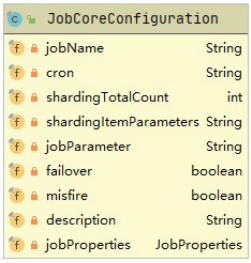
一、Job 配置.....1
二、JobScheduler 初始化流程.....2
三、JobScheduler 调度流程.....3
四、AbstractElasticJobExecutor 调度流程.....4
五、任务执行事件上报流程.....7
六、Failover 机制.....8
七、控制台常用操作.....9
八、ZK 数据结构.....11

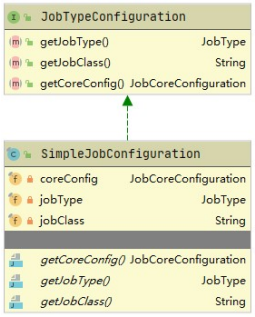
一、Job 配置

JobCoreConfiguration



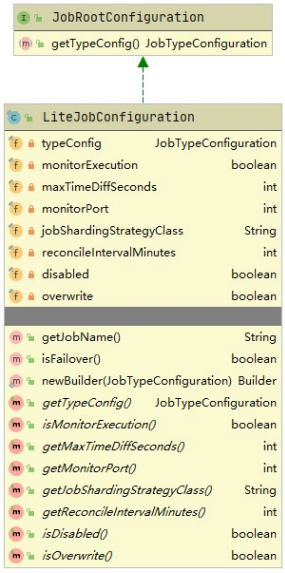
- jobName: String
- cron: String
- shardingTotalCount: int
- shardingItemParameter: String, 0=a,1=b,2=c
- jobParameter: String, 作业自定义参数
- failover: boolean
- misfire: boolean
- description: String
- jobProperties: JobProperties, job_exception_handler(作业异常处理器)与 executor_service_handler(线程池服务处理器)

SimpleJobConfiguration:



- coreConfig: JobCoreConfiguration
- jobType: JobType, SIMPLE, DATAFLOW, SCRIPT
- jobClass: String

LiteJobConfiguration：



- typeConfig: JobTypeConfiguration
- monitorExecution: boolean, 监控作业运行状态
- maxTimeDiffSeconds: int, 最大允许的本机与注册中心的时间误差秒数
- monitorPort: int, 作业监控端口用于 dump 作业信息
- jobShardingStrategyClass: String
- reconcileIntervalMinutes: int

disabled: boolean

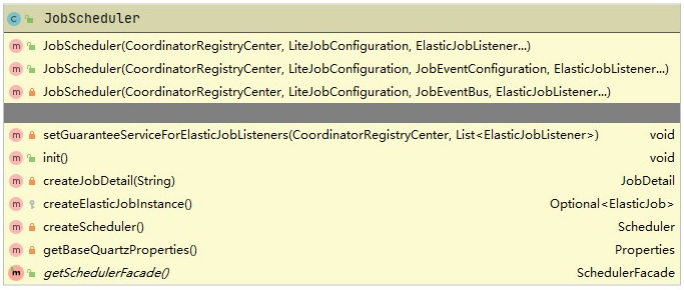
overwrite: boolean

LiteJobConfiguration[SimpleJobConfiguration[JobCoreConfiguration]]

LiteJobConfiguration[DataflowJobConfiguration[JobCoreConfiguration]]

LiteJobConfiguration[ScriptJobConfiguration[JobCoreConfiguration]]

二、JobScheduler 初始化流程



1、添加作业至本地注册表，存于内存

```
JobRegistry.getInstance().addJobInstance(liteJobConfig.getJobName(), new JobInstance());
```

2、初始化 GuaranteeService。提供如下操作：

registerStart: 依次注册各个分片至 guarantee/started/{shardingItemNumber}

isAllStarted: 判断是否所有分片均已启动，首先判断 guarantee/started 节点是否存在，再统计 guarantee/started 的子节点数量是否等于 shardingTotalCount

