基础

读源码需要的基础：

1.zookeeper入门:https://zookeeper.apache.org/doc/r3.6.0/zookeeperStarted.html

2.curator入门:https://curator.apache.org/getting-started.html

3.Quartz入门:http://www.quartz-scheduler.org/documentation/quartz-2.3.0/quick-start.html

4.Elastic-Job入门(中文):http://elasticjob.io/docs/elastic-job-lite/00-overview/

5.Elastic-Job还用到guava和lombok，不过不了解也不影响阅读

6.最好能装个可视化工具，如zooinspector(https://github.com/apache/zookeeper/tree/master/zookeeper-contrib/zookeeper-contrib-zooinspector)；或者zkui(https://github.com/DeemOpen/zkui)【zooinspector我电脑编译jar文件后，swing的图形界面显示一直有问题，尝试过各版本都有问题，jar包在maven中央仓库也有提供，如果还是想用zooinspector，可以看下(https://mvnrepository.com/artifact/org.apache.zookeeper/zookeeper-contrib-zooinspector)】

Demo

Demo:https://github.com/creasylai19/zookeeperdemo/tree/elastic_job_demo

demo只有两个class

MainClass

```
public class MainClass {

    private static final Logger logger = Logger.getLogger(MainClass.class);

    public static void main(String[] args) {
        new JobScheduler(createRegistryCenter(),
createJobConfiguration()).init();
    }

    private static CoordinatorRegistryCenter createRegistryCenter() {
        CoordinatorRegistryCenter regCenter = new ZookeeperRegistryCenter(new
ZookeeperConfiguration("127.0.0.1:2181,127.0.0.1:2182,127.0.0.1:2183",
"elastic-job-demo"));
        regCenter.init();
        return regCenter;
    }

    private static LiteJobConfiguration createJobConfiguration() {
        // 定义作业核心配置，任务分为10片，每片有不同的参数对应不同业务逻辑
        JobCoreConfiguration simpleCoreConfig = JobCoreConfiguration
```

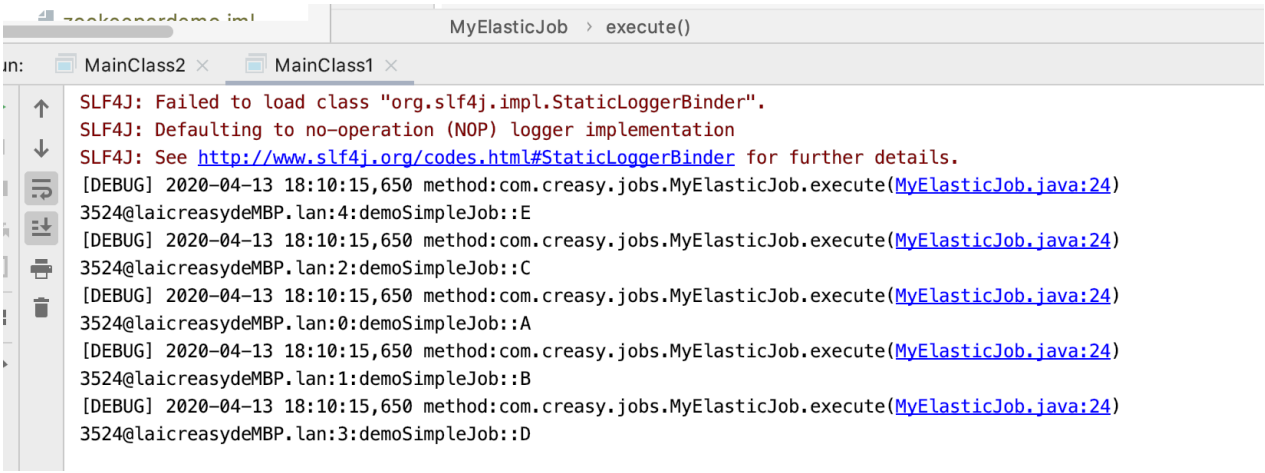
```
                .newBuilder("demoSimpleJob", "0/15 * * * * ?", 10)

.shardingItemParameters("0=A,1=B,2=C,3=D,4=E,5=F,6=G,7=H,8=I,9=J")
                .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();
        return simpleJobRootConfig;
    }

}
```

