

使用 Quartz

Timer、ScheduledExecutorService在非分布式场景中都可以实现任务调度，但对于特定循环周期（如每周三、每晚12点）的任务调度实现就比较麻烦

一、Quartz介绍

1.1 Quartz 介绍

Quartz是OpenSymphony开源组织在Job scheduling领域的一个开源项目，该项目于 2009 年被 Terracotta 收购，目前是 Terracotta 旗下的一个项目。它完全由java编写、可以与J2EE与J2SE应用程序相结合也可以单独使用。Quartz可以用来创建运行十个、百个、乃至几万个Jobs这样复杂的程序。

官网：<http://www.quartz-scheduler.org/>

1.2 Quartz 特点

1. 强大的调度功能，例如支持丰富多样的调度方法，可以满足各种常规及特殊需求；
2. 灵活的应用方式，例如支持任务和调度的多种组合方式，支持调度数据的多种存储方式；
3. 分布式和集群能力，Terracotta 收购后在原来功能基础上作了进一步提升；
4. 作为 Spring 默认的调度框架，Quartz 很容易与 Spring 集成实现灵活可配置的调度功能。

二、Quartz任务调度实现

2.1 创建maven工程

- java工程：下载quartz的jar包，引入到java工程中
- maven工程：使用pom引入maven的依赖坐标

2.2 添加quartz依赖

```
<dependency>
  <groupId>org.quartz-scheduler</groupId>
  <artifactId>quartz</artifactId>
  <version>2.3.2</version>
</dependency>
```

2.3 创建任务

- 创建一个类实现quartz提供的Job接口
- 实现Job接口定义的execute方法，定义任务逻辑

```
/**
 * @Description Quartz任务
 * @Author 千锋涛哥
 * 公众号： Java架构栈
 */
public class MyJob implements Job {
```

```

        private SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd hh:mm:ss");

        public void execute(JobExecutionContext jobExecutionContext)
            throws JobExecutionException {
            //执行任务的逻辑
            String time = sdf.format(new Date());
            System.out.println("~~~~~"+time);
        }
    }
}

```

2.4 启动任务

- 使用任务调度器Scheduler进行任务管理
 - 启动任务
 - 暂停任务
 - 继续任务
 - 停止任务

```

package com.qfedu.job;

import org.quartz.*;
import org.quartz.impl.StdSchedulerFactory;

import java.util.Date;

/**
 * @Description 测试Quartz的任务调度
 * @Author 千锋涛哥
 * 公众号： Java架构栈
 */
public class QuartzTest {

    public static void main(String[] args) {

        try {
            //1.创建Quartz的任务调度器
            Scheduler scheduler = StdSchedulerFactory.getDefaultScheduler();

            //2.创建触发器：定义了任务调度的时机（延时、循环、定时...）
            // simpleScheduleBuilder对象表示任务的触发规则
            SimpleScheduleBuilder simpleScheduleBuilder =
                SimpleScheduleBuilder.simpleSchedule()
                    .withIntervalInSeconds(3)
                    .withRepeatCount(10);

            SimpleTrigger trigger = TriggerBuilder.newTrigger()
                .withIdentity("trigger1", "group1")// withIdentity 设置trigger唯一表示
                .startAt(new Date(System.currentTimeMillis() + 5000))// startAt 设置首
                次触发时间
                .withSchedule(simpleScheduleBuilder)// withSchedule 设置触发器的触发规则
                .build(); // 通过TriggerBuilder对象，构造Trigger对象

            //3.创建任务(JobDetail 用于封装具体的任务)

```

```

        JobDetail job = JobBuilder.newJob(MyJob.class).build();

        //将任务及触发器，绑定到任务调度
        scheduler.scheduleJob(job, trigger);

        //启动任务
        scheduler.start()

    } catch (SchedulerException e) {
        e.printStackTrace();
    }
}
}

