

课程目标

1、了解 Elastic-Job 的基本特性

2、掌握 Elastic-Job 开发与配置方式（包括 Java 开发和 Spring Boot 开发），掌握

任务类型和任务分片策略

3、了解 Elastic-Job 运维平台的使用

4、掌握 Elastic-Job 运行原理

内容定位

适合了解了 Quartz 的调度模型之后，想要知道如何基于 ZK 配置 Quartz 和如何实

现任务分片的同学

Quartz-Misfire

什么情况下错过触发？错过触发怎么办？

线程池只有 5 个线程，当有 5 个任务都在执行的时候，第六个任务即将触发，这个时候任务就不能得到执行。在 quartz.properties 有一个属性 misfireThreshold，用来定义触发器超时的“临界值”，也就是超过了这个时间，就算错过触发了。

例如，如果 misfireThreshold 是 60000（60 秒），9 点整应该执行的任务，9 点零 1 分还没有可用线程执行它，就会超时（misfires）。

下面这些原因可能造成 misfired job:

- 1、 没有可用线程
- 2、 Trigger 被暂停
- 3、 系统重启
- 4、 禁止并发执行的任务在到达触发时间时，上次执行还没有结束。

错过触发怎么办？Misfire 策略设置

每一种 Trigger 都定义了自己的 Misfire 策略，不同的策略通过不同的方法来设置。

standalone 工程 MisfireTest

```
Trigger trigger = TriggerBuilder.newTrigger().withIdentity("trigger1", "group1").startNow()
    .withSchedule(SimpleScheduleBuilder.simpleSchedule())
    .withMisfireHandlingInstructionNowWithExistingCount().
    withIntervalInSeconds(1).
    repeatForever().build();
```

大体上来说有 3 种：

- 1、 忽略
- 2、 立即跑一次
- 3、 下次跑

详细内容参考：

<https://gper.club/articles/7e7e7f7ff7g59gc5g69>

怎么避免任务错过触发？

合理地设置线程池数量，以及任务触发间隔。

1 认识 E-Job

1.1 任务调度高级需求

Quartz 的不足：

- 1、 作业只能通过 DB 抢占随机负载，无法协调
- 2、 任务不能分片——单个任务数据太多了跑不完，消耗线程，负载不均
- 3、 作业日志可视化监控、统计

1.2 发展历史

E-Job 是怎么来的？

在当当的 ddframe 框架中，需要一个任务调度系统（作业系统）。



实现的话有两种思路，一个是修改开源产品，一种是基于开源产品搭建（封装），当当选择了后者，最开始这个调度系统叫做 dd-job。它是一个无中心化的分布式调度框架。因为数据库缺少分布式协调功能（比如选主），替换为 Zookeeper 后，增加了弹性扩容和数据分片的功能。

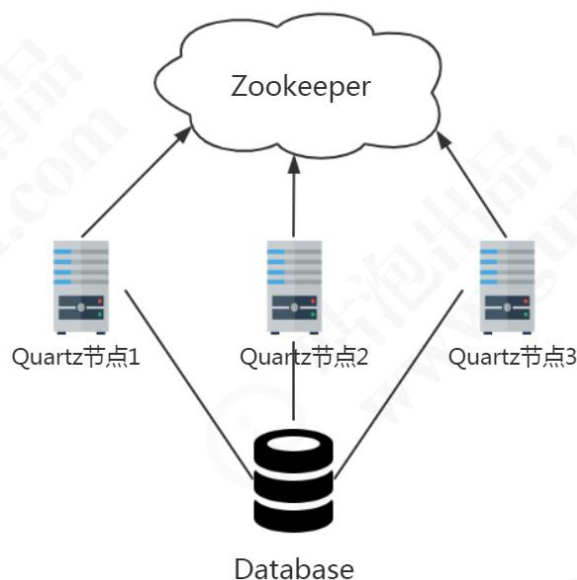
Elastic-Job 是 ddframe 中的 dd-job 作业模块分离出来的作业框架，基于 Quartz 和 Curator 开发，在 2015 年开源。

轻量级，无中心化解决方案。

为什么说去中心化呢？因为没有统一的调度中心。集群的每个节点都是对等的，节点之间通过注册中心进行分布式协调。E-Job 存在主节点的概念，但是主节点没有调度的功能，而是用于处理一些集中式任务，如分片，清理运行时信息等。

思考：如果 ZK 挂了怎么办？

每个任务有独立的线程池。



从官网开始

<http://elasticjob.io/docs/elastic-job-lite/00-overview/>

<https://github.com/elasticjob>

Elastic-Job 最开始只有一个 elastic-job-core 的项目，在 2.X 版本以后主要分为 Elastic-Job-Lite 和 Elastic-Job-Cloud 两个子项目。其中，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。而 Elastic-Job-Cloud 使用 Mesos + Docker 的解决方案，额外提供资源治理、应用分发以