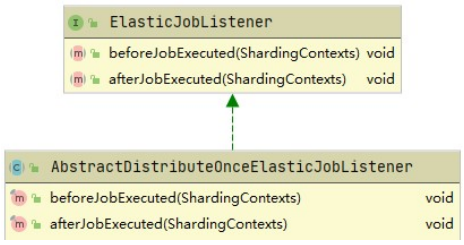
clearAllStartedInfo: 删除 guarantee/started 节点

registerComplete: 依次注册各个分片至 guarantee/completed/{shardingItemNumber}

isAllCompleted: 判断是否所有分片均已执行完成，首先判断 guarantee/completed 节点是否存在，再统计 guarantee/completed 的子节点数量是否等于 shardingTotalCount

clearAllCompletedInfo: 删除 guarantee/completed 节点

3、将 GuaranteeService 绑定至 AbstractDistributeOnceElasticJobListener，仅单一节点执行此监听器



beforeJobExecuted --> registerStart --> isAllStarted ? --> doBeforeJobExecutedAtLastStarted

afterJobExecuted --> registerComplete --> isAllCompleted ? --> doAfterJobExecutedAtLastCompleted

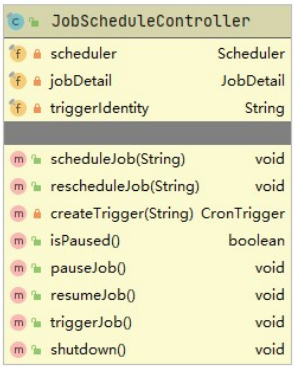
4、若/config 节点不存在或开启了 overwrite 设置，则持久化本地 LiteJobConfiguration 至

/config 节点，

5、更新注册表的任务分片数

```
JobRegistry.getInstance().setCurrentShardingTotalCount(liteJobConfigFromRegCenter.getJobName(),
```

6、使用 Quartz Scheduler 与 JobDetail 初始化 JobScheduleController



7、依次启动所有 Listener

```
/**
 * 开启所有监听器.
 */
public void startAllListeners() {
    electionListenerManager.start();
    shardingListenerManager.start();
    failoverListenerManager.start();
    monitorExecutionListenerManager.start();
    shutdownListenerManager.start();
    triggerListenerManager.start();
    rescheduleListenerManager.start();
    guaranteeListenerManager.start();
    jobNodeStorage.addConnectionStateListener(regCenterConnectionStateListener);
}
```

8、LeaderService 选主

9、若配置 disable 为 true，/servers/{ip}节点写入 DISABLED，表示作业未启用

10、创建临时节点/instances/{ip}@-@{pid}

11、创建/leader/sharding/necessary 节点，表示需要重新分配分片

12、根据配置 monitorExecution 选择是否开启 MonitorService

13、开启 reconcileService，固定间隔执行一次，若当前为主节点，且存在掉线节点，则创建/leader/sharding/necessary 节点，表示需要重新分配分片

14、根据 cron 配置创建 Quartz Trigger，开始执行 Job

三、JobScheduler 调度流程

1、创建 Quartz JobDetail

```
JobDetail result = JobBuilder.newJob(LiteJob.class).withIdentity(liteJobConfig.getJobName()).build();
```

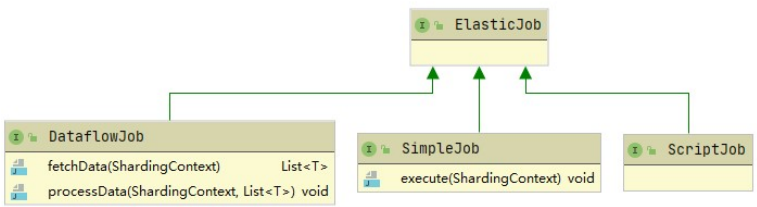
并往 LiteJob 中写入 ElasticJob 实例与 JobFacade。JobFacade 主要提供针对 zk job 和 sharding 相关节点的操作。LiteJob 作为主任务，负责调度用户自定义任务 elasticJob。

```
public final class LiteJob implements Job {

    @Setter
    private ElasticJob elasticJob;

    @Setter
    private JobFacade jobFacade;

    @Override
    public void execute(final JobExecutionContext context) throws JobExecutionException {
        JobExecutorFactory.getJobExecutor(elasticJob, jobFacade).execute();
    }
}
```



