Apache ShardingSphere ElasticJob document

Apache ShardingSphere

2022 年 03 月 17 日

Contents

ElasticJob 是面向互联网生态和海量任务的分布式调度解决方案，由两个相互独立的子项目 ElasticJob-Lite 和 ElasticJob-Cloud 组成。它通过弹性调度、资源管控、以及作业治理的功能，打造一个适用于互联网场景的分布式调度解决方案，并通过开放的架构设计，提供多元化的作业生态。它的各个产品使用统一的作业 API，开发者仅需一次开发，即可随意部署。

ElasticJob 已于 2020 年 5 月 28 日成为 Apache ShardingSphere 的子项目。欢迎通过邮件列表参与讨论。

1 简介

使用 ElasticJob 能够让开发工程师不再担心任务的线性吞吐量提升等非功能需求，使他们能够更加专注于面向业务编码设计；同时，它也能够解放运维工程师，使他们不必再担心任务的可用性和相关管理需求，只通过轻松的增加服务节点即可达到自动化运维的目的。

1.1 ElasticJob-Lite

定位为轻量级无中心化解决方案，使用 jar 的形式提供分布式任务的协调服务。

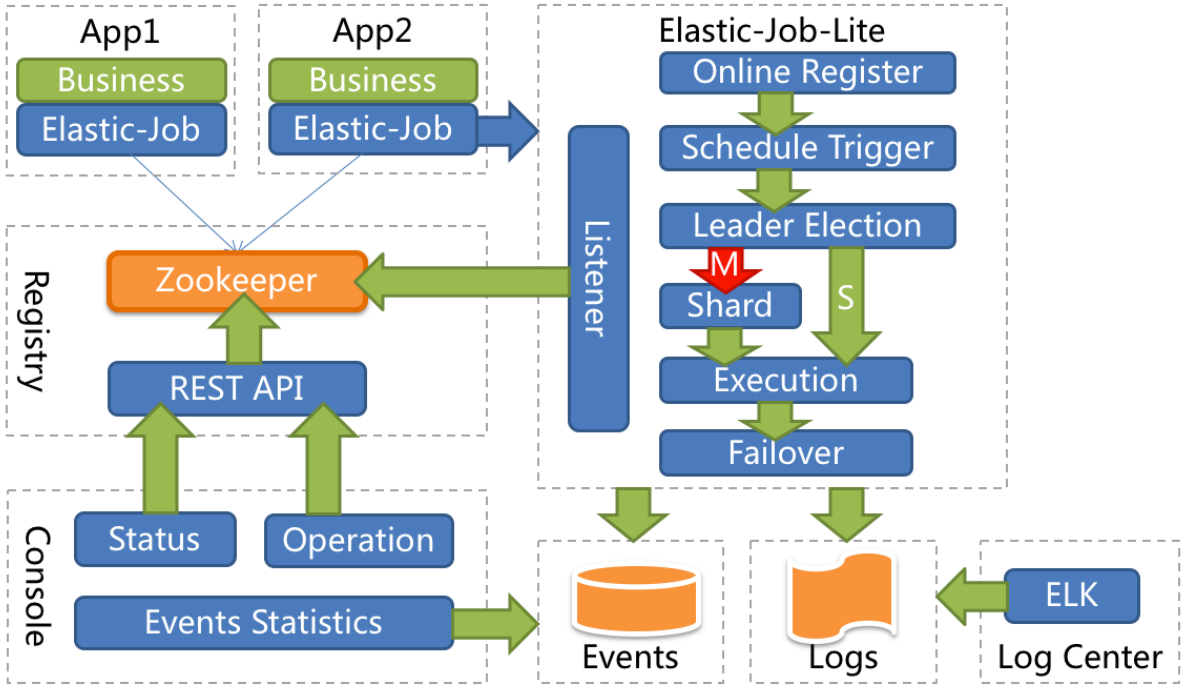

图 1: ElasticJob-Lite Architecture

1.2 ElasticJob-Cloud

采用自研 Mesos Framework 的解决方案，额外提供资源治理、应用分发以及进程隔离等功能。

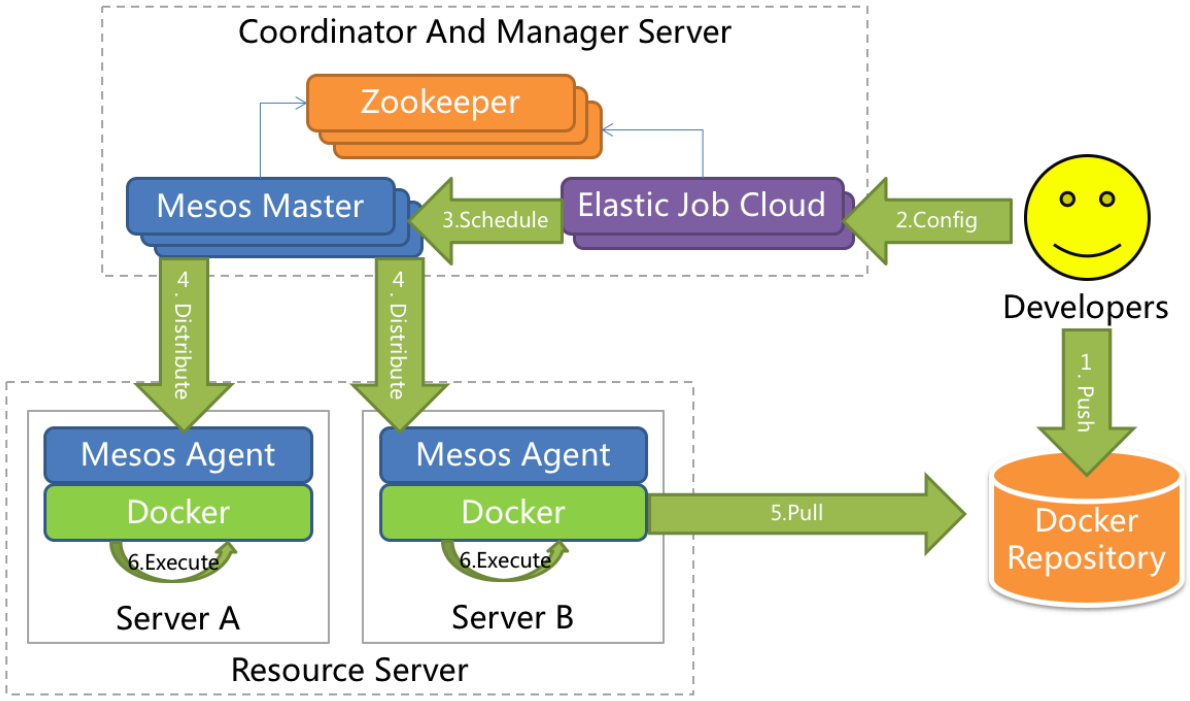

图 2: ElasticJob-Cloud Architecture

	ElasticJob-Lite	*ElasticJob-Cloud*
无中心化	是	否
资源分配	不支持	支持
作业模式	常驻	常驻 + 瞬时
部署依赖	ZooKeeper	ZooKeeper + Mesos

2 功能列表

- 弹性调度
  - 支持任务在分布式场景下的分片和高可用
  - 能够水平扩展任务的吞吐量和执行效率
  - 任务处理能力随资源配备弹性伸缩
- 资源分配
  - 在适合的时间将适合的资源分配给任务并使其生效
  - 相同任务聚合至相同的执行器统一处理
  - 动态调配追加资源至新分配的任务
- 作业治理
  - 失效转移
  - 错过作业重新执行
  - 自诊断修复
- 作业依赖 (TODO)
  - 基于有向无环图（DAG）的作业间依赖
  - 基于有向无环图（DAG）的作业分片间依赖
- 作业开放生态
  - 可扩展的作业类型统一接口
  - 丰富的作业类型库，如数据流、脚本、HTTP、文件、大数据等
  - 易于对接业务作业，能够与 Spring 依赖注入无缝整合
- 可视化管控端
  - 作业管控端
  - 作业执行历史数据追踪
  - 注册中心管理

3 环境要求

3.1 Java

请使用 Java 8 及其以上版本。

3.2 Maven

请使用 Maven 3.5.0 及其以上版本。

3.3 ZooKeeper

请使用 ZooKeeper 3.6.0 及其以上版本。详情参见

3.4 Mesos（仅 ElasticJob-Cloud 使用）

请使用 Mesos 1.1.0 及其兼容版本。详情参见

4 快速入门

本章节以尽量短的时间，为使用者提供最简单的 ElasticJob 的快速入门。

4.1 ElasticJob-Lite

4.1.1 引入 Maven 依赖

```
<dependency>
    <groupId>org.apache.shardingsphere.elasticjob</groupId>
    <artifactId>elasticjob-lite-core</artifactId>
    <version>${latest.release.version}</version>
</dependency>
```

4.1.2 作业开发

```
public class MyJob implements SimpleJob {

    @Override
    public void execute(ShardingContext context) {
        switch (context.getShardingItem()) {
            case 0:
                // do something by sharding item 0
                break;
            case 1:
                // do something by sharding item 1
                break;
            case 2:
                // do something by sharding item 2
                break;
            // case n: ...
        }
```

```
    }
}
```

4.1.3 作业配置

```
JobConfiguration jobConfig = JobConfiguration.newBuilder("MyJob", 3).cron("0/5 * *
* * * ?").build();
```

4.1.4 作业调度

```
public class MyJobDemo {

    public static void main(String[] args) {
        new ScheduleJobBootstrap(createRegistryCenter(), new MyJob(),
createJobConfiguration()).schedule();
    }

    private static CoordinatorRegistryCenter createRegistryCenter() {
        CoordinatorRegistryCenter regCenter = new ZookeeperRegistryCenter(new
ZookeeperConfiguration("zk_host:2181", "my-job"));
        regCenter.init();
        return regCenter;
    }

    private static JobConfiguration createJobConfiguration() {
        // 创建作业配置
        // ...
    }
}
```

4.2 ElasticJob-Cloud

4.2.1 引入 Maven 依赖

```
<dependency>
    <groupId>org.apache.shardingsphere.elasticjob</groupId>
    <artifactId>elasticjob-cloud-executor</artifactId>
    <version>${latest.release.version}</version>
</dependency>
```

4.2.2 作业开发

```
public class MyJob implements SimpleJob {

    @Override
    public void execute(ShardingContext context) {
        switch (context.getShardingItem()) {
            case 0:
                // do something by sharding item 0
                break;
            case 1:
                // do something by sharding item 1
                break;
            case 2:
                // do something by sharding item 2
                break;
            // case n: ...
        }
    }
}
```

4.2.3 作业启动

需定义 main 方法并调用 JobBootstrap.execute()，例子如下：

```
public class MyJobDemo {

    public static void main(final String[] args) {
        JobBootstrap.execute(new MyJob());
    }
}
```

4.2.4 作业打包

```
tar -cvf my-job.tar.gz my-job
```

4.2.5 API 鉴权

```
curl -H "Content-Type: application/json" -X POST http://elasticjob_cloud_host:8899/
api/login -d '{"username": "root", "password": "pwd"}'
```

响应体：

```
{"accessToken":"some_token"}
```

4.2.6 作业发布

```
curl -l -H "Content-type: application/json" -H "accessToken: some_token" -X POST -d
'{"appName":"my_app","appURL":"http://app_host:8080/my-job.tar.gz","cpuCount":0.1,
"memoryMB":64.0,"bootstrapScript":"bin/start.sh","appCacheEnable":true,
"eventTraceSamplingCount":0}' http://elasticjob_cloud_host:8899/api/app
```

4.2.7 作业调度

```
curl -l -H "Content-type: application/json" -H "accessToken: some_token" -X POST -d
'{"jobName":"my_job","appName":"my_app","jobExecutionType":"TRANSIENT","cron":"0/5
* * * * ?","shardingTotalCount":3,"cpuCount":0.1,"memoryMB":64.0}' http://
elasticjob_cloud_host:8899/api/job/register
```

5 概念 & 功能

本章节阐述 ElasticJob 相关的概念与功能，更多使用细节请阅读用户手册。

5.1 调度模型

与大部分的作业平台不同，ElasticJob 的调度模型划分为支持线程级别调度的进程内调度 ElasticJob-Lite，和进程级别调度的 ElasticJob-Cloud。

5.1.1 进程内调度

ElasticJob-Lite 是面向进程内的线程级调度框架。通过它，作业能够透明化的与业务应用系统相结合。它能够方便的与 Spring 、Dubbo 等 Java 框架配合使用，在作业中可自由使用 Spring 注入的 Bean，如数据源连接池、Dubbo 远程服务等，更加方便的贴合业务开发。

5.1.2 进程级调度

ElasticJob-Cloud 拥有进程内调度和进程级别调度两种方式。由于 ElasticJob-Cloud 能够对作业服务器的资源进行控制，因此其作业类型可划分为常驻任务和瞬时任务。常驻任务类似于 ElasticJob-Lite，是进程内调度；瞬时任务则完全不同，它充分的利用了资源分配的削峰填谷能力，是进程级的调度，每次任务的会启动全新的进程处理。

5.2 弹性调度

弹性调度是 ElasticJob 最重要的功能，也是这款产品名称的由来。它是一款能够让任务通过分片进行水平扩展的任务处理系统。

5.2.1 分片

ElasticJob 中任务分片项的概念，使得任务可以在分布式的环境下运行，每台任务服务器只运行分配给该服务器的分片。随着服务器的增加或宕机，ElasticJob 会近乎实时的感知服务器数量的变更，从而重新为分布式的任务服务器分配更加合理的任务分片项，使得任务可以随着资源的增加而提升效率。

任务的分布式执行，需要将一个任务拆分为多个独立的任务项，然后由分布式的服务器分别执行某一个或几个分片项。

举例说明，如果作业分为 4 片，用两台服务器执行，则每个服务器分到 2 片，分别负责作业的 50% 的负载，如下图所示。

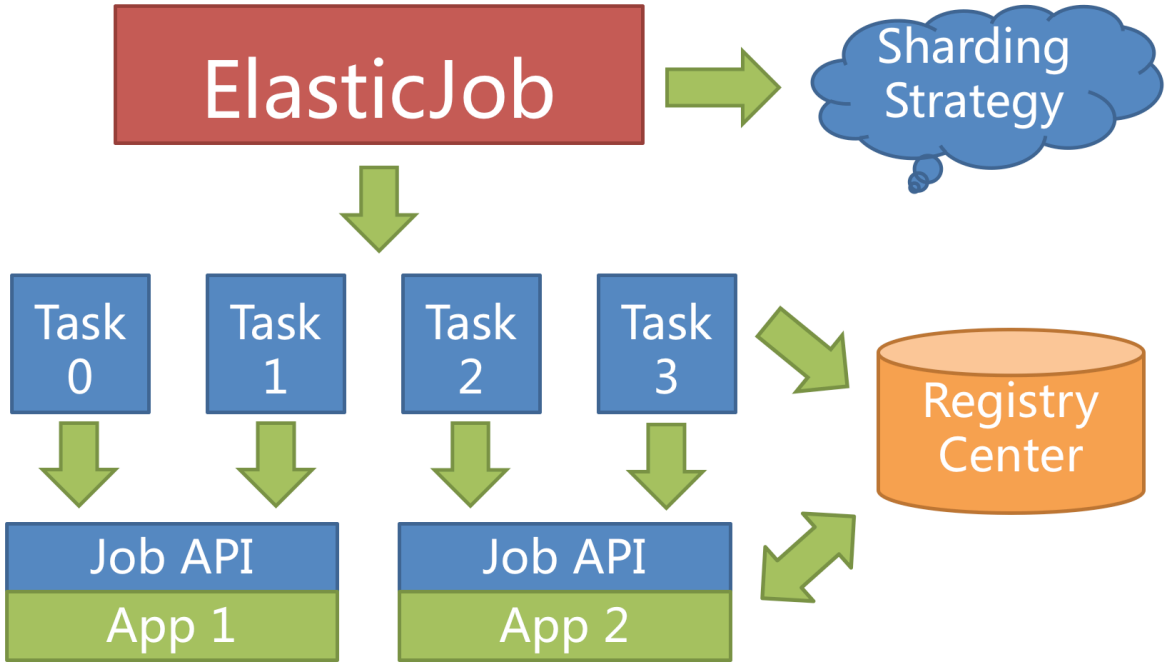

图 1: 分片作业

分片项

ElasticJob 并不直接提供数据处理的功能，而是将分片项分配至各个运行中的作业服务器，开发者需要自行处理分片项与业务的对应关系。分片项为数字，始于 0 而终于分片总数减 1。

个性化分片参数

个性化参数可以和分片项匹配对应关系，用于将分片项的数字转换为更加可读的业务代码。

例如：按照地区水平拆分数据库，数据库 A 是北京的数据；数据库 B 是上海的数据；数据库 C 是广州的数据。如果仅按照分片项配置，开发者需要了解 0 表示北京；1 表示上海；2 表示广州。合理使用个性化参数可以让代码更可读，如果配置为 0= 北京,1= 上海,2= 广州，那么代码中直接使用北京，上海，广州的枚举值即可完成分片项和业务逻辑的对应关系。

5.2.2 资源最大限度利用

ElasticJob 提供最灵活的方式，最大限度的提高执行作业的吞吐量。当新增加作业服务器时，ElasticJob 会通过注册中心的临时节点的变化感知到新服务器的存在，并在下次任务调度的时候重新分片，新的服务器会承载一部分作业分片，如下图所示。