及进程隔离等服务（跟 Lite 的区别只是部署方式不同，他们使用相同的 API，只要开发一次）。

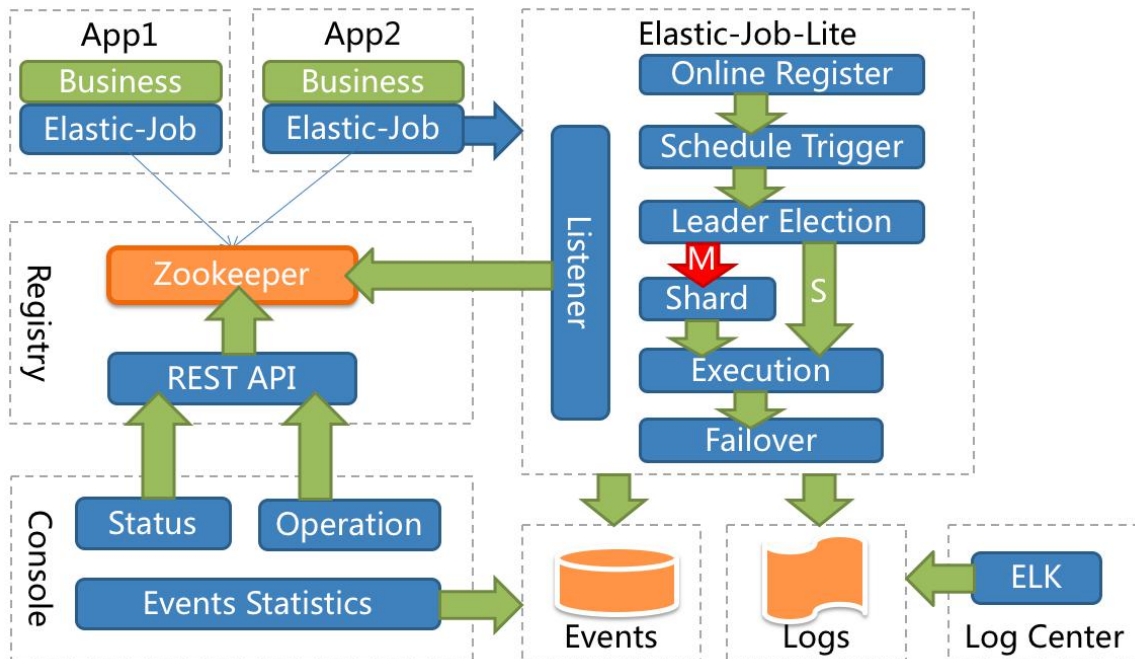
1.3 功能特性

- 分布式调度协调：用 ZK 实现注册中心
- 错过执行作业重触发（Misfire）
- 支持并行调度（任务分片）
- 作业分片一致性，保证同一分片在分布式环境中仅一个执行实例
- 弹性扩容缩容：将任务拆分为 n 个任务项后，各个服务器分别执行各自分配到的任务项。一旦有新的服务器加入集群，或现有服务器下线，elastic-job 将在保留本次任务执行不变的情况下，下次任务开始前触发任务重分片。
- 失效转移 failover：弹性扩容缩容在下次作业运行前重分片，但本次作业执行的过程中，下线的服务器所分配的作业将不会重新被分配。失效转移功能可以在本次作业运行中用空闲服务器抓取孤儿作业分片执行。同样失效转移功能也会牺牲部分性能。
- 支持作业生命周期操作（Listener）
- 丰富的作业类型（Simple、DataFlow、Script）
- Spring 整合以及命名空间提供
- 运维平台

1.4 项目架构

应用在各自的节点执行任务，通过 ZK 注册中心协调。节点注册、节点选举、任务分

片、监听都在 E-Job 的代码中完成。



2 Java 开发

工程：ejob-standalone

2.1 pom 依赖

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

2.2 任务类型

standalone 工程

任务类型有三种：

2.2.1 SimpleJob

SimpleJob: 简单实现，未经任何封装的类型。需实现 SimpleJob 接口。

ejob-standalone MySimpleJob.java

```
public class MyElasticJob implements SimpleJob {
    public void execute(ShardingContext context) {
        System.out.println(String.format("Item: %s | Time: %s | Thread: %s ",
            context.getShardingItem(), new SimpleDateFormat("HH:mm:ss").format(new Date()),
            Thread.currentThread().getId()));
    }
}
```

2.2.2 DataFlowJob

DataFlowJob : Dataflow 类型用于处理数据流，必须实现 fetchData()和 processData()的方法，一个用来获取数据，一个用来处理获取到的数据。

ejob-standalone MyDataFlowJob.java

```
public class MyDataFlowJob implements DataflowJob<String> {
    @Override
    public List<String> fetchData(ShardingContext shardingContext) {
        // 获取到了数据
        return Arrays.asList("qingshan","jack","seven");
    }

    @Override
    public void processData(ShardingContext shardingContext, List<String> data) {
        data.forEach(x-> System.out.println("开始处理数据: "+x));
    }
}
```

2.2.3 ScriptJob

Script : Script 类型作业意为脚本类型作业，支持 shell，python，perl 等所有类型脚本。D 盘下新建 1.bat，内容：

```
@echo ----- 【脚本任务】 Sharding Context: %*
```

ejob-standalone script.ScriptJobTest

只要指定脚本的内容或者位置

2.3 E-Job 配置

2.3.1 配置步骤

配置手册：<http://elasticjob.io/docs/elastic-job-lite/02-guide/config-manual/>

1、ZK 注册中心配置（后面继续分析）

2、作业配置（从底层往上层：Core——Type——Lite）

配置级别	配置类	配置内容
Core	JobCoreConfiguration	用于提供作业核心配置信息，如：作业名称、CRON 表达式、分片总数等。
Type	JobTypeConfiguration	有 3 个子类分别对应 SIMPLE, DATAFLOW 和 SCRIPT 类型作业，提供 3 种作业需要的不同配置，如：DATAFLOW 类型是否流式处理或 SCRIPT 类型的命令行等。Simple 和 DataFlow 需要指定任务类的路径。
Root	JobRootConfiguration	有 2 个子类分别对应 Lite 和 Cloud 部署类型，提供不同部署类型所需的配置，如：Lite 类型的是否需要覆盖本地配置或 Cloud 占用 CPU 或 Memory 数量等。 可以定义分片策略。 http://elasticjob.io/docs/elastic-job-lite/02-guide/job-sharding-strategy/

```
public class SimpleJobTest {

    public static void main(String[] args) {
        // ZK 注册中心
        CoordinatorRegistryCenter regCenter = new ZookeeperRegistryCenter(new
        ZookeeperConfiguration("localhost:2181", "elastic-job-demo"));
        regCenter.init();

        // 定义作业核心配置
        JobCoreConfiguration simpleCoreConfig = JobCoreConfiguration.newBuilder("MyElasticJob", "0/2 * * * * ?",
        1).build();
        // 定义 SIMPLE 类型配置
        SimpleJobConfiguration simpleJobConfig = new SimpleJobConfiguration(simpleCoreConfig,
        MyElasticJob.class.getCanonicalName());

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

        // 构建 Job
```