2、JobScheduleController 使用 Quartz Scheduler 开始作业调度执行

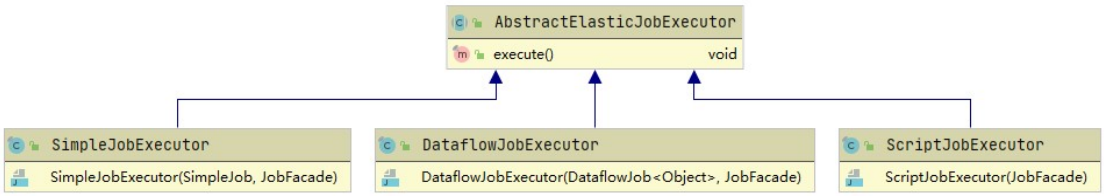
```java
public void scheduleJob(final String cron) {
    try {
        if (!scheduler.checkExists(jobDetail.getKey())) {
            scheduler.scheduleJob(jobDetail, createTrigger(cron));
        }
        scheduler.start();
    } catch (final SchedulerException ex) {
        throw new JobSystemException(ex);
    }
}
```

JobScheduleController 同时提供了 reschedule 操作，可供动态修改调度规则

```java
public synchronized void rescheduleJob(final String cron) {
    try {
        CronTrigger trigger = (CronTrigger) scheduler.getTrigger(TriggerKey.triggerKey(triggerIdentity));
        if (!scheduler.isShutdown() && null != trigger && !cron.equals(trigger.getCronExpression())) {
            scheduler.rescheduleJob(TriggerKey.triggerKey(triggerIdentity), createTrigger(cron));
        }
    } catch (final SchedulerException ex) {
        throw new JobSystemException(ex);
    }
}
```

3、获取不同类型 Job 对应的 JobExecutor，并执行

```java
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());
}
```

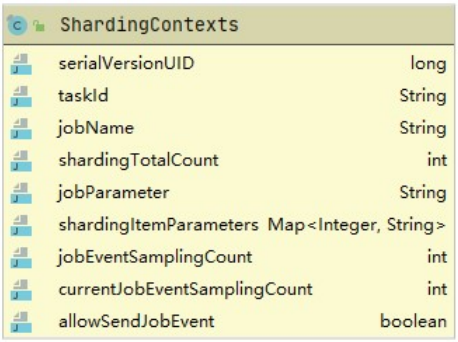


四、AbstractElasticJobExecutor 调度流程

1、使用 JobFacade 检查本机与注册中心的时间误差秒数是否在允许范围。

```java
public void checkMaxTimeDiffSecondsTolerable() throws JobExecutionEnvironmentException {
    int maxTimeDiffSeconds =  load( fromCache: true).getMaxTimeDiffSeconds();
    if (-1  == maxTimeDiffSeconds) {
        return;
    }
    long timeDiff = Math.abs(timeService.getCurrentMillis() - jobNodeStorage.getRegistryCenterTime());
    if (timeDiff > maxTimeDiffSeconds * 1000L) {
        throw new JobExecutionEnvironmentException(
                "Time different between job server and register center exceed '%s' seconds, max time d
    }
}
```

2、使用 JobFacade 获取任务运行上下文，ShardingContexts



JobFacade 在获取 ShardingContexts 前首先进行 failover 操作。首先遍历/sharding 的所有子节点，判断是否存在/sharding/{shardingItem}/failover 节点，若存在，表示此分片需要进行 failover 操作，/sharding/{shardingItem}/failover 节点内存储了执行此 failover 操作的 instanceId。获取与本地 instanceId 匹配的 failover 分片后，构造 ShardingContexts。

ShardingContexts 构造过程见 4.3。

若无需进行 failover 操作，主节点会尝试根据分片策略进行分片操作，见 4.4。

从 zk 获取本机实例对应的所有分片。首先遍历/sharding 节点下的所有子节点，获取/sharding/{shardingItem}/instance 存储的实例 id，获取与本机实例 id 对应的分片。若获取的分片/sharding/{shardingItem}下存在 failover 节点，表示此分片已被失效转移，此分片需要被排除。若获取的分片/sharding/{shardingItem}下存在 disable 节点，表示此分片已被禁用，此分片需要被排除。通过获取的分片构造 ShardingContexts。

3、通过 ExecutionContextService 构造 ShardingContexts

若 job 配置中 monitorExecution 打开，则遍历分片列表，若/sharding/{shardingItem}下存在 running 节点，表示此分片任务还在执行，需要移除此分片。

