### （3）EndTime

指定触发器的不再被触发的时间，它的值类型是java.util.Date。

### 代码示例





## 常用的方法

### （1）String getCalendarName()

### （2）JobDataMap getJobDataMap()

### （3）Date getStartTime();

### （4）Date getEndTime();

### （5）TriggerBuilder<? extends Trigger> getTriggerBuilder();

### （6）ScheduleBuilder<? extends Trigger> getScheduleBuilder();

## SimpleTrigger接口

### 基本概念

SimpleTrigger是一个接口，继承自Trigger。

### 常用的方法

#### int getRepeatCount();

#### long getRepeatInterval();

#### int getTimesTriggered()

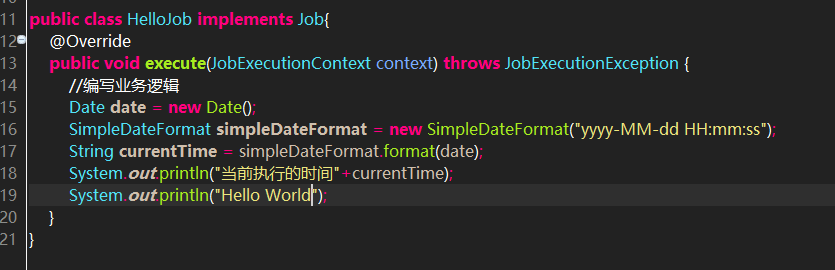
#### TriggerBuilder<SimpleTrigger> getTriggerBuilder();

### SimpleTrigger的作用

在一个指定的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。

### 示例代码

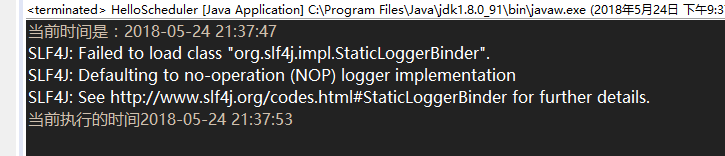
距离当前时间6秒之后执行Job





使用的方法是：startAt();

