

10-异步任务与定时任务

1.实现Async异步任务

- 一、环境准备
- 二、同步调用
- 三、异步调用
- 四、异步回调

2.为异步任务规划线程池

- 一、Spring Boot任务线程池
- 二、自定义线程池
- 三、优雅地关闭线程池

3.通过@Scheduled实现定时任务

- 一、开启定时任务方法
- 二、不同定时方式的解析
 - 1.fixedDelay和fixedRate
 - 2.cron表达式：灵活
- 三、实现定时任务
- 四、解决定时任务单线程运行的问题

4.quartz简单定时任务(内存持久化)

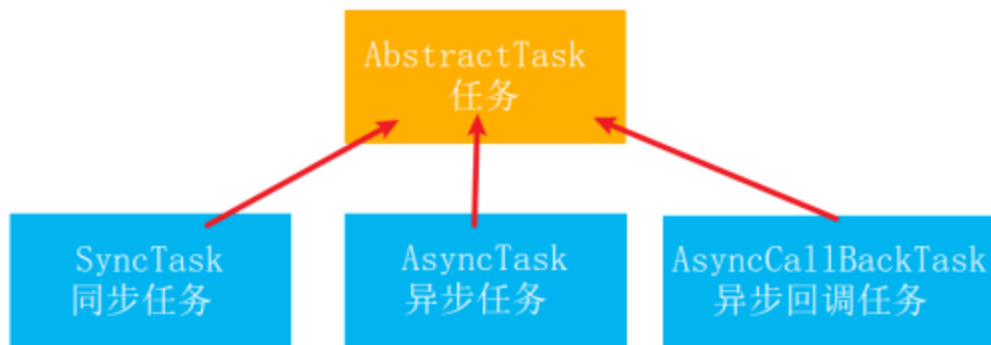
- 一、引入对应的 maven依赖
- 二、创建一个任务类Job
- 三、创建 Quartz 定时配置类
- 四、深入解析
 - 4.1.核心概念
 - 4.2.SimpleTrigger and CronTrigger

5.quartz动态定时任务(数据库持久化)

- 一、前言
- 二、原理
- 三、配置
- 四、quartz建表脚本

1.实现Async异步任务

一、环境准备



在 Spring Boot 入口类上配置 `@EnableAsync` 注解开启异步处理。

创建任务抽象类 `AbstractTask`，并分别配置三个任务方法 `doTaskOne()`，`doTaskTwo()`，`doTaskThree()`。

```
1  public abstract class AbstractTask {
2      private static Random random = new Random();
3
4      public void doTaskOne() throws Exception {
5          System.out.println("开始做任务一");
6          long start = currentTimeMillis();
7          sleep(random.nextInt(10000));
8          long end = currentTimeMillis();
9          System.out.println("完成任务一, 耗时: " + (end - start) + "毫秒");
10     }
11
12     public void doTaskTwo() throws Exception {
13         System.out.println("开始做任务二");
14         long start = currentTimeMillis();
15         sleep(random.nextInt(10000));
16         long end = currentTimeMillis();
17         System.out.println("完成任务二, 耗时: " + (end - start) + "毫秒");
18     }
19
20     public void doTaskThree() throws Exception {
21         System.out.println("开始做任务三");
22         long start = currentTimeMillis();
23         sleep(random.nextInt(10000));
24         long end = currentTimeMillis();
25         System.out.println("完成任务三, 耗时: " + (end - start) + "毫秒");
26     }
27 }
```

二、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，继承 AbstractTask，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）。

```
1  @Component
2  public class SyncTask extends AbstractTask {
3
4  }
```

- 在单元测试用例中，注入 SyncTask 对象，并在测试用例中执行 doTaskOne(), doTaskTwo(), doTaskThree() 三个方法。

```
1  @SpringBootTest
2  @ExtendWith(SpringExtension.class)
3  class SyncTaskTest {
4      @Resource
5      private SyncTask syncTask;
6
7      @Test
8      public void testSyncTasks() throws Exception {
9          syncTask.doTaskOne();
10         syncTask.doTaskTwo();
11         syncTask.doTaskThree();
12     }
13 }
```

- 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：6720毫秒 开始做任务二 完成任务二，耗时：6604毫秒 开始做任务三
完成任务三，耗时：9448毫秒



任务一、任务二、任务三顺序的执行完了，换言之 `doTaskOne()`，`doTaskTwo()`，`doTaskThree()` 三个方法按调用顺序的先后执行完成。

三、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Application启动类上面加上`@EnableAsync`
- 创建 `AsyncTask`类，分别在方法上配置 `@Async` 注解，将原来的 **同步方法** 变为 **异步方法**。

```
1  @Component
2  public class AsyncTask extends AbstractTask {
3      @Async
4      public void doTaskOne() throws Exception {
5          super.doTaskOne();
6      }
7
8      @Async
9      public void doTaskTwo() throws Exception {
10         super.doTaskTwo();
11     }
12
13     @Async
14     public void doTaskThree() throws Exception {
15         super.doTaskThree();
16     }
17 }
```

- 在单元测试用例中，注入 AsyncTask 对象，并在测试用例中执行 doTaskOne(), doTaskTwo(), doTaskThree() 三个方法。

```
1  @SpringBootTest
2  @ExtendWith(SpringExtension.class)
3  class AsyncTaskTest {
4      @Resource
5      private AsyncTask asyncTask;
6
7      @Test
8      public void testAsyncTasks() throws Exception {
9          asyncTask.doTaskOne();
10         asyncTask.doTaskTwo();
11         asyncTask.doTaskThree();
12         System.out.println("执行其他代码");
13     }
14 }
```

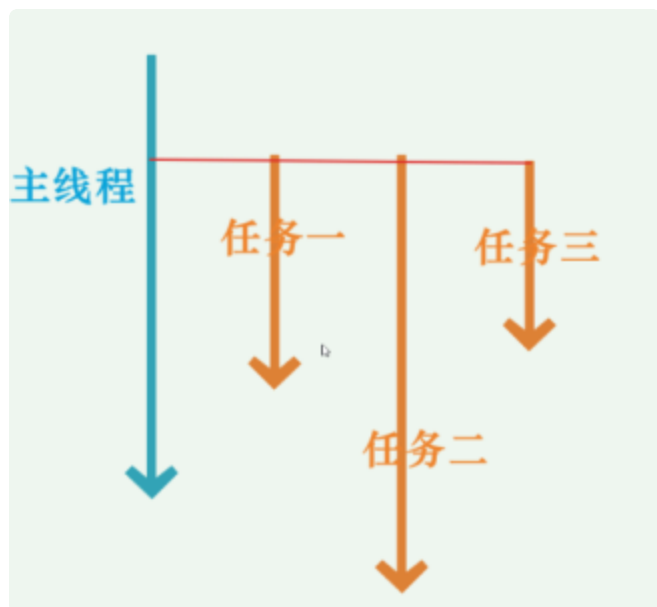
- 执行单元测试，可以看到类似如下输出：

```
1  开始做任务三
2  开始做任务一
3  开始做任务二
```