```

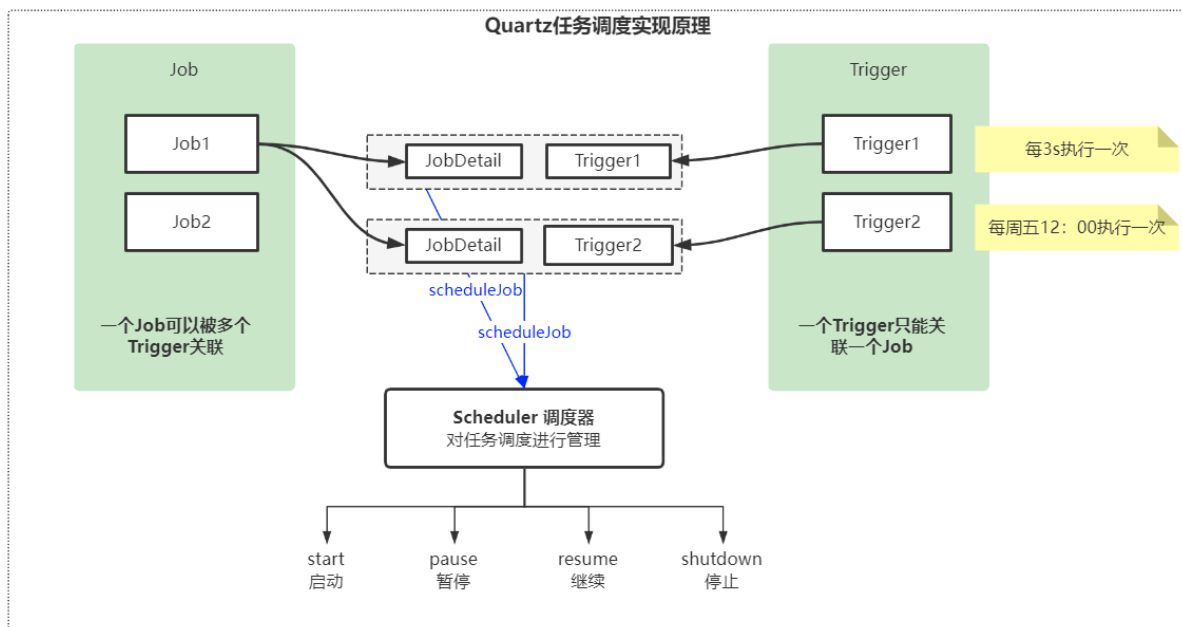
三、Quartz 基本实现原理

Quartz 任务调度的核心元素是 scheduler, trigger 和 job，其中 trigger 和 job 是任务调度的元数据，scheduler 是实际执行调度的控制器。

scheduler：任务调度器

trigger：触发器，用于定义任务调度时间规则

job：任务，即被调度的任务



3.1 Scheduler任务调度器

Scheduler 对任务进行调度和管理

在 Quartz 中，Scheduler 由 SchedulerFactory 工厂创建，主要提供了以下两种工厂实现：

- DirectSchedulerFactory（使用较麻烦，需要进行编码配置）
- StdSchedulerFactory（✓）

Scheduler 主要有三种：

- RemoteMBeanScheduler
- RemoteScheduler
- StdScheduler（✓）

创建任务调度器

```
//创建Quartz的任务调度器
Scheduler scheduler = StdSchedulerFactory.getDefaultScheduler();
```

3.2 Triggers触发器

trigger 是用于定义调度时间的元素，即按照什么时间规则去执行任务。

Quartz 中主要提供了四种类型的 Trigger：

- SimpleTrigger 使用简单便捷
- CronTrigger 使用cron表达式定义触发任务的时间规则
- DateIntervalTrigger 日期周期的规则的触发器
- NthIncludedDayTrigger 排除指定时间周期和日期的触发器

这四种 trigger 可以满足企业应用中的绝大部分需求。

3.2.1 SimpleTrigger

SimpleTrigger可以满足的调度需求是：在具体的时间点执行一次，或者在具体的时间点执行，并且以指定的间隔重复执行若干次。

- 延时任务：当用户提交订单之后启动任务，延时30min检查订单状态，如果未支付则取消订单释放库存
 - 循环任务：每个指定的时间周期执行一次任务，任务可以循环（可以无限循环、可以循环指定的次数、还可以循环执行到指定的时间点）
 - 定时任务：在指定的时间点调度执行任务
-
- 重复次数，可以是0、正整数，以及常量SimpleTrigger.REPEAT_INDEFINITELY
 - 重复的间隔，必须是0，或者long型的正数，表示毫秒。注意，如果重复间隔为0，trigger将会以重复次数并发执行(或者以scheduler可以处理的近似并发数)。
 - endTime属性的值会覆盖设置重复次数的属性值；比如，你可以创建一个trigger，在终止时间之前每隔10秒执行一次，你不需要去计算在开始时间和终止时间之间的重复次数，只需要设置终止时间并将重复次数设置为REPEAT_INDEFINITELY。

//【实例1】指定时间开始触发，不重复

```
SimpleTrigger trigger = (SimpleTrigger) TriggerBuilder.newTrigger()
    .withIdentity("trigger1", "group1")
    .startAt(myStartTime) // 指定任务开始的时间
    .build();
```

