分布式任务调度(TCT)

产品文档

目录

产品简介

概述

分布式任务调度（Cloud Task）是一款高性能、高可靠通用的分布式任务调度中间件，通过指定时间规则严格触发调度任务，保障调度任务的可靠有序执行。该服务支持国际通用的时间表达式、调度任务执行生命周期管理，解决传统定时调度任务单点及并发性能问题。 同时，支持任务分片、流程编排复杂调度任务处理能力，覆盖广泛的任务调度应用场景。

使用场景

- 数据备份
  通过制定数据备份的调度规则，定时触发数据备份操作，便于数据丢失的恢复操作，从而保障数据安全。
- 日志切分
  为避免单个日志文件过大导致数据查询慢，可以基于文件按行索引对文件进行切分，将不同的分片数据分发给多个实例并发执行，提升执行效率。
- 运维监控
  通过广播执行方式在集群中所有工作节点触发定时任务采集监控数据并上报监控平台进行可视化展示，便于进行数据分析。
- 金融日切
  金融日切要求多个系统按照一定顺序严格执行完成日切操作，使用任务编排功能，可轻松完成任务依赖调整，直观地查看任务的执行进度。

功能说明

功能分类	功能	详细说明
部署组管理	部署组生命周期管理	支持界面化创建、编辑和删除部署组。
	标签管理	支持为执行器实例设置标签，并对标签进行管理。标签用于任务调度时指定调度策略。
	执行器实例管理	展示注册到部署组中的执行器示例，展示信息包括执行器实例的 SDK 版本、地址信息、包含哪些内置任务、执行器的状态等。
任务管理	任务生命周期管理	支持多种类型（Java任务、Shell任务、Python任务、外部任务）任务的创建、修改、删除。
	告警通知	支持对任务执行的成功/失败/超时/已终止/漏触发等配置消息通知（通知方式支持邮箱、短信、企微、回调方式）。
	任务参数传递	任务执行传递的参数，可在任务执行中通过上下文参数获取。
	任务导入导出	支持以JSON格式导出和导入任务，导入任务采用异步导入方式，支持查询详细的导入状态。
	多版本管理	对任务参数的修改可以保留历史版本，可以查看任意历史版本详情，也可以将任意历史版本设置为当前版本用于任务的触发。
任务触发方式	定时触发	触发方式为“定时触发”时，可通过 cron 表达式设置任务的执行时间（所设置的 cron 表达式需符合 Quartz 规范）。
	周期触发	通过设置任务触发的间隔时间来设置任务的执行时间。
	指定时间触发	触发方式为“指定时间触发”时，可以为任务指定一个执行时间，到指定时间后触发任务执行。
	手动触发	支持通过 API 接口或手动触发任务执行操作。
	工作流触发	通过工作流的上下游依赖关系触发任务执行。
	触发约束	可以为任务设置最多触发次数，以及触发的起止时间。

功能分类	功能	详细说明
任务执行	随机单节点执行	在所有可用的执行器实例中随机挑选一个合适的实例执行任务。
	广播执行	所有可用的执行器实例均执行任务。
	分片执行	将单一任务按特定逻辑切分为多个独立子任务（分片），将多个独立子任务路由到部署组不同的执行器示例上执行。
		分片参数设置：分片任务下发执行的分片参数，以键值对方式呈现。
	输出结果聚合	支持对任务配置输出聚合规则，用于对任务的输出结果进行聚合，例如对分片任务的某个输出字段计算所有分片的平均值。
	状态停留超时配置	给任务配置在某一个状态长时间停留时的处置措施，例如任务触发后长时间处于下发中（超过配置的时间如 60 秒），可以配置自动将其置为失败状态。状态停留超时也会同步显示在监控指标中。
任务调度	并发控制	支持部署组定义中指定执行器实例的最大并发执行数。
		支持指定子任务的单机并发数。
		任务支持并行执行和串行执行：可配置任务上次触发还未结束的情况下是否允许新的触发执行。
	调度规则	支持执行器标签过滤、标签分组的调度策略。
		支持单元化的调度，基于部署单元（单元对应的标签）信息来调度任务。
工作流编排	工作流生命周期管理	支持工作流的创建、编辑和删除，工作流通过将不同任务节点进行逻辑组合，构建任务的上下游依赖关系。
	工作流依赖编排	支持有向无环图（DAG）方式编排工作流，多个依赖通过逻辑 AND/OR 组合，依赖关系可以指定为成功触发（前置任务执行成功触发后续任务）、失败触发（前置任务执行失败触发后续任务）。工作流中指定核心节点用于复杂工作流中判定整体成功与否，有且只有所有核心任务执行成功工作流判定为执行成功。
	工作流嵌套	支持工作流嵌套编排，工作流可以作为另一个工作流的节点进行编排。
	工作流执行控制	支持给工作流节点额外配置允许触发的时间，例如给工作流节点配置 T 日（工作流开始执行的日期）的 20:00 之后才允许触发执行，那么工作流执行时该节点的前置依赖满足后还需要等待 T 日的 20:00 之后才会开始执行。时间上也支持配置 T+1 日、T+2 日。

功能分类	功能	详细说明
		工作流执行时，支持停止、暂停、继续操作。
		工作流执行时，支持失败节点（也包括超时等状态）的失败重试、失败跳过操作。
		工作流执行时，支持对尚且开始节点进行挂起，挂起后只有手动解除挂起才能执行。
		工作流执行时，支持强制某个还未达到触发条件的节点开始执行。
		工作流执行时，支持强制将一个未结束的节点（如下发中、执行中）标记为成功、失败、超时等终态状态。
	触发计划	支持配置工作流按计划触发工作流，指定每日固定时间自动生成一天的执行计划（预触发记录），然后按照生成的执行计划去触发执行工作流。在生成的执行计划上用户可以做挂起节点等操作，同时用户也可以强制提前开始计划中的某次执行。
		支持对开启按计划触发的工作流手动生成下一次触发记录。
	工作流导入导出	支持以 JSON 格式的工作流导入导出，及导入状态查询。
	告警通知	支持对任务执行的成功/失败/超时/已终止/漏触发等配置消息通知（告警通知支持邮箱、短信、企微、回调方式）。
	版本管理	对工作流参数的修改可以保留历史版本，可以查看任意历史版本详情。
执行记录	批次执行记录	支持查看所有任务的执行历史，并可以手动对任务停止执行或重新执行。
	执行日志	支持查看任务的执行日志，对执行中的任务日志和大日志（>1M）采用 Websocket 方式查看。
	批次执行详情	支持查看批次执行详情。
	工作流执行拓扑	支持查看工作流执行可视化拓扑图，展示工作流执行详情。支持执行中工作流的暂停、继续，工作流节点的挂起、终止、跳过等。
执行看板	看板管理	创建和管理执行看板，将任务和工作流添加进看板中统一查看每天的执行计划以及执行进度。
	近期执行	查看最近几天看板中任务的执行计划和实际执行进度，并提供相应统计信息。

功能分类	功能	详细说明
租户概览	运营概览	展示租户下任务数、工作流数、部署组数、调度次数等概览信息。
	关键监控指标	展示该租户下任务调度 TPS、任务超时次数、任务限流次数等监控指标。
权限管理	基于权限策略的权限	权限分为只读权限（r）、可执行权限（x）、全读写权限（w）。支持全局级别的只读权限、可执行权限和全读写权限，也支持资源（如任务）级别（通过项目实现）的只读权限、可执行权限和全读写权限。通过给用户关联对应的权限策略来赋予用户对应的权限。

