目标：

第一章：概述

1、理解任务调度的概念

2、理解分布式任务调度的概念

3、能够说出Elastic-Job是什么

第二章：Elastic-Job快速入门

1、能够搭建Elastic-Job快速入门工程环境

2、能够编写Elastic-Job快速入门的程序

3、理解Elastic-Job整体架构的组成部分的职责

4、理解ZooKeeper在Elastic-Job中的作用

第三章：Spring Boot开发分布式任务调度

1、能够采用Spring Boot搭建Elastic-Job程序环境

2、理解作业分片的概念

3、能够实现Elastic-Job作业分片案例

第四章：Elastic-Job高级

1、能够使用事件跟踪

2、能够使用elastic-job-lite-console

3、能够使用Dump命令

Elastic-Job分布式任务调度

1.概述

1.1.什么是任务调度

我们可以先思考一下下面业务场景的解决方案：

- 某电商系统需要在每天上午10点，下午3点，晚上8点发放一批优惠券。
- 某银行系统需要在信用卡到期还款日的前三天进行短信提醒。
- 某财务系统需要在每天凌晨0:10结算前一天的财务数据，统计汇总。
- 12306会根据车次的不同，而设置某几个时间点进行分批放票。
- 某网站为了实现天气实时展示，每隔5分钟就去天气服务器获取最新的实时天气信息。

以上场景就是任务调度所需要解决的问题。

任务调度是指系统为了自动完成特定任务，在约定的特定时刻去执行任务的过程。有了任务调度即可解放更多的人力由系统自动去执行任务。

任务调度如何实现？

多线程方式实现：

学过多线程的同学，可能会想到，我们可以开启一个线程，每sleep一段时间，就去检查是否已到预期执行时间。

以下代码简单实现了任务调度的功能：

```
public static void main(String[] args) {
    //任务执行间隔时间
    final long timeInterval = 1000;
    Runnable runnable = new Runnable() {
        public void run() {
            while (true) {
                //TODO：something
                try {
                    Thread.sleep(timeInterval);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    };
    Thread thread = new Thread(runnable);
    thread.start();
}
```

上面的代码实现了按一定的间隔时间执行任务调度的功能。

Jdk也为我们提供了相关支持，如Timer、ScheduledExecutor，下边我们了解下。

Timer方式实现：

```
public static void main(String[] args){
    Timer timer = new Timer();
    timer.schedule(new TimerTask(){
        @Override
        public void run() {
           //TODO：something
        }
    }, 1000, 2000);  //1秒后开始调度，每2秒执行一次
}
```

Timer 的优点在于简单易用，每个Timer对应一个线程，因此可以同时启动多个Timer并行执行多个任务，同一个Timer中的任务是串行执行。

ScheduledExecutor方式实现：

```
public static void main(String [] agrs){
    ScheduledExecutorService service = Executors.newScheduledThreadPool(10);
    service.scheduleAtFixedRate(
            new Runnable() {
                @Override
                public void run() {
                    //TODO：something
                    System.out.println("todo something");
                }
            }, 1,
            2, TimeUnit.SECONDS);
}
```

Java 5 推出了基于线程池设计的 ScheduledExecutor，其设计思想是，每一个被调度的任务都会由线程池中一个线程去执行，因此任务是并发执行的，相互之间不会受到干扰。

Timer 和 ScheduledExecutor 都仅能提供基于开始时间与重复间隔的任务调度，不能胜任更加复杂的调度需求。比如，设置每月第一天凌晨1点执行任务、复杂调度任务的管理、任务间传递数据等等。

Quartz 是一个功能强大的任务调度框架，它可以满足更多更复杂的调度需求，Quartz 设计的核心类包括 Scheduler, Job 以及 Trigger。其中，Job 负责定义需要执行的任务，Trigger 负责设置调度策略，Scheduler 将二者组装在一起，并触发任务开始执行。Quartz支持简单的按时间间隔调度、还支持按日历调度方式，通过设置CronTrigger表达式（包括：秒、分、时、日、月、周、年）进行任务调度。

第三方Quartz方式实现：

```
public static void main(String [] agrs) throws SchedulerException {
    //创建一个Scheduler
    SchedulerFactory schedulerFactory = new StdSchedulerFactory();
    Scheduler scheduler = schedulerFactory.getScheduler();
    //创建JobDetail
    JobBuilder jobDetailBuilder = JobBuilder.newJob(MyJob.class);
    jobDetailBuilder.withIdentity("jobName","jobGroupName");
    JobDetail jobDetail = jobDetailBuilder.build();
    //创建触发的CronTrigger 支持按日历调度
        CronTrigger trigger = TriggerBuilder.newTrigger()
                .withIdentity("triggerName", "triggerGroupName")
                .startNow()
                .withSchedule(CronScheduleBuilder.cronSchedule("0/2 * * * * ?"))
                .build();
        //创建触发的SimpleTrigger 简单的间隔调度
        /*SimpleTrigger trigger = TriggerBuilder.newTrigger()
                .withIdentity("triggerName","triggerGroupName")
                .startNow()
                .withSchedule(SimpleScheduleBuilder
                        .simpleSchedule()
                        .withIntervalInSeconds(2)
                        .repeatForever())
                .build();*/
    scheduler.scheduleJob(jobDetail,trigger);
    scheduler.start();
}
```

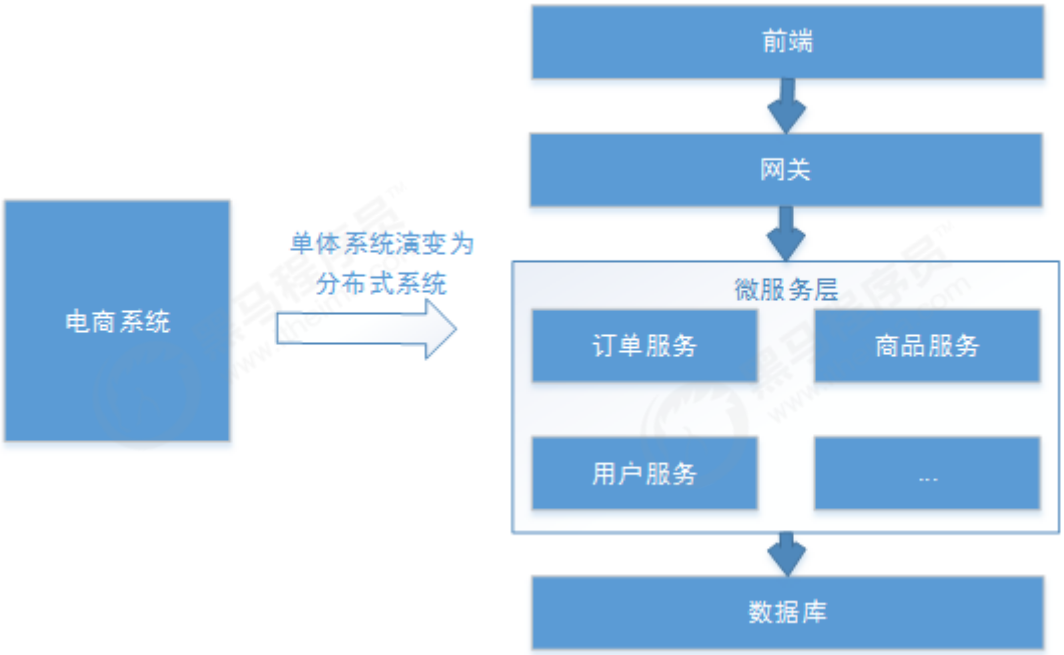
```
public class MyJob implements Job {
    @Override
    public void execute(JobExecutionContext jobExecutionContext){
        System.out.println("todo something");
    }
}
```

通过以上内容我们学习了什么是任务调度，任务调度所解决的问题，以及任务调度的多种实现方式。

1.2.什么是分布式任务调度

什么是分布式？

当前软件的架构正在逐步转变为分布式架构，将单体结构分为若干服务，服务之间通过网络交互来完成用户的业务处理，如下图，电商系统为分布式架构，由订单服务、商品服务、用户服务等组成：



分布式系统具体如下基本特点：

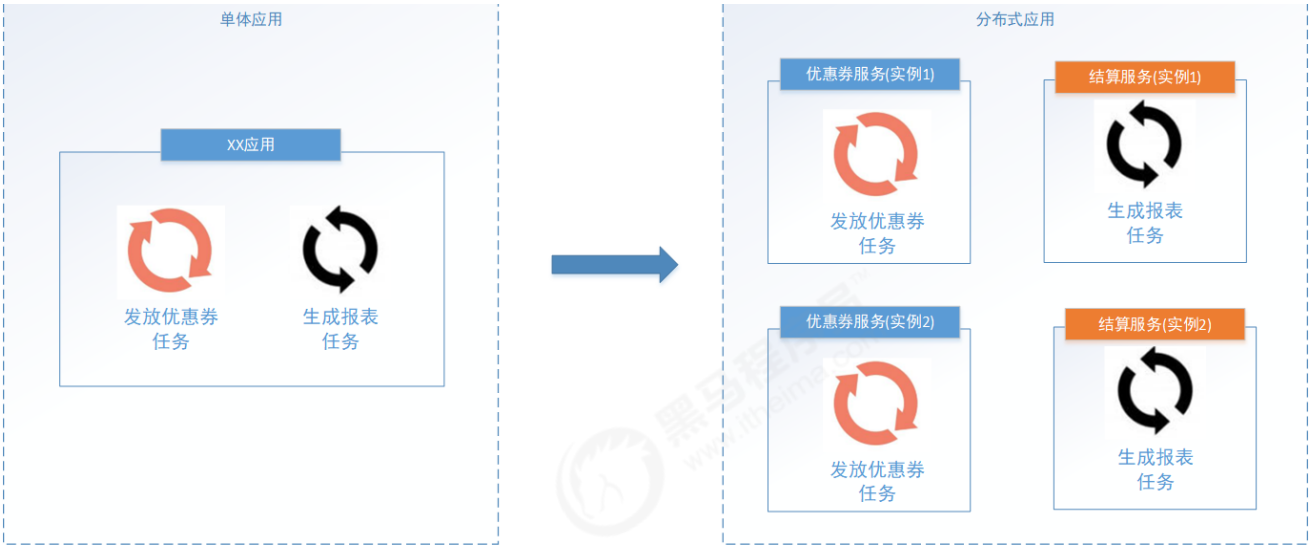
1、分布性：每个部分都可以独立部署，服务之间交互通过网络进行通信，比如：订单服务、商品服务。

2、伸缩性：每个部分都可以集群方式部署，并可针对部分结点进行硬件及软件扩容，具有一定的伸缩能力。

3、高可用：每个部分都可以集群部分，保证高可用。

什么是分布式调度？

通常任务调度的程序是集成在应用中的，比如：优惠卷服务中包括了定时发放优惠卷的的调度程序，结算服务中包括了定期生成报表的任务调度程序，由于采用分布式架构，一个服务往往会部署多个冗余实例来运行我们的业务，在这种分布式系统环境下运行任务调度，我们称之为**分布式任务调度**，如下图：



分布式调度要实现的目标：

不管是任务调度程序集成在应用程序中，还是单独构建的任务调度系统，如果采用分布式调度任务的方式就相当于将任务调度程序分布式构建，这样就可以具有分布式系统的特点，并且提高任务的调度处理能力：

1、并行任务调度

并行任务调度实现靠多线程，如果有大量任务需要调度，此时光靠多线程就会有瓶颈了，因为一台计算机CPU的处理能力是有限的。

如果将任务调度程序分布式部署，每个结点还可以部署为集群，这样就可以让多台计算机共同去完成任务调度，我们可以将任务分割为若干个分片，由不同的实例并行执行，来提高任务调度的处理效率。