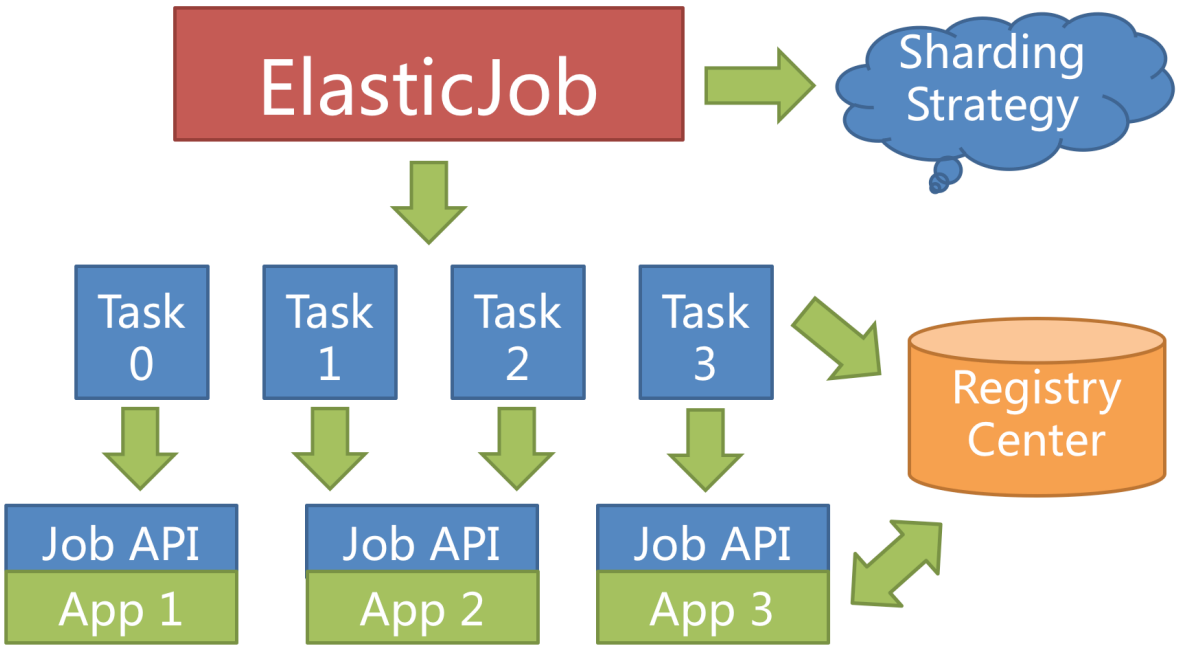

图 2: 作业扩容

将分片项设置为大于服务器的数量，最好是大于服务器倍数的数量，作业将会合理的利用分布式资源，动态的分配分片项。

例如：3 台服务器，分成 10 片，则分片项分配结果为服务器 A = 0,1,2,9；服务器 B = 3,4,5；服务器 C = 6,7,8。如果服务器 C 崩溃，则分片项分配结果为服务器 A = 0,1,2,3,4; 服务器 B = 5,6,7,8,9。在不丢失分片项的情况下，最大限度的利用现有资源提高吞吐量。

5.2.3 高可用

当作业服务器在运行中宕机时，注册中心同样会通过临时节点感知，并将在下次运行时将分片转移至仍存活的服务器，以达到作业高可用的效果。本次由于服务器宕机而未执行完的作业，则可以通过失效转移的方式继续执行。如下图所示。

将分片总数设置为 1，并使用多于 1 台的服务器执行作业，作业将会以 1 主 n 从的方式执行。一旦执行作业的服务器宕机，等待执行的服务器将会在下次作业启动时替补执行。开启失效转移功能效果更好，如果本次作业在执行过程中宕机，备机会立即替补执行。

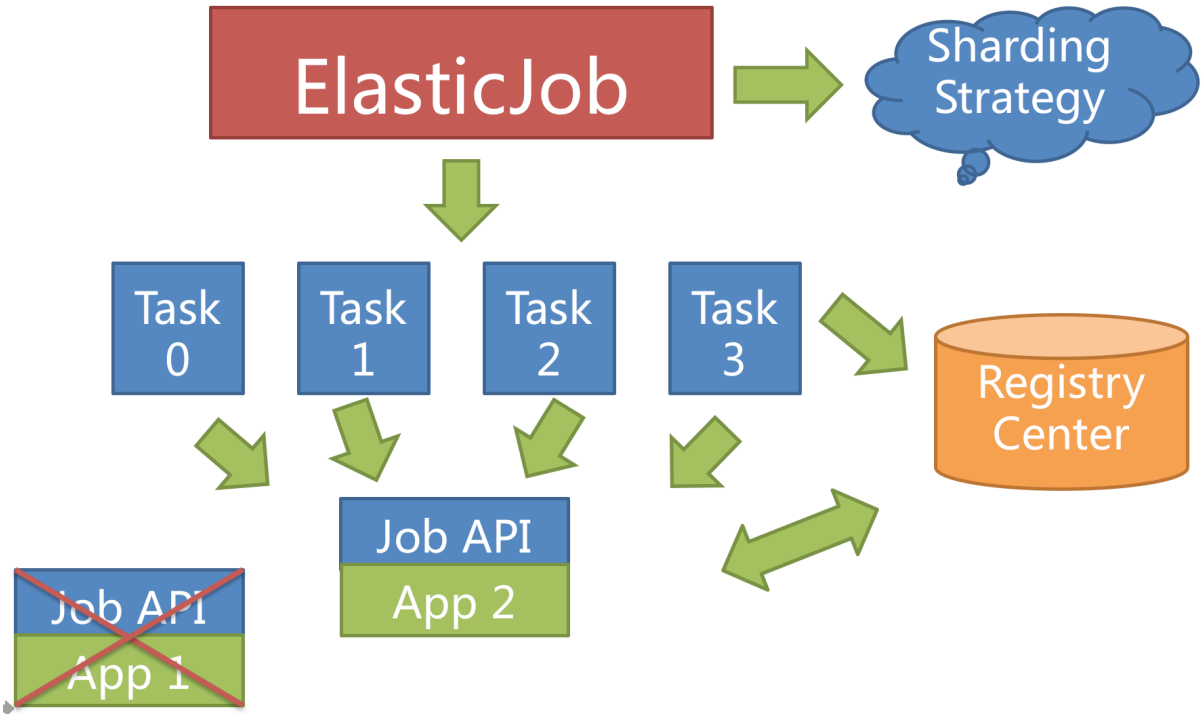

图 3: 作业高可用

5.2.4 ElasticJob-Lite 实现原理

ElasticJob-Lite 并无作业调度中心节点，而是基于部署作业框架的程序在到达相应时间点时各自触发调度。注册中心仅用于作业注册和监控信息存储。而主作业节点仅用于处理分片和清理等功能。

弹性分布式实现

- 第一台服务器上线触发主服务器选举。主服务器一旦下线，则重新触发选举，选举过程中阻塞，只有主服务器选举完成，才会执行其他任务。
- 某作业服务器上线时会自动将服务器信息注册到注册中心，下线时会自动更新服务器状态。
- 主节点选举，服务器上下线，分片总数变更均更新重新分片标记。
- 定时任务触发时，如需重新分片，则通过主服务器分片，分片过程中阻塞，分片结束后才可执行任务。如分片过程中主服务器下线，则先选举主服务器，再分片。
- 通过上一项说明可知，为了维持作业运行时的稳定性，运行过程中只会标记分片状态，不会重新分片。分片仅可能发生在下次任务触发前。
- 每次分片都会按服务器 IP 排序，保证分片结果不会产生较大波动。
- 实现失效转移功能，在某台服务器执行完毕后主动抓取未分配的分片，并且在某台服务器下线后主动寻找可用的服务器执行任务。

注册中心数据结构

注册中心在定义的命名空间下，创建作业名称节点，用于区分不同作业，所以作业一旦创建则不能修改作业名称，如果修改名称将视为新的作业。作业名称节点下又包含 5 个数据子节点，分别是 config，instances，sharding，servers 和 leader。

config 节点

作业配置信息，以 YAML 格式存储。

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。作业运行实例主键由作业运行服务器的 IP 地址和 PID 构成。作业运行实例主键均为临时节点，当作业实例上线时注册，下线时自动清理。注册中心监控这些节点的变化来协调分布式作业的分片以及高可用。可在作业运行实例节点写入 TRIGGER 表示该实例立即执行一次。

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。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。节点详细信息说明：

子节点名	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

servers 节点

作业服务器信息，子节点是作业服务器的 IP 地址。可在 IP 地址节点写入 DISABLED 表示该服务器禁用。在新的云原生架构下，servers 节点大幅弱化，仅包含控制服务器是否可以禁用这一功能。为了更加纯粹的实现作业核心，servers 功能未来可能删除，控制服务器是否禁用的能力应该下放至自动化部署系统。

leader 节点

作业服务器主节点信息，分为 election，sharding 和 failover 三个子节点。分别用于主节点选举，分片和失效转移处理。

leader 节点是内部使用的节点，如果对作业框架原理不感兴趣，可不关注此节点。

子节点名	临时节点	描述
election:raw-latex:‘ instance‘	是	主节点服务器 IP 地址一旦该节点被删除将会触发重新选举重新选举的过程中一切主节点相关的操作都将阻塞
electio n:raw-late x:*latch*	否	主节点选举的分布式锁为 curator 的分布式锁使用
s harding:ra w-latex:*necessary*	否	是否需要重新分片的标记如果分片总数变化，或作业服务器节点上下线或启用/禁用，以及主节点选举，会触发设置重分片标记作业在下次执行时使用主节点重新分片，且中间不会被打断作业执行时不会触发分片
sh arding:raw - latex:*p rocessing*	是	主节点在分片时持有的节点如果有此节点，所有的作业执行都将阻塞，直至分片结束主节点分片结束或主节点崩溃会删除此临时节点
failove r:raw-late x:*items* :raw-latex : 分片项	否	一旦有作业崩溃，则会向此节点记录当有空闲作业服务器时，会从此节点抓取需失效转移的作业项
failov er:raw-lat ex:*items ‘:raw-late x:*latch‘	否	分配失效转移分片项时占用的分布式锁为 curator 的分布式锁使用

流程图

作业启动

作业执行

5.3 资源分配

资源分配功能为 ElasticJob-Cloud 所特有的功能。

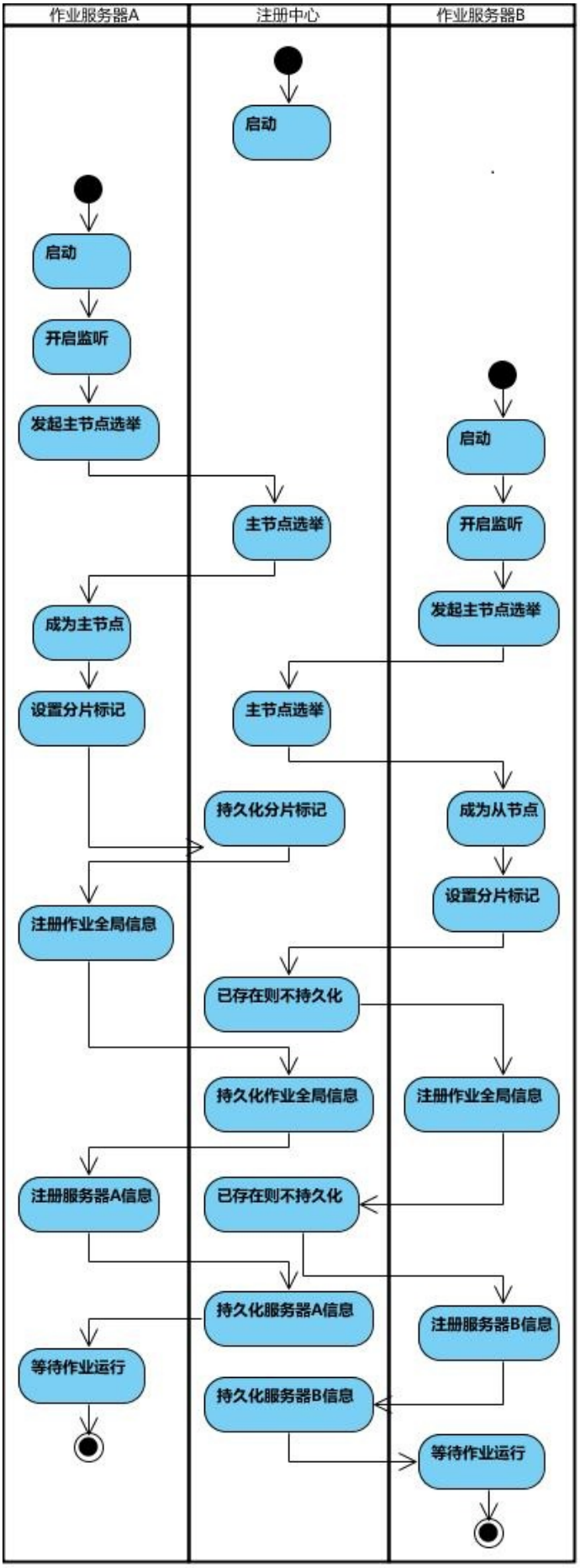
作业服务器A
注册中心
作业服务器B
启动
启动
开启监听
发起主节点选举
主节点选举
成为主节点
设置分片标记
持久化分片标记
注册作业全局信息
已存在则不持久化
持久化作业全局信息
注册服务器A信息
已存在则不持久化
持久化服务器A信息
等待作业运行
持久化服务器B信息
启动
开启监听
发起主节点选举
主节点选举
成为从节点
设置分片标记
注册作业全局信息
注册服务器B信息
等待作业运行

图 4: 作业启动

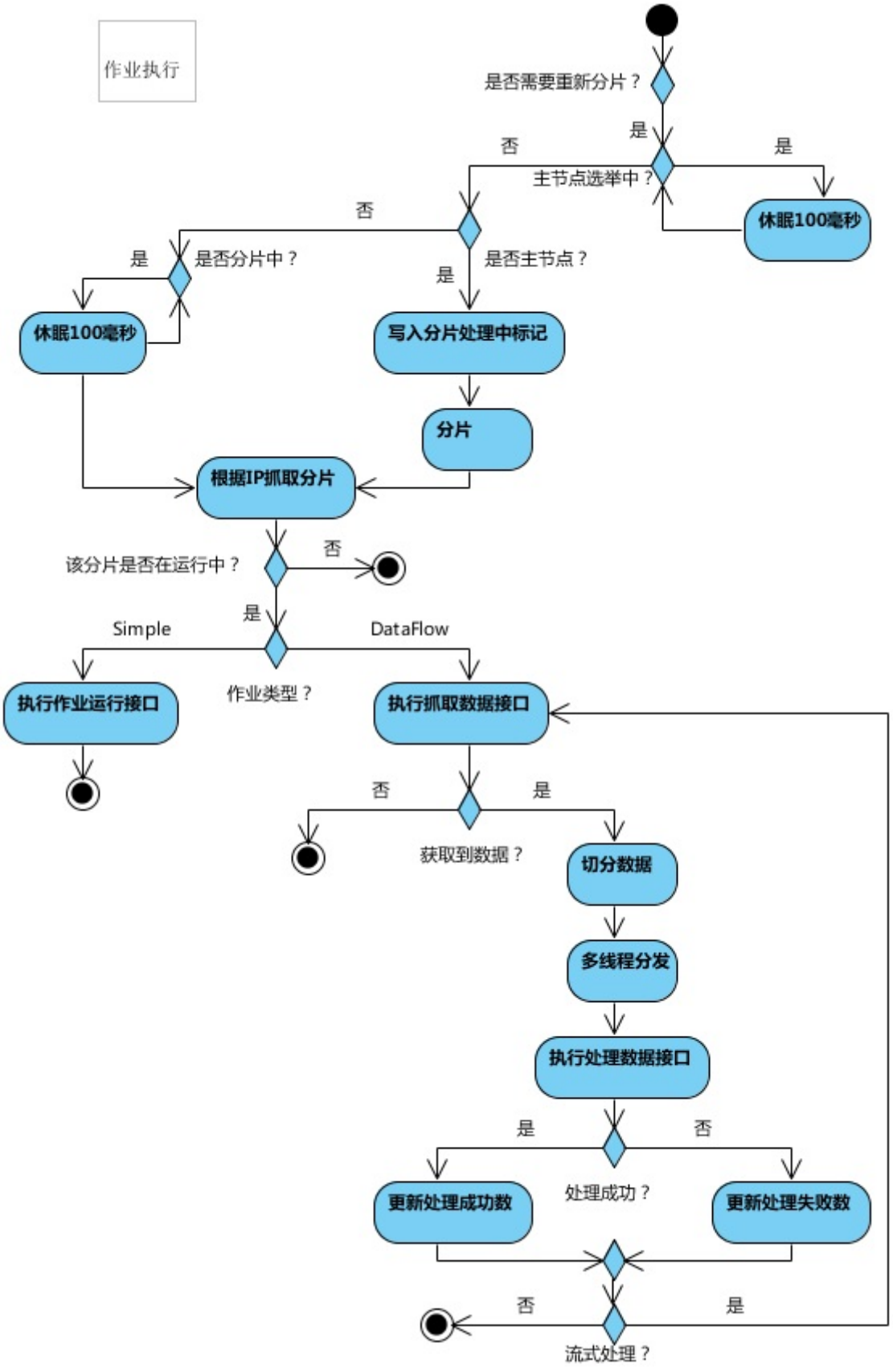
作业执行
是否需要重新分片？
是
否
是
主节点选举中？
休眠100毫秒
否
是
是否分片中？
是否主节点？
是
休眠100毫秒
写入分片处理中标记
分片
根据IP抓取分片
该分片是否在运行中？
否
是
Simple
DataFlow
作业类型？
执行作业运行接口
执行抓取数据接口
否
是
获取到数据？
切分数据
多线程分发
执行处理数据接口
是
否
处理成功？
更新处理成功数
更新处理失败数
否
是
流式处理？

图 5: 作业执行

5.3.1 作业运行模式

ElasticJob-Cloud 分为瞬时作业和常驻作业 2 种运行模式。

瞬时作业

在每一次作业执行完毕后立刻释放资源，保证利用现有资源错峰执行。资源分配和容器启动均占用一定时长，且作业执行时资源不一定充足，因此作业执行会有延迟。瞬时作业适用于间隔时间长，资源消耗多且对执行时间无严格要求的作业。

常驻作业

无论在运行时还是等待运行时，均一直占用分配的资源，可节省过多容器启动和资源分配的开销，适用于间隔时间短，资源需求量稳定的作业。

5.3.2 调度器

ElasticJob-Cloud 基于 Mesos 的 Framework 开发，用于资源调度和应用分发，需要独立启动并提供服务。

5.3.3 作业应用

指作业打包部署后的应用，描述了作业启动需要用到的 CPU、内存、启动脚本及应用下载路径等基本信息。每个作业应用可以包含一个或多个作业。

5.3.4 作业

即实际运行的具体任务，和 ElasticJob-Lite 共用同样的作业生态。在注册作业之前必须先注册作业应用。

5.3.5 资源

指作业启动或运行需要用到的 CPU、内存。配置在作业应用维度表示整个应用启动需要用的资源；配置在作业维度表示每个作业运行需要的资源。作业启动需要的资源为指定作业应用需要的资源与作业需要资源的总和。