建立 shardingItem 与 shardingItemParameter 的对应关系，构建 ShardingContexts。

```
return new ShardingContexts(
        buildTaskId(liteJobConfig, shardingItems),
        liteJobConfig.getJobName(),
        liteJobConfig.getTypeConfig().getCoreConfig().getShardingTotalCount(),
        liteJobConfig.getTypeConfig().getCoreConfig().getJobParameter(),
        getAssignedShardingItemParameterMap(shardingItems, shardingItemParameterMap));
```

4、根据分片策略进行分片操作

首先判断是否需要进行任务分片，若存在/leader/sharding/necessary 节点且 zk 中记录的可用实例不为空，则需要进行分片。

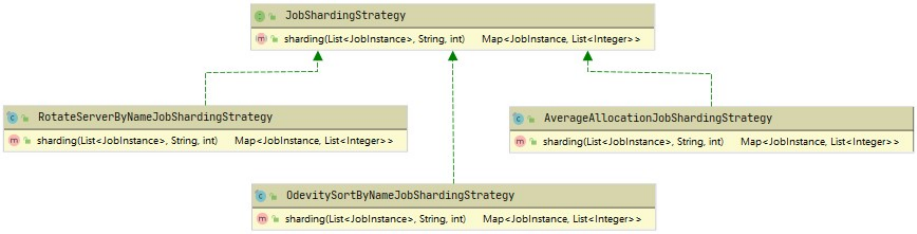
若当前没有主节点，则使用 LeaderLatch 进行主节点选举。主节点负责任务分片，其余节点阻塞直到分片完成，分片完成后会移除/leader/sharding 下的 necessary 节点与 processing 节点。

若 job 配置中 monitorExecution 打开，则主节点还需要等待/sharding 下的所有分片任务执行完成。

主节点开始进行任务分片操作，并在/leader/sharding 下创建 processing 节点，表示分片操作正在进行。接下来主分片会进行正式分片前的清理，清除/sharding/

{shardingItem}下的 instance 节点，若分片前配置已经更改，且新分片数变小，则删除多余的分片/sharding/ {shardingItem}。根据配置 jobShardingStrategyClass 获取相应分片策略执行，并将分片结果写入/sharding/{shardingItem}/instance 节点中，完成后删除/leader/ sharding/necessary 与/leader/sharding/processing 节点。

5、分片策略



AverageAllocationJobShardingStrategy:

对于 0,1,2,3 共三个分片，分配给 3 个实例：[0,3], [1], [2]

OdevitySortByNameJobShardingStrategy:

根据 jobName 哈希值的奇偶，对作业列表进行升降排序，之后应用 AverageAllocationJob ShardingStrategy 策略进行分片。

```
public Map<JobInstance, List<Integer>> sharding(final List<JobInstance> jobInstances, final String jobName, final int shardingTotalCount)
    long jobNameHash = jobName.hashCode();
    if (0 == jobNameHash % 2) {
        Collections.reverse(jobInstances);
    }
    return averageAllocationJobShardingStrategy.sharding(jobInstances, jobName, shardingTotalCount);
```

RotateServerByNameJobShardingStrategy:

通过 jobName 的哈希值对作业列表进行 rotate 操作，之后使用 AverageAllocation JobShardingStrategy 策略进行分片

```
private List<JobInstance> rotateServerList(final List<JobInstance> shardingUnits, final String jobName) {
    int shardingUnitsSize = shardingUnits.size();
    int offset = Math.abs(jobName.hashCode()) % shardingUnitsSize;
    if (0 == offset) {
        return shardingUnits;
    }
    List<JobInstance> result = new ArrayList<>(shardingUnitsSize);
    for (int i = 0; i < shardingUnitsSize; i++) {
        int index = (i + offset) % shardingUnitsSize;
        result.add(shardingUnits.get(index));
    }
    return result;
}
```

若 jobName 哈希值余数为 0，不该变作业列表的顺序，若余数为 1，则原本[0, 1, 2]的顺序变为[1,2,0]。

6、设置 misfire 分片

若任务配置中 monitorExecution 开启，则可监控作业运行状态，若此时仍有分片任务正在执行，即存在/sharding/{shardingItem}/running 节点，则添加 misfire 节点/sharding/{shardingItem}/misfire，表示此分片任务 misfire。此轮任务调度结束。