优势说明

- 高性能
  支持海量任务调度管理、提供精准的秒级任务调度能力。
- 调度精准可靠
  提供任务调度的可靠消息投递能力，平台组件支持水平扩展、高效容错保障任务调度服务的精准可靠运行。
- 丰富应用场景
  提供随机、广播、分片多种执行方式，提供定时、周期、工作流、手动多种触发方式，满足丰富的任务调度应用场景。
- 简单易操作
  提供交互简洁的可视化操作界面，方便用户快速接入使用、调度统计及故障排错需求。

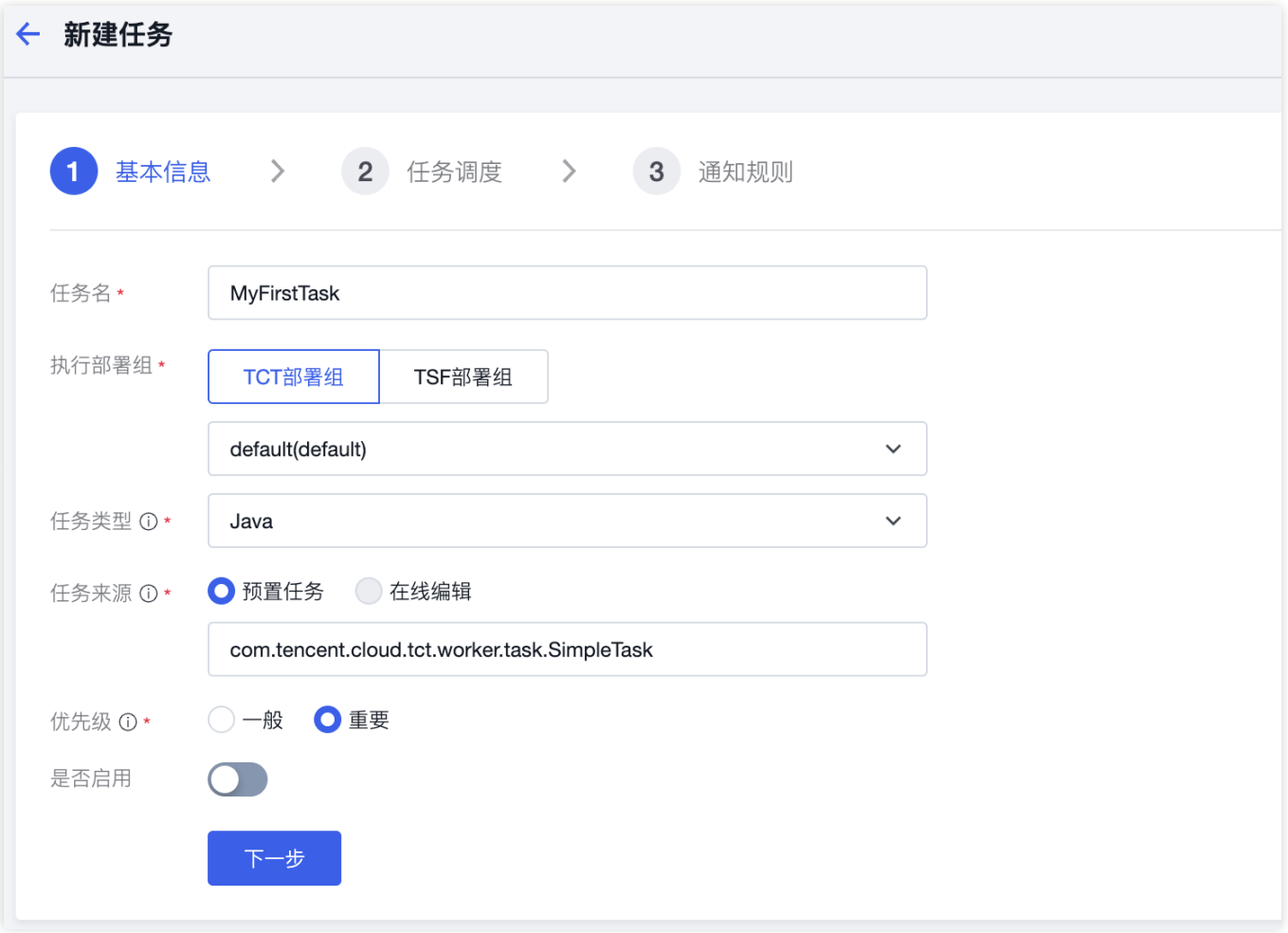
快速入门
第一个TCT任务

创建任务

在任务管理中点击新建任务，并按照以下说明配置任务。

基本信息

- 执行部署组：选择 TCT 自带的 default 部署组，该部署组中默认部署了 3 个执行器实例，并且包含了几个简单的内置任务（涵盖 Java、Shell、Python 任务类型）。
- 任务类型：选择 Java。
- 任务来源：我们选择预制任务中的 SimpleTask 任务。
- 优先级：优先级暂且随意选择
- 是否启用：我们暂时选择不启用。



任务调度

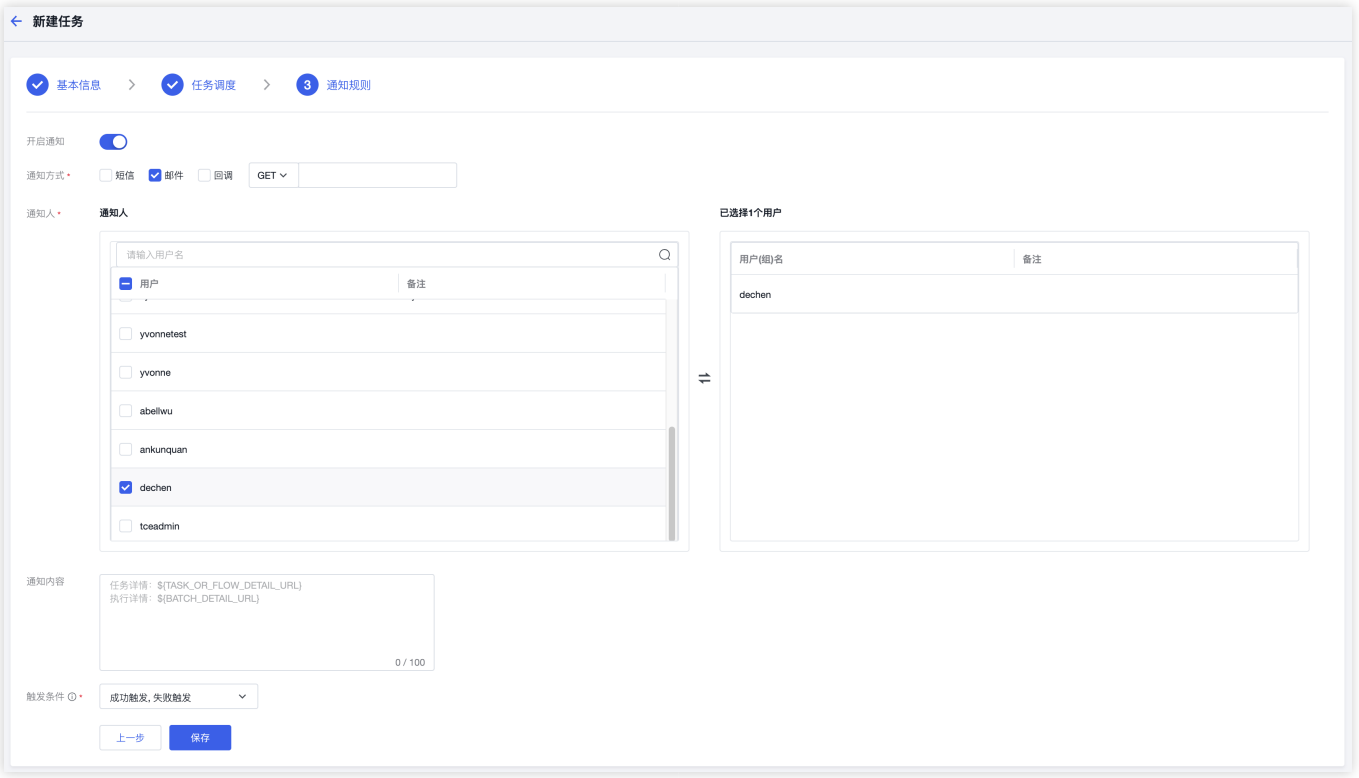
- 触发方式：选择定时触发。
- 触发时间：Cron 表达式 "0/30 * * * * ?" 表示每 30s 触发一次。
- 开始触发时间：暂时不配置。
- 结束触发时间：暂时不配置。
- 最多触发次数：暂时不配置。
- 允许并发：是否允许该任务在上一次执行还未结束的情况下触发新的执行，这里配置成允许。
- 执行器选择：暂时选择默认的所有。
- 执行方式：选择随机单点执行，这样任务将在部署组中的执行器实例中随机选择一个执行。
- 任务参数：传递给执行的任务的参数，我们选择的 SimpleTask 任务支持 --sleep 指定任务 sleep 多久模拟任务执行时间。这里我们指定任务 sleep 5 秒。
- 超时时间：该任务的超时时间，我们配置 60 秒。