//【实例2】指定时间触发，每隔10秒执行一次，重复10次：

```
SimpleScheduleBuilder simpleScheduleBuilder = SimpleScheduleBuilder.simpleSchedule()
    .withIntervalInSeconds(10)
    .withRepeatCount(10);
SimpleTrigger trigger = (SimpleTrigger) TriggerBuilder.newTrigger()
    .withIdentity("trigger3", "group1")
    .startAt(myTimeToStartFiring) //指定触发的时间
    .withSchedule(simpleScheduleBuilder) //指定触发器的时间规则
    .forJob(myJob)
    .build();
```

//【实例3】5分钟以后开始触发，仅执行一次

// `DateBuilder.futureDate(5, DateBuilder.IntervalUnit.MINUTE)` 用于获取当前时间点往后推算某个时间之后的时间点

```
SimpleTrigger trigger = (SimpleTrigger) TriggerBuilder.newTrigger()
    .withIdentity("trigger5", "group1")
    .startAt(DateBuilder.futureDate(5, DateBuilder.IntervalUnit.MINUTE)) //指定任务开始时间
    .forJob(myJobKey)
    .build();
```

//【实例4】立即触发，每个5分钟执行一次，直到22:00:

```
SimpleScheduleBuilder simpleScheduleBuilder =
SimpleScheduleBuilder.simpleSchedule().withIntervalInMinutes(5).repeatForever();
```

```
SimpleTrigger trigger = (SimpleTrigger) TriggerBuilder.newTrigger()
    .withIdentity("trigger7", "group1")
    .withSchedule( simpleScheduleBuilder )
    .endAt(DateBuilder.dateOf(22, 0, 0)) //endAt 用于指定任务结束调度的时间点
    .build();
```

//【案例5】建立一个触发器，将在下一个小时的整点触发，然后每2小时重复一次

// `DateBuilder.evenHourDate(null)` 获取整点时间

```
SimpleScheduleBuilder simpleScheduleBuilder =
SimpleScheduleBuilder.simpleSchedule().withIntervalInHours(2).repeatForever()
```

```
SimpleTrigger trigger = (SimpleTrigger) TriggerBuilder.newTrigger()
    .withIdentity("trigger8")
    .startAt( DateBuilder.evenHourDate(null) ) //从下一个整点触发调度
    .withSchedule(simpleScheduleBuilder) // 每两小时执行一次，一致循环
    .build();
```

3.2.2 CronTirgger

- CronTrigger通常比SimpleTrigger更有用，如果您需要基于日历的概念而不是按照SimpleTrigger的精确指定间隔进行重新启动的作业启动计划。使用CronTrigger，您可以指时间表，例如“每周五中午”或“每个工作日和上午9:30”，甚至“每周一至周五上午9:00至10点之间每5分钟”和1月份的星期五“。
- 和SimpleTrigger一样，CronTrigger有一个startTime，它指定何时生效，以及一个（可选的）endTime，用于指定何时停止计划。

创建CronTrigger

```
// 通过cron表达式定义触发任务的时间规则
CronScheduleBuilder cronScheduleBuilder = CronScheduleBuilder.cronSchedule("0/3 * * * * ?");

// 触发器
CronTrigger cronTrigger = (CronTrigger) TriggerBuilder.newTrigger()
    .withIdentity("cronTrigger1", "group1")
    .startAt( DateBuilder.futureDate(5, DateBuilder.IntervalUnit.SECOND) ) // 设置首次执行任务的时间
    .withSchedule( cronScheduleBuilder ) // 将时间规则设置给触发器
    .endAt(DateBuilder.dateOf(22,0,0) ) // 设置任务终止的时间
    .build();
```

