SOFAStack

任务调度
使用指南

产品版本：AntStack Plus 1.13.1

文档版本：20230705

蚂蚁集团
ANT GROUP

法律声明

蚂蚁集团版权所有©2022，并保留一切权利。

未经蚂蚁集团事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。

商标声明

蚂蚁集团 ANT GROUP及其他蚂蚁集团相关的商标均为蚂蚁集团所有。本文档涉及的第三方的注册商标，依法由权利人所有。

免责声明

由于产品版本升级、调整或其他原因，本文档内容有可能变更。蚂蚁集团保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在蚂蚁集团授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过蚂蚁集团授权渠道下载、获取最新版的用户文档。如因文档使用不当造成的直接或间接损失，本公司不承担任何责任。

通用约定

格式	说明	样例
危险	该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。	危险 重置操作将丢失用户配置数据。
警告	该类警示信息可能会导致系统重大变更甚至故障，或者导致人身伤害等结果。	警告 重启操作将导致业务中断，恢复业务时间约十分钟。
注意	用于警示信息、补充说明等，是用户必须了解的内容。	注意 权重设置为0，该服务器不会再接受新请求。
说明	用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。	说明 您也可以通过按Ctrl+A选中全部文件。
>	多级菜单递进。	单击**设置**> **网络**> **设置网络类型**。
粗体	表示按键、菜单、页面名称等UI元素。	在**结果确认**页面，单击**确定**。
`Courier字体`	命令或代码。	执行 `cd /d C:/window` 命令，进入Windows系统文件夹。
斜体	表示参数、变量。	`bae log list --instanceid` *Instance_ID*
[] 或者 [a|b]	表示可选项，至多选择一个。	`ipconfig [-all|-t]`
{} 或者 {a|b}	表示必选项，至多选择一个。	`switch {active|stand}`

目录

1.什么是任务调度 07
1.1. 概述 07
1.2. 控制台总览 07
1.3. 产品优势 09
1.4. 功能特性 09
1.5. 应用场景 11
1.6. 基础术语 11
2.快速入门 13
3.简单任务 21
3.1. 应用开发 21
3.1.1. 准备工作 21
3.1.2. 实现处理器接口和类 24
3.1.3. 创建和发布应用 25
3.2. 创建调度任务 27
3.3. CRON 表达式详解 32
4.集群任务 36
4.1. 概述 36
4.2. 应用开发 38
4.2.1. 准备工作 38
4.2.2. 任务拆分和执行 41
4.2.3. 创建和发布应用 56
4.3. 创建调度任务 57
4.4. 集群任务限流 61
5.消息任务 65
5.1. 应用开发 65
5.1.1. 准备工作 65

5.1.2. 实现处理器接口和类 65
5.1.3. 创建和发布应用 67
5.2. 创建消息任务 69
6.拓扑任务 74
6.1. 概述 74
6.2. 新建拓扑任务 75
6.3. 触发拓扑任务 78
6.4. 拓扑任务网关 79
6.4.1. 并行网关 79
6.4.2. 条件网关 81
6.4.3. 分片网关 84
6.4.4. 使用跨 zone 网关 86
6.5. 拓扑任务版本 90
6.6. 拓扑执行事件 91
6.7. 任务节点人工干预 93
7.任务管理 96
7.1. 管理简单任务和集群任务 96
7.2. 管理拓扑任务 100
7.3. 任务分类 102
7.4. 操作审计 102
7.5. 任务运行监控 103
8.常见问题 110
8.1. 产品相关问题 110
8.2. 功能相关问题 110
8.3. 任务触发与执行问题 111
8.4. 任务拓扑相关问题 117
8.5. 客户端不在线问题排查 118
8.6. 其他常见问题 119

1.什么是任务调度

1.1. 概述

任务调度 TS（Task Scheduler）提供分布式任务调度框架，实现任务的分布式处理，并能规范化、自动化、可视化和集中化对金融企业不同业务系统的任务进行统一的调度和全方位监控运维管理，达到所有任务有序、高效运行的目的，极大降低开发和运维的成本。

任务调度产品提供多种任务类型满足不同场景需求，进行可靠的自动化任务调度：

- **简单任务**

  简单的 RPC 任务，适用于业务逻辑简单的场景。当需要并发执行并且不关注资源利用率时，可以使用简单任务的分片功能；当需要分成若干步骤按顺序执行时，可以使用简单任务的分步功能。分片数量在任务设置时指定。

- **集群任务**

  支持对数据进行自定义维度的分片，目前最多支持两层拆分，可以充分利用集群的所有机器，当数据量较大时，多层调度任务可以更快的处理完数据。支持动态分片，根据业务需求将任务拆分多个分片，充分利用业务方集群资源，提供并行处理能力。

- **消息任务**

  一种消息队列的消息类型，由消息主题和消息事件码作为唯一标识，需要配合消息队列产品使用。

- **拓扑任务**

  一种特殊的任务，是通过事件触发的任务集合。拓扑任务中的任务执行流程均起始于开始节点，终止于结束节点，任务的执行流程形成一张有向无环图。

1.2. 控制台总览

您可以在任务调度产品控制台 **总览** 页面查看所有任务的动态和统计信息。

任务统计

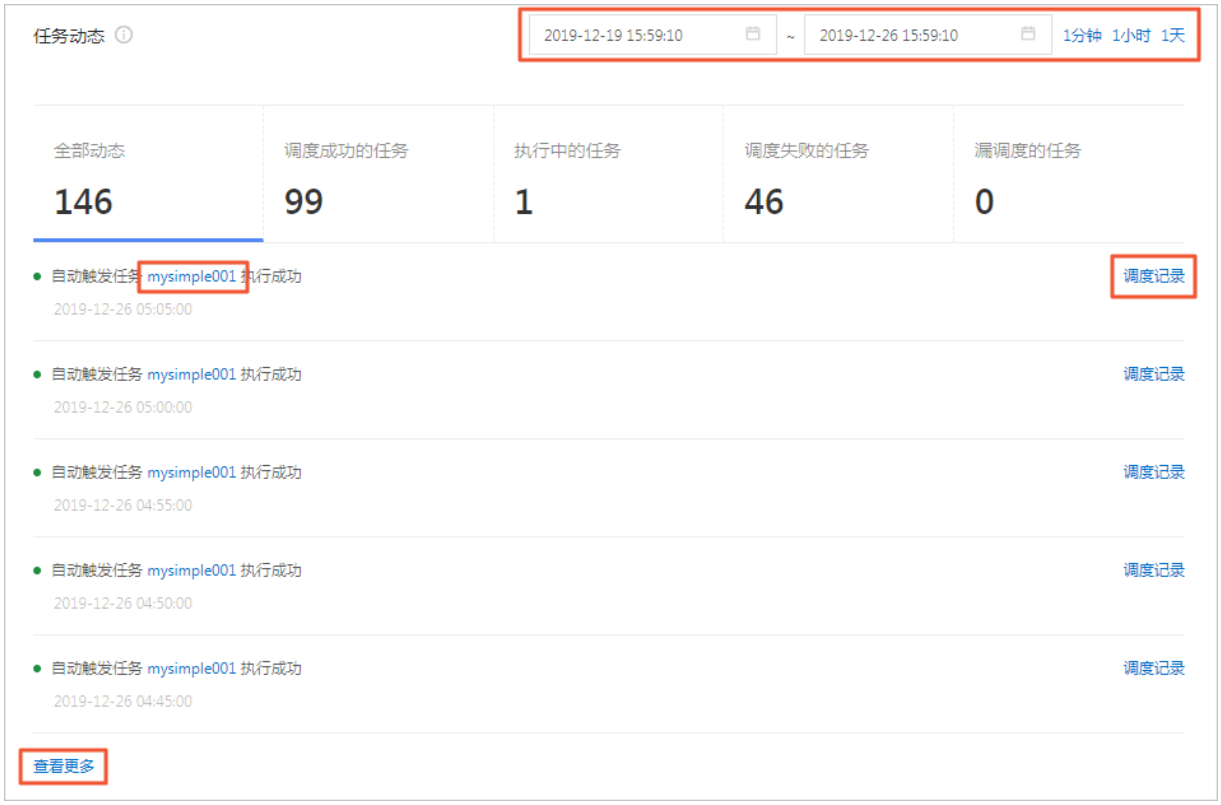
显示当前环境中已存在的各类任务数量统计：

- 任务总数：当前环境中已配置的任务总数，包括简单任务、集群任务。将鼠标悬浮在图标上方以查看每个任务类型的数量比例。单击悬浮窗中的 **查看详情** 链接跳转到任务列表。
- 任务拓扑总数：当前环境中已编排的拓扑任务总数。

任务动态

按时间倒序显示当前环境下的所有调度记录。调度记录保留 7 天。您可以使用时间控件调整要统计的时间区间。如果任务是手动触发，列表中还将显示此次调度的执行人。

- 全部调度记录
- 调度成功的任务
- 执行中的任务
- 调度失败的任务
- 漏调度的任务



在任务动态列表中，单击相应的 **任务名称** 或 **调度记录** 链接以跳转到任务详情页面。

动态列表默认显示最近 5 条记录。如果您需要加载更多记录，单击页面底部的 **查看更多**。

只有通信类型为 CALLBACK 的任务能将执行记录返回控制台。您在控制台界面中只能查看 CALLBACK 类型任务的调度状态。要查看通信类型为单向（ONEWAY）的任务的调度记录，请参考 查看调度记录。

> **说明**
>
> 如果您需要获取更长时间的调度记录，请 提交工单 联系我们。

1.3. 产品优势

金融级高可用

支持机房级容灾，任何单机或单机房故障都不影响当前任务的执行，保障业务的正常执行。

高可靠高扩展

支持无限水平扩展，无性能容量瓶颈，监控任务的执行情况，可以及时发现执行异常或没有执行的任务，通过可配置的异常处理策略对异常任务进行补偿。

高性能

通过分布式以及优化的底层架构、支持多层调度模式、可进行无限拆分、多线程并行处理，显著提升大数据量的批任务处理的性能。

可视化集中式管理

通过简易操作的可视化集中式管理平台可对上万个任务节点进行集中化管理，简化运维管理操作，提高处理效率。

1.4. 功能特性

本文介绍任务调度的功能特性。

分布式任务调度框架

提供多种任务类型满足不同场景需求，进行可靠的自动化任务调度。

- 简单任务

  最基本的任务类型，适用于业务逻辑简单的场景。单个简单任务对应一个 handler，支持任务分片及分步：

  - 当需要并发执行并且不关注资源利用率时，可以使用简单任务的分片功能。
  - 当需要将一个任务分成若干步骤按顺序执行时，可以使用简单任务的分步功能。

- 集群任务

  适用于复杂的调度场景，支持用户按业务的要求，通过自定义的拆分逻辑将一个大的任务拆分到多个客户端上并发执行。

  集群任务可以分成两个阶段：

  - 拆分阶段：对数据进行分片，不限制拆分层数，将拆分结果上报给服务端，由服务端根据拆分的 Chunk（一批待处理数据分片集合） 通知客户端来拉取数据进行处理。
  - 执行阶段：客户端接收到通知后拉取数据进行处理，处理完后继续拉取新的数据，直到数据都处理完成。

- 拓扑任务

  一种特殊的任务，由许多通过事件触发的任务组成的集合。任务拓扑本身可以通过 CRON 表达式或者事件触发，但其包含的子任务必须是通过事件触发的，并且通信方式必须是 CALLBACK。任务拓扑中的任务执行流程均从开始节点开始，到结束节点终止，任务的执行流程形成一张有向无环图。

任务图形化依赖编排

支持将多个任务按执行的顺序编排成一张拓扑图，自动执行，无需人工干预，从而保证对大量任务的调度快速高效地完成。

- 支持多种依赖管理网关

  提供直接依赖，条件网关及并行网关管理任务之间的关系。

- 执行过程中人工干预

  支持对执行中的任务拓扑进行优雅停止及断点续跑。

- 图形化编排操作

  通过图形化操作界面，快速完成任务的编排，提高工作效率。

任务配置项

提供丰富的任务配置项，包括多种路由策略、多种失败重试模式、多种漏触发处理策略，并支持在线配置调度任务入参。

- 多种路由策略

  支持定时任务在客户端集群中随机触发、轮询触发或定向触发。

- 多种失败重试模式

  支持的失败重试模式包括手动重试、不重试、重试三次、重试到下次触发，并支持配置超时时间。

- 多种漏触发处理策略

  漏触发处理策略包括不处理、立即触发、时间充裕则立即触发三种。

- 支持在线配置调度任务入参

  支持配置自定义参数，任务触发时会把这些参数传递给任务。

任务管理功能

对任务提供丰富的管理功能，包括手动触发、配置导入导出、OpenAPI 等。

- 自动触发和手动触发

  支持任务通过定时触发、周期性触发、事件触发及人工手动触发。

- 任务配置导入导出，批量启用及禁用

  支持在控制台上对配置以文件的方式的批量导入及导出，批量启用及禁用。

- 提供 OpenAPI

  为开发者提供了对任务进行查看、增加、触发、删除等丰富的管理接口。

图形化的集中式管理界面

提供简单易用的管理界面，包括任务依赖视图、任务执行监控、客户端节点信息、触发记录查看。

- 任务依赖视图

  提供任务全景图，清晰查看任务的上游下游依赖关系。

- 任务执行监控

  支持实时查看任务的执行状态、进度、时长等信息。

- 支持查看任务的触发记录

包括触发时间、执行的客户端 IP、执行是否成功、执行失败原因等。

1.5. 应用场景

金融行业批处理任务

金融行业批处理任务需要在规定时间内处理完指定的数据。传统的处理方式是根据数据库分片来处理，每台机器处理一个数据分片，这种方式的缺点就是当业务集群的机器数量大于分片数时，很多机器处于闲置状态，资源利用率低。集群任务允许用户自己对数据进行拆分，将待处理数据拆分成一个个索引块，业务集群中的每台机器都会分配到数据进行处理，使用灵活，并可以监控任务的执行情况。

- 高效率：通过将批处理进行多层拆分，拆分为多个任务同步处理，提高任务处理的效率。
- 资源利用率高：业务集群中服务器都会分配数据，提高利用率。
- 简单易用：图形化操作页面，集中管理任务的运行状态，便易运维。

金融行业日切

金融行业的日切是一个很复杂的操作，需要多个任务按照指定顺序协同工作才可以完成。传统实现方式是在系统代码里指定任务的执行顺序，当业务发生变化时需要修改代码才可以正常工作，可维护性很差。使用任务调度产品的任务编排功能，可以轻松的完成任务之间的依赖调整，大大提高了可维护性，并可以直观地看到任务的执行情况。

- 图形化编排页面：提供图形化的任务编排页面，将多个任务编排成一个任务拓扑，按编排的顺序执行。
- 人工干预：支持对任务拓扑执行过程中进行人工干预，包括优雅停止、断点续跑等。
- 监控：支持对拓扑任务的执行过程进行监控，实时掌握任务运行的状态。

金融行业余额缓冲记账

任务调度产品的流式任务可以规避传统定时任务解决缓冲记账模式的局限性，只要数据没有处理完，任务会一直处理下去，待某个时间点数据处理完成后，任务自动休眠，等到有数据产生再次开始执行。

- 流式数据处理：任务一旦触发就会一直执行，用于对流式不间歇的数据进行处理。
- 可靠稳定：任务一旦触发就会一直执行，用于对流式不间歇的数据进行处理。
- 简单易用：提供图形化控制台，实时查看数据的处理情况。

1.6. 基础术语

中文	英文	释义
简单任务	Simple task	最基本的任务类型，适用于业务逻辑简单的场景。单个简单任务对应一个 handler，支持任务分步。
集群任务	Cluster task	支持在单个执行步骤中把数据拆分成多个数据分片（data chunk），把一个任务的数据分散到不同的机器上运行。适用于数据量很大的业务场景。调度过程分为拆分阶段和执行阶段。
任务拓扑	Topology	由许多通过事件触发的任务的集合，描述了任务依赖关系。子任务可以并发执行或根据条件执行。

中文	英文	释义
任务编排	Topology management	将多个子任务根据业务逻辑编辑成拓扑任务的过程。
任务分步	Step	一个简单任务可以拆分成多个执行步骤（step），由不同的执行器（handler）按顺序执行。前置步骤成功完成后才开始执行下一个步骤。 例如对账任务的处理步骤如下： 1. 下载银行的清算文件。 2. 根据约定格式解析文件。 3. 文件数据与机构日间交易数据核对。 4. 结果处理（差错账生成及处理）。 每一个步骤可以由对应的 handler 单独执行。
任务分片	Data chunk	任务分片功能适用于执行时间很长的任务。当数据量很大时，任务调度系统支持在单个执行步骤中把数据拆分成多个分片（data chunk），把一个任务的数据分散到不同的机器上运行。 一个数据分片只由一个线程执行。多个数据分片可以在多台机器上调度执行，也可以在一台机器的多个线程上调度执行，具体调度方式由调度中心根据负载进度进行调整。 使用任务分片可以缩短任务执行时间，同时也可以解决单机计算能力上限的问题，并降低部分任务失败对整体系统的影响。
Cron 表达式	Cron expression	固定格式的字符串，用来指定一个时间点或者一个重复触发的时间间隔。

2.快速入门

任务调度产品支持多种任务类型，包括简单任务、集群任务，您可以根据业务的需求，选择其一实现业务。本文以简单任务为例，介绍如何快速完成一个简单任务的本地开发以及控制台任务配置等。

> **说明**
>
> 若您需要任务调度工程示例代码，请下载 示例工程。

操作步骤

1. 本地应用开发。

   i. 创建 SOFABoot 工程。

   下载并配置好一个 SOFABoot 工程原型。配置步骤，请参见 快速开始。

   > **说明**
   >
   > 任务调度客户端支持使用 SOFABoot 框架或其它 Maven 工程框架开发。此处以 SOFABoot 为例介绍应用开发示例。非 SOFABoot 工程的配置，请参见 非 SOFABoot 工程。

   ii. 引入任务调度的依赖。

   a. 在 SOFABoot 工程中引入任务调度的依赖：

   ```
   <dependency>
       <groupId>com.alipay.sofa</groupId>
       <artifactId>scheduler-enterprise-sofa-boot-starter</artifactId>
   </dependency>
   ```

   b. 对于 SOFABoot 3.4.3 以下的版本，需要添加如下 Tracer 依赖。

   > **说明**
   >
   > - 添加前，请仔细检查工程是否已经依赖了 Tracer，避免 JAR 包冲突问题。
   > - SOFABoot 已在 3.4.3 及其以上版本自动依赖 Tracer，无需再手动添加。SOFABoot 最新版本信息，详见 SOFABoot 版本说明。

   ```
   <dependency>
       <groupId>com.alipay.sofa</groupId>
       <artifactId>tracer-enterprise-sofa-boot-starter</artifactId>
   </dependency>
   ```

   iii. 修改全局配置项。

   在 `application.properties` 文件中，完成中间件全局配置。详情请参见 properties 配置项。

iv. 实现处理器接口。

客户端提供了 `ISimpleJobHandler` 接口给业务方实现，通过实现该接口编写业务逻辑代码。

```
/**
 * 简单任务处理器接口。
 */
public interface ISimpleJobHandler extends IJobHandler{

    /**
     * 处理 request。
     *
     * @param context
     * @return
     */
    ClientCommonResult handle(JobExecuteContext context);
}
```

代码说明：

- `JobExecuteContext` 是触发任务的请求上下文，您可以从里面获取到任务的一些信息。例如在控制台创建任务时设置的自定义参数，可以通过 `getCustomParam(String key)` 方法获取。
- 如果设置了任务分片，可以通过方法 `getSharding()` 获取当前机器分配到的分片号。详细信息可以查看 `com.alipay.antschedulerclient.model.JobExecuteContext` 这个类的源码。

v. 实现类。

类里必须分别实现 `handle` 、 `getThreadPool` 、 `getName` 方法。

- `handle` ：在该方法里执行业务逻辑。
- `getThreadPool` ：获取一个执行该任务的线程池，如果没有设置，会使用客户端自带的线程池。
- `getName` ：获取这个执行器的名字。客户端接收到任务的触发请求时，会根据名字寻找匹配的执行器。

> **说明**
>
> 当任务需要按多个步骤执行时，需要写多个实现类，并在控制台开启分步。

代码示例如下：

```
public class AlwaysSuccessHandler implements ISimpleJobHandler{
    private static final String NAME ="ALWAYS_SUCCESS_JOB";
    private static final ThreadPoolExecutor executor =new ThreadPoolExecutor(20,
            300,1,TimeUnit.HOURS,new ArrayBlockingQueue<Runnable>(100){})

    /**
     * 处理业务逻辑。
     **/
    @Override
    public ClientCommonResult handle(JobExecuteContext context) throws InterruptedExc
eption{
        // 获取自定义参数。
        Integer num =(Integer) context.getCustomParam("num");
        if(num ==null){
            num =0;
        }
        num = num +1;
        // 更新自定义参数。
        context.putCustomParams("num",5);
        return ClientCommonResult.buildSuccessResult();
    }

    @Override
    public ThreadPoolExecutor getThreadPool(){
        // 建议使用自定义实现类。
        return executor;
    }

    @Override
    public String getName(){
        // handler 名字，管控页面配置的 handler 即这部分定义内容。
        return NAME;
    }
}
```

vi. 配置 Spring。

通过以下任一方式将接口实现类配置为 Spring Bean：

- 在 `src/main/resources/META-INF/xxx/xxx-xxx.xml` 文件中进行配置。示例如下：

```
<bean id="sampleService" class="com.antcloud.demo.antscheduler.service.SampleServiceI
mpl"/>
```

- 使用注解驱动（annotation-driven）的方式声明 Bean。

2. 在云端发布应用。

本地客户端开发完成后，您需要将其发布部署到 SOFAStack 经典应用服务。详细步骤，请参见 发布部署应用。

3. 在控制台配置调度任务。

i. 登录任务调度控制台。

ii. 在左侧导航栏选择 **中间件 > 任务调度 > 任务配置**。

iii. 选择 **添加任务** > **简单任务**，然后配置以下参数：

参数	说明
任务名称	配置任务的名称，用于识别任务。
应用名称	填写或选择目标应用名称。 必须与工程中 `application.properties` 文件的 `spring.application.name` 配置的应用名称一致。
调度类型	配置任务调度的类型，可选值： ■ **cron 表达式触发**：该类型的任务在创建时需要配置 CRON 表达式定时触发任务。 ■ **事件触发**：该类型的任务通过外部事件触发任务。 ■ **定时触发**：该类型的任务在创建时也需要配置 CRON 表达式定时触发，并且只能执行一次。
Cron 表达式	配置 CRON 表达式，用于定时触发任务。配置方式请参见 CRON 表达式详解。 仅在 **调度类型** 为 **cron 表达式触发** 时配置。
任务分组（可选）	选择或新建一个任务分组，方便归纳管理任务。
执行器名字	必须与代码中任务处理器名字一致，即 实现类 步骤中 `getName` 方法获取的执行器名字。
是否分片	选择任务是否开启分片： ■ 不开启分片，任务将只调度到一台客户端服务器上执行。 ■ 开启分片，并设置分片数为 N，任务则会同时触发到 N 台客户端服务器上并发执行。 如果您开启了分片，您还需在本地工程编码实现任务分片。代码参见下文 **任务分片示例代码** 。

参数	说明
自定义参数（可选）	自定义参数支持从控制台向任务传递参数，参数也可以在任务间传递，格式为：`key=value`。 参数配置说明如下： ■ **参数名**：设置自定义参数的名称。 支持英文大小写、数字、下划线（_），不能超过 128 字符。 ■ **类型** 和 **参数值**：可选类型和参数值配置方式如下。 ■ STRING：支持通用的字符串。 ■ LIST：支持字符串数组，例如：`["aaa","bbb"]`。 ■ BOOLEAN：支持 true 或 false。 ■ PLACEHOLDER：支持参数替换，目前支持的变量如下： ■ `${sharding}`：替换当前分片值，配合简单任务使用。 ■ `${shardingCount}`：替换简单任务配置的分片数。 ■ `${triggerTime}`：替换预期的触发时间。 **说明** 您也可以在业务端通过上下文 context（该对象必须实现 Serializable。）获取和覆盖自定义参数。代码参见下文 **获取和覆盖自定义参数示例代码**。
路由策略	选择执行任务的路由策略，可选值： ■ **随机**：每次执行都随机分发到客户端服务器上。 ■ **定向**：每次执行都分发到同一个客户端服务器上。 暂不支持指定客户端服务器 IP。 ■ **轮询**：每次执行依次分发到每台客户端服务器上。
通信方式	选择任务的通信方式，可选值： ■ **单向 ONEWAY**：单向操作，没有返回值。控制台不记录触发记录、无超时、无重试。 高频非重要任务时建议使用。 ■ **回调 CALLBACK**：双向操作，有返回值。控制台可查看触发记录、可设置超时、可重试。 重要任务时建议使用，触发间隔至少 5 分钟。

参数	说明
任务互斥	选择是否开启任务互斥模式。开启任务互斥模式时，处在“执行中”或“暂停”状态的任务即使已经达到下一个任务触发时间点，仍无法再次触发。本次调度状态以“跳过”标记。默认为不开启。 仅当 **通信方式** 设置为 **回调 CALLBACK** 时配置。
漏触发策略	设置任务漏触发时的处理策略，可选值： ■ **忽略**：不进行补偿处理。 ■ **立即触发**：立即重新触发。 ■ **时间充裕时触发**：是否进行补偿触发取决于下次触发前的剩余时间是否充裕。若当前时间 + 超时时间 < 下次触发时间，则进行补偿触发，并将此次补偿触发插入触发记录；反之，则不触发。多次漏触发进行恢复时仅进行一次补偿触发。 仅当 **通信方式** 设置为 **回调 CALLBACK**，且 **调度类型** 为 **cron 表达式触发** 时，可配置此项。
超时策略	设置任务执行超时后的处理策略，可选值： ■ **不处理**：触发记录和执行记录状态均标记为失败。 ■ **按失败策略重试**：本次调度记录状态标为失败，并按照 **失败处理策略** 进行失败重试。重试后本次调度记录将被重置为重试结果。 ■ **停止后续触发**：执行记录状态为失败，触发记录状态为超时，并且无法再次触发。仅当开启任务互斥模式时可选择此项。 仅当 **通信方式** 设置为 **回调 CALLBACK** 时配置。
超时时间	设置任务执行的超时时间，当任务超过此时间未回调则认为执行失败。单位可选择为 **分钟** 和 **小时**。 仅当 **通信方式** 设置为 **回调 CALLBACK** 时配置。
失败处理策略	设置任务执行失败后的处理策略，可选值： ■ **不重试**：失败后不进行重试。 ■ **最多重试三次**：失败后立即进行重试，但最多重试三次。 ■ **重试到下次触发**：失败后立即进行重试，直到下次触发时间。
优先级	用于识别任务的重要性，暂无其他作用。 默认为 **中等**。
描述（可选）	配置任务的描述信息，例如业务含义、影响范围等，最多 1024 个字符。