5.4 失效转移

ElasticJob 不会在本次执行过程中进行重新分片，而是等待下次调度之前才开启重新分片流程。当作业执行过程中服务器宕机，失效转移允许将该次未完成的任务在另一作业节点上补偿执行。

5.4.1 概念

失效转移是当前执行作业的临时补偿执行机制，在下次作业运行时，会通过重分片对当前作业分配进行调整。举例说明，若作业以每小时为间隔执行，每次执行耗时 30 分钟。如下如图所示。

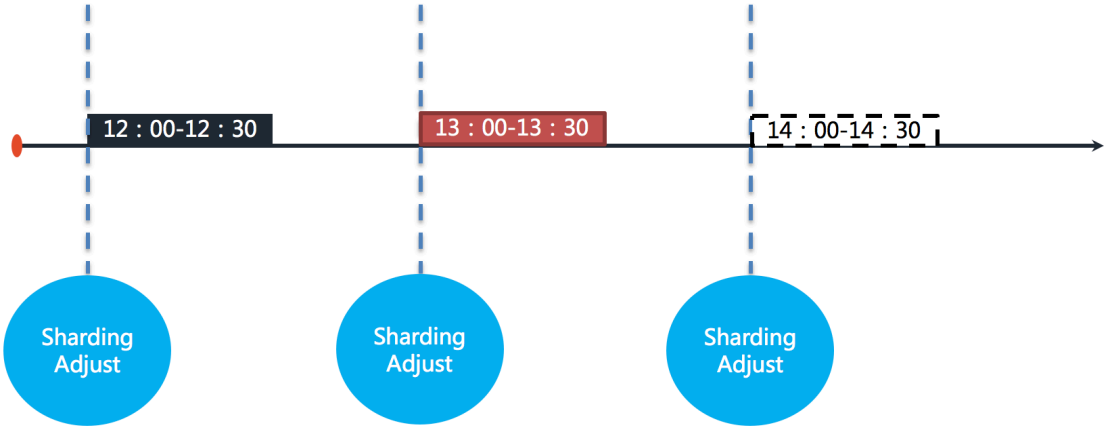

图 6: 定时作业

图中表示作业分别于 12:00，13:00 和 14:00 执行。图中显示的当前时间点为 13:00 的作业执行中。

如果作业的其中一个分片服务器在 13:10 的时候宕机，那么剩余的 20 分钟应该处理的业务未得到执行，并且需要在 14:00 时才能再次开始执行下一次作业。也就是说，在不开启失效转移的情况下，位于该分片的作业有 50 分钟空档期。如下如图所示。

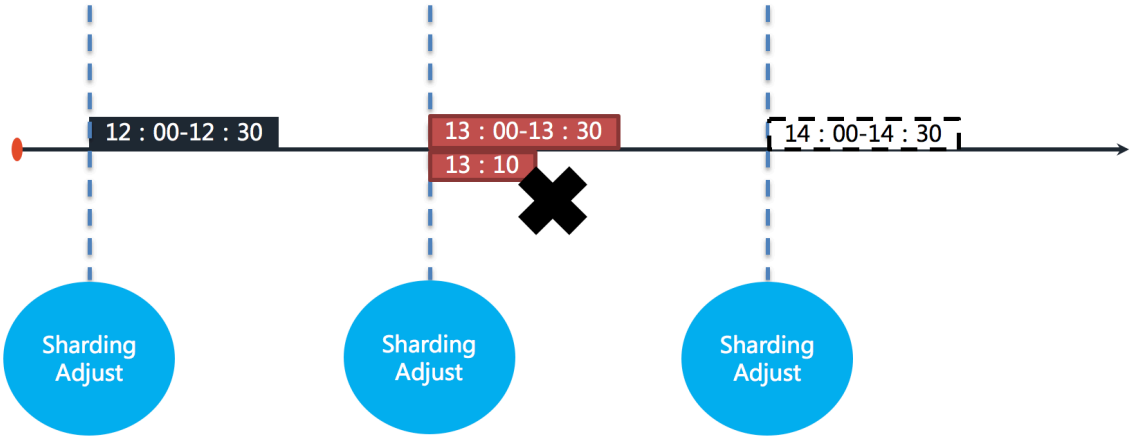

图 7: 作业宕机

在开启失效转移功能之后，ElasticJob 的其他服务器能够在感知到宕机的作业服务器之后，补偿执行该分片作业。如下图所示。

在资源充足的情况下，作业仍然能够在 13:30 完成执行。

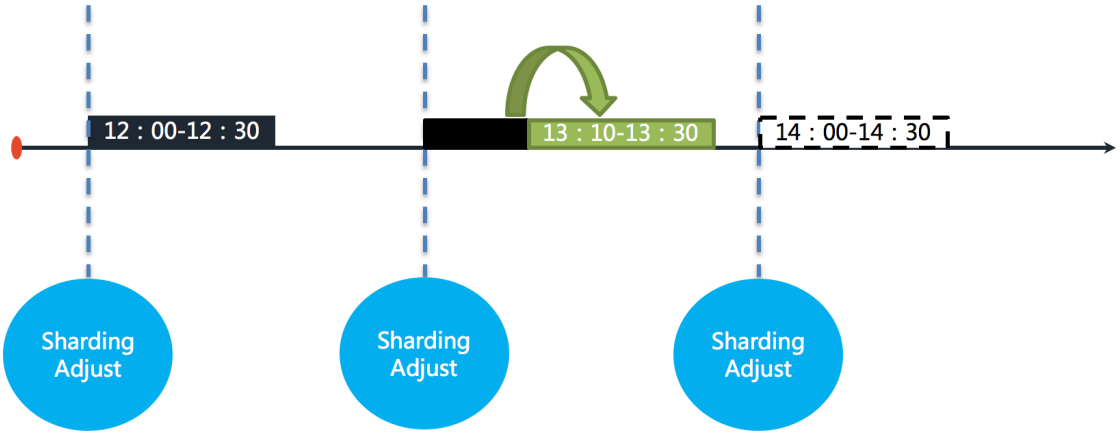

图 8: 补偿执行

5.4.2 执行机制

当作业执行节点宕机时，会触发失效转移流程。ElasticJob 根据触发时的分布式作业执行的不同状况来决定失效转移的执行时机。

通知执行

当其他服务器感知到有失效转移的作业需要处理时，且该作业服务器已经完成了本次任务，则会实时的拉取待失效转移的分片项，并开始补偿执行。也称为实时执行。

问询执行

作业服务在本次任务执行结束后，会向注册中心问询待执行的失效转移分片项，如果有，则开始补偿执行。也称为异步执行。

5.4.3 适用场景

开启失效转移功能，ElasticJob 会监控作业每一分片的执行状态，并将其写入注册中心，供其他节点感知。

在一次运行耗时较长且间隔较长的作业场景，失效转移是提升作业运行实时性的有效手段；对于间隔较短的作业，会产生大量与注册中心的网络通信，对集群的性能产生影响。而且间隔较短的作业并未见得关注单次作业的实时性，可以通过下次作业执行的重分片使所有的分片正确执行，因此不建议短间隔作业开启失效转移。

另外需要注意的是，作业本身的幂等性，是保证失效转移正确性的前提。

5.5 错过任务重执行

ElasticJob 不允许作业在同一时间内叠加执行。当作业的执行时长超过其运行间隔，错过任务重执行能够保证作业在完成上次的任务后继续执行逾期的作业。

5.5.1 概念

错过任务重执行功能可以使逾期未执行的作业在之前作业执行完成之后立即执行。举例说明，若作业以每小时为间隔执行，每次执行耗时 30 分钟。如下如图所示。

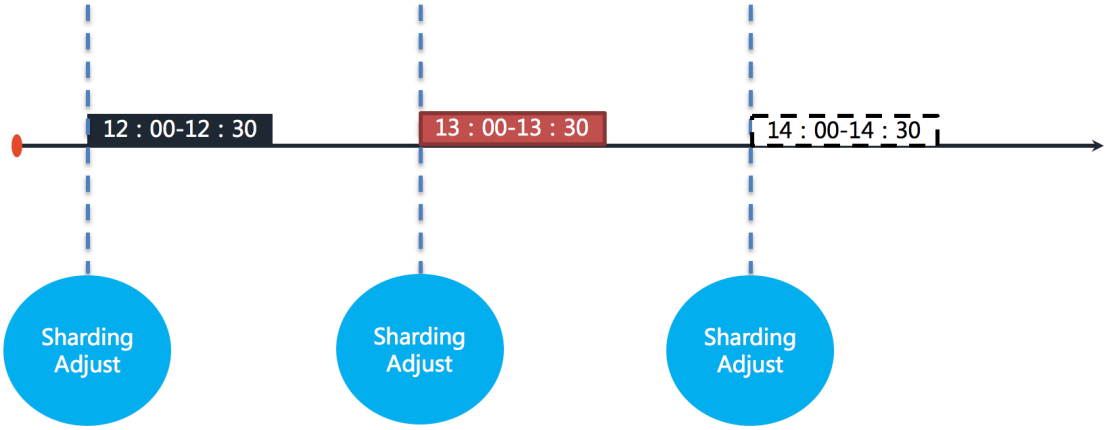

图 9: 定时作业

图中表示作业分别于 12:00，13:00 和 14:00 执行。图中显示的当前时间点为 13:00 的作业执行中。

如果 12：00 开始执行的作业在 13:10 才执行完毕，那么本该由 13:00 触发的作业则错过了触发时间，需要等待至 14:00 的下次作业触发。如下如图所示。

图 10: 错过作业

在开启错过任务重执行功能之后，ElasticJob 将会在上次作业执行完毕后，立刻触发执行错过的作业。如下图所示。

图 11: 错过作业重执行

在 13：00 和 14:00 之间错过的作业将会重新执行。

5.5.2 适用场景

在一次运行耗时较长且间隔较长的作业场景，错过任务重执行是提升作业运行实时性的有效手段；对于未见得关注单次作业的实时性的短间隔的作业来说，开启错过任务重执行并无必要。

5.6 作业开放生态

灵活定制化作业是 ElasticJob 3.x 版本的最重要设计变革。新版本基于 Apache ShardingSphere 可插拔架构的设计理念，打造了全新作业 API。意在使开发者能够更加便捷且相互隔离的方式拓展作业类型，打造 ElasticJob 作业的生态圈。

ElasticJob 提供了对作业的弹性伸缩、分布式治理等功能的同时，并未限定作业的类型。它通过灵活的作业 API，将作业解耦为作业接口和执行器接口。用户可以定制化全新的作业类型，诸如脚本执行、HTTP 服务执行（3.0.0-beta 提供）、大数据类作业、文件类作业等。目前 ElasticJob 内置了简单作业、数据流作业和脚本执行作业，并且完全开放了扩展接口，开发者可以通过 SPI 的方式引入新的作业类型，并且可以便捷的回馈至社区。

5.6.1 作业接口

ElasticJob 的作业可划分为基于 class 类型和基于 type 类型两种。

Class 类型的作业由开发者直接使用，需要由开发者实现该作业接口实现业务逻辑。典型代表：Simple 类型、Dataflow 类型。Type 类型的作业只需提供类型名称即可，开发者无需实现该作业接口，而是通过外置配置的方式使用。典型代表：Script 类型、HTTP 类型。

5.6.2 执行器接口

用于执行用户定义的作业接口，通过 Java 的 SPI 机制织入 ElasticJob 生态。

6 用户手册

本章节详细阐述 ElasticJob 的 2 个相关产品 ElasticJob-Lite 和 ElasticJob-Cloud 的使用。

6.1 ElasticJob-Lite

6.1.1 简介

ElasticJob-Lite 定位为轻量级无中心化解决方案，使用 jar 的形式提供分布式任务的协调服务。

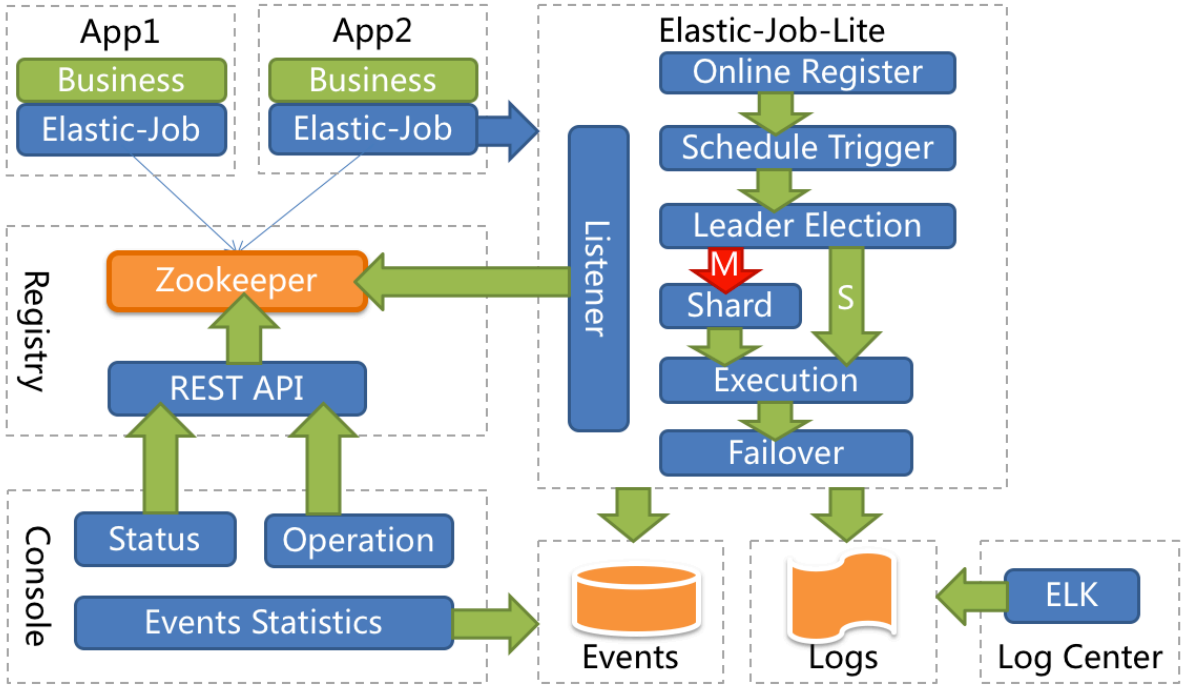

图 1: ElasticJob-Lite Architecture

6.1.2 对比

	ElasticJob-Lite	*ElasticJob-Cloud*
无中心化	是	否
资源分配	不支持	支持
作业模式	常驻	常驻 + 瞬时
部署依赖	ZooKeeper	ZooKeeper + Mesos

ElasticJob-Lite 的优势在于无中心化设计且外部依赖少，适用于资源分配稳定的业务系统。

6.1.3 使用手册

本章节将介绍 ElasticJob-Lite 相关使用。更多使用细节请参见使用示例。

作业 API

ElasticJob-Lite 支持原生 Java、Spring Boot Starter 和 Spring 自定义命名空间 3 种使用方式。本章节将详细介绍他们的使用方式。

作业开发

ElasticJob-Lite 和 ElasticJob-Cloud 提供统一作业接口，开发者仅需对业务作业进行一次开发，之后可根据不同的配置以及部署至不同环境。

ElasticJob 的作业分类基于 class 和 type 两种类型。基于 class 的作业需要开发者自行通过实现接口的方式织入业务逻辑；基于 type 的作业则无需编码，只需要提供相应配置即可。

基于 class 的作业接口的方法参数 `shardingContext` 包含作业配置、片和运行时信息。可通过 `getShardingTotalCount()`, `getShardingItem()` 等方法分别获取分片总数，运行在本作业服务器的分片序列号等。

ElasticJob 目前提供 Simple、Dataflow 这两种基于 class 的作业类型，并提供 Script、HTTP 这两种基于 type 的作业类型，用户可通过实现 SPI 接口自行扩展作业类型。

简单作业

意为简单实现，未经任何封装的类型。需实现 SimpleJob 接口。该接口仅提供单一方法用于覆盖，此方法将定时执行。与 Quartz 原生接口相似，但提供了弹性扩缩容和分片等功能。

```
public class MyElasticJob implements SimpleJob {

    @Override
    public void execute(ShardingContext context) {
        switch (context.getShardingItem()) {
```

```
            case 0:
                // do something by sharding item 0
                break;
            case 1:
                // do something by sharding item 1
                break;
            case 2:
                // do something by sharding item 2
                break;
            // case n: ...
        }
    }
}
```

数据流作业

用于处理数据流，需实现 DataflowJob 接口。该接口提供 2 个方法可供覆盖，分别用于抓取 (fetchData) 和处理 (processData) 数据。

```
public class MyElasticJob implements DataflowJob<Foo> {

    @Override
    public List<Foo> fetchData(ShardingContext context) {
        switch (context.getShardingItem()) {
            case 0:
                List<Foo> data = // get data from database by sharding item 0
                return data;
            case 1:
                List<Foo> data = // get data from database by sharding item 1
                return data;
            case 2:
                List<Foo> data = // get data from database by sharding item 2
                return data;
            // case n: ...
        }
    }

    @Override
    public void processData(ShardingContext shardingContext, List<Foo> data) {
        // process data
        // ...
    }
}
```

流式处理

可通过属性配置 `streaming.process` 开启或关闭流式处理。

如果开启流式处理，则作业只有在 fetchData 方法的返回值为 null 或集合容量为空时，才停止抓取，否则作业将一直运行下去；如果关闭流式处理，则作业只会在每次作业执行过程中执行一次 fetchData 和 processData 方法，随即完成本次作业。

如果采用流式作业处理方式，建议 processData 在处理数据后更新其状态，避免 fetchData 再次抓取到，从而使得作业永不停止。

脚本作业

支持 shell，python，perl 等所有类型脚本。可通过属性配置 `script.command.line` 配置待执行脚本，无需编码。执行脚本路径可包含参数，参数传递完毕后，作业框架会自动追加最后一个参数为作业运行时信息。