Cron表达式

Cron Expressions是由七个子表达式组成的字符串（最后一个年份可省略），用于描述日程表的各个细节，这些子表达式用 空格 分隔，并表示：

Seconds	Minutes	Hours	Day-of-Month	Month	Day-of-Week	Year
秒	分	时	日期	月份	星期几	年份

一个完整的Cron-Expressions的例子是字符串 `0 0 12 ? * WED` - 这意味着“每个星期三12:00”。

- 每个部分可以是一个值，也可以是一个范围，例如第六部分：
 - `WED` 表示周三，
 - `MON-WED` 表示周一到周三，
 - `MON, WED` 表示周一和周三
- 使用 `*` 用于说明该字段的“每个可能的值”
- 所有字段都有一组可以指定的有效值：
 - 秒、分钟的数字0到59
 - 小时的值0到23
 - 日期可以是1-31的任何值，但是您需要注意在给定的月份中有多少天
 - 月份可以指定为0到11之间的值，或者使用字符串JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV和DEC。
 - 星期几可以指定为1到7（1=星期日）之间的值，或者使用字符串SUN, MON, TUE, WED, THU, FRI和SAT。
- `/` 字符可用于指定值的增量。
 - 如果在“分钟”字段中输入 `0/15`，则表示“每隔15分钟，从零开始”
 - 如果您在“分钟”字段中使用 `3/18`，则意味着“每隔20分钟，从三分钟开始”
 - 注意 `/35` 的细微之处并不代表“每35分钟” - 这意味着“每隔35分钟，从零开始”，与指定“0,35”相同。

```
// 【cron表达式】 0/16 * * * * ? 2022
```

```
~~~~~2022-01-11 09:48:16
```

```
~~~~~2022-01-11 09:48:32
```

```
~~~~~2022-01-11 09:48:48
```

```
~~~~~2022-01-11 09:49:00
```

```
~~~~~2022-01-11 09:49:16
```

```
~~~~~2022-01-11 09:49:32
```

```
~~~~~2022-01-11 09:49:48
```

```
~~~~~2022-01-11 09:50:00
```

```
~~~~~2022-01-11 09:50:16
```

- **?** 字符是允许的日期和星期几字段，避免指定的日期和星期几出现冲突
- **L** 字符允许用于月、日、星期几，表示最后一月、最后一天、最后一星期；

```
#每月的最后一天
```

```
0 0 12 L * ? 2022
```

```
#每年的最后一个月
```

```
0 0 12 ? L * 2022
```

```
#每个星期的最后一天（7 SAT 周六）
```

```
0 0 12 ? * L 2022
```

```
# 每月最后一个星期一
```

```
0 0 12 ? * MONL 2022
```

- **W** 用于指定最近给定日期的工作日（星期一至星期五）。例如，如果要将“15W”指定为月日期字段的值，则意思是：“最近的平日到当月15日”。
- **#** 用于指定本月的“第n个”xxx工作日。例如，“星期几”字段中的“6 # 3”或“FRI # 3”的值表示“本月的第三个星期五”。

```
# 每月第一个星期一
```

```
0 0 12 ? * MON#1 2022
```

Cron表达式示例

```
#每5分钟就会触发一次
```

```
"0 0/5 * * * ? "
```

```
#每5分钟触发一次，分钟后10秒（即上午10时10分，上午10:05:10等）。
```

```
"10 0/5 * * * ? "
```

```
#在每个星期三和星期五的10:30，11:30，12:30和13:30创建触发器的表达式。
```

```
"0 30 10-13 ? * WED,FRI"
```

```
#每个月5日和20日上午8点至10点之间每半小时触发一次。
```

```
#注意，触发器将不会在上午10点开始，仅在8:00，8:30，9:00和9:30
```

```
"0 0/30 8-9 5,20 * ?"
```

3.2.3 Misfire策略

misfire策略说明

当已经绑定到调度器Scheduler的触发器及任务在执行的过程中，因Scheduler出现故障停止了、或者线程池中没有充足的线程执行到期的任务，从而导致Trigger触发周期到了但是任务没有执行的情况——Misfire

对于支持持久化的Trigger，可以通过指定Misfire策略，来定义其恢复之后的执行策略

对于所有类型的Trigger，Quartz都为其指定了默认的misfire策略——
MISFIRE_INSTRUCTION_SMART_POLICY

- 立即执行misfire的任务，然后继续按计划往后执行

1.SimpleTrigger的Misfire策略

- Quartz为SimpleTrigger提供了六种Misfire策略

#当资源可用时立即执行所有misfire的任务，执行到设置的endTime剩余的周期次数
MISFIRE_INSTRUCTION_FIRE_NOW

#不会判断misfire，当资源可用时立即执行所有misfire的任务，然后按照原计划执行
MISFIRE_INSTRUCTION_IGNORE_MISFIRE_POLICY

#立即执行第一次misfire任务，修改startTime为当前时间，按照原来的时间间隔执行下一次任务（总次数不变）
MISFIRE_INSTRUCTION_RESCHEDULE_NOW_WITH_EXISTING_REPEAT_COUNT

#立即执行第一次misfire任务，修改startTime为当前时间，按照原来的时间间隔执行下一次任务（剩下的任务）
MISFIRE_INSTRUCTION_RESCHEDULE_NOW_WITH_REMAINING_REPEAT_COUNT

#不会立即执行任务，等到下一次计划时间到达时开始执行，忽略已经发生的misfire任务
MISFIRE_INSTRUCTION_RESCHEDULE_NEXT_WITH_EXISTING_COUNT

#不会立即执行任务，等到下一次计划时间到达时开始执行，忽略已经发生的misfire任务
MISFIRE_INSTRUCTION_RESCHEDULE_NEXT_WITH_REMAINING_COUNT

设置SimpleTrigger的Misfire属性示例