任务分片示例代码

```
public class ShardingSchedule implements ISimpleJobHandler {

    private final Logger LOGGER = LoggerFactory.getLogger(ShardingSchedule.class);

    private ThreadPoolExecutor threadPool;

    @Override
    public String getName() {
        return "SHARDING_SCHEDULE";
    }

    @Override
    public ClientCommonResult handle(JobExecuteContext jobExecuteContext) throws Except
ion {
        int sharding = jobExecuteContext.getSharding();

        //当前机器获取到的分片数
        LOGGER.info("current machine sharding is " + sharding);
        //总的分片数
        System.out.println("total sharding num is " + jobExecuteContext.getShardingCoun
t());

        System.out.println("activity sharding is" + jobExecuteContext.getActivityShardi
ng());

        return ClientCommonResult.buildSuccessResult();
    }

    @Override
    public ThreadPoolExecutor getThreadPool() {
        return threadPool;
    }

 /**
     * Setter method for property threadPool.
     *
     * @param threadPool value to be assigned to property threadPool
     */
    public void setThreadPool(ThreadPoolExecutor threadPool) {
        this.threadPool = threadPool;
    }

}
```

获取和覆盖自定义参数示例代码

```
public class SimpleTaskDemo implements ISimpleJobHandler {

    @Override
    public ClientCommonResult handle(JobExecuteContext context) {
       // 根据 key 获取 obj
        Object obj = context.getCustomParam("intObj");

        // 获取自定义参数
        Map<String, Object> paramsMap = context.getCustomParams();
        context.putCustomParams("num", 2);
        List<String> listparam = newArrayList <>();
        listparam.add("aaaaaa");
        listparam.add("111111");
        context.putCustomParams("intparam", 111);
        context.putCustomParams("stringparam", "dfadsfad1243");
        context.putCustomParams("listparam", listparam);
        // 注意对象必须实现 Serializable
        context.putCustomParams("objectParam", newObject());
        return ClientCommonResult.buildSuccessResult();
    }

....
}
```

iv. 单击 **提交**。

v. 在任务列表中打开目标任务的 **是否启用** 开关。

如果有客户端在线，新配置的任务自动有客户端注册。任务在禁用的状态下，您也可以在操作列单击 **触发**，手动触发一次任务。

查看任务调度记录

1. 在任务列表中，单击目标任务名称。
2. 单击 **调度记录** 页签，查看调度记录。

   调度记录最多保留 7 天。您可以使用页签上方的下拉列表筛选调度记录类型，包括全部调度记录、调度执行成功、调度执行失败以及漏调度记录。

3.简单任务

3.1. 应用开发

3.1.1. 准备工作

本文介绍应用开发前的准备工作。

搭建工程

在开始使用任务调度产品之前，您需要根据实际需求，搭建以下类型的工程：

SOFABoot 工程

1. 下载并配置好一个 SOFABoot 工程原型。

   具体步骤参见 创建 Web 工程。

2. 对于 SOFABoot 3.4.3 以下的版本，需要添加如下 Tracer 依赖。

   > 说明
   > - 添加前，请仔细检查工程是否已经依赖了 Tracer，避免 JAR 包冲突问题。
   > - SOFABoot 将在 3.4.3 及其以上版本自动依赖 Tracer，无需再手动添加。SOFABoot 最新版本信息，详见 SOFABoot 版本说明。

```
<dependency>
     <groupId>com.alipay.sofa</groupId>
     <artifactId>tracer-enterprise-sofa-boot-starter</artifactId>
</dependency>
```

3. 在 SOFABoot 工程的 `pom.xml` 文件中引入任务调度产品的依赖：

```
<dependency>
      <groupId>com.alipay.sofa</groupId>
      <artifactId>scheduler-enterprise-sofa-boot-starter</artifactId>
</dependency>
```

4. 修改 SOFABoot 全局配置项。

   环境参数配置详情，请参见 properties 配置项。

非 SOFABoot 工程

1. 配置工程的 Maven `settings.xml` 文件。

   在本地 Maven 路径下的 `settings.xml` 文件中，配置 `mvn.cloud.alipay.com` 仓库的地址及用户名密码。配置后工程即可通过 Maven 获取该仓库中任务调度的 JAR 依赖包，不配置则无法获取到 JAR 包。

> **说明**
>
> `mvn.cloud.alipay.com` 仓库的地址及用户名密码请 提交工单 获取。

2. 在工程的主 `pom.xml` 文件中引入 `antschedulerclient` 的依赖。

   以 1.0.7 版本为例：

```
<dependency>
    <groupId>com.alipay.antschedulerclient</groupId>
    <artifactId>antschedulerclient-core</artifactId>
    <version>1.0.7</version>
</dependency>
<dependency>
    <groupId>com.alipay.antschedulerclient</groupId>
    <artifactId>antschedulerclient-antvip</artifactId>
    <version>1.0.7</version>
</dependency>

<!-- 间接依赖 -->
<dependency>
    <groupId>com.antcloud.antvip</groupId>
    <artifactId>antcloud-antvip-client</artifactId>
    <version>1.0.4</version>
</dependency>
<dependency>
    <groupId>com.antcloud.antvip</groupId>
    <artifactId>antcloud-antvip-common</artifactId>
    <version>1.0.4</version>
</dependency>
<!-- antvip-common的依赖 -->
<dependency>
    <groupId>commons-codec</groupId>
    <artifactId>commons-codec</artifactId>
    <version>1.4</version>
</dependency>
<dependency>
    <groupId>commons-io</groupId>
    <artifactId>commons-io</artifactId>
    <version>2.2</version>
</dependency>
<dependency>
    <groupId>commons-lang</groupId>
    <artifactId>commons-lang</artifactId>
    <version>2.4</version>
</dependency>
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>fastjson</artifactId>
    <version>1.2.50</version>
</dependency>

<dependency>
    <groupId>com.alipay.sofa.cloud</groupId>
```

```
    <groupId>com.alipay.sofa.cloud</groupId>
    <artifactId>sdk-core</artifactId>
    <version>0.1</version>
</dependency>
<dependency>
    <groupId>com.alipay.sofa.common</groupId>
    <artifactId>sofa-common-tools</artifactId>
    <version>1.0.12</version>
</dependency>

<dependency>
    <groupId>com.alipay.sofa</groupId>
    <artifactId>bolt</artifactId>
    <version>1.4.1</version>
</dependency>
<dependency>
    <groupId>com.alibaba.toolkit.common</groupId>
    <artifactId>toolkit-common-lang</artifactId>
    <version>1.1.1</version>
</dependency>
<dependency>
    <groupId>com.alipay.sofa</groupId>
    <artifactId>hessian</artifactId>
    <version>3.3.0</version>
</dependency>
<dependency>
    <groupId>com.alipay.sofa.lookout</groupId>
    <artifactId>lookout-api</artifactId>
    <version>1.4.3</version>
</dependency>
<dependency>
    <groupId>com.alipay.sofa</groupId>
    <artifactId>tracer-core</artifactId>
    <version>2.1.1</version>
</dependency>
<dependency>
    <groupId>com.alipay.sofa</groupId>
    <artifactId>tracer-extensions</artifactId>
    <version>2.1.1</version>
</dependency>
<dependency>
    <groupId>com.alipay.common</groupId>
    <artifactId>tracer</artifactId>
    <version>2.1.1</version>
</dependency>
<dependency>
    <groupId>com.google.guava</groupId>
    <artifactId>guava</artifactId>
    <version>20.0</version>
</dependency>
```

3. 获取环境配置参数。

   环境参数配置详情，请参见 properties 配置项。

4. 初始化客户端 Client。

   示例如下：

```
@Override
public void onApplicationEvent(ContextRefreshedEvent event) {
    try {
        ApplicationContext applicationContext = event.getApplicationContext();
        Map<String, IJobHandler> handlerBeans = applicationContext
                .getBeansOfType(IJobHandler.class);
        for (IJobHandler handler : handlerBeans.values()) {
            handlerList.add(handler);
        }
// zone(cell),如果为空则认为不分zone(cell)
        zone = System.getProperty("com.alipay.ldc.zone");

        Config config = new Config(app, zone, instanceId, accessKey, securityKey);
        AntVipRouterConfig routerConfig = new AntVipRouterConfig();
        routerConfig.setEndPoint(antvipEndpoint);
        AntVipRouter router = new AntVipRouter(routerConfig);
        antSchedulerClient = new AntSchedulerClient(config, router, handlerList);
        antSchedulerClient.init();
    } catch (Throwable t) {
        LOGGER.error("init antschedulerclient error", t);
        throw new RuntimeException(t);
    }
}
```

下载示例代码

为快速进行任务调度项目的本地工程开发，您需要下载任务调度 示例工程。

3.1.2. 实现处理器接口和类

一个简单任务对应一个 handler，支持任务分片及分步：

- 当需要并发执行并且不关注资源利用率时，可以使用简单任务的分片功能。
- 当需要将一个任务分成若干步骤按顺序执行时，可以使用简单任务的分步功能。

实现处理器接口

```
/**
 * 简单任务处理器接口
 */
public interface ISimpleJobHandler extends IJobHandler {

    /**
     * 处理request
     *
     * @param context
     * @return
     */
    ClientCommonResult handle(JobExecuteContext context);
}
```

代码说明：

`JobExecuteContext` 是触发任务的请求上下文，可以从里面获取到任务的一些信息，比如：

- 在控制台创建任务时设置的自定义参数，可以通过 `getCustomParam(String key)` 方法获取。
- 如果设置了任务分片，可以通过方法 `getSharding()` 来获取当前机器分配到的分片号。

详细信息，可以查看 `com.alipay.antschedulerclient.model.JobExecuteContext` 这个类的源码。

实现类

类里必须实现以下三个方法：

- **handle**：在该方法里执行业务逻辑。
- **getThreadPool**：获取一个执行该任务的线程池，如果没有设置，会使用 Client 自带的线程池。

```
new ThreadPoolExecutor(20,300,1,TimeUnit.HOURS,new ArrayBlockingQueue<Runnable>(100){})
```

- **getName**：获取这个执行器的名字，这个名字很重要，Client 接收到任务的触发请求时，会根据名字寻找匹配的执行器。

当任务需要按多个步骤执行时，需要写多个实现类，并在控制台开启分步功能。

代码示例

```
public class AlwaysSuccessHandler implements ISimpleJobHandler {

    public static final String NAME = "ALWAYS_SUCCESS_JOB";

    @Override
    public Client CommonResulthandle(JobExecuteContext context) throws InterruptedException
{
        //处理业务逻辑
        return ClientCommonResult.buildSuccessResult();
    }

    @Override
    public ThreadPoolExecutor getThreadPool() {
        return null;
    }

    @Override
    public String getName() {
        return NAME;
    }
}
```

3.1.3. 创建和发布应用

工程创建完成后，您需要在 SOFAStack 控制台创建和发布应用。

操作步骤

1. 创建应用。

i. 登录 SOFAStack 控制台。

ii. 在左侧导航栏选择 **应用管理** > **应用列表**。

iii. 单击 **创建应用**，然后配置以下参数：

参数	说明
应用名	配置应用名称。 可以包含字母、数字、连字符、下划线、且必须以字母开头，以字母或者数字结尾。
技术栈	根据项目中使用的 Java 版本选择对应的 SOFABoot 技术栈。
应用分组	选择应用所在分组。
应用标签	配置应用的标签： ■ **标签类型**：可选择 **系统标签** 或 **自定义标签**。 ■ **应用分级**：应用的分级信息。 ■ **变量值**：对应的变量值信息。
应用描述（可选）	配置应用的描述信息。

iv. 单击 **提交**。

2. 上传发布包。

i. 生成发布包。

在本地执行 `mvn clean package` 或者 `mvn clean install` 命令，在工程根目录下的 `target` 下打包生成一个 fat JAR 包

ii. 在 SOFAStack 控制台左侧导航栏，选择 **经典应用服务** > **应用发布** > **发布包管理**。

iii. 找到步骤 1 创建的应用，然后单击右侧的 **管理**。

iv. 单击 **上传发布包**，然后上传应用的发布包后，单击 **确认**。

3. 创建应用服务实例并关联发布包。

i. 在左侧导航栏选择 **经典应用服务** > **应用发布** > **应用服务**。

ii. 单击 **创建服务实例**，然后配置以下参数：

参数	说明
应用	在下拉列表选择步骤 1 创建的应用。
服务实例名称	配置服务实例名称。 可以包含字母、数字、连字符、下划线、且必须以字母开头，以字母或者数字结尾。
发布包	选择步骤 2 上传的发布包。
描述（可选）	配置应用实例的描述信息。
技术栈版本	在下拉列表选择技术栈版本。
部署单元	选择应用的部署单元。

iii. 单击 **确定**。

4. 发布应用。
   i. 在左侧导航栏选择 **经典应用服务** > **应用发布** > **发布包管理**。
   ii. 选中步骤 3 创建的服务实例，然后选择 **运维操作** > **发布部署**。
   iii. 单击 **立即发布**，然后单击 **整体发布**。

3.2. 创建调度任务

应用发布后，您可以为应用创建调度任务。

操作步骤

1. 登录任务调度控制台。
2. 在左侧导航栏选择 **任务配置**。
3. 选择 **添加任务** > **简单任务**，然后配置以下参数：

参数	说明
任务名称	配置任务的名称，用于识别任务。

参数	说明
应用名称	填写或选择目标应用名称。 支持英文、中文、数字、下划线（_）、短划线（-），50 个字符之内。必须与工程中 `application.properties` 文件的 `spring.application.name` 配置的应用名称一致。
调度类型	配置任务调度的类型，可选值： ○ **cron 表达式触发**：该类型的任务在创建时需要配置 CRON 表达式定时触发任务。 ○ **事件触发**：该类型的任务通过外部事件触发任务。 ○ **定时触发**：该类型的任务在创建时也需要配置 CRON 表达式定时触发，并且只能执行一次。
Cron 表达式	配置 CRON 表达式，用于定时触发任务。配置方式请参见 CRON 表达式详解。 仅在 **调度类型** 为 **cron 表达式触发** 时配置。
任务分组（可选）	选择或新建一个任务分组，方便归纳管理任务。
执行器名字	必须与代码中任务处理器名字一致，即 实现类 步骤中 `getName` 方法获取的执行器名字。
是否分片	选择任务是否开启分片： ○ 不开启分片，任务将只调度到一台客户端服务器上执行。 ○ 开启分片，并设置分片数为 N，任务则会同时触发到 N 台客户端服务器上并发执行。 如果您开启了分片，您还需在本地工程编码实现任务分片。代码参见下文 **任务分片示例代码** 。

参数	说明
自定义参数（可选）	自定义参数支持从控制台向任务传递参数，参数也可以在任务间传递，格式为：`key=value`。 参数配置说明如下： ○ **参数名**：设置自定义参数的名称。 支持英文大小写、数字、下划线（_），不能超过 128 字符。 ○ **类型** 和 **参数值**：可选类型和参数值配置方式如下。 ■ STRING：支持通用的字符串。 ■ LIST：支持字符串数组，例如：`["aaa","bbb"]`。 ■ BOOLEAN：支持 true 或 false。 ■ PLACEHOLDER：支持参数替换，目前支持的变量如下： ■ `${sharding}`：替换当前分片值，配合简单任务使用。 ■ `${shardingCount}`：替换简单任务配置的分片数。 ■ `${triggerTime}`：替换预期的触发时间。 **说明** 您也可以在业务端通过上下文 context（该对象必须实现 Serializable。）获取和覆盖自定义参数。代码参见下文 **获取和覆盖自定义参数示例代码**。
路由策略	选择执行任务的路由策略，可选值： ○ **随机**：每次执行都随机分发到客户端服务器上。 ○ **定向**：每次执行都分发到同一个客户端服务器上。 暂不支持指定客户端服务器 IP。 ○ **轮询**：每次执行依次分发到每台客户端服务器上。
通信方式	选择任务的通信方式，可选值： ○ **单向 ONEWAY**：单向操作，没有返回值。控制台不记录触发记录、无超时、无重试。 高频非重要任务时建议使用。 ○ **回调 CALLBACK**：双向操作，有返回值。控制台可查看触发记录、可设置超时、可重试。 重要任务时建议使用，触发间隔至少 5 分钟。

参数	说明
任务互斥	选择是否开启任务互斥模式。开启任务互斥模式时，处在“执行中”或“暂停”状态的任务即使已经达到下一个任务触发时间点，仍无法再次触发。本次调度状态以“跳过”标记。默认为不开启。 仅当 **通信方式** 设置为 **回调 CALLBACK** 时配置。
漏触发策略	设置任务漏触发时的处理策略，可选值： ○ **忽略**：不进行补偿处理。 ○ **立即触发**：立即重新触发。 ○ **时间充裕时触发**：是否进行补偿触发取决于下次触发前的剩余时间是否充裕。若当前时间 + 超时时间 < 下次触发时间，则进行补偿触发，并将此次补偿触发插入触发记录；反之，则不触发。多次漏触发进行恢复时仅进行一次补偿触发。 仅当 **通信方式** 设置为 **回调 CALLBACK**，且 **调度类型** 为 **cron 表达式触发** 时，可配置此项。
超时策略	设置任务执行超时后的处理策略，可选值： ○ **不处理**：触发记录和执行记录状态均标记为失败。 ○ **按失败策略重试**：本次调度记录状态标为失败，并按照 **失败处理策略** 进行失败重试。重试后本次调度记录将被重置为重试结果。 ○ **停止后续触发**：执行记录状态为失败，触发记录状态为超时，并且无法再次触发。仅当开启任务互斥模式时可选择此项。 仅当 **通信方式** 设置为 **回调 CALLBACK** 时配置。
超时时间	设置任务执行的超时时间，当任务超过此时间未回调则认为执行失败。单位可选择为 **分钟** 和 **小时**。 仅当 **通信方式** 设置为 **回调 CALLBACK** 时配置。
失败处理策略	设置任务执行失败后的处理策略，可选值： ○ **不重试**：失败后不进行重试。 ○ **最多重试三次**：失败后立即进行重试，但最多重试三次。 ○ **重试到下次触发**：失败后立即进行重试，直到下次触发时间。
优先级	用于识别任务的重要性，暂无其他作用。 默认为 **中等**。
描述（可选）	配置任务的描述信息，例如业务含义、影响范围等，最多 1024 个字符。

任务分片示例代码

```
public class ShardingSchedule implements ISimpleJobHandler {

    private final Logger LOGGER = LoggerFactory.getLogger(ShardingSchedule.class);

    private ThreadPoolExecutor threadPool;

    @Override
    public String getName() {
        return "SHARDING_SCHEDULE";
    }

    @Override
    public ClientCommonResult handle(JobExecuteContext jobExecuteContext) throws Exceptio
n {
        int sharding = jobExecuteContext.getSharding();

        //当前机器获取到的分片数
        LOGGER.info("current machine sharding is " + sharding);
        //总的分片数
        System.out.println("total sharding num is " + jobExecuteContext.getShardingCount(
));

        System.out.println("activity sharding is" + jobExecuteContext.getActivitySharding
());

        return ClientCommonResult.buildSuccessResult();
    }

@Override
    public ThreadPoolExecutor getThreadPool() {
        return threadPool;
    }

    /**
     * Setter method for property threadPool.
     *
     * @param threadPool value to be assigned to property threadPool
     */
    public void setThreadPool(ThreadPoolExecutor threadPool) {
        this.threadPool = threadPool;
    }

}
```

获取和覆盖自定义参数示例代码

```
public class SimpleTaskDemo implements ISimpleJobHandler {

    @Override
    public ClientCommonResult handle(JobExecuteContext context) {
       // 根据 key 获取 obj
        Object obj = context.getCustomParam("intObj");

        // 获取自定义参数
        Map<String, Object> paramsMap = context.getCustomParams();
        context.putCustomParams("num", 2);
        List<String> listparam = newArrayList <>();
        listparam.add("aaaaaa");
        listparam.add("111111");
        context.putCustomParams("intparam", 111);
        context.putCustomParams("stringparam", "dfadsfad1243");
        context.putCustomParams("listparam", listparam);
        // 注意对象必须实现 Serializable
        context.putCustomParams("objectParam", newObject());
        return ClientCommonResult.buildSuccessResult();
    }

....
}
```

4. 单击 **提交**。
5. 在任务列表中打开目标任务的 **是否启用** 开关。

   如果有客户端在线，新配置的任务自动有客户端注册。任务在禁用的状态下，您也可以在操作列单击 **触发**，手动触发一次任务。

3.3. CRON 表达式详解

CRON 表达式是一个字符串，以 5 或 6 个空格隔开，分为 6 或 7 个域，每一个域代表一个含义。

CRON 有如下两种语法格式：

- 秒 分 小时 日期 月份 星期 年
- 秒 分 小时 日期 月份 星期

每个域允许的值

域	允许的数值	允许的特殊字符	备注
秒	0~59	- * /	-
分	0~59	- * /	-
小时	0~23	- * /	-
日期	1~31	- * ? / L W C	-
月份	1~12	JAN-DEC - * /	-
星期	1~7	SUN-SAT - * ? / L C #	1 表示星期天，2 表示星期一，依次类推
年（可选）	留空，1970~2099	, - * /	自动生成，工具不显示该值

特殊字符的含义

字符	含义	示例
*	表示匹配域的任意值	在分这个域使用 *，即表示每分钟都会触发事件。
?	表示匹配域的任意值，但只能用在日期和星期两个域，因为这两个域会相互影响。	要在每月的 20 号触发调度，不管每个月的 20 号是星期几，则只能使用如下写法：`13 13 15 20 * ?` 。其中，因为日期域已经指定了 20 号，最后一位星期域只能用 `?` ，不能使用 `*` 。如果最后一位使用 `*` ，则表示不管星期几都会触发，与日期域的 20 号相斥，此时表达式不正确。
-	表示起止范围	在分这个域使用 5-20，表示从 5 分到 20 分钟每分钟触发一次。
/	表示起始时间开始触发，然后每隔固定时间触发一次	在分这个域使用 5/20，则意味着 5 分钟触发一次，而 25，45 等分别触发一次。

字符	含义	示例
,	表示列出枚举值	在分这个域使用 5,20，则意味着在 5 和 20 分每分钟触发一次。
L	表示最后，只能出现在日和星期两个域	在星期这个域使用 5L，意味着在最后的一个星期四触发。
W	表示有效工作日（周一到周五），只能出现在日这个域，系统将在离指定日期最近的有效工作日触发事件。	在日这个域使用 5W，如果 5 号是星期六，则将在最近的工作日星期五，即 4 号触发。如果 5 号是星期天，则在 6 号（周一）触发；如果 5 号为工作日，则就在 5 号触发。另外，W 的最近寻找不会跨过月份。
LW	这两个字符可以连用，表示在某个月最后一个工作日，即最后一个星期五。	
#	表示每个月第几个星期几，只能出现在星期这个域	在星期这个域使用 4#2，表示某月的第二个星期三，4 表示星期三，2 表示第二个。

示例

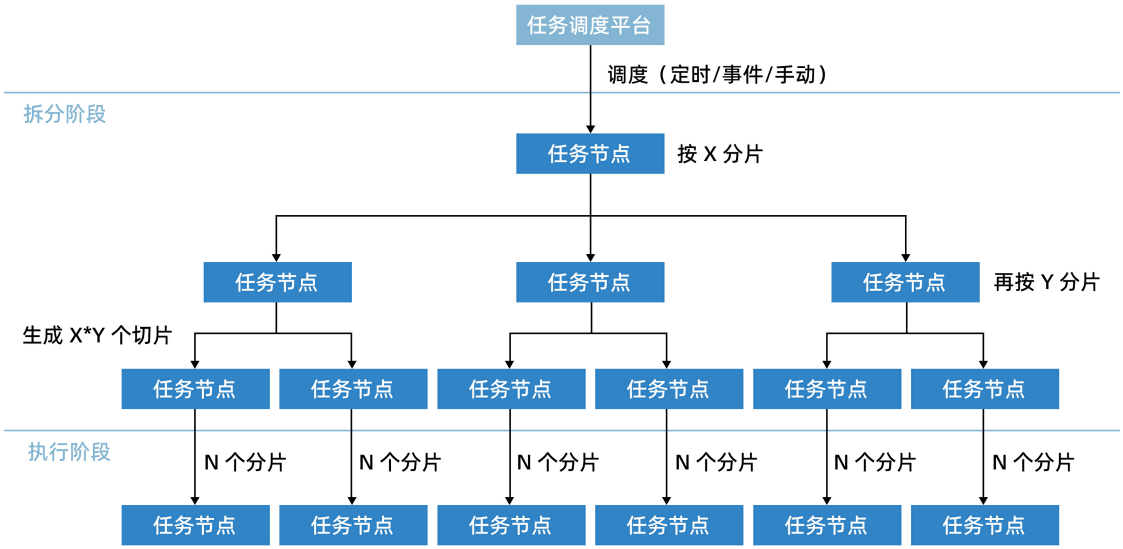
- `*/5 * * * * ?` ：每隔 5 秒执行一次
- `0 */1 * * * ?` ：每隔 1 分钟执行一次
- `0 0 2 1 * ? *` ：每月 1 日的凌晨 2 点执行一次
- `0 15 10 ? * MON-FRI` ：周一到周五每天上午 10：15 执行作业
- `0 15 10 ? 6L 2002-2006` ：2002 年至 2006 年的每个月的最后一个星期五上午 10:15 执行作业
- `0 0 23 * * ?` ：每天 23 点执行一次
- `0 0 1 * * ?` ：每天凌晨 1 点执行一次
- `0 0 1 1 * ?` ：每月 1 日凌晨 1 点执行一次
- `0 0 23 L * ?` ：每月最后一天 23 点执行一次
- `0 0 1 ? * L` ：每周星期天凌晨 1 点执行一次
- `0 26,29,33 * * * ?` ：在 26 分、29 分、33 分执行一次
- `0 0 0,13,18,21 * * ?` ：每天的 0 点、13 点、18 点、21 点都执行一次
- `0 0 10,14,16 * * ?` ：每天上午 10 点，下午 2 点、4 点执行一次
- `0 0/30 9-17 * * ?` ：朝九晚五工作时间内每半小时执行一次
- `0 0 12 ? * WED` ：每个星期三中午 12 点执行一次
- `0 0 12 * * ?` ：每天中午 12 点触发