基本信息
2
任务调度
3
通知规则
任务触发
触发方式 *
定时触发
触发时间 *
0/30 * * * * ?
校验
开始触发时间
结束触发时间
最多触发次数
允许并发
任务执行
执行器选择 *
所有
执行方式 *
随机单节点执行
在所有实例中随机挑选某一实例执行任务。
任务参数
--sleep=5
9 / 10000
超时时间
60
秒
最大超时时间为24小时
高级设置
上一步
下一步

通知规则

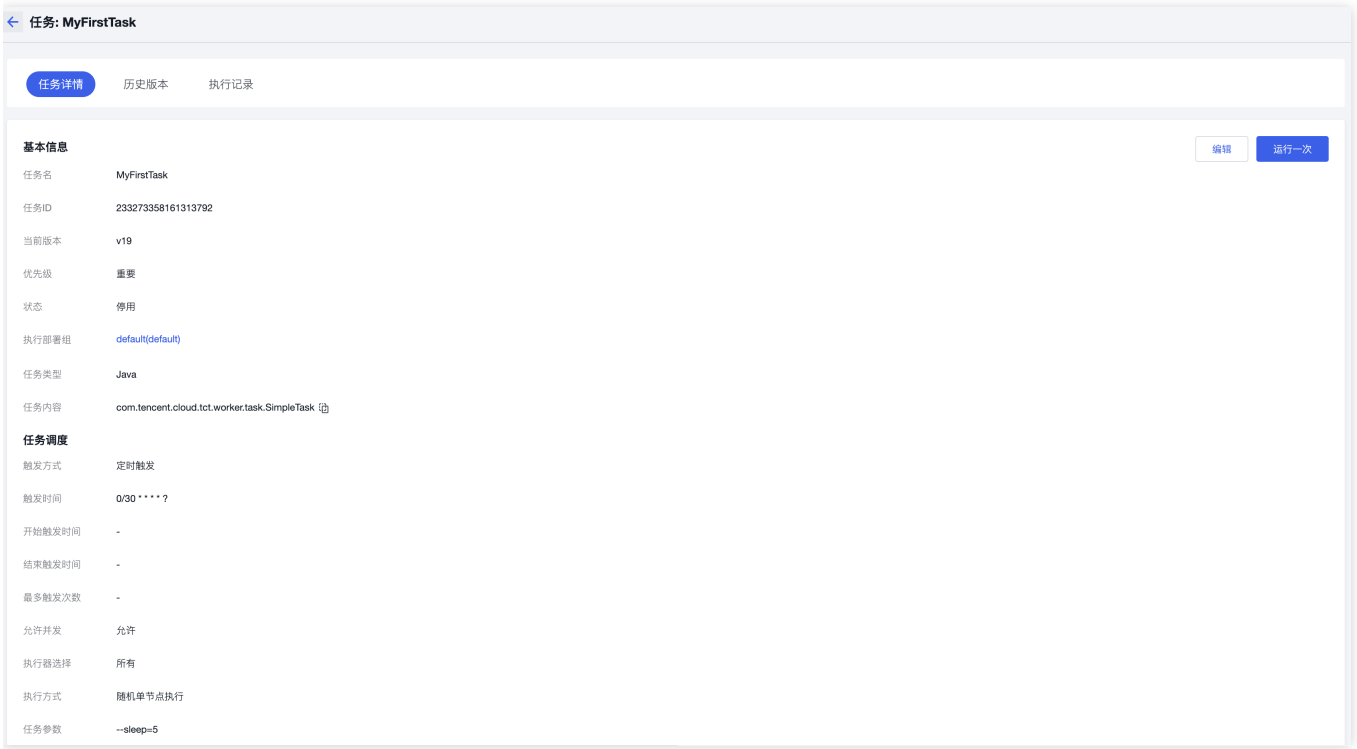
开启通知，并配置通知规则。

- 通知方式：选择邮件方式。
- 通知人：选择需要通知的人，选择自己的账号即可。
- 通知内容：可以不配置，采用默认的即可。
- 触发条件：我们将成功触发、失败触发都选择上。



触发任务

在任务列表中，我们可以找到刚才创建的任务，点击任务 ID 进入任务详情页。



手动触发

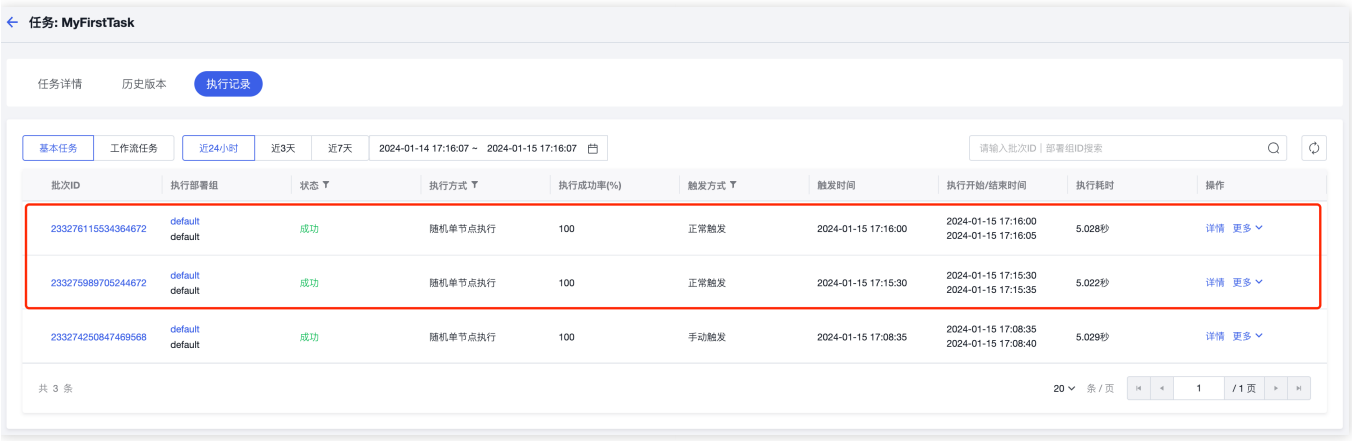
由于我们还未启用该任务，因此任务不会自动触发执行，这里我们先手动触发任务一次，点击详情页右上角【运行一次】按钮触发任务。触发成功后，在【执行记录】TAB 也中查看执行记录，等待任务执行完成后，查看执行时间是否和配置的 5 秒匹配。