```
SimpleScheduleBuilder simpleScheduleBuilder = SimpleScheduleBuilder.simpleSchedule()
    .withIntervalInSeconds(3).repeatForever()
    // .withMisfireHandlingInstructionFireNow()
    // .withMisfireHandlingInstructionIgnoreMisfires()
    // .withMisfireHandlingInstructionNextWithExistingCount()
    // .withMisfireHandlingInstructionNextWithRemainingCount()
    // .withMisfireHandlingInstructionNowWithExistingCount()
    .withMisfireHandlingInstructionNowWithRemainingCount();

SimpleTrigger trigger = TriggerBuilder.newTrigger()
    .withIdentity("trigger1", "group1")
    .withSchedule(simpleScheduleBuilder) .build();
```

2.CronTrigger的Misfire策略

```
# 立即执行所有的misfire任务，然后继续按计划执行
MISFIRE_INSTRUCTION_IGNORE_MISFIRE_POLICY

# 发生的misfire的任务都会被忽略，从当前时间按照原计划执行
MISFIRE_INSTRUCTION_DO_NOTHING

#立即执行第一个发生misfire的任务，忽略其他misfire的任务，然后按照原计划继续执行
MISFIRE_INSTRUCTION_FIRE_ONCE_NOW [default]
```

设置CronTrigger的Misfire属性示例

```
CronScheduleBuilder cronScheduleBuilder = CronScheduleBuilder
    .cronSchedule("0 0 12 ? * MON#1 2022")
    // .withMisfireHandlingInstructionIgnoreMisfires()
    // .withMisfireHandlingInstructionDoNothing()
    .withMisfireHandlingInstructionFireAndProceed();

CronTrigger cronTrigger = (CronTrigger) TriggerBuilder.newTrigger()
    .withIdentity("cronTrigger1", "group1")
    .withSchedule( cronScheduleBuilder )
    .build();
```

3.3 Job与JobDetail

Job

Job表示被调度的任务，在任务调度的业务开发中，quartz提供了Job接口，我们自定义执行任务的类实现此Job接口，在execute方法中完成业务的执行。

- 主要有两种类型的 Job：无状态的（stateless）和有状态的（stateful）。对于同一个 trigger 来说，有状态的 job 不能被并行执行，只有上一次触发的任务被执行完之后，才能触发下一次执行。
- Job 主要有两种属性：volatility 和 durability，其中 volatility 表示任务 是否被持久化 到数据库存储，而 durability 表示在没有 trigger 关联的时候任务是否被保留。两者都是在值为 true 的时候任务被持久化或保留
- 一个 job 可以被多个 trigger 关联，但是一个 trigger 只能关联一个 job。

JobDetail

- JobDetail用于封装Job实例所包含的属性

```
//3.创建任务(JobDetail 用于封装具体的任务)
JobDetail job = JobBuilder.newJob(MyJob.class)
    .withIdentity("job1", "group1") //设置任务的标识
    .build();
```

JobDataMap

- JobDataMap中可以包含不限量的（序列化的）数据对象，在job实例执行的时候，可以使用其中的数据；JobDataMap是Java Map接口的一个实现，额外增加了一些便于存取基本类型的数据的方法。
- 将job加入到scheduler之前，在构建Trigger、JobDetail时，可以将数据放入JobDataMap，如下示例：

```
//1.创建Quartz的任务调度器
Scheduler scheduler = StdSchedulerFactory.getDefaultScheduler();

//2.创建触发器
SimpleScheduleBuilder simpleScheduleBuilder =
SimpleScheduleBuilder.simpleSchedule()
    .withIntervalInSeconds(3)
    .withRepeatCount(10).withMisfireHandlingInstructionNextWithRemainingCount();
SimpleTrigger trigger = TriggerBuilder.newTrigger()
    .withIdentity("trigger1", "group1")
    .withSchedule(simpleScheduleBuilder)
    .usingJobData("key1", "value1") //设置Trigger参数
    .build();