开始做任务三 开始做任务一 开始做任务二

如果反复执行单元测试，可能会遇到各种不同的结果，比如：

1. 没有任何任务相关的输出
2. 有部分任务相关的输出
3. 乱序的任务相关的输出



原因是目前 `doTaskOne()`、`doTaskTwo()`、`doTaskThree()` 这三个方法已经 **异步并发执行** 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，导致了任务 **不完整** 或是 **没有输出** 相关内容的情况。

注意：@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。

四、异步回调

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动用之后记录时间，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**。

- 创建 `AsyncCallbackTask` 类，声明 `doTaskOneCallback()`，`doTaskTwoCallback()`，`doTaskThreeCallback()` 三个方法，对原有的 三个方法进行包装。

```
Java | 复制代码

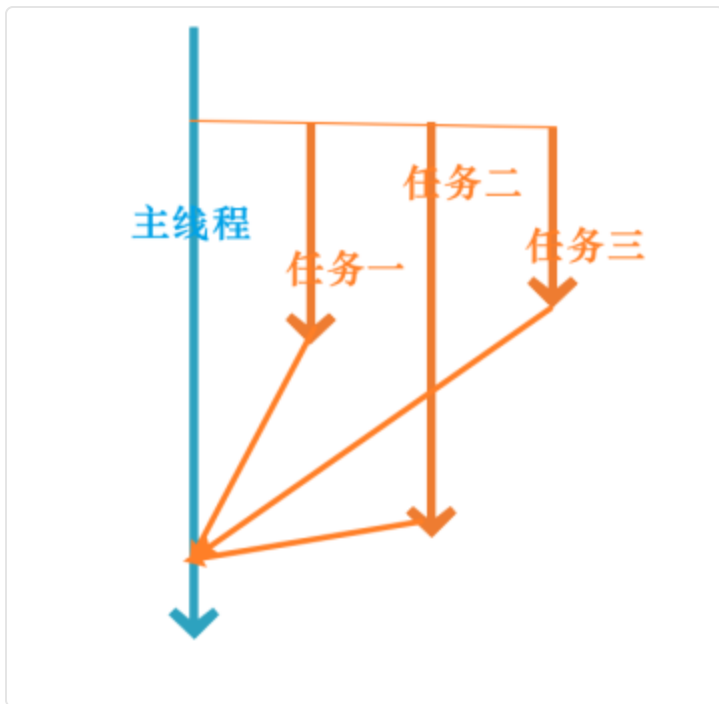
1  @Component
2  ▼ public class AsyncCallbackTask extends AbstractTask {
3      @Async
4  ▼  public Future<String> doTaskOneCallback() throws Exception {
5          super.doTaskOne();
6          return new AsyncResult<>("任务一完成");
7      }
8
9      @Async
10 ▼ public Future<String> doTaskTwoCallback() throws Exception {
11     super.doTaskTwo();
12     return new AsyncResult<>("任务二完成");
13 }
14
15     @Async
16 ▼ public Future<String> doTaskThreeCallback() throws Exception {
17     super.doTaskThree();
18     return new AsyncResult<>("任务三完成");
19 }
20 }
```

- 在 **单元测试** 用例中，注入 `AsyncCallbackTask` 对象，并在测试用例中执行 `doTaskOneCallback()`，`doTaskTwoCallback()`，`doTaskThreeCallback()` 三个方法。循环调用 `Future` 的 `isDone()` 方法 等待三个 **并发任务** 执行完成，记录 **最终执行时间**。


```
1  @Autowired
2  private AsyncCallbackTask asyncCallbackTask;
3
4  @Test
5  public void testAsyncCallbackTask() throws Exception {
6      long start = currentTimeMillis();
7      Future<String> task1 = asyncCallbackTask.doTaskOneCallback();
8      Future<String> task2 = asyncCallbackTask.doTaskTwoCallback();
9      Future<String> task3 = asyncCallbackTask.doTaskThreeCallback();
10
11     // 三个任务都调用完成，退出循环等待
12     while (!task1.isDone() || !task2.isDone() || !task3.isDone()) {
13         sleep(1000);
14     }
15
16     long end = currentTimeMillis();
17     System.out.println("任务全部完成，总耗时: " + (end - start) + "毫秒");
18 }
```

看看都做了哪些改变：

- 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：

开始做任务三 开始做任务一 开始做任务二 完成任务二，耗时：2572毫秒 完成任务一，耗时：7333毫秒
完成任务三，耗时：7647毫秒 任务全部完成，总耗时：8013毫秒

可以看到，通过 **异步调用**，让任务一、任务二、任务三 **并发执行**，有效**减少**了程序的**运行总时间**。

2.为异步任务规划线程池

一、Spring Boot任务线程池

线程池的作用

1. 防止资源占用无限的扩张
2. 调用过程省去资源的创建和销毁所占用的时间

在上一节中，我们的一个异步任务打开了一个线程，完成后销毁。在高并发环境下，不断的分配新资源，可能导致系统资源耗尽。所以为了避免这个问题，我们为异步任务规划一个线程池。当然，如果没有配置线程池的话，springboot会自动配置一个ThreadPoolTaskExecutor 线程池到bean当中。

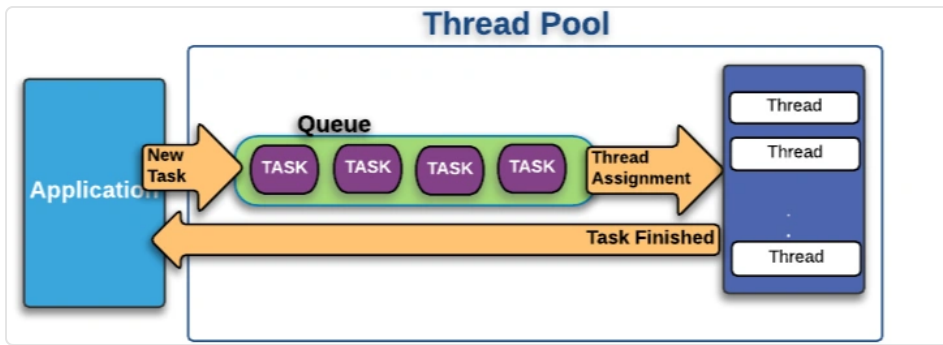
```
1  # 核心线程数
2  spring.task.execution.pool.core-size=8
3  # 最大线程数
4  spring.task.execution.pool.max-size=16
5  # 空闲线程存活时间
6  spring.task.execution.pool.keep-alive=60s
7  # 是否允许核心线程超时
8  spring.task.execution.pool.allow-core-thread-timeout=true
9  # 线程队列数量
10 spring.task.execution.pool.queue-capacity=100
11 # 线程关闭等待
12 spring.task.execution.shutdown.await-termination=false
13 spring.task.execution.shutdown.await-termination-period=
14 # 线程名称前缀
15 spring.task.execution.thread-name-prefix=task-
```

