Elastic-job 源码分析

一、eljob 启动过程

1.1 java 方式启动

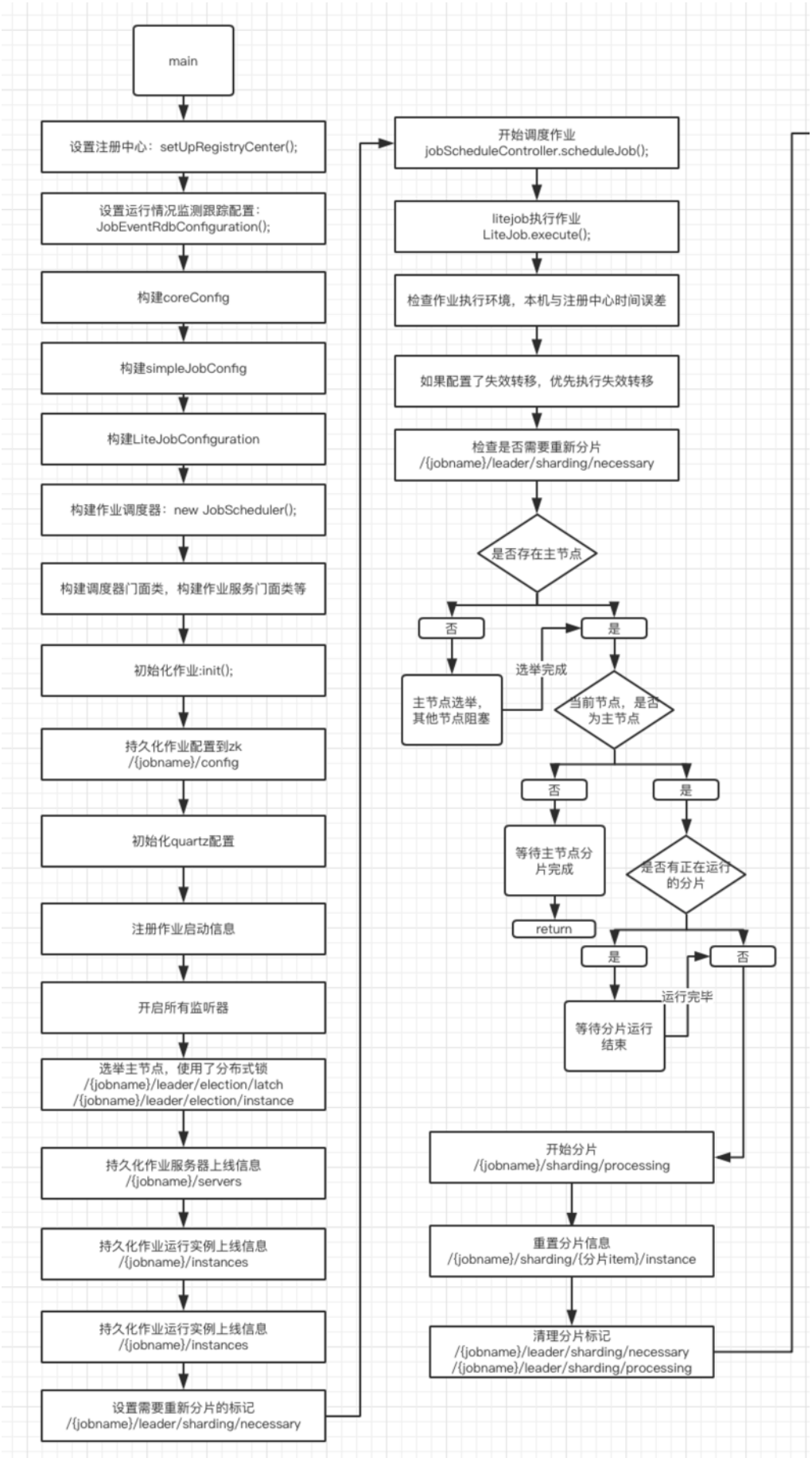
java 方式启动入口：com.dangdang.ddframe.job.example.JavaMain.main();

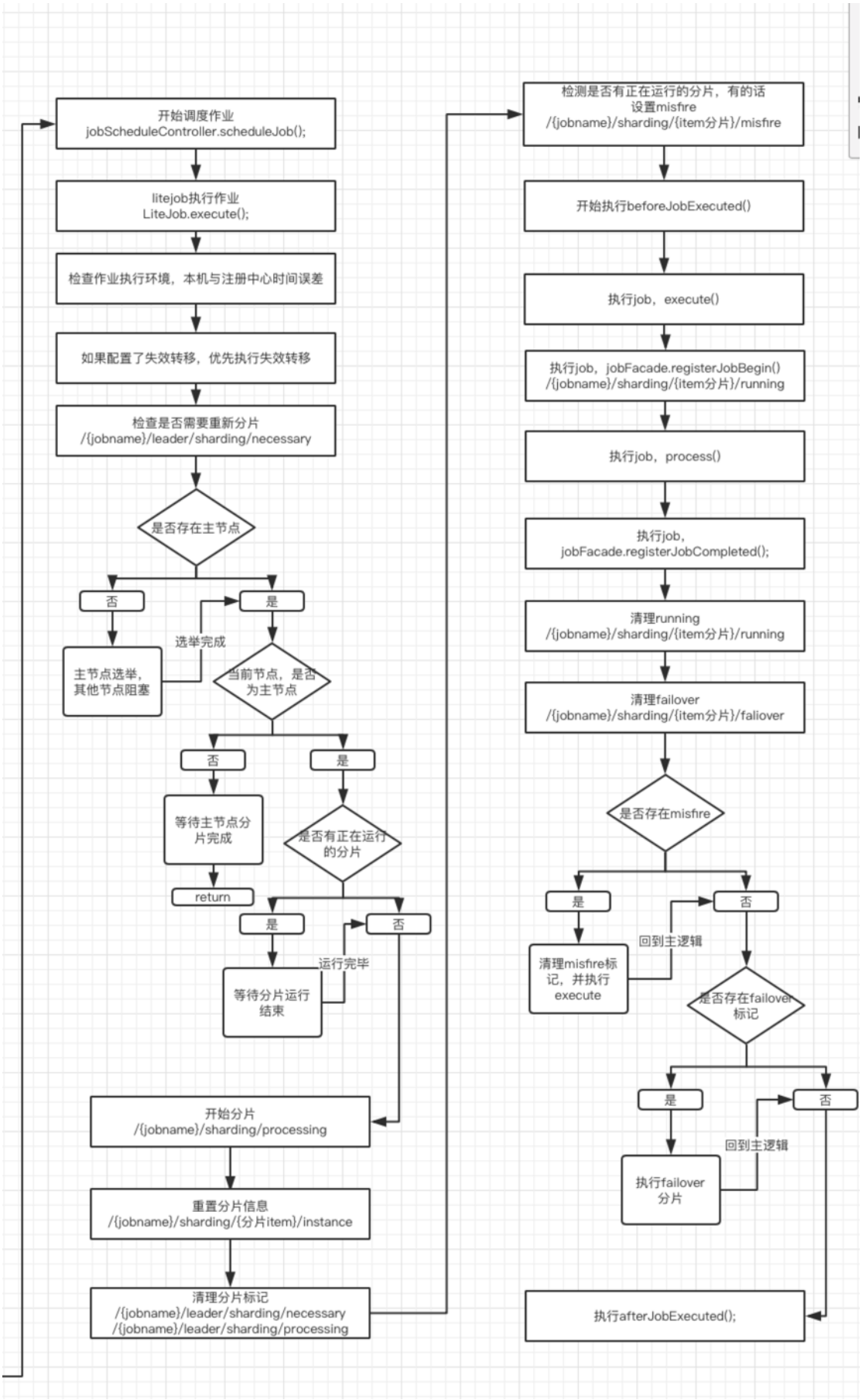
```
1.  public static void main(final String[] args) throws IOException {
2.    CoordinatorRegistryCenter regCenter = setUpRegistryCenter();
3.    JobEventConfiguration jobEventConfig = new JobEventRdbConfiguration(setUpEventTraceDataSource(
    ));
4.    setUpSimpleJob(regCenter, jobEventConfig);
5.  }
```