2、高可用

若某一个实例宕机，不影响其他实例来执行任务。

3、弹性扩容

当集群中增加实例就可以提高并执行任务的处理效率。

4、任务管理与监测

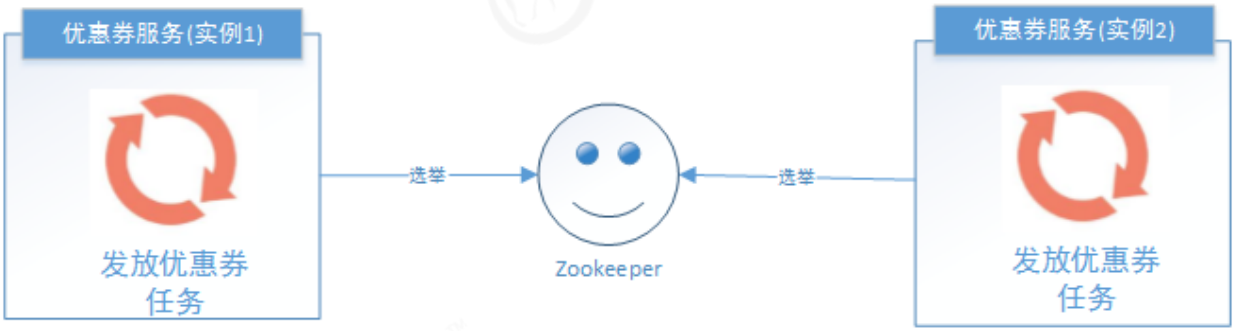
对系统中存在的所有定时任务进行统一的管理及监测。让开发人员及运维人员能够时刻了解任务执行情况，从而做出快速的应急处理响应。

5、避免任务重复执行

当任务调度以集群方式部署，同一个任务调度可能会执行多次，比如在上面提到的电商系统中到点发优惠券的例子，就会发放多次优惠券，对公司造成很多损失，所以我们需要控制相同的任务在多个运行实例上只执行一次，考虑采用下边的方法：

- 分布式锁，多个实例在任务执行前首先需要获取锁，如果获取失败那么久证明有其他服务已经再运行，如果获取成功那么证明没有服务在运行定时任务，那么就可以执行。

- ZooKeeper选举，利用ZooKeeper对Leader实例执行定时任务，有其他业务已经使用了ZK，那么执行定时任务的时候判断自己是否是Leader，如果不是则不执行，如果是则执行业务逻辑，这样也能达到我们的目的。



1.3 Elastic-Job介绍

针对分布式任务调度的需求市场上出现了很多的产品：

1）Elastic-job：当当网基于quartz 二次开发的弹性分布式任务调度系统，功能丰富强大，采用zookeeper实现分布式协调，实现任务高可用以及分片。

2）Saturn： 唯品会开源的一个分布式任务调度平台，可以全域统一配置，统一监控，任务高可用以及分片并发处理。它是在elastic-job基础之上改良出来的。

3）xxl-job：大众点评的分布式任务调度平台，是一个轻量级分布式任务调度平台,其核心设计目标是开发迅速、学习简单、轻量级、易扩展。现已开放源代码并接入多家公司线上产品线,开箱即用。

4）TBSchedule：淘宝的一款非常优秀的高性能分布式调度框架，目前被应用于阿里、京东、支付宝、国美等很多互联网企业的流程调度系统中。

Elastic-Job是一个分布式调度的解决方案，由当当网开源，它由两个相互独立的子项目Elastic-Job-Lite和Elastic-Job-Cloud组成，使用Elastic-Job可以快速实现分布式任务调度。

Elastic-Job的github地址：https://github.com/elasticjob

功能列表：

- **分布式调度协调**

  在分布式环境中，任务能够按指定的调度策略执行，并且能够避免同一任务多实例重复执行。

- **丰富的调度策略：**

  基于成熟的定时任务作业框架Quartz cron表达式执行定时任务。

- **弹性扩容缩容**

当集群中增加某一个实例，它应当也能够被选举并执行任务；当集群减少一个实例时，它所执行的任务能被转移到别的实例来执行。

- **失效转移**

  某实例在任务执行失败后，会被转移到其他实例执行。

- **错过执行作业重触发**

  若因某种原因导致作业错过执行，自动记录错过执行的作业，并在上次作业完成后自动触发。

- **支持并行调度**

  支持任务分片，任务分片是指将一个任务分为多个小任务项在多个实例同时执行。

- **作业分片一致性**

  当任务被分片后，保证同一分片在分布式环境中仅一个执行实例。

- **支持作业生命周期操作**

  可以动态对任务进行开启及停止操作。

- **丰富的作业类型**

  支持Simple、DataFlow、Script三种作业类型，后续会有详细介绍。

- **Spring整合以及命名空间支持**

  对Spring支持良好的整合方式，支持spring自定义命名空间，支持占位符。

- **运维平台**

  提供运维界面，可以管理作业和注册中心。

2 Elastic-Job快速入门

2.1 环境搭建

2.1.1.版本要求

- JDK要求1.7及以上版本
- Maven要求3.0.4及以上版本
- zookeeper要求采用3.4.6及以上版本

2.1.2.Zookeeper安装&运行

https://archive.apache.org/dist/zookeeper/ 下载某版本Zookeeper，并解压。

执行解压目录下的bin/zkServer.cmd。

关于Zookeeper后续章节会有介绍。

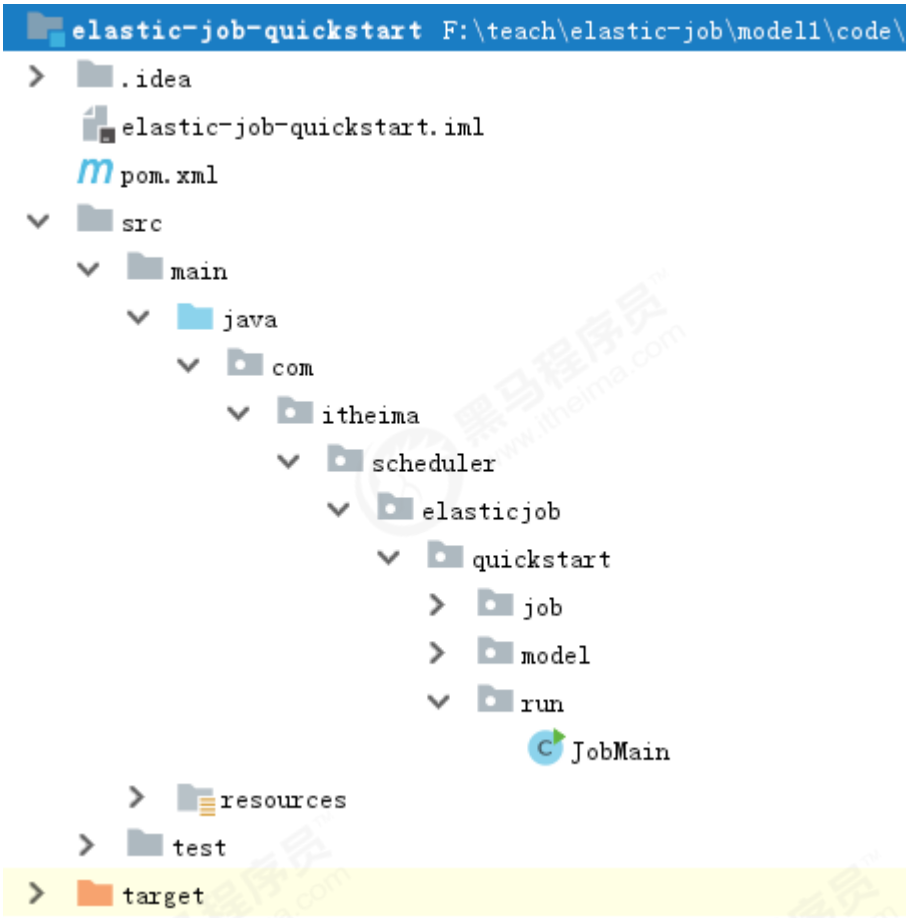
2.1.3.创建maven工程

创建maven工程elastic-job-quickstart，并导入以下依赖：

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.itheima.scheduler</groupId>
    <artifactId>elastic-job-quickstart</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>com.dangdang</groupId>
            <artifactId>elastic-job-lite-core</artifactId>
            <version>2.1.5</version>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.18.0</version>
        </dependency>
    </dependencies>


    <build>
        <finalName>${project.name}</finalName>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

工程结构如下：



2.2 代码实现

2.2.1.编写定时任务类

此任务在**每次执行时获取一定数目的文件，进行备份处理**，由File实体类的backedUp属性来标识该文件是否已备份。

```
public class FileBackupJob implements SimpleJob {

    //每次任务执行要备份文件的数量
    private final int FETCH_SIZE = 1;

    //文件列表（模拟）
    public static List<FileCustom> files = new ArrayList<>();

    /**
     * 任务调度执行方法
     * @param shardingContext
     */
    @Override
    public void execute(ShardingContext shardingContext) {
        //作业分片信息
        int shardingItem = shardingContext.getShardingItem();
        System.out.println(String.format("作业分片：%d",shardingItem));

        //获取未备份的文件
```

```
        List<FileCustom> fileCustoms = fetchUnBackupFiles(FETCH_SIZE);
        //文件备份
        backupFiles(fileCustoms);
    }

    /**
     * 获取未备份的文件
     * @param count
     * @return
     */
    public List<FileCustom> fetchUnBackupFiles(int count){
        List<FileCustom> fetchList = new ArrayList<>();
        int num = 0;
        for(FileCustom fileCustom:files){
            if(num>=count){
                break;
            }
            //未备份的文件则放入列表
            if(!fileCustom.getBackedUp()){
                fetchList.add(fileCustom);
                num++;
            }

        }
        //ManagementFactory.getRuntimeMXBean()获取当前JVM进程的PID
        System.out.println(String.format("%sTime:%s,已获取%d文件",
ManagementFactory.getRuntimeMXBean().getName(),new SimpleDateFormat("hh:mm:ss").format(new
Date()),num));
        return fetchList;
    }

    /**
     * 备份文件
     * @param files
     */
    public void backupFiles(List<FileCustom> files){
        for(FileCustom file : files){
            //标记文件数据为已备份
            file.setBackedUp(Boolean.TRUE);
            System.out.println(String.format("已备份文件:%s  文件类
型:%s",file.getName(),file.getType()));
        }
    }


}
```

文件实体类如下：

```
@Data
public class FileCustom {
    /**
     * 标识
```

```
     */
    private String id;

    /**
     * 文件名
     */
    private String name;

    /**
     * 文件类型，如text、image、radio、vedio
     */
    private String type;

    /**
     * 文件内容
     */
    private String content;

    /**
     * 是否已备份
     */
    private Boolean backedUp = false;

    public FileCustom(String id,String name,String type,String content){
        this.id = id;
        this.name = name;
        this.type = type;
        this.content = content;
    }
}
```

2.2.2.编写启动类

```
public class JobMain {
    //zookeeper端口
    private static final int ZOOKEEPER_PORT = 2181;
    //zookeeper链接字符串 localhost:2181
    private static final String ZOOKEEPER_CONNECTION_STRING = "localhost:" + ZOOKEEPER_PORT;
    //定时任务命名空间
    private static final String JOB_NAMESPACE = "elastic-job-example-java";

    //启动任务
    public static void main(String[] args) {
        //生成测试文件
        generateTestFiles();
        //配置zookeeper
        CoordinatorRegistryCenter registryCenter = setUpRegistryCenter();
        //启动任务
        startJob(registryCenter);
    }
```

```
    //注册中心配置
    private static CoordinatorRegistryCenter setUpRegistryCenter(){
        //注册中心配置
        ZookeeperConfiguration zookeeperConfiguration = new
ZookeeperConfiguration(ZOOKEEPER_CONNECTION_STRING,JOB_NAMESPACE);
        //减少zk的超时时间
        zookeeperConfiguration.setSessionTimeoutMilliseconds(100);
        //创建注册中心
        CoordinatorRegistryCenter registryCenter = new
ZookeeperRegistryCenter(zookeeperConfiguration);
        registryCenter.init();
        return registryCenter;

    }

    //配置并启动任务
    private static void startJob(CoordinatorRegistryCenter registryCenter){
        //创建JobCoreConfiguration
        JobCoreConfiguration jobCoreConfiguration = JobCoreConfiguration.newBuilder("files-job",
"0/3 * * * * ?", 1)
                .build();
        //创建SimpleJobConfiguration
        SimpleJobConfiguration simpleJobConfiguration = new
SimpleJobConfiguration(jobCoreConfiguration, FileBackupJob.class.getCanonicalName());
        //启动任务
        new JobScheduler(registryCenter,
LiteJobConfiguration.newBuilder(simpleJobConfiguration).overwrite(true).build()).init();
    }

    //生成测试文件
    private static void generateTestFiles(){
        for(int i=1;i<11;i++){
            FileBackupJob.files.add(new FileCustom(String.valueOf(i+10),"文件"+
(i+10),"text","content"+ (i+10)));
            FileBackupJob.files.add(new FileCustom(String.valueOf(i+20),"文件"+
(i+20),"image","content"+ (i+20)));
            FileBackupJob.files.add(new FileCustom(String.valueOf(i+30),"文件"+
(i+30),"radio","content"+ (i+30)));
            FileBackupJob.files.add(new FileCustom(String.valueOf(i+40),"文件"+
(i+40),"vedio","content"+ (i+40)));
        }
    }
}
```