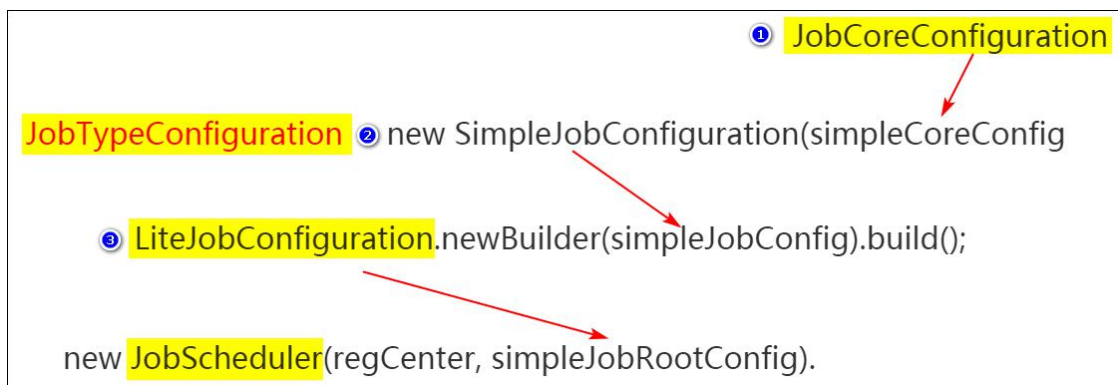


```

new JobScheduler(regCenter, simpleJobRootConfig).init();
}
}

```

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。

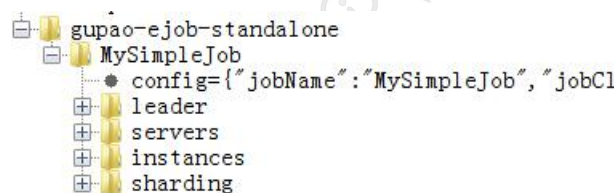
E-Job 使用 ZK 来做分布式协调，所有的配置都会写入到 ZK 节点。

2.3.2 ZK 注册中心数据结构

一个任务一个二级节点。

这里面有些节点是临时节点，只有任务运行的时候才能看到。

注意：修改了任务重新运行任务不生效，是因为 ZK 的信息不会更新，除非把 overwrite 修改成 true。



config 节点

JSON 格式存储。

存储任务的配置信息，包含执行类，cron 表达式，分片算法类，分片数量，分片参数等等。

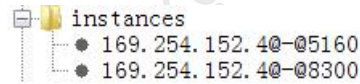
```
{
  "jobName": "MySimpleJob",
  "jobClass": "job.MySimpleJob",
  "jobType": "SIMPLE",
  "cron": "0/2 * * * * ?",
  "shardingTotalCount": 1,
  "shardingItemParameters": "",
  "jobParameter": "",
  "failover": false,
  "misfire": true,
  "description": "",
  "jobProperties": {
    "job_exception_handler": "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
}
```

config 节点的数据是通过 ConfigService 持久化到 zookeeper 中去的。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，除非你在 Lite 级别的配置把参数 overwrite 修改成 true。

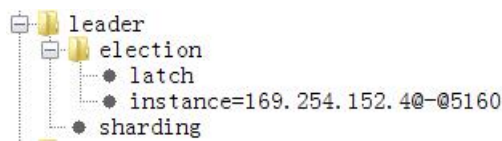
```
LiteJobConfiguration simpleJobRootConfig =
  LiteJobConfiguration.newBuilder(simpleJobConfig).overwrite(true).build();
```

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。只有在运行的时候能看到。



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。在 elastic job 中，任务的执行可以分布在不同的实例（节点）中，但任务分片等核心控制，需要由主节点完成。因此，任务执行前，需要选举出主节点。

下面有三个子节点：

election：主节点选举

sharding：分片

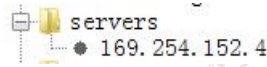
failover：失效转移

election 下面的 instance 节点显示了当前主节点的实例 ID：jobInstanceId。

election 下面的 latch 节点也是一个永久节点用于选举时候的实现分布式锁。

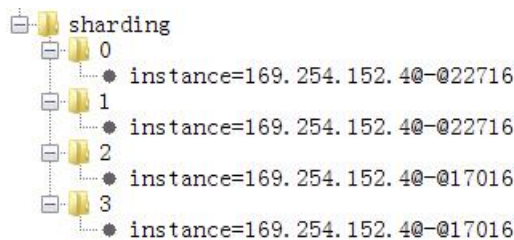
sharding 节点下面有一个临时节点，necessary，是否需要重新分片的标记。如果分片总数变化，或任务实例节点上下线或启用/禁用，以及主节点选举，都会触发设置重分片标记，主节点会进行分片计算。

servers 节点



任务实例的信息，主要是 IP 地址，任务实例的 IP 地址。跟 instances 不同，如果多个任务实例在同一台机器上运行则只会出现一个 IP 子节点。可在 IP 地址节点写入 DISABLED 表示该任务实例禁用。

sharding 节点



任务的分片信息，子节点是分片项序号，从 0 开始。分片个数是在任务配置中设置的。分片项序号的子节点存储详细信息。每个分片项下的子节点用于控制和记录分片运行状态。最主要的子节点就是 instance。

子节点名	是否临时节点	描述
instance	否	执行该分片项的作业运行实例主键
running	是	分片项正在运行的状态 仅配置 monitorExecution 时有效
failover	是	如果该分片项被失效转移分配给其他作业服务器，则此节点值记录执行此分片的作业服务器 IP
misfire	否	是否开启错过任务重新执行
disabled	否	是否禁用此分片项

3 运维平台

3.1 下载解压运行

git 下载源码 <https://github.com/elasticjob/elastic-job-lite>

对 elastic-job-lite-console 打包得到安装包（网盘已提供现成的 console 包）。

解压缩 elastic-job-lite-console- $\{version\}$.tar.gz 并执行 bin\start.sh（Windows 运行.bat）。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。

