第六章 Spring Boot 如何集成定时任务、异步调用

1. Spring Boot 使用定时任务@Scheduled-fixedRate方式

在项目开发中，经常需要定时任务来帮助我们来做一些内容，比如定时发送短信/站内信息、数据汇总统计、业务监控等。

1.1 创建定时任务

在Spring Boot中编写定时任务是非常简单的事，下面通过实例介绍如何在Spring Boot中创建定时任务

- pom 配置(只需要引入 Spring Boot Starter jar包即可，Spring Boot Starter 包中已经内置了定时的方法。)

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
</dependency>
```

- 在Spring Boot的主类中加入`@EnableScheduling`注解，启用定时任务的配置

```
package com.yingxue.lesson;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.scheduling.annotation.EnableScheduling;

@SpringBootApplication
@EnableScheduling
public class SpringbootSchedulingApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringbootSchedulingApplication.class, args);
    }

}
```

- 创建定时任务实现类

```
package com.yingxue.lesson.task;

import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;

import java.text.SimpleDateFormat;
import java.util.Date;

/**
 * @ClassName: SchedulerTask
 * TODO:类文件简单描述
 * @Author: 小霍
 * @UpdateUser: 小霍
 * @Version: 0.0.1
 */
@Component
public class SchedulerTask {
    private static final SimpleDateFormat dateFormat = new SimpleDateFormat("HH:mm:ss");

    @Scheduled(fixedRate = 5000)
    private void processFixedRate(){
        System.out.println("processFixedRate方式：定时任务开始运行,现在时间：" + dateFormat.format(new
Date()));
    }
}
```

- 运行程序，控制台中可以看到类似如下输出，定时任务开始正常运作了。

```
processFixedRate方式：定时任务开始运行,现在时间：10:58:11
processFixedRate方式：定时任务开始运行,现在时间：10:58:16
processFixedRate方式：定时任务开始运行,现在时间：10:58:21
processFixedRate方式：定时任务开始运行,现在时间：10:58:26
processFixedRate方式：定时任务开始运行,现在时间：10:58:31
```

1.2 参数说明

在上面的入门例子中，使用了`@Scheduled(fixedRate = 5000)`注解来定义每过5秒执行的任务，对于`@Scheduled`的使用可以总结如下几种方式：

fixedRate 说明

- `@Scheduled(fixedRate = 5000)`：上一次开始执行时间点之后5秒再执行
- `@Scheduled(fixedDelay = 5000)`：上一次执行完毕时间点之后5秒再执行
- `@Scheduled(initialDelay=1000, fixedRate=5000)`：第一次延迟1秒后执行，之后按fixedRate的规则每5秒执行一次

2. Spring Boot 使用定时任务@Scheduled-cron 方式

2.1 修改 SchedulerTask

```
package com.yingxue.lesson.task;

import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;

import java.text.SimpleDateFormat;
import java.util.Date;

/**
 * @ClassName: SchedulerTask
 * TODO:类文件简单描述
 * @Author: 小霍
 * @UpdateUser: 小霍
 * @Version: 0.0.1
 */
@Component
public class SchedulerTask {
    private static final SimpleDateFormat dateFormat = new SimpleDateFormat("HH:mm:ss");

    @Scheduled(cron="*/5 * * * * ?")
    private void processCron(){
        System.out.println("processCron方式：定时任务开始运行,现在时间：" + dateFormat.format(new Date()));
    }

//    @Scheduled(fixedRate = 5000)
//    private void processFixedRate(){
//        System.out.println("processFixedRate方式：定时任务开始运行,现在时间：" + dateFormat.format(new
Date()));
//    }
}
```

运行程序，控制台中可以看到类似如下输出，定时任务开始正常运作了。