2.2.3.测试

（1）启动main方法查看控制台

```
作业分片：0
116052@USER-20180531TKTime:08:55:42,已获取1文件
已备份文件:文件11  文件类型:text
作业分片：0
116052@USER-20180531TKTime:08:55:45,已获取1文件
已备份文件:文件21  文件类型:image
作业分片：0
116052@USER-20180531TKTime:08:55:48,已获取1文件
已备份文件:文件31  文件类型:radio
..略
```

定时任务每3秒批量执行一次，符合基础预期。

（2）测试窗口1不关闭，再次运行main方法观察控制台日志（窗口2）

会出现以下两种情况：

- 窗口1继续执行任务，窗口2不执行任务
- 窗口2接替窗口1执行任务，窗口1停止执行任务

可通过反复启停窗口2查看到以上现象。

（3）窗口1、窗口2同时运行的情况下，停止正在执行任务的窗口

未停止的窗口开始执行任务。

分片测试：

当前作业没有被分片，所以多个实例共同执行时只有一个实例在执行，如果我们将作业分片执行，作业将被拆分为多个独立的任务项，然后由分布式的应用实例分别执行某一个或几个分片项。

修改上边的代码，改为作业分3片执行：

```
//创建JobCoreConfiguration
JobCoreConfiguration jobCoreConfiguration = JobCoreConfiguration.newBuilder("files-job", "0/3 *
* * * ?", 3)
        .build();
```

同时启动三个JobMain：

每个JobMain窗口分别执行一片作业。

总结：

通过以上简单的测试，就可以看出Elastic-Job帮我们解决了分布式调度的以下三个问题：

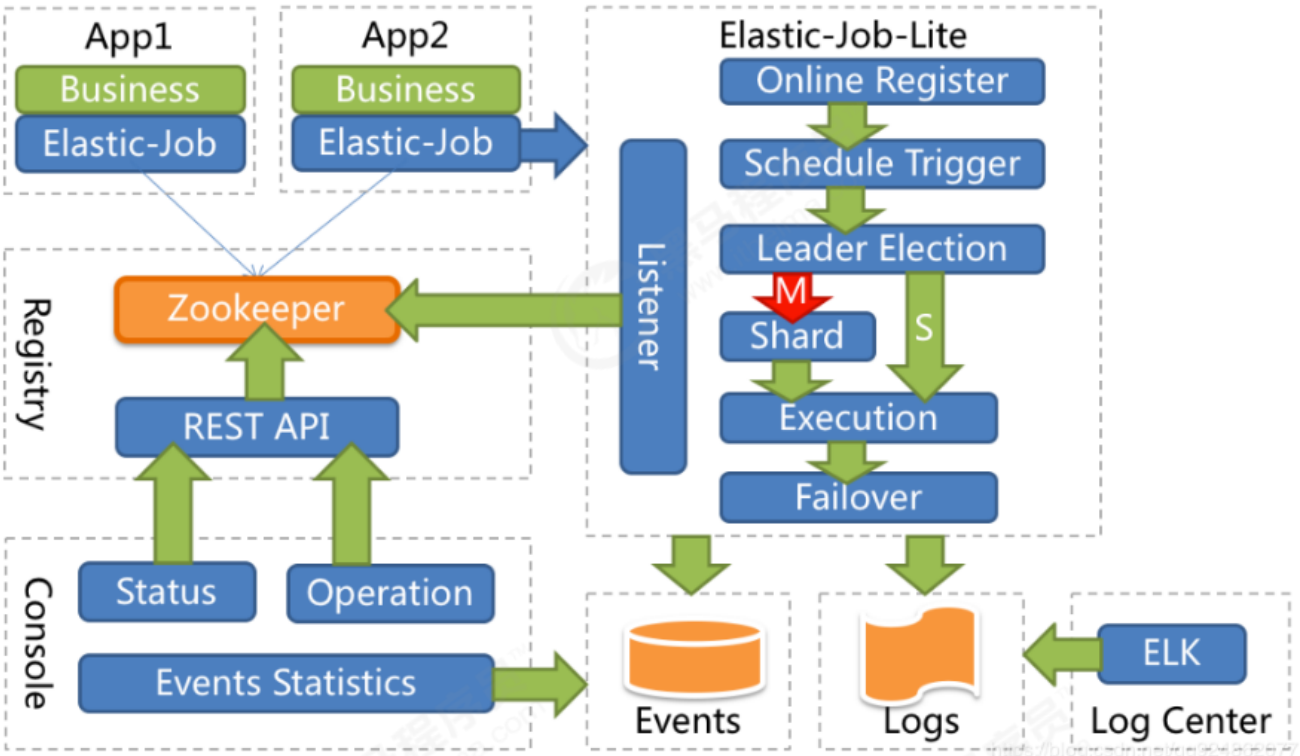
1）多实例部署时避免任务重复执行，在任务执行时间到来时，从所有实例中选举出来一个，让它来执行任务，从而避免多个实例同时执行任务。

2）高可用，若某一个实例宕机，不影响其他实例来执行任务。

3）弹性扩容，当集群中增加某一个实例，它应当也能够被选举并执行任务，如果作业分片将参与执行某个分片作业。

2.3 Elastic-Job工作原理

2.3.1.Elastic-Job整体架构



App：应用程序，内部包含**任务执行业务逻辑**和**Elastic-Job-Lite组件**，其中执行任务需要实现**ElasticJob接口**完成与Elastic-Job-Lite组件的集成，并进行任务的相关配置。应用程序可启动多个实例，也就出现了多个任务执行实例。

Elastic-Job-Lite：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，此组件负责任务的调度，并产生日志及任务调度记录。

无中心化，是指没有调度中心这一概念，每个运行在集群中的作业服务器都是对等的，各个作业节点是自治的、平等的、节点之间通过注册中心进行分布式协调。

Registry：以Zookeeper作为Elastic-Job的注册中心组件，存储了执行任务的相关信息。同时，Elastic-Job利用该组件进行执行任务实例的选举。

Console：Elastic-Job提供了运维平台，它通过读取Zookeeper数据展现任务执行状态，或更新Zookeeper数据修改全局配置。通过Elastic-Job-Lite组件产生的数据来查看任务执行历史记录。

应用程序在启动时，在其内嵌的Elastic-Job-Lite组件会向Zookeeper注册该实例的信息，并触发选举（此时可能已经启动了该应用程序的其他实例），从众多实例中选举出一个Leader，让其执行任务。当到达任务执行时间时，Elastic-Job-Lite组件会调用由应用程序实现的任务业务逻辑，任务执行后会产生任务执行记录。当应用程序的某一个实例宕机时，Zookeeper组件会感知到并重新触发leader选举。

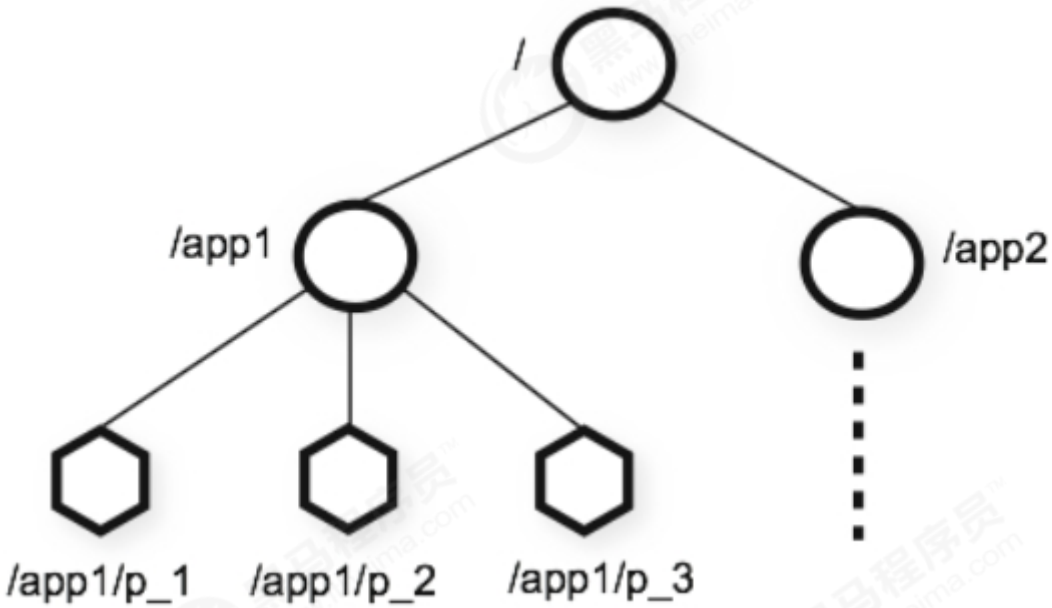
2.3.2.ZooKeeper

在学习Elastic-Job执行原理时，有必要大致了解一下ZooKeeper是用来做什么的，因为：

- Elastic-Job依赖ZooKeeper完成对执行任务**信息的存储**(如任务名称、任务参与实例、任务执行策略等)；
- Elastic-Job依赖ZooKeeper实现选举机制，在任务执行实例数量变化时(如在快速上手中的启动新实例或停止实例)，会触发**选举**机制来决定让哪个实例去执行该任务。

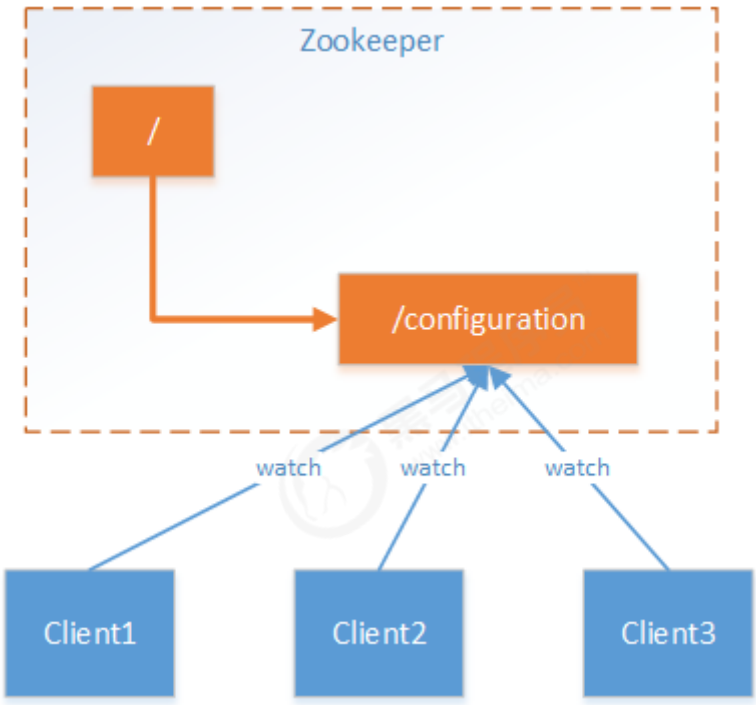
ZooKeeper是一个分布式一致性协调服务，它是Apache Hadoop 的一个子项目，它主要是用来解决分布式应用中经常遇到的一些数据管理问题，如：统一命名服务、状态同步服务、集群管理、分布式应用配置项的管理等。

咱们可以把ZooKeeper想象为一个**特殊的数据库**，它维护着一个类似文件系统的树形数据结构，ZooKeeper的客户端（如Elastic-Job任务执行实例）可以对数据进行存取：



每个子目录项如 /app1都被称作为 znode(目录节点)，和文件系统一样，我们能够自由的增加、删除znode，在一个znode下增加、删除子znode，唯一的不同在于znode是可以存储数据的。

ZooKeeper为什么称之为**一致性协调服务**呢？因为ZooKeeper拥有**数据监听通知机制**，客户端注册监听它关心的znode，当znode发生变化（数据改变、被删除、子目录节点增加删除）时，ZooKeeper会通知所有客户端。简单来说就是，当分布式系统的若干个服务都关心一个数据时，当这个数据发生改变，这些服务都能够得知，那么这些服务就针对此数据达成了一致。



应用场景思考，使用ZooKeeper管理分布式配置项的机制：

假设我们的程序是分布式部署在多台机器上，如果我们要改变程序的配置文件，需要逐台机器去修改，非常麻烦，现在把这些配置全部放到zookeeper上去，保存在 zookeeper 的某个目录节点中，然后所有相关应用程序作为ZooKeeper的客户端对这个目录节点进行监听，一旦配置信息发生变化，每个应用程序就会收到 ZooKeeper的通知，从而获取新的配置信息应用到系统中。