任务MyElasticJob

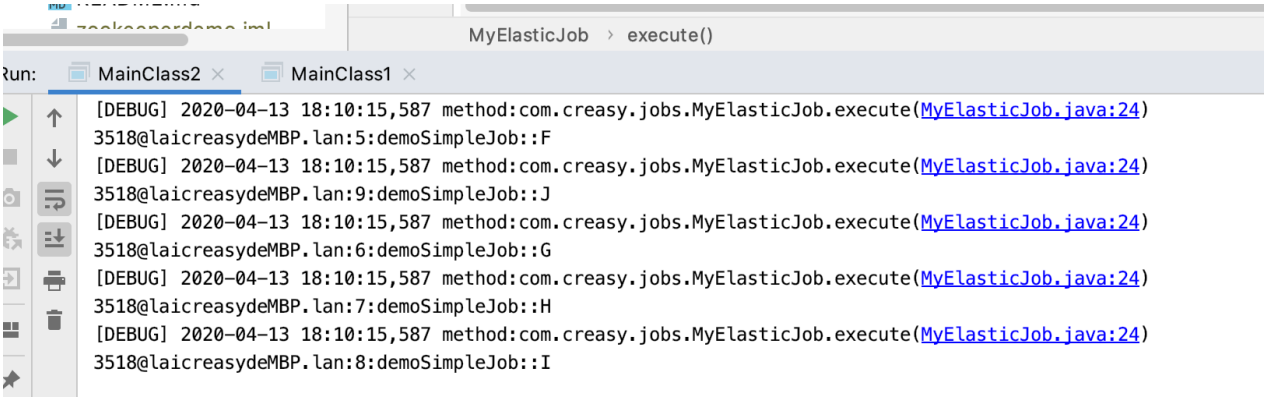
```
public class MyElasticJob implements SimpleJob {

    private static final Logger logger = Logger.getLogger(MyElasticJob.class);

    @Override
    public void execute(ShardingContext context) {

 logger.debug(ManagementFactory.getRuntimeMXBean().getName()+":"+context.getSh
ardingItem()+":"+context.getJobName()+":"+context.getJobParameter()+":"+contex
t.getShardingParameter());
        switch (context.getShardingItem()) {
            case 0:
                break;
            case 1:
                break;
            case 2:
                break;
            default:
        }
    }
}
```

运行结果(开启了两个实例，每个实例各运行5个分片，默认的分配策略是平均分配)

