



QUARTZ

Job Scheduler

版本:

Quartz 2.2.3

[官网链接](#)

第一章 Quartz简介

1.1 简介

Quartz 是一种功能丰富的，开放源码的作业调度库，可以在几乎任何Java应用程序集成 - 从最小的独立的应用程序到规模最大电子商务系统。Quartz可以用来创建简单或复杂的日程安排执行几十，几百，甚至是十万的作业数 - 作业被定义为标准的Java组件，可以执行几乎任何东西，可以编程让它们执行。 Quartz调度包括许多企业级功能，如JTA事务和集群支持。

Quartz 是可自由使用，使用Apache 2.0 license授权方式。

Quartz是一个任务调度框架。比如你遇到这样的问题

想每月29号，信用卡自动还款

想每年4月1日自己给当年暗恋女神发一封匿名贺卡

想每隔1小时，备份一下自己的学习笔记到云盘

这些问题总结起来就是：在某一个有规律的时间点干某件事。并且时间的触发的条件可以非常复杂（比如每月最后一个工作日的17:50），复杂到需要一个专门的框架来干这个事。 Quartz就是来干这样的事，你给它一个触发条件的定义，它负责到了时间点，触发相应的Job起来干活

1.2 作用

如果应用程序需要在给定时间执行任务，或者如果系统有连续维护作业，那么Quartz是理想的解决方案。

使用Quartz作业调度应用的示例：

驱动处理工作流程：作为一个新的订单被初始化放置，调度作业到在正好两个小时内，它将检查订单的状态，如果订单确认消息尚未收到命令触发警告通知，以及改变订单的状态为“等待的干预”。

系统维护：调度工作给数据库的内容，每个工作日（节假日除外平日）在11:30 PM转储到一个XML文件中。

在应用程序内提供提醒服务。

1.3 特点

1.3.1 环境

Quartz 可以运行嵌入在另一个独立式应用程序

Quartz 可以在应用程序服务器(或servlet容器)内被实例化，并且参与XA事务

Quartz 可以作为一个独立的程序运行(其自己的Java虚拟机内)，可以通过RMI使用

Quartz 可以被实例化，作为独立的项目集群(负载平衡和故障转移功能)，用于作业的执行

1.3.2 作业调度

作业被安排在一个给定的触发时运行。触发器可以使用以下指令的接近任何组合来创建：

在一天中的某个时间（到毫秒）

在一周的某几天

在每月的某一天

在一年中的某些日期

不在注册的日历中列出的特定日期（如商业节假日除外）

重复特定次数

重复进行，直到一个特定的时间/日期

无限重复

重复的延迟时间间隔

作业是由其创建者赋予的名字，也可以组织成命名组。触发器也可以给予名称和放置在组中，以方便地将它们调度内组织。作业可以被添加到所述调度器一次，而是具有多个触发器注册。在企业Java环境中，作业可以执行自己的工作作为分布式（XA）事务的一部分

1.3.3 作业执行

作业可以实现简单的作业接口，为作业执行工作的任何Java类。

Job类的实例可以通过Quartz被实例化，或者通过应用程序框架。

当触发时，调度通知实现JobListener和TriggerListener接口零个或多个Java对象（监听器可以是简单的Java对象，或EJB，JMS或发布者等）。这些监听器在作业已经执行之后通知。

由于作业完成后返回JobCompletionCode，它通知的成功或失败的调度。JobCompletionCode还可以指示的基础上，成功的话就采取行动调度/失败的代码 - 如立即重新执行作业。

1.3.4 作业持久性

Quartz的设计包括可被实现以提供的作业存储各种机制一个作业存储接口

通过使用包含的JDBCJobStore, 所有的作业和触发器配置为“非挥发性”都存储在通过JDBC关系数据库。

通过使用包含的RAMJobStore, 所有的作业和触发器存储在RAM, 因此不计划执行仍然存在 - 但这是无需使用外部数据库的优势。

第二章 Quartz使用

2.1 基本使用

Maven+Idea

pom.xml

```
<dependencies>
    <!--Quartz任务调度-->
    <!-- https://mvnrepository.com/artifact/org.quartz-scheduler/quartz -->
    <dependency>
        <groupId>org.quartz-scheduler</groupId>
        <artifactId>quartz</artifactId>
        <version>2.2.3</version>
    </dependency>
</dependencies>
```

HelloQuartz 具体的工作类

```
/**
 * 工作类的具体实现
 * */
public class HelloQuartz implements Job {
    //执行
    public void execute(JobExecutionContext context) throws JobExecutionException {
        //创建工作详情
        JobDetail detail=context.getJobDetail();
        //获取工作的名称
        String name=detail.getJobDataMap().getString("name");
        String job=detail.getJobDataMap().getString("job1");

        System.out.println("任务调度: 组: "+job+", 工作名: "+name+"---->今日整点抢购, 不容错过! ");
    }
}
```

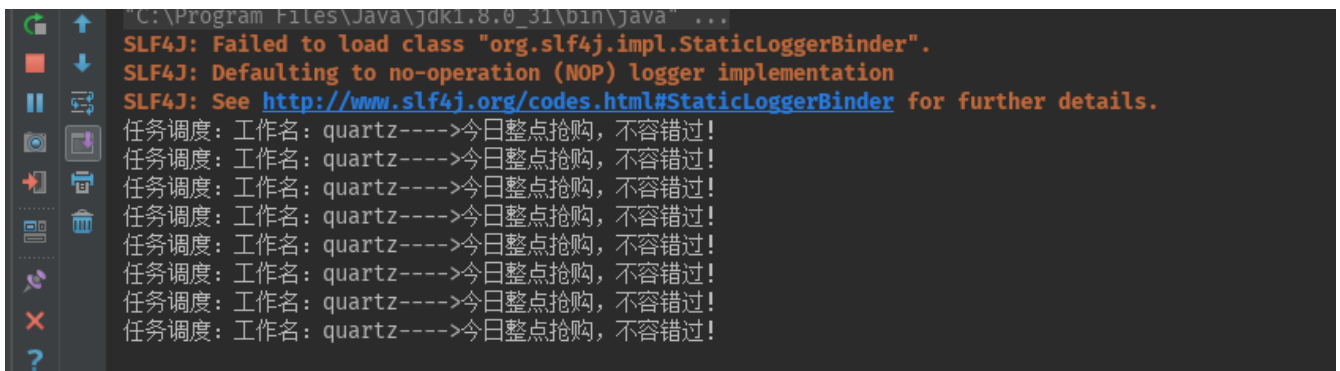
Quartz_1 运行任务调度类

```
public static void main(String[] args) {
    try{
        //创建scheduler, 执行计划
        Scheduler scheduler = StdSchedulerFactory.getDefaultScheduler();
        //定义一个Trigger, 触发条件类
        Trigger trigger = TriggerBuilder.newTrigger().
```

```
        withIdentity("trigger1", "group1") //定义name/group
        .startNow()//一旦加入scheduler, 立即生效
        .withSchedule(SimpleScheduleBuilder.simpleSchedule() //使用SimpleTrigger
            .withIntervalInSeconds(1) //每隔一秒执行一次
            .repeatForever()) //一直执行, 奔腾到老不停歇
        .build();
//定义一个JobDetail
JobDetail job = JobBuilder.newJob(HelloQuartz.class) //定义Job类为HelloQuartz类, 这是真正的执行逻辑所在
    .withIdentity("job1", "group1") //定义name/group
    .usingJobData("name", "quartz") //定义属性
    .build();
//加入这个调度
scheduler.scheduleJob(job, trigger);
//启动任务调度
scheduler.start();

} catch (Exception ex){
    ex.printStackTrace();
}
}
```

运行结果:



```
C:\Program Files\Java\jdk1.8.0_31\bin\java" ...
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
任务调度: 工作名: quartz---->今日整点抢购, 不容错过!
任务调度: 工作名: quartz---->今日整点抢购, 不容错过!
任务调度: 工作名: quartz---->今日整点抢购, 不容错过!
任务调度: 工作名: quartz---->今日整点抢购, 不容错过!
任务调度: 工作名: quartz---->今日整点抢购, 不容错过!
任务调度: 工作名: quartz---->今日整点抢购, 不容错过!
任务调度: 工作名: quartz---->今日整点抢购, 不容错过!
```

2.2 核心类说明

Scheduler: 调度器。所有的调度都是由它控制

Scheduler就是Quartz的大脑, 所有任务都是由它来设施

Scheduler包含一个两个重要组件: JobStore和ThreadPool

JobStore是会来存储运行时信息的, 包括Trigger, Scheduler, JobDetail, 业务锁等

ThreadPool就是线程池, Quartz有自己的线程池实现。所有任务的都会由线程池执行

SchedulerFactory, 顾名思义就是来用创建Scheduler了, 有两个实现: DirectSchedulerFactory和StdSchedulerFactory。前者可以用来在代码里定制你自己的Scheduler参数。后者是直接读取classpath下的quartz.properties (不存在就都使用默认值) 配置来实例化Scheduler。通常来讲, 我们使用StdSchedulerFactory也就足够了。