点击批次 ID 或者点击详情按钮可以查看任务的详细执行记录，以及查看执行日志。

启用任务

在任务管理里找到刚才创建的任务，点击【启用】按钮启用任务，启用后任务会根据之前配置的触发时间 Cron 表达式触发执行。

成功启用后，我们可以点击进入任务详情中，查看任务执行记录，可以看到新的执行记录符合配置的 Cron 触发时间，在每分钟的 0 秒，30 秒的时候触发一次。

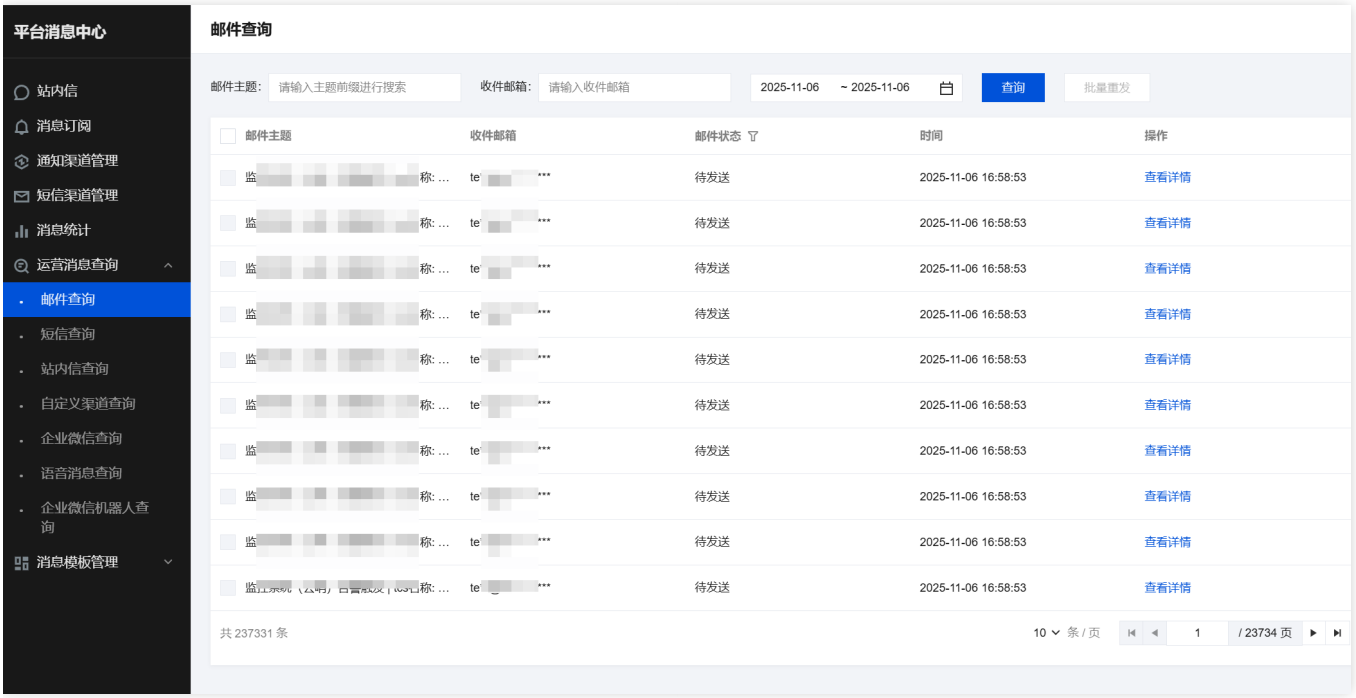


查看执行详情

在执行记录中点击详情，可以打开任务批次的执行详情，在这里可以查看任务的执行日志。

查看通知

TCT 借助的平台的消息通知服务，我们可以在【消息查询】中查看发送的通知消息。


平台消息中心
站内信
消息订阅
通知渠道管理
短信渠道管理
消息统计
运营消息查询
邮件查询
短信查询
站内信查询
自定义渠道查询
企业微信查询
语音消息查询
企业微信机器人查询
消息模板管理
邮件查询
邮件主题:
请输入主题前缀进行搜索
收件邮箱:
请输入收件邮箱
2025-11-06 ~ 2025-11-06
查询
批量重发
邮件主题
收件邮箱
邮件状态
时间
操作
待发送
2025-11-06 16:58:53
查看详情
共 237331 条
10 条 / 页
1
/ 23734 页

开发一个简单的 Java 任务

任务开发

Maven 配置

找到 Maven 所使用的配置文件 settings.xml，一般为 ~/.m2/settings.xml，添加 TCT Maven 地址。

```
<?xml version="1.0" encoding="UTF-8"?>
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 http://maven.apache.org/xsd/set
tings-1.0.0.xsd">

  <pluginGroups></pluginGroups>
  <proxies></proxies>
  <servers></servers>
  <mirrors></mirrors>

  <profiles>
    <profile>
      <id>nexus</id>
      <repositories>
        <repository>
          <id>central</id>
          <url>http://repo1.maven.org/maven2</url>
          <releases>
            <enabled>true</enabled>
          </releases>
          <snapshots>
            <enabled>true</enabled>
          </snapshots>
        </repository>
      </repositories>
      <pluginRepositories>
        <pluginRepository>
          <id>central</id>
          <url>http://repo1.maven.org/maven2</url>
          <releases>
            <enabled>true</enabled>
```

```
            </releases>
            <snapshots>
              <enabled>true</enabled>
            </snapshots>
          </pluginRepository>
        </pluginRepositories>
      </profile>
      <profile>
        <id>tct</id>
        <repositories>
          <repository>
            <id>tct</id>
            <name>tct</name>
            <url>https://mirrors.cloud.tencent.com/nexus/repository/maven-public/</url>
            <releases>
              <enabled>true</enabled>
            </releases>
            <snapshots>
              <enabled>true</enabled>
            </snapshots>
          </repository>
        </repositories>
      </profile>
    </profiles>

    <activeProfiles>
      <activeProfile>nexus</activeProfile>
      <activeProfile>tct</activeProfile>
    </activeProfiles>

  </settings>
```

POM 依赖配置

在 pom.xml 文件中添加 TCT 依赖：

```
  <dependency>
    <groupId>com.tencent.cloud</groupId>
    <artifactId>tct-spring-boot-starter</artifactId>
    <version>2.1.0-rc8</version>
  </dependency>
```