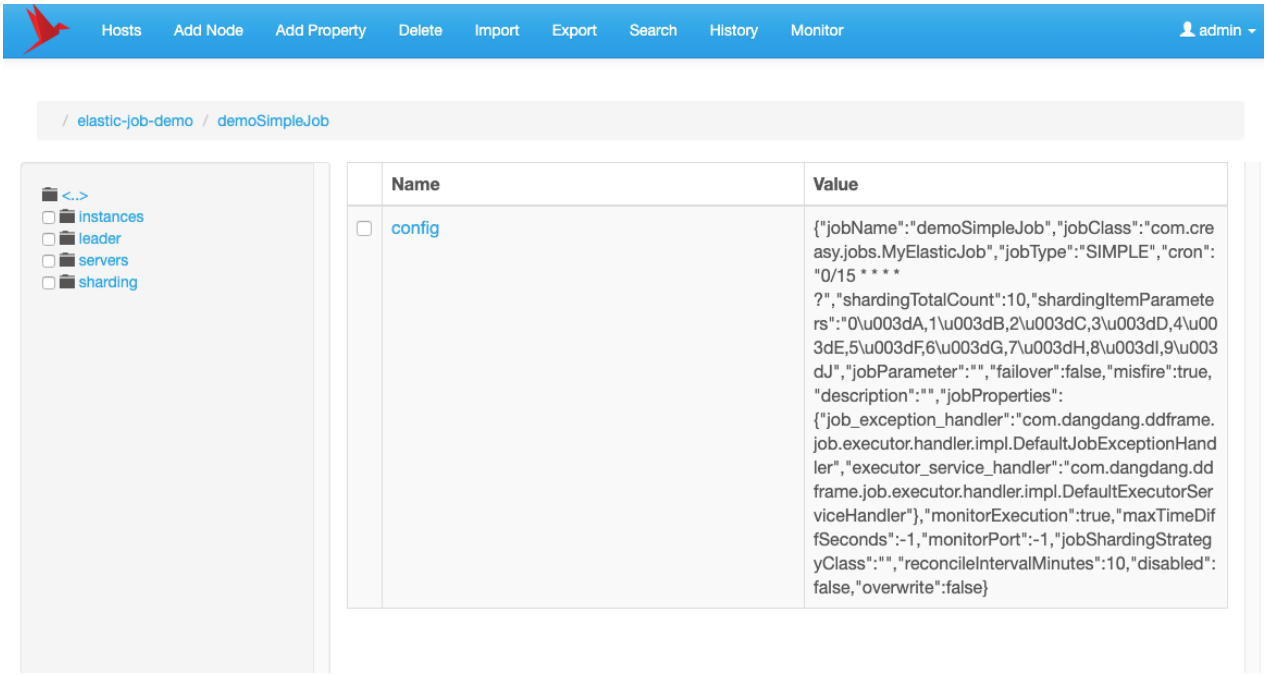
实例1：



实例2：

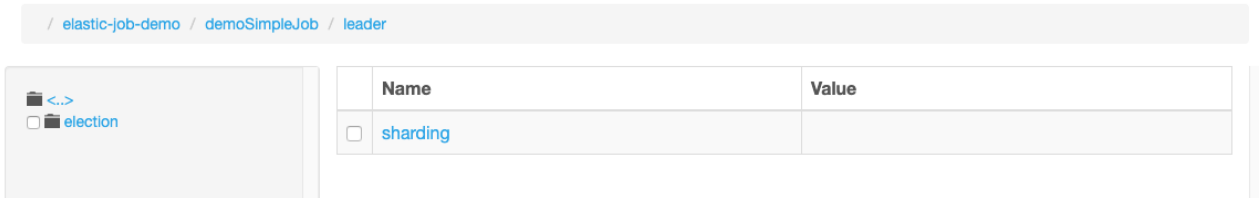


zookeeper中的节点信息：



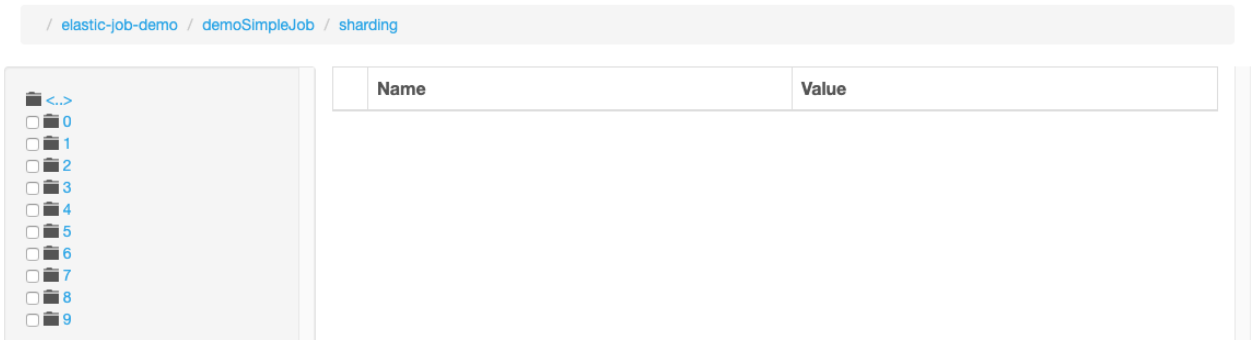
instances节点：作业运行实例信息，子节点是当前作业运行实例的主键

leader节点：作业服务器主节点信息

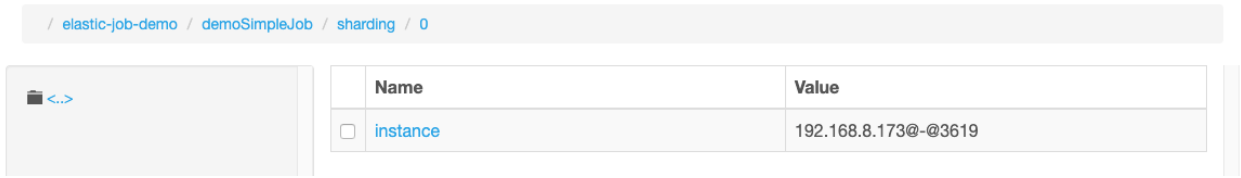


servers节点：作业服务器信息，子节点是作业服务器的IP地址

sharding节点：作业分片信息，子节点是分片项序号，从零开始，至分片总数减一



每个分片存储运行本分片的实例等信息

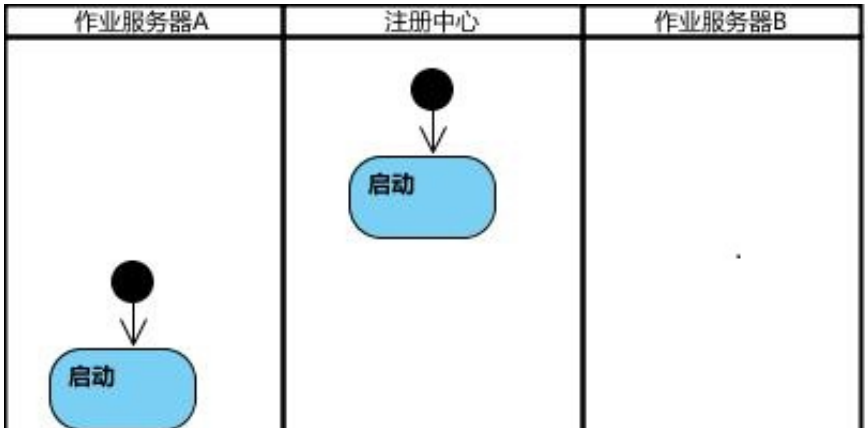


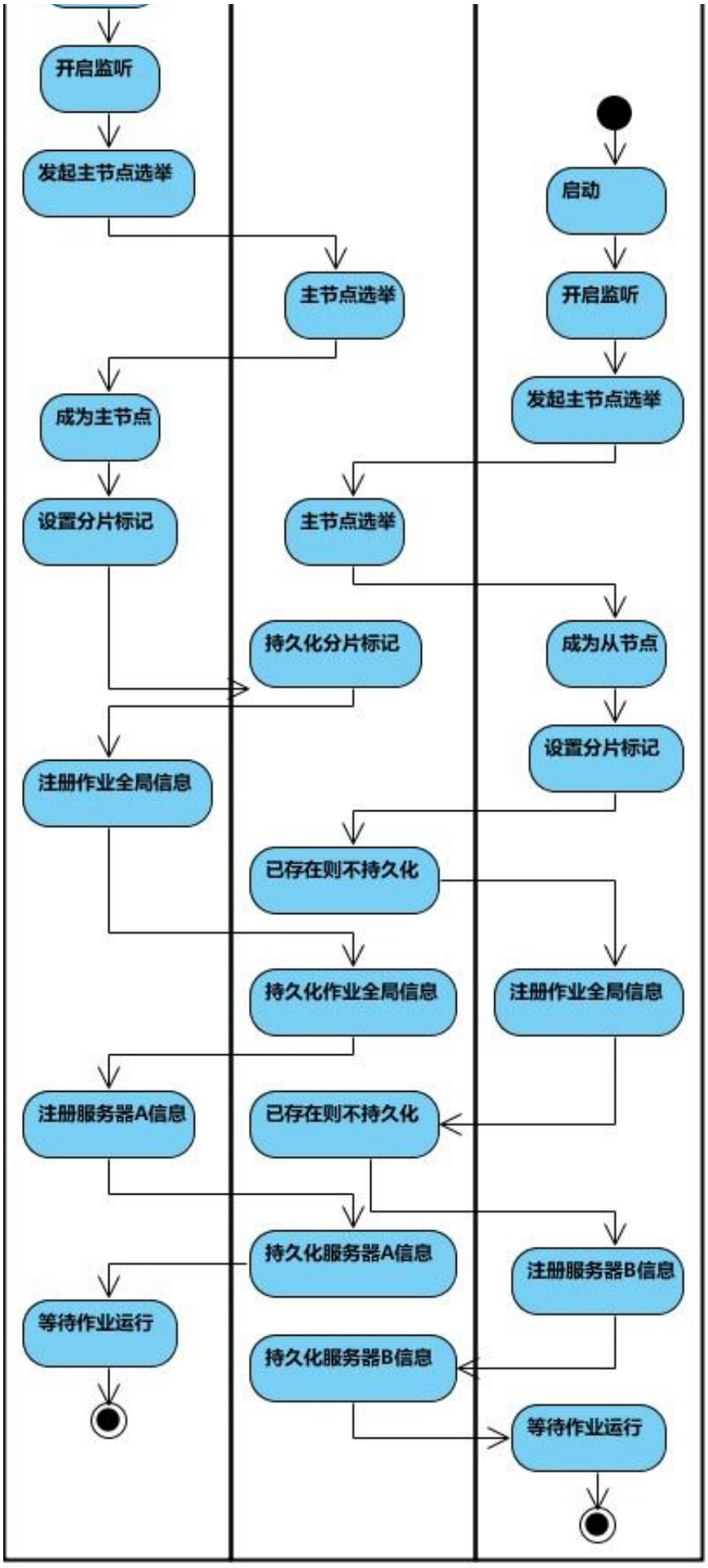
Elastic-Job流程

简单分，可以分为初始化流程和任务执行流程。

初始化流程

初始化流程主要为任务信息的注册、任务处理器实例的注册、leader选举、调度器的启动等。

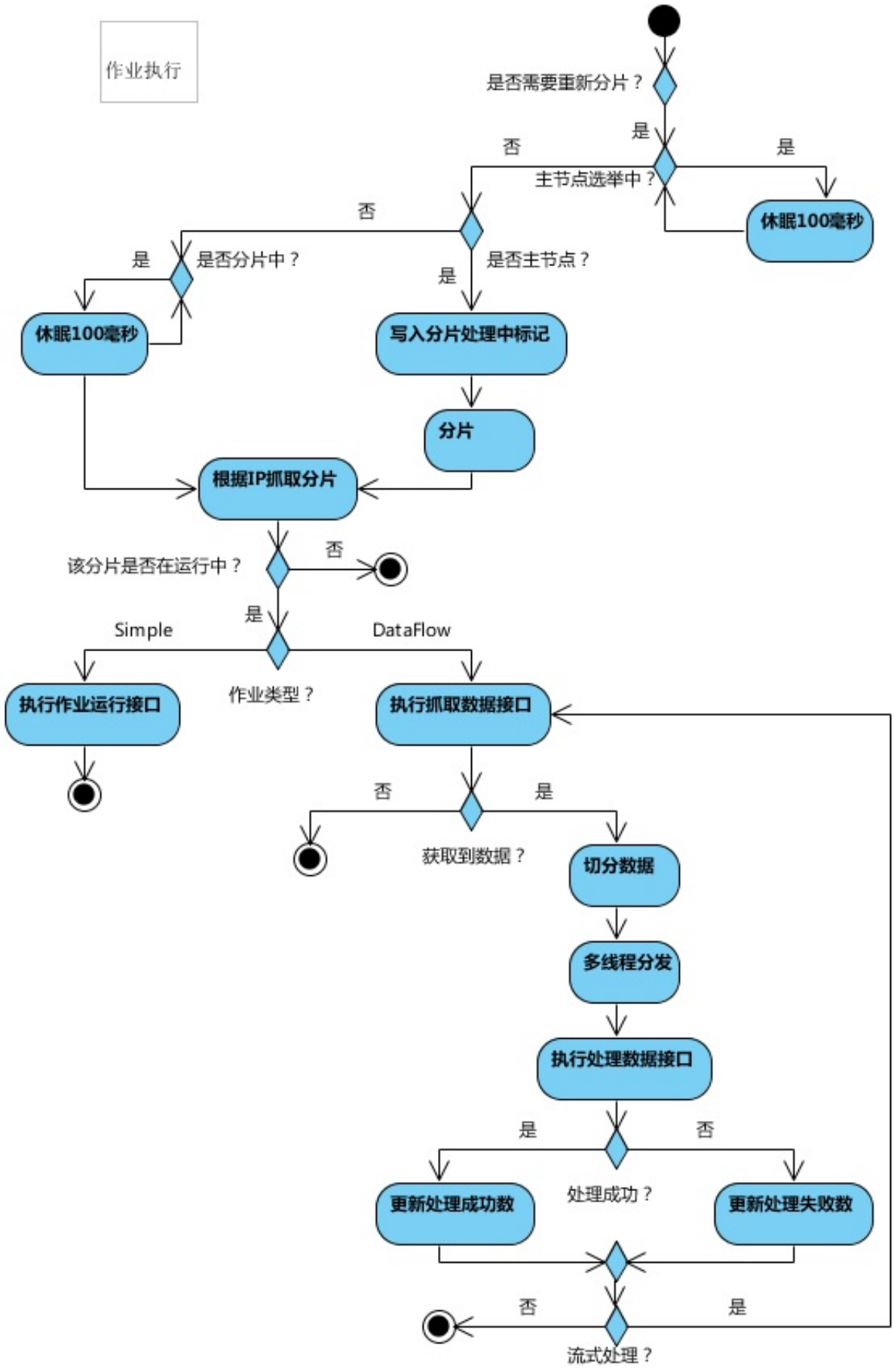




图片：http://elasticjob.io/docs/elastic-job-lite/img/principles/job_start.jpg

任务执行流程

任务执行流程包括获取线程池，做数据分片，根据分片信息生成任务放入线程池中执行。