SchedulerFactory本身是支持创建RMI stub的, 可以用来管理远程的Scheduler, 功能与本地一样

quartz.properties 常用配置示例:

```
org.quartz.scheduler.instanceName = DefaultQuartzScheduler
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.class = org.quartz.simpl.RAMJobStore
```

Trigger: 定义触发的条件。可以使用SimpleTrigger, 每隔1秒中执行一次

JobDetail & Job: JobDetail 定义的是任务数据, 而真正的执行逻辑是在Job中。为什么设计成JobDetail + Job, 不直接使用Job? 这是因为任务是有可能并发执行, 如果Scheduler直接使用Job, 就会存在对同一个Job实例并发访问的问题。而JobDetail & Job 方式, scheduler每次执行, 都会根据JobDetail创建一个新的Job实例, 这样就可以规避并发访问的问题

JobDetail和Trigger都有name和group。
name是它们在这个scheduler里面的唯一标识。如果我们要更新一个JobDetail定义, 只需要设置一个name相同的JobDetail实例即可。
group是一个组织单元, scheduler会提供一些对整组操作的API, 比如 scheduler.resumeJobs()。

2.3 Trigger

2.3.1 trigger常用属性

StartTime & EndTime

StartTime & EndTime

startTime和endTime指定的Trigger会被触发的时间区间。在这个区间之外, Trigger是会被触发的。
所有Trigger都会包含这两个属性

Priority

当scheduler比较繁忙的时候, 可能在同一个时刻, 有多个Trigger被触发了, 但资源不足 (比如线程池不足)。那么这个时候比剪刀石头布更好的方式, 就是设置优先级。优先级高的先执行。

需要注意的是, 优先级只有在同一时刻执行的Trigger之间才会起作用, 如果一个Trigger是9:00, 另一个Trigger是9:30。那么无论后一个优先级多高, 前一个都是先执行。

优先级的值默认是5, 当为负数时使用默认值。最大值似乎没有指定, 但建议遵循Java的标准, 使用1-10, 不然鬼才知道看到【优先级为10】是时, 上头还有没有更大的值。

Misfire

Misfire(错失触发) 策略类似的Scheduler资源不足的时候, 或者机器崩溃重启等, 有可能某一些Trigger在应该触发的时间点没有被触发, 也就是Miss Fire了。这个时候Trigger需要一个策略来处理这种情况。每种Trigger可选的策略各不相同。

这里有两个点需要重点关注:

MisFire的触发是有一个阈值, 这个阈值是配置在JobStore的。比RAMJobStore是org.quartz.jobStore.misfireThreshold。只有超过这个阈值, 才会算MisFire。小于这个阈值, Quartz是会全部重新触发。

所有MisFire的策略实际上都是解答两个问题:

1. 已经MisFire的任务还要重新触发吗?
2. 如果发生MisFire, 要调整现有的调度时间吗?

SimpleTrigger的MisFire策略有

MISFIRE_INSTRUCTION_IGNORE_MISFIRE_POLICY

这个不是忽略已经错失的触发的意思, 而是说忽略MisFire策略。它会在资源合适的时候, 重新触发所有的MisFire任务, 并且不会影响现有的调度时间。

比如, SimpleTrigger每15秒执行一次, 而中间有5分钟时间它都MisFire了, 一共错过了20个, 5分钟后, 假设资源充足了, 并且任务允许并发, 它会被一次性触发。

这个属性是所有Trigger都适用。

MISFIRE_INSTRUCTION_FIRE_NOW

忽略已经MisFire的任务, 并且立即执行调度。这通常只适用于只执行一次的任务。

MISFIRE_INSTRUCTION_RESCHEDULE_NOW_WITH_EXISTING_REPEAT_COUNT

将startTime设置当前时间, 立即重新调度任务, 包括的MisFire的

MISFIRE_INSTRUCTION_RESCHEDULE_NOW_WITH_REMAINING_REPEAT_COUNT

类似MISFIRE_INSTRUCTION_RESCHEDULE_NOW_WITH_EXISTING_REPEAT_COUNT, 区别在于会忽略已经MisFire的任务

MISFIRE_INSTRUCTION_RESCHEDULE_NEXT_WITH_EXISTING_COUNT

在下一次调度时间点, 重新开始调度任务, 包括的MisFire的

MISFIRE_INSTRUCTION_RESCHEDULE_NEXT_WITH_REMAINING_COUNT

类似于MISFIRE_INSTRUCTION_RESCHEDULE_NEXT_WITH_EXISTING_COUNT, 区别在于会忽略已经MisFire的任务。

MISFIRE_INSTRUCTION_SMART_POLICY

所有的Trigger的MisFire默认值都是这个, 大致意思是“把处理逻辑交给聪明的Quartz去决定”。基本策略是,

如果是只执行一次的调度, 使用MISFIRE_INSTRUCTION_FIRE_NOW

如果是无限次的调度(repeatCount是无限的), 使用

MISFIRE_INSTRUCTION_RESCHEDULE_NEXT_WITH_REMAINING_COUNT

否则, 使用MISFIRE_INSTRUCTION_RESCHEDULE_NOW_WITH_EXISTING_REPEAT_COUNT

Calendar

Calendar不是jdk的java.util.Calendar，不是为了计算日期的。它的作用是在于补充Trigger的时间。可以排除或加入某一些特定的时间点。

以“每月29日零点自动还信用卡”为例，我们想排除掉每年的2月29号零点这个时间点（因为平年和闰年2月不一样）。这个时间，就可以用Calendar来实现

Quartz提供以下几种Calendar，注意，所有的Calendar既可以是排除，也可以是包含，取决于：

HolidayCalendar。指定特定的日期，比如20140613。精度到天。

DailyCalendar。指定每天的时间段（rangeStartingTime，rangeEndingTime），格式是HH:MM[:SS[:mmm]]。也就是最大精度可以到毫秒。

WeeklyCalendar。指定每星期的星期几，可选值比如为java.util.Calendar.SUNDAY。精度是天。

MonthlyCalendar。指定每月的几号。可选值为1-31。精度是天

AnnualCalendar。指定每年的哪一天。使用方式如上例。精度是天。

CronCalendar。指定Cron表达式。精度取决于Cron表达式，也就是最大精度可以到秒。

2.3.2 Trigger实现类

SimpleTrigger

指定从某一个时间开始，以一定的时间间隔（单位是毫秒）执行的任务。

它适合的任务类似于：9:00 开始，每隔1小时，执行一次。

它的属性有：

repeatInterval 重复间隔

repeatCount 重复次数。实际执行次数是 repeatCount+1。因为在startTime的时候一定会执行一次。

示例：

```
SimpleScheduleBuilder.simpleSchedule().  
    withIntervalInSeconds(10) //每隔10秒执行一次  
    repeatForever() //永远执行  
    build();
```

```
SimpleScheduleBuilder.simpleSchedule().  
    withIntervalInMinutes(3) //每隔3分钟执行一次  
    withRepeatCount(3) //执行3次  
    build();
```

CalendarIntervalTrigger

类似于SimpleTrigger，指定从某一个时间开始，以一定的时间间隔执行的任务。但是不同的是SimpleTrigger指定的时间间隔为毫秒，没办法指定每隔一个月执行一次（每月的时间间隔不是固定值），而CalendarIntervalTrigger支持的间隔单位有秒，分钟，小时，天，月，年，星期。

相较于SimpleTrigger有两个优势：1、更方便，比如每隔1小时执行，你不用自己去计算1小时等于多少毫秒。2、支持不是固定长度的间隔，比如间隔为月和年。但劣势是精度只能到秒。

它适合的任务类似于：9:00 开始执行，并且以后每周 9:00 执行一次

它的属性有：

interval 执行间隔

intervalUnit 执行间隔的单位（秒，分钟，小时，天，月，年，星期）

示例：

```
CalendarIntervalScheduleBuilder.calendarIntervalSchedule().  
    withIntervalInDays(2) //每2天执行一次  
    .build();
```

```
CalendarIntervalScheduleBuilder.calendarIntervalSchedule().  
    withIntervalInWeeks(1) //每周执行一次  
    .build();
```

DailyTimeIntervalTrigger

指定每天的某个时间段内，以一定的时间间隔执行任务。并且它可以支持指定星期。

它适合的任务类似于：指定每天9:00 至 18:00 ，每隔70秒执行一次，并且只要周一至周五执行。

它的属性有：

startTimeOfDay 每天开始时间

endTimeOfDay 每天结束时间

daysOfWeek 需要执行的星期

interval 执行间隔

intervalUnit 执行间隔的单位（秒，分钟，小时，天，月，年，星期）

repeatCount 重复次数

示例：

```
DailyTimeIntervalScheduleBuilder.dailyTimeIntervalSchedule()  
    .startingDailyAt(TimeOfDay.hourAndMinuteOfDay(9, 0)) //每天9: 00开始  
    .endingDailyAt(TimeOfDay.hourAndMinuteOfDay(18, 0)) //18: 00 结束  
    .onDaysOfTheWeek(MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY) //周一至周五执行  
    .withIntervalInHours(1) //每间隔1小时执行一次  
    .withRepeatCount(100) //最多重复100次（实际执行100+1次）  
    .build();
```

```
DailyTimeIntervalScheduleBuilder.dailyTimeIntervalSchedule()  
    .startingDailyAt(TimeOfDay.hourAndMinuteOfDay(10, 0)) //每天10: 00开始  
    .endingDailyAfterCount(10) //每天执行10次，这个方法实际上根据  
startTimeOfDay+interval*count 算出 endTimeOfDay  
    .onDaysOfTheWeek(MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY) //周一至周五执行  
    .withIntervalInHours(1) //每间隔1小时执行一次  
    .build();
```