2.3.2.1.Elastic-Job任务信息的保存

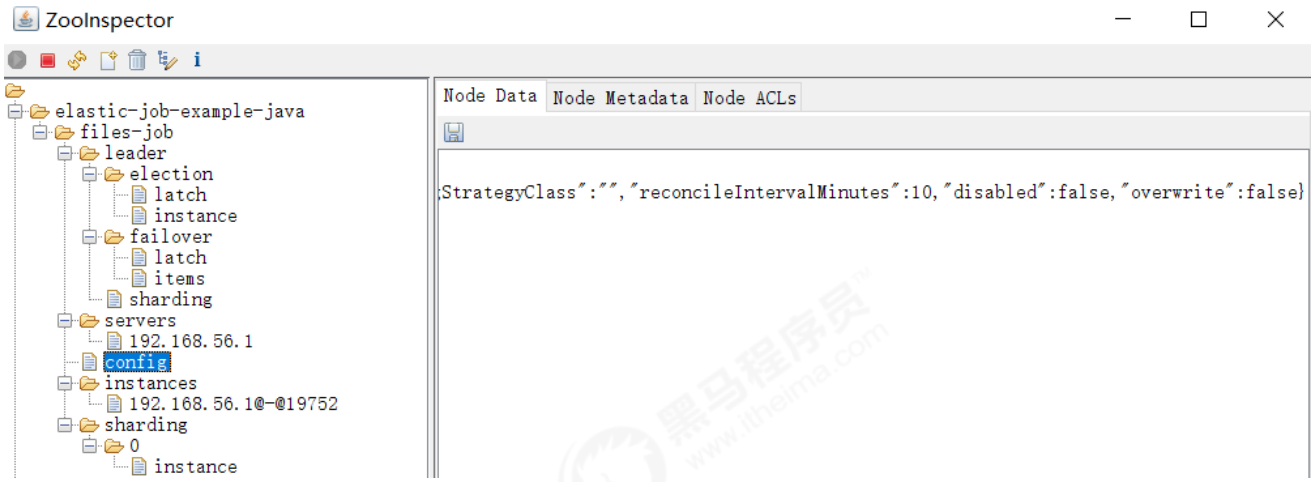
Elastic-Job使用ZooKeeper完成对任务信息的存取，任务执行实例作为ZooKeeper客户端对其znode操作，任务信息保存在znode中。

使用ZooInspector查看zookeeper节点

1、zookeeper图像化客户端工具的下载地址：
https://issues.apache.org/jira/secure/attachment/12436620/ZooInspector.zip；

2、下载完后解压压缩包，双击地址为ZooInspector\build\zookeeper-dev-ZooInspector.jar的jar包；

如果双击没有反应？首先电脑要配好java环境，使用java -jar 再加上你的jar文件的路径 启动即可.



config节点内容如下：

```
{
    "jobName": "files-job",
    "jobClass": "com.itheima.scheduling.job.FileBackupJob",
    "jobType": "SIMPLE",
    "cron": "0/3 * * * * ?",
    "shardingTotalCount": 1,
    "shardingItemParameters": "",
    "jobParameter": "",
    "failover": true,
    "misfire": true,
    "description": "",
    "jobProperties": {
        "job_exception_handler":
"com.dangdang.ddframe.job.executor.handler.impl.DefaultJobExceptionHandler",
        "executor_service_handler":
"com.dangdang.ddframe.job.executor.handler.impl.DefaultExecutorServiceHandler"
    },
    "monitorExecution": true,
    "maxTimeDiffSeconds": -1,
    "monitorPort": -1,
    "jobShardingStrategyClass": "",
    "reconcileIntervalMinutes": 10,
    "disabled": false,
    "overwrite": false
}
```

节点记录了任务的配置信息，包含执行类，cron表达式，分片算法类，分片数量，分片参数。默认状态下，如果你修改了Job的配置比如cron表达式，分片数量等是不会更新到zookeeper上去的，需要把LiteJobConfiguration的参数**overwrite**修改成true，或者删除zk的结点再启动作业重新创建。

instances节点：

同一个Job下的elastic-job的部署实例。一台机器上可以启动多个Job实例，也就是Jar包。instances的命名是[IP+@-@+PID]。

`leader节点：` 任务实例的主节点信息，通过zookeeper的主节点选举，选出来的主节点信息。下面的子节点分为election，sharding和failover三个子节点。分别用于主节点选举，分片和失效转移处理。election下面的instance节点显式了当前主节点的实例ID：jobInstanceId。latch节点也是一个永久节点用于选举时候的实现分布式锁。sharding节点下面有一个临时节点necessary，是否需要重新分片的标记，如果分片总数变化或任务实例节点上下线，以及主节点选举，都会触发设置重分片标记，主节点会进行分片计算。

`sharding节点：` 任务的分片信息，子节点是分片项序号，从零开始，至分片总数减一。从这个节点可以看出哪个分片在哪个实例上运行

2.3.2.2.Elastic-Job任务执行实例选举

Elastic-Job使用ZooKeeper实现任务执行实例选举，若要使用ZooKeeper完成选举，就需要了解ZooKeeper的znode类型了，ZooKeeper有四种类型的znode，客户端在创建znode时可以指定：

- **PERSISTENT-持久化目录节点**

  客户端创建该类型znode，此客户端与ZooKeeper断开连接后该节点依旧存在，如果创建了重复的key，比如/data，第二次创建会失败。

- **PERSISTENT_SEQUENTIAL-持久化顺序编号目录节点**

  客户端与ZooKeeper断开连接后该节点依旧存在，允许重复创建相同key，Zookeeper给该节点名称进行顺序编号，如zk会在后面加一串数字比如 /data/data0000000001，如果重复创建，会创建一个/data/data0000000002节点（一直往后加1）

- **EPHEMERAL-临时目录节点**

  客户端与ZooKeeper断开连接后，该节点被删除，不允许重复创建相同key。

- **EPHEMERAL_SEQUENTIAL-临时顺序编号目录节点**

  客户端与ZooKeeper断开连接后，该节点被删除，允许重复创建相同key，依然采取顺序编号机制。

实例选举实现过程分析：

每个Elastic-Job的任务执行实例作为ZooKeeper的客户端来操作ZooKeeper的znode

1）任意一个实例启动时首先创建一个 /server 的**PERSISTENT**节点 2）多个实例同时创建 /server/leader **EPHEMERAL**子节点 3） /server/leader子节点只能创建一个，后创建的会失败。创建成功的实例被选为leader节点 ，用来执行任务。 4）所有任务实例监听 /server/leader 的变化，一旦节点被删除，就重新进行选举，抢占式地创建 /server/leader节点，谁创建成功谁就是leader。

2.4 小结

通过本章，我们完成了对Elastic-Job技术的快速入门程序，并了解了Elastic-Job整体架构和工作原理。

对于应用程序，只需要将任务执行细节包装为ElasticJob接口的实现类并对任务细节进行配置即可完成与Elastic-Job的集成，而Elastic-Job需要依赖Zookeeper进行执行任务信息的存取，执行任务实例的选举。通过对快速入门程序的测试，我们可以看到Elastic-Job确实解决了分布式任务调度的核心问题。

3.Spring Boot开发分布式任务

3.1.集成Spring Boot

将**Elastic-job快速入门**中的例子改造为spring boot集成方式。

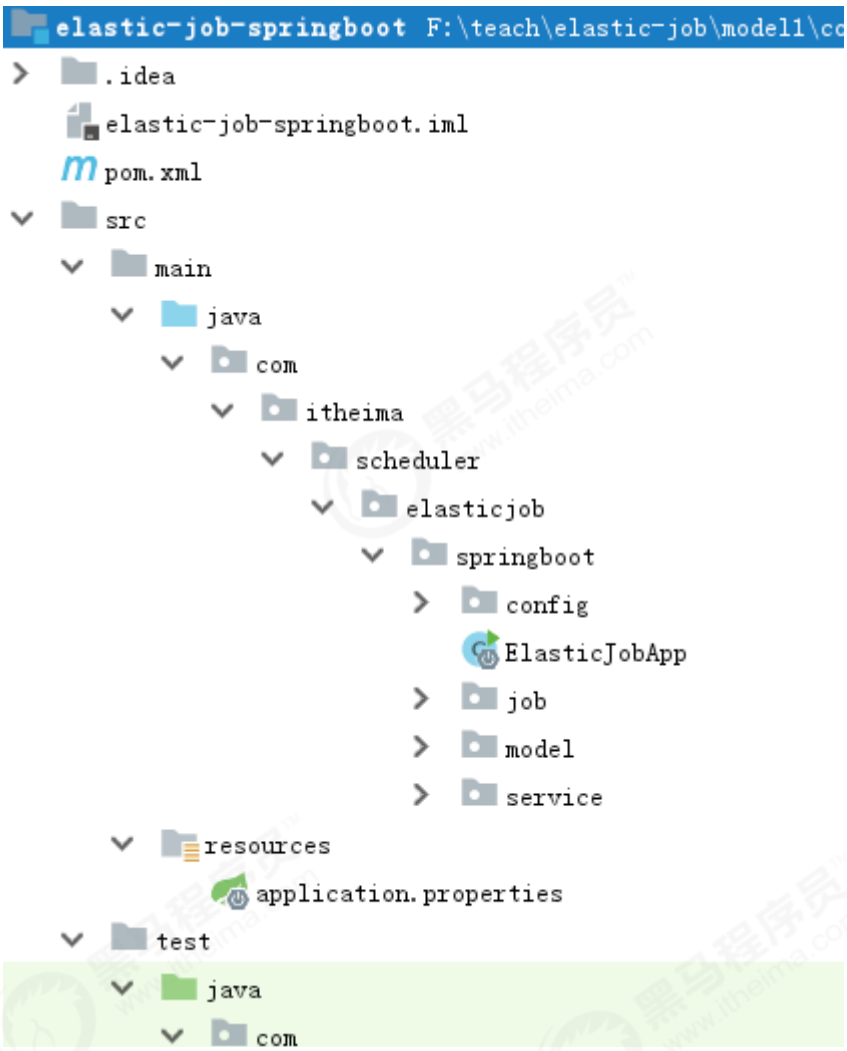
3.1.1.导入maven依赖

创建elastic-job-springboot工程，依赖如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.itheima.scheduler</groupId>
    <artifactId>elastic-job-springboot</artifactId>
    <version>1.0-SNAPSHOT</version>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.1.3.RELEASE</version>
    </parent>
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
    </properties>


    <dependencies>

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

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


        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
        </dependency>
```

```
    </dependencies>



    <build>
        <finalName>${project.name}</finalName>
        <resources>
            <resource>
                <directory>src/main/resources</directory>
                <filtering>true</filtering>
                <includes>
                    <include>**/*</include>
                </includes>
            </resource>
            <resource>
                <directory>src/main/java</directory>
                <includes>
                    <include>**/*.xml</include>
                </includes>
            </resource>
        </resources>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>

            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>

            <plugin>
                <artifactId>maven-resources-plugin</artifactId>
                <configuration>
                    <encoding>utf-8</encoding>
                    <useDefaultDelimiters>true</useDefaultDelimiters>
                </configuration>
            </plugin>
        </plugins>
    </build>


</project>
```

工程结构图如下：



3.1.2.编写spring boot配置文件及启动类

spring boot 配置文件：

```
server.port=56081
spring.application.name = task-scheduling-springboot
logging.level.root = info
```

spring boot 启动类：

```
@SpringBootApplication
public class SchedulingBootstrap {
    public static void main(String [] agrs){
        SpringApplication.run(SchedulingBootstrap.class, agrs);
    }
}
```

3.1.3.编写Elastic-Job配置类及任务类

Zookeeper配置类：

```
@Configuration
public class ElasticJobRegistryCenterConfig {

    private String registryServerList = "localhost:2181";

    private String registryNamespace = "elastic-job-example-springboot";

    /**
     * 配置Zookeeper
     * @return
     */
    @Bean(initMethod = "init")
    public CoordinatorRegistryCenter createRegistryCenter() {
        ZookeeperConfiguration zkConfig = new ZookeeperConfiguration(registryServerList,
registryNamespace);
        zkConfig.setSessionTimeoutMilliseconds(100);
        CoordinatorRegistryCenter regCenter = new ZookeeperRegistryCenter(zkConfig);
        return regCenter;
    }