图片：http://elasticjob.io/docs/elastic-job-lite/img/principles/job_exec.jpg

---

本文只分析任务执行流程【Simple类型的任务】

步骤1

初始化阶段，放入调度器Scheduler中的任务是LiteJob.class，它实现了org.quartz.Job接口

```
//JobScheduler.class
  private JobDetail createJobDetail(final String jobClass) {
        JobDetail result =
JobBuilder.newJob(LiteJob.class).withIdentity(liteJobConfig.getJobName()).buil
d();//在这里生成任务Job
        result.getJobDataMap().put(JOB_FACADE_DATA_MAP_KEY, jobFacade);
        Optional<ElasticJob> elasticJobInstance = createElasticJobInstance();
        if (elasticJobInstance.isPresent()) {
            result.getJobDataMap().put(ELASTIC_JOB_DATA_MAP_KEY,
elasticJobInstance.get());
        } else if (!jobClass.equals(ScriptJob.class.getCanonicalName())) {
            try {
                result.getJobDataMap().put(ELASTIC_JOB_DATA_MAP_KEY,
Class.forName(jobClass).newInstance());
            } catch (final ReflectiveOperationException ex) {
                throw new JobConfigurationException("Elastic-Job: Job class
'%s' can not initialize.", jobClass);
            }
        }
        return result;
    }
```

步骤2

调度器Scheduler会调用LiteJob.class的execute方法

```
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();
    }
}
```

步骤3

在`JobExecutorFactory.getJobExecutor(elasticJob, jobFacade)`如果关联的任务没有对应线程池，则会先生成线程池

```
//AbstractElasticJobExecutor.class
    protected AbstractElasticJobExecutor(final JobFacade jobFacade) {
        this.jobFacade = jobFacade;
        jobRootConfig = jobFacade.loadJobRootConfiguration(true);
        jobName = jobRootConfig.getTypeConfig().getCoreConfig().getJobName();
        executorService =
ExecutorServiceHandlerRegistry.getExecutorServiceHandler(jobName,
(ExecutorServiceHandler)
getHandler(JobProperties.JobPropertiesEnum.EXECUTOR_SERVICE_HANDLER));//在这步生
成线程池
        jobExceptionHandler = (JobExceptionHandler)
getHandler(JobProperties.JobPropertiesEnum.JOB_EXCEPTION_HANDLER);
        itemErrorMessages = new ConcurrentHashMap<>
(jobRootConfig.getTypeConfig().getCoreConfig().getShardingTotalCount(), 1);
    }
```