输出结果：



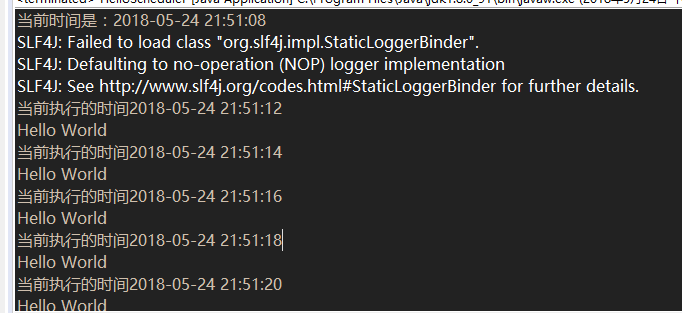
示例：距离当期时间4秒后首次执行任务，之后每隔2秒重复执行一次任务

代码：

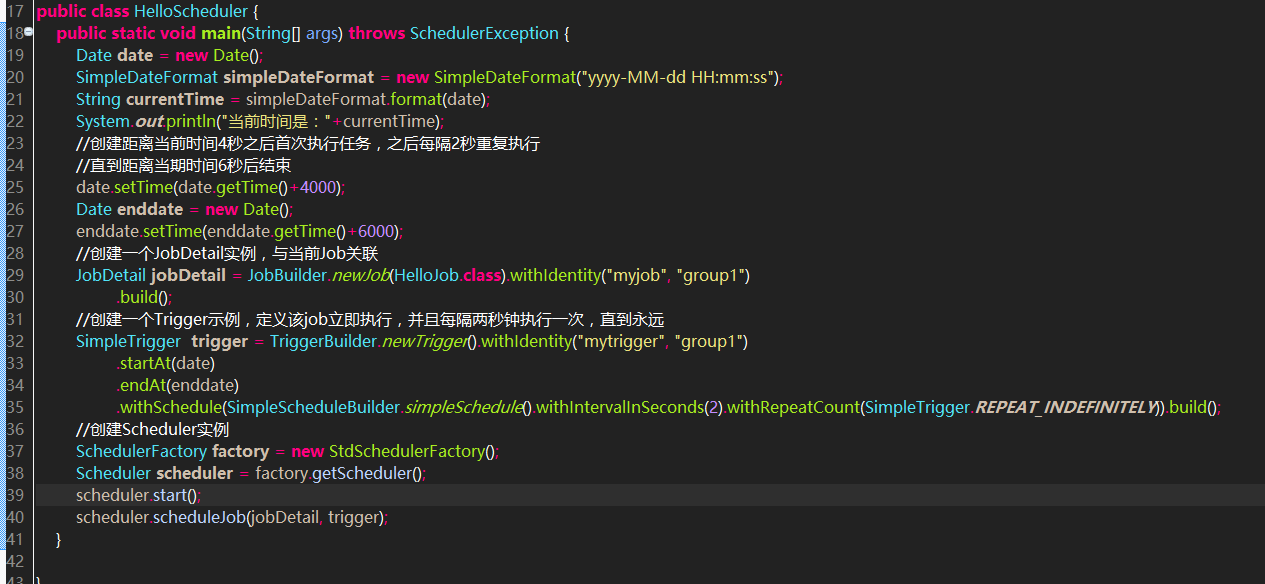


使用的方法是：startAt() withInIntervalInSeconds()

输出的结果：



示例：创建距离当前时间4秒之后首次执行任务，之后每隔2秒重复执行，直到距离当期时间6秒后结束。



使用的方法：startAt() endAt()withInIntervalInSeconds()

需要注意的是：

（1）**重复次数**可以为0、正整数或是SimpleTrigger.REPEAT.INDEFINITELY（无线循环）。

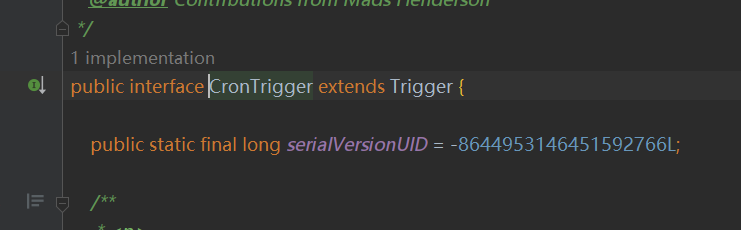
（2）**重复执行间隔**必须为0或为长整数。

（3）一旦指定了endTime参数，那么它会覆盖重复次数参数的效果。

## CronTrigger接口

### 基本概念

### 源码



### 作用

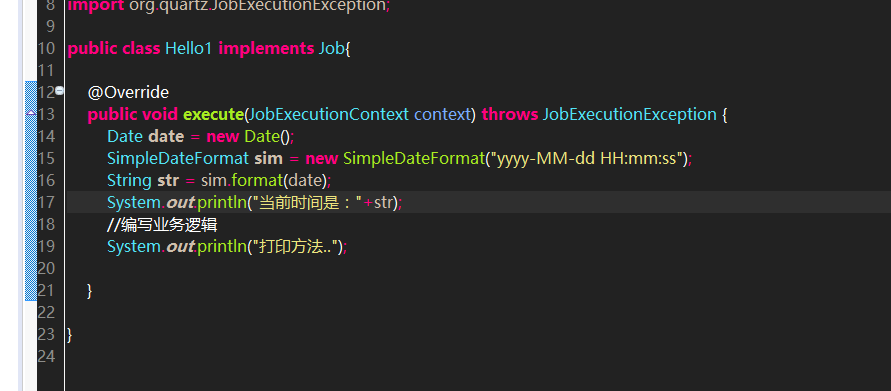
基于日历的作业调度器，而不是像SimpleTrigger那样精确指定时间间隔，比SimpleTrigger更常用。