- `0 15 10 ? * *` ：每天上午 10:15 触发
- `0 15 10 * * ?` ：每天上午 10:15 触发
- `0 15 10 * * ? *` ：每天上午 10:15 触发
- `0 15 10 * * ? 2005` ：2005 年的每天上午 10:15 触发
- `0 * 14 * * ?` ：每天下午 2 点到 2:59 期间的每 1 分钟触发
- `0 0/5 14 * * ?` ：每天下午 2 点到 2:55 期间的每 5 分钟触发
- `0 0/5 14,18 * * ?` ：每天下午 2 点到 2:55 期间和下午 6 点到 6:55 期间的每 5 分钟触发
- `0 0-5 14 * * ?` ：每天下午 2 点到 2:05 期间的每 1 分钟触发
- `0 10,44 14 ? 3 WED` ：每年三月的星期三的下午 2:10 和 2:44 触发
- `0 15 10 ? * MON-FRI` ：周一至周五的上午 10:15 触发
- `0 15 10 15 * ?` ：每月 15 日上午 10:15 触发
- `0 15 10 L * ?` ：每月最后一日的上午 10:15 触发
- `0 15 10 ? * 6L` ：每月的最后一个星期五上午 10:15 触发
- `0 15 10 ? * 6L 2002-2005` ：2002 年至 2005 年的每月的最后一个星期五上午 10:15 触发
- `0 15 10 ? * 6#3` ：每月的第三个星期五上午 10:15 触发

4.集群任务

4.1. 概述

集群任务是一种利用集群能力执行的任务，将一个大数据任务拆分为多个 Chunk (分片)，把这些 Chunk 发到 Worker (业务应用) 集群来执行，提高了并行处理的效率。



一个 Chunk（分片）的执行包括如下三个阶段，这三个阶段构建成一个循环。

- **Read 阶段**

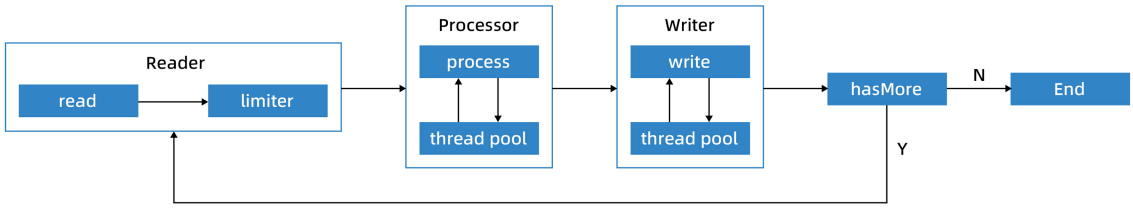
  Reader 读取一批数据，读取数据接口返回数据列表和后续是否有新数据的标志。Reader 根据 Limiter 的限流速率向 Processor 推送数据。

- **Process 阶段**

  Processor 接收到数据后，通过多线程的方式处理这批数据，处理完成后通过队列推送到 Writer。

- **Write 阶段**

  Writer 接收到数据后，通过多线程的方式对数据进行写操作。完成数据写操作后，Writer 根据是否有新数据的标志判断是否有新数据，如果显示有新数据，则会继续读取。

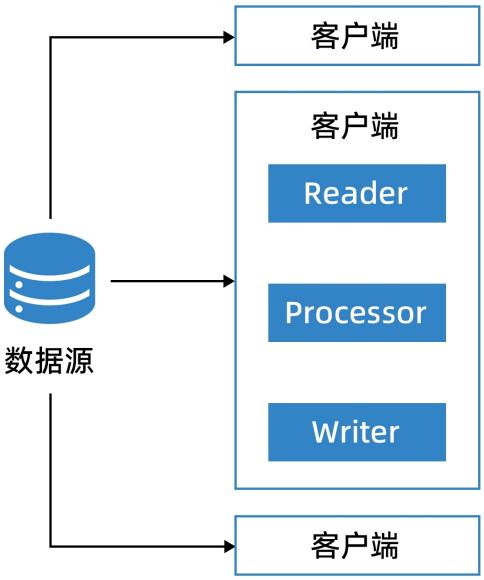


数据进行读写操作后，分为两种执行模式：本地执行和远程执行。

- **本地模式**

  一个 Chunk (分片) 里读写到的所有数据都在本地执行完成。

本机处理模式下，客户端读取出的数据直接在本机处理，可以获取到每条数据的实际处理结果。



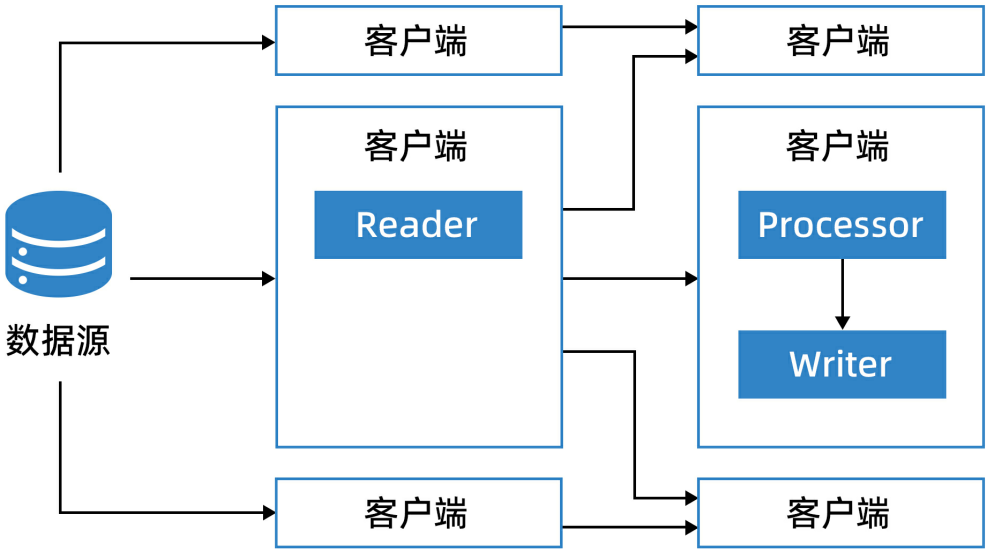
说明

该模式较为安全可靠，建议使用。

- **远程模式**

  将读写到的所有数据，按数据条数粒度发到整个集群里执行。

  远程处理模式下，客户端读取数据后，使用 Oneway 方式的 RPC 调用分发到集群里的其它机器，来执行 Process 和 Write 阶段，类似于三层分发框架的 Execute。这种模式无法获取到每条数据的实际处理结果，框架认为分发成功就处理成功，需要业务控制失败的记录，并再次处理。



> 说明
>
> 该模式提高了数据粒度的并行性，但也增加了 RPC 的开销，适用于给内部适配老系统。

4.2. 应用开发

4.2.1. 准备工作

本文介绍应用开发前的准备工作。

搭建工程

在开始使用任务调度产品之前，您需要根据实际需求，搭建以下类型的工程：

SOFABoot 工程

1. 下载并配置好一个 SOFABoot 工程原型。

   具体步骤参见 创建 Web 工程。

2. 对于 SOFABoot 3.4.3 以下的版本，需要添加如下 Tracer 依赖。

   > 说明
   >
   > - 添加前，请仔细检查工程是否已经依赖了 Tracer，避免 JAR 包冲突问题。
   > - SOFABoot 将在 3.4.3 及其以上版本自动依赖 Tracer，无需再手动添加。SOFABoot 最新版本信息，详见 SOFABoot 版本说明。

   ```
   <dependency>
       <groupId>com.alipay.sofa</groupId>
       <artifactId>tracer-enterprise-sofa-boot-starter</artifactId>
   </dependency>
   ```

3. 在 SOFABoot 工程的 `pom.xml` 文件中引入任务调度产品的依赖：

   ```
   <dependency>
       <groupId>com.alipay.sofa</groupId>
       <artifactId>scheduler-enterprise-sofa-boot-starter</artifactId>
   </dependency>
   ```

4. 修改 SOFABoot 全局配置项。

   环境参数配置详情，请参见 properties 配置项。

非 SOFABoot 工程

1. 配置工程的 Maven `settings.xml` 文件。

   在本地 Maven 路径下的 `settings.xml` 文件中，配置 `mvn.cloud.alipay.com` 仓库的地址及用户名密码。配置后工程即可通过 Maven 获取该仓库中任务调度的 JAR 依赖包，不配置则无法获取到 JAR 包。

> **说明**
>
> `mvn.cloud.alipay.com` 仓库的地址及用户名密码请 提交工单 获取。

2. 在工程的主 `pom.xml` 文件中引入 `antschedulerclient` 的依赖。

   以 1.0.7 版本为例：

```
<dependency>
    <groupId>com.alipay.antschedulerclient</groupId>
    <artifactId>antschedulerclient-core</artifactId>
    <version>1.0.7</version>
</dependency>
<dependency>
    <groupId>com.alipay.antschedulerclient</groupId>
    <artifactId>antschedulerclient-antvip</artifactId>
    <version>1.0.7</version>
</dependency>

<!-- 间接依赖 -->
<dependency>
    <groupId>com.antcloud.antvip</groupId>
    <artifactId>antcloud-antvip-client</artifactId>
    <version>1.0.4</version>
</dependency>
<dependency>
    <groupId>com.antcloud.antvip</groupId>
    <artifactId>antcloud-antvip-common</artifactId>
    <version>1.0.4</version>
</dependency>
<!-- antvip-common的依赖 -->
<dependency>
    <groupId>commons-codec</groupId>
    <artifactId>commons-codec</artifactId>
    <version>1.4</version>
</dependency>
<dependency>
    <groupId>commons-io</groupId>
    <artifactId>commons-io</artifactId>
    <version>2.2</version>
</dependency>
<dependency>
    <groupId>commons-lang</groupId>
    <artifactId>commons-lang</artifactId>
    <version>2.4</version>
</dependency>
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>fastjson</artifactId>
    <version>1.2.50</version>
</dependency>

<dependency>
    <groupId>com.alipay.sofa.cloud</groupId>
```

```
    <groupId>com.alipay.sofa.cloud</groupId>
    <artifactId>sdk-core</artifactId>
    <version>0.1</version>
</dependency>
<dependency>
    <groupId>com.alipay.sofa.common</groupId>
    <artifactId>sofa-common-tools</artifactId>
    <version>1.0.12</version>
</dependency>

<dependency>
    <groupId>com.alipay.sofa</groupId>
    <artifactId>bolt</artifactId>
    <version>1.4.1</version>
</dependency>
<dependency>
    <groupId>com.alibaba.toolkit.common</groupId>
    <artifactId>toolkit-common-lang</artifactId>
    <version>1.1.1</version>
</dependency>
<dependency>
    <groupId>com.alipay.sofa</groupId>
    <artifactId>hessian</artifactId>
    <version>3.3.0</version>
</dependency>
<dependency>
    <groupId>com.alipay.sofa.lookout</groupId>
    <artifactId>lookout-api</artifactId>
    <version>1.4.3</version>
</dependency>
<dependency>
    <groupId>com.alipay.sofa</groupId>
    <artifactId>tracer-core</artifactId>
    <version>2.1.1</version>
</dependency>
<dependency>
    <groupId>com.alipay.sofa</groupId>
    <artifactId>tracer-extensions</artifactId>
    <version>2.1.1</version>
</dependency>
<dependency>
    <groupId>com.alipay.common</groupId>
    <artifactId>tracer</artifactId>
    <version>2.1.1</version>
</dependency>
<dependency>
    <groupId>com.google.guava</groupId>
    <artifactId>guava</artifactId>
    <version>20.0</version>
</dependency>
```

3. 获取环境配置参数。

   环境参数配置详情，请参见 properties 配置项。

4. 初始化客户端 Client。

   示例如下：

```
@Override
public void onApplicationEvent(ContextRefreshedEvent event) {
    try {
        ApplicationContext applicationContext = event.getApplicationContext();
        Map<String, IJobHandler> handlerBeans = applicationContext
                .getBeansOfType(IJobHandler.class);
        for (IJobHandler handler : handlerBeans.values()) {
            handlerList.add(handler);
        }
// zone(cell),如果为空则认为不分zone(cell)
        zone = System.getProperty("com.alipay.ldc.zone");

        Config config = new Config(app, zone, instanceId, accessKey, securityKey);
        AntVipRouterConfig routerConfig = new AntVipRouterConfig();
        routerConfig.setEndPoint(antvipEndpoint);
        AntVipRouter router = new AntVipRouter(routerConfig);
        antSchedulerClient = new AntSchedulerClient(config, router, handlerList);
        antSchedulerClient.init();
    } catch (Throwable t) {
        LOGGER.error("init antschedulerclient error", t);
        throw new RuntimeException(t);
    }
}
```

下载示例代码

为快速进行任务调度项目的本地工程开发，您需要下载任务调度 示例工程。

4.2.2. 任务拆分和执行

集群任务支持用户按业务的要求，通过多层的拆分将一个任务拆分到多个客户端上并发执行。

集群任务的开发可以分成两个阶段：拆分阶段和执行阶段。

- **拆分阶段**：对数据进行分片，不限制拆分层数，将拆分结果上报给服务端，由服务端根据拆分的 chunk（一批待处理数据的索引集合）通知客户端来拉取数据进行处理。
- **执行阶段**：客户端接收到通知后拉取数据进行处理，处理完后继续拉取新的数据，直到数据都处理完成。

应用场景案例

为了便于理解，本文使用一个场景示例来介绍集群任务的开发过程。

假设某基金公司每天需要进行一次用户清算，由于用户规模较大，因此将用户分为 100 张表，每张表有 10 万左右的用户数据。该公司选择使用两层拆分的集群任务，通过集群任务并行处理能力提高数据处理效率。

具体的实现步骤如下：

1. 任务拆分阶段：将用户数据进行拆分，详情请参见 集群任务拆分阶段。
   - 第一层拆分：按用户表维度进行数据拆分。
   - 第二层拆分：按分页维度进行数据拆分。

2. 任务执行阶段：对每个用户数据进行处理，详情请参见 集群任务执行阶段。
   - 执行模式：支持本地执行模式以及远程执行模式。
   - 线程池配置：支持使用默认线程池配置或自定义线程池配置。
3. 控制台配置集群任务：详情请参见 创建调度任务。

任务拆分阶段

任务调度提供了 `IClusterJobSplitHandler` 接口进行任务拆分，目前支持两种拆分方式：

- ShardingChunkData 拆分：指定索引的拆分方式，通过索引对这个子任务（chunk）进行唯一标识，客户端会根据指定的索引拉取数据进行处理。要求填充分片规则。

```
public class ShardingChunkData implements IChunkData {
    /**
     * 分片号
     */
    private String shardingRule;

......
}
```

> 说明
>
> 具体使用代码，可参考 示例工程 中的 `ClusterFstSplitHandler` 。

其中， `shardingRule` 是一个分片规则标识，例如可以使用 `user_00`、`user_01` … `user_99` 作为分片标识。结合上述场景案例，第一层任务拆分可以使用 `shardingChunkData` 对 100 张用户表进行拆分，具体代码如下：

```
public class UserSplitterByTable implements IClusterJobSplitHandler<ShardingChunkData> {

    @Override
    public SplitChunkDataResult<ShardingChunkData> handle(ClusterJobSplitContext context)
{
        SplitChunkDataResult<ShardingChunkData> splitChunkDataResult = new SplitChunkData
Result<>();
        ArrayList<ShardingChunkData> shardingChunkDatas = new ArrayList<>();
        // user_00 ~ user_99
        for (int i = 0; i < 100; i++) {
            String shardingRule = "user_";
            if (i < 10) {
                shardingRule = shardingRule +"0";
            }
            shardingRule = shardingRule + i;
            // 根据表维度拆分分片。
            ShardingChunkData shardingChunkData = new ShardingChunkData(shardingRule);
            shardingChunkDatas.add(shardingChunkData);
        }
        splitChunkDataResult.setChunkDatum(shardingChunkDatas);
        splitChunkDataResult.setSuccess(true);
        return splitChunkDataResult;
    }

    @Override
    public String getName() {
       // 处理器名称。
        return "USER_SPLITTER_BY_TABLE";
    }

    @Override
    public ThreadPoolExecutor getThreadPool() {
       // 线程池，建议使用自定义线程池。如果返回 null 则使用 SDK 自带的线程池。
        return null;
    }
}
```

- RangeChunkData 拆分：指定范围的拆分方式，指定每个子任务处理特定范围内的数据，类似于一个分页效果。要求填充起始索引、结束索引和分片规则。

```
public class RangeChunkData implements IChunkData {
    /**
     * 分片规则
     */
    private String shardingRule;

    /**
     * 起始索引
     */
    private String start;

    /**
     * 结束索引
     */
    private String end;

......
}
```

> **说明**
>
> 具体使用代码，可参考 工程示例 中的 `ClusterSecSplitHandler` 。

参数说明：

- `shardingRule` ：分片规则，例如指定表 `user_01` 。

  例如，以下配置表示处理 `user_01` 表中 1000 到 2000 的数据。

  ```
  RangeChunkData chunk =new RangeChunkData ();
  chunk.setShardingRule("user_01");
  chunk.setStart("1000");
  chunk.setEnd("2000");
  ```

- `start` ：开始索引，记录开始位置。例如 1000。
- `end` ：结束索引，记录结束位置（包含）。例如 2000。

在场景案例中，第二层可以使用 RangeChunkData 将用户数据按分页维度进行拆分，具体代码如下：

```
public class UserSplitterByPage implements IClusterJobSplitHandler<RangeChunkData> {

    @Override
    public SplitChunkDataResult<RangeChunkData> handle(ClusterJobSplitContext context) {
        // 第一层拆分的分片
        ShardingChunkData shardingChunkData = (ShardingChunkData) context.getChunkData();
        String shardingRule = shardingChunkData.getShardingRule();

        // 1. 根据分片查询数量
        int count = queryCountByTable(shardingChunkData.getShardingRule());
        SplitChunkDataResult<RangeChunkData> splitChunkDataResult = new SplitChunkDataRes
ult<>();
        ArrayList<RangeChunkData> shardingChunkDatas = new ArrayList<>();

        // 2. 做分页处理，每页处理 1000 条
        int pageCount = 1000;
        for (int page = 0; page < count / pageCount; page++) {
            String startRows = String.valueOf(page * pageCount);
            // 包含
            String endRows = String.valueOf((page + 1) * pageCount - 1);
            RangeChunkData rangeChunkData = new RangeChunkData(shardingRule, startRows, e
ndRows);
            shardingChunkDatas.add(rangeChunkData);
        }

        splitChunkDataResult.setChunkDatum(shardingChunkDatas);
        splitChunkDataResult.setSuccess(true);
        return splitChunkDataResult;
    }

    // mock
    private int queryCountByTable(String shardingRule) {
        if ("user_00".equals(shardingRule)) {
            // user_00 表 10 万用户
            return 100000;
        } else {
            // 其他表 9 万用户
            return 90000;
        }
    }

    @Override
    public String getName() {
        return "USER_SPLITTER_BY_PAGE";
    }

    @Override
    public ThreadPoolExecutor getThreadPool() {
        return null;
    }
}
```

任务执行阶段

集群任务执行阶段分为三个子阶段：

- Read 阶段：从数据源读取数据。

  读取数据服务需要实现接口 `IReader` ：

```
public interface IReader<T>{

     /**
     * 读取数据
     *
     * @param context
     * @return
     */
        LoadData<T> read(ClusterJobExecuteContext context) throws Exception;
}
```

  LoadData 对象数据结构如下：

```
public class LoadData<T> extends MultiDataItem<T>{

        /**
         * 是否还有未捞取的待处理数据：值为 false 时，处理完本次捞取出的数据就回调服务端；值为 true 时，认为还有数据待处理，处理完本次捞取数据后继续捞取数据。
         */
        private boolean hasMore;

        public LoadData(List<T> itemList,boolean hasMore){
            super(itemList);
            this.hasMore=hasMore;
        }

        public static boolean isEmpty(LoadData loadData){
            return loadData==null || loadData.isEmpty();
        }

        public boolean isHasMore(){
            return hasMore;
        }

        public void setHasMore(boolean hasMore){
            this.hasMore=hasMore;
        }
}
```

  使用 read 方法分批读取数据时，需要注意设置 `hasMore` 的值，当值为 true 时，处理完本次读取的数据后会再次进行读取数据。

  如果需要打开关闭文件，可以使用 `IStreamReader` 接口：

```
public interface IStream {

    /**
     * 打开流
     *
     * @param context
     * @throws Exception
     */
    void open(ClusterJobExecuteContext context) throws Exception;

    /**
     * 关闭流
     *
     * @param context
     * @throws Exception
     */
    void close(ClusterJobExecuteContext context) throws Exception;
}

public interface IStreamReader<T> extends IStream, IReader<T> {

}
```

该接口继承自 `IStream` 接口，提供了 open 和 close 方法，这两个方法分别会在执行分片前和执行分片后调用。

- Process 阶段：将读取的数据对象转换为要写入数据源的对象，该阶段不是必须阶段，可以不设置。

  处理数据服务需要实现接口 `IProcessor`：

```
public interface IProcessor<I, O> {

    /**
     * 数据处理，将读取出的对象加工转换为要处理的对象
     *
     * @param r
     * @return
     */
    DataProcessResult<O> process(ClusterJobExecuteContext context, I i) throws Exception;

    /**
     * 数据处理线程池，没有配置时使用handler的线程池
     *
     * @return
     */
    ThreadPoolExecutor getProcessThreadPool();
}
```

  当不需要对读取的数据进行转换时，可以不设置该服务，这样会直接把读取的数据交给写服务。该服务可以单独指定线程池，当没有设置时默认使用任务 handler 的线程池。

- Write 阶段：将数据写入数据源。

> **说明**
>
> 集群任务 Write 阶段的相关代码，可参考 工程示例 中的 `ClusterExecuteHandler` 。

写数据服务需要实现接口 `IWriter` ：

```
public interface IWriter<T> {

    /**
     * 每次执行的数据块包含的数据量最大值
     *
     * @return
     */
    int getCountPerWrite();

    /**
     * 写数据
     *
     * @param context
     * @param dataItem
     * @return
     * @throws Exception
     */
    ClientCommonResult write(ClusterJobExecuteContext context,
                             IDataItem<T> dataItem) throws Exception;

    /**
     * 写数据线程池，没有配置时使用handler的线程池
     *
     * @return
     */
    ThreadPoolExecutor getWriteThreadPool();
}
```

`getCountPerWrite` 方法用来设置每次批量 write 的数据量，值小于等于 0 时认为一次只写入一条数据。该服务可以单独设置线程池，当没有设置时默认使用任务 handler 的线程池。

`IDataItem` 就是待处理的数据对象，有两种类型：

- `SingleDataItem` ：只包含一条数据，当 `getCountPerWrite` 方法的返回值小于或等于 1 时是该类型。
- `MultiDataItem` ：包含多条数据，当 `getCountPerWrite` 方法的返回值大于 1 时是该类型。

可根据实际情况对 `dataItem` 进行类型转换，例如：

```
switch(dataItem.getType()){
    case SINGLE:
         ((SingleDataItem<Integer>)dataItem).getItem();
         break;
    case MULTIPLE:
         (MultiDataItem<Integer>)dataItem).getItemList();
         break;
    default:
         break;
}
```

如果需要打开关闭文件，可以使用接口 `IStreamWriter`：

```
public interface IStreamWriter<T> extends IStream,IWriter<T>{

}
```

该接口继承了 `IStream` 接口，提供了 open 和 close 方法，这两个方法分别会在执行分片前和执行分片后调用。

代码示例

结合上述应用场景案例，在执行阶段，需要根据拆分的分片从数据库或其他数据源中拉取数据进行处理，然后落库。

- **Read 阶段**：根据分片信息（shardingRule、start、end）到指定表里拉取指定范围的用户数据。
- **Process 阶段**：对拉取到的用户数据进行处理。
- **Write 阶段**：将处理后的数据落库。

具体代码如下：

```
public class UserProcessHandler implements IClusterJobExecuteHandler {
    private final Logger LOGGER = LoggerFactory.getLogger(ClusterJobExecutor.class);
    private ThreadPoolExecutor threadPool;

    @Override
    public void preExecute(ClusterJobExecuteContext context) {
        // 前置处理
        LOGGER.info(String.format("preExecute chunkId:%s, param:%s",
                context.getChunkId(), context.getCustomParams()));
    }

    @Override
    public IReader getReader() {
        return new IReader() {
            @Override
            public LoadData<String> read(ClusterJobExecuteContext context) throws Exception
{
                IChunkData chunkData = context.getChunk();
                RangeChunkData rangeChunkData = (RangeChunkData) chunkData;

                // 根据索引去查询数据
                //  您可以根据开始和结束索引，从数据库中分页查询。集群任务按照您设置的分片规则进行分
片，然后去执行分片。具体的分片内容和执行逻辑，由您自行实现。
```

```
                String shardingRule = rangeChunkData.getShardingRule();
                int start = Integer.parseInt(rangeChunkData.getStart());
                int end = Integer.parseInt(rangeChunkData.getEnd());

                // 模拟查询数据
                List<String> stringList = queryUserInfo(shardingRule, start, end);

               // 一次查询，后面没有数据
                boolean hasMore = false;
                return new LoadData<String>(stringList, hasMore);
            }

            private List<String> queryUserInfo(String shardingRule, int start, int end) {
                List<String> stringList = Lists.newArrayList();
                for (int userId = start; userId <= end; userId++) {
                    stringList.add(shardingRule + "" + userId);
                }
                return stringList;
            }
        };
    }

    @Override
    public IProcessor getProcessor() {
        return new IProcessor() {
            @Override
            public DataProcessResult process(ClusterJobExecuteContext context,
                                             Object data) throws Exception {
                // 处理数据
                System.out.println("process user" + data);
                return new DataProcessResult(true, "", data);
            }

            @Override
            public ThreadPoolExecutor getProcessThreadPool() {
                return null;
            }
        };
    }

    @Override
    public IWriter getWriter() {
        return new IWriter<String>() {
            @Override
            public int getCountPerWrite() {
                return 1;
            }

            @Override
            public ClientCommonResult write(ClusterJobExecuteContext context,
                                            IDataItem<String> dataItem)throws Exception {
                // 数据存储
                switch (dataItem.getType()) {
                    case SINGLE:
                        // 单个数据块，只包含一条数据
```

```
                        // 单个数据块，只包含一条数据
                        SingleDataItem<String> singleDataItem = (SingleDataItem<String>) da
taItem;
                        LOGGER.info(String.format("getWriter write single data:%s", singleD
ataItem.getItem()));
                        System.out.println(String.format("write single data:%s", singleData
Item.getItem()));
                        break;
                    case MULTIPLE:
                        // 复合数据块，包含多条数据。比如任务停止时会将多条数据传入
                        MultiDataItem<String> multiDataItem = (MultiDataItem<String>) dataI
tem;
                        LOGGER.info(String.format("getWriter write multi data:%s", multiDat
aItem.getItemList()));
                        break;
                    default:
                        break;
                }
                return ClientCommonResult.buildSuccessResult();
            }

            @Override
            public ThreadPoolExecutor getWriteThreadPool() {
                return BlockingThreadPool.getThreadPool();
            }
        };
    }

    @Override
    public ILimiter getLimiter() {
        return new DefaultLimiter(10);
    }

    @Override
    public void postExecute(ClusterJobExecuteContext context) {
// 后置处理
        LOGGER.info(String.format("JobExecuteHandler postExecute chunkId:%s, param:%s",
                context.getChunkId(), context.getCustomParams()));
    }

    @Override
    public Progress calProgress(ClusterJobExecuteContext context) {
        return new Progress();
    }

    @Override
    public boolean isProcessAsync() {
        return true;
    }

    @Override
    public String getName() {
        return "USER_PROCESS_HANDLER";
    }
```

```
    @Override
    public ThreadPoolExecutor getThreadPool() {
        return this.threadPool;
    }

    public void setThreadPool(ThreadPoolExecutor threadPool) {
        this.threadPool = CommonThreadPool.getThreadPool("REMOTE_EXECUTE");
    }
}
```