任务编写

通过实现分布式任务调度框架中的 ExecutableTask 及 TerminableTask 接口，编写任务。在 execute 方法中实现任务执行逻辑，在 cancel 方法中实现任务停止逻辑，示例如下：

```
package com.tencent.cloud.tct;

import com.tencent.cloud.task.sdk.client.LogReporter;
import com.tencent.cloud.task.sdk.client.model.ExecutableTaskData;
import com.tencent.cloud.task.sdk.client.model.ProcessResult;
import com.tencent.cloud.task.sdk.client.model.ProcessResultCode;
import com.tencent.cloud.task.sdk.client.model.TerminateResult;
import com.tencent.cloud.task.sdk.client.remoting.TaskExecuteFuture;
import com.tencent.cloud.task.sdk.client.spi.ExecutableTask;
import com.tencent.cloud.task.sdk.client.spi.TerminableTask;
import org.springframework.stereotype.Component;
import org.apache.commons.lang3.RandomUtils;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Component;
import java.lang.invoke.MethodHandles;

/**
 * 模拟简单的任务, 执行时间在10s ~ 15s之间, 可以终止。
 */
 @Component
public class SimpleTask implements ExecutableTask,TerminableTask {
    private static final Logger LOG = LoggerFactory.getLogger(MethodHandles.lookup().lookupClass());
    @Override
    public ProcessResult execute(ExecutableTaskData taskData) {
        ProcessResult result = new ProcessResult();
        try {
            // 上报执行日志
            LogReporter.log(taskData, "Start to execute SimpleLogExecutableTask");
            LogReporter.log(taskData, "Hello, this is a demo for SimpleLogExecutableTask");

            // Sleep 来模拟任务执行时间
            long sleepTime = RandomUtils.nextLong(10000, 15000);
            Thread.sleep(sleepTime);

            result.setResultCode(ProcessResultCode.SUCCESS);
            LogReporter.log(taskData, "Execute SimpleLogExecutableTask succeeded");
        } catch (InterruptedException e) {
            result.setResultCode(ProcessResultCode.TERMINATED);
            LogReporter.log(taskData, "Task is terminated... ");
        } catch (Throwable t) {
```

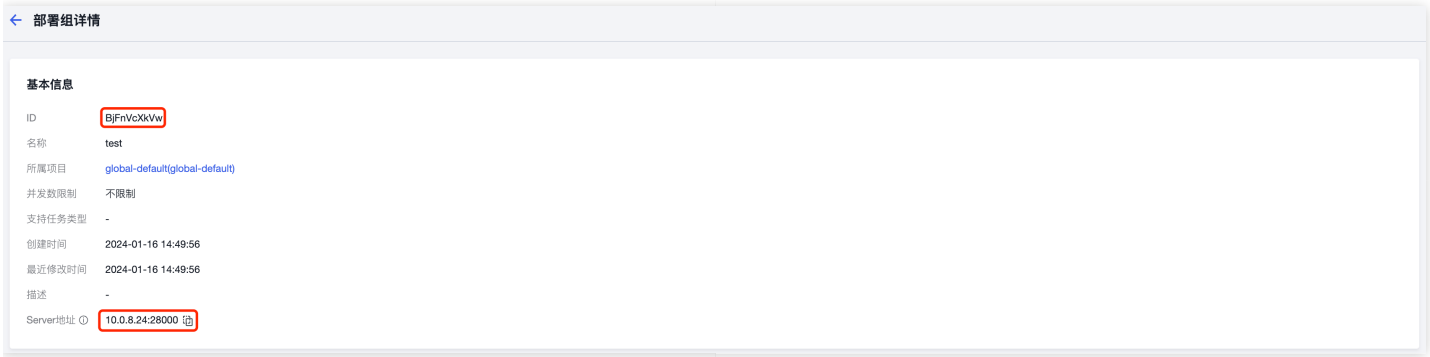
```
            LOG.error(t.getMessage(), t);
            result.setResultCode(ProcessResultCode.FAIL);
        }
        return  result;
    }

    @Override
    public TerminateResult cancel(TaskExecuteFuture future, ExecutableTaskData taskData) {
        LogReporter.log(taskData, "To cancel task");
        future.cancel(true);
        LogReporter.log(taskData,"Task cancel success");
        return TerminateResult.newTerminateSuccessResult();
    }
}
```

注意：单纯通过 Future 的 cancel 方法，并一定能够停止正在运行中的任务，通常需要业务逻辑实现上进行配合。在上述例子中，任务线程处于 Sleep 状态，这时候 Future 的 cancel 方法给任务线程发送中断信号， 任务线程会抛出 InterruptedException 异常，上述例子中我们正确捕获了该异常以终止任务。关于如何正确开发可停止任务请参照后续【Java 开发】中相关章节。

创建部署组

注意：如果通过微服务平台（Service Framework，TSF）部署任务应用，则不需要创建部署组。
在部署组管理中，点击【新建部署组】按钮新建一个部署组。这里项目是用作权限控制，可以先任意选择一个有权限的项目。并发数配置的是该部署组下的任务应用（或者称之为执行器实例）允许多少个任务并发在上面执行，0表示不限制。

创建完成后点开部署组详情，在这里我们可以拿到部署组 ID 以及 TCT 服务端地址，后面配置任务的时候会用到。



任务配置

TCT 任务可以通过两种方式进行配置，一种是 application.yml 文件，一种是通过命令行参数，例如 -Dtct.client.groupId。

详细的配置参数说明以及如何获取，请参考后续【Java 开发】中【任务配置】章节。
这里需要重点注意的是，instanceId 表示一个应用实例的 ID，必须在部署组范围内是唯一的。

application.yml

```
tct:
  enabled: true
  server:
    host: 10.0.8.24
    port: 28000
  client:
    groupId: BjFnVcXkVw
    instanceId: tct-demo-ins1
    accessKey: xxx
    secretKey: xxx
    environments:
      - Java
```

命令行参数

```
java \
-Dtct.server.host=10.0.8.24 \
-Dtct.server.port=28000 \
-Dtct.client.groupId=BjFnVcXkVw \
-Dtct.client.instanceId=tct-demo-ins1 \
-Dtct.client.accessKey=xxx \
-Dtct.client.secretKey=xxx \
-jar tct-demo.jar
```

任务部署

TCT 并不限定任务的部署方式，任务应用可以采用任意的方式进行部署，例如虚机部署、Docker 部署，或者其他应用或服务发布平台进行部署。

TSF 部署

TCT 已经和微服务平台（Service Framework，TSF）打通，如果通过 TSF 进行部署，无需手动在 TCT 创建部署组，TCT 能够直接识别 TSF 部署进行任务调度。
前往 TSF 控制台，上传应用 jar 包，然后前往部署组管理页面部署应用即可。

虚机部署

直接启动 Java 进程，并通过命令行参数配置相关参数。

```
java \
-Dtct.server.host=server.shanghai.tct \
-Dtct.server.port=28000 \
-Dtct.client.groupId=BjFnVcXkVw \
-Dtct.client.instanceId=tct-demo-ins1 \
-Dtct.client.accessKey=xxx \
-Dtct.client.secretKey=xxx \
-jar tct-demo.jar
```