### Cron表达式

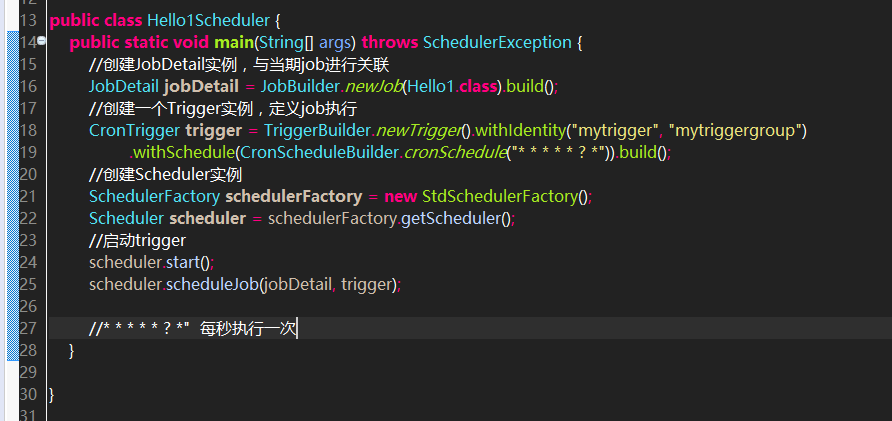
用于配置CronTrigger实例，是由7个子表达式组成的字符串，描述了时间表的详细信息，格式：[秒][分][小时][日][月][周][年]

### 示例代码

首先编写一个类并实现Job接口



然后再编写Hello1Scheduler类



### Cron表达式特殊字符意义对应表



### Cron表达式举例



2017年内每天10点15分触发一次

0 15 10 ？ \* \* 2017

每天的14点至14点59分55秒，以及18点整至18点59分55秒，每5秒钟触发一次

0/5 \* 14,18 \* \* ?

### 通配符说明

\* 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，使用的场景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

/ 用于递增触发

L 表示最后的字段

W 表示离指定日期的最近那个工作日（周一至周五）

# 序号，表示每月的第几个周几

Cron表达式小提示：

（1）L和W可以一起组合使用

（2）周字段英文字母不区分大小写，即MON与mon相同

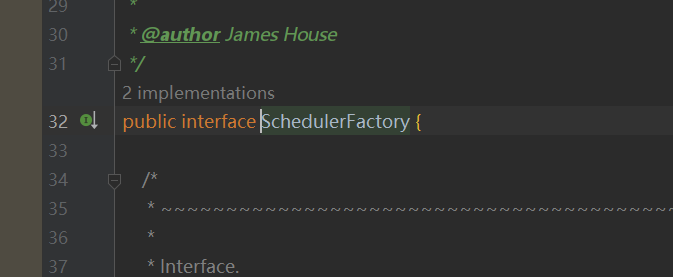
（3）利用工具，在线生成。

http://www.pdtools.net/tools/becron.jsp

# SchedulerFactory接口

## 基本概念

## 源码



## 常用的方法

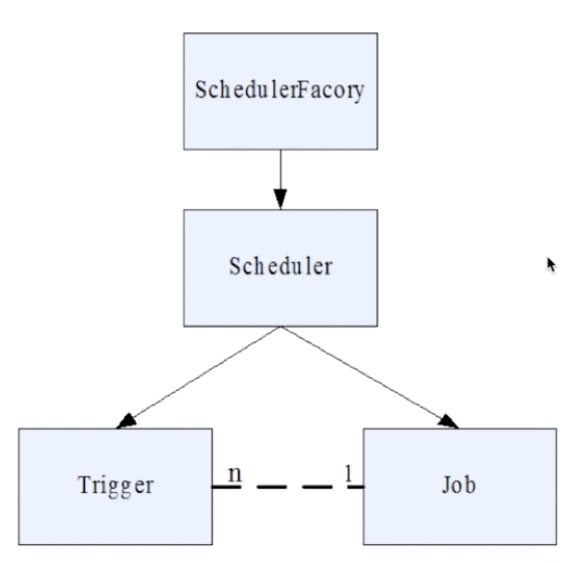
### Scheduler getScheduler()