执行模式

集群任务的执行阶段支持两种模式：

本机执行

本机执行时需要实现接口 `IClusterJobExecuteHandler` ，如下：

```
public interface IJobHandler {

    /**
     * handler的名字
     *
     * @return
     */
    String getName();

    /**
     * 可以留空，使用默认执行线程池
     *
     * @return
     */
    ThreadPoolExecutor getThreadPool();

}

public interface IClusterJobExecuteHandler<I, O> extends IJobHandler {

    /**
     * 预处理
     *
     * @param context
     */
    void preExecute(ClusterJobExecuteContext context);

    /**
     * 获取数据读取服务
     *
     * @return
     */
    IReader<I> getReader();

    /**
     * 获取数据清洗服务，返回NULL时会跳过数据清洗步骤，直接处理读取出的数据
```

```
     *
     * @return
     */
    IProcessor<I, O> getProcessor();

    /**
     * @return
     */
    IWriter<O> getWriter();

    /**
     * 获取限流器，返回NULL时使用默认限流器
     *
     * @return
     */
    ILimiter getLimiter();

    /**
     * 本次执行完的后置处理
     *
     * @param context
     * @return
     */
    void postExecute(ClusterJobExecuteContext context);

    /**
     * 计算处理进度
     *
     * @param context
     * @return
     */
    Progress calProgress(ClusterJobExecuteContext context);

    /**
     * 是否异步处理，开启异步执行后，reader-process-write阶段全部是异步处理
     *
     * @return
     */
    boolean isProcessAsync();
}
```

实现 `IClusterJobExecuteHandler` 的类负责处理任务。实现时需要设置：

- IReader
- IWriter
- IProcessor：非必须设置，当没有设置 IProcessor 时会把读取出的数据直接交给 IWriter 服务。
- ILimiter：没有设置时使用默认的限流服务。

当 `isProcessAsync` 返回 `true` 时，Reader、Process、Write 阶段全部是异步处理，即读取的数据放到队列给 Process 或 Write 消费后会立即开始下一次读取；当返回 `false` 时，读取的数据需要等待被 Write 消费完后开始下一次读取。

> **重要**
>
> IReader 服务如果配置成了 bean，需要确保服务是无状态的，否则多线程场景下会相互干扰。如果需要存储一些处理过程中的数据，需要在 `getReader` 方法里新建一个 IReader 服务实例。IWriter 服务也是如此。

远程执行

远程执行时需要实现接口 `IRemoteClusterJobExecuteHandler` ，如下：

```
public interface IRemoteClusterJobExecuteHandler<T> extends IClusterJobExecuteHandler<T>{

        /**
         * 设置路由策略，没有设置时默认为轮询
         *
         * @return
         */
        IClusterRouter getClusterRouter();

        /**
         * 分发完一次load出来的数据后的休眠时间，防止分发出去的数据还没有被处理完导致的重复捞取，单位是ms，返回值<=0时认为不休眠
         *
         * @return
         */
        int getSleepAfterPerLoad();
}
```

该接口继承自 `IClusterJobExecuteHandler` ，需要额外两个方法： `getClusterRouter` 和 `getSleepAfterPerLoad` 。

- 当远程处理数据时，客户端会把捞取的数据分发给业务集群内的其它机器进行处理， `getClusterRouter` 方法用来设置分发时的路由规则，目前支持随机（ `RandomClusterRouter` ）和轮询（ `RoundRobinClusterRouter` ）两种方式，当方法返回 `null` 时默认为轮询。
- `getSleepAfterPerLoad` 方法用来设置处理完读取的数据后的休眠时间，防止分发出去的数据还没有被处理完导致的重复捞取，单位为 ms，仅当 `isProcessAsync` 返回 `false` 时生效。

集群内的远程 RPC 调用使用 oneway 模式，所以无法获取数据的处理结果，需要业务方自己记录处理结果。如需帮助，请 提交工单 咨询。

可以通过实现接口 `IRemoteProcessorExecutor` 来设置接收远程分发请求的线程池，将实现类发布成 bean 即可，接口如下：

```
public interface IRemoteProcessorExecutor{

/**
     * 获取线程池
     *
     * @return
     */
     Executor getExecutor();
}
```

另外，远程调用需要在客户端启动一个 RpcServer：

- 对于 SOFABoot 项目，在工程的 `application.properties` 文件中使用配置项：

```
com.alipay.sofa.antscheduler.remote.execute.enable=true
com.alipay.sofa.antscheduler.remote.execute.port=xxx
```

- 对于非 SOFABoot 项目，需要在初始化客户端的时候设置下面两个参数：

```
public class Config{

......

/**
   * 是否支持远程执行
   */
private boolean isEnableRemoteExecute =false;

/**
   * 远程服务端口，没有设置时使用默认端口9989
   */
private int  remoteServerPort;
......
}
```

线程池默认配置

IProcessor 和 IWriter 的线程池没有设置时，默认使用 IJobHandler 里设置的线程池。当 IJobHandler 也没有设置时，使用的是客户端默认指定的，配置参数为：

```
minPoolSize: 20
maxPoolSize: 300
queueSize: 100
keepAliveTime: 1小时
```

当没有设置 IRemoteProcessExecutor 时，默认使用 bolt 协议提供的默认线程池，配置参数为：

```
minPoolSize: 20
maxPoolSize: 400
queueSize: 600
keepAliveTime: 60秒
```

您也可以对 IProcessor、IWriter、IJobHandler 以及 IRemoteProcessExecutor 自定义设置线程池。

4.2.3. 创建和发布应用

工程创建完成后，您需要在 SOFAStack 控制台创建和发布应用。

操作步骤

1. 创建应用。
   i. 登录 SOFAStack 控制台。
   ii. 在左侧导航栏选择 **应用管理** > **应用列表**。
   iii. 单击 **创建应用**，然后配置以下参数：

参数	说明
应用名	配置应用名称。 可以包含字母、数字、连字符、下划线、且必须以字母开头，以字母或者数字结尾。
技术栈	根据项目中使用的 Java 版本选择对应的 SOFABoot 技术栈。
应用分组	选择应用所在分组。
应用标签	配置应用的标签： ■ **标签类型**：可选择 **系统标签** 或 **自定义标签**。 ■ **应用分级**：应用的分级信息。 ■ **变量值**：对应的变量值信息。
应用描述（可选）	配置应用的描述信息。

   iv. 单击 **提交**。
2. 上传发布包。
   i. 生成发布包。

      在本地执行 `mvn clean package` 或者 `mvn clean install` 命令，在工程根目录下的 `target` 下打包生成一个 fat JAR 包
   ii. 在 SOFAStack 控制台左侧导航栏，选择 **经典应用服务** > **应用发布** > **发布包管理**。
   iii. 找到步骤 1 创建的应用，然后单击右侧的 **管理**。
   iv. 单击 **上传发布包**，然后上传应用的发布包后，单击 **确认**。
3. 创建应用服务实例并关联发布包。
   i. 在左侧导航栏选择 **经典应用服务** > **应用发布** > **应用服务**。

ii. 单击 **创建服务实例**，然后配置以下参数：

参数	说明
应用	在下拉列表选择步骤 1 创建的应用。
服务实例名称	配置服务实例名称。 可以包含字母、数字、连字符、下划线、且必须以字母开头，以字母或者数字结尾。
发布包	选择步骤 2 上传的发布包。
描述（可选）	配置应用实例的描述信息。
技术栈版本	在下拉列表选择技术栈版本。
部署单元	选择应用的部署单元。

iii. 单击 **确定**。

4. 发布应用。
   i. 在左侧导航栏选择 **经典应用服务** > **应用发布** > **发布包管理**。
   ii. 选中步骤 3 创建的服务实例，然后选择 **运维操作** > **发布部署**。
   iii. 单击 **立即发布**，然后单击 **整体发布**。

4.3. 创建调度任务

应用发布后，您可以为应用创建调度任务。

操作步骤

1. 登录任务调度控制台。
2. 在左侧导航栏选择 **中间件** > **任务调度** > **任务配置**。
3. 选择 **添加任务** > **集群任务**，然后配置以下参数：

参数	说明
任务名称	配置任务的名称，用于识别任务。

应用名称	填写或选择目标应用名称。 支持英文、中文、数字、下划线（_）、短划线（-），50 个字符之内。必须与工程中 `application.properties` 文件的 `spring.application.name` 配置的应用名称一致。
调度类型	配置任务调度的类型，可选值： ○ **cron 表达式触发**：该类型的任务在创建时需要配置 CRON 表达式定时触发任务。 ○ **事件触发**：该类型的任务通过外部事件触发任务。 ○ **定时触发**：该类型的任务在创建时也需要配置 CRON 表达式定时触发，并且只能执行一次。
Cron 表达式	配置 CRON 表达式，用于定时触发任务。配置方式请参见 CRON 表达式详解。 仅在 **调度类型** 为 **cron 表达式触发** 时配置。
任务拆分	输入拆分处理器和执行处理器的名称。获取方式请参见 任务拆分和执行。
任务分类（可选）	选择或新建一个任务分组类，方便归纳管理任务。
自定义参数（可选）	自定义参数支持从控制台向任务传递参数，参数也可以在任务间传递，格式为：`key=value`。 参数配置说明如下： ○ **参数名**：设置自定义参数的名称。 支持英文大小写、数字、下划线（_），不能超过 128 字符。 ○ **类型** 和 **参数值**：可选类型和参数值配置方式如下。 ■ **STRING**：支持通用的字符串。 ■ **LIST**：支持字符串数组，例如：`["aaa","bbb"]`。 ■ **BOOLEAN**：支持 true 或 false。 ■ **PLACEHOLDER**：支持参数替换，目前支持的变量如下： ■ `${sharding}`：替换当前分片值，配合简单任务使用。 ■ `${shardingCount}`：替换简单任务配置的分片数。 ■ `${triggerTime}`：替换预期的触发时间。 **说明** 您也可以在业务端通过上下文 context（该对象必须实现 Serializable。）获取和覆盖自定义参数。代码参见下文 **获取和覆盖自定义参数示例代码**。

路由策略	选择执行任务的路由策略，可选值： ○ **随机**：每次执行都随机分发到客户端服务器上。 ○ **定向**：每次执行都分发到同一个客户端服务器上。 暂不支持指定客户端服务器 IP。 ○ **轮询**：每次执行依次分发到每台客户端服务器上。
通信方式	选择任务的通信方式，可选值： ○ **单向 ONEWAY**：单向操作，没有返回值。控制台不记录触发记录、无超时、无重试。 高频非重要任务时建议使用。 ○ **回调 CALLBACK**：双向操作，有返回值。控制台可查看触发记录、可设置超时、可重试。 重要任务时建议使用，触发间隔至少 5 分钟。
任务互斥	选择是否开启任务互斥模式。开启任务互斥模式时，处在“执行中”或“暂停”状态的任务即使已经达到下一个任务触发时间点，仍无法再次触发。本次调度状态以“跳过”标记。 默认为不开启。仅当 **通信方式** 设置为 **回调 CALLBACK** 时配置。
漏触发策略	设置任务漏触发时的处理策略，可选值： ○ **忽略**：不进行补偿处理。 ○ **立即触发**：立即重新触发。 ○ **时间充裕时触发**：是否进行补偿触发取决于下次触发前的剩余时间是否充裕。若当前时间 + 超时时间 < 下次触发时间，则进行补偿触发，并将此次补偿触发插入触发记录；反之，则不触发。多次漏触发进行恢复时仅进行一次补偿触发。 仅当 **通信方式** 设置为 **回调 CALLBACK**，且 **调度类型** 为 **cron 表达式触发** 时，可配置此项。
超时策略	设置任务执行超时后的处理策略，可选值： ○ **不处理**：触发记录和执行记录状态均标记为失败。 ○ **停止后续触发**：执行记录状态为失败，触发记录状态为超时，并且无法再次触发。仅当开启任务互斥模式时可选择此项。 仅当 **通信方式** 设置为 **回调 CALLBACK** 时配置。
超时时间	设置任务执行的超时时间，当任务超过此时间未回调则认为执行失败。单位可选择为 **分钟** 和 **小时**。 仅当 **通信方式** 设置为 **回调 CALLBACK** 时配置。

失败处理策略	设置任务执行失败后的处理策略，可选值： ○ **不重试**：失败后不进行重试。 ○ **最多重试三次**：失败后立即进行重试，但最多重试三次。
优先级	用于识别任务的重要性，暂无其他作用。 默认为 **中等**。
高级选项	选择是否配置高级选项。开启后，可根据需要配置如下参数： ○ 当 **通信方式** 设置为 **单向 ONEWAY** 时，可配置如下参数： **单台机器最大处理速率**：配置单台客户端在单位时间内处理的最大数据量。默认为机器最大处理速率。 ○ 当 **通信方式** 设置为 **回调 CALLBACK** 时，可配置如下参数： ■ **单台机器最大处理速率**：配置单台机器处理单个 chunk 时允许的最大处理速率，单位是 **条数据/秒**。该功能通过 Limiter 实现，未设置时认为不限速。 ■ **并行执行索引块数量上限**：配置任务维度允许并行处理 chunk 数量的上限，达到上限后客户端服务端不允许客户端再拉取未执行的 chunk。未设置时认为不限制。 ■ **单台并行执行索引块数量上限**：配置单台机器允许并行处理 chunk 数量的上限，达到上限后客户端不再拉取此任务的 chunk。默认值为 5 个。 ■ **单个分片处理失败跳过数据上限**：配置 Process 阶段数据失败后允许跳过的上限，达到上限后认为 chunk 执行失败。
描述（可选）	配置任务的描述信息，例如业务含义、影响范围等，最多 1024 个字符。

获取和覆盖自定义参数示例代码

```
public class SimpleTaskDemo implements ISimpleJobHandler {

    @Override
    public ClientCommonResult handle(JobExecuteContext context) {
       // 根据 key 获取 obj
        Object obj = context.getCustomParam("intObj");

        // 获取自定义参数
        Map<String, Object> paramsMap = context.getCustomParams();
        context.putCustomParams("num", 2);
        List<String> listparam = newArrayList <>();
        listparam.add("aaaaaa");
        listparam.add("111111");
        context.putCustomParams("intparam", 111);
        context.putCustomParams("stringparam", "dfadsfad1243");
        context.putCustomParams("listparam", listparam);
        // 注意对象必须实现 Serializable
        context.putCustomParams("objectParam", newObject());
        return ClientCommonResult.buildSuccessResult();
    }

....
}
```

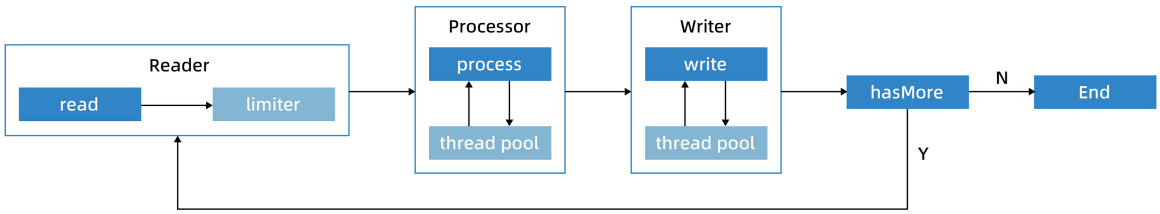
4. 单击 **提交**。
5. 在任务列表中打开目标任务的 **是否启用** 开关。

   如果有客户端在线，新配置的任务自动有客户端注册。任务在禁用的状态下，您也可以在操作列单击 **触发**，手动触发一次任务。

4.4. 集群任务限流

为避免负载过重，控制数据消费的速率，集群任务提供了动态和静态两种流控能力。

限流原理



如上图所示，一个 Chunk（分片）的执行分为如下三个阶段，这三个阶段构建成一个循环。

1. **Read 阶段**

Reader 读取一批数据，读取数据接口返回数据列表和后续是否有新的数据标志。

Reader 根据 limiter 的限流速率向 Processor 推送数据。

2. **Process 阶段**

   Processor 接收到数据后，通过多线程的方式对这批数据进行处理。处理后通过队列推送至 Writer。

3. **Write 阶段**

   Writer 接收到数据后，通过多线程的方式对数据进行写操作。数据写操作完成之后，Writer 根据是否有新数据标志进行判断是否有数据，如果还有数据，则会继续读取。

限流配置

为避免 Process 阶段和 Write 阶段线程处理过程复杂，数据存放到 Process 阶段的队列时会增加一个限流器，控制数据读取和存放的速率。任务调度提供了两种限流类型：

静态限流

静态限流通过代码配置限流规则，并可通过任务调度控制台动态调整限流速率。优势是在页面没有配置限流的情况下，也可以通过限流规则的初始值进行限速，但需要您手动指定限流器。

框架提供了默认的限流器 DefaultLimiter，指定限流器示例如下：

```
public class ClusterJobExecuteOneHandler implements IClusterJobExecuteHandler<Integer, Inte
ger> {

    ... // 省略

    @Override
    public ILimiter getLimiter() {
        // 每秒处理 10 条数据。
        return new DefaultLimiter(10);
    }

    ...   // 省略
}
```

如果此处设置为 null，且任务调度控制台没有设置限流规则，则表示不限速。

如果默认的限流器 DefaultLimiter 无法满足您的需求，例如需要根据业务逻辑、CPU、内存、IO 等方式进行限流时，限流模式提供了扩展接口，支持您自定义限流规则。您可以实现 `ILimiter` 接口，并在 `getLimiter()` 方法中返回实例。`ILimiter` 接口说明如下：

```
public interface ILimiter {

    /**
     * 设置限流速率，当 permitsPerSecond 为 NULL 或 ≤ 0 时表示不限速。
     *
     * @param permitsPerSecond
     */
    void setRate(Integer permitsPerSecond);

    /**
     * 获取许可，阻塞直到获取到许可。
     *
     * @return
     */
    void acquire();
}
```

- `setRate()` 方法：当管控出现一些配置变更时，框架会调用该方法告知用户。如果管控没有配置，则 `permitsPerSecond` 为 null。这部分的实现可以参考 DefaultLimiter 的实现逻辑。
- `acquire()` 方法：获取许可，如果不允许则阻塞。

如果想快速实现自定义限流器，可以参考 DefaultLimiter 的实现。

动态限流

动态限流通过任务调度控制台调整限流速率，不管是否提供限流器，动态限流都可以正常工作。如果没有指定限流器，任务调度客户端使用默认的 DefaultLimiter 限流器。配置方式有以下两种：

- 全局生效

  在配置或编辑任务调度时配置的限流规则全局生效，配置生效后，每次任务都会触发限速规则。

  配置任务调度的步骤请参见 创建调度任务。其中，高级选项需开启，且配置 **单台机器最大处理速率**。如图所示：

- 本次触发生效

  您可以在任务执行前或执行阶段对本次任务进行限速，操作如下：

  i. 登录任务调度控制台。

  ii. 在左侧导航栏选择 **中间件** > **任务调度** > **任务配置**。

  iii. 单击目标任务名称，然后单击 **调度记录** 页签。

iv. 在左侧选择正在进行的任务，然后单击 **处理数据量** 右侧的 图标。

v. 设置指定的处理速率后，单击 ✓ 图标。

限速规则设置完成后立刻生效，客户端会按指定的速率执行任务。

5.消息任务

5.1. 应用开发

5.1.1. 准备工作

本文介绍应用开发前的准备工作。

搭建工程

在开始使用文件批处理产品创建简单任务之前，您需要根据实际需求，搭建以下类型的工程：

SOFABoot 工程

1. 下载并配置好一个 SOFABoot 工程原型。

   具体步骤参见 创建 Web 工程。

2. 对于 SOFABoot 3.4.12 以下的版本，需要添加如下 Tracer 依赖。

   > 说明
   > - 添加前，请仔细检查工程是否已经依赖了 Tracer，避免 JAR 包冲突问题。
   > - SOFABoot 将在 3.4.12 及其以上版本自动依赖 Tracer，无需再手动添加。SOFABoot 最新版本信息，详见 SOFABoot 版本说明。

```
<dependency>
     <groupId>com.alipay.sofa</groupId>
     <artifactId>tracer-enterprise-sofa-boot-starter</artifactId>
</dependency>
```

3. 在 SOFABoot 工程的 `pom.xml` 文件中引入文件批处理产品的依赖：

```
  <dependency>
            <groupId>com.alipay.sofa</groupId>
            <artifactId>batch-enterprise-sofa-boot-starter</artifactId>
        </dependency>
```

4. 修改 SOFABoot 全局配置项。

   环境参数配置详情，请参见 properties 配置项。

下载示例代码

为快速进行任务调度的本地工程开发，您需要下载任务调度 示例工程。

5.1.2. 实现处理器接口和类

一个简单任务对应一个 handler，支持任务分片及分步：

- 当需要并发执行并且不关注资源利用率时，可以使用简单任务的分片功能。
- 当需要将一个任务分成若干步骤按顺序执行时，可以使用简单任务的分步功能。

实现处理器接口

```
/**
 * 简单任务处理器接口
 */
public interface ISimpleJobHandler extends IJobHandler {

    /**
     * 处理request
     *
     * @param context
     * @return
     */
    ClientCommonResult handle(JobExecuteContext context);
}
```

代码说明：

`JobExecuteContext` 是触发任务的请求上下文，可以从里面获取到任务的一些信息，比如：

- 在控制台创建任务时设置的自定义参数，可以通过 `getCustomParam(String key)` 方法获取。
- 如果设置了任务分片，可以通过方法 `getSharding()` 来获取当前机器分配到的分片号。

详细信息，可以查看 `com.alipay.antschedulerclient.model.JobExecuteContext` 这个类的源码。

实现类

类里必须实现以下三个方法：

- **handle**：在该方法里执行业务逻辑。
- **getThreadPool**：获取一个执行该任务的线程池，如果没有设置，会使用 Client 自带的线程池。

  ```
  new ThreadPoolExecutor(20,300,1,TimeUnit.HOURS,new ArrayBlockingQueue<Runnable>(100){})
  ```

- **getName**：获取这个执行器的名字，这个名字很重要，Client 接收到任务的触发请求时，会根据名字寻找匹配的执行器。

当任务需要按多个步骤执行时，需要写多个实现类，并在控制台开启分步功能。

代码示例

```
public class AlwaysSuccessHandler implements ISimpleJobHandler {

    public static final String NAME = "ALWAYS_SUCCESS_JOB";

    @Override
    public Client CommonResulthandle(JobExecuteContext context) throws InterruptedException
{
        //处理业务逻辑
        return ClientCommonResult.buildSuccessResult();
    }

    @Override
    public ThreadPoolExecutor getThreadPool() {
        return null;
    }

    @Override
    public String getName() {
        return NAME;
    }
}
```

5.1.3. 创建和发布应用

工程创建完成后，您需要在 SOFAStack 控制台创建和发布应用。

操作步骤

1. 创建应用。
   i. 登录 SOFAStack 控制台。
   ii. 在左侧导航栏选择 **应用管理** > **应用列表**。

iii. 单击 **创建应用**，然后配置以下参数：

参数	说明
应用名	配置应用名称。 可以包含字母、数字、连字符、下划线、且必须以字母开头，以字母或者数字结尾。
技术栈	根据项目中使用的 Java 版本选择对应的 SOFABoot 技术栈。
应用分组	选择应用所在分组。
应用标签	配置应用的标签： ■ **标签类型**：可选择 **系统标签** 或 **自定义标签**。 ■ **应用分级**：应用的分级信息。 ■ **变量值**：对应的变量值信息。
应用描述（可选）	配置应用的描述信息。

iv. 单击 **提交**。

2. 上传发布包。

   i. 生成发布包。

   在本地执行 `mvn clean package` 或者 `mvn clean install` 命令，在工程根目录下的 `target` 下打包生成一个 fat JAR 包。

   ii. 在 SOFAStack 控制台左侧导航栏，选择 **经典应用服务** > **应用发布** > **发布包管理**。

   iii. 找到步骤 1 创建的应用，然后单击右侧的 **管理**。

   iv. 单击 **上传发布包**，然后上传应用的发布包后，单击 **确认**。

3. 创建应用服务实例并关联发布包。

   i. 在左侧导航栏选择 **经典应用服务** > **应用发布** > **应用服务**。

ii. 单击 **创建服务实例**，然后配置以下参数：

参数	说明
应用	在下拉列表选择 步骤 1 创建的应用。
服务实例名称	配置服务实例名称。 可以包含字母、数字、连字符、下划线、且必须以字母开头，以字母或者数字结尾。
发布包	选择 步骤 2 上传的发布包。
描述（可选）	配置应用实例的描述信息。
技术栈版本	在下拉列表选择技术栈版本。
部署单元	选择应用的部署单元。

iii. 单击 **确定**。

4. 发布应用。
   i. 在左侧导航栏选择 **经典应用服务** > **应用发布** > **发布包管理**。
   ii. 选中 步骤 3 创建的服务实例，然后选择 **运维操作** > **发布部署**。
   iii. 单击 **立即发布**，然后单击 **整体发布**。