java 方式和 spring 启动方式，都会走到相同的逻辑

```
1.  private static void setUpSimpleJob(final CoordinatorRegistryCenter regCenter, final JobEventConf
    iguration jobEventConfig) {
2.    //--构造 coreConfig=>simpleJobConfig=>LiteJobConfiguration
3.    JobCoreConfiguration coreConfig = JobCoreConfiguration.newBuilder("javaSimpleJob", "0/2 * * *
    * ?", 1).shardingItemParameters("0=Beijing,1=Shanghai").failover(true).misfire(false).build();
4.    SimpleJobConfiguration simpleJobConfig = new SimpleJobConfiguration(coreConfig, JavaSimpleJob.
    class.getCanonicalName());
5.    new JobScheduler(regCenter, LiteJobConfiguration.newBuilder(simpleJobConfig).build(), jobEvent
    Config).init();
6.  }
```

每个 job 都会产生一个产生一个实例:new JobScheduler().init();
下面来看启动流程图


main
设置注册中心：setUpRegistryCenter();
设置运行情况监测跟踪配置：
JobEventRdbConfiguration();
构建coreConfig
构建simpleJobConfig
构建LiteJobConfiguration
构建作业调度器：new JobScheduler();
构建调度器门面类，构建作业服务门面类等
初始化作业:init();
持久化作业配置到zk
/{jobname}/config
初始化quartz配置
注册作业启动信息
开启所有监听器
选举主节点，使用了分布式锁
/{jobname}/leader/election/latch
/{jobname}/leader/election/instance
持久化作业服务器上线信息
/{jobname}/servers
持久化作业运行实例上线信息
/{jobname}/instances
持久化作业运行实例上线信息
/{jobname}/instances
设置需要重新分片的标记
/{jobname}/leader/sharding/necessary
开始调度作业
jobScheduleController.scheduleJob();
litejob执行作业
LiteJob.execute();
检查作业执行环境，本机与注册中心时间误差
如果配置了失效转移，优先执行失效转移
检查是否需要重新分片
/{jobname}/leader/sharding/necessary
是否存在主节点
否
是
选举完成
主节点选举，
其他节点阻塞
当前节点，是否
为主节点
否
是
等待主节点分
片完成
是否有正在运行
的分片
return
是
否
运行完毕
等待分片运行
结束
开始分片
/{jobname}/sharding/processing
重置分片信息
/{jobname}/sharding/{分片item}/instance
清理分片标记
/{jobname}/leader/sharding/necessary
/{jobname}/leader/sharding/processing


开始调度作业
jobScheduleController.scheduleJob();
litejob执行作业
LiteJob.execute();
检查作业执行环境，本机与注册中心时间误差
如果配置了失效转移，优先执行失效转移
检查是否需要重新分片
/{jobname}/leader/sharding/necessary
是否存在主节点
否
是
选举完成
主节点选举，
其他节点阻塞
当前节点，是否
为主节点
否
是
等待主节点分
片完成
是否有正在运行
的分片
return
是
否
运行完毕
等待分片运行
结束
开始分片
/{jobname}/sharding/processing
重置分片信息
/{jobname}/sharding/{分片item}/instance
清理分片标记
/{jobname}/leader/sharding/necessary
/{jobname}/leader/sharding/processing
检测是否有正在运行的分片，有的话
设置misfire
/{jobname}/sharding/{item分片}/misfire
开始执行beforeJobExecuted()
执行job，execute()
执行job，jobFacade.registerJobBegin()
/{jobname}/sharding/{item分片}/running
执行job，process()
执行job，
jobFacade.registerJobCompleted();
清理running
/{jobname}/sharding/{item分片}/running
清理failover
/{jobname}/sharding/{item分片}/faliover
是否存在misfire
是
否
回到主逻辑
清理misfire标
记，并执行
execute
是否存在failover
标记
是
否
回到主逻辑
执行failover
分片
执行afterJobExecuted();

1.2 spring 方式启动

spring 方式启动入口：com.dangdang.ddframe.job.example.SpringMain.main();

```
1.  public static void main(final String[] args) {
2.      new ClassPathXmlApplicationContext("classpath:META-INF/applicationContext.xml");
3.  }
```