7、执行任务

若任务配置中 monitorExecution 开启，创建/sharding/{shardingItem}/running 节点，表示分片任务开始执行。此轮任务执行完毕后移除/sharding/{shardingItem}/running 节点，若执

行的是 failover 任务，则移除 sharding/{shardingItem}/failover 节点。

依次执行每个分片任务：

```
private void process(final ShardingContexts shardingContexts, final JobExecutionEvent.ExecutionSource executionSource) {
    Collection<Integer> items = shardingContexts.getShardingItemParameters().keySet();
    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());
    for (final int each : items) {
        final JobExecutionEvent jobExecutionEvent = new JobExecutionEvent(shardingContexts.getTaskId(), jobName, executionSource, each);
        if (executorService.isShutdown()) {
            return;
        }
        executorService.submit((Runnable) () -> {
                try {
                    process(shardingContexts, each, jobExecutionEvent);
                } finally {
                    latch.countDown();
                }
        });
    }
    try {
        latch.await();
    } catch (final InterruptedException ex) {
        Thread.currentThread().interrupt();
    }
}
```

8、执行 misfire 分片任务

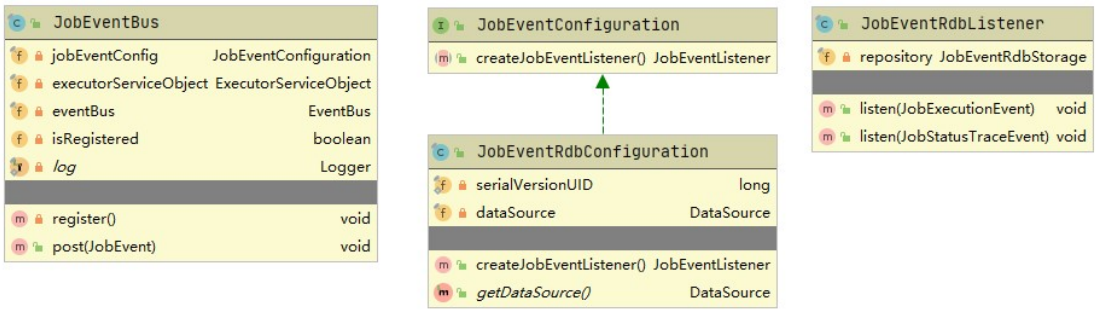
首先删除/sharding/{shardingItem}/misfire 节点，再把分片任务全部执行一次。

```
while (jobFacade.isExecuteMisfired(shardingContexts.getShardingItemParameters().keySet())) {
    jobFacade.clearMisfire(shardingContexts.getShardingItemParameters().keySet());
    execute(shardingContexts, JobExecutionEvent.ExecutionSource.MISFIRE);
}
```

9、执行作业分片失效转移

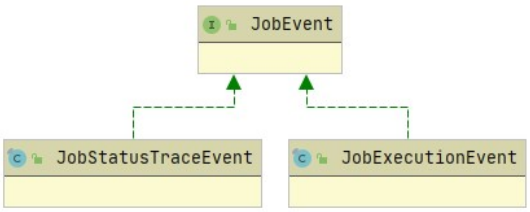
若存在/leader/failover/items 节点，且该节点存在子节点，表示存在失效分片，需要执行分片失效转移。每个实例依次竞争/leader/failover/latch，竞争成功实例取出/leader/failover/ items 下的第一个节点，获取分片编号，并删除该节点。创建/sharding/{shardingItem}/failover 节点，并写入本机实例 id。使用 Quartz scheduler 执行一次任务。

五、任务执行事件上报流程



elasticjob 提供基于数据库的执行事件存储机制。JobEventBus 负责事件分发和监听器注册，事件分发基于 guava AsyncEventBus。JobEventConfiguration 负责创建一个事件监听器，

目前为 JobEventRdbListener，负责往数据库内写入事件记录。目前包含两类事件：JobExecutionEvent 和 JobStatusTraceEvent，前者记录任务执行的最终状态，后者记录任务执行的整个流程。