二、自定义线程池

有的时候，我们希望将系统内的一类任务放到一个线程池，另一类任务放到另外一个线程池，所以使用Spring Boot自带的任务线程池就捉襟见肘了。下面介绍自定义线程池的方法。

创建一个 **线程池配置类** `TaskConfiguration`，并配置一个 **任务线程池对象** `taskExecutor`。

```
1  @Configuration
2  public class TaskConfiguration {
3      @Bean("taskExecutor")
4      public Executor taskExecutor() {
5          ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
6          executor.setCorePoolSize(10);
7          executor.setMaxPoolSize(20);
8          executor.setQueueCapacity(200);
9          executor.setKeepAliveSeconds(60);
10         executor.setThreadNamePrefix("taskExecutor-");
11         executor.setRejectedExecutionHandler(new CallerRunsPolicy());
12         return executor;
13     }
14 }
```



上面我们通过使用 `ThreadPoolTaskExecutor` 创建了一个 **线程池**，同时设置了以下这些参数：

| 线程池属性 | 属性的作用 | 上文代码设置初始值 |
|---|---|-------------------------------|
| 核心线程数 <code>CorePoolSize</code> | 线程池创建时候初始化的线程数，最小线程数 | 10 |
| 最大线程数 <code>MaxPoolSize</code> | 线程池最大的线程数，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 | 20 |
| 缓冲任务队列 <code>QueueCapacity</code> | 用来缓冲执行任务的队列 | 200 |
| 允许线程的空闲时间 <code>KeepAliveSeconds</code> | 超过了核心线程之外的线程，在空闲时间到达之后，没活干的线程会被销毁 | 60秒 |
| 线程池名的前缀 <code>ThreadNamePrefix</code> | 可以用于定位处理任务所在的线程池 | <code>taskExecutor-</code> |
| 线程池对任务的Reject策略 <code>RejectedExecutionHandler</code> | 当线程池运行饱和，或者线程池处于shutdown临界状态时，用来拒绝一个任务的执行 | <code>CallerRunsPolicy</code> |

Reject策略预定义有四种：

- `AbortPolicy`，用于被拒绝任务的处理程序，它将抛出 `RejectedExecutionException`。
- `CallerRunsPolicy`，用于被拒绝任务的处理程序，它直接在 `execute` 方法的调用线程中运行被拒绝的任务。

- DiscardOldestPolicy, 用于被拒绝任务的处理程序, 它放弃最旧的未处理请求, 然后重试execute。
- DiscardPolicy, 用于被拒绝任务的处理程序, 默认情况下它将丢弃被拒绝的任务。

创建 AsyncExecutorTask类, 三个任务的配置和 AsyncTask 一样, 不同的是 @Async 注解需要指定前面配置的 线程池的名称taskExecutor。

```
Java | 复制代码

1  @Component
2  public class AsyncExecutorTask extends AbstractTask {
3      @Async("taskExecutor")
4      public Future<String> doTaskOneCallback() throws Exception {
5          super.doTaskOne();
6          System.out.println("任务一, 当前线程: " +
7              Thread.currentThread().getName());
8          return new AsyncResult<>("任务一完成");
9      }
10
11     @Async("taskExecutor")
12     public Future<String> doTaskTwoCallback() throws Exception {
13         super.doTaskTwo();
14         System.out.println("任务二, 当前线程: " +
15             Thread.currentThread().getName());
16         return new AsyncResult<>("任务二完成");
17     }
18
19     @Async("taskExecutor")
20     public Future<String> doTaskThreeCallback() throws Exception {
21         super.doTaskThree();
22         System.out.println("任务三, 当前线程: " +
23             Thread.currentThread().getName());
24         return new AsyncResult<>("任务三完成");
25     }
26 }
```

在 单元测试 用例中, 注入 AsyncExecutorTask 对象, 并在测试用例中执行 doTaskOne(), doTaskTwo(), doTaskThree() 三个方法。

```
1  @SpringBootTest
2  public class AsyncExecutorTaskTest {
3      @Autowired
4      private AsyncExecutorTask task;
5
6      @Test
7      public void testAsyncExecutorTask() throws Exception {
8          task.doTaskOneCallback();
9          task.doTaskTwoCallback();
10         task.doTaskThreeCallback();
11
12         sleep(30 * 1000L);
13     }
14 }
```

执行上述单元测试，可以看到如下结果：

```
1  开始做任务一
2  开始做任务三
3  开始做任务二
4  完成任务二，耗时：3905毫秒
5  任务二，当前线程：taskExecutor-2
6  完成任务一，耗时：6184毫秒
7  任务一，当前线程：taskExecutor-1
8  完成任务三，耗时：9737毫秒
9  任务三，当前线程：taskExecutor-3
```

执行上面的单元测试，我们观察到 **任务线程池** 的 **线程池名的前缀** 被打印，说明 **线程池** 成功执行了 **异步任务**！

三、优雅地关闭线程池

由于在应用关闭的时候异步任务还在执行，导致类似 **数据库连接池** 这样的对象一并被 **销毁**了，当 **异步任务** 中对 **数据库** 进行操作就会出错。

解决方案如下，重新设置线程池配置对象，新增线程池 `setWaitForTasksToCompleteOnShutdown()` 和 `setAwaitTerminationSeconds()` 配置：

```
Java | 复制代码

1  @Bean("taskExecutor")
2  public Executor taskExecutor() {
3      ThreadPoolTaskScheduler executor = new ThreadPoolTaskScheduler();
4      executor.setPoolSize(20);
5      executor.setThreadNamePrefix("taskExecutor-");
6      executor.setWaitForTasksToCompleteOnShutdown(true);
7      executor.setAwaitTerminationSeconds(60);
8      return executor;
9  }
```

