**Java 定时任务调度--Spring Scheduled 介绍**

Spring 提供了@Scheduled 注解，可用比较便捷的解决定时任务的需求，它的内部实现是基于 java 中的 ScheduledThreadPoolExecutor 类。本文主要介绍 Spring Boot 环境下 @Scheduled 的使用；

软件版本：Spring Boot 2.4.4、jdk1.8.0\_181。

**1、@Scheduled简介**

@Scheduled 注解标注在方法上，可以支持如下几种方式运行：  
1.@Scheduled(fixedRate = 3000) 按固定时间间隔运行  
2.@Scheduled(fixedDelay = 3000) 上一次任务运行完成后等待固定时间运行下一次任务  
3.@Scheduled(cron = "0 0/1 \* \* \* ?") 按照 cron 表达式定义的时间方式运行

**2、Spring Boot 环境下 @Scheduled 使用**

**2.1、pom.xml配置**

如果是基于maven的项目，在pom.xml进行如下配置：

**[pom.xml]**

<?xml version="1.0" encoding="UTF-8"?>  
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">  
 <modelVersion>4.0.0</modelVersion>  
  
 <parent>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-parent</artifactId>  
 <version>2.3.5.RELEASE</version>  
 <relativePath/> *<!-- lookup parent from repository -->* </parent>  
  
 <groupId>com.example</groupId>  
 <artifactId>demo</artifactId>  
 <version>0.0.1-SNAPSHOT</version>  
 <name>demo</name>  
 <description>Demo project for Spring Boot</description>  
  
 <properties>  
 <java.version>1.8</java.version>  
 </properties>  
  
 <dependencies>  
  
 *<!--springboot支持-->* <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
 </dependency>  
 *<!-- lombok支持-->* <dependency>  
 <groupId>org.projectlombok</groupId>  
 <artifactId>lombok</artifactId>  
 <optional>true</optional>  
 </dependency>  
  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-test</artifactId>  
 <scope>test</scope>  
 <exclusions>  
 <exclusion>  
 <groupId>org.junit.vintage</groupId>  
 <artifactId>junit-vintage-engine</artifactId>  
 </exclusion>  
 </exclusions>  
 </dependency>  
 </dependencies>  
  
 <build>  
 <plugins>  
 <plugin>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-maven-plugin</artifactId>  
 </plugin>  
 </plugins>  
 </build>  
  
</project>

**2.2、@EnableScheduling 注解**

在启动类上添加

**[java]**

@EnableScheduling  
@Slf4j  
@SpringBootApplication(scanBasePackages = "com.example.demo")  
public class DemoApplication {  
 public static void main(String[] args) {  
 SpringApplication.run(DemoApplication.class, args);  
 }  
}

或者新建配置类并添加

**[java]**

@Configuration  
@ComponentScan("com.example.demo")  
@EnableScheduling *//1.通过@EnableScheduling注解开启对计划任务的支持*public class TaskSchedulerConfig {  
  
}

**2.3、设置任务线程池大小**

默认任务线程池大小为 1，如想修改，可在 application.yml 中添加如下配置：

**[application.yml]**

spring:  
 task:  
 scheduling:  
 pool:  
 size: 10

**2.4、使用例子**

**[java]**

package com.example.demo;  
  
import lombok.extern.slf4j.Slf4j;  
import org.springframework.scheduling.annotation.Scheduled;  
import org.springframework.stereotype.Component;  
  
*//计划任务执行类*@Slf4j  
@Component *//@Component注解类并注册到spring容器中*public class ScheduledTask {  
  
 @Scheduled(cron = "0 0/1 \* \* \* ?")  
 private void test1() {  
 ***log***.info("cron表达式");  
 }  
  
 @Scheduled(initialDelay = 2000, fixedRate = 3000)  
 private void test2() {  
 ***log***.info("延迟2s运行第一次，然后每隔3s运行一次");  
 }  
  
 @Scheduled(initialDelay = 3000, fixedDelay = 4000)  
 private void test3() {  
 ***log***.info("延迟3s运行第一次，上一次任务运行结束后等4s再运行下一次任务。");  
 }  
  
}

**2.5、动态启动、停止定时任务**

假设需要在页面动态的添加并保存定时任务到数据库，添加完成后可以在页面启动或停止该任务；可以编写一个 Controller 来实现该功能。

task线程实体类

**[java]**

@Data  
@AllArgsConstructor  
public class TaskVO {  
 */\*\*主键\*/* private Integer id;  
  
 */\*\*业务参数\*/* private String bussinessParam;  
  