例如如下脚本：

```
#!/bin/bash
echo sharding execution context is $*
```

作业运行时将输出：

```
sharding execution context is {"jobName":"scriptElasticDemoJob","shardingTotalCount
":10,"jobParameter":"","shardingItem":0,"shardingParameter":"A"}
```

HTTP 作业（3.0.0-beta 提供）

可通过属性配置 `http.url`,`http.method`,`http.data` 等配置待请求的 http 信息。分片信息以 Header 形式传递，key 为 `shardingContext`，值为 json 格式。

```
public class HttpJobMain {

    public static void main(String[] args) {

        new ScheduleJobBootstrap(regCenter, "HTTP", JobConfiguration.newBuilder(
"javaHttpJob", 1)
                .setProperty(HttpJobProperties.URI_KEY, "http://xxx.com/execute")
                .setProperty(HttpJobProperties.METHOD_KEY, "POST")
                .setProperty(HttpJobProperties.DATA_KEY, "source=ejob")
                .cron("0/5 * * * * ?").shardingItemParameters("0=Beijing").
build()).schedule();
    }
}
```

```
@Controller
@Slf4j
public class HttpJobController {

    @RequestMapping(path = "/execute", method = RequestMethod.POST)
```

```
    public void execute(String source, @RequestHeader String shardingContext) {
        log.info("execute from source : {}, shardingContext : {}", source,
shardingContext);
    }
}
```

execute 接口将输出：

```
execute from source : ejob, shardingContext : {"jobName":"scriptElasticDemoJob",
"shardingTotalCount":3,"jobParameter":"","shardingItem":0,"shardingParameter":
"Beijing"}
```

使用 Java API

作业配置

ElasticJob-Lite 采用构建器模式创建作业配置对象。代码示例如下：

```
JobConfiguration jobConfig = JobConfiguration.newBuilder("myJob", 3).cron("0/5 * *
* * ?").shardingItemParameters("0=Beijing,1=Shanghai,2=Guangzhou").build();
```

作业启动

ElasticJob-Lite 调度器分为定时调度和一次性调度两种类型。每种调度器启动时均需要注册中心配置、作业对象（或作业类型）以及作业配置这 3 个参数。

定时调度

```
public class JobDemo {

    public static void main(String[] args) {
        // 调度基于 class 类型的作业
        new ScheduleJobBootstrap(createRegistryCenter(), new MyJob(),
createJobConfiguration()).schedule();
        // 调度基于 type 类型的作业
        new ScheduleJobBootstrap(createRegistryCenter(), "MY_TYPE",
createJobConfiguration()).schedule();
    }

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

```
    private static JobConfiguration createJobConfiguration() {
        // 创建作业配置
        ...
    }
}
```

一次性调度

```
public class JobDemo {

    public static void main(String[] args) {
        OneOffJobBootstrap jobBootstrap = new
OneOffJobBootstrap(createRegistryCenter(), new MyJob(), createJobConfiguration());
        // 可多次调用一次性调度
        jobBootstrap.execute();
        jobBootstrap.execute();
        jobBootstrap.execute();
    }

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

    private static JobConfiguration createJobConfiguration() {
        // 创建作业配置
        ...
    }
}
```

配置作业导出端口

使用 ElasticJob-Lite 过程中可能会碰到一些分布式问题，导致作业运行不稳定。

由于无法在生产环境调试，通过 dump 命令可以把作业内部相关信息导出，方便开发者调试分析；

导出命令的使用请参见运维指南。

以下示例用于展示如何通过 SnapshotService 开启用于导出命令的监听端口。

```
public class JobMain {

    public static void main(final String[] args) {
        SnapshotService snapshotService = new SnapshotService(regCenter, 9888).
listen();
```

```
    }

    private static CoordinatorRegistryCenter createRegistryCenter() {
        // 创建注册中心
    }
}
```

配置错误处理策略

使用 ElasticJob-Lite 过程中当作业发生异常后，可采用以下错误处理策略。

错误处理策略名称	说明	是否内置	是否默认	是否需要额外配置
记录日志策略	记录作业异常日志，但不中断作业执行	是	是	
抛出异常策略	抛出系统异常并中断作业执行	是		
忽略异常策略	忽略系统异常且不中断作业执行	是		
邮件通知策略	发送邮件消息通知，但不中断作业执行			是
企业微信通知策略	发送企业微信消息通知，但不中断作业执行			是
钉钉通知策略	发送钉钉消息通知，但不中断作业执行			是

记录日志策略

```
public class JobDemo {

    public static void main(String[] args) {
        //  定时调度作业
        new ScheduleJobBootstrap(createRegistryCenter(), new MyJob(),
createScheduleJobConfiguration()).schedule();
        // 一次性调度作业
        new OneOffJobBootstrap(createRegistryCenter(), new MyJob(),
createOneOffJobConfiguration()).execute();
    }

    private static JobConfiguration createScheduleJobConfiguration() {
        // 创建定时作业配置，  并且使用记录日志策略
        return JobConfiguration.newBuilder("myScheduleJob", 3).cron("0/5 * * * * ?
").jobErrorHandlerType("LOG").build();
    }

    private static JobConfiguration createOneOffJobConfiguration() {
        // 创建一次性作业配置，  并且使用记录日志策略
        return JobConfiguration.newBuilder("myOneOffJob", 3).jobErrorHandlerType(
"LOG").build();
```

```
    }

    private static CoordinatorRegistryCenter createRegistryCenter() {
        // 配置注册中心
        ...
    }
}
```

抛出异常策略

```
public class JobDemo {

    public static void main(String[] args) {
        //  定时调度作业
        new ScheduleJobBootstrap(createRegistryCenter(), new MyJob(),
createScheduleJobConfiguration()).schedule();
        // 一次性调度作业
        new OneOffJobBootstrap(createRegistryCenter(), new MyJob(),
createOneOffJobConfiguration()).execute();
    }

    private static JobConfiguration createScheduleJobConfiguration() {
        // 创建定时作业配置,  并且使用抛出异常策略
        return JobConfiguration.newBuilder("myScheduleJob", 3).cron("0/5 * * * * ?
").jobErrorHandlerType("THROW").build();
    }

    private static JobConfiguration createOneOffJobConfiguration() {
        // 创建一次性作业配置,  并且使用抛出异常策略
        return JobConfiguration.newBuilder("myOneOffJob", 3).jobErrorHandlerType(
"THROW").build();
    }

    private static CoordinatorRegistryCenter createRegistryCenter() {
        // 配置注册中心
        ...
    }
}
```

忽略异常策略

```
public class JobDemo {

    public static void main(String[] args) {
        //  定时调度作业
        new ScheduleJobBootstrap(createRegistryCenter(), new MyJob(),
createScheduleJobConfiguration()).schedule();
        // 一次性调度作业
        new OneOffJobBootstrap(createRegistryCenter(), new MyJob(),
createOneOffJobConfiguration()).execute();
    }

    private static JobConfiguration createScheduleJobConfiguration() {
        // 创建定时作业配置,  并且使用忽略异常策略
        return JobConfiguration.newBuilder("myScheduleJob", 3).cron("0/5 * * * * ?
").jobErrorHandlerType("IGNORE").build();
    }

    private static JobConfiguration createOneOffJobConfiguration() {
        // 创建一次性作业配置,  并且使用忽略异常策略
        return JobConfiguration.newBuilder("myOneOffJob", 3).jobErrorHandlerType(
"IGNORE").build();
    }

    private static CoordinatorRegistryCenter createRegistryCenter() {
        // 配置注册中心
        ...
    }
}
```

邮件通知策略

请参考 这里 了解更多。

Maven POM:

```
<dependency>
    <groupId>org.apache.shardingsphere.elasticjob</groupId>
    <artifactId>elasticjob-error-handler-email</artifactId>
    <version>${latest.release.version}</version>
</dependency>
```

```
public class JobDemo {

    public static void main(String[] args) {
        //  定时调度作业
```

```
        new ScheduleJobBootstrap(createRegistryCenter(), new MyJob(),
createScheduleJobConfiguration()).schedule();
        // 一次性调度作业
        new OneOffJobBootstrap(createRegistryCenter(), new MyJob(),
createOneOffJobConfiguration()).execute();
    }

    private static JobConfiguration createScheduleJobConfiguration() {
        // 创建定时作业配置,  并且使用邮件通知策略
        JobConfiguration jobConfig = JobConfiguration.newBuilder("myScheduleJob",
3).cron("0/5 * * * * ?").jobErrorHandlerType("EMAIL").build();
        setEmailProperties(jobConfig);
        return jobConfig;

    }

    private static JobConfiguration createOneOffJobConfiguration() {
        // 创建一次性作业配置,  并且使用邮件通知策略
        JobConfiguration jobConfig = JobConfiguration.newBuilder("myOneOffJob", 3).
jobErrorHandlerType("EMAIL").build();
        setEmailProperties(jobConfig);
        return jobConfig;
    }

    private static void setEmailProperties(final JobConfiguration jobConfig) {
        // 设置邮件的配置
        jobConfig.getProps().setProperty(EmailPropertiesConstants.HOST, "host");
        jobConfig.getProps().setProperty(EmailPropertiesConstants.PORT, "465");
        jobConfig.getProps().setProperty(EmailPropertiesConstants.USERNAME,
"username");
        jobConfig.getProps().setProperty(EmailPropertiesConstants.PASSWORD,
"password");
        jobConfig.getProps().setProperty(EmailPropertiesConstants.FROM, "from@xxx.
xx");
        jobConfig.getProps().setProperty(EmailPropertiesConstants.TO, "to1@xxx.xx,
to1@xxx.xx");
    }

    private static CoordinatorRegistryCenter createRegistryCenter() {
        // 配置注册中心
        ...
    }
}
```

企业微信通知策略

请参考 这里 了解更多。

Maven POM:

```
<dependency>
    <groupId>org.apache.shardingsphere.elasticjob</groupId>
    <artifactId>elasticjob-error-handler-wechat</artifactId>
    <version>${latest.release.version}</version>
</dependency>
```

```
public class JobDemo {

    public static void main(String[] args) {
        //  定时调度作业
        new ScheduleJobBootstrap(createRegistryCenter(), new MyJob(),
createScheduleJobConfiguration()).schedule();
        // 一次性调度作业
        new OneOffJobBootstrap(createRegistryCenter(), new MyJob(),
createOneOffJobConfiguration()).execute();
    }

    private static JobConfiguration createScheduleJobConfiguration() {
        // 创建定时作业配置,  并且使用企业微信通知策略
        JobConfiguration jobConfig = JobConfiguration.newBuilder("myScheduleJob",
3).cron("0/5 * * * * ?").jobErrorHandlerType("WECHAT").build();
        setWechatProperties(jobConfig);
        return jobConfig;

    }

    private static JobConfiguration createOneOffJobConfiguration() {
        // 创建一次性作业配置,  并且使用企业微信通知策略
        JobConfiguration jobConfig = JobConfiguration.newBuilder("myOneOffJob", 3).
jobErrorHandlerType("WECHAT").build();
        setWechatProperties(jobConfig);
        return jobConfig;
    }

    private static void setWechatProperties(final JobConfiguration jobConfig) {
        // 设置企业微信的配置
        jobConfig.getProps().setProperty(WechatPropertiesConstants.WEBHOOK, "you_
webhook");
    }

    private static CoordinatorRegistryCenter createRegistryCenter() {
        // 配置注册中心
        ...
```

```
    }
}
```

钉钉通知策略

请参考 这里 了解更多。

Maven POM:

```
<dependency>
    <groupId>org.apache.shardingsphere.elasticjob</groupId>
    <artifactId>elasticjob-error-handler-dingtalk</artifactId>
    <version>${latest.release.version}</version>
</dependency>
```

```
public class JobDemo {

    public static void main(String[] args) {
        //  定时调度作业
        new ScheduleJobBootstrap(createRegistryCenter(), new MyJob(),
createScheduleJobConfiguration()).schedule();
        // 一次性调度作业
        new OneOffJobBootstrap(createRegistryCenter(), new MyJob(),
createOneOffJobConfiguration()).execute();
    }

    private static JobConfiguration createScheduleJobConfiguration() {
        // 创建定时作业配置,  并且使用企业微信通知策略
        JobConfiguration jobConfig = JobConfiguration.newBuilder("myScheduleJob",
3).cron("0/5 * * * * ?").jobErrorHandlerType("DINGTALK").build();
        setDingtalkProperties(jobConfig);
        return jobConfig;

    }

    private static JobConfiguration createOneOffJobConfiguration() {
        // 创建一次性作业配置,  并且使用钉钉通知策略
        JobConfiguration jobConfig = JobConfiguration.newBuilder("myOneOffJob", 3).
jobErrorHandlerType("DINGTALK").build();
        setDingtalkProperties(jobConfig);
        return jobConfig;
    }

    private static void setDingtalkProperties(final JobConfiguration jobConfig) {
        // 设置钉钉的配置
        jobConfig.getProps().setProperty(DingtalkPropertiesConstants.WEBHOOK, "you_
webhook");
```

```
        jobConfig.getProps().setProperty(DingtalkPropertiesConstants.KEYWORD, "you_
keyword");
        jobConfig.getProps().setProperty(DingtalkPropertiesConstants.SECRET, "you_
secret");
    }

    private static CoordinatorRegistryCenter createRegistryCenter() {
        // 配置注册中心
        ...
    }
}
```

使用 Spring Boot Starter

ElasticJob-Lite 提供自定义的 Spring Boot Starter，可以与 Spring Boot 配合使用。基于 ElasticJob Spring Boot Starter 使用 ElasticJob ，用户无需手动创建 CoordinatorRegistryCenter、JobBootstrap 等实例，只需实现核心作业逻辑并辅以少量配置，即可利用轻量、无中心化的 ElasticJob 解决分布式调度问题。

作业配置

实现作业逻辑

作业逻辑实现与 ElasticJob 的其他使用方式并没有较大的区别，只需将当前作业注册为 Spring 容器中的 bean。

线程安全问题

Bean 默认是单例的，如果该作业实现会在同一个进程内被创建出多个 `JobBootstrap` 的实例，可以考虑设置 Scope 为 `prototype`。

```
@Component
public class SpringBootDataflowJob implements DataflowJob<Foo> {

    @Override
    public List<Foo> fetchData(final ShardingContext shardingContext) {
        // 获取数据
    }

    @Override
    public void processData(final ShardingContext shardingContext, final List<Foo>
data) {
        // 处理数据
    }
}
```

配置协调服务与作业

在配置文件中指定 ElasticJob 所使用的 Zookeeper。配置前缀为 `elasticjob.reg-center`。

`elasticjob.jobs` 是一个 Map，key 为作业名称，value 为作业类型与配置。Starter 会根据该配置自动创建 `OneOffJobBootstrap` 或 `ScheduleJobBootstrap` 的实例并注册到 Spring 容器中。

配置参考：

```
elasticjob:
  regCenter:
    serverLists: localhost:6181
    namespace: elasticjob-lite-springboot
  jobs:
    dataflowJob:
      elasticJobClass: org.apache.shardingsphere.elasticjob.dataflow.job.
DataflowJob
      cron: 0/5 * * * * ?
      shardingTotalCount: 3
      shardingItemParameters: 0=Beijing,1=Shanghai,2=Guangzhou
    scriptJob:
      elasticJobType: SCRIPT
      cron: 0/10 * * * * ?
      shardingTotalCount: 3
      props:
        script.command.line: "echo SCRIPT Job: "
```

作业启动

定时调度

定时调度作业在 Spring Boot 应用程序启动完成后会自动启动，无需其他额外操作。

一次性调度

一次性调度的作业的执行权在开发者手中，开发者可以在需要调用作业的位置注入 `OneOffJobBootstrap`，通过 `execute()` 方法执行作业。

`OneOffJobBootstrap` bean 的名称通过属性 jobBootstrapBeanName 配置，注入时需要指定依赖的 bean 名称。具体配置请参考配置文档。

```
elasticjob:
  jobs:
    myOneOffJob:
      jobBootstrapBeanName: myOneOffJobBean
      ....
```

```
@RestController
public class OneOffJobController {

    // 通过 "@Resource" 注入
    @Resource(name = "myOneOffJobBean")
    private OneOffJobBootstrap myOneOffJob;

    @GetMapping("/execute")
    public String executeOneOffJob() {
        myOneOffJob.execute();
        return "{\"msg\":\"OK\"}";
    }

    // 通过 "@Autowired" 注入
    @Autowired
    @Qualifier(name = "myOneOffJobBean")
    private OneOffJobBootstrap myOneOffJob2;

    @GetMapping("/execute2")
    public String executeOneOffJob2() {
        myOneOffJob2.execute();
        return "{\"msg\":\"OK\"}";
    }
}
```

配置错误处理策略

使用 ElasticJob-Lite 过程中当作业发生异常后，可采用以下错误处理策略。

错误处理策略名称	说明	是否内置	是否默认	是否需要额外配置
记录日志策略	记录作业异常日志，但不中断作业执行	是	是	
抛出异常策略	抛出系统异常并中断作业执行	是		
忽略异常策略	忽略系统异常且不中断作业执行	是		
邮件通知策略	发送邮件消息通知，但不中断作业执行			是
企业微信通知策略	发送企业微信消息通知，但不中断作业执行			是
钉钉通知策略	发送钉钉消息通知，但不中断作业执行			是

记录日志策略

```
elasticjob:
  regCenter:
    ...
  jobs:
    ...
    jobErrorHandlerType: LOG
```

抛出异常策略

```
elasticjob:
  regCenter:
    ...
  jobs:
    ...
    jobErrorHandlerType: THROW
```

忽略异常策略

```
elasticjob:
  regCenter:
    ...
  jobs:
    ...
    jobErrorHandlerType: IGNORE
```

邮件通知策略

请参考 这里 了解更多。

Maven POM:

```
<dependency>
    <groupId>org.apache.shardingsphere.elasticjob</groupId>
    <artifactId>elasticjob-error-handler-email</artifactId>
    <version>${latest.release.version}</version>
</dependency>
```