}
```

Elastic-Job配置类：

```
@Configuration
public class ElasticJobConfig {

    @Autowired
    private CoordinatorRegistryCenter registryCenter;

    @Autowired
    private FileBackupJob fileBackupJob;

   /**
     * 配置任务详细信息
     * @param jobClass 任务执行类
     * @param cron  执行策略
     * @param shardingTotalCount 分片数量
     * @param shardingItemParameters 分片个性化参数
     * @return
     */
    private LiteJobConfiguration createJobConfiguration(final Class<? extends SimpleJob>
jobClass,
                                                        final String cron,
                                                        final int shardingTotalCount,
                                                        final String shardingItemParameters) {
        // 定义作业核心配置
        JobCoreConfiguration.Builder simpleCoreConfigBuilder =
JobCoreConfiguration.newBuilder(jobClass.getName(), cron, shardingTotalCount);

        if(!StringUtils.isEmpty(shardingItemParameters)){
```

```
            simpleCoreConfigBuilder.shardingItemParameters(shardingItemParameters);
        }
        JobCoreConfiguration jobCoreConfiguration =simpleCoreConfigBuilder.build();
       // 定义SIMPLE类型配置
        SimpleJobConfiguration simpleJobConfig = new
SimpleJobConfiguration(jobCoreConfiguration, jobClass.getCanonicalName());

        // 定义Lite作业根配置
        JobRootConfiguration simpleJobRootConfig =
LiteJobConfiguration.newBuilder(simpleJobConfig).overwrite(true).build();

        return  (LiteJobConfiguration) simpleJobRootConfig;
    }

    /**
     * 任务启动
     * @return
     */
    @Bean(initMethod = "init")
    public SpringJobScheduler initSimpleElasticJob() {
        SimpleJob job1 = fileBackupJob;
        SpringJobScheduler jobScheduler = new SpringJobScheduler(job1,
                registryCenter,
                createJobConfiguration(job1.getClass(),"0/3 * * * * ?",1,null));
        return  jobScheduler;
    }
}
```

Elastic-Job任务类：

使用**快速入门**中的FileBackupJob类。

3.2.作业分片

3.2.1.分片概念

作业分片是指任务的分布式执行，需要将一个任务拆分为多个独立的任务项，然后由分布式的应用实例分别执行某一个或几个分片项。

例如：**Elastic-Job快速入门**中文件备份的例子，现有2台服务器，每台服务器分别跑一个应用实例。为了快速的执行作业，那么可以将作业分成4片，每个应用实例个执行2片。作业遍历数据的逻辑应为：实例1查找text和image类型文件执行备份；实例2查找radio和video类型文件执行备份。 如果由于服务器扩容应用实例数量增加为4，则作业遍历数据的逻辑应为：4个实例分别处理text、image、radio、video类型的文件。

可以看到，通过对任务合理的分片化，从而达到任务并行处理的效果，最大限度的提高执行作业的吞吐量。

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。

最大限度利用资源

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。

3.2.2.作业分片实现

基于Spring boot集成方式的而产出的工程代码，完成对作业分片的实现，文件数据备份采取更接近真实项目的数据库存取方式。

3.2.2.1.创建数据库

数据库：mysql-5.7.25

创建elastic_job_demo数据库：

```
CREATE DATABASE `elastic_job_demo` CHARACTER SET 'utf8' COLLATE 'utf8_general_ci';
```

在elastic_job_demo库中创建t_file表：

```
DROP TABLE IF EXISTS `t_file`;
CREATE TABLE `t_file`  (
  `id` varchar(11) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL,
  `name` varchar(255) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL,
  `type` varchar(255) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL,
  `content` varchar(255) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL,
  `backedUp` tinyint(1) NULL DEFAULT NULL,
  PRIMARY KEY (`id`) USING BTREE
) ENGINE = InnoDB CHARACTER SET = utf8 COLLATE = utf8_general_ci ROW_FORMAT = Dynamic;
```

3.2.2.2.新增maven依赖

```
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-jdbc</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
        </dependency>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.47</version>
        </dependency>
```

3.2.2.3.编写文件服务类

```
@Service
```

```
public class FileService {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    /**
     * 获取某文件类型未备份的文件
     * @param fileType 文件类型
     * @param count 获取条数
     * @return
     */
    public List<FileCustom> fetchUnBackupFiles(String fileType, Integer count){
        List<FileCustom> files = jdbcTemplate.query(
                "select * from t_file t where t.type = ? and t.backedUp = 0 order by id limit
0,? "
                ,new Object[]{fileType, count}
                ,new BeanPropertyRowMapper(FileCustom.class));
        return files;
    }

    /**
     * 备份文件
     * @param files 要备份的文件
     */
    public void backupFiles(List<FileCustom> files){
        for(FileCustom file : files){
            jdbcTemplate.update("update t_file set backedUp = 1 where id = ?",new Object[]
{file.getId()});
            System.out.println(String.format("线程 %d | 已备份文件:%s  文件类型:%s"
                    ,Thread.currentThread().getId()
                    ,file.getName()
                    ,file.getType()));
        }
    }

}
```

3.2.2.4 Elastic-Job任务类

```
/**
 * 文件备份任务类
 * 每次任务执行时获取一定数目的文件，进行备份处理
 */
@Componet
public class FileBackupJobDb implements SimpleJob {

    @Autowired
    FileService fileService;

    /**
     * 每次任务执行要备份的文件数目
     */
    private final int FETCH_SIZE = 2;
```

```
    public void execute(ShardingContext shardingContext) {
        //1.获取未备份文件
        List<File> unBackupFiles =
fileService.fetchUnBackupFiles(shardingContext.getShardingParameter(), FETCH_SIZE);
        System.out.println(String.format("Time: %s | 线程 %d | 分片 %s | 已获取文件数据 %d 条"
                ,new SimpleDateFormat("HH:mm:ss").format(new Date())
                ,Thread.currentThread().getId()
                ,shardingContext.getShardingParameter()
                ,unBackupFiles.size()));
        //2.执行文件备份操作
        fileService.backupFiles(unBackupFiles);
    }
}
```

3.2.2.5.编写Elastic-Job配置类及任务类

Zookeeper配置类：

```
@Configuration
public class ElasticJobRegistryCenterConfig {

    private String registryServerList = "localhost:2181";

    private String registryNamespace = "elastic-job-example-springboot";

    /**
     * 配置Zookeeper
     * @return
     */
    @Bean(initMethod = "init")
    public CoordinatorRegistryCenter createRegistryCenter() {
        ZookeeperConfiguration zkConfig = new ZookeeperConfiguration(registryServerList,
registryNamespace);
        CoordinatorRegistryCenter regCenter = new ZookeeperRegistryCenter(zkConfig);
        return regCenter;
    }
}
```

Elastic-Job配置类：

```
@Configuration
public class ElasticJobConfig {

    @Autowired
    private CoordinatorRegistryCenter registryCenter;

    @Autowired
    private DataSource dataSource;
```

```
    @Autowired
    FileBackupJob fileBackupJob;

    /**
     * 配置任务详细信息
     * @param jobClass 任务执行类
     * @param cron  执行策略
     * @param shardingTotalCount 分片数量
     * @param shardingItemParameters 分片个性化参数
     * @return
     */
    private LiteJobConfiguration createJobConfiguration(final Class<? extends SimpleJob>
jobClass,
                                                        final String cron,
                                                        final int shardingTotalCount,
                                                        final String shardingItemParameters) {
        // 定义作业核心配置
        JobCoreConfiguration.Builder simpleCoreConfigBuilder =
JobCoreConfiguration.newBuilder(jobClass.getName(), cron, shardingTotalCount);
        if(!StringUtils.isEmpty(shardingItemParameters)){
            simpleCoreConfigBuilder.shardingItemParameters(shardingItemParameters);
        }
        JobCoreConfiguration simpleCoreConfig =simpleCoreConfigBuilder.build();
        // 定义SIMPLE类型配置
        SimpleJobConfiguration simpleJobConfig = new SimpleJobConfiguration(simpleCoreConfig,
jobClass.getCanonicalName());
        // 定义Lite作业根配置
        JobRootConfiguration simpleJobRootConfig =
LiteJobConfiguration.newBuilder(simpleJobConfig)
                .monitorPort(9888)//设置dump端口
                .build();

        return  (LiteJobConfiguration) simpleJobRootConfig;
    }

    /**
     * 任务启动
     * @return
     */
    @Bean(initMethod = "init")
    public SpringJobScheduler initSimpleElasticJob() {
        SimpleJob job1 = fileBackupJob;
        // 定义Lite作业根配置
        JobEventConfiguration jobEventConfig = new JobEventRdbConfiguration(dataSource);// 增加
任务事件追踪配置
        SpringJobScheduler jobScheduler = new SpringJobScheduler(job1,
                registryCenter,
                createJobConfiguration(job1.getClass(),"0/10 * * * *
?",4,"0=text,1=image,2=radio,3=vedio"),
                jobEventConfig);
        return  jobScheduler;
    }

}
```

3.2.2.6.编写spring boot配置文件及启动类

spring boot 配置文件：

```
server.port=56081
spring.application.name = task-scheduling-springboot
logging.level.root = info

# 数据源定义
spring.datasource.driver-class-name = com.mysql.jdbc.Driver
spring.datasource.url = jdbc:mysql://localhost:3306/elastic_job_demo?useUnicode=true
spring.datasource.username = root
spring.datasource.password = root
```

spring boot 启动类：

```
@SpringBootApplication
public class SchedulingBootstrap {
    public static void main(String [] agrs){
        SpringApplication.run(SchedulingBootstrap.class, agrs);
    }
}
```

3.2.2.7.测试

增加测试数据：

通过junit单元测试程序来增加：

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

    @Autowired
    JdbcTemplate jdbcTemplate;

    @Test
    public void testGenerateTestData(){
        //清除数据
        clearTestFiles();
        //制造数据
        generateTestFiles();
    }

    /**
     * 清除模拟数据
     */
    public void clearTestFiles(){
```

```
        jdbcTemplate.update("delete from t_file");
    }

    /**
     * 创建模拟数据
     */
    public void generateTestFiles(){
        List<FileCustom> files =new ArrayList<>();
        for(int i=1;i<11;i++){
            files.add(new FileCustom(String.valueOf(i),"文件"+ i,"text","content"+ i));
            files.add(new FileCustom(String.valueOf((i+10)),"文件"+(i+10),"image","content"+
(i+10)));
            files.add(new FileCustom(String.valueOf((i+20)),"文件"+(i+20),"radio","content"+
(i+20)));
            files.add(new FileCustom(String.valueOf((i+30)),"文件"+(i+30),"vedio","content"+
(i+30)));
        }
        for(FileCustom file : files){
            jdbcTemplate.update("insert into t_file (id,name,type,content,backedUp) values
(?,?,?,?,?)",
                    new Object[]
{file.getId(),file.getName(),file.getType(),file.getContent(),file.getBackedUp()});
        }
    }
}
```

启动Spring boot的main方法SchedulingBootstrap，并查看控制台：

```
Time: 23:31:00 | 线程 62 | 分片 text | 已获取文件数据 2 条
Time: 23:31:00 | 线程 63 | 分片 image | 已获取文件数据 2 条
Time: 23:31:00 | 线程 64 | 分片 radio | 已获取文件数据 2 条
Time: 23:31:00 | 线程 65 | 分片 vedio | 已获取文件数据 2 条
线程 62 | 已备份文件:文件1  文件类型:text
线程 63 | 已备份文件:文件11  文件类型:image
线程 64 | 已备份文件:文件21  文件类型:radio
线程 65 | 已备份文件:文件31  文件类型:vedio
线程 62 | 已备份文件:文件10  文件类型:text
线程 63 | 已备份文件:文件12  文件类型:image
线程 64 | 已备份文件:文件22  文件类型:radio
线程 65 | 已备份文件:文件32  文件类型:vedio
Time: 23:31:10 | 线程 68 | 分片 image | 已获取文件数据 2 条
Time: 23:31:10 | 线程 67 | 分片 text | 已获取文件数据 2 条
Time: 23:31:10 | 线程 69 | 分片 radio | 已获取文件数据 2 条
Time: 23:31:10 | 线程 70 | 分片 vedio | 已获取文件数据 2 条
线程 67 | 已备份文件:文件2  文件类型:text
线程 68 | 已备份文件:文件13  文件类型:image
线程 69 | 已备份文件:文件23  文件类型:radio
线程 70 | 已备份文件:文件33  文件类型:vedio
线程 68 | 已备份文件:文件14  文件类型:image
线程 67 | 已备份文件:文件3  文件类型:text
线程 70 | 已备份文件:文件34  文件类型:vedio
线程 69 | 已备份文件:文件24  文件类型:radio
```