5.2. 创建消息任务

应用发布后，您可以为应用创建调度任务。

> **重要**
>
> 消息任务需要配合消息队列产品使用。

操作步骤

1. 登录任务调度控制台。
2. 在左侧导航栏选择 **任务配置**。
3. 选择 **添加任务** > **消息任务**，然后配置以下参数：

参数	说明
任务名称	配置任务的名称，用于识别任务。
应用名称	填写或选择目标应用名称。 支持英文、中文、数字、下划线（_）、短划线（-），50 个字符之内。必须与工程中 `application.properties` 文件的 `spring.application.name` 配置的应用名称一致。
消息事件码	填写消息时间码，请先到消息队列产品配置消息主题及消息事件码。
调度类型	配置任务调度的类型，可选值： ○ **cron 表达式触发**：该类型的任务在创建时需要配置 CRON 表达式定时触发任务。 ○ **事件触发**：该类型的任务通过外部事件触发任务。 ○ **定时触发**：该类型的任务在创建时也需要配置 CRON 表达式定时触发，并且只能执行一次。
Cron 表达式	配置 CRON 表达式，用于定时触发任务。配置方式请参见 CRON 表达式详解。 仅在 **调度类型** 为 **cron 表达式触发** 时配置。
任务分类（可选）	选择或新建一个任务分组，方便归纳管理任务。
是否分片	选择任务是否开启分片： ○ 不开启分片，任务将只调度到一台客户端服务器上执行。 ○ 开启分片，并设置分片数为 N，任务则会同时触发到 N 台客户端服务器上并发执行。 如果您开启了分片，您还需在本地工程编码实现任务分片。代码参见下文 **任务分片示例代码** 。

参数	说明
自定义参数（可选）	自定义参数支持从控制台向任务传递参数，参数也可以在任务间传递，格式为：`key=value`。 参数配置说明如下： ○ **参数名**：设置自定义参数的名称。 支持英文大小写、数字、下划线（_），不能超过 128 字符。 ○ **类型** 和 **参数值**：可选类型和参数值配置方式如下。 ■ **STRING**：支持通用的字符串。 ■ **LIST**：支持字符串数组，例如：`["aaa","bbb"]`。 ■ **BOOLEAN**：支持 true 或 false。 ■ **PLACEHOLDER**：支持参数替换，目前支持的变量如下： ■ `${sharding}`：替换当前分片值，配合简单任务使用。 ■ `${shardingCount}`：替换简单任务配置的分片数。 ■ `${triggerTime}`：替换预期的触发时间。 **说明** 您也可以在业务端通过上下文 context（该对象必须实现 Serializable。）获取和覆盖自定义参数。代码参见下文 **获取和覆盖自定义参数示例代码**。
优先级	用于识别任务的重要性，暂无其他作用。 默认为 **中等**。
描述（可选）	配置任务的描述信息，例如业务含义、影响范围等，最多 1024 个字符。

任务分片示例代码

```
public class ShardingSchedule implements ISimpleJobHandler {

    private final Logger LOGGER = LoggerFactory.getLogger(ShardingSchedule.class);

    private ThreadPoolExecutor threadPool;

    @Override
    public String getName() {
        return "SHARDING_SCHEDULE";
    }

    @Override
    public ClientCommonResult handle(JobExecuteContext jobExecuteContext) throws Exceptio
n {
        int sharding = jobExecuteContext.getSharding();

        //当前机器获取到的分片数
        LOGGER.info("current machine sharding is " + sharding);
        //总的分片数
        System.out.println("total sharding num is " + jobExecuteContext.getShardingCount(
));

        System.out.println("activity sharding is" + jobExecuteContext.getActivitySharding
());

        return ClientCommonResult.buildSuccessResult();
    }

@Override
    public ThreadPoolExecutor getThreadPool() {
        return threadPool;
    }

    /**
     * Setter method for property threadPool.
     *
     * @param threadPool value to be assigned to property threadPool
     */
    public void setThreadPool(ThreadPoolExecutor threadPool) {
        this.threadPool = threadPool;
    }

}
```

获取和覆盖自定义参数示例代码

```
public class SimpleTaskDemo implements ISimpleJobHandler {

    @Override
    public ClientCommonResult handle(JobExecuteContext context) {
       // 根据 key 获取 obj
        Object obj = context.getCustomParam("intObj");

        // 获取自定义参数
        Map<String, Object> paramsMap = context.getCustomParams();
        context.putCustomParams("num", 2);
        List<String> listparam = newArrayList <>();
        listparam.add("aaaaaa");
        listparam.add("111111");
        context.putCustomParams("intparam", 111);
        context.putCustomParams("stringparam", "dfadsfad1243");
        context.putCustomParams("listparam", listparam);
        // 注意对象必须实现 Serializable
        context.putCustomParams("objectParam", newObject());
        return ClientCommonResult.buildSuccessResult();
    }

....
}
```

4. 单击 **提交**。
5. 在任务列表中打开目标任务的 **是否启用** 开关。

   如果有客户端在线，新配置的任务自动有客户端注册。任务在禁用的状态下，您也可以在操作列单击 **触发**，手动触发一次任务。

6.拓扑任务

6.1. 概述

拓扑任务是一种特殊的任务，是通过事件触发的任务集合。拓扑任务中的任务执行流程均起始于开始节点，终止于结束节点，任务的执行流程形成一张有向无环图。

拓扑任务本身可以通过 CRON 表达式或者事件触发，但其中的任务节点必须符合以下要求：

- 触发方式必须是事件触发。
- 通信方式必须是回调 CALLBACK。
- 任务节点可以是简单任务、集群任务或其它拓扑任务。

一个子任务允许出现在不同的拓扑关系中，由于所属拓扑关系不同，执行时携带的上下文也不相同，不会产生交叉影响。即使手工或通过事件单独触发一个子任务，由于没有对应的图实例，也不会影响拓扑关系中的其他任务。

拓扑任务元素说明

开始与结束节点

所有拓扑任务有且仅有一个开始节点和一个结束节点。任务执行流程从开始节点起始，执行到结束节点终止。二者成对出现。

并行网关

并行网关用来在流程中实现并发，成对出现。一对并行网关间的所有分支被同时执行，不进行条件判断。并行网关的一个分支执行完毕后，需要等待其他分支全部执行完流程才会走到下一个节点。使用方法，参见 使用并行网关。

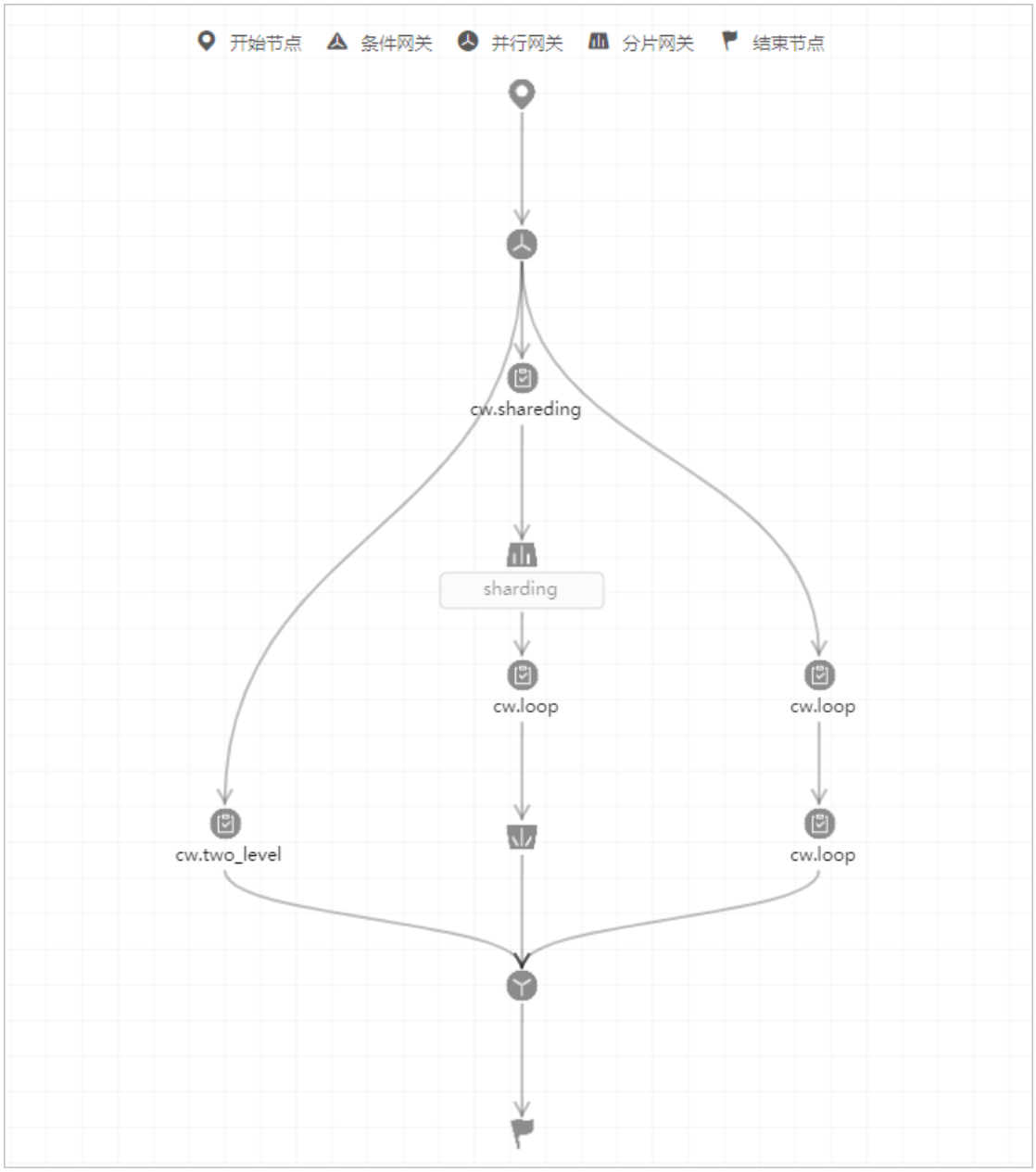
条件网关

条件网关，即排它网关，也叫异或（XOR）网关，用来在流程中实现决策，成对出现。一对排它网关包括一个判断条件和两个执行分支，分别对应 true 和 false 两个判断结果。判断结果为 true 的分支将被执行。使用方法，参见 使用条件网关。

分片网关

分片网关根据设定的分片维度对任务数据进行拆分，动态生成需要执行的分片数，成对出现。在拓扑图中使用分片网关时，需要在分片网关的开始节点配置分片维度。拓扑任务执行过程中，会根据拓扑任务执行上下文获取对应的分片配置，动态生成需要执行的分片。使用方法，参见 使用分片网关。

拓扑任务示例



6.2. 新建拓扑任务

操作步骤

1. 创建拓扑任务。
   i. 登录任务调度控制台。
   ii. 单击左侧导航栏的 **任务编排**。
   iii. 单击 **添加任务拓扑**，然后配置以下参数：

参数	说明
拓扑名称	配置拓扑任务的名称。
调度类型	配置任务调度的类型，可选值： ■ **cron 表达式触发**：该类型的任务在创建时需要配置 CRON 表达式定时触发任务。 ■ **事件触发**：该类型的任务通过外部事件触发任务。
Cron 表达式	配置 CRON 表达式，用于定时触发任务。配置方式请参见 CRON 表达式详解。 仅在 **调度类型** 为 **cron 表达式触发** 时配置。
任务互斥	选择是否开启任务互斥模式。开启任务互斥模式时，处在“执行中”或“暂停”状态的任务即使已经达到下一个任务触发时间点，仍无法再次触发。本次调度状态以“跳过”标记。
漏触发策略	设置任务漏触发时的处理策略，可选值： ■ **忽略**：不进行补偿处理。 ■ **立即触发**：立即重新触发。 仅当 **调度类型** 为 **cron 表达式触发** 时配置此项。
自定义参数（可选）	自定义参数支持从控制台向任务传递参数，参数也可以在任务间传递，格式为：`key=value`。参数配置说明如下： ■ **参数名**：设置自定义参数的名称。 支持英文大小写、数字、下划线（_），不能超过 128 字符。 ■ **类型** 和 **参数值**：可选类型和参数值配置方式如下。 ■ STRING：支持通用的字符串。 ■ LIST：支持字符串数组，例如：`["aaa","bbb"]`。 ■ BOOLEAN：支持 true 或 false。 ■ PLACEHOLDER：支持参数替换，目前支持的变量如下： ■ `${sharding}`：替换当前分片值，配合简单任务使用。 ■ `${shardingCount}`：替换简单任务配置的分片数。 ■ `${triggerTime}`：替换预期的触发时间。
描述（可选）	配置任务的描述信息，例如业务含义、影响范围等，最多 1024 个字符。

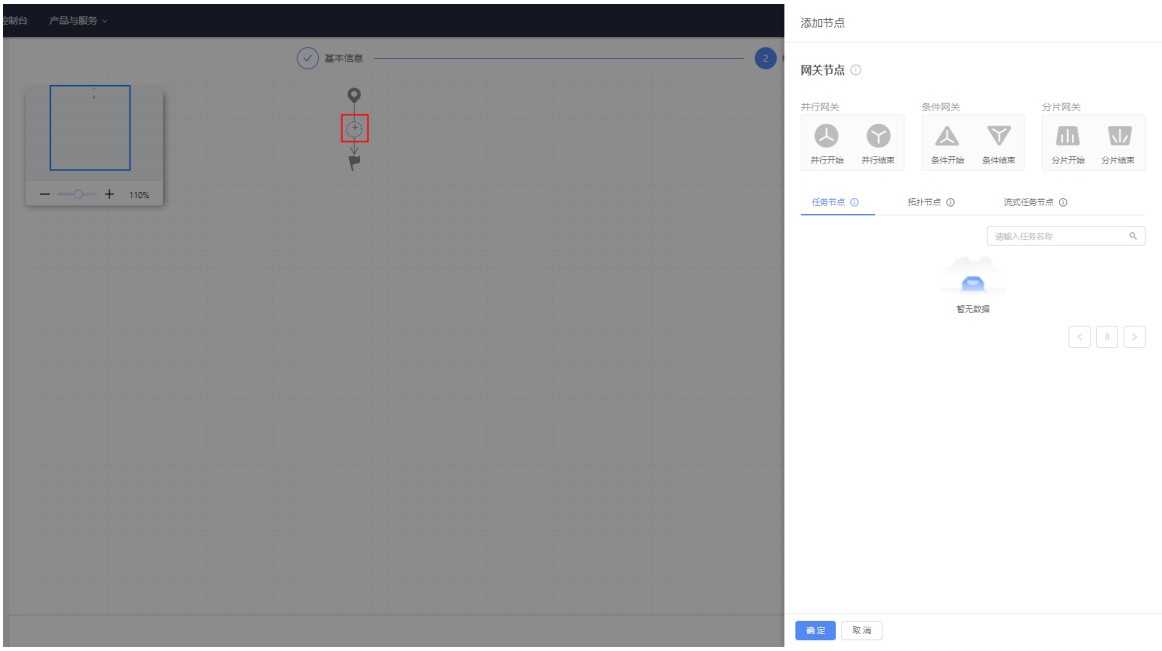
说明

关于自定义参数，您也可以在业务端通过上下文 context（该对象必须实现 Serializable。）获取和覆盖自定义参数。代码如下：

```
public class SimpleTaskDemo implements ISimpleJobHandler {

    @Override
    public ClientCommonResult handle(JobExecuteContext context) {
       // 根据 key 获取 obj
        Object obj = context.getCustomParam("intObj");

        // 获取自定义参数
        Map<String, Object> paramsMap = context.getCustomParams();
        context.putCustomParams("num", 2);
        List<String> listparam = newArrayList <>();
        listparam.add("aaaaaa");
        listparam.add("111111");
        context.putCustomParams("intparam", 111);
        context.putCustomParams("stringparam", "dfadsfad1243");
        context.putCustomParams("listparam", listparam);
        // 注意对象必须实现 Serializable
        context.putCustomParams("objectParam", newObject());
        return ClientCommonResult.buildSuccessResult();
    }

....
}
```

2. 编辑拓扑图。

   i. 单击 **下一步**，然后单击拓扑图中的加号（+）。



ii. 在 **添加节点** 面板添加网关、任务或拓扑子流程后，单击 **确定**。

- 在 **网关节点** 区域，选择要添加的网关类型。网关成对出现。
  - **并行网关**：并行网关间的所有分支被同时执行。更多信息，请参见 并行网关。
  - **条件网关**：需要设置边条件，例如 isSuccess。拓扑执行时，将对该网关下的执行分支进行条件判断，判断结果为 true 的分支会被执行。更多信息，请参见 条件网关。
  - **分片网关**：需要设置分片规则。拓扑执行时会根据规则对任务数据进行拆分，生成需要执行的分片数。更多信息，请参见 分片网关。

> **重要**
>
> 如果您开启了 LDC 共享，且开启了跨 zone 调度，此处仅可以创建跨 zone 网关。详情请参见 跨 zone 网关。

- 在 **任务列表** 页签下，单击要添加的任务名称，即可添加该任务进入拓扑图。
- 在 **拓扑列表** 页签下，单击要添加作为子拓扑的任务名称，即可添加该拓扑作为子流程。

iii. 单击目标节点，然后单击旁边的⚙图标。

iv. 根据添加的节点配置事件，然后单击 **确定**。

事件详细介绍，请参见 拓扑执行事件。

- 对于条件网关，您还需要添加边条件。
- 对于分片网关，您还需要添加分片规则。

v. 单击 **提交**。

3. 在任务拓扑列表，将目标任务状态改为 **启用**。

6.3. 触发拓扑任务

拓扑任务配置完成后，您可以通过自动触发、手动触发、BETA 触发三种方式触发任务。

自动触发

- CRON 表达式触发：拓扑任务的调度类型为 CRON 表达式触发时，会根据 CRON 表达式设置的时间周期自动触发任务。
- 事件触发：拓扑任务的调度类型为事件触发时，您可以通过外部事件进行触发。例如 API 调用、拓扑图中的父节点驱动。

> **警告**
>
> 拓扑任务的子节点必须是通过事件触发的任务。因此，只有通过事件触发的拓扑任务可以作为子任务节点被嵌套添加进另一个拓扑任务中。

手动触发

您可以通过手动的方式使任务立即开始执行。处在 **禁用** 状态下的任务也可以被手动触发。

1. 登录任务调度控制台。

2. 在左侧导航栏选择 **中间件 > 任务调度 > 任务编排**。
3. 在任务列表中，单击目标任务右侧的 **立即触发**，然后单击 **确定**。

BETA 触发

BETA 触发仅适用于有 BETA 执行事件的拓扑任务。在 BETA 触发模式下，添加了 BETA 事件的拓扑节点在该节点执行完成后自动中止，等待人工确认。后置节点只有在人工确认完成且满足其它执行条件时才会开始执行。

> **说明**
>
> 有关如何添加 BETA 事件的说明，详见 拓扑执行事件。

1. 在任务列表中，单击目标任务右侧的 **BETA 触发**。
2. 单击 **确定**。

6.4. 拓扑任务网关

6.4.1. 并行网关

在拓扑任务中，您可以添加并行网关实现多个节点并行处理效果，满足业务存在多个独立任务需要并行处理的需求，以此提高整体处理性能。本文将快速引导您如何在拓扑任务中使用并行网关。

操作步骤

1. 创建拓扑子任务。

   创建两个或两个以上的简单或集群任务，以作为后续拓扑任务中的子流程。此处以简单任务为例，配置步骤请参见 创建调度任务。其中：

   - **调度类型**：必须选择 **事件触发**。
   - **通信方式**：必须选择 **回调 CALLBACK**。

2. 使用并行网关。

您可以在创建或编辑拓扑任务时，为任务添加并行网关。这里以新建拓扑任务为例，配置步骤请参见 新建拓扑任务。其中，编辑拓扑图步骤的操作如下：

i. 添加并行网关。

在拓扑图中单击 ⊕ 图标，添加一对 **并行网关**，然后单击 **确定**。

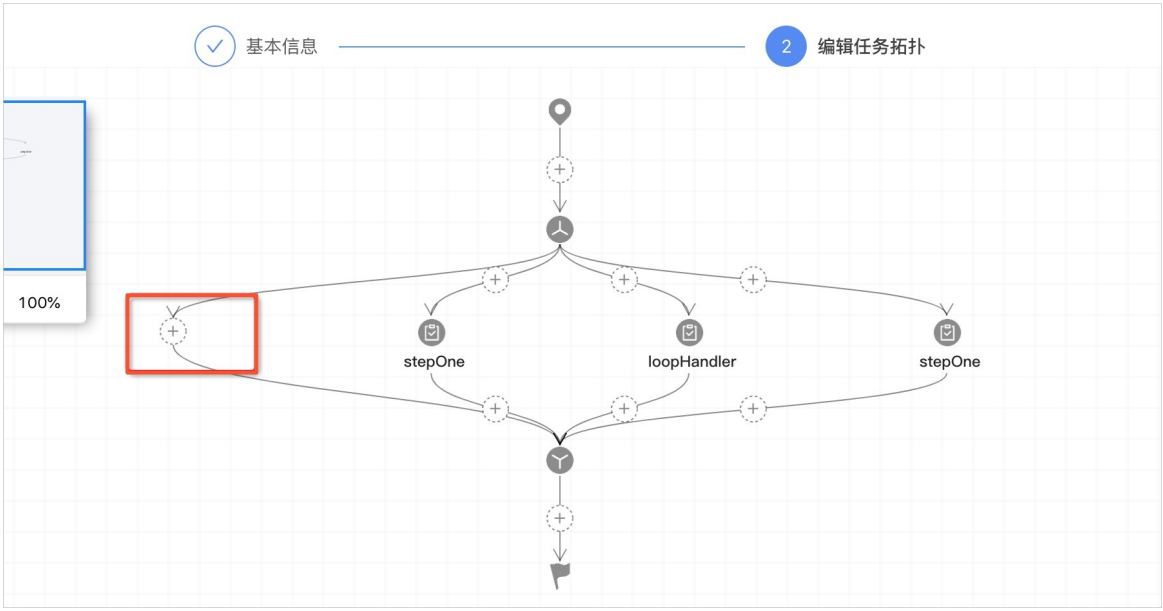
ii. 添加一个子任务。

单击并行网关之间的 ⊕ 图标，然后在 **添加节点** 面板中选择步骤 1 创建的一个任务，单击 **确定**，将其添加为子任务。

iii. 添加其他子任务。

添加完一个子任务后，会出现两个分支。单击另外一个分支上的 图标，继续在 **添加节点** 面板中选择步骤 1 创建的另一个任务。此时，这两个子任务为并行处理关系。

> **说明**
>
> 并行网关支持添加多个并行分支，同一个分支上可以添加多个子任务节点。



iv. 单击 **提交**。

v. 在 **版本管理** 页签，发布该任务版本。

vi. 单击右上角的 **立即触发**，触发该任务。

您可在 **调度记录** 中查看该任务的执行实例。

6.4.2. 条件网关

在拓扑任务中，您可以添加条件网关来设置执行逻辑。例如，当指定条件满足时执行任务 A，条件不满足则执行任务 B。这些条件还支持通过代码动态修改、调整，实现更加灵活的任务编排模式。本文将快速引导您如何在拓扑任务中使用条件网关。

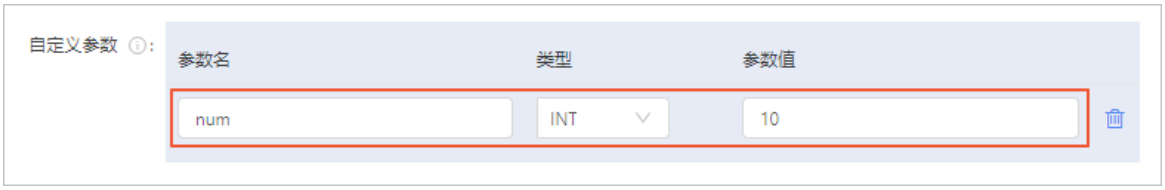
操作步骤

1. 创建拓扑子任务。

   创建两个或两个以上的简单或集群任务，并为其添加自定义参数（即条件参数），以作为后续拓扑任务中的子流程。此处以简单任务为例，配置步骤请参见 创建调度任务。其中：

   - **调度类型**：必须选择 **事件触发**。
   - **通信方式**：必须选择 **回调 CALLBACK**。

- **自定义参数**：必须配置，作为拓扑任务的条件规则。示例如下：



如未在控制台任务配置中自定义该参数，也可在本地代码中设置，示例如下：

```
@Override
public ClientCommonResult handle(JobExecuteContext context) throws Exception{

    context.putCustomParams("num",10);
    return ClientCommonResult.buildSuccessResult();
}
```