```
processCron方式：定时任务开始运行,现在时间：11:11:30
processCron方式：定时任务开始运行,现在时间：11:11:35
processCron方式：定时任务开始运行,现在时间：11:11:40
processCron方式：定时任务开始运行,现在时间：11:11:45
processCron方式：定时任务开始运行,现在时间：11:11:50
```

2.2 参数说明

cron 一共有七位，最后一位是年，Spring Boot 定时方案中只需要设置六位即可：

第一位，表示秒，取值 0 ~ 59； 第二位，表示分，取值 0 ~ 59； 第三位，表示小时，取值 0 ~ 23； 第四位，日期天/日，取值 1 ~ 31； 第五位，日期月份，取值 1~12； 第六位，星期，取值 1 ~ 7，星期一，星期二...，注，1 表示星期 天，2 表示星期一； 第七位，年份，可以留空，取值 1970 ~ 2099。 cron 中，还有一些特殊的符号，含义如下：（*）星号，可以理解为每的意思，每秒、每分、每天、每月、每年...。（?）问号，问号只能出现在日期和星期这两个位置，表示这个位置的值不确定。（-）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“1、2、

4”，则表示星期一、星期二、星期四。（/）斜杠，如 x/y，x 是开始值，y 是步长，比如在第一位（秒），0/15 就是从 0 秒开始，每隔 15 秒执 行一次。 下面列举几个常用的例子。 0 0 1 * * ？：每天凌晨1 点执行； 0 5 1 * * ?：每天 凌晨1 点 5 分执行；

以上就是 Spring Boot 自定的定时方案，使用起来非常的简单方便。

3. Spring Boot 使用@Async 实现异步调用

什么是“异步调用”？

“异步调用”对应的是“同步调用”，*同步调用*指程序按照定义顺序依次执行，每一行程序都必须等待上一行程序执行完成之后才能执行；*异步调用*指程序在顺序执行时，不等待异步调用的语句返回结果就执行后面的程序。

3.1 同步调用

下面通过一个简单示例来直观的理解什么是同步调用：

- 定义Task类，创建三个处理函数分别模拟三个执行任务的操作，操作消耗时间随机取（10秒内）

```
package com.yingxue.lesson.task;

import org.springframework.stereotype.Component;

import java.util.Random;

/**
 * @ClassName: MyTask
 * TODO:类文件简单描述
 * @Author: 小霍
 * @UpdateUser: 小霍
 * @Version: 0.0.1
 */
@Component
public class MyTask {

    public static Random random =new Random();

    public void doTaskOne() throws Exception {
        System.out.println("开始做任务一");
        long start = System.currentTimeMillis();
        Thread.sleep(random.nextInt(10000));
        long end = System.currentTimeMillis();
        System.out.println("完成任务一，耗时：" + (end - start) + "毫秒");
    }

    public void doTaskTwo() throws Exception {
        System.out.println("开始做任务二");
        long start = System.currentTimeMillis();
        Thread.sleep(random.nextInt(10000));
        long end = System.currentTimeMillis();
        System.out.println("完成任务二，耗时：" + (end - start) + "毫秒");
    }

    public void doTaskThree() throws Exception {
        System.out.println("开始做任务三");
        long start = System.currentTimeMillis();
        Thread.sleep(random.nextInt(10000));
        long end = System.currentTimeMillis();
        System.out.println("完成任务三，耗时：" + (end - start) + "毫秒");
    }

}
```

- 在单元测试用例中，注入Task对象，并在测试用例中执行`doTaskOne`、`doTaskTwo`、`doTaskThree`三个函数。

```
package com.yingxue.lesson;

import com.yingxue.lesson.task.MyTask;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;
```

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class SpringbootAsyncApplicationTests {

    @Test
    public void contextLoads() {
    }

    @Autowired
    private MyTask myTask;

    @Test
    public void testTask() throws Exception{
        myTask.doTaskOne();
        myTask.doTaskTwo();
        myTask.doTaskThree();
    }

}
```

- 执行单元测试，可以看到类似如下输出：

```
开始做任务一
完成任务一，耗时：8653毫秒
开始做任务二
完成任务二，耗时：5215毫秒
开始做任务三
完成任务三，耗时：648毫秒
```

任务一、任务二、任务三顺序的执行完了，换言之 `doTaskOne` 、 `doTaskTwo` 、 `doTaskThree` 三个函数顺序的执行完成。

3.2 异步调用

上述的同步调用虽然顺利的执行完了三个任务，但是可以看到执行时间比较长，若这三个任务本身之间不存在依赖关系，可以并发执行的话，同步调用在执行效率方面就比较差，可以考虑通过异步调用的方式来并发执行。

在Spring Boot中，我们只需要通过使用 `@Async` 注解就能简单的将原来的同步函数变为异步函数，Task类改在为如下模式：

```
package com.yingxue.lesson.task;

import org.springframework.scheduling.annotation.Async;
import org.springframework.stereotype.Component;

import java.util.Random;

/**
 * @ClassName: MyTask
 * TODO:类文件简单描述
 * @Author: 小霍
 * @UpdateUser: 小霍
 * @Version: 0.0.1
 */
@Component
public class MyTask {

    public static Random random =new Random();

    @Async
    public void doTaskOne() throws Exception {
        System.out.println("开始做任务一");
        long start = System.currentTimeMillis();
        Thread.sleep(random.nextInt(10000));
        long end = System.currentTimeMillis();
        System.out.println("完成任务一，耗时：" + (end - start) + "毫秒");
    }
    @Async
    public void doTaskTwo() throws Exception {
        System.out.println("开始做任务二");
        long start = System.currentTimeMillis();
        Thread.sleep(random.nextInt(10000));
        long end = System.currentTimeMillis();
```

```
        System.out.println("完成任务二，耗时：" + (end - start) + "毫秒");
    }
    @Async
    public void doTaskThree() throws Exception {
        System.out.println("开始做任务三");
        long start = System.currentTimeMillis();
        Thread.sleep(random.nextInt(10000));
        long end = System.currentTimeMillis();
        System.out.println("完成任务三，耗时：" + (end - start) + "毫秒");
    }

}
```