可以看出，text、image、radio、vedio四个分片被分布到这一个实例中执行。

分片弹性扩容缩容机制测试：

elastic-job的分片是通过zookeeper来实现的。分片由主节点分配，如下三种情况都会触发主节点上的分片算法执行：

- 新的job实例加入集群
- 现有的job实例下线（如果下线的是leader节点，那么先选举然后触发分片算法的执行）
- 主节点选举

测试1：测试窗口1不关闭，再次运行main方法查看控制台日志，注意修改application.properties中的server.port，保证端口不冲突

```
测试窗口1 控制台日志如下
Time: 23:43:50 | 线程 76 | 分片 image | 已获取文件数据 2 条
Time: 23:43:50 | 线程 77 | 分片 text | 已获取文件数据 2 条
线程 76 | 已备份文件:文件17  文件类型:image
线程 77 | 已备份文件:文件6  文件类型:text
线程 76 | 已备份文件:文件18  文件类型:image
线程 77 | 已备份文件:文件7  文件类型:text

测试窗口2 控制台日志如下
Time: 23:43:50 | 线程 70 | 分片 radio | 已获取文件数据 2 条
Time: 23:43:50 | 线程 71 | 分片 vedio | 已获取文件数据 2 条
线程 70 | 已备份文件:文件27  文件类型:radio
线程 71 | 已备份文件:文件37  文件类型:vedio
线程 70 | 已备份文件:文件28  文件类型:radio
线程 71 | 已备份文件:文件38  文件类型:vedio
```

测试2：测试窗口1 和测试窗口2 不关闭，再次运行2次main方法，达到4个任务实例，查看控制台日志

```
测试窗口1 控制台日志如下
Time: 23:46:10 | 线程 33 | 分片 text | 已获取文件数据 2 条
线程 33 | 已备份文件:文件10  文件类型:text
线程 33 | 已备份文件:文件3  文件类型:text

测试窗口2 控制台日志如下
Time: 23:46:10 | 线程 34 | 分片 image | 已获取文件数据 2 条
线程 34 | 已备份文件:文件13  文件类型:image
线程 34 | 已备份文件:文件14  文件类型:image

测试窗口3 控制台日志如下
Time: 23:46:10 | 线程 33 | 分片 vedio | 已获取文件数据 2 条
线程 33 | 已备份文件:文件33  文件类型:vedio
线程 33 | 已备份文件:文件34  文件类型:vedio

测试窗口4 控制台日志如下
Time: 23:46:10 | 线程 34 | 分片 radio | 已获取文件数据 2 条
线程 34 | 已备份文件:文件23  文件类型:radio
线程 34 | 已备份文件:文件24  文件类型:radio
```

测试3：测试窗口1 和测试窗口2 不关闭，将测试窗口3和测试窗口4任务停止

```
测试窗口1 控制台日志如下
Time: 23:48:20 | 线程 82 | 分片 text | 已获取文件数据 0 条
Time: 23:48:20 | 线程 80 | 分片 image | 已获取文件数据 0 条

测试窗口2 控制台日志如下
Time: 23:48:30 | 线程 83 | 分片 text | 已获取文件数据 0 条
Time: 23:48:30 | 线程 85 | 分片 image | 已获取文件数据 0 条
```

测试4：测试窗口1不关闭 将测试窗口2 任务停止

```
测试窗口1 控制台日志如下
Time: 23:49:00 | 线程 62 | 分片 image | 已获取文件数据 0 条
Time: 23:49:00 | 线程 65 | 分片 text | 已获取文件数据 0 条
Time: 23:49:00 | 线程 69 | 分片 radio | 已获取文件数据 0 条
Time: 23:49:00 | 线程 70 | 分片 vedio | 已获取文件数据 0 条
```

查看控制台输出可以得出如下结论： 1、任务运行期间，如果有新机器加入，则会立刻触发分片机制，将任务相对平均的分配到每台机器上并行执行调度。 2、如果有机器退出集群，则经过短暂的一段时间（大约40秒）后又会重新触发分片机制

如果在设置zookeeper注册中心时，设置了session超时时间100 毫秒，则下次任务前就会触发分片

```
@Bean(initMethod = "init")
    public CoordinatorRegistryCenter createRegistryCenter() {
        ZookeeperConfiguration zkConfig = new ZookeeperConfiguration(registryServerList,
registryNamespace);
        zkConfig.setSessionTimeoutMilliseconds(100);//这里设置了session超时时间100 毫秒
        CoordinatorRegistryCenter regCenter = new ZookeeperRegistryCenter(zkConfig);
        return regCenter;
    }
```

如果在sessionTimeoutMs的时间段之内触发任务，则异常分片的任务会丢失。举个例子：假如sessionTimeoutMs被设置成1分钟，而本身的任务是30秒执行一次，有三个任务实例在三台机器各自执行分片1,2,3。当分片3所在的机器出现问题，和zookeeper断开了，那么zookeeper节点失效至少要到1分钟以后。期间30秒执行一次的任务分片3，至少会少执行一次。1分钟过后，zookeeper节点失效，触发ListenServersChangedJobListener类的dataChanged方法，在这里方法中判断instance节点变化，然后通过方法shardingService.setReshardingFlag设置重新分片标志位，下次执行任务的时候，leader节点重新分配分片，分片3就会转移到其他好的机器上。

3.2.3.作业配置说明

注册中心配置

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

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。

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，允许将该次未完成的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

3.2.4.作业分片策略

AverageAllocationJobShardingStrategy

全路径：

com.dangdang.ddframe.job.lite.api.strategy.impl.AverageAllocationJobShardingStrategy

策略说明：

基于平均分配算法的分片策略，也是默认的分片策略。

如果分片不能整除，则不能整除的多余分片将依次追加到序号小的服务器。如：

如果有3台服务器，分成9片，则每台服务器分到的分片是：1=[0,1,2], 2=[3,4,5], 3=[6,7,8]

如果有3台服务器，分成8片，则每台服务器分到的分片是：1=[0,1,6], 2=[2,3,7], 3=[4,5]

如果有3台服务器，分成10片，则每台服务器分到的分片是：1=[0,1,2,9], 2=[3,4,5], 3=[6,7,8]

OdevitySortByNameJobShardingStrategy

全路径：

com.dangdang.ddframe.job.lite.api.strategy.impl.OdevitySortByNameJobShardingStrategy

策略说明：

根据作业名的哈希值奇偶数决定IP升降序算法的分片策略。

作业名的哈希值为奇数则IP升序。

作业名的哈希值为偶数则IP降序。

用于不同的作业平均分配负载至不同的服务器。

AverageAllocationJobShardingStrategy的缺点是，一旦分片数小于作业服务器数，作业将永远分配至IP地址靠前的服务器，导致IP地址靠后的服务器空闲。而OdevitySortByNameJobShardingStrategy则可以根据作业名称重新分配服务器负载。如：

如果有3台服务器，分成2片，作业名称的哈希值为奇数，则每台服务器分到的分片是：1=[0], 2=[1], 3=[]

如果有3台服务器，分成2片，作业名称的哈希值为偶数，则每台服务器分到的分片是：3=[0], 2=[1], 1=[]

RotateServerByNameJobShardingStrategy

全路径：

com.dangdang.ddframe.job.lite.api.strategy.impl.RotateServerByNameJobShardingStrategy

策略说明：

根据作业名的哈希值对服务器列表进行轮转的分片策略。

配置分片策略

与配置通常的作业属性相同，在spring命名空间或者JobConfiguration中配置jobShardingStrategyClass属性，属性值是作业分片策略类的全路径。

分片策略配置xml方式：

```
 <job:simple id="hotelSimpleSpringJob" class="com.chuanzhi.spiderhotel.job.SpiderJob" registry-center-ref="regCenter" cron="0/10 * * * * ?" sharding-total-count="4" sharding-item-parameters="0=A,1=B,2=C,3=D" monitor-port="9888"  reconcile-interval-minutes="10" job-sharding-strategy-class="com.dangdang.ddframe.job.lite.api.strategy.impl.RotateServerByNameJobShardingStrategy"/>
```

分片策略配置java方式：

```
// 定义Lite作业根配置
        JobRootConfiguration simpleJobRootConfig = LiteJobConfiguration.newBuilder(simpleJobConfig).jobShardingStrategyClass("com.dangdang.ddframe.job.lite.api.strategy.impl.OdevitySortByNameJobShardingStrategy").build();
```

3.3.Dataflow类型定时任务

Dataflow类型的定时任务需实现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。咱们继续对例子进行改造。

Dataflow类型用于处理数据流，它和SimpleJob不同，它以数据流的方式执行，调用fetchData抓取数据，直到抓取不到数据才停止作业。

新增FileBackupDataFlowJob：

```
/**
 * 文件备份任务类
 * 数据流模式
 */
@Componet
public class FileBackupDataFlowJob implements DataflowJob<FileCustom> {

    @Autowired
    FileService fileService;
    /**
     * 每次任务执行要备份的文件数目
     */
    private final int FETCH_SIZE = 2;

    //抓取数据
    @Override
    public List<File> fetchData(ShardingContext shardingContext) {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        //1.获取未备份文件
        List<File> unBackupFiles = fileService.fetchUnBackupFiles(shardingContext.getShardingParameter(), FETCH_SIZE);

        System.out.println(String.format("Time: %s | 线程 %d | 分片 %s | 已获取文件数据 %d 条"
```

```
                ,new SimpleDateFormat("HH:mm:ss").format(new Date())
                ,Thread.currentThread().getId()
                ,shardingContext.getShardingParameter()
                ,unBackupFiles.size()));
        return unBackupFiles;
    }

    //处理数据
    @Override
    public void processData(ShardingContext shardingContext, List<File> unBackupFiles) {

        //1.执行文件备份操作
        fileService.backupFiles(unBackupFiles);
    }
}
```

ElasticJobConfig中新增配置：

```
    @Autowired
    FileBackupDataFlowJob fileBackupDataFlowJob;
/**
     * 配置数据流处理任务详细信息
     * @param jobClass 任务执行类
     * @param cron  执行策略
     * @param shardingTotalCount 分片数量
     * @param shardingItemParameters 分片个性化参数
     * @return
     */
    private LiteJobConfiguration createFlowJobConfiguration(final Class<? extends ElasticJob>
jobClass,
                                                            final String cron,
                                                            final int shardingTotalCount,
                                                            final String shardingItemParameters) {
        // 定义作业核心配置
        JobCoreConfiguration.Builder coreConfigBuilder =
JobCoreConfiguration.newBuilder(jobClass.getName(), cron, shardingTotalCount);
        if(!StringUtils.isEmpty(shardingItemParameters)){
            coreConfigBuilder.shardingItemParameters(shardingItemParameters);
        }
        JobCoreConfiguration coreConfig =coreConfigBuilder.build();
        // 定义数据流类型任务配置
        DataflowJobConfiguration jobConfig = new DataflowJobConfiguration(coreConfig,
jobClass.getCanonicalName(),true);

        // 定义Lite作业根配置
        JobRootConfiguration simpleJobRootConfig =
LiteJobConfiguration.newBuilder(jobConfig).build();

        return  (LiteJobConfiguration) simpleJobRootConfig;
    }
```

```
    /**
     * 启动数据流任务
     * @return
     */
    @Bean(initMethod = "init")
    public SpringJobScheduler initFlowElasticJob() {
        SpringJobScheduler jobScheduler = new SpringJobScheduler(fileBackupDataFlowJob,
                registryCenter,
                createFlowJobConfiguration(fileBackupDataFlowJob.getClass(),"0/10 * * * *
?",4,"0=text,1=image,2=radio,3=vedio"));
        return  jobScheduler;
    }
```