```
elasticjob:
  regCenter:
    ...
  jobs:
    ...
```

```
    jobErrorHandlerType: EMAIL
  props:
    email:
      host: host
      port: 465
      username: username
      password: password
      useSsl: true
      subject: ElasticJob error message
      from: from@xxx.xx
      to: to1@xxx.xx,to2@xxx.xx
      cc: cc@xxx.xx
      bcc: bcc@xxx.xx
      debug: false
```

企业微信通知策略

请参考 这里 了解更多。

Maven POM:

```
<dependency>
    <groupId>org.apache.shardingsphere.elasticjob</groupId>
    <artifactId>elasticjob-error-handler-wechat</artifactId>
    <version>${latest.release.version}</version>
</dependency>
```

```
elasticjob:
  regCenter:
    ...
  jobs:
    ...
    jobErrorHandlerType: WECHAT
  props:
    wechat:
      webhook: you_webhook
      connectTimeout: 3000
      readTimeout: 5000
```

钉钉通知策略

请参考 这里 了解更多。

Maven POM:

```
<dependency>
    <groupId>org.apache.shardingsphere.elasticjob</groupId>
    <artifactId>elasticjob-error-handler-dingtalk</artifactId>
    <version>${latest.release.version}</version>
</dependency>
```

```
elasticjob:
  regCenter:
    ...
  jobs:
    ...
    jobErrorHandlerType: DINGTALK
  props:
    dingtalk:
       webhook: you_webhook
       keyword: you_keyword
       secret: you_secret
       connectTimeout: 3000
       readTimeout: 5000
```

使用 Spring 命名空间

ElasticJob-Lite 提供自定义的 Spring 命名空间，可以与 Spring 容器配合使用。开发者能够便捷的在作业中通过依赖注入使用 Spring 容器管理的数据源等对象，并使用占位符从属性文件中取值。

作业配置

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:elasticjob="http://shardingsphere.apache.org/schema/elasticjob"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
                        http://www.springframework.org/schema/beans/spring-beans.
xsd
                        http://shardingsphere.apache.org/schema/elasticjob
                        http://shardingsphere.apache.org/schema/elasticjob/
elasticjob.xsd
                        ">
    <!--配置作业注册中心 -->
    <elasticjob:zookeeper id="regCenter" server-lists="yourhost:2181" namespace=
"my-job" base-sleep-time-milliseconds="1000" max-sleep-time-milliseconds="3000"
max-retries="3" />
```

```
    <!-- 配置作业 Bean -->
    <bean id="myJob" class="xxx.MyJob">
        <property name="fooService" ref="xxx.FooService" />
    </bean>

    <!-- 配置基于 class 的作业调度 -->
    <elasticjob:job id="${myJob.id}" job-ref="myJob" registry-center-ref="regCenter
" sharding-total-count="${myJob.shardingTotalCount}" cron="${myJob.cron}" />

    <!-- 配置基于 type 的作业调度 -->
    <elasticjob:job id="${myScriptJob.id}" job-type="SCRIPT" registry-center-ref=
"regCenter" sharding-total-count="${myScriptJob.shardingTotalCount}" cron="$
{myScriptJob.cron}">
        <props>
            <prop key="script.command.line">${myScriptJob.scriptCommandLine}</prop>
        </props>
    </elasticjob:job>
</beans>
```

作业启动

定时调度

将配置 Spring 命名空间的 xml 通过 Spring 启动，作业将自动加载。

一次性调度

一次性调度的作业的执行权在开发者手中，开发者可以在需要调用作业的位置注入 OneOffJobBootstrap，通过 execute() 方法执行作业。

```
<bean id="oneOffJob" class="org.apache.shardingsphere.elasticjob.lite.example.job.
simple.SpringSimpleJob" />
<elasticjob:job id="oneOffJobBean" job-ref="oneOffJob" ...  />
```

```
public final class SpringMain {
    public static void main(final String[] args) {
        ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext("classpath:META-INF/application-context.xml");
        OneOffJobBootstrap oneOffJobBootstrap = context.getBean("oneOffJobBean",
OneOffJobBootstrap.class);
        oneOffJobBootstrap.execute();
    }
}
```

配置作业导出端口

使用 ElasticJob-Lite 过程中可能会碰到一些分布式问题，导致作业运行不稳定。

由于无法在生产环境调试，通过 dump 命令可以把作业内部相关信息导出，方便开发者调试分析；

导出命令的使用请参见运维指南。

以下示例用于展示如何通过 Spring 命名空间开启用于导出命令的监听端口。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:elasticjob="http://shardingsphere.apache.org/schema/elasticjob"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-
beans.xsd
                           http://shardingsphere.apache.org/schema/elasticjob
                           http://shardingsphere.apache.org/schema/elasticjob/
elasticjob.xsd
                         ">
    <!--配置作业注册中心 -->
    <elasticjob:zookeeper id="regCenter" server-lists="yourhost:2181" namespace=
"dd-job" base-sleep-time-milliseconds="1000" max-sleep-time-milliseconds="3000"
max-retries="3" />

    <!--配置任务快照导出服务 -->
    <elasticjob:snapshot id="jobSnapshot" registry-center-ref="regCenter" dump-
port="9999" />
</beans>
```

配置错误处理策略

使用 ElasticJob-Lite 过程中当作业发生异常后，可采用以下错误处理策略。

错误处理策略名称	说明	是否内置	是否默认	是否需要额外配置
记录日志策略	记录作业异常日志，但不中断作业执行	是	是	
抛出异常策略	抛出系统异常并中断作业执行	是		
忽略异常策略	忽略系统异常且不中断作业执行	是		
邮件通知策略	发送邮件消息通知，但不中断作业执行			是
企业微信通知策略	发送企业微信消息通知，但不中断作业执行			是
钉钉通知策略	发送钉钉消息通知，但不中断作业执行			是

以下示例用于展示如何通过 Spring 命名空间配置错误处理策略。

```
<?xml version="1.0" encoding="UTF-8"?>
<elasticjob:job xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:elasticjob="http://shardingsphere.apache.org/schema/elasticjob"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-
beans.xsd
                           http://shardingsphere.apache.org/schema/elasticjob
                           http://shardingsphere.apache.org/schema/elasticjob/
elasticjob.xsd
                         ">
    <!-- 记录日志策略 -->
    <elasticjob:job  ... job-error-handler-type="LOG" />

    <!-- 抛出异常策略 -->
    <elasticjob:job  ... job-error-handler-type="THROW" />

    <!-- 忽略异常策略 -->
    <elasticjob:job  ... job-error-handler-type="IGNORE" />

    <!-- 邮件通知策略 -->
    <elasticjob:job ... job-error-handler-type="EMAIL">
        <props>
            <prop key="email.host">${host}</prop>
            <prop key="email.port">${port}</prop>
            <prop key="email.username">${username}</prop>
            <prop key="email.password">${password}</prop>
            <prop key="email.useSsl">${useSsl}</prop>
            <prop key="email.subject">${subject}</prop>
            <prop key="email.from">${from}</prop>
            <prop key="email.to">${to}</prop>
            <prop key="email.cc">${cc}</prop>
            <prop key="email.bcc">${bcc}</prop>
            <prop key="email.debug">${debug}</prop>
        </props>
    </elasticjob:job>

    <!-- 企业微信通知策略 -->
    <elasticjob:job  ... job-error-handler-type="WECHAT">
        <props>
            <prop key="wechat.webhook">${webhook}</prop>
            <prop key="wechat.connectTimeoutMilliseconds">$
{connectTimeoutMilliseconds}</prop>
            <prop key="wechat.readTimeoutMilliseconds">${readTimeoutMilliseconds}</
prop>
        </props>
    </elasticjob:job>
```

```
    <!-- 钉钉通知策略 -->
    <elasticjob:job  ... job-error-handler-type="DINGTALK">
        <props>
            <prop key="dingtalk.webhook">${webhook}</prop>
            <prop key="dingtalk.keyword">${keyword}</prop>
            <prop key="dingtalk.secret">${secret}</prop>
            <prop key="dingtalk.connectTimeoutMilliseconds">$
{connectTimeoutMilliseconds}</prop>
            <prop key="dingtalk.readTimeoutMilliseconds">${readTimeoutMilliseconds}
</prop>
        </props>
    </elasticjob:job>
</beans>
```

作业监听器

ElasticJob-Lite 提供作业监听器，用于在任务执行前和执行后执行监听的方法。监听器分为每台作业节点均执行的常规监听器和分布式场景中仅单一节点执行的分布式监听器。本章节将详细介绍他们的使用方式。

在作业依赖（DAG）功能开发完成之后，可能会考虑删除作业监听器功能。

监听器开发

常规监听器

若作业处理作业服务器的文件，处理完成后删除文件，可考虑使用每个节点均执行清理任务。此类型任务实现简单，且无需考虑全局分布式任务是否完成，应尽量使用此类型监听器。

```
public class MyJobListener implements ElasticJobListener {

    @Override
    public void beforeJobExecuted(ShardingContexts shardingContexts) {
        // do something ...
    }

    @Override
    public void afterJobExecuted(ShardingContexts shardingContexts) {
        // do something ...
    }

    @Override
    public String getType() {
        return "simpleJobListener";
    }
}
```

分布式监听器

若作业处理数据库数据，处理完成后只需一个节点完成数据清理任务即可。此类型任务处理复杂，需同步分布式环境下作业的状态同步，提供了超时设置来避免作业不同步导致的死锁，应谨慎使用。

```
public class MyDistributeOnceJobListener extends
AbstractDistributeOnceElasticJobListener {

    private static final long startTimeoutMills = 3000;
    private static final long completeTimeoutMills = 3000;

    public MyDistributeOnceJobListener() {
        super(startTimeoutMills, completeTimeoutMills);
    }


    @Override
    public void doBeforeJobExecutedAtLastStarted(ShardingContexts shardingContexts)
{
        // do something ...
    }

    @Override
    public void doAfterJobExecutedAtLastCompleted(ShardingContexts
shardingContexts) {
        // do something ...
    }

    @Override
    public String getType() {
        return "distributeOnceJobListener";
    }
}
```

添加 SPI 实现

将 JobListener 实现添加至 infra-common 下 resources/META-INF/services/org.apache.shardingsphere.elasticjob.infra.l

使用 Java API

常规监听器

```
public class JobMain {

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

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

    private static JobConfiguration createJobConfiguration() {
        JobConfiguration jobConfiguration = JobConfiguration.newBuilder("test", 2)
                                .jobListenerTypes("simpleListener",
"distributeListener").build();
    }
}
```

分布式监听器

```
public class JobMain {

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

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

    private static JobConfiguration createJobConfiguration() {
        JobConfiguration jobConfiguration = JobConfiguration.newBuilder("test", 2)
                        .jobListenerTypes("simpleListener", "distributeListener").
build();
    }
```

```
}
```

使用 Spring 命名空间

监听器配置

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:elasticjob="http://shardingsphere.apache.org/schema/elasticjob"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
                        http://www.springframework.org/schema/beans/spring-beans.
xsd
                        http://shardingsphere.apache.org/schema/elasticjob
                        http://shardingsphere.apache.org/schema/elasticjob/
elasticjob.xsd
                        ">
    <!--配置作业注册中心 -->
    <elasticjob:zookeeper id="regCenter" server-lists="yourhost:2181" namespace=
"my-job" base-sleep-time-milliseconds="1000" max-sleep-time-milliseconds="3000"
max-retries="3" />

    <!-- 配置作业 Bean -->
    <bean id="myJob" class="xxx.MyJob" />

    <elasticjob:job id="${myJob.id}" job-ref="myJob" registry-center-ref="regCenter
" sharding-total-count="3" cron="0/1 * * * * ?" job-listener-types=
"simpleJobListener,distributeOnceJobListener">
    </elasticjob:job>
</beans>
```

作业启动

将配置 Spring 命名空间的 xml 通过 Spring 启动，作业将自动加载。

事件追踪

ElasticJob 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。目前提供了基于关系型数据库的事件订阅方式记录事件，开发者也可以通过 SPI 自行扩展。

使用 Java API

ElasticJob-Lite 在配置中提供了 TracingConfiguration, 目前支持数据库方式配置。开发者也可以通过 SPI 自行扩展。

```
// 初始化数据源
DataSource dataSource = ...;
// 定义日志数据库事件溯源配置
TracingConfiguration tracingConfig = new TracingConfiguration<>("RDB", dataSource);
// 初始化注册中心
CoordinatorRegistryCenter regCenter = ...;
// 初始化作业配置
JobConfiguration jobConfig = ...;
jobConfig.getExtraConfigurations().add(tracingConfig);
new ScheduleJobBootstrap(regCenter, jobConfig).schedule();
```

使用 Spring Boot Starter

ElasticJob-Lite 的 Spring Boot Starter 集成了 TracingConfiguration 自动配置, 开发者只需注册一个 DataSource 到 Spring 容器中并在配置文件指定事件追踪数据源类型, Starter 就会自动创建一个 TracingConfiguration 实例并注册到 Spring 容器中。

引入 Maven 依赖

引入 spring-boot-starter-jdbc 注册数据源或自行创建一个 DataSource Bean。

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
    <version>${springboot.version}</version>
</dependency>
```

配置

```
spring:
  datasource:
    url: jdbc:h2:mem:job_event_storage
    driver-class-name: org.h2.Driver
    username: sa
    password:

elasticjob:
  tracing:
    type: RDB
```

作业启动

指定事件追踪数据源类型为 RDB，TracingConfiguration 会自动注册到容器中，如果与 elasticjob-lite-spring-boot-starter 配合使用，开发者无需进行其他额外的操作，作业启动器会自动使用创建的 TracingConfiguration。

使用 Spring 命名空间

引入 Maven 依赖

引入 elasticjob-lite-spring

```
<dependency>
    <groupId>org.apache.shardingsphere.elasticjob</groupId>
    <artifactId>elasticjob-lite-spring-namespace</artifactId>
    <version>${elasticjob.latest.version}</version>
</dependency>
```

配置

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:elasticjob="http://shardingsphere.apache.org/schema/elasticjob"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
                        http://www.springframework.org/schema/beans/spring-beans.
xsd
                        http://shardingsphere.apache.org/schema/elasticjob
                        http://shardingsphere.apache.org/schema/elasticjob/
elasticjob.xsd
                        ">
    <!--配置作业注册中心 -->
    <elasticjob:zookeeper id="regCenter" server-lists="yourhost:2181" namespace=
"my-job" base-sleep-time-milliseconds="1000" max-sleep-time-milliseconds="3000"
max-retries="3" />

    <!-- 配置作业 Bean -->
    <bean id="myJob" class="xxx.MyJob" />

    <!-- 配置数据源 -->
    <bean id="tracingDataSource" class="org.apache.commons.dbcp2.BasicDataSource"
destroy-method="close">
        <property name="driverClassName" value="${driver.class.name}" />
        <property name="url" value="${url}" />
        <property name="username" value="${username}" />
        <property name="password" value="${password}" />
```

```
    </bean>
    <!-- 配置事件追踪 -->
    <elasticjob:rdb-tracing id="elasticJobTrace" data-source-ref=
"elasticJobTracingDataSource" />

    <!-- 配置作业 -->
    <elasticjob:job id="${myJob.id}" job-ref="myJob" registry-center-ref="regCenter
" tracing-ref="elasticJobTrace" sharding-total-count="3" cron="0/1 * * * * ?" />
</beans>
```

作业启动

将配置 Spring 命名空间的 xml 通过 Spring 启动，作业将自动加载。

表结构说明

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。

JOB_EXECUTION_LOG 字段含义

字段名称	字段类型	是否必填	描述
id	VARCHAR(40)	是	主键
job_name	VARCHAR(100)	是	作业名称
task_id	VARCHAR(1000)	是	任务名称, 每次作业运行生成新任务
hostname	VARCHAR(255)	是	主机名称
ip	VARCHAR(50)	是	主机 IP
sharding_item	INT	是	分片项
execution_source	VARCHAR(20)	是	作业执行来源。可选值为 NORMAL_TRIGGER, MISFIRE, FAILOVER
failure_cause	VARCHAR(2000)	否	执行失败原因
is_success	BIT	是	是否执行成功
start_time	TIMESTAMP	是	作业开始执行时间
complete_time	TIMESTAMP	否	作业结束执行时间

JOB_EXECUTION_LOG 记录每次作业的执行历史。分为两个步骤：

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

JOB_STATUS_TRACE_LOG 字段含义

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

操作 API

ElasticJob-Lite 提供了 Java API，可以通过直接对注册中心进行操作的方式控制作业在分布式环境下的生命周期。

该模块目前仍处于孵化状态。

配置类 API

类　　名　　称：`org.apache.shardingsphere.elasticjob.lite.lifecycle.api.JobConfigurationAPI`

获取作业配置

方法签名：YamlJobConfiguration getJobConfiguration(String jobName)

- **Parameters:**
  - jobName —作业名称
- **Returns:** 作业配置对象

更新作业配置

方法签名：void updateJobConfiguration(YamlJobConfiguration yamlJobConfiguration)

- **Parameters:**
  - jobConfiguration —作业配置对象

删除作业设置

方法签名：void removeJobConfiguration(String jobName)

- **Parameters:**
  - jobName —作业名称

操作类 API

类名称：`org.apache.shardingsphere.elasticjob.lite.lifecycle.api.JobOperateAPI`

触发作业执行

作业在不与当前运行中作业冲突的情况下才会触发执行，并在启动后自动清理此标记。

方法签名：void trigger(Optional jobName, Optional serverIp)

- **Parameters:**
  - jobName —作业名称
  - serverIp —作业服务器 IP 地址

禁用作业

禁用作业将会导致分布式的其他作业触发重新分片。

方法签名：void disable(Optional jobName, Optional serverIp)

- **Parameters:**
  - jobName —作业名称
  - serverIp —作业服务器 IP 地址

启用作业

方法签名：void enable(Optional jobName, Optional serverIp)

- **Parameters:**
  - jobName —作业名称
  - serverIp —作业服务器 IP 地址

停止调度作业

方法签名：void shutdown(Optional jobName, Optional serverIp)

- **Parameters:**
  - jobName —作业名称
  - serverIp —作业服务器 IP 地址

删除作业

方法签名：void remove(Optional jobName, Optional serverIp)

- **Parameters:**
  - jobName —作业名称
  - serverIp —作业服务器 IP 地址

操作分片的 API

类　　名　　称：`org.apache.shardingsphere.elasticjob.lite.lifecycle.api.ShardingOperateAPI`