说明

以上示例为简单任务代码处理器中的自定义参数配置，配置的参数名为 `num` 。您也可以根据业务需求自主配置。

2. 使用条件网关。

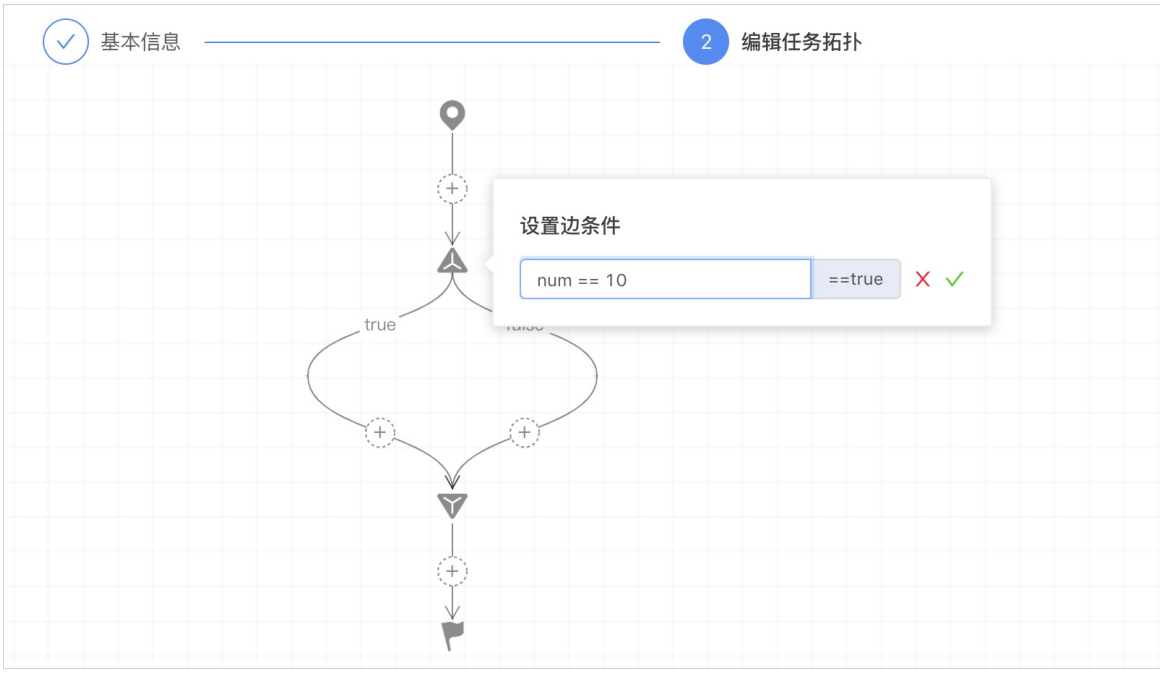
   您可以在创建或编辑拓扑任务时，为任务添加条件网关。这里以新建拓扑任务为例，配置步骤请参见 新建拓扑任务。其中，编辑拓扑图步骤的操作如下：

   i. 添加条件网关。

   在拓扑图中单击 ⊕ 图标，添加一对 **条件网关**，然后单击 **确定**。

ii. 设置边条件。

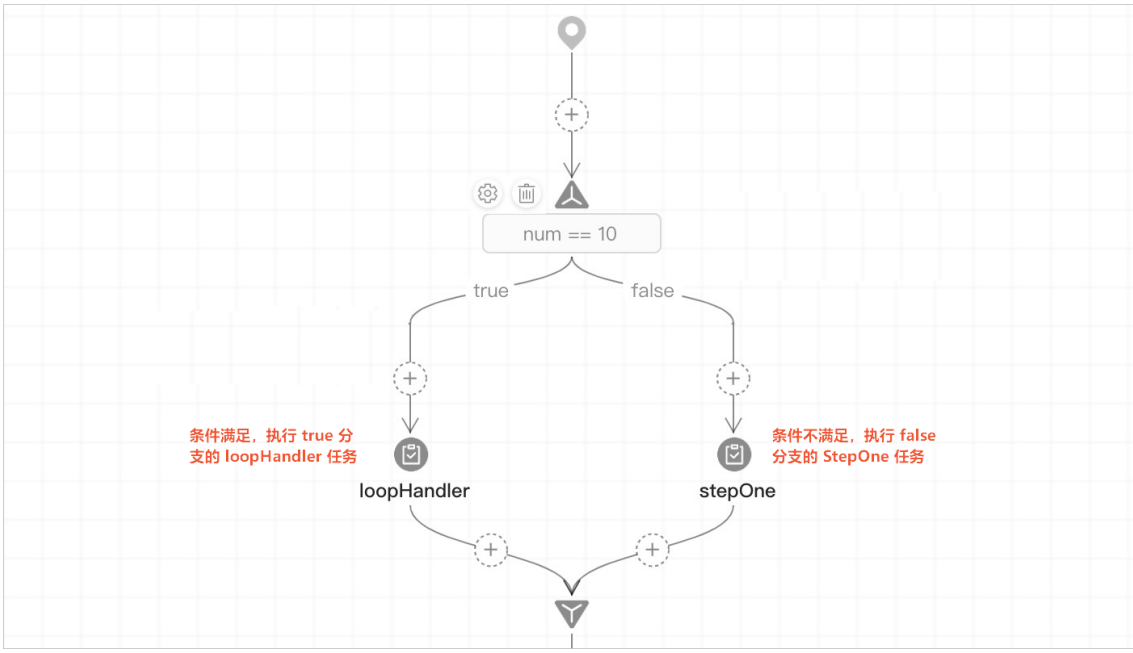
单击条件开始网关，在 **设置边条件** 对话框输入步骤 1 创建的自定义参数名，如上述示例中的 `num==10` ，然后单击 ✓ 保存条件。



iii. 添加子任务节点。

在条件网关之间的两个分支，分别添加步骤 1 创建的简单或集群任务作为子任务节点。

- 如果条件满足，则会执行 true 分支上的子任务。
- 如果条件不满足，则会执行 false 分支上的子任务。



iv. 单击 **提交**。

v. 在 **版本管理** 页签，发布该任务版本。

vi. 单击右上角的 **立即触发**，触发该任务。

您可在 **调度记录** 中查看该任务的执行实例。

6.4.3. 分片网关

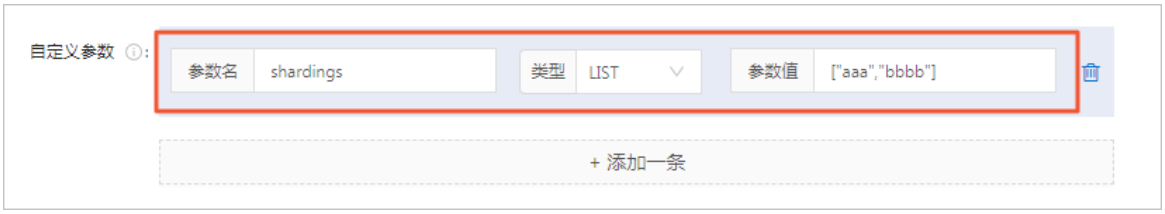
在拓扑任务中，您可以添加分片网关并设定相应的分片维度。如此，拓扑任务在执行过程中，即可根据执行上下文获取的分片配置，动态生成需要执行的分片。本文将快速引导您如何在拓扑任务中使用分片网关。

操作步骤

1. 创建拓扑子任务。

   创建两个或两个以上的简单或集群任务，并为其添加自定义参数（即条件参数），以作为后续拓扑任务中的子流程。此处以简单任务为例，配置步骤请参见 创建调度任务。其中：

   - **调度类型**：必须选择 **事件触发**。
   - **通信方式**：必须选择 **回调 CALLBACK**。
   - **自定义参数**：必须配置，作为拓扑任务的分片规则。示例如下：



   如未在控制台任务配置中自定义该参数，也可在本地代码中设置，示例如下：

```
@Override
public ClientCommonResult handle(JobExecuteContext context) throws Exception{
      context.putCustomParams("shardings",Lists.newArrayList("aaa","bbbb"));
      // 分片规则还支持配置整型，如 context.putCustomParams("shardings", 4);
      return ClientCommonResult.buildSuccessResult();
}
```

   > **说明**
   >
   > 以上示例为简单任务代码处理器中的自定义参数配置，配置的参数名为 `shardings` 。您也可以根据业务需求自主配置。

2. 使用分片网关。

   您可以在创建或编辑拓扑任务时，为任务添加分片网关。这里以新建拓扑任务为例，配置步骤请参见 新建拓扑任务。其中，编辑拓扑图步骤的操作如下：

   i. 添加分片网关。

   在拓扑图中单击 ⊕ 图标，添加一对 **分片网关**，然后单击 **确定**。

   ii. 添加一个子任务。

   单击分片网关之间的 ⊕ 图标，然后在 **添加节点** 面板中选择步骤 1 创建的一个任务，单击 **确定**，将其添加为子任务。

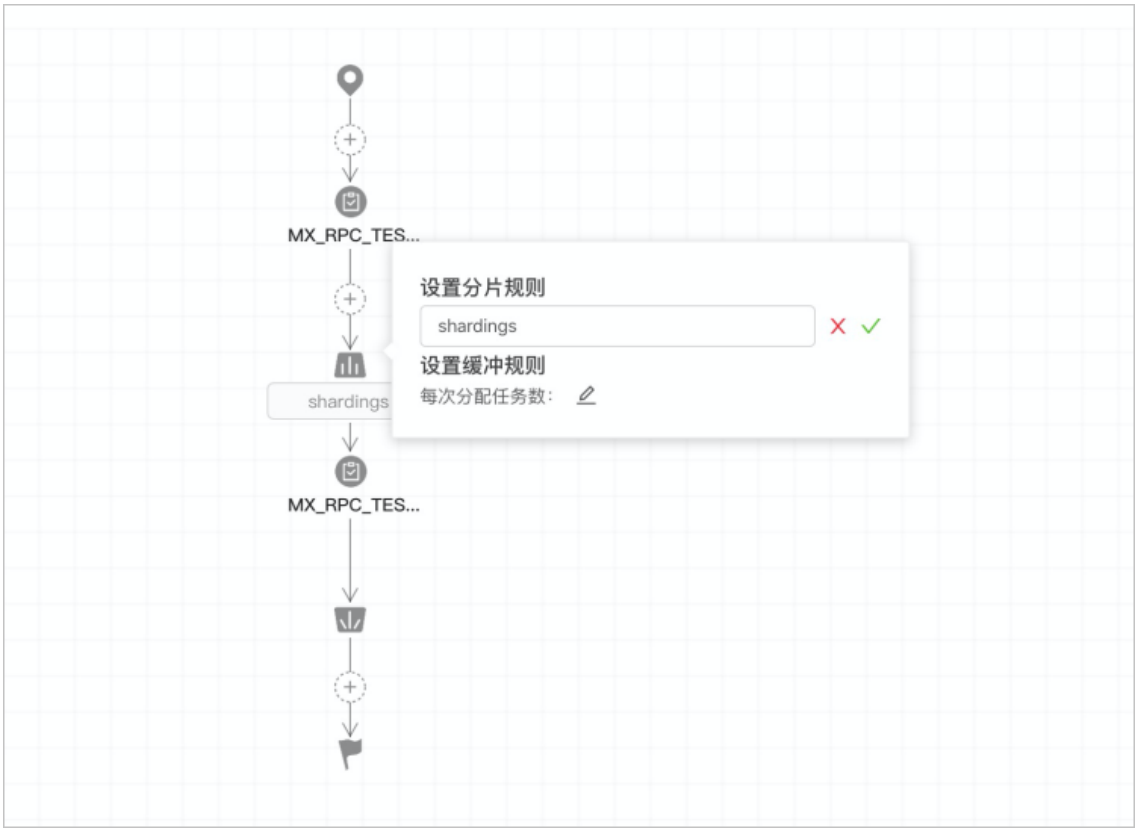
iii. 设置分片网关规则。

单击分片开始网关，然后设置以下参数：

- **设置分片规则**：输入简单或集群任务创建的自定义参数名，例如上述示例中的 `shardings` ，然后单击 ✓ 保存分片规则。
- **设置缓冲规则**（可选）：设置每次并发分片的任务数量，然后单击 ✓ 保存缓冲规则。

  配置缓冲规则可避免分片一次性下发给客户端处理，有效减少客户端缓存和处理压力。同时，缓冲队列支持动态伸缩，您可以在任务执行时新增客户端以分担未处理的分片任务。

> **说明**
>
> 任务执行过程中，若分片网关的开始节点处于“执行中”状态时，您也可以动态修改分片执行的并发数。如开始节点仍处于未执行状态，则无法设置。



iv. 单击 **提交**。

v. 在 **版本管理** 页签，发布该任务版本。

vi. 单击右上角的 **立即触发**，触发该任务。

您可在 **调度记录** 中查看该任务的执行实例。

查看分片节点状态

您可以在拓扑调度记录中查看各个分片的执行状态。

1. 在拓扑任务列表，单击目标任务名称。

2. 单击 **调度记录** 页签。

3. 单击分片数右侧 ⋮ 按钮查看分片状态。

   例如，在下图的拓扑任务图中，分片节点有 5 个分片。其中，第 1、5 分片执行成功，第 2、4 分片执行失败，而第 3 分片执行了跳过操作，处于跳过状态。

   对于失败状态的分片节点，您可以进行实时的人工干预，选择重试或跳过全部失败任务。

6.4.4. 使用跨 zone 网关

在任务拓扑中，您可以通过使用跨 zone 网关，将多个 zone 的节点任务添加到一个拓扑编排中，实现任务跨 zone 编排。

> **说明**
>
> 跨 zone 网关仅适用于开启了共享 LDC 功能的用户。

操作步骤

1. 创建拓扑子任务。

   创建两个或两个以上的简单或集群任务，以作为后续任务拓扑中的子流程。此处以简单任务为例，配置步骤请参见 创建调度任务。其中，**跨 zone 调度** 必须开启。

2. 使用跨 zone 网关。

   您可以在创建或编辑拓扑任务时，为任务添加跨 zone 网关。这里以新建拓扑任务为例，配置步骤请参见 新建拓扑任务。其中，编辑拓扑图步骤的操作如下：

i. 添加跨 zone 网关。

在拓扑图中单击 图标，添加一对跨 zone 网关，然后单击 **确定**。

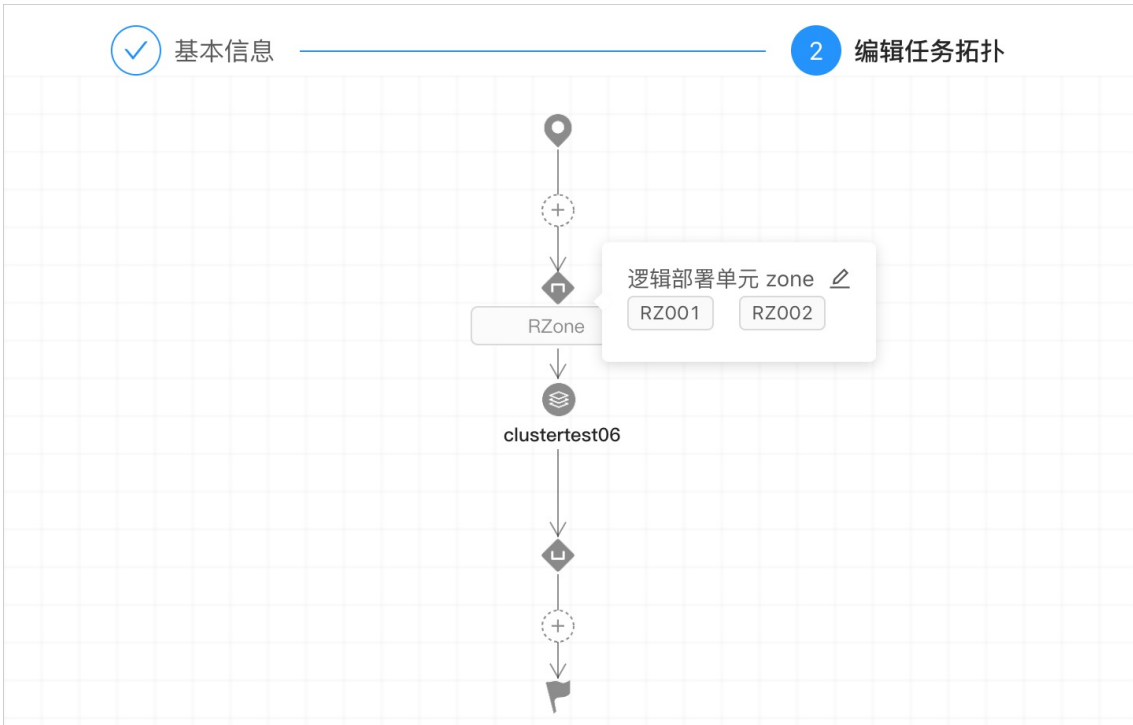
ii. 单击 zone 开始网关，在弹出窗口中，配置网关内需要执行任务的 zone，单击 **确定**。

- 如果在全 zone 执行，只需选择逻辑单元 zone 即可。例如，想在 Rzone 所有 zone 执行，只需在 **配置逻辑单元 zone 名称** 栏选择 Rzone 即可。
- 如果想要指定更细粒度的 zone，可在 **配置部署单元 zone 名称** 栏选择具体的 zone，例如选择 RZ001 和 RZ002，如下图所示：

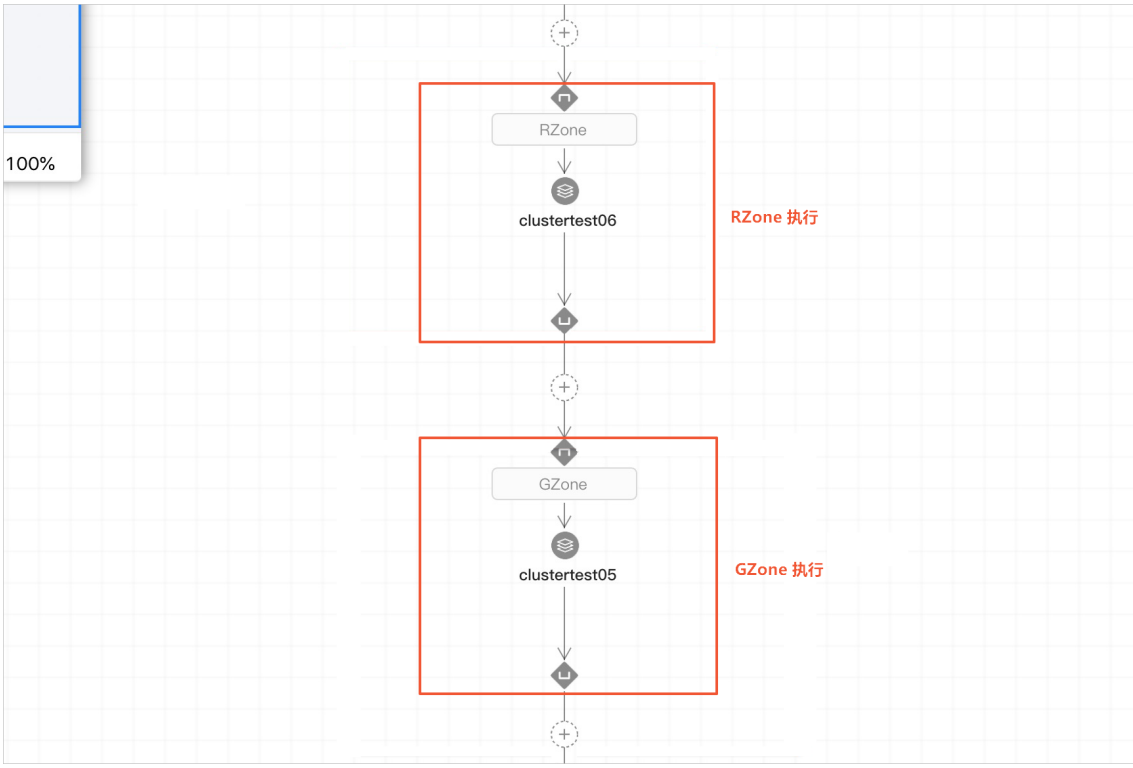
iii. 单击跨 zone 网关之间的 + 图标，在右侧窗口中，添加目标任务节点。

- 跨 zone 网关内只能添加一个任务节点或一个子拓扑节点。如果任务编排流程比较复杂，需要编排多个节点，建议使用拓扑子节点来编辑一个复杂逻辑流程，添加至跨 zone 网关内，然后通过跨 zone 网关指定在哪个 zone 执行。

- 任务节点必须在跨 zone 网关内执行，通过跨 zone 网关将任务指定在特定 zone 执行。如下图所示，clustertest06 会在 RZ001 和 RZ002 执行。



- 支持在一个跨 zone 拓扑任务中，添加多个跨 zone 网关，实现将不同的 zone 任务编排在同一个任务里执行。如下图所示，clustertest06 会在 Rzone 执行，而 clustertest05 则会在 Gzone 执行。



iv. 单击 **提交**，完成任务创建。

v. 在 **版本管理** 页签，发布该任务版本。

vi. 单击右上角的 **立即触发**，触发该任务。

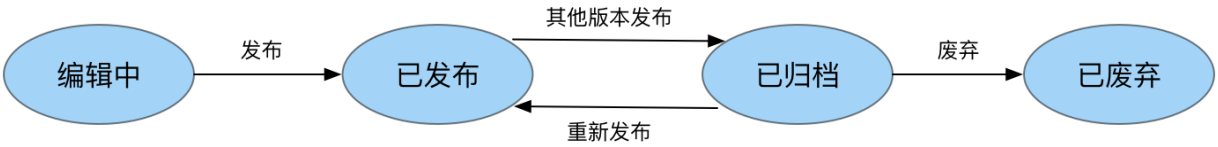
您可在 **调度记录** 中查看该任务的执行实例。

6.5. 拓扑任务版本

您可以通过编辑拓扑任务的版本修改拓扑任务。

拓扑版本生命周期

生命周期示意图：



状态	说明	可用操作
已发布	当前正在使用的拓扑版本。 处于“编辑中”或“已归档”状态的版本被发布后，会变更为“已发布”状态。只能有一个版本能处于“已发布”状态。	• 查看
编辑中	新建或正在编辑中的拓扑版本草稿。 同一基准版本最多只能有一个编辑中的版本草稿。	• 查看 • 编辑 • 发布 • 删除
已归档	新版本发布后，旧的版本自动被归档。 已归档的版本仍可被再次发布，也可以作为草稿的基准版本被使用。	• 查看 • 发布 • 废弃
已废弃	不再使用的版本。 已废弃的版本不可再作为新版本的基准版本。	• 查看 • 删除

创建拓扑版本

1. 登录任务调度控制台。
2. 在左侧导航栏选择 **中间件** > **任务调度** > **任务编排**。
3. 在任务列表中单击目标任务名称，然后单击 **版本管理** 页签。
4. 单击 **添加版本**，然后选择新版本的基准版本。

> **重要**
>
> 您只能选择已发布或已归档的版本作为基准版本，同一版本只允许被一个编辑中的版本选为基准版本。

5. 单击 **确定**。

编辑拓扑版本

您只能编辑处于“编辑中”状态的版本，若要编辑“已发布”版本，请以目标版本为基准创建一个新版本。

1. 在版本列表中，单击目标版本右侧的 **编辑**。
2. 根据您的需求编辑拓扑图。

   操作步骤请参见 操作步骤。
3. 单击 **提交**。

发布拓扑版本

新创建的拓扑任务默认将拓扑初始版本置为“编辑中”，您必须发布该版本才能开始使用拓扑任务。

1. 在版本列表中，单击目标版本右侧的 **发布**。

   您只能发布处于“编辑中”或“已归档”状态的版本。若当前有处于“已发布”状态的版本，原有发布版本会在新版本发布后自动变更为“已归档”状态。
2. 单击 **确定**。

   您可以在 **任务拓扑** 页签查看已发布的拓扑任务。

废弃拓扑版本

发布新版本后，上一个发布的版本会变更为“已归档”状态，您可以将不再使用的“已归档”版本做废弃处理。废弃后的版本不可被重新发布，也不可作为新版本的基准版本。

1. 在版本列表中，单击目标版本右侧的 **废弃**。
2. 单击 **确定**。

删除拓扑版本

您可以将处于“已废弃”状态的版本从版本列表中删除。版本删除后不可恢复，请谨慎操作。

1. 在版本列表中，单击目标版本右侧的 **删除**。
2. 单击 **确定**。

6.6. 拓扑执行事件

在编辑拓扑任务时，您可以为任务节点添加执行事件，以支持更多的业务场景。

您可以在编辑拓扑图时，单击目标节点，然后单击旁边的图标添加执行事件。

> **重要**
>
> 对于网关节点，仅支持 **等待事件**。

BETA 事件

仅在 **BETA 触发** 模式下有效。当任务节点添加 BETA 事件后，拓扑在该节点执行完成后自动中止，等待人工确认。后置节点只有在人工确认完成且满足其它执行条件时才会开始执行。

在拓扑执行因 BETA 事件而中止时，该拓扑的执行状态仍为“执行中”。

循环事件

添加循环事件的节点在开始执行后进入自循环模式。任务在每次执行完成后会更新自定义参数值，并判断自定义参数值是否满足跳出循环条件。当满足跳出条件时循环结束，拓扑任务继续向下执行。否则本节点继续循环执行直到条件满足。

循环事件要求设置如下参数：

- **跳出循环条件**：设置自定义参数的条件判断表达式。
- **循环间隔时间**：设置循环事件执行的间隔时间。单位为时或分。
- **最大循环次数**：设置循环事件的执行次数，超过设置次数后，停止执行该任务。取值范围：1~20。默认为 20 次。

等待事件

添加等待事件的任务节点会自动延迟至特定时间节点才开始执行。

等待事件支持以下四种等待模式：

- **直接等待**：任务流程到达此等待事件时，直接进入等待状态。
- **定时等待**：延迟任务至当天或第二天的特定时间执行。时间格式为 `HH:mm`，24 小时制。
- **间隔等待**：将任务延迟指定间隔时间后执行。单位为时或分。
- **条件等待**：根据上下文传入的参数判断是否要等待。

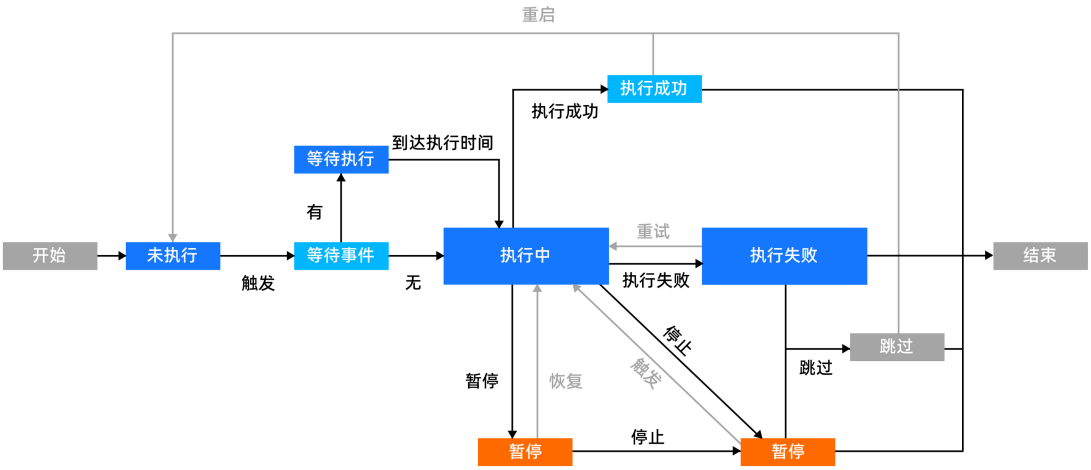
处于等待状态的任务节点，您可以在调度记录中，单击节点旁的跳过按钮 跳过该任务节点，继续执行后续任务。

6.7. 任务节点人工干预

在拓扑任务中，您可以通过查看单个任务节点的执行状态以快速发现和定位拓扑中的执行问题。您还可以通过对拓扑进行人工干预以控制拓扑任务的执行。

任务节点状态

任务节点的生命周期如下：



在拓扑任务中，单个任务节点可能处于以下状态之一：

- **未执行**：该节点未实例化。
- **执行中**：该节点正在执行中。
- **执行成功**：节点执行成功。在无其它人工干预操作的情况下，后续节点将被自动触发。
- **执行失败**：节点执行失败，所有后续节点将不被触发。整个拓扑任务执行失败。
- **暂停**：执行中的节点在被人工暂停后处于暂停状态。暂停中的任务被恢复执行后将从任务断点继续执行。
- **已停止**：节点被强制终止。
- **已跳过**：某个节点执行失败或被强制终止后可被标记为跳过，以继续执行拓扑中的后续节点。
- **等待执行**：某个节点设置了等待事件时，前置节点执行完成后，该节点会进入等待执行状态，到达延时时间后转变为执行中。

人工干预操作

您可以通过人工操作对执行中的拓扑任务进行实时干预，以改变当前的执行逻辑。例如，您可以通过对异常节点进行暂停操作，以查看问题并从断点恢复执行，或通过加锁、解锁操作对任务节点强制进行人工确认。