CronTrigger

适合于更复杂的任务，它支持类型于Linux Cron的语法（并且更强大）。基本上它覆盖了以上三个Trigger的绝大部分能力（但不是全部）——当然，也更难理解。

它适合的任务类似于：每天0:00, 9:00, 18:00各执行一次。

它的属性只有：

Cron表达式。但这个表示式本身就够复杂了

示例：


```
CronScheduleBuilder.cronSchedule("0 0/2 10-12 * * ?") // 每天10:00-12:00, 每隔2分钟执行一次
.build();
```

```
cronSchedule("0 30 9 ? * MON") // 每周一, 9:30执行一次
.build();
```

```
CronScheduleBuilder.weeklyOnDayAndHourAndMinute(MONDAY, 9, 30) //等同于 0 30 9 ? * MON
.build();
```

2.3.3 Cron表达式

位置	时间域	允许值	特殊值
1	秒	0-59	, - * /
2	分钟	0-59	, - * /
3	小时	0-23	, - * /
4	日期	1-31	, - * ? / L W C
5	月份	1-12	, - * /
6	星期	1-7	, - * ? / L C #
7	年份 (可选)	1-31	, - * /

星号(*)：可用在所有字段中，表示对应时间域的每一个时刻，例如， 在分钟字段时，表示“每分钟”；

问号(?)：该字符只在日期和星期字段中使用，它通常指定为“无意义的值”，相当于点位符；

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“MON,WED,FRI”，则表示星期一，星期三和星期五；

斜杠(/)：x/y表达一个等步长序列，x为起始值，y为增量步长值。如在分钟字段中使用0/15，则表示为0,15,30和45秒，而5/15在分钟字段中表示5,20,35,50，你也可以使用*/y，它等同于0/y；

L：该字符只在日期和星期字段中使用，代表“Last”的意思，但它在两个字段中意思不同。L在日期字段中，表示这个月份的最后一天，如一月的31号，非闰年二月的28号；如果L用在星期中，则表示星期六，等同于7。但是，如果L出现在星期字段里，而且在前面有一个数值x，则表示“这个月的最后x天”，例如，6L表示该月的最后星期五；

w：该字符只能出现在日期字段里，是对前导日期的修饰，表示离该日期最近的工作日。例如15w表示离该月15号最近的工作日，如果该月15号是星期六，则匹配14号星期五；如果15日是星期日，则匹配16号星期一；如果15号是星期二，那结果就是15号星期二。但必须注意关联的匹配日期不能够跨月，如你指定1w，如果1号是星期六，结果匹配的是3号星期一，而非上个月最后的那天。w字符串只能指定单一日期，而不能指定日期范围；

LW组合：在日期字段可以组合使用LW，它的意思是当月的最后一个工作日；

井号(#)：该字符只能在星期字段中使用，表示当月某个工作日。如6#3表示当月的第三个星期五(6表示星期五，#3表示当前的第三个)，而4#5表示当月的第五个星期三，假设当月没有第五个星期三，忽略不触发；

C: 该字符只在日期和星期字段中使用, 代表“Calendar”的意思。它的意思是计划所关联的日期, 如果日期没有被关联, 则相当于日历中所有日期。例如5C在日期字段中就相当于日历5日以后的第一天。1C在星期字段中相当于星期日后的第一天。

表达式示例:

表示式	说明
0 0 12 * * ?	每天12点运行
0 15 10 ? * *	每天10:15运行
0 15 10 * * ?	每天10:15运行
0 15 10 * * ? *	每天10:15运行
0 15 10 * * ? 2008	在2008年的每天10: 15运行
0 * 14 * * ?	每天14点到15点之间每分钟运行一次, 开始于14:00, 结束于14:59。
0 0/5 14 * * ?	每天14点到15点每5分钟运行一次, 开始于14:00, 结束于14:55。
0 0/5 14,18 * * ?	每天14点到15点每5分钟运行一次, 此外每天18点到19点每5分钟也运行一次。
0 0-5 14 * * ?	每天14:00点到14:05, 每分钟运行一次。
0 10,44 14 ? 3 WED	3月每周三的14:10分和14:44, 每分钟运行一次。
0 15 10 ? * MON-FRI	每周一, 二, 三, 四, 五的10:15分运行。
0 15 10 15 * ?	每月15日10:15分运行。
0 15 10 L * ?	每月最后一天10:15分运行。
0 15 10 ? * 6L	每月最后一个星期五10:15分运行。
0 15 10 ? * 6L 2007-2009	在2007,2008,2009年每个月的最后一个星期五的10:15分运行。
0 15 10 ? * 6#3	每月第三个星期五的10:15分运行。

2.4 Job&JobDetail

2.4.1 JobDetail

JobDetail是任务的定义, 而Job是任务的执行逻辑。在JobDetail里会引用一个Job Class定义

任务步骤:

- 1、创建一个org.quartz.Job的实现类, 并实现自己的业务逻辑。比如上面的DoNothingJob。
- 2、定义一个JobDetail, 引用这个实现类
- 3、加入scheduleJob

核心代码：

```
JobClass jobClass=JobDetail.getJobClass()
```

Job jobInstance=jobClass.newInstance()。所以Job实现类，必须有一个public的无参构建方法。

jobInstance.execute(JobExecutionContext context)。JobExecutionContext是Job运行的上下文，可以获得Trigger、Scheduler、JobDetail的信息。

也就是说，每次调度都会创建一个新的Job实例，这样的好处是有些任务并发执行的时候，不存在对临界资源的访问问题——当然，如果需要共享JobDataMap的时候，还是存在临界资源的并发访问的问题。

JobDataMap

Job都次都是newInstance的实例，那我怎么传值给它？比如我现在有两个发送邮件的任务，一个是发给"liLei"，一个发给"hanmeimei"，不能说我要写两个Job实现类LiLeiSendEmailJob和HanMeiMeiSendEmailJob。实现的办法是通过JobDataMap。

每一个JobDetail都会有一个JobDataMap。JobDataMap本质就是一个Map的扩展类，只是提供了一些更便捷的方法，比如getString()之类的。

我们可以在定义JobDetail，加入属性值，方式有二：

第一种：

```
newJob().usingJobData("age", 18) //加入属性到ageJobDataMap
```

第二种：

```
job.getJobDataMap().put("name", "quertz"); //加入属性name到JobDataMap
```

在Job中可以获取这个JobDataMap的值，方式同样有二：

```
JobDetail detail = context.getJobDetail();
```

```
JobDataMap map = detail.getJobDataMap(); //方法一：获得JobDataMap
```

```
private String name;
```

```
//方法二：属性的setter方法，会将JobDataMap的属性自动注入
```

```
public void setName(String name) {
```

```
    this.name = name;
```

```
}
```

对于同一个JobDetail实例，执行的多个Job实例，是共享同样的JobDataMap，也就是说，如果你在任务里修改了里面的值，会对其他Job实例（并发的或者后续的）造成影响。

除了JobDetail，Trigger同样有一个JobDataMap，共享范围是所有使用这个Trigger的Job实例

2.4.2 Job并发

job是有可能并发执行的，比如一个任务要执行10秒中，而调度算法是每秒中触发1次，那么就有可能多个任务被并发执行。

有时候我们并不想任务并发执行，比如这个任务要去“获得数据库中所有未发送邮件的名单”，如果是并发执行，就需要一个数据库锁去避免一个数据被多次处理。这个时候一个@DisallowConcurrentExecution解决这个问题

```
public class DoNothingJob implements Job {
    @DisallowConcurrentExecution
    public void execute(JobExecutionContext context) throws JobExecutionException {
        System.out.println("操作");
    }
}
```

注意, @DisallowConcurrentExecution是对JobDetail实例生效, 也就是如果你定义两个JobDetail, 引用同一个Job类, 是可以并发执行的

JobExecutionException

Job.execute()方法是不允许抛出除JobExecutionException之外的所有异常的 (包括RuntimeException), 所以编码的时候, 最好是try-catch住所有的Throwable, 小心处理。

代码示例:

```
public class MyJob implements Job {
    public void execute(JobExecutionContext context) throws JobExecutionException {
        System.out.println("双11秒杀通知");
    }
}
```