LiteJobFacade 实例持有 JobEventBus 实例，提供发布事件接口，JobExecutor 在其各个执行点发布对应事件，JobEventRdbListener 负责接收并持久化这些事件。

```
@Override
public void postJobExecutionEvent(final JobExecutionEvent jobExecutionEvent) {
    jobEventBus.post(jobExecutionEvent);
}

@Override
public void postJobStatusTraceEvent(final String taskId, final State state, final String message) {
    TaskContext taskContext = TaskContext.from(taskId);
    jobEventBus.post(new JobStatusTraceEvent(taskContext.getMetaInfo().getJobName(), taskContext.getId(),
            taskContext.getSlaveId(), Source.LITE_EXECUTOR, taskContext.getType(),
            taskContext.getMetaInfo().getShardingItems().toString(), state, message));
    if (!Strings.isNullOrEmpty(message)) {
        log.trace(message);
    }
}
```

六、Failover 机制

FailoverListenerManager 负责处理 failover 事务，注册两个监听器，分别为 JobCrashedJob Listener 与 FailoverSettingsChangedJobListener，前者处理节点离线事件，后者处理 failover 配置变更事件。

1、JobCrashedJobListener，负责监听/instances 下的子节点删除事件。若实例掉线，对应的临时节点/instances/{instanceId}被删除，此 listener 会将实例对应的所有分片添加至/leader/failover/items/ {shardingItem}，表示此分片需要执行 failover 操作。

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

各个实例会竞争对/leader/failover/items 的处理权，竞争成功的实例会获取/leader/failover/items 下的第一个分片，并删除此节点/leader/failover/items/{shardingItem}。同时创建节点/sharding/{shardingItem}/failover，并往此节点中写入本地实例 id，表示此分片的 failover 由本地实例处理。

```
[zk: localhost:2181(CONNECTED) 210] get /elastic-job-example-lite-java/javaSimpleJob/sharding/3/failover
172.23.81.17@-@59364
```

最后会触发一次任务，处理失效分片，见 4.2。

```
public void execute() {
    if (JobRegistry.getInstance().isShutdown(jobName) || !needFailover()) {
        return;
    }
    int crashedItem = Integer.parseInt(jobNodeStorage.getJobNodeChildrenKeys(FailoverNode.ITEMS_ROOT).get(0));
    log.debug("Failover job '{}' begin, crashed item '{}'", jobName, crashedItem);
    jobNodeStorage.fillEphemeralJobNode(FailoverNode.getExecutionFailoverNode(crashedItem),
            JobRegistry.getInstance().getJobInstance(jobName).getJobInstanceId());
    jobNodeStorage.removeJobNodeIfExisted(FailoverNode.getItemsNode(crashedItem));
    // TODO 不应使用triggerJob, 而是使用executor统一调度
    JobScheduleController jobScheduleController = JobRegistry.getInstance().getJobScheduleController(jobName);
    if (null != jobScheduleController) {
        jobScheduleController.triggerJob();
    }
}
```

2、FailoverSettingsChangedJobListener 负责监听/config 节点的内容变更，若配置中 failover 设置被关闭，会依次移除所有 sharding 下的 failover 节点，/sharding/{shardingItem}/failover，关闭 failover 操作。

```
protected void dataChanged(final String path, final Type eventType, final String data) {
    if (configNode.isConfigPath(path) && Type.NODE_UPDATED == eventType && !LiteJobConfigurationGsonFactory.fromJson(data).isFailover()) {
        failoverService.removeFailoverInfo();
    }
}
```

七、控制台常用操作

1、shutdown

shutdown 操作会通过控制台删除/instances 下的所有节点

```
JobNodePath jobNodePath = new JobNodePath(jobName.get());
for (String each : regCenter.getChildrenKeys(jobNodePath.getInstancesNodePath())) {
    regCenter.remove(jobNodePath.getInstanceNodePath(each));
}
```

InstanceShutdownStatusJobListener 监听到节点下线，且下线实例 id 为本地实例，会直接停掉本地的 Quartz 定时任务，若本地实例为主，会删掉节点/leader/election/instance 让出主

```
protected void dataChanged(final String path, final Type eventType, final String data) {
    if (!JobRegistry.getInstance().isShutdown(jobName) && !JobRegistry.getInstance().getJobScheduleController(jobName).isPaused()
            && isRemoveInstance(path, eventType) && !isReconnectedRegistryCenter()) {
        schedulerFacade.shutdownInstance();
    }
}
```