//3.创建任务(JobDetail 用于封装具体的任务)
JobDetail job = JobBuilder.newJob(MyJob.class)
    .withIdentity("job1", "group1")
    .usingJobData("key2", "value2") //设置JobDetail参数
    .build();

scheduler.scheduleJob(job, trigger);
scheduler.start();
```

- 在job的执行过程中，可以从JobDataMap中取出数据，如下示例：

```
/**
 * @Description Quartz任务
 * @Author 千锋涛哥
 * 公众号： Java架构栈
 */
public class MyJob implements Job {

    private SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd hh:mm:ss");

    public void execute(JobExecutionContext jobExecutionContext)
        throws JobExecutionException {
        //执行任务的逻辑
        // 通过execute方法的JobExecutionContext参数获取触发器、JobDetail绑定的参数
        // 1.获取trigger中传递的参数
        JobDataMap jobDataMap1 = jobExecutionContext.getTrigger().getJobDataMap();
        String value1 = (String) jobDataMap1.get("key1");

        // 2.获取JobDetail中传递的参数
        JobDataMap jobDataMap2 =
            jobExecutionContext.getJobDetail().getJobDataMap();
        String value2 = (String) jobDataMap2.get("key2");

        String time = sdf.format(new Date());
        System.out.println("~~~~~"+time+"\t"+value1+"\t"+value2);
    }
}
```

3.4 Listener监听器

Quartz还提供了监听器，用于监听调度器、触发器及任务的相关事件

- JobListener
- TriggerListener
- SchedulerListener

3.4.1 JobListener

JobListener 接收与jobs相关的事件

job相关事件：job即将执行的通知，以及job完成执行时的通知

JobListener使用案例

1. 创建监听

```
/**
 * @Description JobListener任务监听器
 * @Author 千锋涛哥
 * 公众号： Java架构栈
 */
public class MyJobListener implements JobListener {

    //getName返回此监听器的唯一标识字符串
    public String getName() {
        return "MyJobListener1";
    }

    //监听任务开始执行事件，在任务开始之前触发此监听方法
    public void jobToBeExecuted(JobExecutionContext jobExecutionContext) {
        System.out.println("-----jobToBeExecuted");
    }

    //监听任务开始失效事件，在任务失效触发此监听方法
    public void jobExecutionVetoed(JobExecutionContext jobExecutionContext) {
        System.out.println("-----jobExecutionVetoed");
    }

    //监听任务执行完成事件，在任务执行结束之前触发此监听方法
    public void jobWasExecuted(JobExecutionContext jobExecutionContext,
        JobExecutionException e) {
        System.out.println("-----jobWasExecuted");
    }
}
```

2. 注册监听器到Scheduler调度器

一个任务调度器，可以绑定多个任务。当我们向Scheduler注册Job监听器时需要说明是只监听某一个任务，还是监听某一组任务，或者是监听当前调度器的所有任务

```

//a.监听当前调度器上的所有任务
scheduler.getListenerManager().addJobListener(new MyJobListener());

//b.监听group1任务组中的所有任务
scheduler.getListenerManager().addJobListener(new
MyJobListener(), GroupMatcher.jobGroupEquals("group1"));

//c.监听某个特定
scheduler.getListenerManager().addJobListener(new MyJobListener(),
KeyMatcher.keyEquals(new JobKey("job1", "group1")));

```

3.4.2 TriggerListener

TriggerListener接收到与触发器（trigger）相关的事件

trigger相关事件：触发器触发，触发失灵，触发完成（触发器关闭的作业完成）。

注册TriggerListeners的工作原理相同。

1. 创建触发器监听器

```

/**
 * @Description TriggerListener触发器监听器
 * @Author 千锋涛哥
 * 公众号： Java架构栈
 */
public class MyTriggerListener implements TriggerListener {
    public String getName() {
        return "MyTriggerListener1";
    }

    public void triggerFired(Trigger trigger, JobExecutionContext
jobExecutionContext) {
        //任务触发
        System.out.println("~~~~~triggerFired");
    }

    public boolean vetoJobExecution(Trigger trigger, JobExecutionContext
jobExecutionContext) {
        return false;
    }

    public void triggerMisfired(Trigger trigger) {
        //错失触发
        System.out.println("~~~~~triggerMisfired");
    }

    public void triggerComplete(Trigger trigger, JobExecutionContext
jobExecutionContext, Trigger.CompletedExecutionInstruction
completedExecutionInstruction) {
        //触发器触发完成
        System.out.println("~~~~~triggerComplete");
    }
}

```

2. 注册触发器监听器