### Scheduler getScheduler(String schedName)

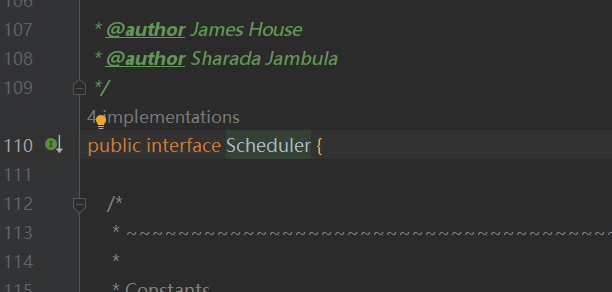
# Scheduler接口

## 概念

所有的Scheduler实例应该由SchedulerFactory来创建的。主要有StdSchedulerFactory和DirectSchedulerFactory这两个类。



## 源码



## 方法

### Date scheduleJob(JobDetail jobDetail,Trigger trigger)

//把trigger和jobDetail关联起来，返回最近的一次执行时间。

### void start()

//开启调度器

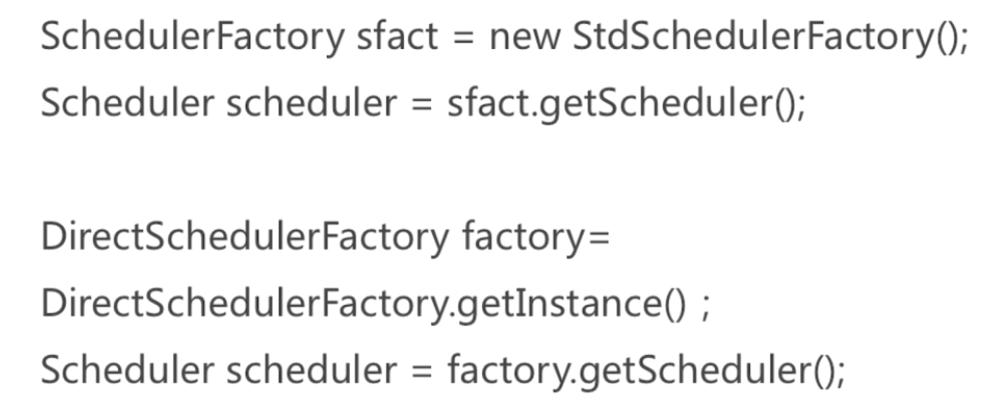
### void standby()

//挂起这个调度器，只是暂时被挂起，可以重新被开启。

### void shutdown()

//将这个scheduler完全关闭，不能再重启。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)即shutdown()，表示直接关闭scheduler。

## Scheduler创建方式



# StdSchedulerFactory类

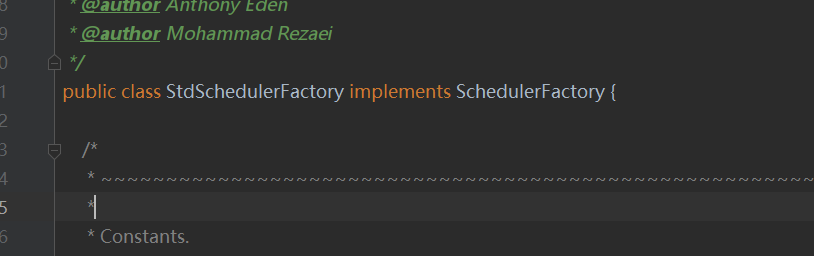
## 基本概念

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

（2）配置参数一般存储在quartz.properties文件中

（3）调用getScheduler方法就能创建和初始化调度器对象。

## 源码



# quartz.properties文件

## 文档的位置和加载顺序

如果项目中没有quartz.properties文件，项目就会自动读取quartz.jar包中的配置文件，jar包中的quartz.properties文件的位置在quartz-xx.jar包中的com.quartz包中。