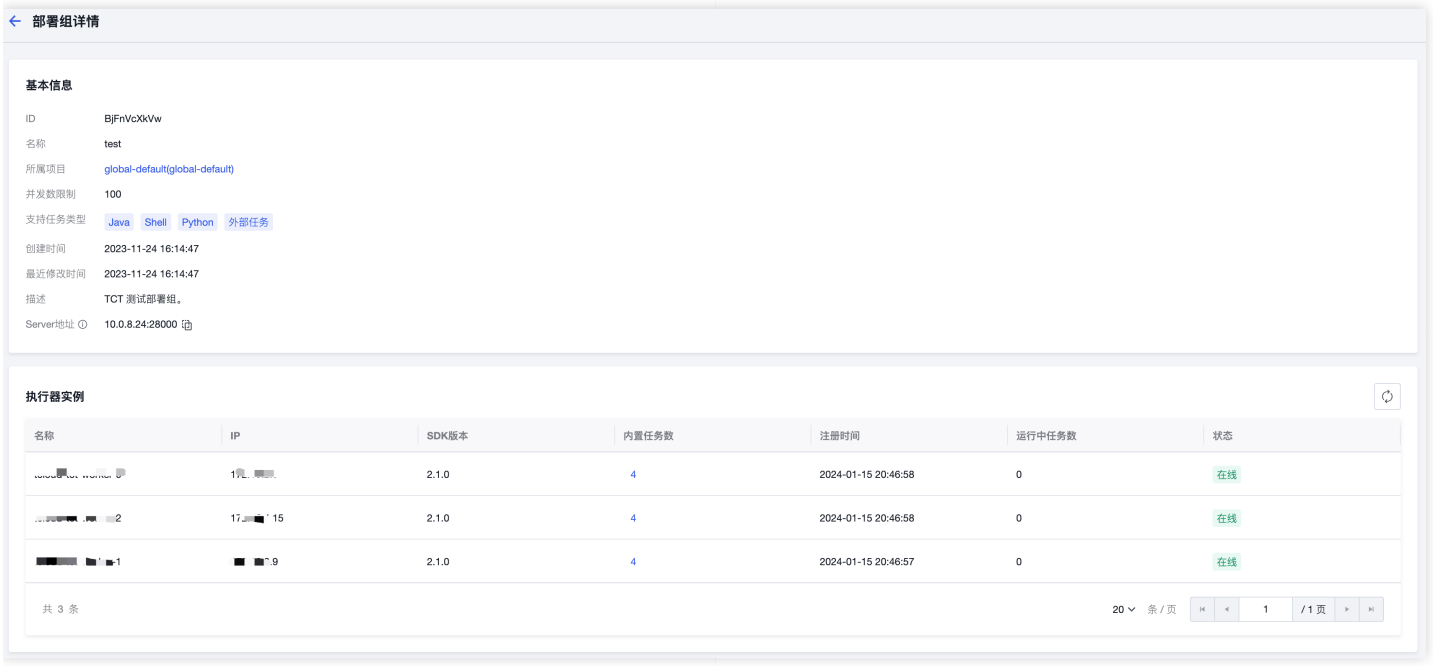
容器部署

将 Jar 包打包成镜像，并通过 Docker 进行启动：

```
docker run tct-demo:v0.1 java \
-Dtct.server.host=server.shanghai.tct \
-Dtct.server.port=28000 \
-Dtct.client.groupId=BjFnVcXkVw \
-Dtct.client.instanceId=tct-demo-ins1 \
-Dtct.client.accessKey=xxx \
-Dtct.client.secretKey=xxx \
-jar tct-demo.jar
```

查看执行器实例

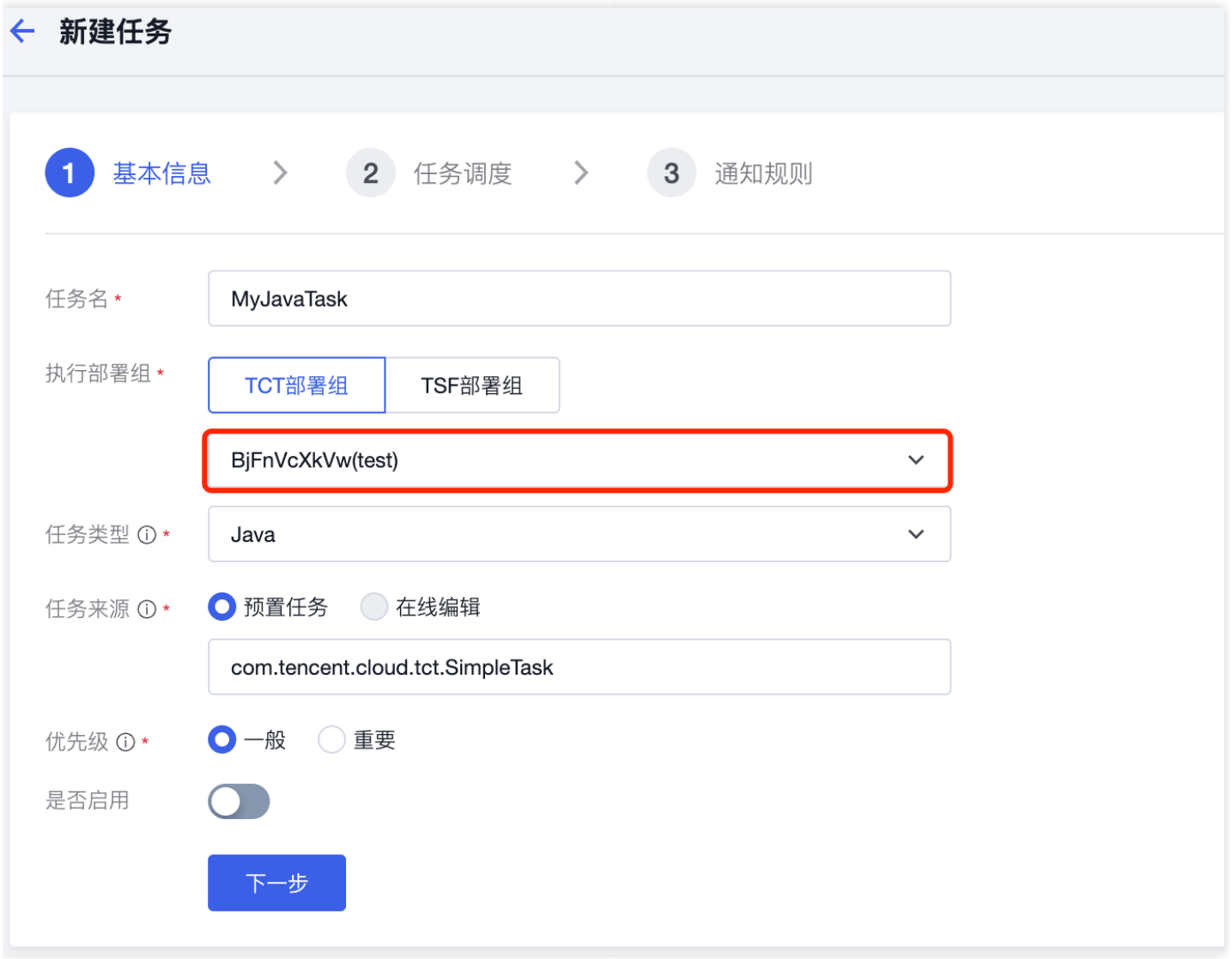
任务应用成功启动后，会注册到 TCT 对应的部署组中，并且会上报该任务应用支持的任务类型以及内置的任务列表。可以在部署组详情页查看对应的执行器信息：