- `setWaitForTasksToCompleteOnShutdown(true)`: 该方法用来设置 线程池关闭 的时候 等待 所有任务都完成后，再继续 销毁 其他的 Bean，这样这些 异步任务 的 销毁 就会先于 数据库连接池对象 的 销毁。
- `setAwaitTerminationSeconds(60)`: 该方法用来设置线程池中 任务的等待时间，如果超过这个时间还没有销毁就 强制销毁，以确保应用最后能够被关闭，而不是阻塞住。

3.通过@Scheduled实现定时任务

一、开启定时任务方法

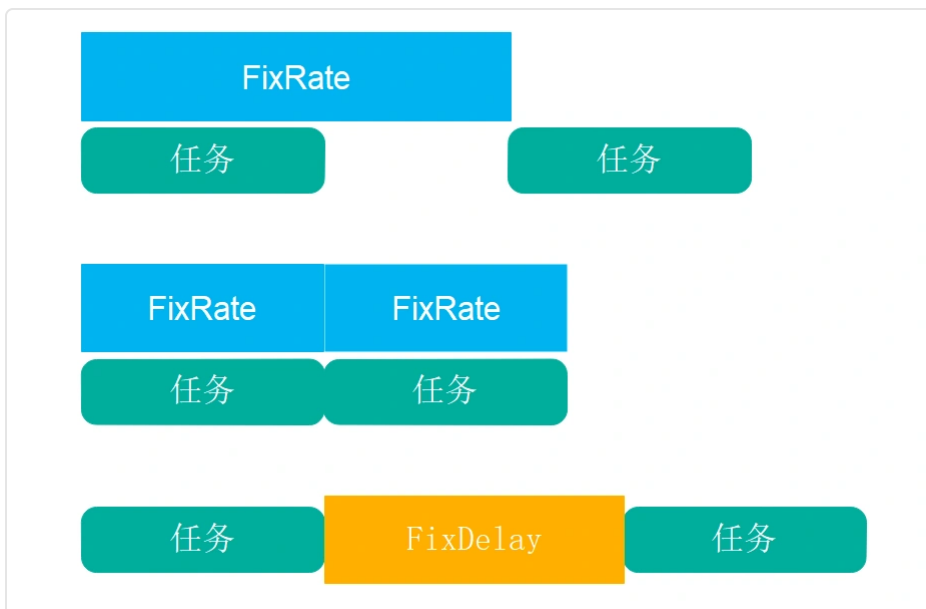
Scheduled定时任务是Spring boot自身提供的功能，所以不需要引入Maven依赖包
在项目入口main方法上加注解

`@EnableScheduling` //开启定时任务

二、不同定时方式的解析

1.fixedDelay和fixedRate

单位是毫秒，它们的区别就是：



- fixedRate就是每隔多长时间执行一次。(开始----->X时间----->再开始)。如果间隔时间小于任务执行时间，上一次任务执行完成下一次任务就立即执行。如果间隔时间大于任务执行时间，就按照每隔X时间运行一次。
- 而fixedDelay是当任务执行完毕后一段时间再次执行。(开始--->结束(隔一分钟)开始----->结束)。上一次执行任务未完成，下一次任务不会开始。

2.cron表达式：灵活

举例说明

| 表达式 | 说明 |
|----------------|--------------------------------------|
| 0 0 3 * * ? | 每天3点执行 |
| 0 5 3 * * ? | 每天3点5分执行 |
| 0 5 3 ? * * | 每天3点5分执行，与上面作用相同 |
| 0 5/10 3 * * ? | 每天3点的 5分，15分，25分，35分，45分，55分这几个时间点执行 |
| 0 10 3 ? * 1 | 每周星期天，3点10分 执行，注：1表示星期天 |
| 0 10 3 ? * 1#3 | 每个月的第三个星期，星期天 执行，#号只能出现在星期的位置 |

- 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：不是第1周，第二周的意思，另外：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。

(-)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15秒，最后就是0，15，30，45，60 另：/y，等同于0/y

cron表达式在线：<http://cron.qqe2.com/>

三、实现定时任务

```
Java | 复制代码

1  @Component
2  public class ScheduledJobs {
3
4      //表示方法执行完成后5秒再开始执行
5      @Scheduled(fixedDelay=5000)
6      public void fixedDelayJob() throws InterruptedException{
7          System.out.println("fixedDelay 开始:" + new Date());
8          Thread.sleep(10 * 1000);
9          System.out.println("fixedDelay 结束:" + new Date());
10     }
11
12     //表示每隔3秒
13     @Scheduled(fixedRate=3000)
14     public void fixedRateJob()throws InterruptedException{
15         System.out.println("=====fixedRate 开始:" + new Date());
16         Thread.sleep(5 * 1000);
17         System.out.println("=====fixedRate 结束:" + new Date());
18     }
19
20     //表示每隔10秒执行一次
21     @Scheduled(cron="0/10 * * * * ? ")
22     public void cronJob(){
23         System.out.println("===== ...>>cron...." +
24             new Date());
25     }
26 }
```

从运行结果上看，并未按照预期的时间规律运行。仔细看线程打印，所有的定时任务使用的都是一个线程，所以彼此互相影响。

四、解决定时任务单线程运行的问题

```

1  @Configuration
2  @EnableScheduling
3  public class ScheduleConfig implements SchedulingConfigurer {
4
5      @Override
6      public void configureTasks(ScheduledTaskRegistrar taskRegistrar) {
7          taskRegistrar.setScheduler(scheduledTaskExecutor());
8      }
9
10     @Bean
11     public Executor scheduledTaskExecutor() {
12         //指定线程池大小
13         return Executors.newScheduledThreadPool(3);
14     }
15 }

```

4.quartz简单定时任务(内存持久化)

Quartz是OpenSymphony开源组织在工作计划-定时任务领域的另一个开源项目。它是完全由Java开发的，可用于执行预定任务。它类似于java.util.Timer定时器。但是与timer相比，quartz增加了许多功能。