默认管理员用户名和密码是 root/root。右上角可以切换语言。

3.2 添加 ZK 注册中心

第一步，添加注册中心，输入 ZK 地址和命名空间，并连接。

注册中心名称	注册中心地址	命名空间
standalone	localhost:2181	gupao-ejob-standalone

运维平台和 elastic-job-lite 并无直接关系，是通过读取作业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。

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动，因为控制台和作业本身服务器是完全分离的，控制台并不能控制作业服务器。

可以对作业进行操作。

状态	操作
正常	修改 详情 触发 失效 终止

修改作业

作业名称	MySimpleJob	作业类型	SIMPLE		
作业实现类	simple.MySimpleJob				
Cron表达式	0/2 * * * * ?	作业分片总数	2	自定义参数	
最大容忍本机与注册中心的时间误差秒数	-1	监听作业端口	-1	作业服务器状态修复周期	10
监控作业执行时状态	<input checked="" type="checkbox"/>	支持自动失效转移	<input type="checkbox"/>	支持错过重执行	<input checked="" type="checkbox"/>
分片序号/参数对照表					
作业分片策略实现类全路径					
定制异常处理类全路径	com.dangdang.ddframe.job.executor.handler.impl.DefaultJobExceptionHandler				
定制线程池全路径	com.dangdang.ddframe.job.executor.handler.impl.DefaultExecutorServiceHandler				

3.3 事件追踪

<http://elasticjob.io/docs/elastic-job-lite/02-guide/event-trace/>

Elastic-Job 提供了事件追踪功能 , 可通过事件订阅的方式处理调度过程的重要事件 , 用于查询、统计和监控。

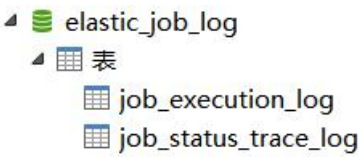
Elastic-Job-Lite 在配置中提供了 JobEventConfiguration , 目前支持数据库方式配置。

ejob-standalone : simple.SimpleJobTest

```
BasicDataSource dataSource = new BasicDataSource();
dataSource.setDriverClassName("com.mysql.jdbc.Driver");
dataSource.setUrl("jdbc:mysql://localhost:3306/elastic_job_log");
dataSource.setUsername("root");
dataSource.setPassword("123456");
JobEventConfiguration jobEventConfig = new JobEventRdbConfiguration(dataSource);
.....
new JobScheduler(regCenter, simpleJobRootConfig, jobEventConfig).init();
```

事件追踪的 event_trace_rdb_url 属性对应库自动创建 JOB_EXECUTION_LOG 和

JOB_STATUS_TRACE_LOG 两张表以及若干索引。



需要在运维平台中添加数据源信息，并且连接：

全局配置11

注册中心配置

事件追踪数据源配置

作业操作00

作业历史<

事件追踪数据源配置

事件追踪数据源名称	事件追踪数据源驱动
elastic_job_log	com.mysql.jdbc.Driver

在作业历史中查询：

历史轨迹

作业名称

服务器IP

开始时间

完成时间

作业名称	服务器IP	分片项	执行结果
MySimpleJob	169.254.152.4	3	成功
MySimpleJob	169.254.152.4	1	成功

4 Spring 集成与分片详解

ejob-springboot 工程

4.1 pom 依赖

```
<properties>
<elastic-job.version>2.1.5</elastic-job.version>
</properties>
<dependency>
  <groupId>com.dangdang</groupId>
  <artifactId>elastic-job-lite-core</artifactId>
  <version>${elastic-job.version}</version>
</dependency>
<!-- elastic-job-lite-spring -->
<dependency>
  <groupId>com.dangdang</groupId>
  <artifactId>elastic-job-lite-spring</artifactId>
  <version>${elastic-job.version}</version>
</dependency>
```

4.2 application.properties

定义配置类和任务类中要用到的参数

```
server.port=${random.int[10000,19999]}
regCenter.serverList = localhost:2181
regCenter.namespace = gupao-ejob-springboot

gupaoJob.cron = 0/3 * * * * ?
gupaoJob.shardingTotalCount = 2
gupaoJob.shardingItemParameters = 0=0,1=1
```

4.3 创建任务

创建任务类，加上@Component 注解

```
@Component
public class SimpleJobDemo implements SimpleJob {
  public void execute(ShardingContext shardingContext) {
    System.out.println(String.format("-----Thread ID: %s, %s,任务总片数: %s, " +
      "当前分片项: %s.当前参数: %s," +
      "当前任务名称: %s.当前任务参数 %s",
      Thread.currentThread().getId(),
```

```

        new SimpleDateFormat("HH:mm:ss").format(new Date()),
        shardingContext.getShardingTotalCount(),
        shardingContext.getShardingItem(),
        shardingContext.getShardingParameter(),
        shardingContext.getJobName(),
        shardingContext.getJobParameter()
    ));
}
}

```

4.4 注册中心配置

Bean 的 `initMethod` 属性用来指定 Bean 初始化完成之后要执行的方法，用来替代继承 `InitializingBean` 接口，以便在容器启动的时候创建注册中心。

```

@Configuration
public class ElasticRegCenterConfig {
    @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));
    }
}

```

4.5 作业三级配置

Core——Type——Lite

```
return LiteJobConfiguration.newBuilder(new SimpleJobConfiguration(JobCoreConfiguration.newBuilder(
```

```

@Configuration
public class ElasticJobConfig {
    @Autowired
    private ZookeeperRegistryCenter regCenter;

    @Bean(initMethod = "init")
    public JobScheduler simpleJobScheduler(final SimpleJobDemo simpleJob,
        @Value("${gupaoJob.cron}") final String cron,
        @Value("${gupaoJob.shardingTotalCount}") final int shardingTotalCount,

```

```

@Value("${gupaoJob.shardingItemParameters}") final String
shardingItemParameters) {
    return new SpringJobScheduler(simpleJob, regCenter,
        getLiteJobConfiguration(simpleJob.getClass(), cron, shardingTotalCount, shardingItemParameters));
}

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();
}
}