操作类型	操作对象	说明	适用任务状态
暂停	节点	适用于任务执行过程中发现任务有异常，进行优雅的暂停，暂停后客户端不再处理数据，队列中数据不会被清空，如果此时客户端宕机，服务端会修改 chunk 状态为“未知”状态，可以被其它客户端拉取再次处理。问题修复后需要继续执行，对应的操作为恢复。 重要 • 拓扑子流程必须是集群任务或简单任务。 • 简单任务和集群任务执行到最后一个 handler 时不允许暂停，因此只有一个 handler 的简单任务和集群不允许暂停操作。	执行中
恢复	节点	继续执行暂停中的任务，队列中的数据会被继续处理。	暂停
取消	节点	终止客户端继续执行并清空队列信息，取消后本次调度达到终结状态。取消后继续执行，只能通过重新触发任务来再次执行。	• 执行中 • 暂停 • 等待执行
调整限速	节点	在任务执行过程中根据实际负载情况动态调整限速。	• 执行中 • 暂停 • 等待执行
停止	节点	强制终止任务，不需要再执行。	• 执行中 • 暂停 • 等待执行
重试	节点	在失败的调度记录中重新执行该任务节点。	• 执行失败
跳过	节点	跳过当前节点并驱动后续节点的执行。如果节点处于加锁状态，跳过以后必须人工解锁才会触发后续节点执行。	• 执行失败 • 停止

操作类型	操作对象	说明	适用任务状态
重置	节点	将当前任务节点重置到未执行状态，删除其后置已经实例化的节点。	• 执行成功 • 跳过
重新触发	节点	重新触发该任务节点，生成新的调度记录。仅适用于简单任务或集群任务节点。	• 停止 • 未执行（重置后）
加锁	节点	加锁后的任务节点在执行完成后自动中止后续任务并等待人工确认。经人工确认并解锁后，该拓扑再继续执行后续操作。	• 未执行 • 执行中 • 暂停 • 执行失败 • 停止
解锁	节点	解除锁定。如果当前节点处于执行成功或跳过状态，则解锁后自动触发后续节点。	锁定中
全量暂停	拓扑	暂停当前拓扑任务中的所有执行中的集群任务节点和简单任务（包含多个步骤并且没有执行到最后一步）节点。	执行中
全量取消	拓扑	强制停止当前拓扑任务中的所有执行中或暂停中的节点。	• 执行中 • 暂停
全量恢复	拓扑	恢复处于暂停状态的拓扑任务，恢复执行暂停状态的节点，驱动已执行成功节点的后置节点。	暂停

说明

任务子流程支持暂停、取消和恢复操作，在子流程进行人工干预还会影响到父拓扑任务的执行。

7.任务管理

7.1. 管理简单任务和集群任务

任务列表提供当前环境中所有任务的信息总览。

查看任务

查看任务列表

1. 登录任务调度控制台。
2. 在左侧导航栏选择 **中间件 > 任务调度 > 任务配置**。

   您可以在任务列表中查看已创建的调度任务，包括任务的名称、所属应用、任务优先级等信息。

查看订阅节点

1. 在任务列表中，单击目标任务名称。
2. 单击 **客户端** 页签。

   您可以查看已订阅的客户端 IP、版本和执行记录。单击 **立即触发** 可触发任务。

查看调度记录

当前仅支持查看回调 CALLBACK 任务的调度记录、集群任务拆分时各索引块的执行情况以及客户端日志。

1. 在任务列表中，单击目标任务名称。
2. 单击 **调度记录** 页签。

   您可以查看如下内容：

   - 请求 ID：用于故障排查。

- 触发状态：包括触发成功、触发失败、漏触发。
- 触发原因：包括手动触发、定时触发、事件触发、拓扑触发。
- 开始时间：任务执行的开始时间。
- 结束时间：任务执行的结束时间。
- 耗时：请求执行时长，单位为秒。
- 客户端：订阅节点的 IP 地址。
- 执行状态：包括执行成功、执行失败等。
- 上下文：可查看触发上下文及执行上下文。

任务分组

您可以对业务逻辑对任务进行分组管理。

1. 在任务调度控制台，选择 **中间件 > 任务调度 > 任务分组**。
2. 单击 **添加任务分组**，然后配置 **分组名称** 和 **描述**。
3. 单击目标分组右侧的 **批量添加任务**。
4. 选中目标任务后，单击 **批量添加选中**。

导出和导入任务

您可以在控制台上以 JSON 文件的形式对任务进行批量导入及导出。解决线下到线上的配置同步问题和不同环境之间的任务配置同步问题。

导出任务

1. 在任务调度控制台，选择 **中间件 > 任务调度 > 任务配置**。
2. 在任务列表选中目标任务，然后单击 **导出**。

   导出的 JSON 文件存放在浏览器设置的默认下载文件夹中。

导入任务

1. 在任务调度控制台，选择 **中间件 > 任务调度 > 任务配置**。
2. 单击 **导入**，然后单击 **浏览**。
3. 选中目标文件，然后单击 **打开**。
4. 单击 **确定**。

示例文件

- 简单任务示例

```
{
    "file":[
             {
                "app":"schedulertutorial",
                "context":{},
                "cron":"0 0/5 * * * ?",
                "failHandleStrategy":"THREE_TIMES",
                "handlers":[
                "stepOne",
                "stepTwo"],
                "invokeType":"CALLBACK",
                "name":"stepTest",
                "routeStrategy":"RANDOM",
                "shardingCount":1,
                "timeUnit":"MINUTES",
                "timeout":1,
                "triggerType":"CRON",
                "type":"SIMPLE"
             },
            {
            "app":"watchmen",
            "context":{
                "customParam":{
                "k1":"test1",
                "k2":"test2",
                "k3":"test3"}
            },
            "cron":"0 0 0 * * ? *",
            "des":"callback分片任务",
            "failHandleStrategy":"NO_RETRY",
            "handlers":["CALLBACK_SHARDS_JOB"],
            "invokeType":"CALLBACK",
            "name":"CALLBACK_SHARDS_JOB",
            "routeStrategy":"RANDOM",
            "shardingCount":3,
            "timeUnit":"SECONDS",
            "timeout":10,
            "triggerType":"CRON",
            "type":"SIMPLE"
        },
        {
            "app":"watchmen",
            "context":{
                "customParam":{
                "k1":"v1",
                "k2":"v2"}
             },
            "cron":"0 0 0 * * ? *",
            "des":"既分片又分步任务",
            "failHandleStrategy":"THREE_TIMES",
            "handlers":[
                "WATCHMEN_CALLBACK_RETRY_THREE_TIMES_JOB",
```

```
                "CALLBACK_SHARDS_JOB"
            ],
            "invokeType":"CALLBACK",
            "name":"CALLBACK_SHARDS_AND_STEPS_JOB",
            "routeStrategy":"RANDOM",
            "shardingCount":2,
            "timeUnit":"SECONDS",
            "timeout":10,
            "triggerType":"CRON",
            "type":"SIMPLE"
        },
    ],
    "triggerMode":"RPC"
}
```

- **集群任务示例**

```
{
    "file":[
        {
            "app":"schedulertutorial",
            "context":{},
            "des":"",
            "failHandleStrategy":"NO_RETRY",
            "handlers":[
                "ANTSCHEDULER_CLUSTER_FIRST_SPLITOR",
                "ANTSCHEDULER_CLUSTER_SECOND_SPLITOR",
                "ANTSCHEDULER_CLUSTER_EXECUTOR"
            ],
            "invokeType":"CALLBACK",
            "name":"testCluster",
            "routeStrategy":"RANDOM",
            "shardingCount":1,
            "timeUnit":"MINUTES",
            "timeout":5,
            "triggerType":"EVENT",
            "type":"CLUSTER"
        }
    ],
    "triggerMode":"RPC"
}
```

调整部署单元状态

您可以在 **任务配置** 页面查看每个任务部署的单元总数以及启用的单元数，并根据需求调整各部署单元的启用状态。

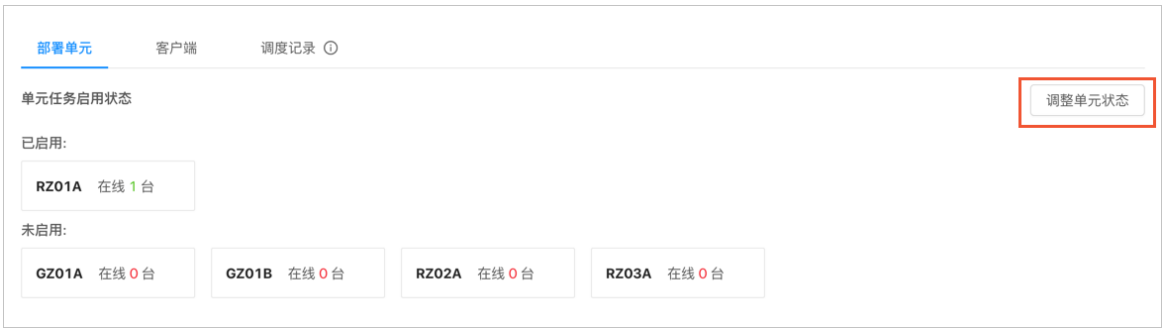
> **说明**
>
> 该功能仅适用于支持 LDC 单元化架构的环境。

1. 在任务调度控制台，选择 **中间件 > 任务调度 > 任务配置**。
2. 打开部署单元设置页面。

您可以通过以下任一方式打开部署单元设置页面：

- 在 **任务配置** 页面的任务列表中，单击目标任务 **部署单元** 列的铅笔图标。

- 在目标任务的详情页，单击 **部署单元** 页签下的 **调整单元状态**。



3. 在 **设置部署单元开关状态** 对话框，选择您需要启用的部署单元后，单击 **确定**。

7.2. 管理拓扑任务

查看任务列表

1. 登录任务调度控制台。
2. 在左侧导航栏选择 **中间件 > 任务调度 > 任务编排**。

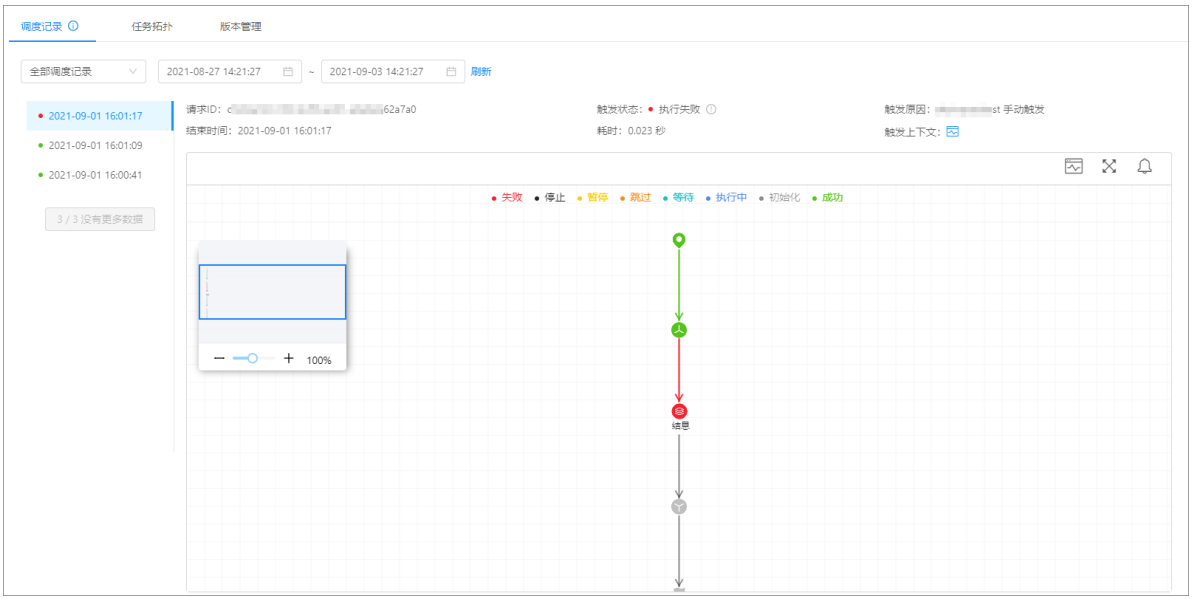
   您可以在任务列表中查看已创建的拓扑任务，包括拓扑名称、调度类型、创建时间等信息。

查看调度记录

1. 在拓扑任务列表，单击目标任务名称。
2. 单击 **调度记录** 页签。

   您可以查看以下内容：

   - 请求 ID：用于故障排查。
   - 触发状态：包括执行成功、执行失败等。
   - 触发原因：包括手动触发、定时触发、事件触发、拓扑触发。
   - 结束时间：任务执行的结束时间。
   - 耗时：请求执行时长，单位为秒。
   - 客户端：订阅节点的 IP 地址。
   - 上下文：可查看触发上下文及执行上下文。
   - 拓扑图：查看任务执行的状态，包括失败、停止、暂停、跳过、等待、执行中、初始化、成功。



查看任务拓扑

1. 在拓扑任务列表，单击目标任务名称。
2. 单击 **任务拓扑** 页签。

   您可以查看当前任务拓扑图。

查看拓扑版本

1. 在拓扑任务列表，单击目标任务名称。
2. 单击 **版本管理** 页签。

   您可以查看当前已有版本，并根据需要进行版本管理。详情请参见 拓扑任务版本。

导出和导入拓扑任务

您可以在控制台上以 JSON 文件的形式对任务进行导入及导出。解决线下到线上的配置同步问题和不同环境之间的任务配置同步问题。

导出任务

在拓扑任务列表，单击目标任务右侧的 **导出**。导出的 JSON 文件存放在浏览器设置的默认下载文件夹中。

导入任务

1. 单击拓扑任务列表右上角的 **导入**，然后单击 **浏览**。
2. 选中目标文件，然后单击 **打开**。
3. 单击 **确定**。

7.3. 任务分类

1. 登录任务调度控制台。
2. 在左侧导航栏选择 **任务分类** 页面，单击 **添加任务分类**。
3. 在弹出的创建任务分组页面填写分组名称和描述。
   - 分组名称：支持英文、数字、下划线。
   - 描述：最多 1024 个字符。

7.4. 操作审计

您可以通过操作审计页面查询用户的操作内容。

操作步骤

1. 登录任务调度控制台。

2. 在左侧导航栏选择 **中间件 > 任务调度 > 操作审计**。

操作审计页面记录的内容包括：

- 任务操作：任务的新增、修改、删除、触发、启用、禁用、导入、暂停执行、恢复执行等。
- 任务拓扑：
  - 针对任务拓扑的新增、修改、触发等常规任务操作。
  - 节点触发、节点重置、节点加锁等人工干预操作。
- 任务分组：分组的新增、修改、删除等。

一条操作审计记录包括：

- **时间**：操作时间。
- **状态**：成功或失败。
- **操作人**：操作用户 ID。
- **对象类型**：包括任务操作、任务拓扑、任务分组。
- **操作类型**：任务的触发、分组、节点人工干预等。
- **操作对象信息**：任务名称、所属应用、拓扑节点信息等。
- **操作**：查看任务基本信息、失败原因等。

> **说明**
>
> 操作审计保留 14 天。如有特殊需求，请 提交工单。

7.5. 任务运行监控

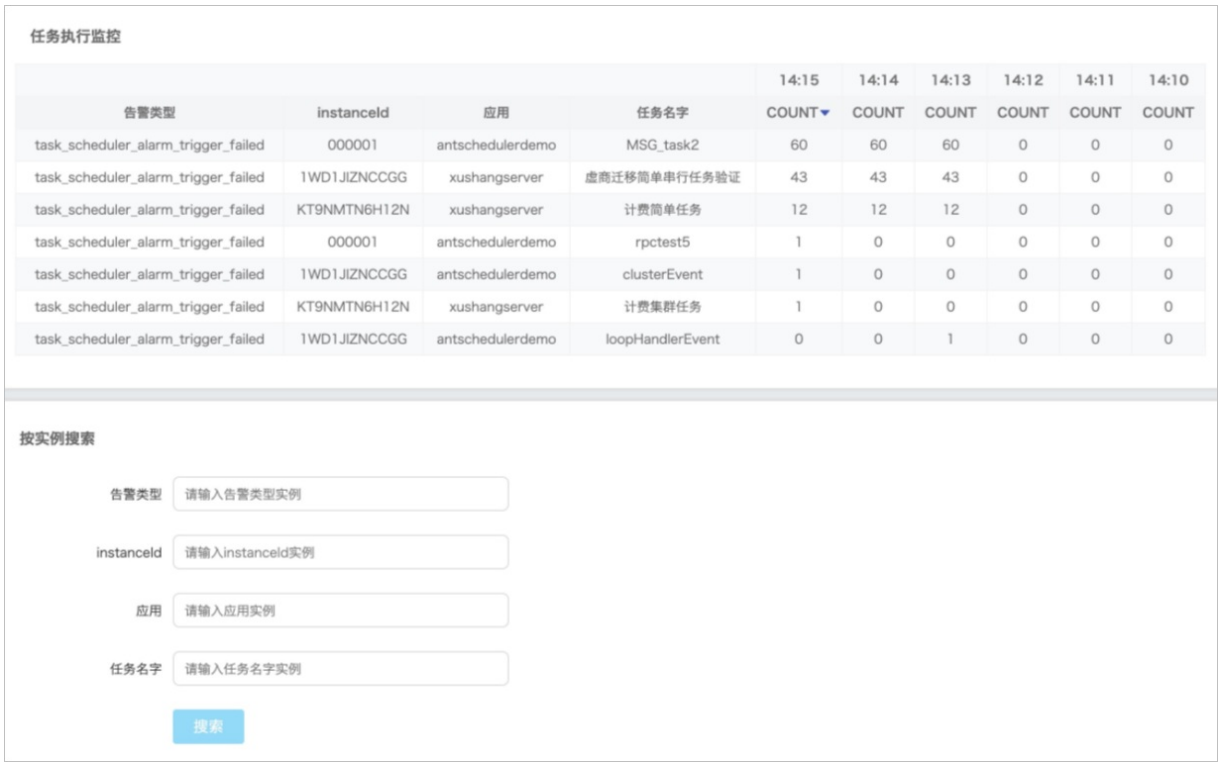
任务调度监控应用系统的运行情况，对告警通知提供日志数据支撑。

常用场景

- 任务触发失败需要告警通知
- 任务执行失败需要告警通知
- 任务执行超时需要告警通知

功能

任务调度系统打印任务执行异常日志 `job-metrics.log` ，业务如果自己有监控平台只要采集该日志，并配置报警即可。业务系统如果使用 RMS 采集日志并配置报警，调度系统不做报警工作，只做数据生产者。



监控配置指引

任务监控日志说明

`job-metrics.log` 日志示例如下：

```
2021-05-24 13:41:24.219 [pool-20-thread-1] INFO  job.metrics - [metric] task_scheduler_alarm_trigger_failed,000001,antschedulerdemo,cw_cluster_test,CLUSTER,TRIGGER_FAILED,SOFA_TS_1506
```

日志格式为： `监控错误码,instanceId,应用名称,任务名称,任务类型,状态,错误码` 。

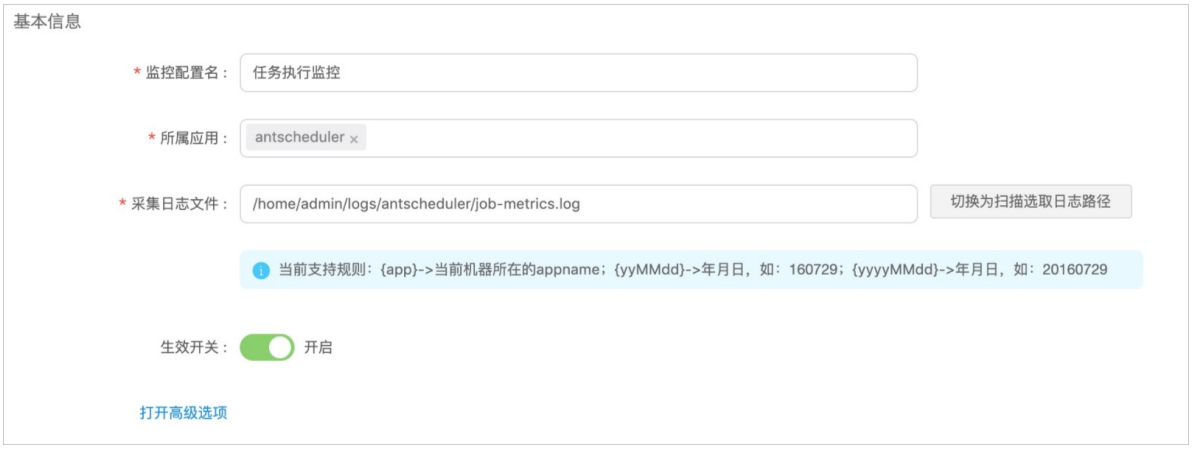
日志字段说明如下：

字段名	字段示例	字段说明

字段名	字段示例	字段说明
监控错误码	task_scheduler_alarm_trigger_failed	取值如下： • task_scheduler_alarm_trigger_failed：触发失败 • task_scheduler_alarm_timeout_execute：任务执行超时 • task_scheduler_exception_execute：任务执行异常 • task_scheduler_error_address_not_exist：找不到客户端地址 • task_scheduler_notify_manual：需要人工操作 • task_scheduler_alarm_blocked：节点阻塞 • task_scheduler_alarm_trigger_omission：漏触发 • task_scheduler_alarm_execute_callback_failed：任务回调失败，需要检查客户端情况。
instanceId	000001	工作空间对应的实例 id。
应用名称	antschedulerdemo	应用名称
任务名称	cw_cluster_test	任务名称
任务类型	CLUSTER	取值如下： • CLUSTER：集群任务 • SIMPLE：简单任务 • TOPOLOGY：拓扑任务 • MSG：消息任务
状态	TRIGGER_FAILED	状态
错误码	SOFA_TS_1506	表示任务执行错误的原因，这里仅展示错误码。

监控配置 RMS

1. 登录监控配置 RMS 控制台，然后单击左侧导航栏 **自定义监控**。
2. 单击 **新建 > 新建监控产品** 新建自定义监控。
3. 填写基本信息。

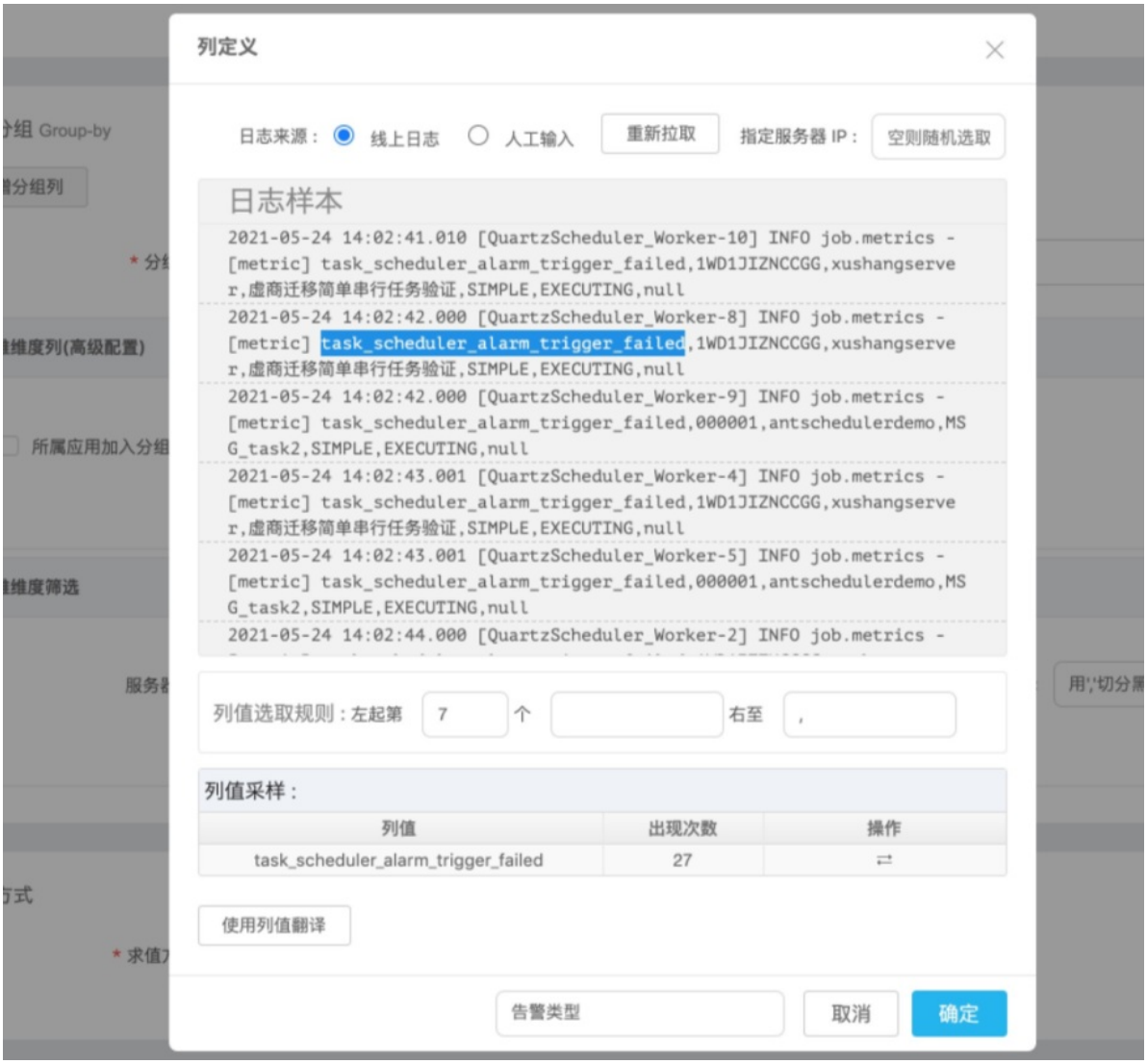


日志文件： `/home/admin/logs/antscheduler/job-metrics.log` 。

4. 增加分组列，添加告警类型。

   增加分组列，添加告警类型，详情请参见 配置自定义业务监控>步骤4：设置列值分组。

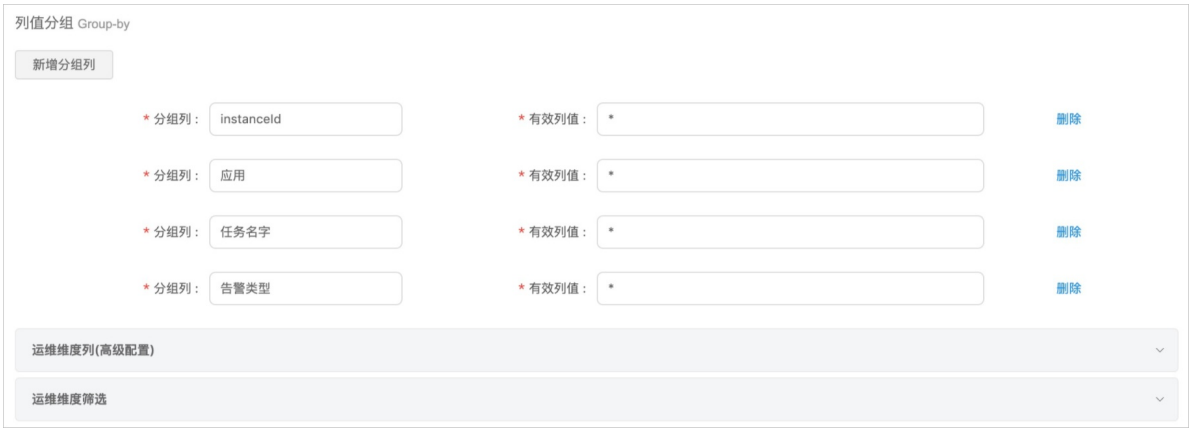
   定义一个告警类型列，如下图所示：



如果没有数据，可以人工输入如下内容：

```
2021-05-24 13:41:24.219 [pool-20-thread-1] INFO  job.metrics - [metric] task_scheduler_alarm_trigger_failed,000001,antschedulerdemo,cw_cluster_test,CLUSTER,TRIGGER_FAILED,SOFA_TS_1506
```