一、引入对应的 maven依赖

在 springboot2.0 后官方添加了 Quartz 框架的依赖，所以只需要在 pom 文件当中引入

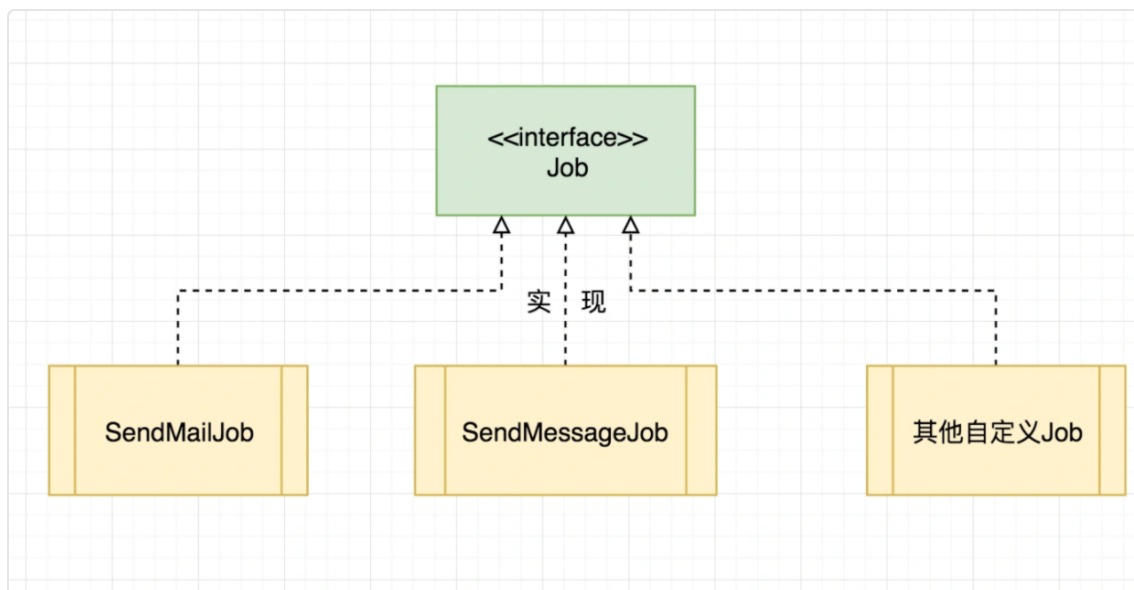
```

1  <!--引入quartz定时框架-->
2  <dependency>
3  <groupId>org.springframework.boot</groupId>
4  <artifactId>spring-boot-starter-quartz</artifactId>
5  </dependency>

```

二、创建一个任务类Job

首先，我们需要定义一个接口来实现计时功能。我们可以将其称为任务（或任务），例如：定期发送电子邮件的任务，重新启动机器的任务以及在优惠券到期时发送SMS提醒的任务。



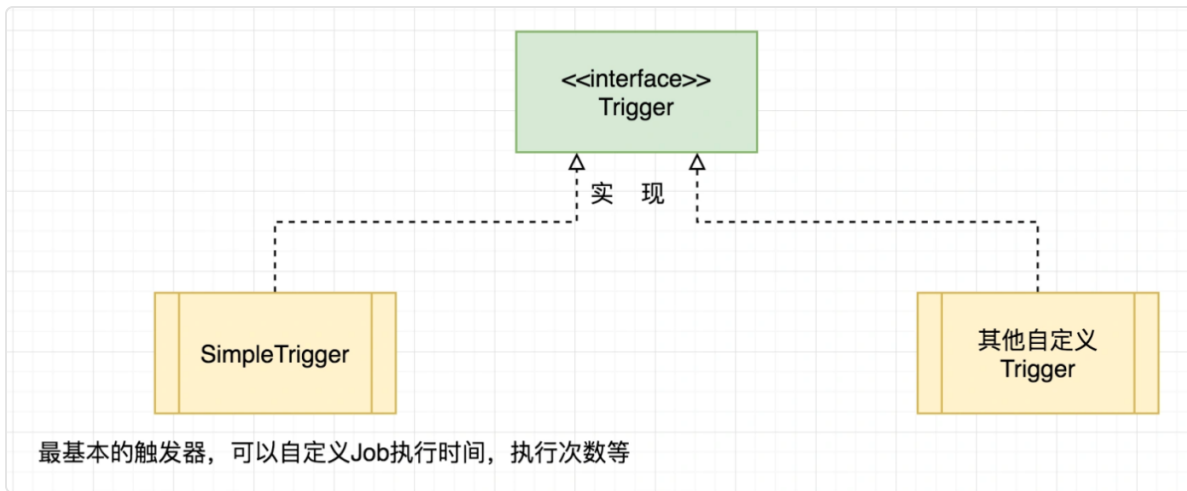
由于 SpringBoot2.0 自动进行了依赖所以创建的定时任务类直接继承 QuartzJobBean 就可以了，新建一个定时任务类：QuartzSimpleTask

```
Java | 复制代码

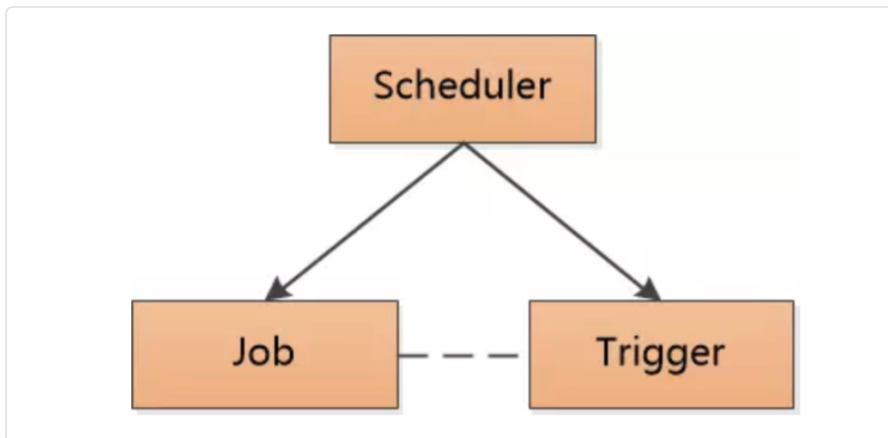
1 public class QuartzSimpleTask extends QuartzJobBean {
2     @Override
3     protected void executeInternal(JobExecutionContext
4         jobExecutionContext) throws JobExecutionException {
5         System.out.println("quartz简单的定时任务执行时间: "+new
6             Date().toLocaleString());
7     }
8 }
```

三、创建 Quartz 定时配置类

还需要一个可以触发任务执行的触发器。触发器触发器的基本功能是指定作业的执行时间，执行间隔和运行时间。



如何结合作业与触发？也就是说，如何分配触发器以执行指定的作业？此时，需要一个Schedule来实现此功能。



将之前创建的定时任务添加到定时调度里面

```
1  @Configuration
2  public class QuartzSimpleConfig {
3      //指定具体的定时任务类
4      @Bean
5      public JobDetail uploadTaskDetail() {
6          return JobBuilder.newJob(QuartzSimpleTask.class)
7              .withIdentity("QuartzSimpleTask")
8              .storeDurably().build();
9      }
10
11     @Bean
12     public Trigger uploadTaskTrigger() {
13         //这里设定触发执行的方式
14         CronScheduleBuilder scheduleBuilder =
15             CronScheduleBuilder.cronSchedule("* /5 * * * * ?");
16         // 返回任务触发器
17         return TriggerBuilder.newTrigger().forJob(uploadTaskDetail())
18             .withIdentity("QuartzSimpleTask")
19             .withSchedule(scheduleBuilder)
20             .build();
21     }
22 }
```