我们可以在项目中新建quartz.properties文件，然后在文件中编写配置内容，这样项目就会读取我们创建的配置文件。

## 文档的内容

# Default Properties file for use by StdSchedulerFactory

# to create a Quartz Scheduler Instance, if a different

# properties file is not explicitly specified.

#

org.quartz.scheduler.instanceName: DefaultQuartzScheduler

org.quartz.scheduler.rmi.export: false

org.quartz.scheduler.rmi.proxy: false

org.quartz.scheduler.wrapJobExecutionInUserTransaction: false

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.misfireThreshold: 60000

org.quartz.jobStore.class: org.quartz.simpl.RAMJobStore

## 组成部分

（1）调度器属性

（2）线程池属性

（3）作业存储设置

（4）插件配置

### 调度器属性

（1）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。

（2）org.quartz.scheduler.instanceId属性和前者一样，也允许任何字符串，但是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。假如你想Quartz帮你生成这个值的话，可以设置为AUTO。

### 线程池属性

（1）threadCount

设置工作线程的数目

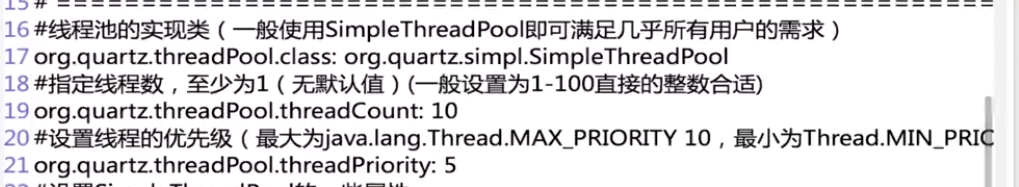
（2）threadPriority

设置工作线程的优先级。它的值范围在1-10，正常值是5。

（3）org.quartz.threadPool.class

线程池的实现类，一般使用SimpleThreadPool即可满足几乎所有用户的需求。

示例：



### 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

满足了特定需求用到的Quartz插件的配置，这个一般在项目中不会用到。

# Spring与Quartz整合

## 使用Quartz配置作业

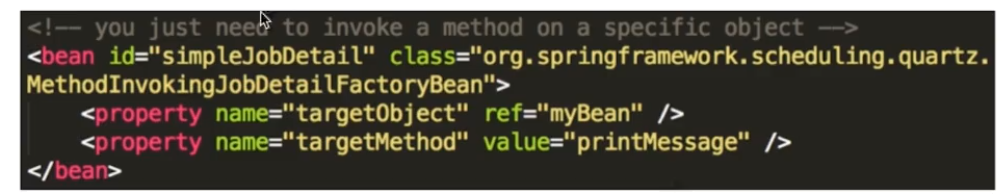
两种方法：

### （1）MethodInvokingJobDetailFactoryBean

代码示例：

在springmvc的配置文件中编写如下代码：

调用myBean的printMessage方法



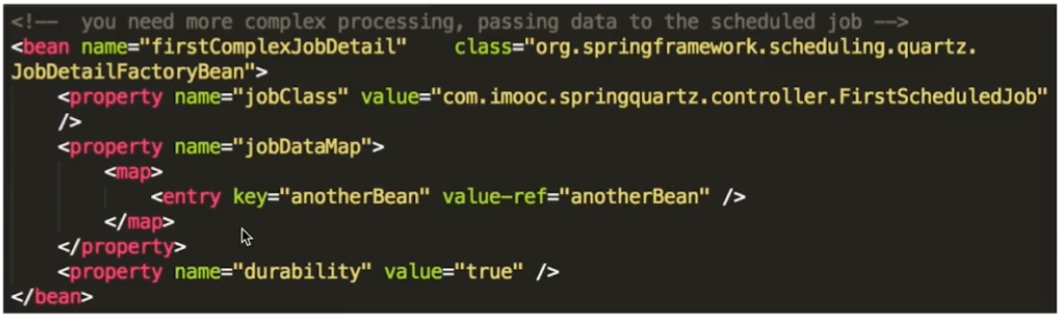
MyBean类



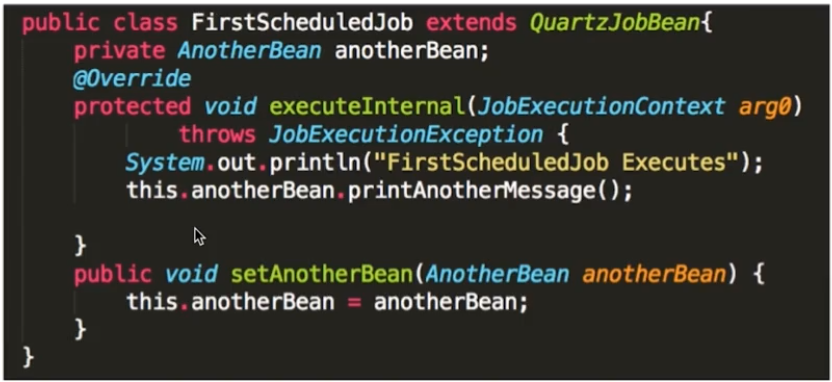
### （2）JobDetailFactoryBean

需要给作业Job传递数据，想更加灵活的话就使用这种方式

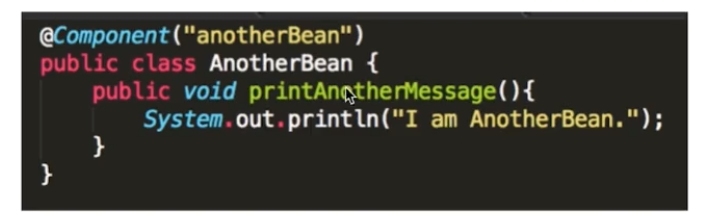
代码示例：



编写FirstScheduledJob类，这个类需要继承QuartzJobBean类



编写AnotherBean类



## 在配置文件中进行配置

代码如下：

<!-- 添加注解驱动 -->

<mvc:annotation-driven />

<!-- 默认扫描的包路径 -->

<context:component-scan base-package="quartz" />

//配置myBean类

<bean id="simpleJobDetail"