禁用作业分片

方法签名：void disable(String jobName, String item)

- **Parameters:**
  - jobName —作业名称
  - item —作业分片项

启用作业分片

方法签名：void enable(String jobName, String item)

- **Parameters:**
  - jobName —作业名称
  - item —作业分片项

作业统计 API

类　　名　　称：`org.apache.shardingsphere.elasticjob.lite.lifecycle.api.JobStatisticsAPI`

获取作业总数

方法签名：int getJobsTotalCount()

- **Returns:** 作业总数

获取作业简明信息

方法签名：JobBriefInfo getJobBriefInfo(String jobName)

- **Parameters:**
  - jobName —作业名称
- **Returns:** 作业简明信息

获取所有作业简明信息

方法签名：Collection getAllJobsBriefInfo()

- **Returns:** 作业简明信息集合

获取该 IP 下所有作业简明信息

方法签名：Collection getJobsBriefInfo(String ip)

- **Parameters:**
  - ip —服务器 IP
- **Returns:** 作业简明信息集合

作业服务器状态展示 API

类　　名　　称：`org.apache.shardingsphere.elasticjob.lite.lifecycle.api.ServerStatisticsAPI`

获取作业服务器总数

方法签名：int getServersTotalCount()

- **Returns:** 作业服务器总数

获取所有作业服务器简明信息

方法签名：Collection getAllServersBriefInfo()

- **Returns:** 作业服务器简明信息集合

作业分片状态展示 API

类　　名　　称：`org.apache.shardingsphere.elasticjob.lite.lifecycle.api.ShardingStatisticsAPI`

获取作业分片信息集合

方法签名：Collection getShardingInfo(String jobName)

- **Parameters:**
  - jobName —作业名称
- **Returns:** 作业分片信息集合

6.1.4 配置手册

通过配置可以快速清晰的理解 ElasticJob-Lite 所提供的功能。

本章节是 ElasticJob-Lite 的配置参考手册，需要时可当做字典查阅。

ElasticJob-Lite 提供了 3 种配置方式，用于不同的使用场景。

注册中心配置项

可配置属性

属性名	类型	缺省值	描述
serverLists	String		连接 ZooKeeper 服务器的列表
namespace	String		ZooKeeper 的命名空间
baseS leepTimeMilliseconds	int	1000	等待重试的间隔时间的初始毫秒数
maxS leepTimeMilliseconds	String	3000	等待重试的间隔时间的最大毫秒数
maxRetries	String	3	最大重试次数
sessio nTimeoutMilliseconds	boolean	60000	会话超时毫秒数
connectio nTimeoutMilliseconds	boolean	15000	连接超时毫秒数
digest	String	无需验证	连接 ZooKeeper 的权限令牌

核心配置项说明

serverLists:

包括 IP 地址和端口号，多个地址用逗号分隔，如: host1:2181,host2:2181

作业配置项

可配置属性

属性名	类型	缺省值	描述
jobName	String		作业名称
sha rdingTotalCount	int		作业分片总数
cron	String		CRON 表达式，用于控制作业触发时间
timeZone	String		CRON 的时区设置
shardin gItemParameters	String		个性化分片参数
jobParameter	String		作业自定义参数
m onitorExecution	boolean	true	监控作业运行时状态
failover	boolean	false	是否开启任务执行失效转移
misfire	boolean	true	是否开启错过任务重新执行
max TimeDiffSeconds	int	-1（不检查）	最大允许的本机与注册中心的时间误差秒数
reconcile IntervalMinutes	int	10	修复作业服务器不一致状态服务调度间隔分钟
jobShard ingStrategyType	String	AVG_ALLOCATION	作业分片策略类型
jobExecutorSer viceHandlerType	String	CPU	作业线程池处理策略
jobE rrorHandlerType	String		作业错误处理策略
description	String		作业描述信息
props	Properties		作业属性配置信息
disabled	boolean	false	作业是否禁止启动
overwrite	boolean	false	本地配置是否可覆盖注册中心配置

核心配置项说明

shardingItemParameters:

分片序列号和参数用等号分隔，多个键值对用逗号分隔。分片序列号从 0 开始，不可大于或等于作业分片总数。如：0=a,1=b,2=c

jobParameter:

可通过传递该参数为作业调度的业务方法传参，用于实现带参数的作业例：每次获取的数据量、作业实例从数据库读取的主键等。

monitorExecution:

每次作业执行时间和间隔时间均非常短的情况，建议不监控作业运行时状态以提升效率。因为是瞬时状态，所以无必要监控。请用户自行增加数据堆积监控。并且不能保证数据重复选取，应在作业中实现幂等性。每次作业执行时间和间隔时间均较长的情况，建议监控作业运行时状态，可保证数据不会重复选取。

maxTimeDiffSeconds:

如果时间误差超过配置秒数则作业启动时将抛异常。

reconcileIntervalMinutes:

在分布式的场景下由于网络、时钟等原因，可能导致 ZooKeeper 的数据与真实运行的作业产生不一致，这种不一致通过正向的校验无法完全避免。需要另外启动一个线程定时校验注册中心数据与真实作业状态的一致性，即维持 ElasticJob 的最终一致性。

配置为小于 1 的任意值表示不执行修复。

jobShardingStrategyType:

详情请参见内置分片策略列表。

jobExecutorServiceHandlerType:

详情请参见内置线程池策略列表。

jobErrorHandlerType:

详情请参见内置错误处理策略列表。

props:

详情请参见作业属性配置列表。

disabled:

可用于部署作业时，先禁止启动，部署结束后统一启动。

overwrite:

如果可覆盖，每次启动作业都以本地配置为准。

作业监听器配置项

常规监听器配置项

可配置属性：无

分布式监听器配置项

可配置属性

属性名	类型	缺省值	描述
started-tim eout-milliseconds	long	Long.MAX_VALUE	最后一个作业执行前的执行方法的超时毫秒数
completed-tim eout-milliseconds	long	Long.MAX_VALUE	最后一个作业执行后的执行方法的超时毫秒数

事件追踪配置项

可配置属性

属性名	类型	缺省值	描述
type	String		事件追踪存储适配器类型
storage	泛型		事件追踪存储适配器对象

Java API

注册中心配置

用于注册和协调作业分布式行为的组件，目前仅支持 ZooKeeper。

类名称：org.apache.shardingsphere.elasticjob.reg.zookeeper.ZookeeperConfiguration

可配置属性：

属性名	构造器注入
serverLists	是
namespace	是
baseSleepTimeMilliseconds	否
maxSleepTimeMilliseconds	否
maxRetries	否
sessionTimeoutMilliseconds	否
connectionTimeoutMilliseconds	否
digest	否

作业配置

类名称：org.apache.shardingsphere.elasticjob.api.JobConfiguration

可配置属性：

属性名	构造器注入
jobName	是
shardingTotalCount	是
cron	否
timeZone	否
shardingItemParameters	否
jobParameter	否
monitorExecution	否
failover	否
misfire	否
maxTimeDiffSeconds	否
reconcileIntervalMinutes	否
jobShardingStrategyType	否
jobExecutorServiceHandlerType	否
jobErrorHandlerType	否
jobListenerTypes	否
description	否
props	否
disabled	否
overwrite	否

Spring Boot Starter

使用 Spring-boot 需在 pom.xml 文件中添加 elasticjob-lite-spring-boot-starter 模块的依赖。

```
<dependency>
    <groupId>org.apache.shardingsphere.elasticjob</groupId>
    <artifactId>elasticjob-lite-spring-boot-starter</artifactId>
    <version>${latest.release.version}</version>
</dependency>
```

注册中心配置

配置前缀：`elasticjob.reg-center`

可配置属性：

属性名	是否必填
server-lists	是
namespace	是
base-sleep-time-milliseconds	否
max-sleep-time-milliseconds	否
max-retries	否
session-timeout-milliseconds	否
connection-timeout-milliseconds	否
digest	否

配置格式参考：

YAML

```
elasticjob:
  regCenter:
    serverLists: localhost:6181
    namespace: elasticjob-lite-springboot
```

Properties

```
elasticjob.reg-center.namespace=elasticjob-lite-springboot
elasticjob.reg-center.server-lists=localhost:6181
```

作业配置

配置前缀：`elasticjob.jobs`

可配置属性：

属性名	是否必填
elasticJobClass / elasticJobType	是
cron	否
timeZone	否
jobBootstrapBeanName	否
sharding-total-count	是
sharding-item-parameters	否
job-parameter	否
monitor-execution	否
failover	否
misfire	否
max-time-diff-seconds	否
reconcile-interval-minutes	否
job-sharding-strategy-type	否
job-executor-service-handler-type	否
job-error-handler-type	否
job-listener-types	否
description	否
props	否
disabled	否
overwrite	否

elasticJobClass 与 elasticJobType 互斥，每项作业只能有一种类型

如果配置了 cron 属性则为定时调度作业, Starter 会在应用启动时自动启动; 否则为一次性调度作业, 需要通过 jobBootstrapBeanName 指定 OneOffJobBootstrap Bean 的名称, 在触发点注入 OneOffJobBootstrap 的实例并手动调用 execute() 方法。

配置格式参考:

YAML

```
elasticjob:
  jobs:
    simpleJob:
      elasticJobClass: org.apache.shardingsphere.elasticjob.lite.example.job.
SpringBootSimpleJob
      cron: 0/5 * * * * ?
      timeZone: GMT+08:00
      shardingTotalCount: 3
      shardingItemParameters: 0=Beijing,1=Shanghai,2=Guangzhou
    scriptJob:
      elasticJobType: SCRIPT
      cron: 0/10 * * * * ?
      shardingTotalCount: 3
      props:
        script.command.line: "echo SCRIPT Job: "
```

```
    manualScriptJob:
      elasticJobType: SCRIPT
      jobBootstrapBeanName: manualScriptJobBean
      shardingTotalCount: 9
      props:
        script.command.line: "echo Manual SCRIPT Job: "
```

Properties

```
elasticjob.jobs.simpleJob.elastic-job-class=org.apache.shardingsphere.elasticjob.
lite.example.job.SpringBootSimpleJob
elasticjob.jobs.simpleJob.cron=0/5 * * * * ?
elasticjob.jobs.simpleJob.timeZone=GMT+08:00
elasticjob.jobs.simpleJob.sharding-total-count=3
elasticjob.jobs.simpleJob.sharding-item-parameters=0=Beijing,1=Shanghai,2=Guangzhou
elasticjob.jobs.scriptJob.elastic-job-type=SCRIPT
elasticjob.jobs.scriptJob.cron=0/5 * * * * ?
elasticjob.jobs.scriptJob.sharding-total-count=3
elasticjob.jobs.scriptJob.props.script.command.line=echo SCRIPT Job:
elasticjob.jobs.manualScriptJob.elastic-job-type=SCRIPT
elasticjob.jobs.manualScriptJob.job-bootstrap-bean-name=manualScriptJobBean
elasticjob.jobs.manualScriptJob.sharding-total-count=3
elasticjob.jobs.manualScriptJob.props.script.command.line=echo Manual SCRIPT Job:
```

事件追踪配置

配置前缀：`elasticjob.tracing`

属性名	可选值	是否必填	描述
type	RDB	否	
includeJobNames		否	作业白名单
excludeJobNames		否	作业黑名单

includeJobNames 与 excludeJobNames 互斥，事件追踪配置只能有一种属性

includeJobNames 与 excludeJobNames 都为空时，默认为所有作业加载事件追踪

目前仅提供了 RDB 类型的事件追踪数据源实现。Spring IoC 容器中存在 DataSource 类型的 bean 且配置数据源类型为 RDB 时会自动配置事件追踪，无须显式创建。

配置格式参考：

YAML

```
elasticjob:
  tracing:
    type: RDB
    excludeJobNames: [ job-name-1, job-name-2 ]
```

Properties

```
elasticjob.tracing.type=RDB
elasticjob.tracing.excludeJobNames=[ job-name ]
```

作业信息导出配置

配置前缀：elasticjob.dump

属性名	缺省值	是否必填
enabled	true	否
port		是

Spring Boot 提供了作业信息导出端口快速配置，只需在配置中指定导出所用的端口号即可启用导出功能。如果没有指定端口号，导出功能不会生效。

配置参考：

YAML

```
elasticjob:
  dump:
    port: 9888
```

Properties

```
elasticjob.dump.port=9888
```

Spring 命名空间

使用 Spring 命名空间需在 pom.xml 文件中添加 elasticjob-lite-spring 模块的依赖。

```
<dependency>
    <groupId>org.apache.shardingsphere.elasticjob</groupId>
    <artifactId>elasticjob-lite-spring-namespace</artifactId>
    <version>${latest.release.version}</version>
</dependency>
```

命名空间：http://shardingsphere.apache.org/schema/elasticjob/elasticjob.xsd

注册中心配置

<elasticjob:zookeeper />

可配置属性:

属性名	是否必填
id	是
server-lists	是
namespace	是
base-sleep-time-milliseconds	否
max-sleep-time-milliseconds	否
max-retries	否
session-timeout-milliseconds	否
connection-timeout-milliseconds	否
digest	否

作业配置

<elasticjob:job />

可配置属性:

属性名	是否必填
id	是
class	否
job-ref	否
registry-center-ref	是
tracing-ref	否
cron	是
timeZone	否
sharding-total-count	是
sharding-item-parameters	否
job-parameter	否
monitor-execution	否
failover	否
misfire	否
max-time-diff-seconds	否
reconcile-interval-minutes	否
job-sharding-strategy-type	否
job-executor-service-handler-type	否
job-error-handler-type	否
job-listener-types	否
description	否
props	否
disabled	否
overwrite	否

事件追踪配置

<elasticjob:rdb-tracing />

可配置属性:

属性名	类型	是否必填	缺省值	描述
id	String	是		事件追踪 Bean 主键
data-source-ref	DataSource	是		事件追踪数据源 Bean 名称

快照导出配置

<elasticjob:snapshot />

可配置属性：

属性名	类型	是否必填	缺省值	描述
id	String	是		监控服务在 Spring 容器中的主键
registry-center-ref	String	是		注册中心 Bean 的引用，需引用 reg:zookeeper 的声明
dump-port	String	是		导出作业信息数据端口使用方法: echo“dump@jobName” | nc 127.0.0.1 9888

内置策略

简介

ElasticJob 通过 SPI 方式允许开发者扩展策略；与此同时，ElasticJob 也提供了大量的内置策略以便于开发者使用。

使用方式

内置策略通过 type 进行配置。本章节根据功能区分并罗列 ElasticJob 全部的内置算法，供开发者参考。

作业分片策略

平均分片策略

类型：AVG_ALLOCATION

根据分片项平均分片。

如果作业服务器数量与分片总数无法整除，多余的分片将会顺序的分配至每一个作业服务器。

举例说明：1. 如果 3 台作业服务器且分片总数为 9，则分片结果为：1=[0,1,2], 2=[3,4,5], 3=[6,7,8]；2. 如果 3 台作业服务器且分片总数为 8，则分片结果为：1=[0,1,6], 2=[2,3,7], 3=[4,5]；3. 如果 3 台作业服务器且分片总数为 10，则分片结果为：1=[0,1,2,9], 2=[3,4,5], 3=[6,7,8]。

奇偶分片策略

类型：ODEVITY

根据作业名称哈希值的奇偶数决定按照作业服务器 IP 升序或是降序的方式分片。

如果作业名称哈希值是偶数，则按照 IP 地址进行升序分片；如果作业名称哈希值是奇数，则按照 IP 地址进行降序分片。可用于让服务器负载在多个作业共同运行时分配的更加均匀。

举例说明：1. 如果 3 台作业服务器，分片总数为 2 且作业名称的哈希值为偶数，则分片结果为：1 = [0], 2 = [1], 3 = []；2. 如果 3 台作业服务器，分片总数为 2 且作业名称的哈希值为奇数，则分片结果为：3 = [0], 2 = [1], 1 = []。

轮询分片策略

类型：ROUND_ROBIN

根据作业名称轮询分片。

线程池策略

CPU 资源策略

类型：CPU

根据 CPU 核数 * 2 创建作业处理线程池。

单线程策略

类型：SINGLE_THREAD

使用单线程处理作业。

错误处理策略

记录日志策略

类型：LOG

默认内置：是

记录作业异常日志，但不中断作业执行。

抛出异常策略

类型：THROW

默认内置：是

抛出系统异常并中断作业执行。

忽略异常策略

类型：IGNORE

默认内置：是

忽略系统异常且不中断作业执行。

邮件通知策略

类型：EMAIL

默认内置：否

发送邮件消息通知，但不中断作业执行。

Maven 坐标：

```
<dependency>
    <groupId>org.apache.shardingsphere.elasticjob</groupId>
    <artifactId>elasticjob-error-handler-email</artifactId>
    <version>${latest.release.version}</version>
</dependency>
```

可配置属性：

属性名	说明	是否必填	默认值
email.host	邮件服务器地址	是	•
email.port	邮件服务器端口	是	•
email.username	邮件服务器用户名	是	•
email.password	邮件服务器密码	是	•
email.useSsl	是否启用 SSL 加密传输	否	true
email.subject	邮件主题	否	ElasticJob error message
email.from	发送方邮箱地址	是	•
email.to	接收方邮箱地址	是	•
email.cc	抄送邮箱地址	否	null
email.bcc	密送邮箱地址	否	null
email.debug	是否开启调试模式	否	false

企业微信通知策略

类型：WECHAT

默认内置：否

发送企业微信消息通知，但不中断作业执行。

Maven 坐标：

```
<dependency>
    <groupId>org.apache.shardingsphere.elasticjob</groupId>
    <artifactId>elasticjob-error-handler-wechat</artifactId>
    <version>${latest.release.version}</version>
</dependency>
```

可配置属性：

属性名	说明	是否必填	默认值
wechat.webhook	企业微信机器人的 webhook 地址	是	•
wechat.connectTimeoutMilliseconds	与企业微信服务器建立连接的超时时间	否	3000 毫秒
wechat.readTimeoutMilliseconds	从企业微信服务器读取到可用资源的超时时间	否	5000 毫秒

钉钉通知策略

类型：DINGTALK

默认内置：否

发送钉钉消息通知，但不中断作业执行。

Maven 坐标：

```
<dependency>
    <groupId>org.apache.shardingsphere.elasticjob</groupId>
    <artifactId>elasticjob-error-handler-dingtalk</artifactId>
    <version>${latest.release.version}</version>
</dependency>
```

可配置属性：

属性名	说明	是否必填	默认值
dingtalk.webhook	钉钉机器人的 webhook 地址	是	•
dingtalk.keyword	自定义关键词	否	null
dingtalk.secret	签名的密钥	否	null
dingtalk.connectTimeoutMilliseconds	与钉钉服务器建立连接的超时时间	否	3000 毫秒
dingtalk.readTimeoutMilliseconds	从钉钉服务器读取到可用资源的超时时间	否	5000 毫秒

作业属性配置

简介

ElasticJob 提供属性配置的方式为不同类型的作业提供定制化配置。

作业类型

简单作业

接口名称：org.apache.shardingsphere.elasticjob.simple.job.SimpleJob

可配置属性：无

数据流作业

接口名称：org.apache.shardingsphere.elasticjob.dataflow.job.DataflowJob

可配置属性：

名称	数据类型	说明	默认值
streaming.process	boolean	是否开启流式处理	false

脚本作业

类型：SCRIPT

可配置属性：

名称	数据类型	说明	默认值
script.command.line	String	脚本内容或运行路径	.