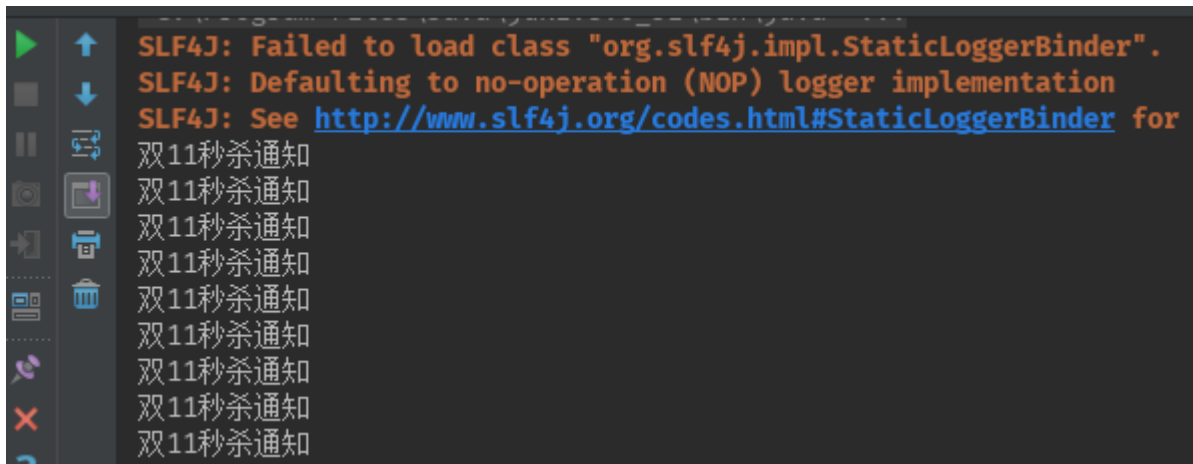
```
public class Quartz_2 {

    public static void main(String[] args) throws Exception{
        JobDetail job=newJob()
            .ofType(MyJob.class) //引用Job class
            .withIdentity("job1", "group1") //设置name/group
            .withDescription("this is a test job") //设置描述
            .usingJobData("age", 18) //加入属性到age, JobDataMap
            .build();

        job.getJobDataMap().put("name", "quartz"); //加入属性name到JobDataMap

        //定义一个每秒执行一次的SimpleTrigger
        Trigger trigger=newTrigger()
            .startNow()
            .withIdentity("trigger1")
            .withSchedule(simpleSchedule()
                .withIntervalInSeconds(1)
                .repeatForever())
            .build();
        //创建任务调度对象
        Scheduler sche= StdSchedulerFactory.getDefaultScheduler();
        //添加工作计划
        sche.scheduleJob(job, trigger);
        //启动任务调度
        sche.start();
        Thread.sleep(10000);
        //关闭任务调度
    }
}
```

```
sche.shutdown();  
}  
}
```



第三章 SpringBoot整合Quartz

使用springboot自带的定时任务可以很简单很方便的完成一些简单的定时任务，但是我们想动态的执行我们的定时任务就比较困难了。然而使用quartz却可以很容易的管理我们的定时任务，很容易动态的操作定时任务。下面我们就讲解下如何使用quartz动态实现定时任务！

3.1 pom.xml引入依赖

```
<dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-web</artifactId>  
</dependency>  
<!--quartz相关依赖-->  
<dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-quartz</artifactId>  
</dependency>  
<!--lombok库提供了一些注解来简化java代码,如@Slf4j -->  
<dependency>  
    <groupId>org.projectlombok</groupId>  
    <artifactId>lombok</artifactId>  
</dependency>
```

3.2 编写job接口

MyJob实现Job接口,重写execute方法在里面操作我们要执行的业务逻辑。

3.3 定时任务管理

定义QuartzManager管理我们的定时任务方法

```
@Configuration
public class QuartzManager {
    public static final String JOB1="job1";
    public static final String GROUP1="group1";
    /**默认每个星期凌晨一点执行*/
    //public static final String DEFAULT_CRON="0 0 1 ? * L";
    /**默认5秒执行一次*/
    public static final String DEFAULT_CRON="*/5 * * * * ?";

    /**
     * 任务调度
     */
    @Autowired
    private Scheduler scheduler;

    /**
     * 开始执行定时任务
     */
    public void startJob() throws SchedulerException {
        startJobTask(scheduler);
        scheduler.start();
    }

    /**
     * 启动定时任务
     * @param scheduler
     */
    private void startJobTask(Scheduler scheduler) throws SchedulerException {
        JobDetail jobDetail=
JobBuilder.newJob(MyJob.class).withIdentity(JOB1,GROUP1).build();
        CronScheduleBuilder
cronScheduleBuilder=CronScheduleBuilder.cronSchedule(DEFAULT_CRON);
        CronTrigger cronTrigger=TriggerBuilder.newTrigger().withIdentity(JOB1,GROUP1)
                .withSchedule(cronScheduleBuilder).build();
        scheduler.scheduleJob(jobDetail,cronTrigger);

    }

    /**
     * 获取Job信息
     * @param name
     * @param group
     */
    public String getjobInfo(String name,String group) throws SchedulerException {
        TriggerKey triggerKey=new TriggerKey(name,group);
        CronTrigger cronTrigger= (CronTrigger) scheduler.getTrigger(triggerKey);
        return String.format("time:%s,state:%s",cronTrigger.getCronExpression(),
                scheduler.getTriggerState(triggerKey).name());
    }

    /**
     * 修改任务的执行时间
     * @param name
     * @param group
     */
}
```

```
* @param cron cron表达式
* @return
* @throws SchedulerException
*/
public boolean modifyJob(String name,String group,String cron) throws
SchedulerException{
    Date date=null;
    TriggerKey triggerKey=new TriggerKey(name, group);
    CronTrigger cronTrigger= (CronTrigger) scheduler.getTrigger(triggerKey);
    String oldTime=cronTrigger.getCronExpression();
    if (!oldTime.equalsIgnoreCase(cron)){
        CronScheduleBuilder cronScheduleBuilder=CronScheduleBuilder.cronSchedule(cron);
        CronTrigger trigger=TriggerBuilder.newTrigger().withIdentity(name,group)
            .withSchedule(cronScheduleBuilder).build();
        date=scheduler.rescheduleJob(triggerKey,trigger);
    }
    return date !=null;
}

/**
 * 暂停所有任务
 * @throws SchedulerException
 */
public void pauseAllJob()throws SchedulerException{
    scheduler.pauseAll();
}

/**
 * 暂停某个任务
 * @param name
 * @param group
 * @throws SchedulerException
 */
public void pauseJob(String name,String group)throws SchedulerException{
    JobKey jobKey=new JobKey(name,group);
    JobDetail jobDetail=scheduler.getJobDetail(jobKey);
    if (jobDetail==null)
        return;
    scheduler.pauseJob(jobKey);
}

/**
 * 恢复所有任务
 * @throws SchedulerException
 */
public void resumeAllJob()throws SchedulerException{
    scheduler.resumeAll();
}

/**
 * 恢复某个任务
 */
public void resumeJob(String name,String group)throws SchedulerException{
```

```
        JobKey jobKey=new JobKey(name,group);
        JobDetail jobDetail=scheduler.getJobDetail(jobKey);
        if (jobDetail==null)
            return;
        scheduler.resumeJob(jobKey);
    }

    /**
     * 删除某个任务
     * @param name
     * @param group
     * @throws SchedulerException
     */
    public void deleteJob(String name,String group)throws SchedulerException{
        JobKey jobKey=new JobKey(name, group);
        JobDetail jobDetail=scheduler.getJobDetail(jobKey);
        if (jobDetail==null)
            return;
        scheduler.deleteJob(jobKey);
    }
}
```

3.4 启动我们的定时任务

定义ApplicationStartQuartzJobListener实现当应用启动,或刷新时, 启动我们的定时任务.

```
@Slf4j
@Configuration
public class ApplicationStartQuartzJobListener implements
    ApplicationListener<ContextRefreshedEvent> {

    @Autowired
    private QuartzManager quartzManager;

    @Override
    public void onApplicationEvent(ContextRefreshedEvent contextRefreshedEvent) {
        try {
            quartzManager.startJob();
            log.info("任务已经启动.....");
        } catch (SchedulerException e) {
            e.printStackTrace();
        }
    }

    /**
     * 初始注入scheduler
     */
    @Bean
    public Scheduler scheduler() throws SchedulerException{
        SchedulerFactory schedulerFactoryBean = new StdSchedulerFactory();
        return schedulerFactoryBean.getScheduler();
    }
}
```



```
}  
}
```

3.5 创建Quartz实例

定义MyJobFactory创建Quartz实例

```
@Component  
public class MyJobFactory extends AdaptableJobFactory{  
    @Autowired  
    private AutowireCapableBeanFactory capableBeanFactory;  
    @Override  
    protected Object createJobInstance(TriggerFiredBundle bundle) throws Exception {  
        // 调用父类的方法  
        Object jobInstance = super.createJobInstance(bundle);  
        // 进行注入  
        capableBeanFactory.autowireBean(jobInstance);  
        return jobInstance;  
    }  
}
```