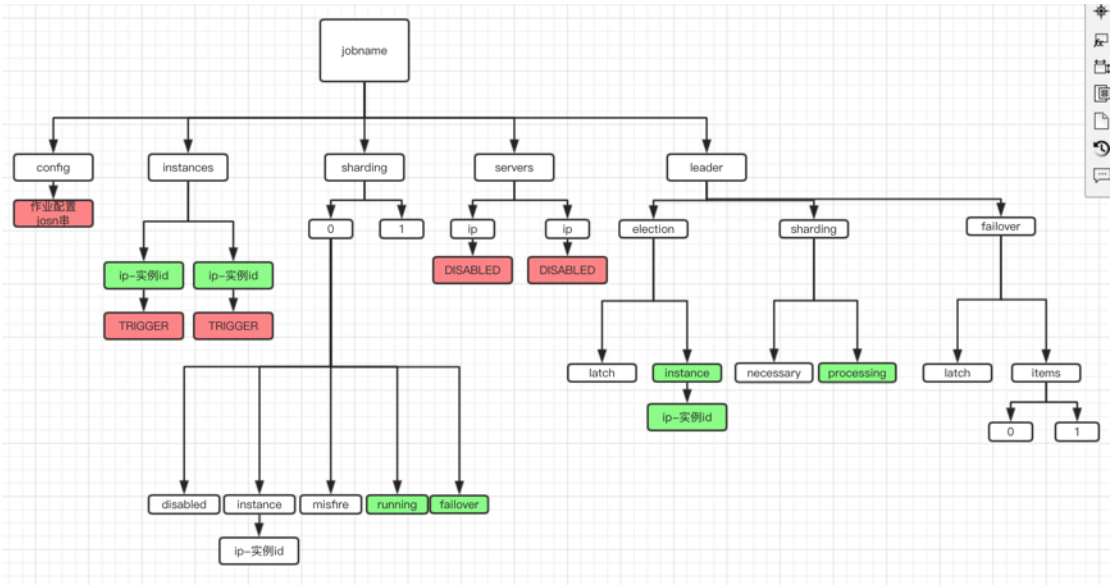
1.3 zk 数据中心存储结构

namespace=elastic-job-example-lite-java

jobname=javaSimpleJob

绿色节点为临时节点

红色节点为数据节点



二、eljob-listener 分析

2.1 listenerManager 介绍

它的作用是 zk 作业注册中心的监听器管理者，所有的 zk 节点发生变化,都由该

listener 下的各个 manager 来负责处理。

listenerManager 位置：

com.dangdang.ddframe.job.lite.internal.listener.ListenerManager

```
1.  public final class ListenerManager {
2.      private final JobNodeStorage jobNodeStorage;
3.      private final ElectionListenerManager electionListenerManager;//--主节点选举监听管理器
4.      private final ShardingListenerManager shardingListenerManager;//--分片监听管理器
5.      private final FailoverListenerManager failoverListenerManager;//--失效转移监听管理
    器 https://blog.csdn.net/prestigeding/article/details/80106418
6.      private final MonitorExecutionListenerManager monitorExecutionListenerManager;//--幂等性监听
    管理器
7.      private final ShutdownListenerManager shutdownListenerManager;//--运行实例关闭监听管理器
8.      private final TriggerListenerManager triggerListenerManager;//--作业触发监听管理器
9.      private final RescheduleListenerManager rescheduleListenerManager;//--重调度监听管理器
10.     private final GuaranteeListenerManager guaranteeListenerManager;//--保证分布式任务全部开始和结
    束状态监听管理器
11.     private final RegistryCenterConnectionStateListener regCenterConnectionStateListener;//--注
    册中心连接状态监听器
12.     public ListenerManager(final CoordinatorRegistryCenter regCenter, final String jobName, fina
    l List<ElasticJobListener> elasticJobListeners) {
13.         jobNodeStorage = new JobNodeStorage(regCenter, jobName);
14.         electionListenerManager = new ElectionListenerManager(regCenter, jobName);
15.         shardingListenerManager = new ShardingListenerManager(regCenter, jobName);
16.         failoverListenerManager = new FailoverListenerManager(regCenter, jobName);
17.         monitorExecutionListenerManager = new MonitorExecutionListenerManager(regCenter, jobName
    );
18.         shutdownListenerManager = new ShutdownListenerManager(regCenter, jobName);
19.         triggerListenerManager = new TriggerListenerManager(regCenter, jobName);
20.         rescheduleListenerManager = new RescheduleListenerManager(regCenter, jobName);
21.         guaranteeListenerManager = new GuaranteeListenerManager(regCenter, jobName, elasticJobLi
    steners);
```

```
22.          regCenterConnectionStateListener = new RegistryCenterConnectionStateListener(regCenter,
    jobName);
23.      }
24.      /**
25.       * 开启所有监听器.
26.       */
27.      public void startAllListeners() {
28.          electionListenerManager.start();
29.          shardingListenerManager.start();
30.          failoverListenerManager.start();
31.          monitorExecutionListenerManager.start();
32.          shutdownListenerManager.start();
33.          triggerListenerManager.start();
34.          rescheduleListenerManager.start();
35.          guaranteeListenerManager.start();
36.          jobNodeStorage.addConnectionStateListener(regCenterConnectionStateListener);
37.      }
38. }
```

2.2 主节点选举监听器

作业上线之后，会产生一台主服务器，其余服务器为从服务器。zk 表示形式如下：

```
[zk: localhost:2181(CONNECTED) 6] ls /elastic-job-example-lite-java/javaSimpleJob/leader/election/instance
[]
[zk: localhost:2181(CONNECTED) 7] get /elastic-job-example-lite-java/javaSimpleJob/leader/election/instance
100.66.157.131@-@73861
```

instance 节点上，会存储主服务器的信息

当主节点信息发生变化的时候，会触发选主监听器。选主监听器上，总计监控了两种类型的节点变化情况，如下

```
1.  public void start() {
2.      //--LeaderNode.INSTANCE 被删除的监听器
3.      addDataListener(new LeaderElectionJobListener());
4.      //--主退位监听器，其目的就是删除 LeaderNode.INSTANCE 节点
```

```
5.      addDataListener(new LeaderAbdicationJobListener());
6.  }
```