```

4.6 作业运行

先把 application.properties 中的分片数全部改成 1

启动 com.gupaoedu.EjobApp 的 main 方法

4.7 分片策略

4.7.1 分片项与分片参数

任务分片，是为了实现把一个任务拆分成多个子任务，在不同的 ejob 示例上执行。

例如 100W 条数据，在配置文件中指定分成 10 个子任务（分片项），这 10 个子任务再按照一定的规则分配到 5 个实际运行的服务器上执行。除了直接用分片项 ShardingItem 获取分片任务之外，还可以用 item 对应的 parameter 获取任务。

standalone 工程：simple.SimpleJobTest

```

JobCoreConfiguration coreConfig = JobCoreConfiguration.newBuilder("MySimpleJob", "0/2 * * * * ?",
4).shardingItemParameters("0=RDP, 1=CORE, 2=SIMS, 3=ECIF").build();

```

springboot 工程，在 application.properties 中定义。

定义几个分片项，一个任务就会有几个线程去运行它。

注意：分片个数和分片参数要一一对应。通常把分片项设置得比 E-Job 服务器个数大一些，比如 3 台服务器，分成 9 片，这样如果有服务器宕机，分片还可以相对均匀。

4.7.2 分片验证

为避免运行的任务太多看不清楚运行结果，可以注释在 ElasticJobConfig 中注释 DataFlowJob 和 ScriptJob。SimpleJob 的分片项改成 2。

直接运行 com.gupaoedu.EjobApp。

或者打成 jar 包：mvn package -Dmaven.test.skip=true

Jar 包路径：ejob-springboot\target\ejob-springboot-0.0.1-SNAPSHOT.jar

修改名称为 ejob.jar 放到 D 盘下。

多实例运行（单机）：

java -jar ejob.jar

- 1、多运行一个点，任务不会重跑（两个节点各获得一个分片项）
- 2、关闭一个节点，任务不会漏跑

```

管理: C:\Windows\system32\cmd.exe - java -jar ejob.jar
-----【简单任务】Thread ID: 20, 16:14:42,任务总片数: 2, 当前分片项: 1, 当前参数:
jack,当前任务名称: com.gupaoedu.job.MySimpleJob,当前任务参数 gupao2673
-----【简单任务】Thread ID: 20, 16:14:44,任务总片数: 2, 当前分片项: 1, 当前参数:
jack,当前任务名称: com.gupaoedu.job.MySimpleJob,当前任务参数 gupao2673
-----【简单任务】Thread ID: 20, 16:14:46,任务总片数: 2, 当前分片项: 1, 当前参数:
jack,当前任务名称: com.gupaoedu.job.MySimpleJob,当前任务参数 gupao2673
-----【简单任务】Thread ID: 20, 16:14:48,任务总片数: 2, 当前分片项: 1, 当前参数:
jack,当前任务名称: com.gupaoedu.job.MySimpleJob,当前任务参数 gupao2673
-----【简单任务】Thread ID: 20, 16:14:50,任务总片数: 2, 当前分片项: 1, 当前参数:
jack,当前任务名称: com.gupaoedu.job.MySimpleJob,当前任务参数 gupao2673
-----【简单任务】Thread ID: 20, 16:14:52,任务总片数: 2, 当前分片项: 1, 当前参数:
jack,当前任务名称: com.gupaoedu.job.MySimpleJob,当前任务参数 gupao2673
-----【简单任务】Thread ID: 20, 16:14:54,任务总片数: 2, 当前分片项: 1, 当前参数:
jack,当前任务名称: com.gupaoedu.job.MySimpleJob,当前任务参数 gupao2673
-----【简单任务】Thread ID: 20, 16:14:56,任务总片数: 2, 当前分片项: 1, 当前参数:
jack,当前任务名称: com.gupaoedu.job.MySimpleJob,当前任务参数 gupao2673
-----【简单任务】Thread ID: 20, 16:14:58,任务总片数: 2, 当前分片项: 1, 当前参数:
jack,当前任务名称: com.gupaoedu.job.MySimpleJob,当前任务参数 gupao2673
-----【简单任务】Thread ID: 20, 16:15:00,任务总片数: 2, 当前分片项: 1, 当前参数:
jack,当前任务名称: com.gupaoedu.job.MySimpleJob,当前任务参数 gupao2673
-----【简单任务】Thread ID: 20, 16:15:02,任务总片数: 2, 当前分片项: 1, 当前参数:
jack,当前任务名称: com.gupaoedu.job.MySimpleJob,当前任务参数 gupao2673
-----【简单任务】Thread ID: 20, 16:15:04,任务总片数: 2, 当前分片项: 1, 当前参数:
jack,当前任务名称: com.gupaoedu.job.MySimpleJob,当前任务参数 gupao2673

管理: C:\Windows\system32\cmd.exe - java -jar ejob.jar
-----【简单任务】Thread ID: 20, 16:14:44,任务总片数: 2, 当前分片项: 0, 当前参数:
qingshan,当前任务名称: com.gupaoedu.job.MySimpleJob,当前任务参数 gupao2673
-----【简单任务】Thread ID: 20, 16:14:46,任务总片数: 2, 当前分片项: 0, 当前参数:
qingshan,当前任务名称: com.gupaoedu.job.MySimpleJob,当前任务参数 gupao2673
-----【简单任务】Thread ID: 20, 16:14:48,任务总片数: 2, 当前分片项: 0, 当前参数:
qingshan,当前任务名称: com.gupaoedu.job.MySimpleJob,当前任务参数 gupao2673
-----【简单任务】Thread ID: 20, 16:14:50,任务总片数: 2, 当前分片项: 0, 当前参数:
qingshan,当前任务名称: com.gupaoedu.job.MySimpleJob,当前任务参数 gupao2673
-----【简单任务】Thread ID: 20, 16:14:52,任务总片数: 2, 当前分片项: 0, 当前参数:
qingshan,当前任务名称: com.gupaoedu.job.MySimpleJob,当前任务参数 gupao2673
-----【简单任务】Thread ID: 20, 16:14:54,任务总片数: 2, 当前分片项: 0, 当前参数:
qingshan,当前任务名称: com.gupaoedu.job.MySimpleJob,当前任务参数 gupao2673
-----【简单任务】Thread ID: 20, 16:14:56,任务总片数: 2, 当前分片项: 0, 当前参数:
qingshan,当前任务名称: com.gupaoedu.job.MySimpleJob,当前任务参数 gupao2673
-----【简单任务】Thread ID: 20, 16:14:58,任务总片数: 2, 当前分片项: 0, 当前参数:
qingshan,当前任务名称: com.gupaoedu.job.MySimpleJob,当前任务参数 gupao2673
-----【简单任务】Thread ID: 20, 16:15:00,任务总片数: 2, 当前分片项: 0, 当前参数:
qingshan,当前任务名称: com.gupaoedu.job.MySimpleJob,当前任务参数 gupao2673
-----【简单任务】Thread ID: 20, 16:15:02,任务总片数: 2, 当前分片项: 0, 当前参数:
qingshan,当前任务名称: com.gupaoedu.job.MySimpleJob,当前任务参数 gupao2673
-----【简单任务】Thread ID: 20, 16:15:04,任务总片数: 2, 当前分片项: 0, 当前参数:
qingshan,当前任务名称: com.gupaoedu.job.MySimpleJob,当前任务参数 gupao2673
  