```
scheduler.getListenerManager().addTriggerListener(new MyTriggerListener());
```

3.4.3 SchedulerListener

SchedulerListener非常类似于TriggerListener和JobListener，除了它们在Scheduler本身中接收到事件的通知 - 不一定与特定触发器（trigger）或job相关的事件。

监听事件：添加job/触发器，删除job/触发器，调度程序中的严重错误，关闭调度程序的通知等。

- SchedulerListener注册到调度程序的ListenerManager。SchedulerListener几乎可以实现任何实现org.quartz.SchedulerListener接口的对象。
- 添加SchedulerListener：

```
scheduler.getListenerManager().addSchedulerListener(mySchedListener);
```

- 删除SchedulerListener：

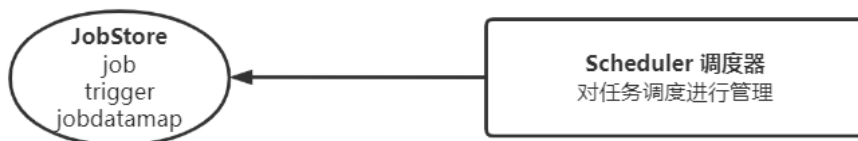
```
scheduler.getListenerManager().removeSchedulerListener(mySchedListener);
```

3.5 Quartz存储JobStore与持久化

3.5.1 JobStore

JobStore负责存储调度器的“工作数据”：任务（Job），触发器（Trigger），数据（JobDataMap）

通俗的讲：JobStore中存储的是什么，调度器就执行什么



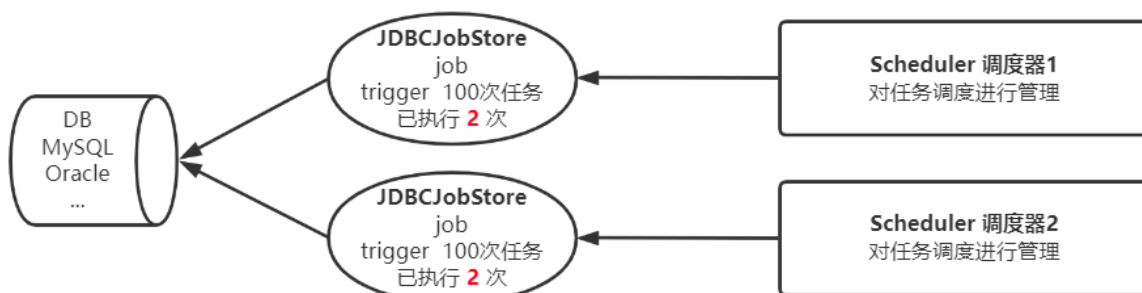
3.5.2 JDBCJobStore持久化

比如我们要执行100次任务，当执行了40次时系统崩溃了，系统重启时任务的执行计数器默认会从0开始。这种策略能够满足多数业务场景需求；但是在某个特定的场景中我们需要继续之前的任务执行，我们可以通过对JobStore进行持久化来实现。

Quartz提供了两种作业类型：

- RAMJobStore 基于内存存储调度器的工作数据 优点：速度快 缺点：不支持持久化
- JDBCJobStore 基于数据库存储调度器的工作数据 优点：支持持久化,支持quartz集群 缺点：速度慢

通过修改quartz的属性配置，将quartz的JobStore修改为JDBCJobStore，并配置数据源；如果是集群部署，只需多个调度器的JDBCJobStore使用相同的数据源即可。

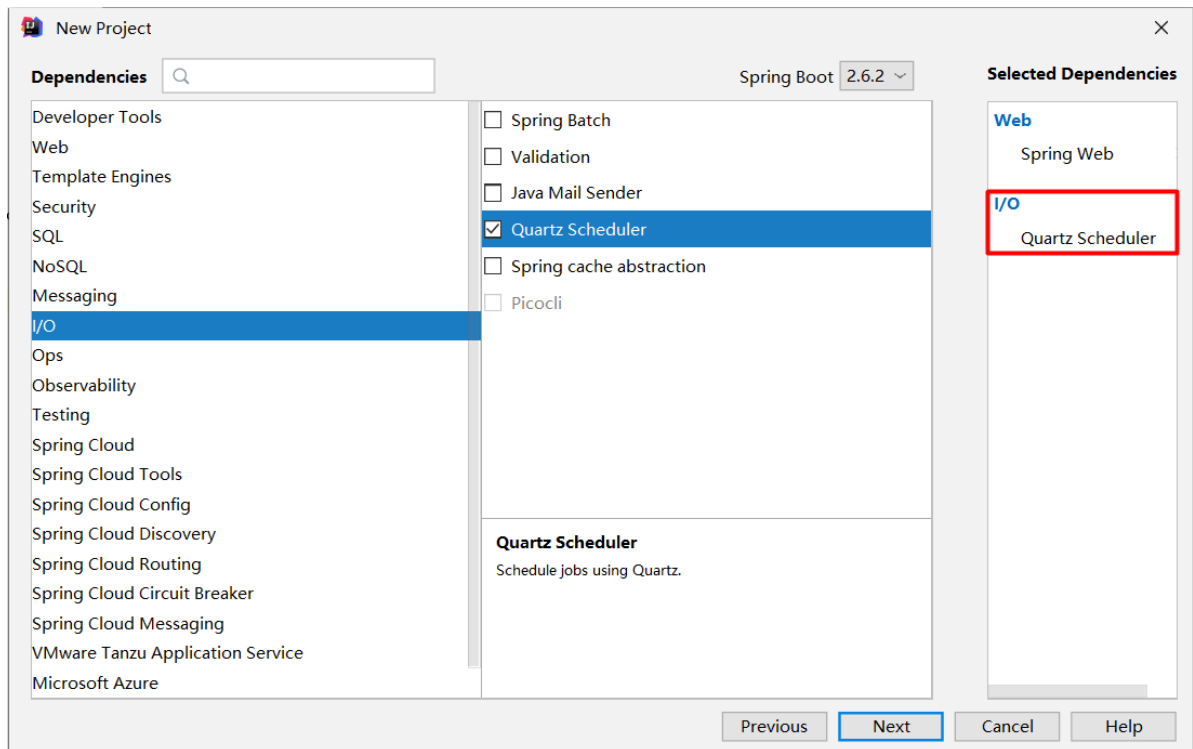


四、在SpringBoot应用中整合Quartz

4.1 创建SpringBoot应用

添加对应的quartz的依赖

- spring web
- quartz-starter 【quartzScheduler】
- mysql (如果quartz要进行集群部署，需要进行持久化，暂时可以不添加)



当我们在SpringBoot应用中，引入quartz-starter之后，SpringBoot应用就会对Quartz进行自动配置，并且初始化调度器对象quartzScheduler

4.2 创建Quartz任务

- 继承 `QuartzJobBean` 类
- 实现 `executeInternal`，在此方法中完成任务逻辑