LeaderElectionJobListener 的作用是，当节点数据发生变化时，重新进行选举。这里存在两种情况：

```
1.  class LeaderElectionJobListener extends AbstractJobListener {
2.      @Override
3.      protected void dataChanged(final String path, final Type eventType, final String data) {
4.      //--如果该 job 未停止，并且可以进行选主或 LeaderNode.INSTANCE 节点被删除时，触发一次选主。
5.          if (!JobRegistry.getInstance().isShutdown(jobName) && (isActiveElection(path, data) || isP
    assiveElection(path, eventType))) {
6.              leaderService.electLeader();
7.          }
8.      }
9.      //--如果不存在 leader 节点，并且当前服务器运行正常，运行正常的依据是存
    在 /namespace/{jobname}/servers/server-ip，并且节点内容不为 DISABLED。
10.     private boolean isActiveElection(final String path, final String data) {
11.         return !leaderService.hasLeader() && isLocalServerEnabled(path, data);
12.     }
13.     //--如果当前事件节点(path)为 LeaderNode.INSTANCE，并且事件类型(eventType)为删除，并且该 job 的当前对
    应的实例(namespace/namespace/{jobname}/instances/ip)存在并且状态不为 DISABLED。
14.     private boolean isPassiveElection(final String path, final Type eventType) {
15.         return isLeaderCrashed(path, eventType) && serverService.isAvailableServer(JobRegistry.getI
    nstance().getJobInstance(jobName).getIp());
16.     }
17.     private boolean isLeaderCrashed(final String path, final Type eventType) {
18.         return leaderNode.isLeaderInstancePath(path) && Type.NODE_REMOVED == eventType;
19.     }
20.     private boolean isLocalServerEnabled(final String path, final String data) {
21.         return serverNode.isLocalServerPath(path) && !ServerStatus.DISABLED.name().equals(data);
22.     }
23. }
```

2.2.1 主动选举

isActiveElection()

1.当不存在主节点，或者当前作业为开启状态时，触发选举。

2.!leaderService.hasLeader()；表示不存在主节点

3.isLocalServerEnable();判断作业是否开启，这里判断有点特殊，它不是通过作业节点是否处于开启状态，而是判断该数据不是将作业节点更新成禁用状态。那么什么时候会发生这种情况呢？举个例子，当前节点处于禁用状态，再使用运维平台，将节点状态变为启用，则会触发该判断，进行选举。第二种情况，如果当前节点处于启用状态，再使用运维平台禁用，则不会触发该条件。

2.2.2 被动选举

isPassiveElection();

1.当主节点因为各种情况被删除时，则触发该判断条件，进行重新选举。

删除情况有：

1.主节点进程正常关闭。

2.主节点 crashed。

3.作业被禁用。

4.主节点进程远程关闭。

2.判断条件为：1.主节点被删除。2 当前节点正在运行中。满足这两个条件，则可以参加选举。

2.2.3 服务禁用

LeaderAbdicationJobListener

监听/namespace/jobname/server/${ip}节点，如果/namespace/jobname/servers/${ip}节点，在管理界面被禁用，并且判断到禁用的服务器，是主节点。那么就删除 leaderNode.instance。从而再次触发上面的 LeaderElectionJobListener，发起主节点选举。

```
class LeaderAbdicationJobListener extends AbstractJobListener {
    /**
     * --当管理员让节点 disable，如果该节点是本地服务，并且还是 leader，则放弃 leader 权限
     * @param path
     * @param eventType
     * @param data
     */
    @Override
    protected void dataChanged(final String path, final Type eventType, final String data) {
        if (leaderService.isLeader() && isLocalServerDisabled(path, data)) {
          leaderService.removeLeader();
        }
    }
    private boolean isLocalServerDisabled(final String path, final String data) {
        return serverNode.isLocalServerPath(path) && ServerStatus.DISABLED.name().equals(data);
    }
}
```

参考文献：http://www.iocoder.cn/Elastic-Job/election/?sf&2017-11-15
https://blog.csdn.net/prestigeding/article/details/79796978

2.3 分片监听管理器

2.3.1 分片数发生变化

ShardingTotalCountChangedJobListener 的作用是监听 /namespace/{jobname}/config 分片数变化，当 zk 上的 config 下的分片数发生变化之后，会触发该监听器。触发之后，会设置分片标记：/namespace/{jobname}/leader/sharding/necessary

```
1.  class ShardingTotalCountChangedJobListener extends AbstractJobListener {
2.      /**
3.       * //如果配置平台上面发生了配置改变,并且分片总数发生了改变，则更新本地缓存
    (setCurrentShardingTotalCount)，同时添加 leader/sharding/necessary 节点。
4.       * --job 配置的分片总节点数发生变化监听器（ElasticJob 允许通过 Web 界面修改每个任务配置的分片总数
    量）。
5.       job 的配置信息存储在${namespace}/jobname/config 节点上，存储内容为 json 格式的配置信息。
6.       如果${namespace}/jobname/config 节点的内容发生变化，zk 会触发该节点的节点数据变化事件，
7.       如果 zk 中存储的分片节点数量与内存中的分片数量(JobRegistry.getInstance())不相同的话，
8.       调用 ShardingService 设置需要重新分片标记（创建${namespace}/jobname/leader/sharding/necessary
    持久节点）并更新内存中的分片节点总数。
9.      * @param path
10.     * @param eventType
11.     * @param data
12.     */
13.     @Override
14.     protected void dataChanged(final String path, final Type eventType, final String data) {
15.         if (configNode.isConfigPath(path) && 0 != JobRegistry.getInstance().getCurrentShardingTot
    alCount(jobName)) {
16.             int newShardingTotalCount = LiteJobConfigurationGsonFactory.fromJson(data).getTypeCon
    fig().getCoreConfig().getShardingTotalCount();
17.             if (newShardingTotalCount != JobRegistry.getInstance().getCurrentShardingTotalCount(j
    obName)) {
18.                 shardingService.setReshardingFlag();
19.                 JobRegistry.getInstance().setCurrentShardingTotalCount(jobName, newShardingTotalCo
    unt);
20.             }
21.         }
22.   }
23. }
```

2.2.2 作实例发生变化

ListenServersChangedJobListener 的作用是，当有 servers 节点或者 instances 节点数量发生变化之后，会触发该监听器。通常来说，有两种情况，会导致这两个接口发生变化，一种是服务宕机，一种是加入了新的服务。

```
1.  class ListenServersChangedJobListener extends AbstractJobListener {
2.      @Override
3.      protected void dataChanged(final String path, final Type eventType, final String data) {//
    如果服务没有停止，并且是实例或者服务器发生了改变，需要设置重新分片标记
4.          if (!JobRegistry.getInstance().isShutdown(jobName) && (isInstanceChange(eventType, path)
     || isServerChange(path))) {
5.              shardingService.setReshardingFlag();
6.          }
7.      }
8.      private boolean isInstanceChange(final Type eventType, final String path) {
9.          return instanceNode.isInstancePath(path) && Type.NODE_UPDATED != eventType;
10.     }
11.     private boolean isServerChange(final String path) {
12.         return serverNode.isServerPath(path);
13.     }
14. }
```