按照以上配置方式，增加其他分组，获取更加详细的信息，如下所示：



5. 配置告警。

   配置告警详情请参见 配置自定义业务监控>步骤6：配置报警。

   图中配置表示当前值大于 0 则告警。

6. 订阅告警。

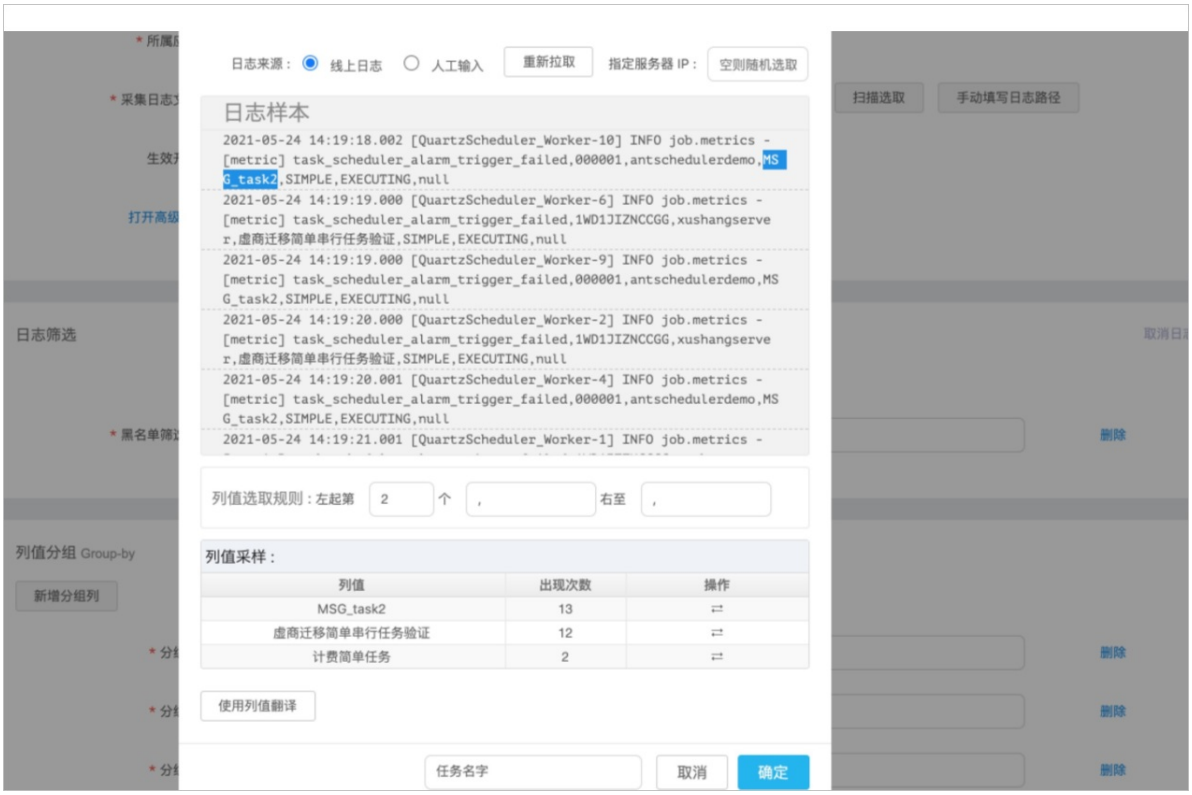
   配置告警订阅，这样任务触发、执行失败就会有告警。详情请参见 订阅告警。

7. 告警过滤。

   有的任务或应用本身有问题，为了减少噪音，可以通过配置黑名单将这些应用或任务直接过滤。

   操作详情请参见 创建自定义监控>操作步骤6。

   根据任务名称进行过滤示例如下：



配置需要过滤的任务名字：

8.常见问题

8.1. 产品相关问题

任务调度的核心功能是什么？

- 分布式调度：提高系统的可用性、伸缩性，且调度与业务的分离，让研发更专注业务逻辑的开发。
- 多种任务类型：简单任务、集群任务、消息任务、流式任务，满足不同场景需求。
- 丰富的管理功能：包括定时触发、事件触发、手动触发、优雅停止、断点续跑、配置导入导出、OpenAPI 等。
- 丰富的任务配置项：提供丰富的任务配置项，包括多种路由策略、多种失败重试模式、多种漏触发处理策略，并支持在线配置调度任务入参。
- 可视化任务编排：简化多任务调度的运维管理，并支持暂停/恢复、跳过、加锁/解锁、重置、失败重试等多种人工预操作。
- 任务监控：支持实时查看任务的执行状态及执行进度，执行时长，执行的客户端 IP、执行是否成功、执行失败原因等信息。

更多信息，请参见 任务调度概述。

任务调度适用于什么场景？

解决大规模分布式任务统一调度的问题，助力企业能够更加迅速地满足业务需求。典型的应用场景包括金融行业结息处理、日切处理、余额缓冲记账、政务数据批处理等。

任务调度支持哪些任务类型？

任务调度支持的基本任务类型包括：简单任务、集群任务、消息任务。此外，任务拓扑编排功能支持将多个任务按执行的顺序编排成一张拓扑图，并自动执行，简化多任务调度的运维管理。

更多信息，请参见 功能特性。

任务调度客户端对于开发框架有何要求？

任务调度客户端对于开发框架没有要求，支持直接在您的 Java 工程中引入相应 Maven 依赖包进行开发。

为了简化您的开发流程，全方位体验金融科技平台的各项功能，推荐使用 SOFA 金融级分布式中间件套件进行应用开发。SOFA 中间件是蚂蚁科技自主研发的金融级分布式中间件，包含了构建金融级云原生架构所需的各个组件，是一套分布式架构的完整的解决方案，也是在金融场景里锤炼出来的最佳实践。其中，SOFABoot 是基于 Spring Boot 的研发框架。更多信息，请参见 SOFABoot 快速入门。

任务调度是如何计费的？

任务调度公测已结束。目前，该组件的计费维度为 **调度次数**，按月累计调度次数阶梯计费。计费单位为 **元/万次**。

详细的计费信息，请参见 产品定价。

8.2. 功能相关问题

任务互斥的作用是什么，是否会造成任务堆叠？

互斥的任务，即处在“执行中”或“暂停”状态的任务，即使已经达到下一个任务触发时间点，仍无法再次触发。本次调度状态以“跳过”标记。因此，并不会造成任务堆叠。

任务分组的作用是什么？

根据不同系统进行划分定时任务的仅此而已，与其他开源的任务调度的“分组”概念不同。可以理解成不同的系统有不同的任务调度。

如何使用简单任务的任务分步？

一个简单任务可以拆分成多个执行步骤（step），由不同的执行器（handler）按顺序执行。前置步骤成功完成后，才开始执行下一个步骤。

详细的操作步骤，参见 新建任务。

如何使用简单任务的分片功能？

简单任务的分片功能适用于数据量巨大的场景。通过使用分片，可以让多台客户端服务器来处理业务数据。

详细的操作步骤，参见 创建调度任务。

如何监控任务触发结果或任务执行结果？

- 对于任务触发状态的监控，您可以通过业务实时监控平台配置自定义监控指标与告警即可。
- 对于任务执行结果的监控，您需要在业务代码中进行相应的日志打印，继而通过监控平台配置对应的日志监控告警。

8.3. 任务触发与执行问题

为什么任务导出只支持全量操作？

目前仅 MS 2.28.4 版本支持选择任务方式导出，只支持全量导入，其余版本均只支持全量导出和导入。

任务调度触发失败

检查 `home/admin/logs/scheduler` 目录下的 `common-error.log` 错误日志，如果原因是线程被占满，可以扩大线程池的数量。

同一台 ESC 机器不能部署多个 AntScheduler 客户端

应避免同一台 ESC 机器部署多个 AntScheduler 客户端，否则会出现报错“target zone is null”。

任务执行超时

问题现象

出现 TIMEOUT 错误。

问题原因

客户端处理超时。

解决方案

- 查看超时时间是否配置太小。

  建议根据任务的实际执行情况设置合理的超时时间。
- 查看客户端错误日志 `common-error.log` 。

自定义参数不存在

问题现象

出现”javax.el.PropertyNotFoundException: Cannot find property \${property_name}“报错。

问题原因

任务拓扑的判断条件中，指定的自定义参数不存在。

解决方案

检查判断条件与任务拓扑配置中的自定义参数列表是否匹配，并更正。

客户端未注册到服务端

问题现象

出现如下报错：

- no targets was found
- target zone is null
- no available target zones was found, please check if clients are available
- no available targets was found in zone

问题原因

客户端没有注册到服务端。

解决方案

1. 确认客户端是否启动。

   请先确认控制台配置的应用名称和执行器名字是否与客户端代码里写的一致，可以查看 endpoint 配置是否正确。

2. 如果客户端已经启动，则检查 `logs/scheduler` 目录下的 `common-default.log` 和 `common-error.log` 错误日志。
3. 如果错误日志中都没有报错，则查看健康检查日志，确认健康检查是否通过。

客户端未提供执行器

问题现象

出现 no suitable job handler for {handler_name} 报错。

问题原因

任务中使用的 handler 执行器存在，但任务应用中没有找到。

解决方案

1. 检查代码里的 `handlerName` 返回值和任务配置里的执行器名字是否一致。
2. 在客户端加上 handler 实现，`getName` 值设置为当前 `{handler_name}` 。

拓扑无可执行的发布版本

问题现象

出现”no published topology process was found“报错。

问题原因

拓扑图没有发布的版本，就直接执行拓扑任务，即无可执行的发布版本。

解决方案

编辑拓扑图，发布一个版本。

服务端异常

问题现象

- 整个应用的所有任务均触发失败。
- 控制台提示失败原因：Server exception when invoke with callback. Please check the server log!

问题原因

该项目工程缺少 Tracer 依赖：

```
<dependency>
      <groupId>com.alipay.sofa</groupId>
      <artifactId>tracer-enterprise-sofa-boot-starter</artifactId>
</dependency>
```

解决方案

- 检查工程 `pom.xml` 文件，是否已经添加了 Tracer 依赖。如没有，需手动添加该依赖。
- 对于 SOFABoot 框架开发的工程，需要确认 SOFABoot 的版本是否低于 3.4.3。
  - 如低于 3.4.3 版本，需要手动添加 Tracer 依赖。
  - 而 SOFABoot 3.4.3 及其以上版本，则会自动加上 Tracer 依赖。无需再手动添加。
  - 有关 SOFABoot 的版本信息，参见 版本说明。

任务互斥导致执行失败

问题现象

出现如下报错：

- job is exclusive
- can not be triggered, please check the value of timeoutStrategy and isExclusive

问题原因

- 互斥任务正在执行中，不允许再次触发。
- 互斥任务已经超时且未进行重置，则不允许再次触发。

解决方案

查看调度记录：

- 如果任务正在执行中，则等待执行完成后再次触发。
- 如果有 **触发超时** 的执行记录，尝试将该任务的触发记录重置为 **成功** 或 **失败** 状态以后再次触发该任务。

任务处在禁用状态

问题现象

任务配置了重试策略，但是执行失败后没有重试。

问题原因

仅当任务处于“启用”状态时重试策略才会生效，“禁用”时执行失败不会进行重试。

解决方案

1. 登录 **任务调度** 控制台。
2. 单击 **任务配置**，然后在任务列表中启用该任务。

任务触发时间间隔过短

问题现象

出现”the interval for tasks of type callback must takes more than 5 minutes“报错。

问题原因

对于以 Cron 表达式触发的 CALLBACK 类型的任务，两次触发的时间间隔必须大于 5 分钟。

解决方案

修改 Cron 表达式，使得定时任务的触发间隔时间大于 5 分钟，例如 `1 0/5 * * ?` 。有关 Cron 表达式的详细说明，参见 CRON 表达式详解。

客户端任务执行失败

问题现象

控制面板显示执行失败。

问题原因

找不到客户端链接或客户端异常。

解决方案

根据执行失败的错误信息，查看客户端 `logs/scheduler` 目录下的 `common-error.log` 错误日志。

任务调度控制台无法基于 Workspace 隔离

问题原因

未传入参数 `x-instance-id header` 。

解决方案

用户可以将中间件产品集升级到 1.7.7 及以上版本，MS 升级到 2.25.5 及以上版本。

任务调度通过应用名未找到消息任务

问题现象

在任务调度 - 》任务配置 - 》所属应用 里无法找到基于消息的任务的应用，故无法根据应用名查找消息任务。RPC 任务可以通过应用列表筛选，消息任务没有应用筛选的方式。

问题原因

`Antscheduler` 的 `com.alipay.antscheduler.facade.IAntJobFacade` 的 `getAllApps` 限制了查找方式。

解决方案

将 Antscheduler 升级到 2.26.0 及以上版本。

基于消息的定时任务无法定时或者手动触发

问题现象

出现报错“send msg error”。

问题原因

负责触发该定时任务的 AntScheduler 的线程池被打满，拒绝了新的任务触发请求。

解决方案

1. 重启客户端。
2. 登录 **任务调度** 控制台。
3. 单击 **任务配置**，然后在任务列表中重新触发该任务。

任务调度任务创建失败

问题现象

创建任务时，页面显示报错，报错信息：任务名称已存在。

问题原因

可能曾经建立过相同名称的任务，删除回收到到回收站中。

解决方案

建议不要使用曾经使用过的任务名称，即使该任务已被删除，也不要使用。

启动时报错“AntScheduler_Cloud not found”

问题现象

应用启动 `scheduler/common-error.log` 日志出现报错“AntScheduler_Cloud not found”。

解决方案

- 开通任务调度 TS。
- 如有需要，您可以提交 工单申请，申请后台手动配置 ANTVIP 的 Domain。

配置时间过期导致报错

问题现象

AntScheduler 出现报错“Based on configured schedule, the given trigger 'DEFAULT.2002901' will never fire”。

问题原因

Cron 任务的配置时间已过期，任务无法执行。

解决方案

- 在任务调度控制台，将报错任务的 Cron 表达式根据需求配置为正确的，配置方式请参见 CRON 表达式详解。
- 关闭报错任务。

发布任务调度服务时出现报错

问题现象

服务启动时 `stdout.log` 日志出现报错“report client load to server 168.5.24.76:9001 error”。

问题原因

SOFABoot 3.2.2 版本新增 SOFA Scheduler 客户端的 metric 上报功能，可能 Scheduler 服务端版本较低，不支持此功能导致出现报错。

解决方案

将 SOFA Scheduler 服务端升级到 2.36.0 版本。

RMS 分钟级定时消息任务成功率异常

问题现象

- 出现告警信息：分钟级 MSG 消息任务成功率过低。
- `logs/antscheduler/job-msg-error.log` 日志显示“Msgbroker Server rejects the message type XXXX”异常，表明消息被 MSGBroker 拒收。

问题原因

出现报错的消息任务是任务调度系统的测试用例，未配置消息类型和订阅关系，故出现报错。

解决方案

测试用例仅用于部署环境后验证功能，后续需禁用 Watchmen 创建的测试用例所对应的消息任务。

任务调度

问题现象

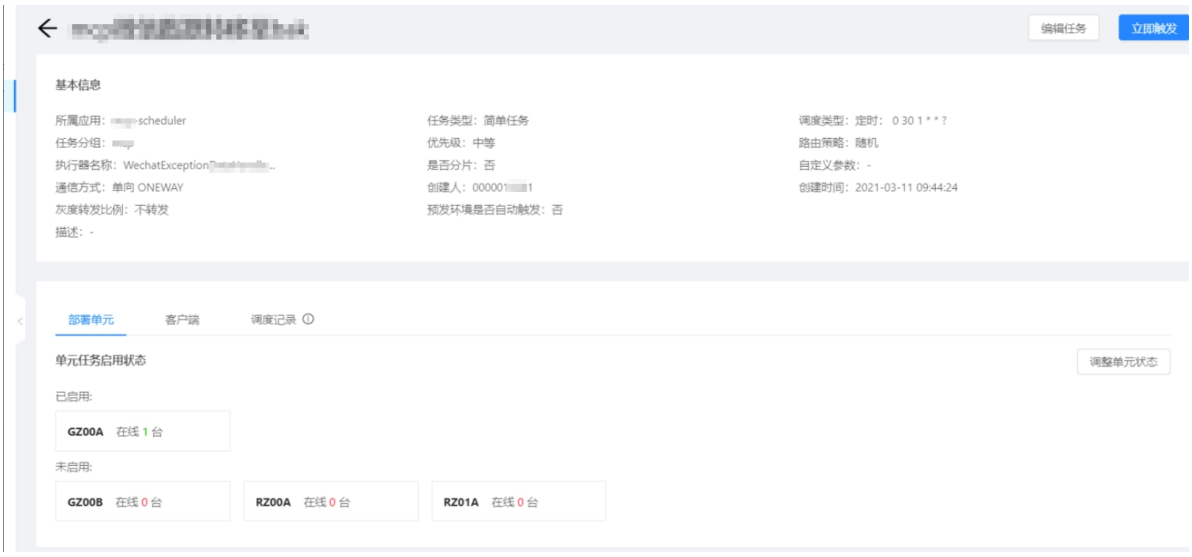
任务配置为每天调度一次，实际每分钟调度一次。

问题原因

任务调度控制台同时配置了两个共用一个执行器的任务，一个配置为每天调度一次，另一个配置为每分钟调度一次。

解决方案

1. 在任务调度控制台，单击 **任务配置**，查看任务的调度类型是否为每天一次。



2. 若任务的调度类型配置为每天执行一次，查看业务日志 `getName()` 、 `scheduler-accept-digest.log` 日志和 AntScheduler 每分钟的调度记录。
3. 若日志显示有每分钟调度一次的调度任务、而调度记录没有，则查看 AntScheduler 日志 `XXX WechatExceptionDataHandler /home/admin/logs/antscheduler/job-rpc.log` ，查找该每分钟调度一次的任务。
4. 找到该任务后，若该任务在控制台不显示，但数据库有该任务，并且 Cron 配置为每分钟一次，则该任务可能是开发环境的测试任务，删除该每分钟调度一次的测试任务即可。

8.4. 任务拓扑相关问题

子节点任务类型错误

问题现象

出现 job is in topology, trigger type can not be CRON and invoke type can not be ONEWAY 报错。

问题原因

拓扑子任务节点不支持 ONEWAY 调用，不支持使用 Cron 表达式触发。

解决方案

将子任务修改为使用事件触发、通信类型为 CALLBACK。

分片网关分片规则错误

问题现象

出现 Sharding rule cannot be parsed 报错。

问题原因

拓扑分片网关中的分片规则不符合要求。分片规则只支持正整数或 `Collection<String>` 类型。

解决方案

修改分片规则，使用正整数或在自定义参数中定义 `Collection<String>` 类型的参数。

节点暂停失败

问题现象

出现 can not suspend job that is executing last handler 报错。

问题原因

执行到最后一个 handler 的任务不允许暂停。

解决方案

等待任务执行完毕，或强制终止任务。

人工干预操作对象错误

问题现象

- **节点触发错误：** can not trigger activity when topology is PAUSED or CANCEL
- **节点恢复错误：** only allowed to resume PAUSED trigger instance
- **节点重试错误：** activity instance could not be retried which status is EXECUTING

- **节点取消执行/停止错误：** only allowed to cancel EXECUTING or PAUSED trigger instance
- **节点重置错误：** can not reset when executing post activity was existed
- **节点加锁错误：** activity instance could not be locked which status is SUCCEED or SKIPPED

问题原因

拓扑中的人工干预操作只适用于特定状态下的任务节点。

解决方案

只在适用状态下使用人工干预操作。详见 任务节点人工干预。

在拓扑任务图中，选择任务节点时，没有任务显示

问题现象

拓扑任务图中，选择任务节点时，没有任务显示。

问题原因

任务拓扑本身可以通过 CRON 表达式或者事件触发，但可以加入拓扑的任务节点必须符合以下要求：

- 触发方式为事件触发
- 通信方式为 CALLBACK
- 任务节点为简单任务、集群任务或其它拓扑任务

解决方案

修改待加入拓扑的任务的触发方式或通信方式。

任务调度添加循环事件

问题现象

拓扑执行时间设置循环次数不生效，跳出循环失败。

问题原因

代码配置错误。

解决方案

如有需要，您可以 提交工单，申请对应拓扑的跳出循环 Demo。

8.5. 客户端不在线问题排查

本文介绍客户端不在线问题的特征、排查思路以及解决思路。

相关问题

问题表现特征如下：

- 客户端显示中没有某个机器的 IP。
- 任务触发时报：No available target zone was found。
- 客户端显示不在线。

排查思路

1. 检查任务配置的应用配置是否有误。

- 检查 InstanceId 和页面对应的 InstanceId 是否一致，如果不一致修改为一致，按页面对应的 InstanceId 进行修改。

  用 Google 浏览器打开开发者工具，单击调度任何一个页面，请求上会带着一个 InstanceId，这个就是你当前页面账号对应的 InstanceId。
- 检查 AK/SK 是否正确。
- 检查 `com.antcloud.antvip.endpoint` 的配置是否正确，是否连错环境。
- 检查 `spring.application.name` 和页面任务配置中的应用名字是否一致。

2. 检查应用进程是否正常。
   - 如果是 SOFABoot 或 Spring Boot 应用，可以使用这个命令 `curl http://127.0.0.1:8080/actuator/readiness` 查看（端口配置应用端口）是否健康，如果不健康，则说明业务进程有问题。
   - 查看 `~/conf/acvip-java-client-cache/domains/xxx-ANTSCHEDULER_CLOUD.json` 是否能拉取下来。
     - 如果配置未拉取下来，则查看 `~/logs/acvip-java-client/inner-error.log` 是否有报错。
     - 如果配置能拉取下来，查看内容是否为空，是否有 IP。
3. 进入任务配置的应用机器（任意一台机器）查看连接情况。

   输入以下命令查看是否连接到任务调度机器。

   ```
   netstat -ant | grep 9001
   ```

   - 如果不存在连接，则请看自身进程问题，回到步骤 1、2 继续检查。
   - 如果存在连接，可以联系研发同学一起排查。

8.6. 其他常见问题

客户端未注册到服务端

问题现象

订阅客户端列表中无数据。

问题原因

客户端未注册到服务端。

解决方案

1. 确认客户端是否启动。

   请先确认控制台配置的应用名称和执行器名字是否与客户端代码里写的一致，可以查看 endpoint 配置是否正确。
2. 如果客户端已经启动，则检查 `logs/scheduler` 目录下的 `common-default.log` 和 `common-error.log` 错误日志。
3. 如果错误日志中都没有报错，则查看健康检查日志，确认健康检查是否通过。

单实例部署多个客户端，导致找不到连接

一个实例（ECS、虚拟机、Pod）不允许部署多个 AntScheduler 客户端，否则会出现连接互踢的问题。

AntVIP 地址配置错误

问题现象

出现 [AntVIPRouter] Get scheduler servers failed 报错。

问题原因

AntVIP 地址配置错误。

解决方案

检查客户端应用配置文件中的 AntVIP 地址。详情请参见 properties 配置项。

任务调度客户端不在线

问题现象

任务调度控制台显示客户端不在线。

问题原因

客户端和服务端之间的 TCP 长连接建立失败。

解决方案

控制台显示客户端在线，就表示客户端和服务端成功建立了 **TCP 长连接**。建立 TCP 连接需要有两个因素：

- 客户端获取到服务端的 **IP 和端口**。
- 客户端与服务端地址之间的 **网络** 是互通的。

客户端会先从 AntVIP 获取服务端的信息（IP、端口），然后与其建立连接。建连成功后，控制台就会显示客户端在线。但如果 antvip、instanceId 这些配置信息错误，则会出现 endpoint might be wrong 错误。

任务删除失败

问题现象

出现 can not delete running job 报错。

问题原因

任务正在运行，无法删除。

解决方案

等待任务运行结束或手动暂停任务，然后删除。

版本编辑冲突问题

问题现象

出现 ${draft_version} is editing，can not create new version with ${base_version} 报错。

问题原因

基于同一个版本 ${base_version} 只能有一个编辑中的版本，如 ${draft_version}，所以不能创建新的版本。

解决方案

通过以下任一方法发布一个新版本：

- 将编辑中的版本 ${draft_version} 发布后，再创建一个基于 ${base_version} 的新版本。
- 创建新版本时选择其它已发布版本作为基准版本。

调度记录的重试问题

问题现象

在任务的调度记录里，有的记录提供了 **重试** 按钮，可以重试，有的则不能。

问题原因

同一个任务，如果有多次执行记录，那么只有最新的执行记录是有重试选项的，因为旧的记录已经没有意义。

基于消息的任务中，业务应用未收到消息任务

排查思路

1. 查看任务调度控制台任务配置的消息事件码是否与消息控制台配置的一致。
2. 若消息时间码一致，查看消息控制台配置的消息任务的 Topic 是否为 **TP_F_SC**，否则修改为 **TP_F_SC**。
3. 若 Topic 配置正确，查看任务调度服务端日志，是否将消息成功发送。
4. 若任务调度已将消息成功发送，请参考消息队列的 快速开始相关问题 继续排查。

AntScheduler 健康检查 503

问题现象

- 在接口 curl 127.0.0.1:8080/health 进行健康检查，显示失败“DOWN”。

```
#curl 127.0.0.1:8080/health
{"status":"DOWN"}
```

- 在另一个检查接口 curl 127.0.0.1:8080/health/readiness 进行健康检查，显示正常“UP"。

问题原因

接口错误。

解决方案

- 将 MS 版本升级到 2.34 及以上，版本提升后可直接将 /health 接口内部转发到 /actuator/readiness 接口。
- 接口 curl 127.0.0.1:8080/health/readiness 也可作为健康检查。