3.6 Application注入MyJobFactory和Scheduler

```
@SpringBootApplication(exclude = DataSourceAutoConfiguration.class)  
@Configuration  
public class Start {  
    public static void main(String[] args) {  
        SpringApplication.run(Start.class, args);  
    }  
  
    @Autowired  
    private MyJobFactory myJobFactory;  
  
    @Bean  
    public SchedulerFactoryBean schedulerFactoryBean() {  
        SchedulerFactoryBean schedulerFactoryBean = new SchedulerFactoryBean();  
        schedulerFactoryBean.setJobFactory(myJobFactory);  
        System.out.println("myJobFactory:"+myJobFactory);  
        return schedulerFactoryBean;  
    }  
    @Bean  
    public Scheduler scheduler() {  
        return schedulerFactoryBean().getScheduler();  
    }  
}
```

3.7 定义Controller动态修改定时任务

```
@RestController
public class ModifyCronController {
    @Autowired
    private QuartzManager quartzManager;
    @GetMapping("modify")
    public String modify() throws SchedulerException {
        /**10秒执行一次*/
        String cron="*/10 * * * * ?";
        quartzManager.pauseJob(QuartzManager.JOB1,QuartzManager.GROUP1);
        quartzManager.modifyJob(QuartzManager.JOB1,QuartzManager.GROUP1,cron);
        return "ok";
    }
}
```

第四章 定时任务集群

4.1 数据量评估

定时任务一般的使用场景是定时查询出一批数据，对这一批数据进行业务操作。

根据数据量的大小决定是否使用分布式任务，如果数据量不大或者实时性要求不高，单机任务就够了，也可以减少相应复杂度。

如果数据量大就需要部署分布式任务。分布式集群中的节点对数据进行分片处理，即每个节点拿一部分数据进行业务处理。

4.2 仅解决并发问题方案

(1) Quartz + 分布式锁

假设定时任务部署了3台机器，在任务启动时3台机器竞争分布式锁，谁竞争到谁就执行，剩下2台不执行。分布式锁可以使用Redis或者Zookeeper

(2) 开关方案

方案一：配置文件中设置开关是否开启，执行任务前读取该开关，开启则执行。这种方式实现比较简单，但是如果需要换另外一台机器执行，必须修改配置项并发布项目，维护成本较高

方案二：建一个数据库配置表，配置表中配置可执行任务的机器标识，每台机器执行前读取这个配置，看看是否是本机。如果是则执行，否则不执行

方案三：Zookeeper做一个全局配置，配置项内容是可执行任务的机器标识，执行原理同方案一

4.3 真正分布式任务方案

(1) Quartz官方分布式方案

这种方式比较重，需要根据官方文档新建数据表，并不推荐

(2) Elastic Job

当当网开源的一个分布式调度解决方案，在互联网公司比较通用

(3) 自研分布式任务平台

有一定技术实力的公司，可以选择自研分布式任务平台

第五章 Elastic Job

Elastic job是当当网基于Zookeeper、Quartz开发并开源的一个Java分布式定时任务，解决了Quartz不支持分布式的弊端。Elastic job主要的功能有支持弹性扩容，通过Zookeeper集中管理和监控job，支持失效转移等。项目由两个相互独立的子项目Elastic-Job-Lite和Elastic-Job-Cloud组成。

大多数情况下，定时任务我们一般使用quartz开源框架就能满足应用场景。但如果考虑到健壮性等其它一些因素，就需要自己下点工夫，比如：要避免单点故障，至少得部署2个节点吧，但是部署多个节点，又有其它问题，有些数据在某一个时刻只能处理一次，比如 $i = i + 1$ 这些无法保证幂等的操作，run多次跟run一次，完全是不同的效果。

对于上面的问题，可以使用quartz+zk或redis分布式锁的解决方案：

- 1、每类定时job，可以分配一个独立的标识（比如：xxx_job）
- 2、这类job的实例，部署在多个节点上时，每个节点启动前，向zk申请一个分布式锁（在xxx_job节点下）
- 3、拿到锁的实例，才允许启动定时任务(通过代码控制quartz的schedule)，没拿到锁的，处于standby状态，一直监听锁的变化
- 4、如果某个节点挂了，分布式锁自动释放，其它节点这时会抢到锁，按上面的逻辑，就会从standby状态，转为激活状态，小三正式上位，继续执行定时job。

这个方案，基本上解决了HA(High Availability)和业务正确性的问题，但是美中不足的地方有2点：

- 1、无法充分利用机器性能，处于standby的节点，实际上只是一个备胎，平时啥也不干
- 2、性能不方便扩展，比如：某个job一次要处理上千万的数据，仅1个激活节点，要处理很久

好了，前戏铺垫了这么久，该请主角登场了，elastic-job相当于quartz+zk的加强版，它允许对定时任务分片，可以集群部署(每个job的"分片"会分散到各个节点上)，如果某个节点挂了，该节点上的分片，会调度到其它节点上。般情况下，使用SimpleJob这种就可以了。

5.1 概述

官网地址：<http://elasticjob.io>

Elastic-Job是一个分布式调度解决方案，由两个相互独立的子项目Elastic-Job-Lite和Elastic-Job-Cloud组成。

Elastic-Job-Lite定位为轻量级无中心化解决方案，使用jar包的形式提供分布式任务的协调服务。

5.2 基本概念

1. 分片概念

任务的分布式执行，需要将一个任务拆分为多个独立的任务项，然后由分布式的服务器分别执行某一个或几个分片项。

例如：有一个遍历数据库某张表的作业，现有2台服务器。为了快速的执行作业，那么每台服务器应执行作业的50%。为满足此需求，可将作业分成2片，每台服务器执行1片。作业遍历数据的逻辑应为：服务器A遍历ID以奇数结尾的数据；服务器B遍历ID以偶数结尾的数据。如果分成10片，则作业遍历数据的逻辑应为：每片分到的分片项应为ID%10，而服务器A被分配到分片项0,1,2,3,4；服务器B被分配到分片项5,6,7,8,9，直接的结果就是服务器A遍历ID以0-4结尾的数据；服务器B遍历ID以5-9结尾的数据。

2. 分片项与业务处理解耦

Elastic-Job并不直接提供数据处理的功能，框架只会将分片项分配至各个运行中的作业服务器，开发者需要自行处理分片项与真实数据的对应关系。

3. 个性化参数的适用场景

个性化参数即 `shardingItemParameter`，可以和分片项匹配对应关系，用于将分片项的数字转换为更加可读的业务代码。

例如：按照地区水平拆分数据库，数据库A是北京的数据；数据库B是上海的数据；数据库C是广州的数据。如果仅按照分片项配置，开发者需要了解0表示北京；1表示上海；2表示广州。合理使用个性化参数可以让代码更可读，如果配置为0=北京,1=上海,2=广州，那么代码中直接使用北京，上海，广州的枚举值即可完成分片项和业务逻辑的对应关系。

5.3 核心理念

5.3.1 分布式调度

Elastic-Job-Lite并无作业调度中心节点，而是基于部署作业框架的程序在到达相应时间点时各自触发调度。

注册中心仅用于作业注册和监控信息存储。而主作业节点仅用于处理分片和清理等功能。

5.3.2 作业高可用

Elastic-Job-Lite提供最安全的方式执行作业。将分片总数设置为1，并使用多于1台的服务器执行作业，作业将会以1主n从的方式执行。

一旦执行作业的服务器崩溃，等待执行的服务器将会在下次作业启动时替补执行。开启失效转移功能效果更好，可以保证在本次作业执行时崩溃，备机立即启动替补执行。

5.3.3 最大限度利用资源

Elastic-Job-Lite也提供最灵活的方式，最大限度的提高执行作业的吞吐量。将分片项设置为大于服务器的数量，最好是大于服务器倍数的数量，作业将会合理的利用分布式资源，动态的分配分片项。