2.4 失效转移监听器

2.4.1 job 实例节点宕机事件监听器

作业实例上线之后，会在 zk 上注册作业实例信息，下图是两个 job 实例 zk 表现

```
[zk: localhost:2181(CONNECTED) 4] ls /elastic-job-example-lite-java/javaSimpleJob/instances
[169.254.238.237@-@69089, 169.254.238.237@-@69095]
```

当其中一个作业崩溃之后，instanes 下的实例会消失一个，这时候监听器会监听到节点变化通知

```
1.  class JobCrashedJobListener extends AbstractJobListener {
2.      @Override
3.      protected void dataChanged(final String path, final Type eventType, final String data) {
4.      //--如果配置文件中设置开启故障失效转移机制，监听到${namespace}/jobname/instances 节点下子节点的删除事件时，则被认为有节点宕机，将执行故障失效转移相关逻辑。
5.     //示例值:path=/javaSimpleJob/instances/100.66.157.110@-@67665,eventType=NODE_REMOVED
6.     if (isFailoverEnabled() && Type.NODE_REMOVED == eventType && instanceNode.isInstancePath(path)) {
7.         String jobInstanceId = path.substring(instanceNode.getInstanceFullPath().length() + 1);//--获取被宕机的任务实例 ID(jobInstanceId)。
8.         if (jobInstanceId.equals(JobRegistry.getInstance().getJobInstance(jobName).getJobInstanceId())) {//--如果被删除的任务节点 ID 与当前实例的 ID 相同，则忽略。
9.             return;
10.         }
11.         //示例值:jobInstanceId=100.66.157.110@-@67665
12.        List<Integer> failoverItems = failoverService.getFailoverItems(jobInstanceId);//--根据宕机 jobInstanceId 获取作业服务器的失效转移分片项集合。
13.        if (!failoverItems.isEmpty()) {//判断是否有失败分片转移到当前节点，初始状态肯定为空，将执行代码@6，设置故障转移相关准备环境。
14.            for (int each : failoverItems) {
15.                failoverService.setCrashedFailoverFlag(each);
16.                failoverService.failoverIfNecessary();
17.            }
18.         } else {
19.            for (int each : shardingService.getShardingItems(jobInstanceId)) {//--@6 获取分配给 Crashed(宕机的 job 实例)的所有分片节点，遍历已发生故障的分片，
20.            //--将这些分片设置为故障，待故障转移，设置为故障的实现方法为：创建 ${namespace}/jobname/leader/failover/items/{item}。
21.                failoverService.setCrashedFailoverFlag(each);
22.                failoverService.failoverIfNecessary();
23.            }
24.        }
25.     }
```

```
26.     }
27. }
```

dataChange 在收到通知时，会进行如下的处理

1. 判断失效转移策略是否开启
2. 判断事件类型是否为删除事件
3. 判断变化的 path 是否为 instances 的 path，也就是该 listener 要处理的 path
4. 获取崩溃的实例节点 id 信息
5. 优先处理该实例节点 id 上，未处理的实效转移分片项
6. 再者抓取分配给该实例节点的分片项
7. 将要转移的分片项，设置到 namespace/jobname/leader/failover/items/{num} 上，接着这里会使用到一个分布式锁 namespace/jobname/leader/failover/latch，获得分布式锁之后,执行下面的 callback 逻辑
8. 执行失效转移 FailoverLeaderExecutionCallback
9. 创建一个临时节点，把失效转移分片项，设置到 sharding 节点上，namespace/jobname/sharding/{num}/failover
10. 删除 namespace/jobname/leader/failover/items/{num}
11. 手动触发任务 triggerJob
12. 在 job 执行完成之后，会删除 namespace/jobname/sharding/{num}/failover

2.4.2 失效转移配置变化事件监听器

因为 eljob 管理平台，可以手动修改作业配置，所以当失效转移功能，被设置为 false 时，就会触发该监听器

```
1.  class FailoverSettingsChangedJobListener extends AbstractJobListener {
2.      @Override
3.      protected void dataChanged(final String path, final Type eventType, final String data) {
4.          if (configNode.isConfigPath(path) && Type.NODE_UPDATED == eventType && !LiteJobConfigurat
    ionGsonFactory.fromJson(data).isFailover()) {
5.              failoverService.removeFailoverInfo();
6.          }
7.      }
```

```
8. }
```

该监听器会做如下的处理：

1. 得到 namespace/jobname/sharding 下的所有子节点，循环进行删除 failover 项

参考文献：https://blog.csdn.net/prestigeding/article/details/80106418

2.5 幂等性监听器

2.6 运行实例关闭监听器

ShutdownListenerManager 的作用是，监听/namespace/{jobname}/instance/ip-实例 id 的删除事件。

这个删除事件，是由运维平台触发的。监听到该事件之后，就会终止作业调度。

```
1.  class InstanceShutdownStatusJobListener extends AbstractJobListener {
2.      @Override
3.      protected void dataChanged(final String path, final Type eventType, final String data) {
4.          if (!JobRegistry.getInstance().isShutdown(jobName) && !JobRegistry.getInstance().getJobS
    cheduleController(jobName).isPaused()  && isRemoveInstance(path, eventType) && !isReconnectedReg
    istryCenter()) {
5.              schedulerFacade.shutdownInstance();
6.            }
7.       }
8.      private boolean isRemoveInstance(final String path, final Type eventType) {
9.         return instanceNode.isLocalInstancePath(path) && Type.NODE_REMOVED == eventType;
10.     }
11.     private boolean isReconnectedRegistryCenter() {
12.         return instanceService.isLocalJobInstanceExisted();
13.     }
14. }
```