启动应用后，日志输出如下：

```
Time: 00:08:11 | 线程 66 | 分片 image | 已获取文件数据 2 条
Time: 00:08:11 | 线程 68 | 分片 vedio | 已获取文件数据 2 条
Time: 00:08:11 | 线程 65 | 分片 text | 已获取文件数据 2 条
Time: 00:08:11 | 线程 67 | 分片 radio | 已获取文件数据 2 条
线程 66 | 已备份文件:文件11  文件类型:image
线程 68 | 已备份文件:文件31  文件类型:vedio
线程 65 | 已备份文件:文件1  文件类型:text
线程 67 | 已备份文件:文件21  文件类型:radio
线程 66 | 已备份文件:文件12  文件类型:image
线程 65 | 已备份文件:文件10  文件类型:text
线程 68 | 已备份文件:文件32  文件类型:vedio
线程 67 | 已备份文件:文件22  文件类型:radio
Time: 00:08:12 | 线程 66 | 分片 image | 已获取文件数据 2 条
Time: 00:08:12 | 线程 65 | 分片 text | 已获取文件数据 2 条
Time: 00:08:12 | 线程 68 | 分片 vedio | 已获取文件数据 2 条
线程 66 | 已备份文件:文件13  文件类型:image
Time: 00:08:12 | 线程 67 | 分片 radio | 已获取文件数据 2 条
线程 68 | 已备份文件:文件33  文件类型:vedio
线程 65 | 已备份文件:文件2  文件类型:text
线程 67 | 已备份文件:文件23  文件类型:radio
线程 66 | 已备份文件:文件14  文件类型:image
线程 65 | 已备份文件:文件3  文件类型:text
线程 68 | 已备份文件:文件34  文件类型:vedio
线程 67 | 已备份文件:文件24  文件类型:radio
Time: 00:08:13 | 线程 66 | 分片 image | 已获取文件数据 2 条
Time: 00:08:13 | 线程 68 | 分片 vedio | 已获取文件数据 2 条
Time: 00:08:13 | 线程 65 | 分片 text | 已获取文件数据 2 条
Time: 00:08:13 | 线程 67 | 分片 radio | 已获取文件数据 2 条
线程 66 | 已备份文件:文件15  文件类型:image
线程 68 | 已备份文件:文件35  文件类型:vedio
线程 65 | 已备份文件:文件4  文件类型:text
线程 67 | 已备份文件:文件25  文件类型:radio
线程 66 | 已备份文件:文件16  文件类型:image
线程 65 | 已备份文件:文件5  文件类型:text
线程 67 | 已备份文件:文件26  文件类型:radio
线程 68 | 已备份文件:文件36  文件类型:vedio
Time: 00:08:14 | 线程 66 | 分片 image | 已获取文件数据 2 条
Time: 00:08:14 | 线程 67 | 分片 radio | 已获取文件数据 2 条
```

```
Time: 00:08:14 | 线程 65 | 分片 text | 已获取文件数据 2 条
Time: 00:08:14 | 线程 68 | 分片 vedio | 已获取文件数据 2 条
线程 66 | 已备份文件:文件17  文件类型:image
线程 68 | 已备份文件:文件37  文件类型:vedio
线程 67 | 已备份文件:文件27  文件类型:radio
线程 65 | 已备份文件:文件6  文件类型:text
线程 66 | 已备份文件:文件18  文件类型:image
线程 65 | 已备份文件:文件7  文件类型:text
线程 68 | 已备份文件:文件38  文件类型:vedio
线程 67 | 已备份文件:文件28  文件类型:radio
Time: 00:08:15 | 线程 66 | 分片 image | 已获取文件数据 2 条
Time: 00:08:15 | 线程 65 | 分片 text | 已获取文件数据 2 条
线程 66 | 已备份文件:文件19  文件类型:image
Time: 00:08:15 | 线程 67 | 分片 radio | 已获取文件数据 2 条
Time: 00:08:15 | 线程 68 | 分片 vedio | 已获取文件数据 2 条
线程 65 | 已备份文件:文件8  文件类型:text
线程 68 | 已备份文件:文件39  文件类型:vedio
线程 66 | 已备份文件:文件20  文件类型:image
线程 67 | 已备份文件:文件29  文件类型:radio
线程 65 | 已备份文件:文件9  文件类型:text
线程 68 | 已备份文件:文件40  文件类型:vedio
线程 67 | 已备份文件:文件30  文件类型:radio
Time: 00:08:16 | 线程 66 | 分片 image | 已获取文件数据 0 条
Time: 00:08:16 | 线程 65 | 分片 text | 已获取文件数据 0 条
Time: 00:08:16 | 线程 68 | 分片 vedio | 已获取文件数据 0 条
Time: 00:08:16 | 线程 67 | 分片 radio | 已获取文件数据 0 条
Time: 00:08:21 | 线程 70 | 分片 image | 已获取文件数据 0 条
Time: 00:08:21 | 线程 69 | 分片 text | 已获取文件数据 0 条
Time: 00:08:21 | 线程 71 | 分片 radio | 已获取文件数据 0 条
Time: 00:08:21 | 线程 72 | 分片 vedio | 已获取文件数据 0 条
```

从输出日志可以看出，每次运行定时任务都会开启4个线程执行fetchData抓取数据，抓取以后调用processData处理数据，如果是流式处理数据（new DataflowJobConfiguration第三个参数为true）且fetchData方法的返回值为null或集合长度为空时，作业才停止处理。

4.Elastic-Job高级

4.1 事件追踪

Elastic-Job-Lite在配置中提供了JobEventConfiguration，支持数据库方式配置，会在数据库中自动创建JOB_EXECUTION_LOG和JOB_STATUS_TRACE_LOG两张表以及若干索引，来记录作业的相关信息。

4.1.1.修改Elastic-Job配置类

在ElasticJobConfig中修改：

```
    @Autowired
    private DataSource dataSource; //数据源已经存在，直接引入

    @Bean(initMethod = "init")
    public SpringJobScheduler initSimpleElasticJob() {
        SimpleJob job1 = new FileBackupJob();
        JobEventConfiguration jobEventConfig = new JobEventRdbConfiguration(dataSource);// 增加
任务事件追踪配置
        SpringJobScheduler jobScheduler = new SpringJobScheduler(job1,
                registryCenter,
                createJobConfiguration(job1.getClass(),"0/10 * * * *
?",4,"0=text,1=image,2=radio,3=vedio"),
                jobEventConfig);
        return  jobScheduler;
    }
```

4.1.2.启动项目

启动后会发现在elastic_job_demo数据库中新增以下两个表。

job_execution_log：

对象 | job_execution_log @elastic...

保存 | 添加字段 | 插入字段 | 删除字段 | 主键 | 上移 | 下移

字段 | 索引 | 外键 | 触发器 | 选项 | 注释 | SQL 预览

名	类型	长度	小数点	不是 null	虚拟
id	varchar	40	0	☑	☐
job_name	varchar	100	0	☑	☐
task_id	varchar	255	0	☑	☐
hostname	varchar	255	0	☑	☐
ip	varchar	50	0	☑	☐
sharding_item	int	11	0	☑	☐
execution_source	varchar	20	0	☑	☐
failure_cause	varchar	4000	0	☐	☐
is_success	int	11	0	☑	☐
start_time	timestamp	0	0	☐	☐
complete_time	timestamp	0	0	☐	☐

job_status_trace_log：

对象 job_status_trace_log @elast...

保存 添加字段 插入字段 删除字段 主键 上移 下移

字段 索引 外键 触发器 选项 注释 SQL 预览

名	类型	长度	小数点	不是 null	虚拟	键
id	varchar	40	0	☑	☐	1
job_name	varchar	100	0	☑	☐	
original_task_id	varchar	255	0	☑	☐	
task_id	varchar	255	0	☑	☐	
slave_id	varchar	50	0	☑	☐	
source	varchar	50	0	☑	☐	
execution_type	varchar	20	0	☑	☐	
sharding_item	varchar	100	0	☑	☐	
state	varchar	20	0	☑	☐	
message	varchar	4000	0	☐	☐	
creation_time	timestamp	0	0	☐	☐	

对象 job_execution_log @elastic...

开始事务 文本 筛选 排序 导入 导出

id	job_name	task_id	hostname	ip	sharding_item	execution_source	failure_cause	is_success	start_time	complete_time
298c716f-fec0-4678	com.itheima.schedu	com.itheima.schedu	DESKTOP-VVN6STP	192.168.56.1	0	NORMAL_TRIGGER	(Null)	1		
47f7f64e-1dff-4000-	com.itheima.schedu	com.itheima.schedu	DESKTOP-VVN6STP	192.168.56.1	2	NORMAL_TRIGGER	(Null)	1		
4876831e-d47d-4b0	com.itheima.schedu	com.itheima.schedu	DESKTOP-VVN6STP	192.168.56.1	1	NORMAL_TRIGGER	(Null)	1		
867f165f-c50b-4c90	com.itheima.schedu	com.itheima.schedu	DESKTOP-VVN6STP	192.168.56.1	0	NORMAL_TRIGGER	(Null)	1		
9147c901-eef4-43a6	com.itheima.schedu	com.itheima.schedu	DESKTOP-VVN6STP	192.168.56.1	3	NORMAL_TRIGGER	(Null)	1		
9bc36d72-3563-4ac	com.itheima.schedu	com.itheima.schedu	DESKTOP-VVN6STP	192.168.56.1	2	NORMAL_TRIGGER	(Null)	1		
bcbe0e83-da97-48a	com.itheima.schedu	com.itheima.schedu	DESKTOP-VVN6STP	192.168.56.1	1	NORMAL_TRIGGER	(Null)	1		
d10ade94-bc28-456	com.itheima.schedu	com.itheima.schedu	DESKTOP-VVN6STP	192.168.56.1	0	NORMAL_TRIGGER	(Null)	1		
da14670b-0fbe-40c	com.itheima.schedu	com.itheima.schedu	DESKTOP-VVN6STP	192.168.56.1	2	NORMAL_TRIGGER	(Null)	1		
dfe0fe0d-e5c8-4788	com.itheima.schedu	com.itheima.schedu	DESKTOP-VVN6STP	192.168.56.1	3	NORMAL_TRIGGER	(Null)	1		
e537e201-fb52-48c1	com.itheima.schedu	com.itheima.schedu	DESKTOP-VVN6STP	192.168.56.1	3	NORMAL_TRIGGER	(Null)	1		
fefc7f9a-86c2-4eae-	com.itheima.schedu	com.itheima.schedu	DESKTOP-VVN6STP	192.168.56.1	1	NORMAL_TRIGGER	(Null)	1		

JOB_EXECUTION_LOG记录每次作业的执行历史。分为两个步骤：

1. 作业开始执行时向数据库插入数据，除failure_cause和complete_time外的其他字段均不为空。
2. 作业完成执行时向数据库更新数据，更新is_success, complete_time和failure_cause(如果作业执行失败)。

JOB_STATUS_TRACE_LOG记录作业状态变更痕迹表。可通过每次作业运行的task_id查询作业状态变化的生命周期和运行轨迹。

4.2 运维

elastic-job中提供了一个elastic-job-lite-console控制台

设计理念

1. 本控制台和Elastic Job并无直接关系，是通过读取Elastic Job的注册中心数据展现作业状态，或更新注册中心数据修改全局配置。
2. 控制台只能控制作业本身是否运行，但不能控制作业进程的启停，因为控制台和作业本身服务器是完全分布式的，控制台并不能控制作业服务器。

主要功能

1. 查看作业以及服务器状态
2. 快捷的修改以及删除作业设置
3. 启用和禁用作业

4. 跨注册中心查看作业
5. 查看作业运行轨迹和运行状态

不支持项

1. 添加作业。因为作业都是在首次运行时自动添加，使用控制台添加作业并无必要。直接在作业服务器启动包含Elastic Job的作业进程即可

具体搭建步骤如下：

3.4.1.搭建

下载地址：https://raw.githubusercontent.com/miguangying/elastic-job-lite-console/master/elastic-job-lite-console-2.1.4.tar.gz

解压缩 `elastic-job-lite-console-${version}.tar.gz` 。

进入 bin目录 并执行：

```
bin\start.sh
```

打开浏览器访问 `http://localhost:8899/` 即可访问控制台。8899为默认端口号，可通过启动脚本输入-p自定义端口号。

`elastic-job-lite-console-${version}.tar.gz` 也可通过 elastic-job 源码用 mvn install编译获取。

输入用户名 root 密码 root 即可打开主界面 如下图

提供两种账户，管理员及访客，管理员拥有全部操作权限，访客仅拥有察看权限。默认管理员用户名和密码是root/root，访客用户名和密码是guest/guest，可通过conf\auth.properties修改管理员及访客用户名及密码。