HTTP 作业

类型：HTTP

可配置属性：

名称	数据类型	说明	默认值
http.uri	String	http 请求 uri	.
http.method	String	http 请求方法	.
http.data	String	http 请求数据	.
http.connect.timeout.milliseconds	String	http 连接超时	3000
http.read.timeout.milliseconds	String	http 读超时	5000
http.content.type	String	http 请求 ContentType	.

6.1.5 运维手册

本章节是 ElasticJob-Lite 的运维参考手册。

部署指南

应用部署

1. 启动 ElasticJob-Lite 指定注册中心的 ZooKeeper。
2. 运行包含 ElasticJob-Lite 和业务代码的 jar 文件。不限于 jar 或 war 的启动方式。
3. 当作业服务器配置多网卡时，可通过设置系统变量 `elasticjob.preferred.network.interface` 指定网卡地址或 `elasticjob.preferred.network.ip` 指定 IP。ElasticJob 默认获取网卡列表中第一个非回环可用 IPV4 地址。

运维平台和 RESTFul API 部署 (可选)

1. 解压缩 `elasticjob-lite-console-${version}.tar.gz` 并执行 `bin\start.sh`。
2. 打开浏览器访问 `http://localhost:8899/` 即可访问控制台。8899 为默认端口号，可通过启动脚本输入 `-p` 自定义端口号。
3. 访问 RESTFul API 方法同控制台。
4. `elasticjob-lite-console-${version}.tar.gz` 可通过 `mvn install` 编译获取。

导出作业信息

使用 ElasticJob-Lite 过程中可能会碰到一些分布式问题，导致作业运行不稳定。

由于无法在生产环境调试，通过 dump 命令可以把作业内部相关信息导出，方便开发者调试分析；另外为了不泄露隐私，已将相关信息中的 IP 地址以 ip1, ip2…的形式过滤，可以在互联网上公开传输环境信息，便于进一步完善 ElasticJob。

开启监听端口

使用 Java 开启导出端口配置请参见Java API 使用指南。使用 Spring 开启导出端口配置请参见Spring 使用指南。

执行导出命令

导出命令完全参照 ZooKeeper 的四字命令理念。

导出至标准输出

```
echo "dump@jobName" | nc < 任意一台作业服务器 IP> 9888
```

```
[chris:elastic-job]echo "dump" | nc localhost 9888
/simpleElasticJob/servers |
/simpleElasticJob/servers/ip1 |
/simpleElasticJob/servers/ip1/status | READY
/simpleElasticJob/servers/ip1/sharding | 0,1,2,3,4,5,6,7,8,9
/simpleElasticJob/servers/ip1/hostName | localhost
/simpleElasticJob/leader |
/simpleElasticJob/leader/sharding |
/simpleElasticJob/leader/execution |
/simpleElasticJob/leader/election |
/simpleElasticJob/leader/election/latch |
/simpleElasticJob/leader/election/host | ip1
/simpleElasticJob/config |
/simpleElasticJob/config/shardingTotalCount | 10
/simpleElasticJob/config/shardingItemParameters | 0=A,1=B,2=C,3=D,4=E,5=F,6=G,7=H,8=I,9=J
/simpleElasticJob/config/processCountIntervalSeconds | 300
/simpleElasticJob/config/monitorPort | 9888
/simpleElasticJob/config/monitorExecution | false
/simpleElasticJob/config/misfire | true
/simpleElasticJob/config/maxTimeDiffSeconds | -1
/simpleElasticJob/config/jobShardingStrategyClass |
/simpleElasticJob/config/jobParameter |
/simpleElasticJob/config/jobClass | com.dangdang.example.elasticjob.spring.job.SimpleJobDemo
/simpleElasticJob/config/fetchDataCount | 1
/simpleElasticJob/config/failover | true
/simpleElasticJob/config/description | 只运行一次的作业示例
/simpleElasticJob/config/cron | 0/5 * * * * ?
/simpleElasticJob/config/concurrentDataProcessThreadCount | 1
```

图 2: 导出命令

导出至文件

```
echo "dump@jobName" | nc < 任意一台作业服务器 IP> 9888 > job_debug.txt
```

作业运行状态监控

通过监听 ElasticJob-Lite 的 ZooKeeper 注册中心的几个关键节点即可完成作业运行状态监控功能。

监听作业服务器存活

监听 job_name:raw-latex:*instances*:raw-latex:‘\job‘_instance_id 节点是否存在。该节点为临时节点，如果作业服务器下线，该节点将删除。

运维平台

解压缩 elasticjob-lite-console-${version}.tar.gz 并执行 bin\start.sh。打开浏览器访问 http://localhost:8899/ 即可访问控制台。8899 为默认端口号，可通过启动脚本输入 -p 自定义端口号。

登录

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

```
auth.root_username=root
auth.root_password=root
auth.guest_username=guest
auth.guest_password=guest
```

功能列表

- 登录安全控制
- 注册中心、事件追踪数据源管理
- 快捷修改作业设置
- 作业和服务器维度状态查看
- 操作作业禁用:raw-latex: 启用、停止和删除等生命周期
- 事件追踪查询

设计理念

运维平台和 ElasticJob-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。

不支持项

- 添加作业

作业在首次运行时将自动添加。ElasticJob-Lite 以 jar 方式启动，并无作业分发功能。如需完全通过运维平台发布作业，请使用 ElasticJob-Cloud。

6.2 ElasticJob-Cloud

6.2.1 简介

ElasticJob-Cloud 采用自研 Mesos Framework 的解决方案，额外提供资源治理、应用分发以及进程隔离等功能。

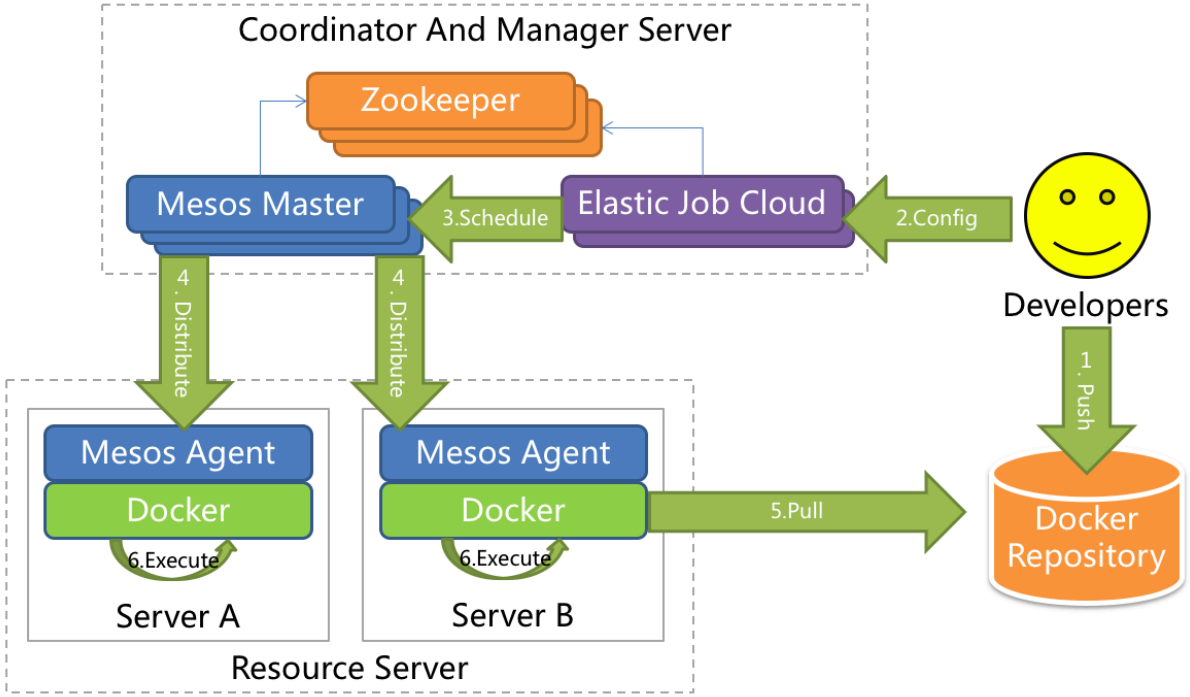

图 3: ElasticJob-Cloud Architecture

6.2.2 对比

	ElasticJob-Lite	*ElasticJob-Cloud*
无中心化	是	否
资源分配	不支持	支持
作业模式	常驻	常驻 + 瞬时
部署依赖	ZooKeeper	ZooKeeper + Mesos

ElasticJob-Cloud 的优势在于对资源细粒度治理，适用于需要削峰填谷的大数据系统。

6.2.3 使用手册

本章节将介绍 ElasticJob-Cloud 相关使用。更多使用细节请参见使用示例。

开发指南

作业开发

ElasticJob-Lite 和 ElasticJob-Cloud 提供统一作业接口，开发者仅需对业务作业进行一次开发，之后可根据不同的配置以及部署至不同环境。

作业开发详情请参见 ElasticJob-Lite 使用手册。

作业启动

需定义 `main` 方法并调用 `JobBootstrap.execute()`，例子如下：

```
public class MyJobDemo {

    public static void main(final String[] args) {
        JobBootstrap.execute(new MyJob());
    }
}
```

本地运行模式

在开发 ElasticJob-Cloud 作业时，开发人员可以脱离 Mesos 环境，在本地运行和调试作业。可以利用本地运行模式充分的调试业务功能以及单元测试，完成之后再部署至 Mesos 集群。

本地运行作业无需安装 Mesos 环境。

```
// 创建作业配置
JobConfiguration jobConfig = JobConfiguration.newBuilder("myJob", 3).cron("0/5 * *
* * * ?").build();

// 配置当前运行的作业的分片项
int shardingItem = 0;

// 创建本地执行器
new LocalTaskExecutor(new MyJob(), jobConfig, shardingItem).execute();
```

6.2.4 配置手册

ElasticJob-Cloud 提供应用发布及作业注册等 RESTful API，可通过 curl 操作。

请求 url 前缀为 `/api`

鉴权 API

获取 AccessToken

url: login

方法：POST

参数类型：application/json

参数列表：

属性名	类型	是否必填	缺省值	描述
username	String	是		API 鉴权用户名
password	String	是		API 鉴权密码

响应体：

属性名	类型	描述
accessToken	String	API 鉴权 token

示例：

```
curl -H "Content-Type: application/json" -X POST http://elasticjob_cloud_host:8899/
api/login -d '{"username": "root", "password": "pwd"}'
```

响应体：

```
{"accessToken":"some_token"}
```

应用 API

发布应用

url：app

方法：POST

参数类型：application/json

参数列表：

属性名	类型	是否必填	缺省值	描述
appName	String	是		作业应用名称
appURL	String	是		作业应用所在路径
cpuCount	double	否	1	作业应用启动所需要的 CPU 数量
memoryMB	double	否	128	作业应用启动所需要的内存 MB
bo otstrapScript	String	是		启动脚本
a ppCacheEnable	boolean	否	true	每次执行作业时是否从缓存中读取应用
eventTrace Sampling-Count	int	否	0（不采样）	常驻作业事件采样率统计条数

参数详细说明：

appName:

为 ElasticJob-Cloud 的作业应用唯一标识。

appURL:

必须提供可以通过网络访问的路径。

bootstrapScript:

如：bin:raw-latex:*start*.sh

appCacheEnable:

禁用则每次执行任务均从应用仓库下载应用至本地。

eventTraceSamplingCount:

为避免数据量过大，可对频繁调度的常驻作业配置采样率，即作业每执行 N 次，才会记录作业执行及追踪相关数据。

示例：

```
curl -l -H "Content-type: application/json" -X POST -d '{"appName":"my_app","appURL
":"http://app_host:8080/my-job.tar.gz","cpuCount":0.1,"memoryMB":64.0,
"bootstrapScript":"bin/start.sh","appCacheEnable":true,"eventTraceSamplingCount":0}
' http://elastic_job_cloud_host:8899/api/app
```

修改应用配置

url：app

方法：PUT

参数类型：application/json

参数列表：

属性名	类型	是否必填	缺省值	描述
appName	String	是		作业应用名称
a ppCacheEnable	boolean	是	true	每次执行作业时是否从缓存中读取应用
eventTrace Sampling-Count	int	否	0（不采样）	常驻作业事件采样率统计条数

示例：

```
curl -l -H "Content-type: application/json" -X PUT -d '{"appName":"my_app",
"appCacheEnable":true}' http://elastic_job_cloud_host:8899/api/app
```

作业 API

注册作业

url：job/register

方法：POST

参数类型：application/json

参数列表：

属性名	类型	是否必填	缺省值	描述
appName	String	是		作业应用名称
cpuCount	double	是		单片作业所需要的 CPU 数量，最小值为 0.001
memoryMB	double	是		单片作业所需要的内存 MB，最小值为 1
j obExecutionType	Enum	是		作业执行类型。TRANSIENT 为瞬时作业，DAEMON 为常驻作业
jobName	String	是		作业名称
cron	String	否		cron 表达式，用于配置作业触发时间
sha rdingTotalCount	int	是		作业分片总数
shardin gItemParameters	String	否		自定义分片参数
jobParameter	String	否		作业自定义参数
failover	boolean	否	false	是否开启失效转移
misfire	boolean	否	false	是否开启错过任务重新执行
jobExecutorSer viceHandlerType	boolean	否	false	作业线程池处理策略
jobE rrorHandlerType	boolean	否	false	作业错误处理策略
description	String	否		作业描述信息
props	Properties	否		作业属性配置信息

使用脚本类型的瞬时作业可直接将脚本上传至 appURL，而无需打成 tar 包。如果只有单个脚本文件可无需压缩。如是复杂脚本应用，仍可上传 tar 包，支持各种常见压缩格式。

示例：

```
curl -l -H "Content-type: application/json" -X POST -d '{"appName":"my_app",
"cpuCount":0.1,"memoryMB":64.0,"jobExecutionType":"TRANSIENT","jobName":"my_job",
"cron":"0/5 * * * * ?","shardingTotalCount":5,"failover":true,"misfire":true}'
http://elastic_job_cloud_host:8899/api/job/register
```

修改作业配置

url：job/update

方法：PUT

参数类型：application/json

参数：同注册作业

示例：

```
curl -l -H "Content-type: application/json" -X PUT -d '{"appName":"my_app","jobName
":"my_job","cpuCount":0.1,"memoryMB":64.0,"jobExecutionType":"TRANSIENT","cron":"0/
5 * * * * ?","shardingTotalCount":5,"failover":true,"misfire":true}' http://
elastic_job_cloud_host:8899/api/job/update
```

注销作业

url：job/deregister

方法：DELETE

参数类型：application/json

参数：作业名称

示例：

```
curl -l -H "Content-type: application/json" -X DELETE -d 'my_job' http://elastic_
job_cloud_host:8899/api/job/deregister
```

触发一次作业

url：job/trigger

方法：POST

参数类型：application/json

参数：作业名称

说明：即事件驱动，通过调用 API 而非定时的触发作业。目前仅对瞬时作业生效。

示例：

```
curl -l -H "Content-type: application/json" -X POST -d 'my_job' http://elastic_job_
cloud_host:8899/api/job/trigger
```

6.2.5 运维手册

本章节是 ElasticJob-Cloud 的运维参考手册。

部署指南

调度器部署步骤

1. 启动 ElasticJob-Cloud-Scheduler 和 Mesos 指定作为注册中心的 ZooKeeper
2. 启动 Mesos Master 和 Mesos Agent
3. 解压 `elasticjob-cloud-scheduler-${version}.tar.gz`
4. 执行 `bin\start.sh` 脚本启动 elasticjob-cloud-scheduler

作业部署步骤

1. 确保 ZooKeeper, Mesos Master/Agent 以及 ElasticJob-Cloud-Scheduler 已正确启动
2. 将打包作业的 tar.gz 文件放至网络可访问的位置，如：ftp 或 http。打包的 tar.gz 文件中 `main` 方法需要调用 ElasticJob-Cloud 提供的 `JobBootstrap.execute` 方法
3. 使用 curl 命令调用 RESTful API 发布应用及注册作业。详情请参见：配置指南

调度器配置步骤

可修改 `conf\elasticjob-cloud-scheduler.properties` 文件变更系统配置。

配置项说明:

属性名称	是 | 默认值否 | 必 | 填 |	描述
hostname	是 |	服务器真实的 IP 或 hostname，不能是 127.0.0.1 或 localhost
user	否 |	Mesos framework 使用的用户名称
m esos_url	是 | **zk://12** 7.0.0.1:2 181/mesos	Mesos 所使用的 ZooKeeper 地址
zk _servers	是 | **127.0** .0.1:2181	ElasticJob-Cloud 所使用的 ZooKeeper 地址
zk_n amespace	否 | **elastic** job-cloud	ElasticJob-Cloud 所使用的 ZooKeeper 命名空间
z k_digest	否 |	ElasticJob-Cloud 所使用的 ZooKeeper 登录凭证
h ttp_port	是 | **8899**	RESTful API 所使用的端口号
job_ state_qu eue_size	是 | **10000**	堆积作业最大值，超过此阀值的堆积作业将直接丢弃。阀值过大可能会导致 ZooKeeper 无响应，应根据实测情况调整
event_ trace_rd b_driver	否 |	作业事件追踪数据库驱动
eve nt_trace _rdb_url	否 |	作业事件追踪数据库 URL
event_tr **ace_rdb_** username	否 |	作业事件追踪数据库用户名
event_tr **ace_rdb_** password	否 |	作业事件追踪数据库密码
auth_ username	否 | **root**	API 鉴权用户名
auth_ password	否 | **pwd**	API 鉴权密码

- 停止：不提供停止脚本，可直接使用 kill 命令终止进程。

高可用

介绍

调度器的高可用是通过运行几个指向同一个 ZooKeeper 集群的 ElasticJob-Cloud-Scheduler 实例来实现的。ZooKeeper 用于在当前主 ElasticJob-Cloud-Scheduler 实例失败的情况下执行领导者选举。通过至少两个调度器实例来构成集群，集群中只有一个调度器实例提供服务，其他实例处于待命状态。当该实例失败时，集群会选举剩余实例中的一个来继续提供服务。

配置

每个 ElasticJob-Cloud-Scheduler 实例必须使用相同的 ZooKeeper 集群。例如，如果 ZooKeeper 的 Quorum 为 zk://1.2.3.4:2181,2.3.4.5:2181,3.4.5.6:2181/elasticjob-cloud，则 `elasticjob-cloud-scheduler.properties` 中 ZooKeeper 相关配置为：

```
# ElasticJob-Cloud's ZooKeeper address
zk_servers=1.2.3.4:2181,2.3.4.5:2181,3.4.5.6:2181

# ElasticJob-Cloud's ZooKeeper namespace
zk_namespace=elasticjob-cloud
```

运维平台

运维平台内嵌于 elasticjob-cloud-scheduler 的 jar 包中，无需额外启动 WEB 服务器。可通过修改配置文件中 http_port 参数来调整启动端口，默认端口为 8899，访问地址为 `http://{your_scheduler_ip}:8899`。

登录

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

功能列表

- 应用管理（发布、修改、查看）
- 作业管理（注册、修改、查看以及删除）
- 作业状态查看（待运行、运行中、待失效转移）
- 作业历史查看（运行轨迹、执行状态、历史仪表盘）

设计理念

运维平台采用纯静态 HTML + JavaScript 方式与后台的 RESTful API 交互，通过读取作业注册中心展示作业配置和状态，数据库展现作业运行轨迹及执行状态，或更新作业注册中心数据修改作业配置。

7 开发者手册

ElasticJob 可插拔架构提供了 SPI 的扩展点。对于开发者来说，可以十分方便的对功能进行定制化扩展。

本章节将 ElasticJob 的 SPI 扩展点悉数列出。如无特殊需求，用户可以使用 ElasticJob 提供的内置实现；高级用户则可以参考各个功能模块的接口进行自定义实现。

ElasticJob 社区非常欢迎开发者将自己的实现类反馈至开源社区，让更多用户从中收益。

7.1 作业分片策略

作业分片策略，用于将作业在分布式环境下分解成为任务使用。

SPI 名称	详细说明
JobShardingStrategy	作业分片策略

已知实现类	详细说明
AverageAllocationJobShardingStrategy	根据分片项平均分片
OdevitySortByNameJobShardingStrategy	根据作业名称哈希值的奇偶数决定按照作业服务器 IP 升序或是降序的方式分片
RotateServerByNameJobShardingStrategy	根据作业名称轮询分片

7.2 线程池策略

线程池策略，用于执行作业的线程池创建。

SPI 名称	详细说明
JobExecutorServiceHandler	作业执行线程池策略

已知实现类	详细说明
C PUUsageJobExecutorServiceHandler	根据 CPU 核数 * 2 创建作业处理线程池
Singl eThreadJobExecutorServiceHandler	使用单线程处理作业

7.3 错误处理策略

错误处理策略，用于作业失败时的处理策略。

SPI 名称	详细说明
JobErrorHandler	作业执行错误处理策略

已知实现类	详细说明
LogJobErrorHandler	记录作业异常日志，但不中断作业执行
ThrowJobErrorHandler	抛出系统异常并中断作业执行
IgnoreJobErrorHandler	忽略系统异常且不中断作业执行
EmailJobErrorHandler	发送邮件消息通知，但不中断作业执行
WechatJobErrorHandler	发送企业微信消息通知，但不中断作业执行
DingtalkJobErrorHandler	发送钉钉消息通知，但不中断作业执行

7.4 作业类名称提供策略

作业类名称提供策略，用于在不同的容器环境下提供准确的作业类名称。

SPI 名称	详细说明
JobClassNameProvider	作业类名称提供策略

已知实现类	详细说明
DefaultJobClassNameProvider	标准环境下的作业类名称提供策略
SpringProxyJobClassNameProvider	Spring 容器环境下的作业类名称提供策略

7.5 线路规划

7.5.1 Kernel

- ☒ Unified Job Config API
  - ☒ Core Config
  - ☒ Type Config
  - ☒ Root Config
- ☒ Job Types
  - ☒ Simple
  - ☒ Dataflow
  - ☒ Script
  - ☒ Http（3.0.0-beta 提供）
- ☒ Event Trace
  - ☒ Event Publisher
  - ☒ Database Event Listener
  - ☐ Other Event Listener
- ☐ Unified Schedule API
- ☐ Unified Resource API

7.5.2 ElasticJob-Lite

- ☒ Distributed Features
  - ☒ High Availability
  - ☒ Elastic scale in/out
  - ☒ Failover
  - ☒ Misfire
  - ☒ Idempotency
  - ☒ Reconcile
- ☒ Registry Center
  - ☒ ZooKeeper
  - ☐ Other Registry Center Supported
- ☒ Lifecycle Management
  - ☒ Add/Remove

  - ☒ Pause/Resume
  - ☒ Disable/Enable
  - ☒ Shutdown
  - ☒ Restful API
  - ☒ Web Console
- ☒ Job Dependency
  - ☒ Listener
  - ☐ DAG
- ☒ Spring Integrate
  - ☒ Namespace
  - ☒ Bean Injection
  - ☒ Spring Boot Starter（3.0.0-alpha 提供）

7.5.3 ElasticJob-Cloud

- ☒ Transient Job
  - ☒ High Availability
  - ☒ Elastic scale in/out
  - ☒ Failover
  - ☒ Misfire
  - ☒ Idempotency
- ☒ Daemon Job
  - ☒ High Availability
  - ☒ Elastic scale in/out
  - ☐ Failover
  - ☐ Misfire
  - ☒ Idempotency
- ☒ Mesos Scheduler
  - ☒ High Availability
  - ☒ Reconcile
  - ☐ Redis Based Queue Improvement
  - ☐ Http Driver
- ☒ Mesos Executor

  - ☒ Executor Reuse Pool
  - ☐ Progress Reporting
  - ☐ Health Detection
  - ☐ Log Redirect
- ☒ Lifecycle Management
  - ☒ Job Add/Remove
  - ☐ Job Pause/Resume
  - ☒ Job Disable/Enable
  - ☐ Job Shutdown
  - ☒ App Add/Remove
  - ☒ App Disable/Enable
  - ☒ Restful API
  - ☒ Web Console
- ☐ Job Dependency
  - ☐ Listener
  - ☐ Workflow
  - ☐ DAG
- ☒ Job Distribution
  - ☒ Mesos Based Distribution
  - ☐ Docker Based Distribution
- ☒ Resources Management
  - ☒ Resources Allocate
  - ☐ Cross Data Center
  - ☐ A/B Test
- ☒ Spring Integrate
  - ☒ Bean Injection

8 下载

8.1 最新版本

ElasticJob 的发布版包括源码包及其对应的二进制包。由于下载内容分布在镜像服务器上，所以下载后应该进行 GPG 或 SHA-512 校验，以此来保证内容没有被篡改。

8.1.1 ElasticJob - 版本: 3.0.1 (发布日期: Oct 11, 2021)

- 源码: [SRC] [ASC] [SHA512]
- ElasticJob-Lite 二进制包: [TAR] [ASC] [SHA512]
- ElasticJob-Cloud-Scheduler 二进制包: [TAR] [ASC] [SHA512]
- ElasticJob-Cloud-Executor 二进制包: [TAR] [ASC] [SHA512]

8.1.2 ElasticJob-UI - 版本: 3.0.1 (发布日期: Jan 19, 2022)

- 源码: [SRC] [ASC] [SHA512]
- ElasticJob-Lite-UI 二进制包: [TAR] [ASC] [SHA512]
- ElasticJob-Cloud-UI 二进制包: [TAR] [ASC] [SHA512]

即将发布

8.2 全部版本

全部版本请到 Archive repository 查看。

8.3 校验版本

PGP 签名文件

使用 PGP 或 SHA 签名验证下载文件的完整性至关重要。可以使用 GPG 或 PGP 验证 PGP 签名。请下载 KEYS 以及发布的 asc 签名文件。建议从主发布目录而不是镜像中获取这些文件。

```
gpg -i KEYS
```

或者

```
pgpk -a KEYS
```

或者

```
pgp -ka KEYS
```

要验证二进制文件或源代码，您可以从主发布目录下载相关的 asc 文件，并按照以下指南进行操作。

```
gpg --verify apache-shardingsphere-elasticjob-********.asc apache-shardingsphere-
elasticjob-*********
```

或者

```
pgpv apache-shardingsphere-elasticjob-********.asc
```

或者

```
pgp apache-shardingsphere-elasticjob-********.asc
```

9 采用公司

9.1 登记

欢迎采用了 ElasticJob 的公司在此登记，您的支持是我们最大的动力。

请按公司名 + 首页 + 应用案例（可选）的格式在此处登记。

9.2 谁在使用 ElasticJob?

共计 83 家公司。

9.2.1 电子商务

当当

三只松鼠

宝视佳

海智在线

走秀网

河姆渡

中航电子采购平台

帮 5 采

春播

惠农网

飞马大宗

洋桃跨境供应链

点购集团

晶泓科技

京东

9.2.2 金融行业

甜橙金融 (翼支付)

无锡锡商银行

拍拍贷

银盛支付

众安保险

金财互联

连连支付

耀莱在线

浙江汇信科技

捞财宝

卡牛信用管家

借贷宝

金汇金融

91 科技集团

9.2.3 数字化与云服务

云嘉云计算

金柚网

树熊网络

南方电网深圳数研院

9.2.4 出行

吉祥航空

曹操出行

途虎养车

首汽约车

iTrip 爱去

卖好车

天天拍车

滴滴出行

9.2.5 物流

圆通速递

好运虎物流

德坤物流

9.2.6 房地产

自如网

优客工场

链家网

9.2.7 互联网教育

贝聊科技

爱启航

会通教育

新课堂教育

跟谁学

起点学院

9.2.8 互联网文娱

咪咕互娱

磨铁文学

松鼠白菜

9.2.9 新闻资讯

房价网

凤凰汽车

淘股吧

饭好约

搜狐网

9.2.10 通信科技

魅族

一加科技

9.2.11 物联网

联想懂的通信

有方科技

机智云

沅朋物联

汇通天下

广联赛讯

商物云

9.2.12 软件开发及服务

神州泰岳

兑吧

新意互动

Yeahmobi

雷铭科技

众畅网络科技

深绘智能

未来信封

广州中软信息技术有限公司

售后宝

9.2.13 医疗健康

健合集团

云医科技

壹宝健康

尚一健康

9.2.14 零售业

永辉超市

9.2.15 人工智能

深兰科技

10 FAQ

10.1 阅读源码时为什么会出现编译错误?

回答:

ElasticJob 使用 lombok 实现极简代码。关于更多使用和安装细节，请参考 lombok 官网。

10.2 是否支持动态添加作业?

回答:

动态添加作业这个概念每个人理解不尽相同。

ElasticJob-Lite 为 jar 包，由开发或运维人员负责启动。启动时自动向注册中心注册作业信息并进行分布式协调，因此并不需要手工在注册中心填写作业信息。但注册中心与作业部署机无从属关系，注册中心并不能控制将单点的作业分发至其他作业机，也无法将远程服务器未启动的作业启动。ElasticJob-Lite 并不会包含 ssh 免密管理等功能。

ElasticJob-Cloud 为 mesos 框架，由 mesos 负责作业启动和分发。但需要将作业打包上传，并调用 ElasticJob-Cloud 提供的 RESTful API 写入注册中心。打包上传属于部署系统的范畴 ElasticJob-Cloud 并未涉及。

综上所述，ElasticJob 已做了基本动态添加功能，但无法做到真正意义的完全自动化添加。

10.3 为什么在代码或配置文件中修改了作业配置，注册中心配置却没有更新?

回答:

ElasticJob-Lite 采用无中心化设计，若每个客户端的配置不一致，不做控制的话，最后一个启动的客户端配置将会成为注册中心的最终配置。

ElasticJob-Lite 提出了 overwrite 概念，可通过 JobConfiguration 或 Spring 命名空间配置。`overwrite=true` 即允许客户端配置覆盖注册中心，反之则不允许。如果注册中心无相关作业的配置，则无论 overwrite 是否配置，客户端配置都将写入注册中心。

10.4 作业与注册中心无法通信会如何?

回答:

为了保证作业的在分布式场景下的一致性，一旦作业与注册中心无法通信，运行中的作业会立刻停止执行，但作业的进程不会退出。这样做的目的是为了防止作业重分片时，将与注册中心失去联系的节点执行的分片分配给另外节点，导致同一分片在两个节点中同时执行。当作业节点恢复与注册中心联系时，将重新参与分片并恢复执行新的分配到的分片。

10.5 ElasticJob-Lite 有何使用限制?

回答:

- 作业启动成功后修改作业名称视为新作业，原作业废弃。
- 一旦有服务器波动，或者修改分片项，将会触发重新分片；触发重新分片将会导致运行中的流式处理的作业在执行完本次作业后不再继续执行，等待分片结束后再恢复正常。
- 开启 monitorExecution 才能实现分布式作业幂等性（即不会在多个作业服务器运行同一个分片）的功能，但 monitorExecution 对短时间内执行的作业（如秒级触发）性能影响较大，建议关闭并自行实现幂等性。

10.6 怀疑 ElasticJob-Lite 在分布式环境中有问题，但无法重现又不能在线上环境调试，应该怎么做?

回答:

分布式问题非常难于调试和重现，为此 ElasticJob-Lite 提供了 dump 命令。

如果您怀疑某些场景出现问题，可参照作业信息导出将作业运行时信息提交至社区。ElasticJob 已将 IP 地址等敏感信息过滤，导出的信息可在公网安全传输。

10.7 ElasticJob-Cloud 有何使用限制?

回答:

- 作业启动成功后修改作业名称视为新作业，原作业废弃。

10.8 在 ElasticJob-Cloud 中添加任务后，为什么任务一直在 ready 状态，而不开始执行?

回答:

任务在 mesos 有单独的 agent 可提供所需的资源时才会启动，否则会等待直到有足够的资源。

10.9 控制台界面无法正常显示?

回答:

使用控制台时应确保与 ElasticJob 相关版本保持一致，否则会导致不可用。

10.10 为什么控制台界面中的作业状态是分片待调整?

回答:

分片待调整表示作业已启动但尚未获得分片时的状态。

10.11 为什么首次启动存在任务调度延迟的情况?

回答：ElasticJob 执行任务会获取本机 IP，首次可能存在获取 IP 较慢的情况。尝试设置 `-Djava.net.preferIPv4Stack=true`.

10.12 Windows 环境下, 运行 ShardingSphere-ElasticJob-UI, 找不到或无法加载主类 org.apache.shardingsphere.elasticjob.lite.ui.Bootstrap, 如何解决?

回答:

某些解压缩工具在解压 ShardingSphere-ElasticJob-UI 二进制包时可能将文件名截断，导致找不到某些类。

解决方案:

打开 cmd.exe 并执行下面的命令:

```
tar zxvf apache-shardingsphere-elasticjob-${RELEASE.VERSION}-lite-ui-bin.tar.gz
```

10.13 运行 Cloud Scheduler 持续输出日志“Elastic job: IP:PORT has leadership”，不能正常运行

回答:

Cloud Scheduler 依赖 Mesos 库，启动时需要通过 `-Djava.library.path` 指定 Mesos 库所在目录。

例如, Mesos 库位于 `/usr/local/lib`, 启动 Cloud Scheduler 前需要设置 `-Djava.library.path=/usr/local/lib`。

Mesos 相关请参考 Apache Mesos。

10.14 在多网卡的情况下无法获取到合适的 IP

回答:

可以通过系统变量 `elasticjob.preferred.network.interface` 指定网卡或 `elasticjob.preferred.network.ip` 指定 IP 地址。

例如:

1. 指定网卡 eno1：`-Delasticjob.preferred.network.interface=eno1`。
2. 指定 IP 地址 192.168.0.100：`-Delasticjob.preferred.network.ip=192.168.0.100`。
3. 泛指 IP 地址 (正则表达式) 192.168.*：`-Delasticjob.preferred.network.ip=192.168.*`。

11 博客

- 2020-07 InfoQ 文章：ElasticJob 的产品定位与新版本设计理念
- 2020-07 开源中国：GitHub 上持续冲榜，ElasticJob 重启
- 2020-05 官微快讯：分布式调度项目 ElasticJob 即将重新起航
- 2017-09 Mesosphere 新闻：Q&A with Zhang Liang of Dangdang: the biggest book seller in China
- 2017-04 InfoQ 新闻：分布式调度中间件 Elastic-Job 2.1.0 发布：Cloud Native 里程碑版本
- 2017-03 源码分析：Elastic-Job 项目源码分析系列
- 2015-12 InfoQ 文章：详解当当网的分布式作业框架 elastic-job
- 2015-11 高可用架构群分享：新一代分布式任务调度框架，elastic-job 开源项目的 10 项特性
- 2015-11 CSDN 专访：深度解读分布式作业调度框架 elastic-job
- 2015-09 InfoQ 新闻：当当开源 elastic-job，分布式作业调度框架