schedulerFacade.shutdownInstance()；终止步骤如下：

1. 如果当前节点是主节点，则删除主节点。
2. 关闭 monitor 服务
3. 关闭调节分布式状态不一致的服务
4. 把 job 实例，设置为关闭。

2.7 作业触发监听器

TriggerListenerManager 的作用是，监听/namespace/{jobname}/instances/ip-实例 id 的变更事件。

```
1.  class JobTriggerStatusJobListener extends AbstractJobListener {
2.      //--TRIGGER 节点的设置，也由平台来操纵 zk 来设置
3.      @Override
4.      protected void dataChanged(final String path, final Type eventType, final String data) {
5.          //--如果 data 不等于 TRIGGER || path 不是 instance 节点，|| 事件类型不是更新，则返回
6.          if (!InstanceOperation.TRIGGER.name().equals(data) || !instanceNode.isLocalInstancePath(
    path) || Type.NODE_UPDATED != eventType) {
7.              return;
8.          }
9.          //--马上清理掉 TRIGGER 标记
10.         instanceService.clearTriggerFlag();
11.         if (!JobRegistry.getInstance().isShutdown(jobName) && !JobRegistry.getInstance().isJobRu
    nning(jobName)) {
12.             // TODO 目前是作业运行时不能触发，未来改为堆积式触发
13.             //--马上触发一次任务执行
14.             JobRegistry.getInstance().getJobScheduleController(jobName).triggerJob();
15.         }
16.     }
17. }
```

这个变更事件，是由运维平台触发的。运维平台触发之后，会在 zk 的 /namespace/{jobname}/instances/ip-实例 id 节点上，设置一个 data 值 TRIGGRE。

监听器捕获到该事件之后，进行如下处理：

1. 清理 TRIGGRE 标记
2. 手动触发一次任务执行

2.8 重调度监听管理器

三、eljob 功能解读

3.1 错过任务重新执行-misfire

什么叫 misfire，举个例子，任务每 2 秒执行一次，正常执行顺序，2 秒,4 秒,6 秒,8 秒，依次类推。

假如实际执行的任务，执行时间需要 10 秒，那么它就不能在计划时间[4,6,8]执行，则判定为 misfire。

作业配置可设置属性 misfire，misfire=true 表示开启错误任务重新执行，misfire=false 表述关闭错误任务重新执行的策略。

eljob 本身没有使用 quartz 的 misfire 策略，而是关闭了其 misfire 策略，见如下代码

```
1.  public void init() {
2.      //1.持久化/jobName/config 信息到 zk
3.      LiteJobConfiguration liteJobConfigFromRegCenter = schedulerFacade.updateJobConfiguration(lit
    eJobConfig);
4.      //2.在 jobreg 上设置作业分片数
5.      JobRegistry.getInstance().setCurrentShardingTotalCount(liteJobConfigFromRegCenter.getJobName
    (), liteJobConfigFromRegCenter.getTypeConfig().getCoreConfig().getShardingTotalCount());
6.      //3.初始化 quartz 的实例和配置
7.      JobScheduleController jobScheduleController = new JobScheduleController(createScheduler(), c
    reateJobDetail(liteJobConfigFromRegCenter.getTypeConfig().getJobClass()), liteJobConfigFromRegCe
    nter.getJobName());
```

```
8.      JobRegistry.getInstance().registerJob(liteJobConfigFromRegCenter.getJobName(), jobScheduleCo
    ntroller, regCenter);
9.      //4.注册启动信息，ElasticJob 的任务服务器的启动流程就在这里定义
10.     schedulerFacade.registerStartUpInfo(!liteJobConfigFromRegCenter.isDisabled());
11.     jobScheduleController.scheduleJob(liteJobConfigFromRegCenter.getTypeConfig().getCoreConfig()
    .getCron());
12. }
```

```
1.  private Scheduler createScheduler() {
2.      Scheduler result;
3.      try {
4.          StdSchedulerFactory factory = new StdSchedulerFactory();
5.          factory.initialize(getBaseQuartzProperties());
6.          result = factory.getScheduler();
7.          result.getListenerManager().addTriggerListener(schedulerFacade.newJobTriggerListener());
8.       } catch (final SchedulerException ex) {
9.           throw new JobSystemException(ex);
10.     }
11.     return result;
12. }
13. private Properties getBaseQuartzProperties() {
14.    Properties result = new Properties();
15.    //--线程池的名字。可以使用后 Quartz 的 “org.quartz.simpl.SimpleThreadPool”。
16.    result.put("org.quartz.threadPool.class", org.quartz.simpl.SimpleThreadPool.class.getName());
17.    //--指定线程数量。一般 1-100 足以满足你的应用需求了。
18.    result.put("org.quartz.threadPool.threadCount", "1");
19.    //--使用 StdSchedulerFactory 的 getScheduler()方法创建的 scheduler 实例名称，在同一个程序中可以根据
    该名称来区分 scheduler。如果是在集群环境中使用，你必须使用同一个名称——集群环境下”逻辑”相同的
    scheduler。
20.    result.put("org.quartz.scheduler.instanceName", liteJobConfig.getJobName());
21.    //--如果任务超时 1 毫秒,则执行超时处理策略.(也就是说,不允许任务超时)
```

```
22.     //--https://www.cnblogs.com/daxin/p/3919927.html
23.     //--https://www.cnblogs.com/pzy4447/p/5201674.html
24.     result.put("org.quartz.jobStore.misfireThreshold", "1");
25.     result.put("org.quartz.plugin.shutdownhook.class", JobShutdownHookPlugin.class.getName());
26.     result.put("org.quartz.plugin.shutdownhook.cleanShutdown", Boolean.TRUE.toString());
27.     return result;
28. }
```

org.quartz.jobStore.misfireThreshold=1，表示如果任务错误执行时间 1 毫秒，就触发 misfire 策略

触发之后的动作，见如下代码

```
1.  private CronTrigger createTrigger(final String cron) {
2.   //--在每个星期的周一至周五的上午 9 点到下午 17 点，每隔一个小时执行一次。
3.  //--withMisfireHandlingInstructionIgnoreMisfires（所有 misfire 的任务会马上执行）
4.  //--打个比方，如果 9 点 misfire 了，在 10：15 系统恢复之后，9 点，10 点的 misfire 会马上执行
5.  //--withMisfireHandlingInstructionDoNothing(所有的 misfire 不管，执行下一个周期的任务)
6.  //--withMisfireHandlingInstructionFireAndProceed（会合并部分的 misfire,正常执行下一个周期的任务）
7.  //--假设 9，10 的任务都 misfire 了，系统在 10：15 分起来了。只会执行一次 misfire，下次正点执行。
8.    return TriggerBuilder.newTrigger().withIdentity(triggerIdentity).withSchedule(CronScheduleBuil
    der.cronSchedule(cron).withMisfireHandlingInstructionDoNothing()).build();
9.  }
```