```

4.7.3 分片策略

<http://elasticjob.io/docs/elastic-job-lite/02-guide/job-sharding-strategy/>

分片项如何分配到服务器？这个跟分片策略有关。

策略类	描述	具体规则
AverageAllocationJobShardingStrategy	基于平均分配算法的分片策略，也是默认的分片策略。	<p>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：</p> <ul style="list-style-type: none"> ● 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	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。	<p>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。</p> <ul style="list-style-type: none"> ● 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。 <p>用于不同的作业平均分配负载至不同的服务器。</p>
RotateServerByNameJobShardingStrategy	根据作业名的哈希值对服务器列表进行轮转的分片策略。	
自定义分片策略		实现 JobShardingStrategy 接口并实现 sharding 方法，接口方法参数为作业服务器 IP 列表和分片策略选项，分片策略选项包括作业名称，分片总数以及分片序列号和个性化参数对照表，可以根据需求定制化自己的分片策略。

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=[]

在 Lite 配置中指定分片策略：

```
String jobShardingStrategyClass = AverageAllocationJobShardingStrategy.class.getCanonicalName();
```



```
LiteJobConfiguration simpleJobRootConfig =
LiteJobConfiguration.newBuilder(simpleJobConfig).jobShardingStrategyClass(jobShardingStrategyClass).build();
```

4.7.4 分片方案

获取到分片项 shardingItem 之后，怎么对数据进行分片呢？

1、对业务主键进行取模，获取余数等于分片项的数据

举例：获取到的 sharding item 是 0,1

在 SQL 中加入过滤条件：where mod(id, 4) in (1, 2)。

这种方式的缺点：会导致索引失效，查询数据时会全表扫描。

解决方案：在查询条件中在增加一个索引条件进行过滤。

2、在表中增加一个字段，根据分片数生成一个 mod 值。取模的基数要大于机器数。否则在增加机器后，会导致机器空闲。例如取模基数是 2，而服务器有 5 台，那么有三台服务器永远空闲。而取模基数是 10，生成 10 个 shardingItem，可以分配到 5 台服务器。当然，取模基数也可以调整。

3、如果从业务层面，可以用 ShardingParamter 进行分片。

例如 0=RDP, 1=CORE, 2=SIMS, 3=ECIF

```
List<users> = SELECT * FROM user WHERE status = 0 AND SYSTEM_ID =
'RDP' limit 0, 100。
```

在 Spring Boot 中要 Elastic-Job 要配置的内容太多了，有没有更简单的添加任务的方法呢？比如在类上添加一个注解？这个时候我们就要用到 starter 了。

4.8 e-job starter

Git 上有一个现成的实现

<https://github.com/TFdream/elasticjob-spring-boot-starter>

工程：elasticjob-spring-boot-starter

需求（一个 starter 应该有什么样子）：

需求	实现	作用
可以在启动类上使用 <code>@Enable</code> 功能开启 E-Job 任务调度	注解 <code>@EnableElasticJob</code>	在自动配置类上用 <code>@ConditionalOnBean</code> 决定是否自动配置
可以在 properties 或 yml 中识别配置内容	配置类 <code>RegCenterProperties.java</code>	支持在 properties 文件中使用 <code>elasticjob.regCenter</code> 前缀，配置注册中心参数
在类上加上注解，直接创建任务	注解 <code>@JobScheduled</code>	配置任务参数，包括定分片项、分片参数等等
不用创建 ZK 注册中心	自动配置类 <code>RegCentreAutoConfiguration.java</code>	注入从 <code>RegCenterProperties.java</code> 读取到的参数，自动创 <code>ZookeeperConfiguration</code>
不用创建三级（Core、Type、Lite）配置	自动配置类 <code>JobAutoConfiguration.java</code>	读取注解的参数，创建 <code>JobCoreConfiguration</code> 、 <code>JobTypeConfiguration</code> 、 <code>LiteJobConfiguration</code> 在注册中心创建之后再创建
Spring Boot 启动时自动配置	创建 <code>Resource/META-INF/spring.factories</code>	指定两个自动配置类

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
io.dreamstudio.elasticjob.autoconfigure.RegCentreAutoConfiguration,\
io.dreamstudio.elasticjob.autoconfigure.JobAutoConfiguration
```

打包 starter 的工程，引入 starter 的依赖，即可在项目中使用注解开启任务调度功能。

5 E-Job 原理

5.1 启动

standalone 工程