```
/**
 * @Description quartz任务
 * @Author 千锋涛哥
 * 公众号： Java架构栈
 */
public class MyJob extends QuartzJobBean {

    @Override
    protected void executeInternal(JobExecutionContext context)
        throws JobExecutionException {
        //执行任务的代码逻辑
        String v = (String) context.getTrigger().getJobDataMap().get("key1");

        System.out.println(v+"~~~quartz task");
    }
}
```

```
}
```

4.3 配置触发器

在springboot应用中添加quartz-starter之后，配置的触发器会自动注册到quartzScheduler调度器

```
/**
 * @Description Quartz触发器配置
 * @Author 千锋涛哥
 * 公众号： Java架构栈
 */
@Configuration
public class QuartzConfig {

    @Bean
    public JobDetail getJobDetail(){
        JobDetail job = JobBuilder.newJob(MyJob.class)
            .withIdentity("job1","jobGroup1")
            .storeDurably() //设置当前任务支持持久化
            .build();
        return job;
    }

    @Bean
    public CronTrigger getCronTrigger(JobDetail job){
        CronTrigger cronTrigger = (CronTrigger) TriggerBuilder.newTrigger()
            .withIdentity("cronTrigger1","triggerGroup1")
            .withSchedule(CronScheduleBuilder.cronSchedule("0/3 * * * * ?"))
            .usingJobData("key1","trigger1")
            .forJob(job) //触发器绑定job (一个触发器只能绑定一个job，一个job可以被多个触发
器绑定)
            .build();
        return cronTrigger;
    }

    @Bean("trigger")
    public CronTrigger getCronTrigger2(JobDetail job){
        CronTrigger cronTrigger = (CronTrigger) TriggerBuilder.newTrigger()
            .withIdentity("cronTrigger2","triggerGroup1")
            .withSchedule(CronScheduleBuilder.cronSchedule("0/1 * * * * ?"))
            .usingJobData("key1","trigger2")
            .forJob(job) //触发器绑定job (一个触发器只能绑定一个job，一个job可以被多个触发
器绑定)
            .build();
        return cronTrigger;
    }
}
```