withMisfireHandlingInstructionDoNothing（）表示触发之后，什么事都不做。

那么 eljob，到底是如何处理 misfire 的？

我们找到如下代码

com.dangdang.ddframe.job.executor.AbstractElasticJobExecutor.execute(); 任 务执行的代码

```
1.  public final void execute() {
2.     try {
3.        //--检查作业执行环境,本机与注册中心的时间误差秒数不在允许范围所抛出的异常
4.        jobFacade.checkJobExecutionEnvironment();
5.     } catch (final JobExecutionEnvironmentException cause) {
```

```
6.          jobExceptionHandler.handleException(jobName, cause);
7.      }
8.      ShardingContexts shardingContexts = jobFacade.getShardingContexts();// --获取分片上下文环境
9.        if (shardingContexts.isAllowSendJobEvent()) {
10.          jobFacade.postJobStatusTraceEvent(shardingContexts.getTaskId(), State.TASK_STAGING, Stri
    ng.format("Job '%s' execute begin.", jobName));
11.       }
12.       if (jobFacade.misfireIfRunning(shardingContexts.getShardingItemParameters().keySet())) {
13.          if (shardingContexts.isAllowSendJobEvent()) {
14.             jobFacade.postJobStatusTraceEvent(shardingContexts.getTaskId(), State.TASK_FINISHED,
    String.format("Previous job '%s' - shardingItems '%s' is still running, misfired job will start
    after previous job completed.", jobName,shardingContexts.getShardingItemParameters().keySet()));
15.           }
16.          return;
17.   }
```

在任务触发之后，去检测 zk 上，是否有正在运行的分片，如果有，则设置分片 misfire 节点到 zk 上。往下看代码

```
1.   while (jobFacade.isExecuteMisfired(shardingContexts.getShardingItemParameters().keySet())) {
2.       jobFacade.clearMisfire(shardingContexts.getShardingItemParameters().keySet());
3.       System.out.println("开始执行 misfire========");
4.       execute(shardingContexts, JobExecutionEvent.ExecutionSource.MISFIRE);
5.       System.out.println("结束执行 misfire========");
6.   }
```

在这里，去检测 zk 上是否存在 misfire 节点，如果存在，首先清理点 misfire 节点，然后，再调用 execute 来触发任务的执行。

3.2 调解分布式状态不一致服务

在注册作业启动信息的时候，会启动该服务，看代码：

```
1.   public void registerStartUpInfo(final boolean enabled) {
```

```
2.      listenerManager.startAllListeners();//开启所有监听者
3.      leaderService.electLeader();//--选举主节点.
4.      serverService.persistOnline(enabled);//--持久化作业服务器上线信息,创建 jobname/servers 节点
5.      instanceService.persistOnline();//--持久化作业运行实例上线相关信息,创建 jobname/instances 节点
6.      shardingService.setReshardingFlag();//--设置需要重新分片的标记,创建
    jobname/leader/sharding/necessary
7.      monitorService.listen();//--初始化作业监听服务
8.      if (!reconcileService.isRunning()) {
9.          reconcileService.startAsync();
10.     }
11. }
```

reconcileService 继承了 AbstractScheduledService，它是 google-guava 包里的服务，该服务本质上讲，就是一个用于处理周期类任务的服务。
继承该类之后，需要实现两个方法：
runOneIteration()和 scheduler()
scheduler()方法返回周期任务执行的间隔时间。
runOneIteration()方法是具体执行任务的方法。
下面我们来看 runOneIteration()具体干了什么。

```
1.  protected void runOneIteration() throws Exception {
2.      LiteJobConfiguration config = configService.load(true);
3.      int reconcileIntervalMinutes = null == config ? 1 : config.getReconcileIntervalMinutes();
4.      //--每 60 秒去检查一下，/{jobname}/sharding/{item 分片}/instance 实例，是否存在，已经离线的服务器。
    如果存在，则设置需要重新分片的标记
5.      if (reconcileIntervalMinutes > 0 && (System.currentTimeMillis() - lastReconcileTime >= reconc
    ileIntervalMinutes * 60 * 1000)) {
6.          lastReconcileTime = System.currentTimeMillis();
7.          if (leaderService.isLeaderUntilBlock() && !shardingService.isNeedSharding() && shardingSe
    rvice.hasShardingInfoInOfflineServers()) {
8.              log.warn("Elastic Job: job status node has inconsistent value,start reconciling...");
9.              shardingService.setReshardingFlag();
10.         }
```

```
    }
}
```

可以看到该方法内部，去检测了
/namespace/{jobname}/sharding/{item 分片}/instance 实例是否存活。
举个例子：有 A，B 两个实例，分片数为 2。A 得到 0 号分片，B 得到 1 号分片。
这是 B 实例宕机，那么就需要把 1 号分片，重新进行分配，分到 A 实例上去。
这里的 shardingService.setReshardingFlag();就是设置了分片标记：
/namespace/{jobname}/leader/sharding/necessary

3.3 作业事件追踪

参考文献: https://www.jianshu.com/p/058051ba8204

四、eljob 里的设计模式

4.1 单例模式

设计模式中最基本也是最常用的一种，见如下代码：

```
@NoArgsConstructor(access = AccessLevel.PRIVATE)
public final class JobRegistry {
    private static volatile JobRegistry instance;
    private Map<String, JobScheduleController> schedulerMap = new ConcurrentHashMap<>();//--存放正在跑的 job 的计划映射
    private Map<String, CoordinatorRegistryCenter> regCenterMap = new ConcurrentHashMap<>();
    private Map<String, JobInstance> jobInstanceMap = new ConcurrentHashMap<>();//--保持 job 本机实例映射
    private Map<String, Boolean> jobRunningMap = new ConcurrentHashMap<>();
    private Map<String, Integer> currentShardingTotalCountMap = new ConcurrentHashMap<>();//--存放每个 job 对应的分片总数。key：jobname，value：分片总数
    /**
```

```
10.      * 获取作业注册表实例.
11.      * @return 作业注册表实例
12.      */
13.     public static JobRegistry getInstance() {
14.         if (null == instance) {
15.             synchronized (JobRegistry.class) {
16.                 if (null == instance) {
17.                     instance = new JobRegistry();
18.                 }
19.             }
20.         }
21.         return instance;
22.     }
```