最后运行项目查看效果，"* /5 * * * * ?"表示定时任务，每隔5秒钟执行一次。

四、深入解析

4.1.核心概念

- **Job**：一个仅包含一个`void execute(JobExecutionContext context)` Abstract方法的简单接口。在实际开发中，要执行的任务是通过实现接口自定义实现的。`JobExecutionContext`提供调度上下文信息。

```
1  public interface Job {
2      void execute(JobExecutionContext context)
3          throws JobExecutionException;
4  }
```

- **JobDetail**: 包含多个构造函数，最常用的是`JobDetail(String name, String group, Class jobClass)`。Jobclass是实现作业接口的类，name是调度程序中任务的名称，group是调度程序中任务的组名。默认组名称为`Scheduler.DEFAULT_GROUP`。
- **Trigger**: 描述触发作业执行的时间规则的类。包含：
 1. **SimpleTrigger**: 一次或固定间隔时间段的触发规则。
 2. **CronTrigger**: 通过cron表达式描述更复杂的触发规则。
- **Calendar**: Quartz 提供的`Calendar`类。触发器可以与多个Calendar关联以排除特殊日期。
- **Scheduler**: 代表独立于Quartz 的运行容器。在Scheduler 中注册了Trigger和JobDetail。它们在调度程序中具有自己的名称（名称）和组名称（Group）。触发器和JobDetail名称和组名称的组合必须唯一，但是触发器名称和组名称的组合可以与JobDetail相同。一个Job可以绑定到多个触发器，也可以不绑定。

Job还具有一个子接口：statefuljob，这是一个没有方法的标签接口，表示有状态任务。

1. 无状态任务：它具有jobdatamap复制，因此可以并发运行；
2. 有状态任务statefuljob：共享一个jobdatamap，并且将保存对jobdatamap的每次修改。因此，前一个有statefuljob将阻止下一个statefuljob。

4.2.SimpleTrigger and CronTrigger

- **SimpleTrigger**可以在指定的时间段内执行一个Job任务，也可以在一个时间段内多次执行。
- **CronTrigger**功能非常强大，它基于Calendar进行作业调度，并且可以比simpletrigger更精确地指定间隔，因此crotrigger比simpletrigger更常用。Crotrigger基于cron表达式。

首先，让我们了解cron表达式： 由七个子表达式组成的字符串的格式如下：

[秒][分钟][小时][天][月][周][年]

例如：`00:00:00? * 10,11,12 1 # 5 2018` ,表示2018年10月，11月和12月的第一周星期五的00:00:00。

看上去不是很容易书写与记忆，但是我们可以通过网络上的在线Cron表达式生成工具，来帮助我们写表达式：在线生成cron表达式的工具：[http: //cron.qqe2.com/](http://cron.qqe2.com/)

| 位置 | 时间域名 | 容许值 | 允许特殊字符 |
|----|--------|------------|----------|
| 1个 | 秒 | 0-59 | , - * / |
| 2 | 分钟 | 0-59 | , - * / |
| 3 | 小时 | 0-23 | , - * / |
| 4 | 日期 | 1-31 | , - *? / |
| 5 | 月 | 1-12 | , - * / |
| 6 | 周 | 1-7 | , - *? / |
| 7 | 年份（可选） | 空1970-2099 | , - * / |

特殊字符的含义如下：

- 星号 (*)：可在所有字段中使用以指示相应时域中的每次时间。例如，分钟字段中的*表示“每分钟”；
- 问号 (?)：此字符仅在日期和星期字段中使用。通常将其指定为“无意义的值”，等同于点字符；
- 减号 (-)：表示范围。如果在小时字段中使用“10-12”，则表示10到12，即10、11、12；
- 逗号 (,)：表示列表值。如果在星期字段中使用“星期一，星期三，星期五”，则表示星期一，星期三和星期五；
- 斜线 (/)：X / Y表示相等的步长序列，其中X为起始值，y为增量步长值。如果在分钟字段中使用0/15，则表示0、15、30和45秒，而5/15在分钟字段中表示5、20、35、50，也可以使用* / y，这等效到0 / y；

5.quartz动态定时任务(数据库持久化)

一、前言

在项目开发过程当中，某些定时任务，可能在运行一段时间之后就不需要了，或者需要修改定时任务的执行时间等等。

这就需要在代码中进行修改，重新打包发布，就很麻烦。

使用Quartz定时任务框架来实现，就不需要重新修改代码而达到要求。

二、原理

1. 使用quartz提供的API完成配置任务的增删改查
2. 将任务的配置保存在数据库中

三、配置

除了之前的 quartz依赖，再加入 mysql依赖和Mybatis依赖。

```
XML | 复制代码

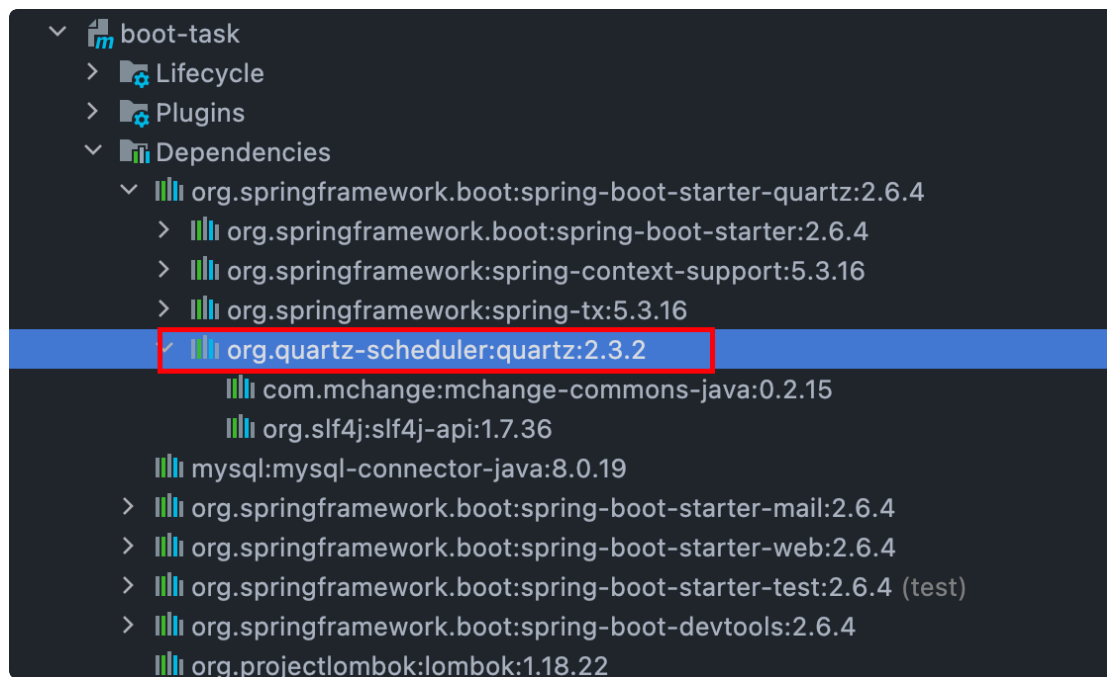
1 <dependency>
2   <groupId>mysql</groupId>
3   <artifactId>mysql-connector-java</artifactId>
4 </dependency>
5
6 <dependency>
7   <groupId>org.mybatis.spring.boot</groupId>
8   <artifactId>mybatis-spring-boot-starter</artifactId>
9   <version>2.2.2</version>
10 </dependency>
```