 */\*\*cron表达式\*/* private String cron;  
}

业务逻辑类

**[java]**

package com.example.demo;  
  
import lombok.extern.slf4j.Slf4j;  
  
@Slf4j  
public class TaskRunnable implements Runnable {  
 private String param;  
 public TaskRunnable(String param) {  
 this.param = param;  
 }  
  
 @Override  
 public void run() {  
 ***log***.info("TaskRunnable:{}", param);  
 }  
}

控制器类

**[java]**

package com.example.demo.controller;  
  
import com.example.demo.TaskRunnable;  
import com.example.demo.vo.TaskVO;  
import lombok.extern.slf4j.Slf4j;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.scheduling.TaskScheduler;  
import org.springframework.scheduling.support.CronTrigger;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.RestController;  
  
import javax.annotation.PostConstruct;  
import java.util.HashMap;  
import java.util.Map;  
import java.util.concurrent.ConcurrentHashMap;  
import java.util.concurrent.ScheduledFuture;  
  
  
*/\*\*  
 \* 这里只涉及任务的启动，停止，重启  
 \*  
 \*/*@Slf4j  
@RequestMapping("/schedule")  
@RestController  
public class ScheduleController {  
  
 private static Map<Integer, ScheduledFuture> futures = new ConcurrentHashMap<>();  
  
 *//模拟数据库的任务配置数据* private Map<Integer, TaskVO> taskVOMap = new HashMap(){{  
 put(1, new TaskVO(1, "任务1", "0/5 \* \* \* \* \*"));  
 put(2, new TaskVO(2, "任务2", "0/6 \* \* \* \* \*"));  
 put(3, new TaskVO(3, "任务3", "0/7 \* \* \* \* \*"));  
 put(4, new TaskVO(4, "任务4", "0/8 \* \* \* \* \*"));  
 put(5, new TaskVO(5, "任务5", "0/9 \* \* \* \* \*"));  
 }};  
  
 @Autowired  
 private TaskScheduler taskScheduler;  
  
 */\*\*程序启动时，启动所有任务\*/* @PostConstruct  
 private void init() {  
 for (Map.Entry<Integer, TaskVO> entry : taskVOMap.entrySet()) {  
 TaskVO taskVO = entry.getValue();  
 ScheduledFuture<?> future = taskScheduler.schedule(  
 new TaskRunnable(taskVO.getBussinessParam()),   
 new CronTrigger(taskVO.getCron())  
 );  
 futures.put(entry.getKey(), future);  
 }  
 }  
  
 *//http://localhost:8080/schedule/startTask?taskId=1* @RequestMapping("/startTask")  
 public String startTask(Integer taskId) {  
 ScheduledFuture<?> future = futures.get(taskId);  
 if (future == null) {  
 TaskVO taskVO = taskVOMap.get(taskId);  
 future = taskScheduler.schedule(  
 new TaskRunnable(taskVO.getBussinessParam()),   
 new CronTrigger(taskVO.getCron())  
 );  
 futures.put(taskId, future);  
 ***log***.info("启动任务:"+taskId);  
 } else {  
 ***log***.info("任务已启动");  
 }  
 return "startTask Success:"+taskId;  
 }  
  
 *//http://localhost:8080/schedule/stopTask?taskId=1* @RequestMapping("/stopTask")  
 public String stopTask(Integer taskId) {  
 ScheduledFuture<?> future = futures.get(taskId);  
 if (future != null) {  
 future.cancel(true);  
 futures.remove(taskId);  
 ***log***.info("停止任务:"+taskId);  
 } else {  
 ***log***.info("任务已停止");  
 }  
 return "stopTask Success:"+taskId;  
 }  
  
 *//http://localhost:8080/schedule/restartTask?taskId=1* @RequestMapping("/restartTask")  
 public String restartTask(Integer taskId) {  
 ScheduledFuture<?> future = futures.get(taskId);  
 if (future == null) {  
 TaskVO taskVO = taskVOMap.get(taskId);  
 future = taskScheduler.schedule(  
 new TaskRunnable(taskVO.getBussinessParam()),   
 new CronTrigger(taskVO.getCron())  
 );  
 futures.put(taskId, future);  
 ***log***.info("创建任务:"+taskId);  
 } else {  
 future.cancel(true);  
 TaskVO taskVO = taskVOMap.get(taskId);  
 future = taskScheduler.schedule(  
 new TaskRunnable(taskVO.getBussinessParam()),   
 new CronTrigger(taskVO.getCron())  
 );  
 futures.put(taskId, future);  
 ***log***.info("重启任务:"+taskId);  
 }  
 return "restartTask Success:"+taskId;  
 }  
}