默认线程池大小是可用CPU的两倍

```
//ExecutorServiceObject.class
    public ExecutorServiceObject(final String namingPattern, final int
threadSize) {
        workQueue = new LinkedBlockingQueue<>();
        threadPoolExecutor = new ThreadPoolExecutor(threadSize, threadSize,
5L, TimeUnit.MINUTES, workQueue,
                new BasicThreadFactory.Builder().namingPattern(Joiner.on("-
").join(namingPattern, "%s")).build());
        threadPoolExecutor.allowCoreThreadTimeOut(true);
    }
```

步骤4

执行execute方法

```
//AbstractElasticJobExecutor.class
    public final void execute() {
        ...
        ShardingContexts shardingContexts = jobFacade.getShardingContexts();//
获取当前作业处理器要处理的分片(如果分片未分配，leader需要执行任务分片分配)
        ...
        execute(shardingContexts,
JobExecutionEvent.ExecutionSource.NORMAL_TRIGGER);//根据分片执行任务
        ...
    }
```

步骤4.1

任务分片分配过程：

```
//ShardingService.class
public void shardingIfNecessary() {
        List<JobInstance> availableJobInstances =
instanceService.getAvailableJobInstances();//所有注册的实例
        if (!isNeedSharding() || availableJobInstances.isEmpty()) {//如果不需要
分配或者作业处理器为空则直接返回
            return;
        }
        if (!leaderService.isLeaderUntilBlock()) {//判断是否主节点(如果没有主节点，
则进行leader选举)
            blockUntilShardingCompleted();//非主节点，并且主节点在分片过程中，则阻塞
等待。每次sleep100毫秒，sleep结束了重新判断是否需要再次sleep
            return;
        }
  //下方都是leader要做的事情
        waitingOtherJobCompleted();
        LiteJobConfiguration liteJobConfig = configService.load(false);//从zk中
获取配置信息
        int shardingTotalCount =
liteJobConfig.getTypeConfig().getCoreConfig().getShardingTotalCount();
        log.debug("Job '{}' sharding begin.", jobName);
        jobNodeStorage.fillEphemeralJobNode(ShardingNode.PROCESSING, "");//设置
正在分片标识
        resetShardingInfo(shardingTotalCount);//重置分片信息，删除原来的分片信息再插
入新的分片信息
        JobShardingStrategy jobShardingStrategy =
JobShardingStrategyFactory.getStrategy(liteJobConfig.getJobShardingStrategyCla
ss());//获取分片分配策略，默认为平均分配
        jobNodeStorage.executeInTransaction(new
PersistShardingInfoTransactionExecutionCallback(jobShardingStrategy.sharding(a
vailableJobInstances, jobName, shardingTotalCount)));//在事务中执行分片分配，并把信
息注册到zk中
        log.debug("Job '{}' sharding complete.", jobName);
    }
```

其他非leader节点执行

```
//ShardingService.class
//如果不是leader，并且正在分片中，则循环休眠，每次sleep100毫秒
private void blockUntilShardingCompleted() {
        while (!leaderService.isLeaderUntilBlock() &&
(jobNodeStorage.isJobNodeExisted(ShardingNode.NECESSARY) ||
jobNodeStorage.isJobNodeExisted(ShardingNode.PROCESSING))) {
            log.debug("Job '{}' sleep short time until sharding completed.",
jobName);
            BlockUtils.waitingShortTime();
        }
    }
```

平均分配的算法

```
//AverageAllocationJobShardingStrategy.class
    /**
     * 过程是：
     *      1.分片总数/作业处理器总数=N，每个作业处理器拿N个任务，
     *      2.分片总数%作业处理器总数=剩余未分配数，每个作业处理器各拿一个分片，直至分完
     * @param jobInstances 所有的作业处理器
     * @param jobName 作业名称
     * @param shardingTotalCount 分片大小
     * @return
     */
    public Map<JobInstance, List<Integer>> sharding(final List<JobInstance>
jobInstances, final String jobName, final int shardingTotalCount) {
        if (jobInstances.isEmpty()) {
            return Collections.emptyMap();
        }
        Map<JobInstance, List<Integer>> result = shardingAliquot(jobInstances,
shardingTotalCount);//先平均分片
        addAliquant(jobInstances, shardingTotalCount, result);//剩余未分片的再顺
序分给各个作业处理器
        return result;
    }
```