为了让@Async注解能够生效，还需要在Spring Boot的主程序中配置@EnableAsync，如下所示：

```
package com.yingxue.lesson;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.scheduling.annotation.EnableAsync;

@SpringBootApplication
@EnableAsync
public class SpringbootAsyncApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringbootAsyncApplication.class, args);
    }

}
```

此时可以反复执行单元测试，您可能会遇到各种不同的结果，比如：

- 没有任何任务相关的输出
- 有部分任务相关的输出
- 乱序的任务相关的输出

原因是目前 `doTaskOne`、`doTaskTwo`、`doTaskThree` 三个函数的时候已经是异步执行了。主程序在异步调用之后，主程序并不会理会这三个函数是否执行完成了，由于没有其他需要执行的内容，所以程序就自动结束了，导致了不完整或是没有输出任务相关内容的情况。

注： @Async所修饰的函数不要定义为static类型，这样异步调用不会生效

4. Spring Boot 使用@Async 实现异步调用-异步回调结果

为了让 `doTaskOne`、`doTaskTwo`、`doTaskThree` 能正常结束，假设我们需要统计一下三个任务并发执行共耗时多少，这就需要等到上述三个函数都完成调动之后记录时间，并计算结果。

那么我们如何判断上述三个异步调用是否已经执行完成呢？我们需要使用 `Future<T>` 来返回异步调用的结果，改造完成后如下：

```
package com.yingxue.lesson.task;

import org.springframework.scheduling.annotation.Async;
import org.springframework.scheduling.annotation.AsyncResult;
import org.springframework.stereotype.Component;

import java.util.Random;
import java.util.concurrent.Future;

/**
 * @ClassName: MyTask
 * TODO:类文件简单描述
 * @Author: 小霍
 * @UpdateUser: 小霍
 * @Version: 0.0.1
 */
@Component
public class MyTask {

    public static Random random =new Random();
```

```
    @Async
    public Future<String> doTaskOne() throws Exception {
        System.out.println("开始做任务一");
        long start = System.currentTimeMillis();
        Thread.sleep(random.nextInt(10000));
        long end = System.currentTimeMillis();
        System.out.println("完成任务一，耗时：" + (end - start) + "毫秒");
        return new AsyncResult<>("完成任务一");
    }
    @Async
    public Future<String> doTaskTwo() throws Exception {
        System.out.println("开始做任务二");
        long start = System.currentTimeMillis();
        Thread.sleep(random.nextInt(10000));
        long end = System.currentTimeMillis();
        System.out.println("完成任务二，耗时：" + (end - start) + "毫秒");
        return new AsyncResult<>("完成任务二");
    }
    @Async
    public Future<String> doTaskThree() throws Exception {
        System.out.println("开始做任务三");
        long start = System.currentTimeMillis();
        Thread.sleep(random.nextInt(10000));
        long end = System.currentTimeMillis();
        System.out.println("完成任务三，耗时：" + (end - start) + "毫秒");
        return new AsyncResult<>("完成任务三");
    }

}
```

下面我们改造一下测试用例，让测试在等待完成三个异步调用之后来做一些其他事情。

```
package com.yingxue.lesson;

import com.yingxue.lesson.task.MyTask;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

import java.util.concurrent.Future;

@RunWith(SpringRunner.class)
@SpringBootTest
public class SpringbootAsyncApplicationTests {

    @Test
    public void contextLoads() {
    }

    @Autowired
    private MyTask myTask;

    @Test
    public void testTask() throws Exception{
//        myTask.doTaskOne();
//        myTask.doTaskTwo();
//        myTask.doTaskThree();
        long start = System.currentTimeMillis();

        Future<String> task1 = myTask.doTaskOne();
        Future<String> task2 = myTask.doTaskTwo();
        Future<String> task3 = myTask.doTaskThree();

        while(true) {
            if(task1.isDone() && task2.isDone() && task3.isDone()) {
                // 三个任务都调用完成，退出循环等待
                break;
            }
            Thread.sleep(1000);
        }
```

```
        long end = System.currentTimeMillis();

        System.out.println("任务全部完成，总耗时：" + (end - start) + "毫秒");
    }

}
```