**3、cron 表达式简介**

cron表达式是一个字符串，字符串分为6或7个字段：

秒 分 小时 月份中的日期 月份 星期中的日期 [年份]

各字段含义：

|  |  |  |
| --- | --- | --- |
| 字段 | 允许值 | 允许的特殊字符 |
| 秒（Seconds） | 0~59 的整数 | , - \* / |
| 分（Minutes） | 0~59 的整数 | , - \* / |
| 小时（Hours） | 0~23 的整数 | , - \* / |
| 日期（DayofMonth） | 1~31 的整数（但是你需要考虑你月的天数） | , - \* ? / L W C |
| 月份（Month） | 1~12 的整数或者 JAN-DEC | , - \* / |
| 星期（DayofWeek） | 1~7 的整数或者 SUN-SAT（1=SUN） | , - \* ? / L C # |
| 年(可选)（Year） | 1970~2099 | , - \* / |

cron 表达式中特殊符号的含义:

|  |  |
| --- | --- |
| \* | 表示匹配该域的任意值。如在 Minutes 域使用 '\*', 即表示每分钟都会触发事件。 |
| ? | 只能用在 DayofMonth 和 DayofWeek 两个域，表示这个位置的值不确定。DayofMonth 和 DayofWeek 是两个相互排斥的域，需把其中一个域设为 '?'。如 1 月 10 日是星期一，如果在 DayofWeek 的位置另指定星期二，就前后冲突矛盾了。 |
| - | 表示范围。如在 Minutes 域使用 '5-20'，表示从 5 分到 20 分钟每分钟触发一次 |
| / | 表示起始时间开始触发，然后每隔固定时间触发一次。例如在 Minutes 域使用 '5/20'，则意味着 5 分钟触发一次，而 25，45 等分别触发一次. |
| , | 表示列出枚举值。例如：在 Minutes 域使用 '5,20'，则意味着在 5 和 20 分别触发一次。 |
| L | 表示最后，只能出现在 DayofWeek 和 DayofMonth 域。在 DayofMonth 域中表示一个月的最后一天，在 DayofWeek 域表示 7 或者 SAT，如果在 DayofWeek 域中前面加上数字，它表示一个月的最后一个星期几，例如 '6L' 就表示一个月的最后一个星期五。 |
| W | 只能出现在 DayofMonth 域，用于指定日期的最近工作日(周一到周五)。例如在日期域使用 '15W'，表示这个月 15 号最近的工作日；所以，如果 15 号是周六，则任务会在 14 号触发；如果 15 号是周日，则任务会在周一也就是 16 号触发。如果在日期域填写 1W，即使 1 号是周六，那么任务也只会在下周一，也就是 3 号触发，'W' 指定的最近工作日是不能够跨月的。'W' 只能配合一个单独的数值使用，不能够是一个数字段，例如 '1-15W' 是错误的。 |
| LW | 'L' 和 'W' 可以在日期域中联合使用，表示在某个月最后一个工作日，即最后一个星期五。 |
| # | 用于确定每个月第几个星期几，只能出现在 DayofWeek 域。例如 '4#2'，表示某月的第二个星期三。 |

cron 表达式例子

'0 0 12 \* \* ?' 每天中午12点触发  
'0 15 10 ? \* \*' 每天上午10:15触发  
'0 15 10 \* \* ?' 每天上午10:15触发  
'0 15 10 \* \* ? \*' 每天上午10:15触发  
'0 15 10 \* \* ? 2005' 2005年的每天上午 10:15触发  
'0 \* 14 \* \* ?' 每天下午2点到下午 2:59 期间的每1分钟触发  
'0 0/5 14 \* \* ?' 每天下午2点到下午 2:55 期间的每5分钟触发  
'0 0/5 14,18 \* \* ?' 每天下午2点到 2:55 期间和下午6点到6:55期间的每5分钟触发  
'0 0-5 14 \* \* ?' 每天下午2点到下午 2:05 期间的每1分钟触发  
'0 10,44 14 ? 3 WED' 每年三月的星期三的下午2:10和2:44触发  
'0 15 10 ? \* MON-FRI' 周一至周五的上午 10:15触发  
'0 15 10 15 \* ?' 每月15日上午10:15触发  
'0 15 10 L \* ?' 每月最后一日的上午10:15触发  
'0 15 10 ? \* 6L' 每月的最后一个星期五上午10:15触发  
'0 15 10 ? \* 6L 2002-2005' 2002年至2005年的每月的最后一个星期五上午10:15触发  
'0 15 10 ? \* 6#3' 每月的第三个星期五上午10:15触发

参考文章：<https://www.cnblogs.com/wuyongyin/p/14721248.html>