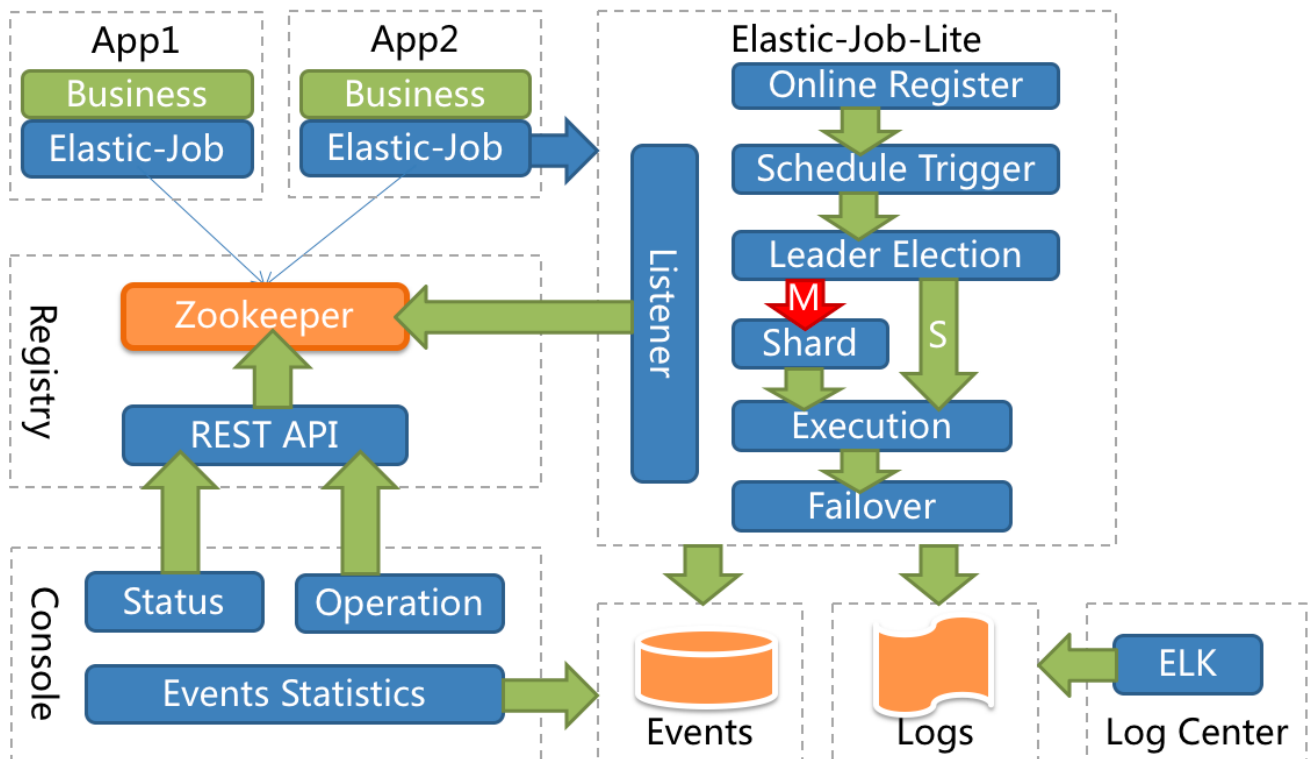
例如：3台服务器，分成10片，则分片项分配结果为服务器A=0,1,2;服务器B=3,4,5;服务器C=6,7,8,9。如果服务器C崩溃，则分片项分配结果为服务器A=0,1,2,3,4;服务器B=5,6,7,8,9。在不丢失分片项的情况下，最大限度的利用现有资源提高吞吐量。

5.3.4 整体架构图

弹性分布式实现

- 第一台服务器上线触发主服务器选举。主服务器一旦下线，则重新触发选举，选举过程中阻塞，只有主服务器选举完成，才会执行其他任务。

- 某作业服务器上线时会自动将服务器信息注册到注册中心，下线时会自动更新服务器状态。
- 主节点选举，服务器上下线，分片总数变更均更新重新分片标记。
- 定时任务触发时，如需重新分片，则通过主服务器分片，分片过程中阻塞，分片结束后才可执行任务。如分片过程中主服务器下线，则先选举主服务器，再分片。
- 通过上一项说明可知，为了维持作业运行时的稳定性，运行过程中只会标记分片状态，不会重新分片。分片仅可能发生在下次任务触发前。
- 每次分片都会按服务器 IP 排序，保证分片结果不会产生较大波动。
- 实现失效转移功能，在某台服务器执行完毕后主动抓取未分配的分片，并且在某台服务器下线后主动寻找可用的服务器执行任务。



5.4 快速入门

5.4.1 引入Maven依赖

```
<!-- 引入elastic-job-lite核心模块 -->
<dependency>
  <groupId>com.dangdang</groupId>
  <artifactId>elastic-job-lite-core</artifactId>
  <version>${latest.release.version}</version>
</dependency>

<!-- 使用springframework自定义命名空间时引入 -->
<dependency>
  <groupId>com.dangdang</groupId>
  <artifactId>elastic-job-lite-spring</artifactId>
  <version>${latest.release.version}</version>
</dependency>
```

5.4.2 作业开发

Elastic-Job-Lite和Elastic-Job-Cloud提供统一作业接口，开发者仅需对业务作业进行一次开发，之后可根据不同的配置以及部署至不同的Lite或Cloud环境。

Elastic-Job提供Simple、Dataflow和Script 3种作业类型。方法参数shardingContext包含作业配置、片和运行时信息。可通过getShardingTotalCount(), getShardingItem()等方法分别获取分片总数，运行在本作业服务器的分片序列号等。

5.4.2.1 Simple类型作业

意为简单实现，未经任何封装的类型。需实现SimpleJob接口。该接口仅提供单一方法用于覆盖，此方法将定时执行。与Quartz原生接口相似，但提供了弹性扩缩容和分片等功能。

```
public class MyElasticJob implements SimpleJob {

    @Override
    public void execute(ShardingContext context) {
        switch (context.getShardingItem()) {
            case 0:
                // do something by sharding item 0
                break;
            case 1:
                // do something by sharding item 1
                break;
            case 2:
                // do something by sharding item 2
                break;
            // case n: ...
        }
    }
}
```

5.4.2.2 Dataflow类型作业

Dataflow类型用于处理数据流，需实现DataflowJob接口。该接口提供2个方法可供覆盖，分别用于抓取(fetchData)和处理(processData)数据。

```
public class MyElasticJob implements DataflowJob<Foo> {

    @Override
    public List<Foo> fetchData(ShardingContext context) {
        switch (context.getShardingItem()) {
            case 0:
                List<Foo> data = // get data from database by sharding item 0
                return data;
            case 1:
                List<Foo> data = // get data from database by sharding item 1
                return data;
            case 2:
                List<Foo> data = // get data from database by sharding item 2
                return data;
            // case n: ...
        }
    }
}
```

```
    }  
}  
@Override  
public void processData(ShardingContext shardingContext, List<Foo> data) {  
    // process data  
    // ...  
}  
}
```

流式处理

可通过DataflowJobConfiguration配置是否流式处理。

流式处理数据只有fetchData方法的返回值为null或集合长度为空时，作业才停止抓取，否则作业将一直运行下去；非流式处理数据则只会在每次作业执行过程中执行一次fetchData方法和processData方法，随即完成本次作业。

如果采用流式作业处理方式，建议processData处理数据后更新其状态，避免fetchData再次抓取到，从而使得作业永不停止。流式数据处理参照TbSchedule设计，适用于不间断的数据处理。

5.4.3 作业配置

```
<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
    xmlns:reg="http://www.dangdang.com/schema/ddframe/reg"  
    xmlns:job="http://www.dangdang.com/schema/ddframe/job"  
    xsi:schemaLocation="http://www.springframework.org/schema/beans  
        http://www.springframework.org/schema/beans/spring-beans.xsd  
        http://www.dangdang.com/schema/ddframe/reg  
        http://www.dangdang.com/schema/ddframe/reg/reg.xsd  
        http://www.dangdang.com/schema/ddframe/job  
        http://www.dangdang.com/schema/ddframe/job/job.xsd"  
    >  
    <!--配置作业注册中心 -->  
    <reg:zookeeper id="regCenter" server-lists="yourhost:2181" namespace="dd-job" base-  
sleep-time-milliseconds="1000" max-sleep-time-milliseconds="3000" max-retries="3" />  
  
    <!-- 配置作业-->  
    <job:simple id="oneOffElasticJob" class="xxx.MyElasticJob" registry-center-  
ref="regCenter" cron="0/10 * * * * ?" sharding-total-count="3" sharding-item-  
parameters="0=A,1=B,2=C" />  
</beans>
```

5.4.4 注册中心配置

用于注册和协调作业分布式行为的组件，目前仅支持Zookeeper。

ZookeeperConfiguration属性详细说明

属性名	类型	构造器注入	缺省值	描述
serverLists	String	是		连接Zookeeper服务器的列表 包括IP地址和端口号 多个地址用逗号分隔 如: host1:2181,host2:2181
namespace	String	是		Zookeeper的命名空间
baseSleepTimeMilliseconds	int	否	1000	等待重试的间隔时间的初始值 单位: 毫秒
maxSleepTimeMilliseconds	String	否	3000	等待重试的间隔时间的最大值 单位: 毫秒
maxRetries	String	否	3	最大重试次数
sessionTimeoutMilliseconds	boolean	否	60000	会话超时时间 单位: 毫秒
connectionTimeoutMilliseconds	boolean	否	15000	连接超时时间 单位: 毫秒
digest	String	否		连接Zookeeper的权限令牌 缺省为不需要权限验证

5.4.5 作业配置

作业配置分为3级，分别是JobCoreConfiguration，JobTypeConfiguration和LiteJobConfiguration。LiteJobConfiguration使用JobTypeConfiguration，JobTypeConfiguration使用JobCoreConfiguration，层层嵌套。JobTypeConfiguration根据不同实现类型分为SimpleJobConfiguration，DataflowJobConfiguration和ScriptJobConfiguration。