创建任务

创建任务时选择刚才创建的部署组，然后选中开发的 Java 预制任务 SimpleTask。

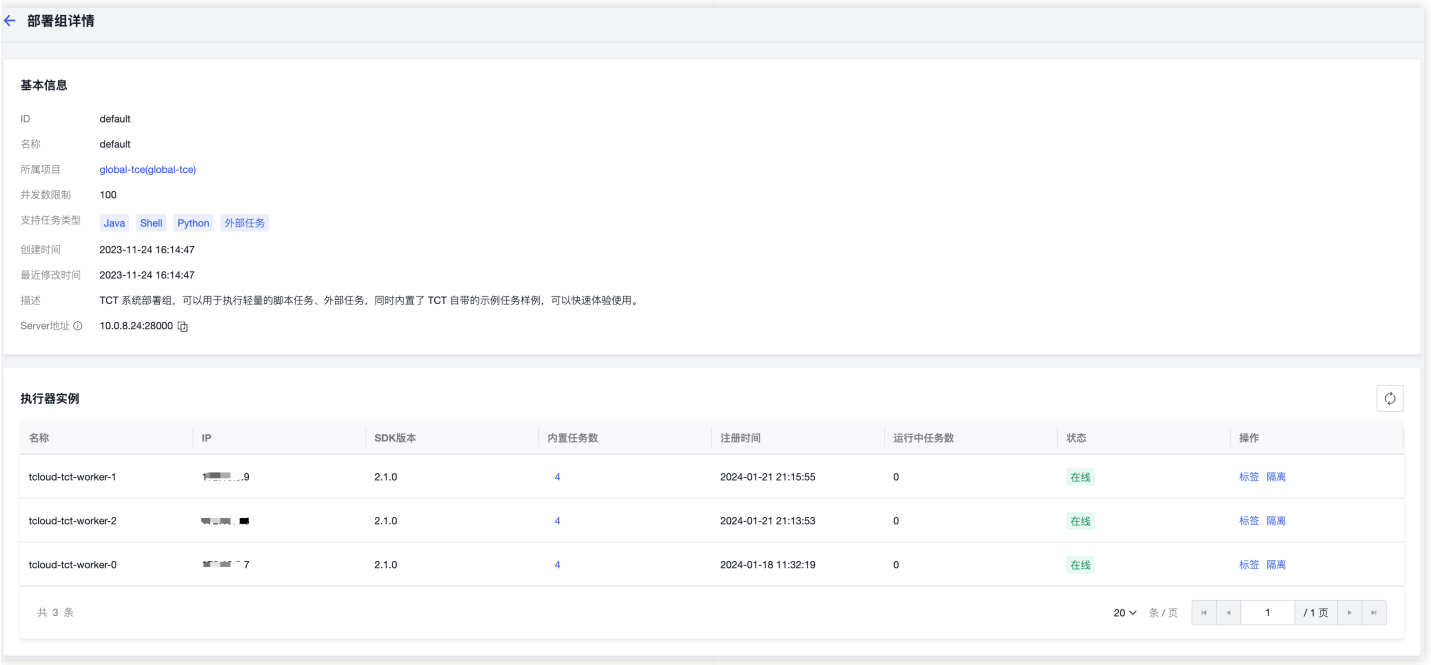
如果前面是通过 TSF 部署的任务应用，则创建任务时选择 TSF 部署组，选中对应的部署组。


新建任务
1 基本信息
2 任务调度
3 通知规则
任务名 *
MyJavaTask
执行部署组 *
TCT部署组
TSF部署组
BjFnVcXkVw(test)
任务类型 *
Java
任务来源 *
预置任务
在线编辑
com.tencent.cloud.tct.SimpleTask
优先级 *
一般
重要
是否启用
下一步

用户手册
部署组

部署组

部署组在 TCT 中用于管理执行器实例（任务应用实例），执行相同业务功能的执行器实例注册到同一个部署组中，在部署组中进行任务的调度（如广播、分片）执行，同时 TCT 通过部署组进行相关的权限控制。
注意，同一部署组中的执行器实例必须是功能相同的，即支持相同的任务类型，包含相同的内置任务。

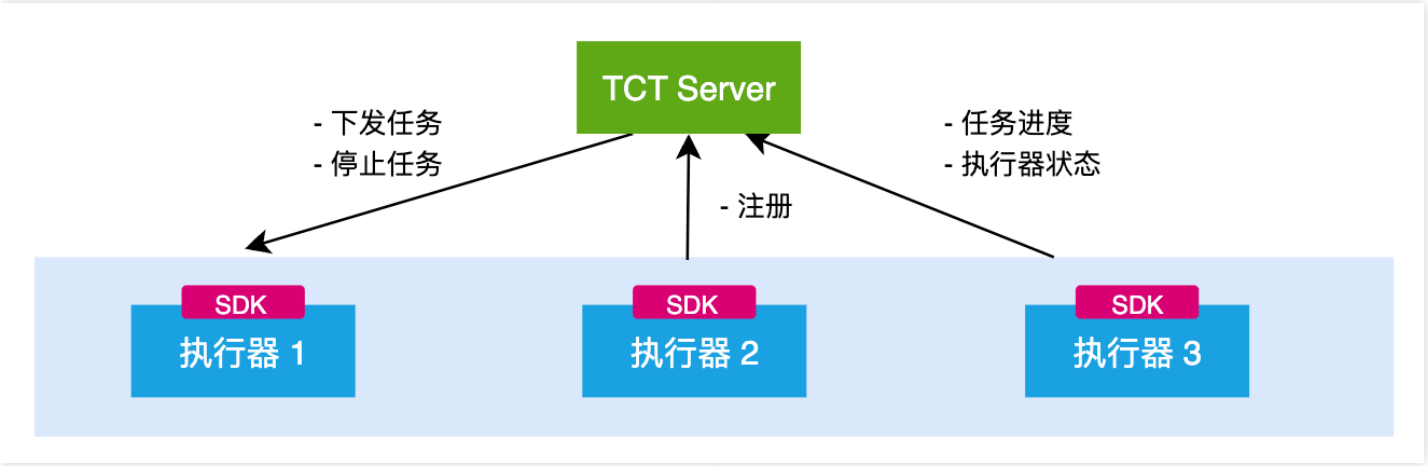


部署组中主要涉及几个重要的属性：

- 并发数限制：用于指定该部署组下的执行器实例允许多少个任务同时执行。可以在创建部署组时配置该参数。需要特别注意的是，这里的并发限制并不是一个绝对严格的并发控制，在一些特殊场景下可能会略微偏离这个限制值，例如限制 100，可能某个特殊场景下运行中任务数为 101。因此对于并发数很敏感的场景不要通过它来做并发控制，例如期望严格控制并发数为 1 的场景。
- 支持任务类型：表明该部署组下的执行器实例支持哪种类型的任务，该属性不是创建部署组时指定的，而是部署组下的执行器实例上报的，执行器实例上报的时候会报告其支持的任务类型，例如支持 Python 脚本的执行器实例上必须具备执行 Python 的环境。

执行器

TCT 中任务最终由执行器执行，执行器是用户开发的业务应用，它通过 TCT SDK 和 TCT 进行交互，接受 TCT 调度的任务并且上报任务执行进度信息。
执行器启动的时候，TCT SDK 会将该执行器注册（执行器和服务端时钟偏移不能超过3分钟，否则会注册失败）到配置的部署组上，同时启动的时候会扫描该执行器中的内置任务（Java Bean，或者特定目录下的任务脚本）并上报给 TCT 服务端。