class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">

<property name="targetObject" ref="myBean" />

<property name="targetMethod" value="printMessage" />

</bean>

//配置FirstScheduledJob类

<bean id="firstComplexJobDetail"

class="org.springframework.scheduling.quartz.JobDetailFactoryBean">

<property name="jobClass"

value="com.imooc.springquartz.quartz.FirstScheduledJob" />

<property name="jobDataMap">

<map>

<entry key="anotherBean" value-ref="anotherBean" />

</map>

</property>

<property name="Durability" value="true"/>

</bean>

//为simpleJobDetail配置SimpleTrigger

<!-- 距离当前时间1秒之后执行，之后每隔两秒钟执行一次 -->

<bean id="mySimpleTrigger" class="org.springframework.scheduling.quartz.SimpleTriggerFactoryBean">

<property name="jobDetail" ref="simpleJobDetail"/>

<property name="startDelay" value="1000"/>

<property name="repeatInterval" value="2000"/>

</bean>

//为firstComplexJobDetail配置CronTriggerFactoryBean

<!-- 每隔5秒钟执行一次 -->

<bean id="myCronTrigger" class="org.springframework.scheduling.quartz.CronTriggerFactoryBean">

<property name="jobDetail" ref="firstComplexJobDetail"/>

<property name="cronExpression" value="0/5 \* \* ? \* \*"/>

</bean>

//配置Scheduler

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

<property name="jobDetails">

<list>

<ref bean="simpleJobDetail"/>

<ref bean="firstComplexJobDetail"/>

</list>

</property>

<property name="triggers">

<list>

<ref bean="mySimpleTrigger"/>

<ref bean="myCronTrigger"/>

</list>

</property>

</bean>

这就完成了spring与Quartz的结合。

# @Scheduled注解

这个注解用于开启定时调度任务，开发思路如下：

## （1）****在spring mvc的xml文件里加入task的命名空间****

xmlns:task="http://www.springframework.org/schema/task"

http://www.springframework.org/schema/task http://www.springframework.org/schema/task/spring-task-4.1.xsd