JobCoreConfiguration属性详细说明

属性名	类型	构造器注入	缺省值	描述
jobName	String	是		作业名称
cron	String	是		cron表达式，用于控制作业触发时间
shardingTotalCount	int	是		作业分片总数
shardingItemParameters	String	否		分片序列号和参数用等号分隔，多个键值对用逗号分隔 分片序列号从0开始，不可大于或等于作业分片总数 如： 0=a,1=b,2=c
jobParameter	String	否		作业自定义参数 作业自定义参数，可通过传递该参数为作业调度的业务方法传参，用于实现带参数的作业例：每次获取的数据量、作业实例从数据库读取的主键等
failover	boolean	否	false	是否开启任务执行失效转移，开启表示如果作业在一次任务执行中途宕机，允许将该次未完成任务在另一作业节点上补偿执行
misfire	boolean	否	true	是否开启错过任务重新执行
description	String	否		作业描述信息
jobProperties	Enum	否		配置jobProperties定义的枚举控制Elastic-Job的实现细节 JOB_EXCEPTION_HANDLER用于扩展异常处理类 EXECUTOR_SERVICE_HANDLER用于扩展作业处理线程池类

5.5 SpringBoot2.x整合Elastic-Job

教学代码见 `elastic-job-example`。

5.5.1 pom.xml引入依赖

pom.xml示例：elastic-job-example下的pom.xml

```
<dependency>
  <groupId>com.dangdang</groupId>
  <artifactId>elastic-job-lite-core</artifactId>
  <version>2.1.5</version>
</dependency>
<dependency>
  <groupId>com.dangdang</groupId>
  <artifactId>elastic-job-lite-spring</artifactId>
  <version>2.1.5</version>
</dependency>
```

5.5.2 配置文件application.yml

配置文件示例：elastic-job-example下的application.yml

```
#zookeeper注册中心配置
regCenter:
  #zookeeper注册中心IP与端口列表
  serverList: 192.168.157.145:2181
  #Zookeeper的命名空间
  namespace: elastic-job-lite-springboot

simpleJob:
  #cron表达式，用于控制作业触发时间
  cron: 0/10 * * * * ?
  #作业分片总数
  shardingTotalCount: 3
  #分片序号和参数用等号分隔，多个键值对用逗号分隔，分片序号从0开始，不可大于或等于作业分片总数，如：
  #0=a,1=b,2=c
  shardingItemParameters: 0=Beijing,1=Shanghai,2=Guangzhou

dataflowJob:
  #cron表达式，用于控制作业触发时间
  cron: 0/5 * * * * ?
  #作业分片总数
  shardingTotalCount: 3
  #分片序号和参数用等号分隔，多个键值对用逗号分隔，分片序号从0开始，不可大于或等于作业分片总数，如：
  #0=a,1=b,2=c
  shardingItemParameters: 0=Beijing,1=Shanghai,2=Guangzhou
```

5.5.3 连接注册中心

启动时调用初始化方法，连接注册中心ZK。

JAVA代码示例：com.qianfeng.elasticjob.config.RegistryCenterConfig

```
package com.qianfeng.elasticjob.config;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.autoconfigure.condition.ConditionalOnExpression;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import com.dangdang.ddframe.job.reg.zookeeper.ZookeeperConfiguration;
import com.dangdang.ddframe.job.reg.zookeeper.ZookeeperRegistryCenter;

@Configuration
@ConditionalOnExpression("'${regCenter.serverList}'.length() > 0")
public class RegistryCenterConfig {

    @Bean(initMethod = "init")
    public ZookeeperRegistryCenter regCenter(@Value("${regCenter.serverList}") final String
serverList, @Value("${regCenter.namespace}") final String namespace) {
        return new ZookeeperRegistryCenter(new ZookeeperConfiguration(serverList,
```

```
namespace));  
    }  
}
```

5.5.4 开发Simple类型作业

5.5.4.1 任务配置

JAVA代码示例: com.qianfeng.elasticjob.config.SimpleJobConfig

```
@Configuration  
public class SimpleJobConfig {  
  
    @Resource  
    private ZookeeperRegistryCenter regCenter;  
  
    /**  
     * 自己实现的job  
     */  
    @Bean  
    public SimpleJob simpleJob() {  
        return new SpringSimpleJob();  
    }  
  
    /**  
     * 将自己实现的job加入调度中执行  
     * @param simpleJob  
     * @param cron  
     * @param shardingTotalCount  
     * @param shardingItemParameters  
     * @return  
     */  
    @Bean(initMethod = "init")  
    public Jobscheduler simpleJobscheduler(final SimpleJob simpleJob,  
@Value("${simpleJob.cron}") final String cron,  
@Value("${simpleJob.shardingTotalCount}") final  
int shardingTotalCount,  
@Value("${simpleJob.shardingItemParameters}")  
final String shardingItemParameters) {  
        return new SpringJobscheduler(simpleJob, regCenter,  
getLiteJobConfiguration(simpleJob.getClass(), cron,  
shardingTotalCount, shardingItemParameters));  
    }  
  
    /**  
     * 作业的配置  
     * @param jobClass  
     * @param cron  
     * @param shardingTotalCount  
     * @param shardingItemParameters  
     * @return  
     */  
}
```

```
private LiteJobConfiguration getLiteJobConfiguration(final Class<? extends SimpleJob>
jobClass, final String cron,
final int shardingTotalCount, final
String shardingItemParameters) {
    return LiteJobConfiguration.newBuilder(new
SimpleJobConfiguration(JobCoreConfiguration.newBuilder(
    jobClass.getName(), cron,
shardingTotalCount).shardingItemParameters(shardingItemParameters).build(),
    jobClass.getCanonicalName()).overwrite(true).build());
}
}
```

5.5.4.2 作业实现

具体实现JOB的过程。

JAVA代码示例: com.qianfeng.elasticjob.config.SpringSimpleJob

```
public void execute(final ShardingContext shardingContext) {
    System.out.println(String.format("Item: %s | ShardingParameter: %s | Time: %s |
Thread: %s | %s",
        shardingContext.getShardingItem(), shardingContext.getShardingParameter() ,
        new SimpleDateFormat("HH:mm:ss").format(new Date()),
        Thread.currentThread().getId(), "SIMPLE"));
    //查询某个分片的信息
    List<Foo> data = fooRepository.findTodoData(shardingContext.getShardingParameter(),
10);
    //批理更新, 把Foo.Status.TODO更新为Foo.Status.COMPLETED
    for (Foo each : data) {
        fooRepository.setCompleted(each.getId());
    }
    System.out.println("Thread:" + Thread.currentThread().getId() + " , 更新了数据条数: " +
data.size());
    //查看更新后的结果
    List<Foo> newData =
fooRepository.findTodoData(shardingContext.getShardingParameter(), 10);
    //查看更新后的日志
    //      for (Foo each : data) {
    //          System.out.println("Thread:" + Thread.currentThread().getId() +"
"+each.toString());
    //      }
}
```

5.5.4.3 模拟数据源

```
@Repository
public class FooRepository {

    private Map<Long, Foo> data = new ConcurrentHashMap<>(300, 1);

    public FooRepository() {
        init();
    }
}
```

```
}

private void init() {
    addData(0L, 10L, "Beijing");
    addData(10L, 20L, "Shanghai");
    addData(20L, 30L, "Guangzhou");
}

private void addData(final long idFrom, final long idTo, final String location) {
    for (long i = idFrom; i < idTo; i++) {
        data.put(i, new Foo(i, location, Foo.Status.TODO));
    }
}

public List<Foo> findTodoData(final String location, final int limit) {
    List<Foo> result = new ArrayList<>(limit);
    int count = 0;
    for (Map.Entry<Long, Foo> each : data.entrySet()) {
        Foo foo = each.getValue();
        if (foo.getLocation().equals(location) && foo.getStatus() == Foo.Status.TODO) {
            result.add(foo);
            count++;
            if (count == limit) {
                break;
            }
        }
    }
    return result;
}

public void setCompleted(final long id) {
    data.get(id).setStatus(Foo.Status.COMPLETED);
}
}
```

5.5.4.4 运行结果

运行com.qianfeng.elasticjob.SpringBootMain

每隔10秒运行一次，初次执行结果示例如下：

```
Item: 0 | Time: 15:25:30 | Thread: 66 | SIMPLE
Item: 1 | Time: 15:25:30 | Thread: 67 | SIMPLE
Item: 2 | Time: 15:25:30 | Thread: 68 | SIMPLE
Thread:66 id: 0, location: Beijing, status: COMPLETED
Thread:66 id: 1, location: Beijing, status: COMPLETED
Thread:68 id: 20, location: Guangzhou, status: COMPLETED
Thread:67 id: 10, location: Shanghai, status: COMPLETED
Thread:68 id: 21, location: Guangzhou, status: COMPLETED
Thread:66 id: 2, location: Beijing, status: COMPLETED
Thread:68 id: 22, location: Guangzhou, status: COMPLETED
Thread:67 id: 11, location: Shanghai, status: COMPLETED
Thread:68 id: 23, location: Guangzhou, status: COMPLETED
```

```
Thread:66 id: 3, location: Beijing, status: COMPLETED
Thread:68 id: 24, location: Guangzhou, status: COMPLETED
Thread:67 id: 12, location: Shanghai, status: COMPLETED
Thread:68 id: 25, location: Guangzhou, status: COMPLETED
Thread:66 id: 4, location: Beijing, status: COMPLETED
Thread:68 id: 26, location: Guangzhou, status: COMPLETED
Thread:67 id: 13, location: Shanghai, status: COMPLETED
Thread:68 id: 27, location: Guangzhou, status: COMPLETED
Thread:66 id: 5, location: Beijing, status: COMPLETED
Thread:66 id: 6, location: Beijing, status: COMPLETED
Thread:68 id: 28, location: Guangzhou, status: COMPLETED
Thread:67 id: 14, location: Shanghai, status: COMPLETED
Thread:68 id: 29, location: Guangzhou, status: COMPLETED
Thread:66 id: 7, location: Beijing, status: COMPLETED
Thread:67 id: 15, location: Shanghai, status: COMPLETED
Thread:66 id: 8, location: Beijing, status: COMPLETED
Thread:66 id: 9, location: Beijing, status: COMPLETED
Thread:67 id: 16, location: Shanghai, status: COMPLETED
Thread:67 id: 17, location: Shanghai, status: COMPLETED
Thread:67 id: 18, location: Shanghai, status: COMPLETED
Thread:67 id: 19, location: Shanghai, status: COMPLETED
```