该单例模式采用双重检查锁进行实现，需要注意静态实例用 volatile 关键字修饰是很有必要的，如果没有 volatile 则该单例模式可能出现多例的情况，原因如下：
volatile（java5）：可以保证多线程下的可见性;

读 volatile：每当子线程某一语句要用到 volatile 变量时，都会从主线程重新拷贝一份，这样就保证子线程的会跟主线程的一致。

写 volatile: 每当子线程某一语句要写 volatile 变量时，都会在读完后同步到主线程去，这样就保证主线程的变量及时更新。

4.2 工厂模式

```
1.  public final class LiteJob implements Job {
2.      @Setter
3.      private ElasticJob elasticJob;
4.      @Setter
5.      private JobFacade jobFacade;
6.      @Override
7.      public void execute(final JobExecutionContext context) throws JobExecutionException {
8.          JobExecutorFactory.getJobExecutor(elasticJob, jobFacade).execute();
```

```
9.      }
10. }
```

此处使用的是简单工厂模式，通过不同参数生成不同 AbstractElasticJobExecutor 实例对象

```
1.  @NoArgsConstructor(access = AccessLevel.PRIVATE)
2.  public final class JobExecutorFactory {
3.      /**
4.       * 获取作业执行器.
5.       * @param elasticJob 分布式弹性作业
6.       * @param jobFacade 作业内部服务门面服务
7.       * @return 作业执行器
8.       */
9.      @SuppressWarnings("unchecked")
10.     public static AbstractElasticJobExecutor getJobExecutor(final ElasticJob elasticJob, final J
    obFacade jobFacade) {
11.         if (null == elasticJob) {
12.             return new ScriptJobExecutor(jobFacade);
13.         }
14.         if (elasticJob instanceof SimpleJob) {
15.             return new SimpleJobExecutor((SimpleJob) elasticJob, jobFacade);
16.         }
17.         if (elasticJob instanceof DataflowJob) {
18.             return new DataflowJobExecutor((DataflowJob) elasticJob, jobFacade);
19.         }
20.         throw new JobConfigurationException("Cannot support job type '%s'", elasticJob.getClass(
    ).getCanonicalName());
21.     }
22. }
```

4.3 模版模式

```
1.  @Slf4j
2.  public abstract class AbstractElasticJobExecutor {
3.      @Getter(AccessLevel.PROTECTED)
4.      private final JobFacade jobFacade;
5.      @Getter(AccessLevel.PROTECTED)
6.      private final JobRootConfiguration jobRootConfig;
7.      private final String jobName;
8.      private final ExecutorService executorService;
9.      private final JobExceptionHandler jobExceptionHandler;
10.     private final Map<Integer, String> itemErrorMessages;
11.     protected AbstractElasticJobExecutor(final JobFacade jobFacade) {
12.         this.jobFacade = jobFacade;
13.         jobRootConfig = jobFacade.loadJobRootConfiguration(true);
14.         jobName = jobRootConfig.getTypeConfig().getCoreConfig().getJobName();
15.         executorService = ExecutorServiceHandlerRegistry.getExecutorServiceHandler(jobName, (Exe
    cutorServiceHandler) getHandler(JobProperties.JobPropertiesEnum.EXECUTOR_SERVICE_HANDLER));
16.         jobExceptionHandler = (JobExceptionHandler) getHandler(JobProperties.JobPropertiesEnum.J
    OB_EXCEPTION_HANDLER);
17.         itemErrorMessages = new ConcurrentHashMap<>(jobRootConfig.getTypeConfig().getCoreConfig(
    ).getShardingTotalCount(), 1);
18.     }
19.     protected abstract void process(ShardingContext shardingContext);
```

此处使用模板模式，使父类算法的具体实现延迟到子类，使子类不必改变一个算法的结构即可重定义该算法的步骤。

4.4 策略模式

策略模式是比较常用的一种设计模式，其设计理念是：定义一系列算法，将它们一个个封装起来，并且使他们之间可以相互替换。

在 eljob 中的使用中是针对不同类型的任务类型设计不同的业务逻辑，其源码如

下：

```
1.  public interface JobTypeConfiguration {
2.      /**
3.       * 获取作业类型.
4.       * @return 作业类型
5.       */
6.      JobType getJobType();
7.      /**
8.       * 获取作业实现类名称.
9.       * @return 作业实现类名称
10.      */
11.     String getJobClass();
12.     /**
13.      * 获取作业核心配置.
14.      * @return 作业核心配置
15.      */
16.     JobCoreConfiguration getCoreConfig();
17. }
```

```
1.  @RequiredArgsConstructor
2.  @Getter
3.  public final class SimpleJobConfiguration implements JobTypeConfiguration {
4.      private final JobCoreConfiguration coreConfig;
5.      private final JobType jobType = JobType.SIMPLE;
6.      private final String jobClass;
7.  }
```

```
1.  @RequiredArgsConstructor
2.  @Getter
```

```
3.  public final class DataflowJobConfiguration implements JobTypeConfiguration {
4.      private final JobCoreConfiguration coreConfig;
5.      private final JobType jobType = JobType.DATAFLOW;
6.      private final String jobClass;
7.      private final boolean streamingProcess;
8.  }
```

```
1.      /**
2.       * 创建 Lite 作业配置构建器.
3.       * @param jobConfig 作业配置
4.       * @return Lite 作业配置构建器
5.       */
6.      public static Builder newBuilder(final JobTypeConfiguration jobConfig) {
7.          return new Builder(jobConfig);
8.      }
```

通过传入不同的 jobConfig 对象实现不同类型的任务。

参考文献：https://blog.csdn.net/u011507568/article/details/55189082

五、运维平台

5.1 运维平台启动

1.进入项目目录，编译项目 mvn clean install –DskipTests

2.cd elastic-job/elastic-job-lite/elastic-job-lite-console/target

3.tar –zxvf elastic-job-lite-console-2.1.6-SNAPSHOT.tar.gz

4.cd elastic-job-lite-console-2.1.6-SNAPSHOT/bin

5.sh start.sh

6.启动之后，会监听到本地的 8899 端口

7.打开浏览器，输入 http://127.0.0.1:8899 用户名：root，密码：root。