```
new JobScheduler(regCenter, simpleJobRootConfig).init();
```

init 方法

```
public void init() {
    LiteJobConfiguration liteJobConfigFromRegCenter = schedulerFacade.updateJobConfiguration(liteJobConfig);
    // 设置分片数
    JobRegistry.getInstance().setCurrentShardingTotalCount(liteJobConfigFromRegCenter.getJobName(),
        liteJobConfigFromRegCenter.getTypeConfig().getCoreConfig().getShardingTotalCount());
    // 构建任务，创建调度器
    JobScheduleController jobScheduleController = new JobScheduleController(
        createScheduler(), createJobDetail(liteJobConfigFromRegCenter.getTypeConfig().getJobClass()),
        liteJobConfigFromRegCenter.getJobName());
    // 在 ZK 上注册任务
    JobRegistry.getInstance().registerJob(liteJobConfigFromRegCenter.getJobName(), jobScheduleController, regCenter);
    // 添加任务信息并进行节点选举
    schedulerFacade.registerStartUpInfo(!liteJobConfigFromRegCenter.isDisabled());
    // 启动调度器
    jobScheduleController.scheduleJob(liteJobConfigFromRegCenter.getTypeConfig().getCoreConfig().getCron());
}
```

registerStartUpInfo 方法

```
public void registerStartUpInfo(final boolean enabled) {
    // 启动所有的监听器
    listenerManager.startAllListeners();
    // 节点选举
    leaderService.electLeader();
    // 服务信息持久化（写到 ZK）
    serverService.persistOnline(enabled);
    // 实例信息持久化（写到 ZK）
    instanceService.persistOnline();
    // 重新分片
}
```

```

shardingService.setReshardingFlag();
// 监控信息监听器
monitorService.listen();
// 自诊断修复，使本地节点与 ZK 数据一致
if (!reconcileService.isRunning()) {
    reconcileService.startAsync();
}
}

```

监听器用于监听 ZK 节点信息的变化。

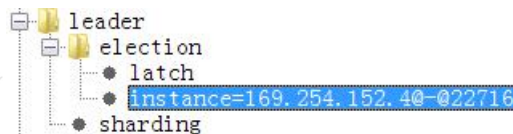
启动的时候进行主节点选举

```

/**
 * 选举主节点.
 */
public void electLeader() {
    log.debug("Elect a new leader now.");
    jobNodeStorage.executeInLeader(LeaderNode.LATCH, new LeaderElectionExecutionCallback());
    log.debug("Leader election completed.");
}

```

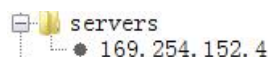
Latch 是一个分布式锁，选举成功后在 instance 写入服务器信息。



// 服务信息持久化（写到 ZK servers 节点）

```
serverService.persistOnline(enabled);
```

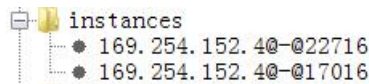
以下是单机运行多个实例：



// 实例信息持久化 (写到 ZK instances 节点)

```
instanceService.persistOnline();
```

运行了两个实例：



5.2 任务执行与分片原理

关注两个问题：

- 1、LiteJob 是怎么被执行的？
- 2、分片项是怎么分配给不同的服务实例的？

在创建 Job 的时候 (createJobDetail)，创建的是实现了 Quartz 的 Job 接口的 LiteJob 类，LiteJob 类实现了 Quartz 的 Job 接口。

在 LiteJob 的 execute 方法中获取对应类型的执行器，调用 execute()方法。

```

public static AbstractElasticJobExecutor getJobExecutor(final ElasticJob elasticJob, final JobFacade jobFacade) {
    if (null == elasticJob) {
        return new ScriptJobExecutor(jobFacade);
    }
    if (elasticJob instanceof SimpleJob) {
        return new SimpleJobExecutor((SimpleJob) elasticJob, jobFacade);
    }
    if (elasticJob instanceof DataflowJob) {
        return new DataflowJobExecutor((DataflowJob) elasticJob, jobFacade);
    }
    throw new JobConfigurationException("Cannot support job type '%s'", elasticJob.getClass().getCanonicalName());
}
  
```

EJOB 提供管理任务执行器的抽象类 AbstractElasticJobExecutor，核心动作在

execute()方法中执行。

```
public final void execute() {
```

调用了另一个 execute()方法，122 行：

```
execute(shardingContexts, JobExecutionEvent.ExecutionSource.NORMAL_TRIGGER);
```

```
private void execute(final ShardingContexts shardingContexts, final JobExecutionEvent.ExecutionSource
executionSource) {
```

在这个 execute 方法中又调用了 process()方法，150 行

```
private void process(final ShardingContexts shardingContexts, final JobExecutionEvent.ExecutionSource
executionSource) {
    Collection<Integer> items = shardingContexts.getShardingItemParameters().keySet();
    // 只有一个分片项时，直接执行
    if (1 == items.size()) {
        int item = shardingContexts.getShardingItemParameters().keySet().iterator().next();
        JobExecutionEvent jobExecutionEvent = new JobExecutionEvent(shardingContexts.getTaskId(), jobName,
executionSource, item);
        process(shardingContexts, item, jobExecutionEvent);
        return;
    }
    final CountDownLatch latch = new CountDownLatch(items.size());
    // 本节点遍历执行相应的分片信息
    for (final int each : items) {
        final JobExecutionEvent jobExecutionEvent = new JobExecutionEvent(shardingContexts.getTaskId(), jobName,
executionSource, each);
        if (executorService.isShutdown()) {
            return;
        }
        executorService.submit(new Runnable() {

            @Override
            public void run() {
                try {
```

```

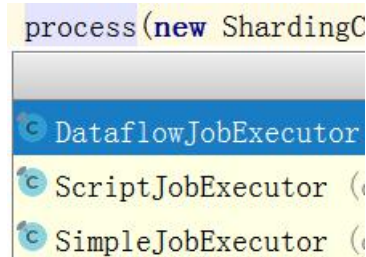
        process(shardingContexts, each, jobExecutionEvent);
    } finally {
        latch.countDown();
    }
}
});
}
try {
    // 等待所有的分片项任务执行完毕
    latch.await();
} catch (final InterruptedException ex) {
    Thread.currentThread().interrupt();
}
}

```

又调用了另一个 process() 方法，206 行