通过上面可以看出分三个实例来运行的程序。

5.5.5 开发dataFlow类型作业

流式处理数据只有fetchData方法的返回值为null或集合长度为空时，作业才停止抓取，否则作业将一直运行下去；非流式处理数据则只会在每次作业执行过程中执行一次fetchData方法和processData方法，随即完成本次作业。

如果采用流式作业处理方式，建议processData处理数据后更新其状态，避免fetchData再次抓取到，从而使得作业永不停止。流式数据处理参照TbSchedule设计，适用于不间断的数据处理。

5.5.5.1 任务配置

JAVA代码示例：com.qianfeng.elasticjob.config.DataflowJobConfig

```
@Configuration
public class DataflowJobConfig {

    @Resource
    private ZookeeperRegistryCenter regCenter;

    @Bean
    public DataflowJob dataflowJob() {
        return new SpringDataflowJob();
    }

    @Bean(initMethod = "init")
    public Jobscheduler dataflowJobscheduler(final DataflowJob dataflowJob,
@Value("${dataflowJob.cron}") final String cron, @Value("${dataflowJob.shardingTotalCount}")
final int shardingTotalCount,
                                                @Value("${dataflowJob.shardingItemParameters}")
final String shardingItemParameters) {
```

```
        return new SpringJobsScheduler(dataflowJob, regCenter,
getLiteJobConfiguration(dataflowJob.getClass(), cron, shardingTotalCount,
shardingItemParameters));
    }

    private LiteJobConfiguration getLiteJobConfiguration(final Class<? extends DataflowJob>
jobClass, final String cron, final int shardingTotalCount, final String
shardingItemParameters) {
        return LiteJobConfiguration.newBuilder(new
DataflowJobConfiguration(JobCoreConfiguration.newBuilder(
jobClass.getName(), cron,
shardingTotalCount).shardingItemParameters(shardingItemParameters).build(),
jobClass.getCanonicalName(), true)).overwrite(true).build();
    }
}
```

5.5.5.2 作业实现

JAVA代码示例: com.qianfeng.elasticjob.job.dataflow.SpringDataflowJob

```
public class SpringDataflowJob implements DataflowJob<Foo> {

    @Resource
    private FooRepository fooRepository;

    @Override
    public List<Foo> fetchData(final ShardingContext shardingContext) {
        List<Foo> todoData =
fooRepository.findTodoData(shardingContext.getShardingParameter(), 10);
        System.out.println(String.format("Item: %s | Time: %s | Thread: %s | %s|取出数据条数:
%s",
shardingContext.getShardingItem(), new
SimpleDateFormat("HH:mm:ss").format(new Date()),
Thread.currentThread().getId(), "DATAFLOW FETCH",todoData.size()));
        return todoData;
    }

    @Override
    public void processData(final ShardingContext shardingContext, final List<Foo> data) {
        System.out.println(String.format("Item: %s | Time: %s | Thread: %s | %s",
shardingContext.getShardingItem(), new
SimpleDateFormat("HH:mm:ss").format(new Date()),
Thread.currentThread().getId(), "DATAFLOW PROCESS"));
        for (Foo each : data) {
            fooRepository.setCompleted(each.getId());
        }
    }
}
```

5.5.5.3 运行结果

```
Item: 0 | Time: 17:32:00 | Thread: 64 | DATAFLOW FETCH|取出数据条数: 10
Item: 1 | Time: 17:32:00 | Thread: 65 | DATAFLOW FETCH|取出数据条数: 10
Item: 2 | Time: 17:32:00 | Thread: 66 | DATAFLOW FETCH|取出数据条数: 10
Item: 0 | Time: 17:32:00 | Thread: 64 | DATAFLOW PROCESS
Item: 1 | Time: 17:32:00 | Thread: 65 | DATAFLOW PROCESS
Item: 2 | Time: 17:32:00 | Thread: 66 | DATAFLOW PROCESS
Item: 0 | Time: 17:32:00 | Thread: 64 | DATAFLOW FETCH|取出数据条数: 0
Item: 1 | Time: 17:32:00 | Thread: 65 | DATAFLOW FETCH|取出数据条数: 0
Item: 2 | Time: 17:32:00 | Thread: 66 | DATAFLOW FETCH|取出数据条数: 0
```

第二次返回fetchData方法返回null停止一个任务的运行。

```
List<Foo> todoData = fooRepository.findTodoData(shardingContext.getShardingParameter(), 10);
```

修改为每次取5条:

```
List<Foo> todoData = fooRepository.findTodoData(shardingContext.getShardingParameter(), 5);
```

运行日志如下所示:

```
Item: 0 | Time: 17:36:00 | Thread: 66 | DATAFLOW FETCH|取出数据条数: 5
Item: 2 | Time: 17:36:00 | Thread: 68 | DATAFLOW FETCH|取出数据条数: 5
Item: 1 | Time: 17:36:00 | Thread: 67 | DATAFLOW FETCH|取出数据条数: 5
Item: 0 | Time: 17:36:00 | Thread: 66 | DATAFLOW PROCESS
Item: 1 | Time: 17:36:00 | Thread: 67 | DATAFLOW PROCESS
Item: 2 | Time: 17:36:00 | Thread: 68 | DATAFLOW PROCESS
Item: 1 | Time: 17:36:00 | Thread: 67 | DATAFLOW FETCH|取出数据条数: 5
Item: 0 | Time: 17:36:00 | Thread: 66 | DATAFLOW FETCH|取出数据条数: 5
Item: 1 | Time: 17:36:00 | Thread: 67 | DATAFLOW PROCESS
Item: 2 | Time: 17:36:00 | Thread: 68 | DATAFLOW FETCH|取出数据条数: 5
Item: 0 | Time: 17:36:00 | Thread: 66 | DATAFLOW PROCESS
Item: 2 | Time: 17:36:00 | Thread: 68 | DATAFLOW PROCESS
Item: 1 | Time: 17:36:00 | Thread: 67 | DATAFLOW FETCH|取出数据条数: 0
Item: 0 | Time: 17:36:00 | Thread: 66 | DATAFLOW FETCH|取出数据条数: 0
Item: 2 | Time: 17:36:00 | Thread: 68 | DATAFLOW FETCH|取出数据条数: 0
Item: 0 | Time: 17:36:30 | Thread: 69 | DATAFLOW FETCH|取出数据条数: 0
Item: 1 | Time: 17:36:30 | Thread: 70 | DATAFLOW FETCH|取出数据条数: 0
Item: 2 | Time: 17:36:30 | Thread: 71 | DATAFLOW FETCH|取出数据条数: 0
```

由上面日志可以看出来, 每次取5条, 取了二次, 第三次返回null就停止执行。

第六章 学生练习题以及课后作业

假设有三个订单数据库(我们用三张表: `pop_order_data1`, `pop_order_data2`, `pop_order_data3` 来模拟三个数据库), 每天晚上2点会定时把已完成订单移到HBASE, 然后删除数据库中的已完成订单。

现在我们通过elastic-job分布式任务来分成三片来模拟删除已完成的数据。

伪表结构如下所示:


```
CREATE TABLE `pop_order_data` (  
  `id` bigint(20) NOT NULL COMMENT 'id',  
  `order_id` bigint(20) NOT NULL COMMENT '订单id',  
  `vender_id` bigint(20) NOT NULL COMMENT '商家id',  
  `order_state` bigint(20) DEFAULT NULL COMMENT '订单状态 (英文) ; 枚举值: 1) wait_seller_stock_out 等待出库 2) wait_goods_receive_confirm 等待确认收货 3) wait_seller_delivery 等待发货 (只适用于海外购商家, 含义为“等待境内发货”标签下的订单, 非海外购商家无需使用) 4) pop_order_pause pop暂停 5) finished_l 完成 6) trade_canceled 取消 7) locked 已锁定 8) wait_send_code 等待发码 (loc 订单特有状态) ',  
  PRIMARY KEY (`id`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='订单数据表';
```

完成功能如下:

1. 在数据库中建立三张表
2. 模拟每个表生成1000条的已完成订单状态的数据
3. 写分布式任务程序按表分成三片来模拟每天定时某个时间点删除已完成的订单