2、disable

disable 操作会通过控制台往/servers 下的所有子节点/servers/{ip}写入 DISABLED 标志，表示此 ip 上的所有实例均被禁用，客户端进行作业调度时，读取到 DISABLED 标志，会停止获取作业分片，跳过此次调度。

```
JobNodePath jobNodePath = new JobNodePath(jobName.get());
for (String each : regCenter.getChildrenKeys(jobNodePath.getServerNodePath())) {
    if (disabled) {
        regCenter.persist(jobNodePath.getServerNodePath(each), "DISABLED");
    } else {
        regCenter.persist(jobNodePath.getServerNodePath(each), "");
    }
}
```

3、trigger

trigger 操作会立即触发一次任务，控制台往/instances 下的所有节点/instances/{id}写入 TRIGGER 标志，

```
JobNodePath jobNodePath = new JobNodePath(jobName.get());
for (String each : regCenter.getChildrenKeys(jobNodePath.getInstancesNodePath())) {
    regCenter.persist(jobNodePath.getInstanceNodePath(each), "TRIGGER");
}
```

JobTriggerStatusJobListener 会监听节点/instances/{id}的更新事件，若更新内容为 TRIGGER 且实例 id 为本地实例 id，则立即触发一次作业调度

```
if (!InstanceOperation.TRIGGER.name().equals(data) || !instanceNode.isLocalInstancePath(path) || Type.NODE_UPDATED != eventType) {
    return;
}
instanceService.clearTriggerFlag();
if (!JobRegistry.getInstance().isShutdown(jobName) && !JobRegistry.getInstance().isJobRunning(jobName)) {
    // TODO 目前是作业运行时不能触发，未来改为堆积式触发
    JobRegistry.getInstance().getJobScheduleController(jobName).triggerJob();
}
```

4、作业配置更新

控制台往/config 节点写入更改后的配置，

```
public void updateJobSettings(final JobSettings jobSettings) {
    Preconditions.checkArgument(!Strings.isNullOrEmpty(jobSettings.getJobName()), errorMessage: "jobName can not be empty.");
    Preconditions.checkArgument(!Strings.isNullOrEmpty(jobSettings.getCron()), errorMessage: "cron can not be empty.");
    Preconditions.checkArgument( expression: jobSettings.getShardingTotalCount() > 0, errorMessage: "shardingTotalCount should larger than zero.");
    JobNodePath jobNodePath = new JobNodePath(jobSettings.getJobName());
    regCenter.update(jobNodePath.getConfigNodePath(), LiteJobConfigurationGsonFactory.toJsonForObject(jobSettings));
}
```

CronSettingAndJobEventChangedJobListener 负责监听/config 节点变化，并重新调度任务

```
protected void dataChanged(final String path, final Type eventType, final String data) {
    if (configNode.isConfigPath(path) && Type.NODE_UPDATED == eventType && !JobRegistry.getInstance().isShutdown(jobName)) {
        JobRegistry.getInstance().getJobScheduleController(jobName).rescheduleJob(
                LiteJobConfigurationGsonFactory.fromJson(data).getTypeConfig().getCoreConfig().getCron());
    }
}
```

八、ZK 数据结构

---**config**
---**instances** (作业运行实例信息，子节点是当前作业运行实例的 id)
-----|{ip}@--@{pid} (临时节点)
-----|{ip}@--@{pid}
---**shardings**
-----|**0**(分片编号)
----------|instance(执行该分片项的作业运行实例 id)
----------|running(临时节点，任务是否正在执行, 仅配置 monitorExecution 时有效)
----------|failover(临时节点，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实例 id)
----------|misfire(表示有任务被错过，需要重新执行)
----------|disabled(表示此分片被禁用)
---**servers**
-----|{ip}
-----|{ip}
---**leader**
-----|**election**
----------|instance(当前主实例 id)
----------|latch(主实例选举分布式锁)
-----|**sharding**
----------|necessary(表示需要重新分片，实例上下线时会开启此标志)
----------|processing(分片正在进行中，分片完成后会删除此节点)
-----|**failover**
----------|**items**
--------------|latch(各实例竞争失效分片时占用的分布式锁)
--------------|0(分片编号，表示此分片已失效，实例竞争到此分片后会删除此节点)
--------------|1(分片编号)