## ****（2）启用注解驱动的定时任务****

<task:annotation-driven  />

说明：在spring mvc的配置文件中进行编写。

## ****（3）配置定时任务的线程池****

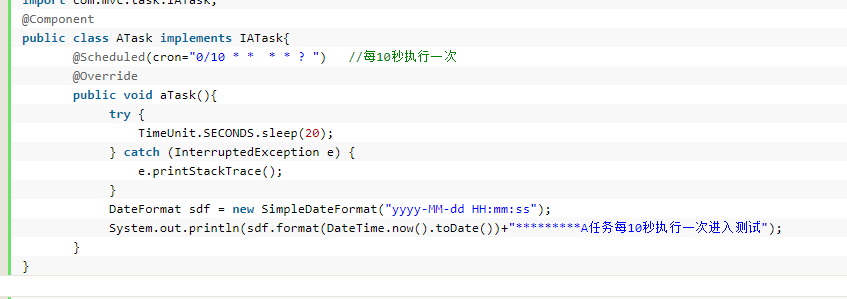
推荐配置线程池，若不配置多任务下会有问题。后面会详细说明单线程的问题。

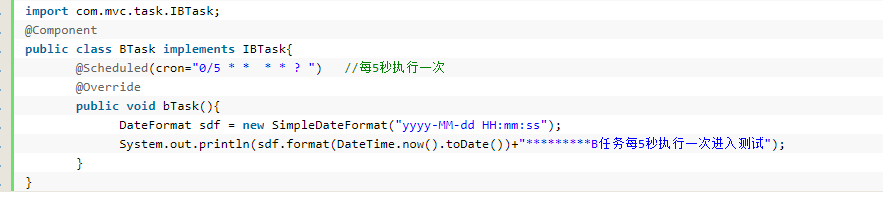
<task:scheduler id="myScheduler" pool-size="5"/>

说明：这步可以省略不要。

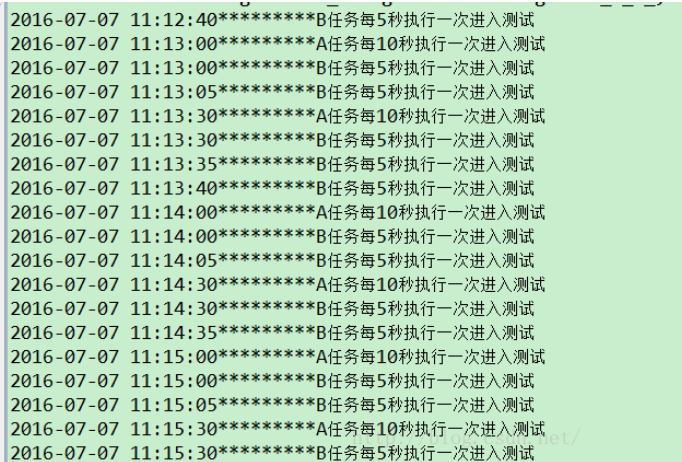
## ****（4）编写定时任务****

@Scheduled注解为定时任务，cron表达式里写执行的时机，代码如下：

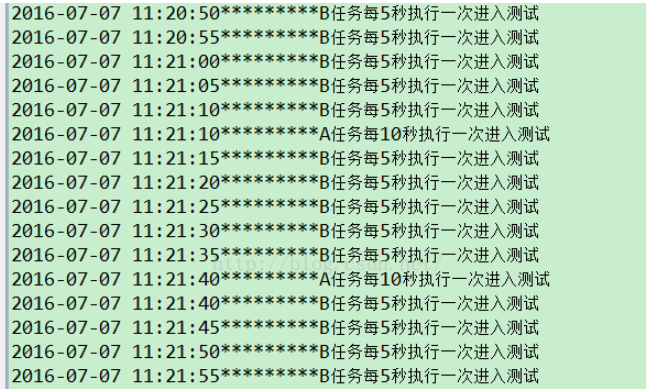




spring的定时任务默认是单线程**，**多个任务执行起来时间会有问题（B任务会因为A任务执行起来需要20S而被延后20S执行），如下图所示：



当我们配置了线程池后再来看结果（多线程下，B任务再也不会因为A任务执行起来要20S而被延后了）



# 问题

## 1、Only no-arg methods may be annotated with @Scheduled

解决方法：@Scheduled修改的方法不能有参数，把参数去掉就可以了。