然后开始配置quartz：

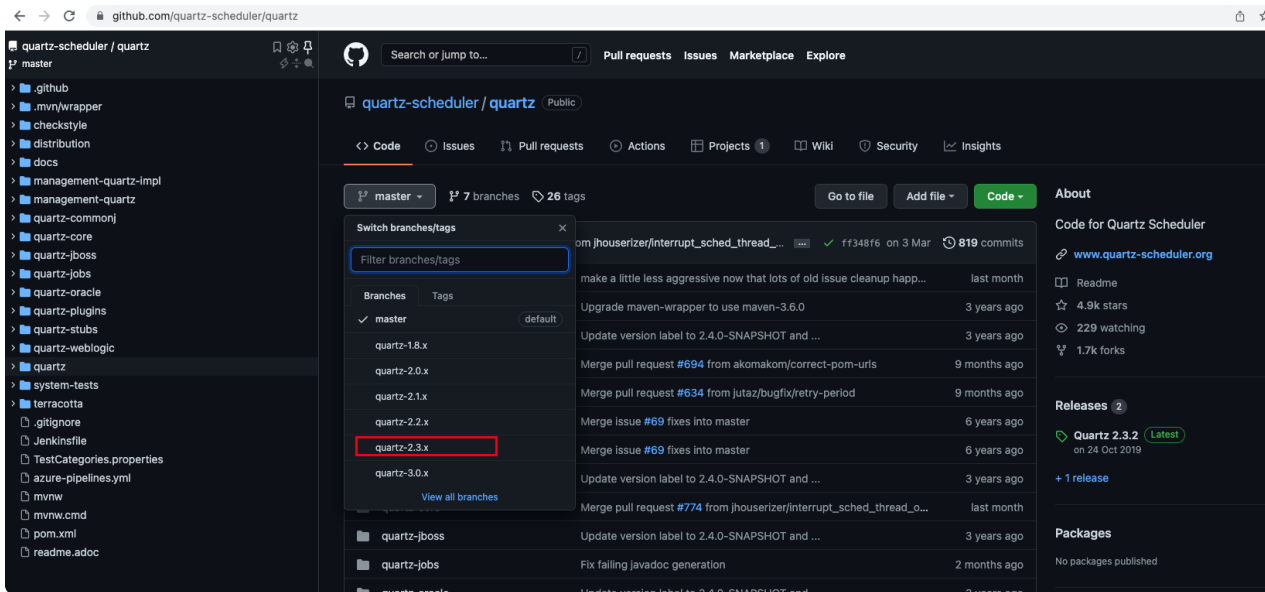
```
1  spring:
2    datasource:
3      url: jdbc:mysql://localhost:3306/spring_boot?
4        useUnicode=true&characterEncoding=utf-8&useSSL=false
5      username: root
6      password: 123456
7      driverClassName: com.mysql.cj.jdbc.Driver
8    quartz:
9      job-store-type: JDBC #数据库存储quartz任务配置
10     jdbc:
11       initialize-schema: NEVER #自动初始化表结构，第一次启动的时候这里写always
```

四、quartz建表脚本

先查看当前工程Maven中 quartz的版本，发现是 2.3.2



然后到 quartz 的 [github主页](#)，选择 版本对应的分支



然后按如下方式，找到mysql的建表脚本，到本地数据库执行，生成数据表。

https://github.com/quartz-scheduler/quartz/blob/quartz-2.3.x/quartz-core/src/main/resources/org/quartz/impl/jdbcjobstore/tables_mysql.sql

| 名 | 行 | 数据长度 | 引擎 | 创建日期 | 修改日期 | 排序规则 | 注 |
|-----------------------|----|----------|--------|---------------------|---------------------|--------------------|---|
| article | 3 | 16.00 KB | InnoDB | 2022-03-27 20:53:42 | | utf8mb4_general_ci | |
| boot_user | 1 | 16.00 KB | InnoDB | 2022-03-31 19:20:40 | | utf8mb4_general_ci | |
| logging_event | 81 | 48.00 KB | InnoDB | 2022-04-11 15:34:37 | 2022-04-11 20:27:04 | utf8mb4_general_ci | |
| logging_event_ex... | 0 | 16.00 KB | InnoDB | 2022-04-11 15:34:37 | | utf8mb4_general_ci | |
| logging_event_pr... | 0 | 16.00 KB | InnoDB | 2022-04-11 15:34:37 | | utf8mb4_general_ci | |
| logging_event_prop... | 0 | 16.00 KB | InnoDB | 2022-04-12 18:26:16 | | utf8mb4_general_ci | |
| QRTZ_BLOB_TRIGGERS | 0 | 16.00 KB | InnoDB | 2022-04-12 18:26:16 | | utf8mb4_general_ci | |
| QRTZ_CALEDARS | 0 | 16.00 KB | InnoDB | 2022-04-12 18:26:16 | | utf8mb4_general_ci | |
| QRTZ_CRON_TRI... | 0 | 16.00 KB | InnoDB | 2022-04-12 18:26:16 | | utf8mb4_general_ci | |
| QRTZ_FIRED_TRI... | 0 | 16.00 KB | InnoDB | 2022-04-12 18:26:16 | | utf8mb4_general_ci | |
| QRTZ_JOB_DETA... | 0 | 16.00 KB | InnoDB | 2022-04-12 18:26:16 | | utf8mb4_general_ci | |
| QRTZ_LOCKS | 0 | 16.00 KB | InnoDB | 2022-04-12 18:26:16 | | utf8mb4_general_ci | |
| QRTZ_PAUSED_T... | 0 | 16.00 KB | InnoDB | 2022-04-12 18:26:16 | | utf8mb4_general_ci | |
| QRTZ_SCHEDUL... | 0 | 16.00 KB | InnoDB | 2022-04-12 18:26:16 | | utf8mb4_general_ci | |
| QRTZ_SIMPLE_T... | 0 | 16.00 KB | InnoDB | 2022-04-12 18:26:16 | | utf8mb4_general_ci | |
| QRTZ_SIMPROP... | 0 | 16.00 KB | InnoDB | 2022-04-12 18:26:16 | | utf8mb4_general_ci | |
| QRTZ_TRIGGERS | 0 | 16.00 KB | InnoDB | 2022-04-12 18:26:16 | | utf8mb4_general_ci | |
| t_clazz | 2 | 16.00 KB | InnoDB | 2022-03-27 21:27:37 | | utf8mb4_general_ci | |
| t_course | 2 | 16.00 KB | InnoDB | 2022-03-27 21:29:24 | | utf8mb4_general_ci | |
| t_course_studen... | 10 | 16.00 KB | InnoDB | 2022-03-27 21:29:44 | | utf8mb4_general_ci | |
| t_student | 8 | 16.00 KB | InnoDB | 2022-03-27 21:29:04 | | utf8mb4_general_ci | |
| t_teacher | 2 | 16.00 KB | InnoDB | 2022-03-27 21:26:42 | | utf8mb4_general_ci | |