看看我们做了哪些改变：

- 在测试用例一开始记录开始时间
- 在调用三个异步函数的时候，返回 `Future<String>` 类型的结果对象
- 在调用完三个异步函数之后，开启一个循环，根据返回的 `Future<String>` 对象来判断三个异步函数是否都结束了。若都结束，就结束循环；若没有都结束，就等1秒后再判断。
- 跳出循环之后，根据结束时间 - 开始时间，计算出三个任务并发执行的总耗时。

执行一下上述的单元测试，可以看到如下结果：

```
开始做任务二
开始做任务一
开始做任务三
完成任务二，耗时：1904毫秒
完成任务三，耗时：1914毫秒
完成任务一，耗时：4246毫秒
任务全部完成，总耗时：5008毫秒
```

5. Spring Boot 使用@Async 实现异步调用-自定义线程池

开启异步注解 @EnableAsync 方法上加 @Async 默认实现 SimpleAsyncTaskExecutor 不是真的线程池，这个类不重用线程，每次调用都会创建一个新的线程

- 配置线程池

```
package com.yingxue.lesson;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.scheduling.annotation.EnableAsync;
import org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor;

import java.util.concurrent.Executor;
import java.util.concurrent.ThreadPoolExecutor;

@SpringBootApplication
@EnableAsync
public class SpringbootAsyncApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringbootAsyncApplication.class, args);
    }

    @Bean("myTaskExecutor")
    public Executor myTaskExecutor() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        executor.setCorePoolSize(10);//核心线程数量，线程池创建时候初始化的线程数
        executor.setMaxPoolSize(15);//最大线程数，只有在缓冲队列满了之后才会申请超过核心线程数的线程
        executor.setQueueCapacity(200);//缓冲队列，用来缓冲执行任务的队列
        executor.setKeepAliveSeconds(60);//当超过了核心线程数之外的线程在空闲时间到达之后会被销毁
        executor.setThreadNamePrefix("myTask-");//设置好了之后可以方便我们定位处理任务所在的线程池
        executor.setWaitForTasksToCompleteOnShutdown(true);//用来设置线程池关闭的时候等待所有任务都完成再继续销毁其他的Bean
        executor.setAwaitTerminationSeconds(60);//该方法用来设置线程池中任务的等待时间，如果超过这个时候还没有销毁就强制销毁，以确保应用最后能够被关闭，而不是阻塞住。
        //线程池对拒绝任务的处理策略：这里采用了CallerRunsPolicy策略，当线程池没有处理能力的时候，该策略会直接在 execute 方法的调用线程中运行被拒绝的任务；如果执行程序已关闭，则会丢弃该任务
        executor.setRejectedExecutionHandler(new ThreadPoolExecutor.CallerRunsPolicy());
        return executor;
    }
```

```
}
```

- 改造MyTask

```
package com.yingxue.lesson.task;

import org.springframework.scheduling.annotation.Async;
import org.springframework.scheduling.annotation.AsyncResult;
import org.springframework.stereotype.Component;

import java.util.Random;
import java.util.concurrent.Future;

/**
 * @ClassName: MyTask
 * TODO:类文件简单描述
 * @Author: 小霍
 * @UpdateUser: 小霍
 * @Version: 0.0.1
 */
@Component
public class MyTask {

    public static Random random =new Random();

    @Async("myTaskExecutor")
    public Future<String> doTaskOne() throws Exception {
        System.out.println("开始做任务一");
        long start = System.currentTimeMillis();
        Thread.sleep(random.nextInt(10000));
        long end = System.currentTimeMillis();
        System.out.println("完成任务一，耗时：" + (end - start) + "毫秒");
        return new AsyncResult<>("完成任务一");
    }
    @Async("myTaskExecutor")
    public Future<String> doTaskTwo() throws Exception {
        System.out.println("开始做任务二");
        long start = System.currentTimeMillis();
        Thread.sleep(random.nextInt(10000));
        long end = System.currentTimeMillis();
        System.out.println("完成任务二，耗时：" + (end - start) + "毫秒");
        return new AsyncResult<>("完成任务二");
    }
    @Async("myTaskExecutor")
    public Future<String> doTaskThree() throws Exception {
        System.out.println("开始做任务三");
        long start = System.currentTimeMillis();
        Thread.sleep(random.nextInt(10000));
        long end = System.currentTimeMillis();
        System.out.println("完成任务三，耗时：" + (end - start) + "毫秒");
        return new AsyncResult<>("完成任务三");
    }

}
```

执行一下上述的单元测试，可以看到如下结果：

```
开始做任务二
开始做任务三
开始做任务一
完成任务一，耗时：1090毫秒
完成任务三，耗时：4808毫秒
完成任务二，耗时：5942毫秒
任务全部完成，总耗时：6018毫秒
```