在事物中完成分片

```
        @Override
        public void execute(final CuratorTransactionFinal
curatorTransactionFinal) throws Exception {
          //循环遍历每个实例的分片，在zk中生成对应节点
【sharding/n/instance:instanceId】
            for (Map.Entry<JobInstance, List<Integer>> entry :
shardingResults.entrySet()) {
                for (int shardingItem : entry.getValue()) {

 curatorTransactionFinal.create().forPath(jobNodePath.getFullPath(ShardingNode
.getInstanceNode(shardingItem)),
entry.getKey().getJobInstanceId().getBytes()).and();
                }
            }

 curatorTransactionFinal.delete().forPath(jobNodePath.getFullPath(ShardingNode
.NECESSARY)).and();//删除/leader/sharding/necessary标识

 curatorTransactionFinal.delete().forPath(jobNodePath.getFullPath(ShardingNode
.PROCESSING)).and();//删除/leader/sharding/processing标识
        }
```

步骤4.2

获取当前任务处理器要处理的分片

```
//ShardingService.class
public List<Integer> getShardingItems(final String jobInstanceId) {
        JobInstance jobInstance = new JobInstance(jobInstanceId);
        if (!serverService.isAvailableServer(jobInstance.getIp())) {
            return Collections.emptyList();
        }
        List<Integer> result = new LinkedList<>();
        int shardingTotalCount =
configService.load(true).getTypeConfig().getCoreConfig().getShardingTotalCount
();
        for (int i = 0; i < shardingTotalCount; i++) {
            if
(jobInstance.getJobInstanceId().equals(jobNodeStorage.getJobNodeData(ShardingN
ode.getInstanceNode(i)))) {//如果分片的instance节点信息和当前任务处理器的实例ID相同，
则代表这个分片归当前任务处理器处理
                result.add(i);
            }
        }
        return result;
    }
```

步骤4.3

执行每个分片任务

```
//AbstractElasticJobExecutor.class
    private void process(final ShardingContexts shardingContexts, final
JobExecutionEvent.ExecutionSource executionSource) {
        Collection<Integer> items =
shardingContexts.getShardingItemParameters().keySet();
        ...
        final CountDownLatch latch = new CountDownLatch(items.size());
        for (final int each : items) {//当前任务处理器要处理的分片，循环放到线程池中处
理
            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);//处
理每个分片
                    } finally {
                        latch.countDown();
                    }
                }
            });
        }
        try {
            latch.await();
        } catch (final InterruptedException ex) {
            Thread.currentThread().interrupt();
        }
    }
```

组装参数

```
//AbstractElasticJobExecutor.class
    private void process(final ShardingContexts shardingContexts, final int
item, final JobExecutionEvent startEvent) {
        ...
        try {
            process(new ShardingContext(shardingContexts, item));//这里会根据
item，拿到item这个分片的参数，如item为0，初始化时参数0=A,1=B,2=C...，那会解析拿到A这个参
数
            ...
            // CHECKSTYLE:OFF
        } catch (final Throwable cause) {
            // CHECKSTYLE:ON
            ...
        }
    }
```

process这个调用就会调到我们自己定义的SimpleJob实现类了

```
public final class SimpleJobExecutor extends AbstractElasticJobExecutor {

    private final SimpleJob simpleJob;

    public SimpleJobExecutor(final SimpleJob simpleJob, final JobFacade
jobFacade) {
        super(jobFacade);
        this.simpleJob = simpleJob;
    }

    @Override
    protected void process(final ShardingContext shardingContext) {
        simpleJob.execute(shardingContext);//实现类execute(shardingContext)接口
调用
    }
}
```

以上是Elastic-Job任务执行流程的大概分析，Elastic-Job支持弹性扩容缩容、失效转移、监控运维等，后续再做分析吧