```
protected abstract void process(ShardingContext shardingContext);
```

交给具体的实现类（SimpleJobExecutor、DataflowJobExecutor、ScriptJobExecutor）去处理。



```
process(new ShardingC
DataflowJobExecutor
ScriptJobExecutor (
SimpleJobExecutor (
```

最终调用到任务类

```

@Override
protected void process(final ShardingContext shardingContext) {
    simpleJob.execute(shardingContext);
}

```

5.3 失效转移

所谓失效转移，就是在执行任务的过程中发生异常时，这个分片任务可以在其他节

点再次执行。

simple.SimpleJobTest , failover 方法 :

```
// 设置失效转移
JobCoreConfiguration coreConfig = JobCoreConfiguration.newBuilder("MySimpleJob", "0/2 * * * * ?",
4).shardingItemParameters("0=RDP, 1=CORE, 2=SIMS, 3=ECIF").failover(true).build();
```

FailoverListenerManager 监听的是 zk 的 instance 节点删除事件。如果任务配置了 failover 等于 true , 其中某个 instance 与 zk 失去联系或被删除 , 并且失效的节点又不是本身 , 就会触发失效转移逻辑。

Job 的失效转移监听来源于 FailoverListenerManager 中内部类 JobCrashedJobListener 的 dataChanged 方法。

当节点任务失效时会调用 JobCrashedJobListener 监听器 , 此监听器会根据实例 id 获取所有的分片 , 然后调用 FailoverService 的 setCrashedFailoverFlag 方法 , 将每个分片 id 写到/jobName/leader/failover/items 下 , 例如原来的实例负责 1、2 分片项 , 那么 items 节点就会写入 1、2 , 代表这两个分片项需要失效转移。

```
protected void dataChanged(final String path, final Type eventType, final String data) {
    if (isFailoverEnabled() && Type.NODE_REMOVED == eventType && instanceNode.isInstancePath(path)) {
        String jobId = path.substring(instanceNode.getInstanceFullPath().length() + 1);
        if (jobId.equals(JobRegistry.getInstance().getJobInstance(jobName).getJobInstanceId())) {
            return;
        }
        List<Integer> failoverItems = failoverService.getFailoverItems(jobId);
        if (!failoverItems.isEmpty()) {
            for (int each : failoverItems) {
                // 设置失效的分片项标记
                failoverService.setCrashedFailoverFlag(each);
                failoverService.failoverIfNecessary();
            }
        } else {
            for (int each : shardingService.getShardingItems(jobId)) {
                failoverService.setCrashedFailoverFlag(each);
            }
        }
    }
}
```

```

        failoverService.failoverIfNecessary();
    }
}
}
}

```

然后接下来调用 FailoverService 的 failoverIfNecessary 方法，首先判断是否需要失败转移，如果可以需要则只需作业失败转移。

```

public void failoverIfNecessary() {
    if (needFailover()) {
        jobNodeStorage.executeInLeader(FailoverNode.LATCH, new FailoverLeaderExecutionCallback());
    }
}

```

条件一：\${JOB_NAME}/leader/failover/items/\${ITEM_ID} 有失效转移的作业分片项。

条件二：当前作业不在运行中。

```

private boolean needFailover() {
    return jobNodeStorage.isJobNodeExisted(FailoverNode.ITEMS_ROOT)
    && !jobNodeStorage.getJobNodeChildrenKeys(FailoverNode.ITEMS_ROOT).isEmpty()
    && !JobRegistry.getInstance().isJobRunning(jobName);
}

```

在主节点执行操作

```

public void executeInLeader(final String latchNode, final LeaderExecutionCallback callback) {
    try (LeaderLatch latch = new LeaderLatch(getClient(), jobNodePath.getFullPath(latchNode))) {
        latch.start();
        latch.await();
        callback.execute();
        //CHECKSTYLE:OFF
    } catch (final Exception ex) {
        //CHECKSTYLE:ON
        handleException(ex);
    }
}

```



```
}
}
```

- 1、再次判断是否需要失效转移；
- 2、从注册中心获得一个 `\${JOB_NAME}/leader/failover/items/\${ITEM_ID}` 作业分片项；
- 3、在注册中心节点 `\${JOB_NAME}/sharding/\${ITEM_ID}/failover` 注册作业分片项为当前作业节点；
- 4、然后移除任务转移分片项；
- 5、最后调用执行，提交任务。

```
class FailoverLeaderExecutionCallback implements LeaderExecutionCallback {
    @Override
    public void execute() {
        // 判断是否需要失效转移
        if (JobRegistry.getInstance().isShutdown(jobName) || !needFailover()) {
            return;
        }
        // 从`${JOB_NAME}/leader/failover/items/${ITEM_ID}`获得一个分片项
        int crashedItem =
Integer.parseInt(jobNodeStorage.getJobNodeChildrenKeys(FailoverNode.ITEMS_ROOT).get(0));
        log.debug("Failover job '{}' begin, crashed item '{}'", jobName, crashedItem);

        // 在注册中心节点`${JOB_NAME}/sharding/${ITEM_ID}/failover`注册作业分片项为当前作业节点
        jobNodeStorage.fillEphemeralJobNode(FailoverNode.getExecutionFailoverNode(crashedItem),
JobRegistry.getInstance().getJobInstance(jobName).getJobInstanceId());
        // 移除任务转移分片项
        jobNodeStorage.removeJobNodeIfExisted(FailoverNode.getItemsNode(crashedItem));
        JobScheduleController jobScheduleController = JobRegistry.getInstance().getJobScheduleController(jobName);
        if (null != jobScheduleController) {
            // 提交任务
            jobScheduleController.triggerJob();
        }
    }
}
```

这里仅仅是触发作业，而不是立即执行。

作者：咕泡学院-青山

最后更新时间：2019 年 10 月 28 日