五、测试

以上工作准备好之后，重启应用，随着前面定时任务的执行，会自动在 QRTZ_CRON_TRIGGERS表插入记录。

其他几张表也会插入相应数据，自行查看。

The screenshot shows a database client window with the title 'QRTZ_CRON_TRIGGERS | springboot@localhost'. The table 'QRTZ_CRON_TRIGGERS' is displayed with the following columns: SCHED_NAME, TRIGGER_NAME, TRIGGER_GROUP, CRON_EXPRESSION, and TIME_ZONE_ID. The data row shows: 1, quartzScheduler, QuartzSimpleTask, DEFAULT, */5 * * * * ?, Asia/Shanghai.

Below the table, the 'Run' section shows the 'TaskApplication' console output. The log displays the following messages:

```
quartz简单的定时任务执行时间: Tue Apr 12 19:10:45 CST 2022
quartz简单的定时任务执行时间: Tue Apr 12 19:10:50 CST 2022
quartz简单的定时任务执行时间: Tue Apr 12 19:10:55 CST 2022
quartz简单的定时任务执行时间: Tue Apr 12 19:11:00 CST 2022
quartz简单的定时任务执行时间: Tue Apr 12 19:11:05 CST 2022
quartz简单的定时任务执行时间: Tue Apr 12 19:11:10 CST 2022
quartz简单的定时任务执行时间: Tue Apr 12 19:11:15 CST 2022
quartz简单的定时任务执行时间: Tue Apr 12 19:11:20 CST 2022
quartz简单的定时任务执行时间: Tue Apr 12 19:11:25 CST 2022
quartz简单的定时任务执行时间: Tue Apr 12 19:11:30 CST 2022
quartz简单的定时任务执行时间: Tue Apr 12 19:11:35 CST 2022
quartz简单的定时任务执行时间: Tue Apr 12 19:11:40 CST 2022
quartz简单的定时任务执行时间: Tue Apr 12 19:11:45 CST 2022
```

我们修改代码中的 cron表达式，修改执行频率为 每2秒执行一次，可以看到实时生效了（配置了热更新，没配置的话需要重启应用）

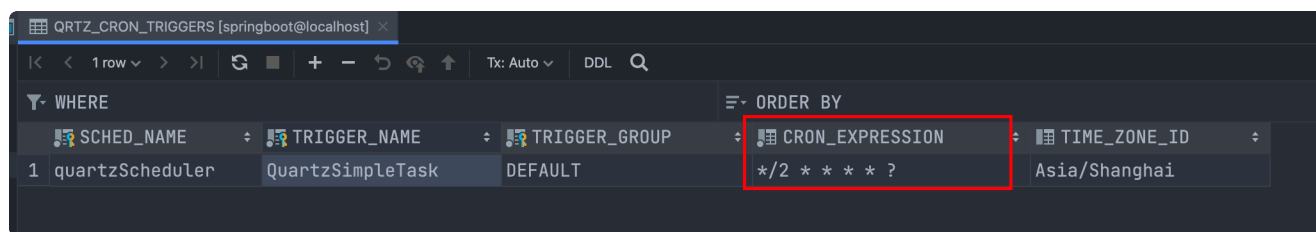
The screenshot shows an IDE window with the following code snippet:

```
@Bean
public Trigger uploadTaskTrigger() {
    //这里设定触发执行的方式
    CronScheduleBuilder scheduleBuilder = CronScheduleBuilder.cronSchedule("*/2 * * * * ?");
    // 返回任务触发器
    return TriggerBuilder.newTrigger().forJob(uploadTaskDetail()) TriggerBuilder<Trigger>
        .withIdentity("QuartzSimpleTask")
        .withSchedule(scheduleBuilder) TriggerBuilder<CronTrigger>
        .build();
}
```

Below the code, the 'Run' section shows the 'TaskApplication' console output. The log displays the following messages:

```
2022-04-12 19:17:10.683 INFO 41716 --- [ restartedMain] com.mqxu.boot.TaskApplication : Started TaskApplication
for 75.265)
2022-04-12 19:17:10.684 INFO 41716 --- [ restartedMain] .ConditionEvaluationDeltaLoggingListener : Condition evaluation u
quartz简单的定时任务执行时间: Tue Apr 12 19:17:10 CST 2022
quartz简单的定时任务执行时间: Tue Apr 12 19:17:12 CST 2022
quartz简单的定时任务执行时间: Tue Apr 12 19:17:14 CST 2022
quartz简单的定时任务执行时间: Tue Apr 12 19:17:16 CST 2022
```

数据表中的数据也自动更改成一致了



| | SCHED_NAME | TRIGGER_NAME | TRIGGER_GROUP | CRON_EXPRESSION | TIME_ZONE_ID |
|---|-----------------|------------------|---------------|-----------------|---------------|
| 1 | quartzScheduler | QuartzSimpleTask | DEFAULT | */2 * * * * ? | Asia/Shanghai |

over，初识 quartz框架搞定。