3.4.2.配置及使用

1、 配置注册中心地址

先启动zookeeper 然后在注册中心配置界面 点添加

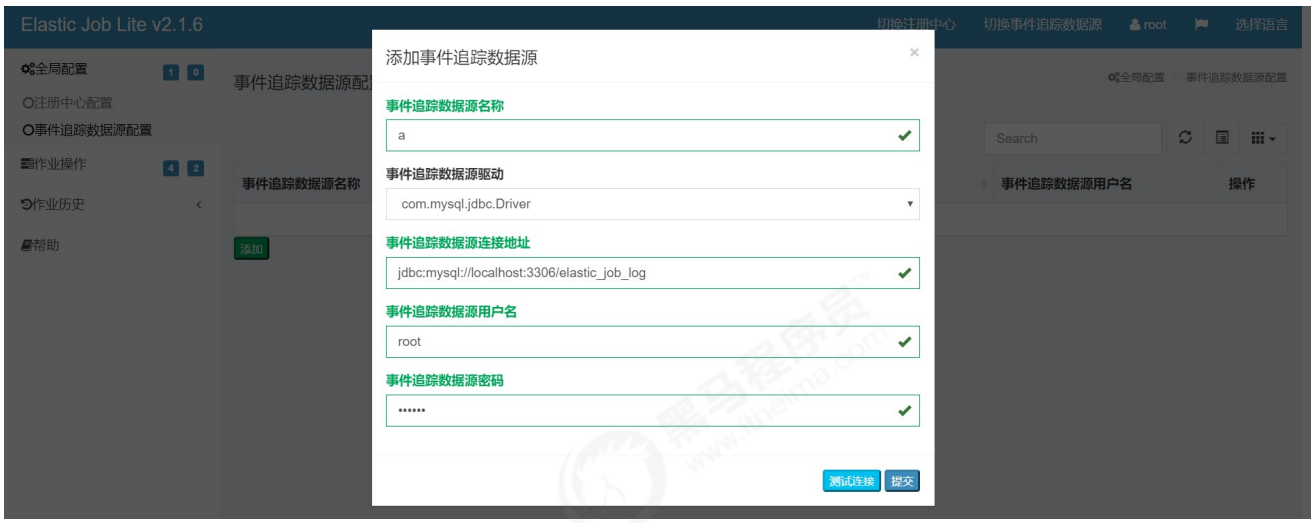
点击提交后，然后点连接（zookeeper必须处于启动状态）

连接成功后，在作业维度下可以显示该命名空间下作业名称、分片数量及该作业的cron表达式等信息

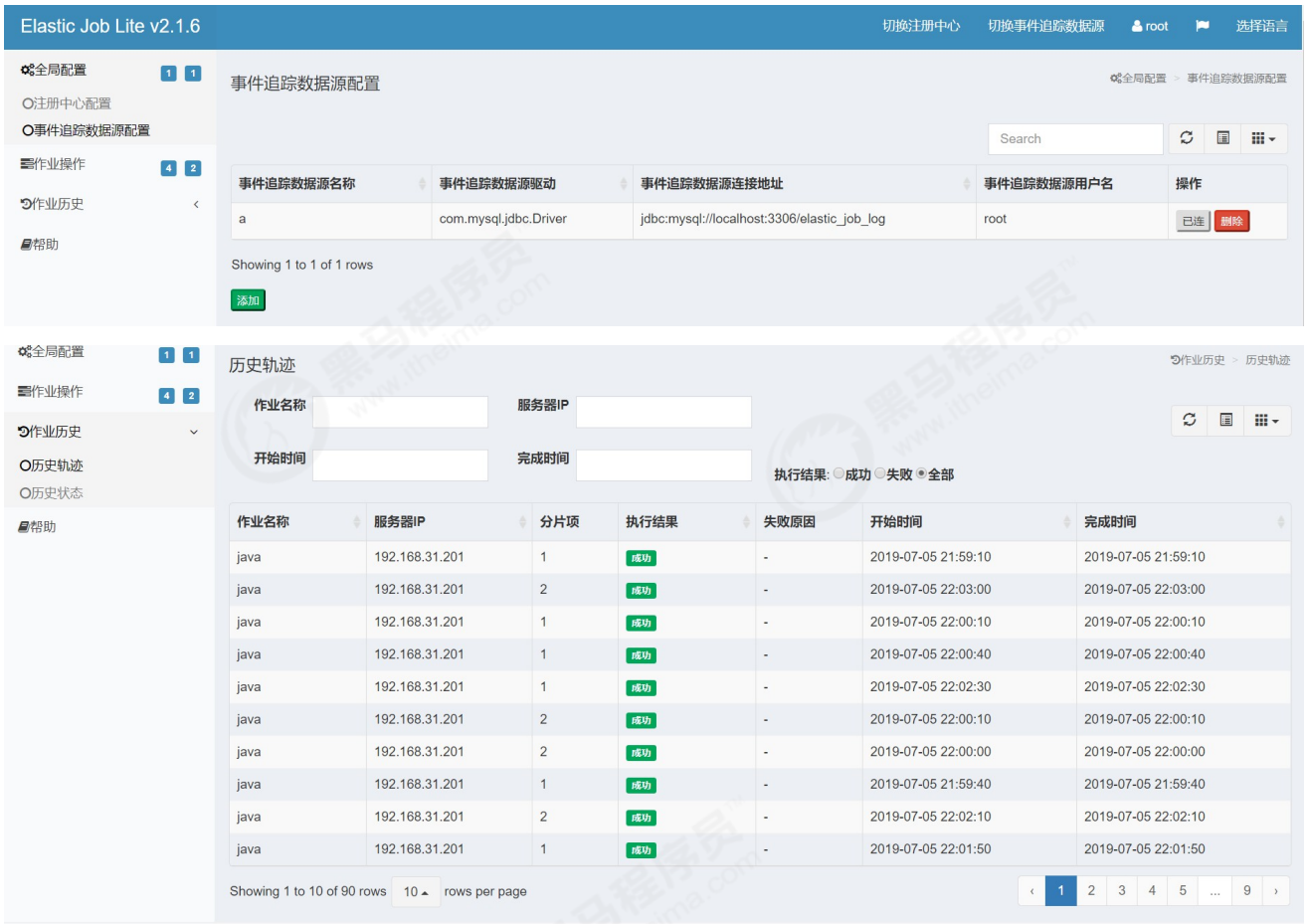
在服务器维度可以查看服务器ip、当前运行的实例数、作业总数等信息。

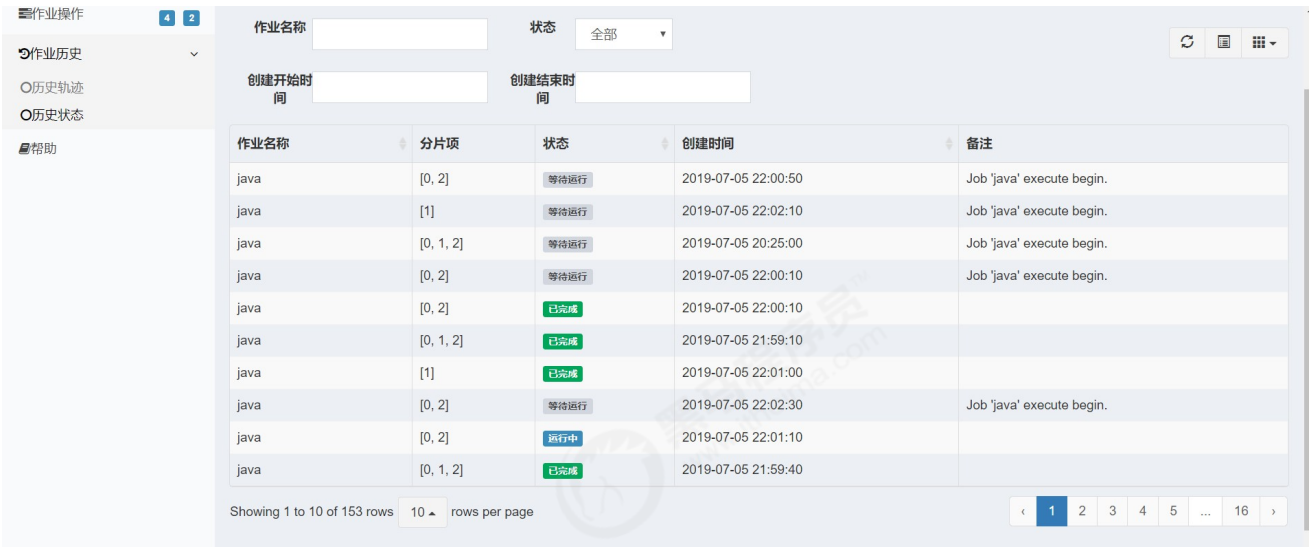
2、配置事件追踪数据源

在事件追踪数据源配置页面点添加按钮，输入相关信息



提交后点击连接即可在作业历史下查看作业历史记录





4.3 dump命令

使用Elastic-Job-Lite过程中可能会碰到一些问题，导致作业运行不稳定。由于无法在生产环境调试，通过dump命令可以把作业内部相关信息dump出来，方便开发者debug分析。

（1）开启dump监控端口，并运行程序

修改中ElasticJobConfig中的createJobConfiguration方法里JobRootConfiguration的配置，开启dump监控端口：

```
    JobRootConfiguration simpleJobRootConfig = LiteJobConfiguration.newBuilder(simpleJobConfig)
            .monitorPort(9888)//设置dump端口
            .build();
```

（2）windows中安装netcat（若操作系统中已经有nc命令，此步骤可略过）

tools文件夹内包含netcat-win32-1.12.zip，解压即可。

（3）执行dump命令

打开命令行工具，进入netcat-win32-1.12.zip的解压目录，执行以下命令：

```
echo dump| nc 127.0.0.1 9888 > job_debug_dump.txt
```

会在当前目录生成job_debug_dump.txt文件，打开job_debug_dump.txt后看到：

```
/com.itheima.scheduling.job.FileBackupJob/sharding |
/com.itheima.scheduling.job.FileBackupJob/sharding/3 |
/com.itheima.scheduling.job.FileBackupJob/sharding/3/instance | ip1@-@20708
/com.itheima.scheduling.job.FileBackupJob/sharding/2 |
/com.itheima.scheduling.job.FileBackupJob/sharding/2/instance | ip1@-@20708
/com.itheima.scheduling.job.FileBackupJob/sharding/1 |
/com.itheima.scheduling.job.FileBackupJob/sharding/1/instance | ip1@-@20708
/com.itheima.scheduling.job.FileBackupJob/sharding/0 |
/com.itheima.scheduling.job.FileBackupJob/sharding/0/instance | ip1@-@20708
/com.itheima.scheduling.job.FileBackupJob/servers |
/com.itheima.scheduling.job.FileBackupJob/servers/ip1 |
```

```
/com.itheima.scheduling.job.FileBackupJob/leader |
/com.itheima.scheduling.job.FileBackupJob/leader/sharding |
/com.itheima.scheduling.job.FileBackupJob/leader/election |
/com.itheima.scheduling.job.FileBackupJob/leader/election/latch |
/com.itheima.scheduling.job.FileBackupJob/leader/election/instance | ip1@-@20708
/com.itheima.scheduling.job.FileBackupJob/instances |
/com.itheima.scheduling.job.FileBackupJob/instances/ip1@-@20708 |
/com.itheima.scheduling.job.FileBackupJob/config |
{"jobName":"com.itheima.scheduling.job.FileBackupJob","jobClass":"com.itheima.scheduling.job.File
BackupJob","jobType":"SIMPLE","cron":"0/10 * * * *
?","shardingTotalCount":4,"shardingItemParameters":"0\u003dtext,1\u003dimage,2\u003dradio,3\u003d
vedio","jobParameter":"","failover":false,"misfire":true,"description":"","jobProperties":
{"job_exception_handler":"com.dangdang.ddframe.job.executor.handler.impl.DefaultJobExceptionHandl
er","executor_service_handler":"com.dangdang.ddframe.job.executor.handler.impl.DefaultExecutorSer
viceHandler"},"monitorExecution":true,"maxTimeDiffSeconds":-1,"monitorPort":9888,"jobShardingStra
tegyClass":"","reconcileIntervalMinutes":10,"disabled":false,"overwrite":false}
```

里面展示的其实就是FileBackupJob任务在Zookeeper中的信息。

5.课程总结

重要知识点回顾：

什么是任务调度？

任务调度的应用场景？

什么是分布式任务调度？

分布式任务调度需要解决那些问题？这些问题的大致解决思路？

Elastic-Job是什么？

Zookeeper在Elastic-Job整个架构中起到了什么作用？

Elastic-Job分片的概念？分片是为了解决什么问题？

Elastic-Job主要的配置类有哪些？各职责？

Dataflow任务类型和SimpleJob类型有什么不同？