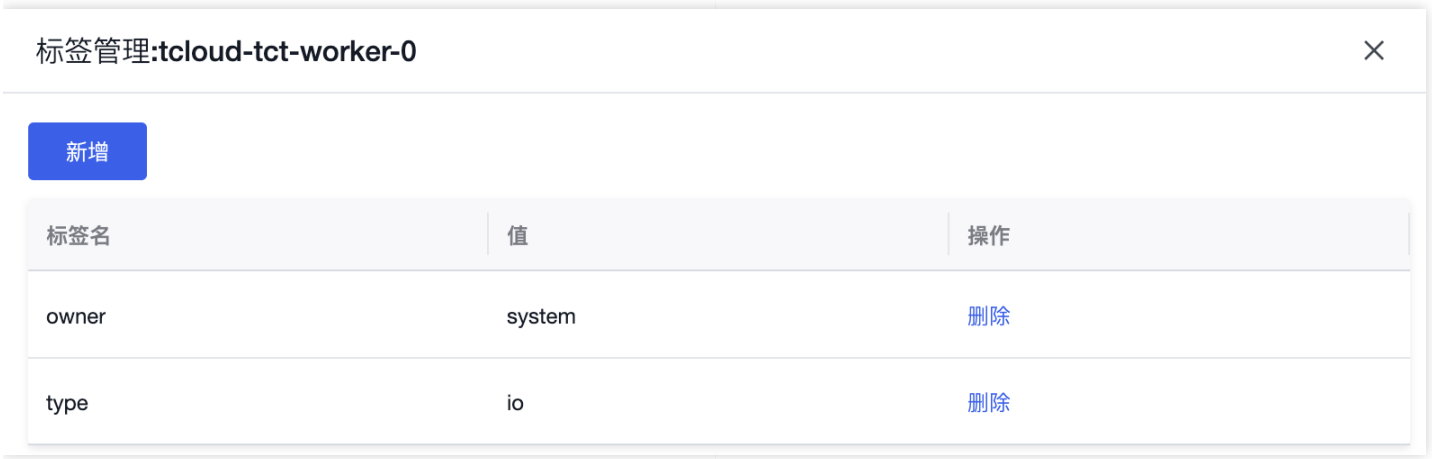
类型

执行器按照使用方式不同可以分为以下的两种类型：

- 专属执行器：用户通过 Java 开发好需要执行的任务，部署到部署组中，这些执行器（应用实例）只用于执行这些预置的 Java 任务。专属执行器可以是一个专门用于任务的应用，也可以在已有的服务中添加 TCT SDK 依赖并开发特定的任务，这样应用部署后照常提供之前的服务，但是额外多了这些任务的调度能力。专属执行器的好处是执行器只用于特定任务的执行，避免其他任务的干扰（例如资源占用），确保了隔离性和可靠性。
- 通用执行器：执行器也可以用于执行通用的任务，这时候执行器类似于执行任务的 Agent，我们既可以在执行器中包含 Java 开发的预置任务，也可以在特定目录中管理预置的 Python、Shell 等任务类型。同时除了这些预置的任务，也可以由 TCT 直接动态下发任务内容（如 Python、Shell 脚本）到这些执行器上执行。TCT 自带的默认部署中的 tct-worker 执行器就属于通用执行器。

标签管理

标签是附加在执行器实例上的键值对，用于指定对用户有意义且和执行器相关的标识属性。标签可以用于组织和选择执行器子集。执行器在启动的时候，可以通过配置项指定标签集合，也可以在 TCT 控制台上手动管理（添加、删除）执行器实例的标签。执行器的标签可以用于指定任务调度的调度策略，例如通过标签过滤执行器实例。

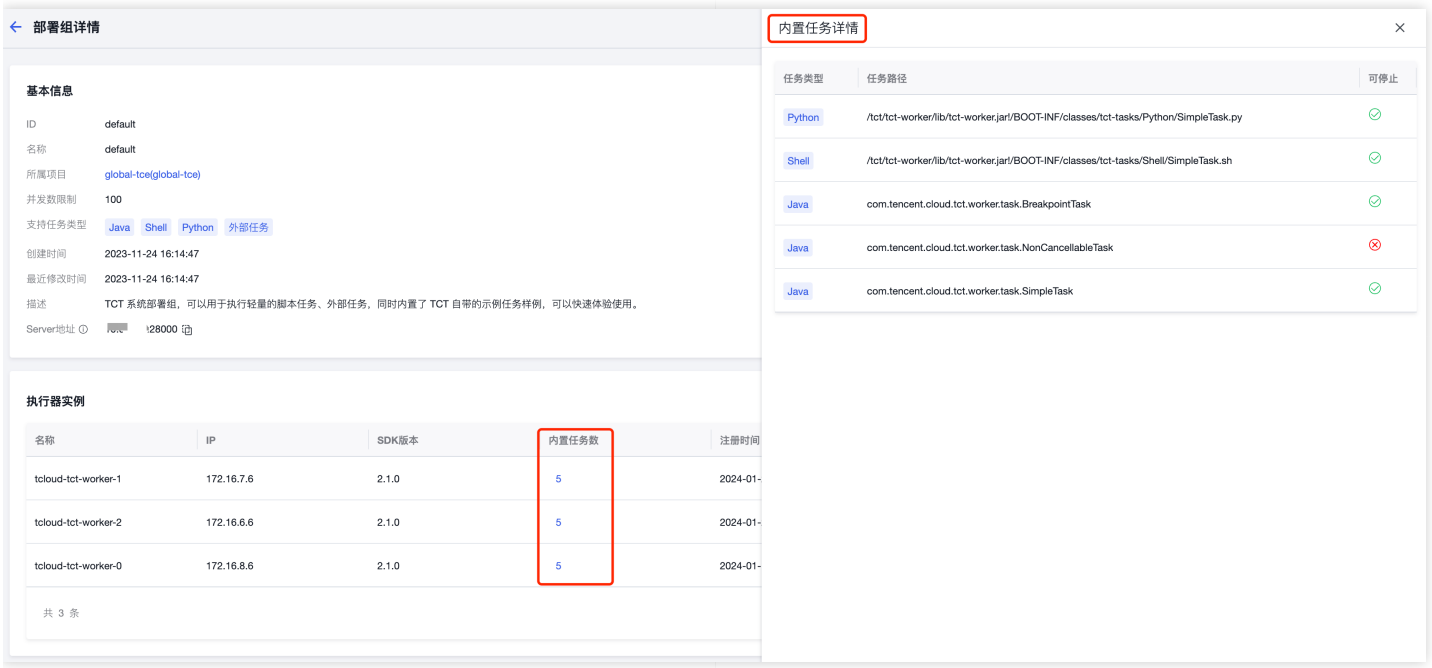


隔离/恢复

在某些情况下，我们会希望隔离部署组中的部分执行器实例，不让任务调度其中，例如需要对某个执行器实例进行运维（升级版本、调整资源等）或者下线。隔离后的执行器实例不会有新的任务再调度其上执行，但是已经调度的任务会继续执行。隔离的实例也可以在 TCT 控制台上进行恢复，恢复后执行器实例继续接受任务调度。

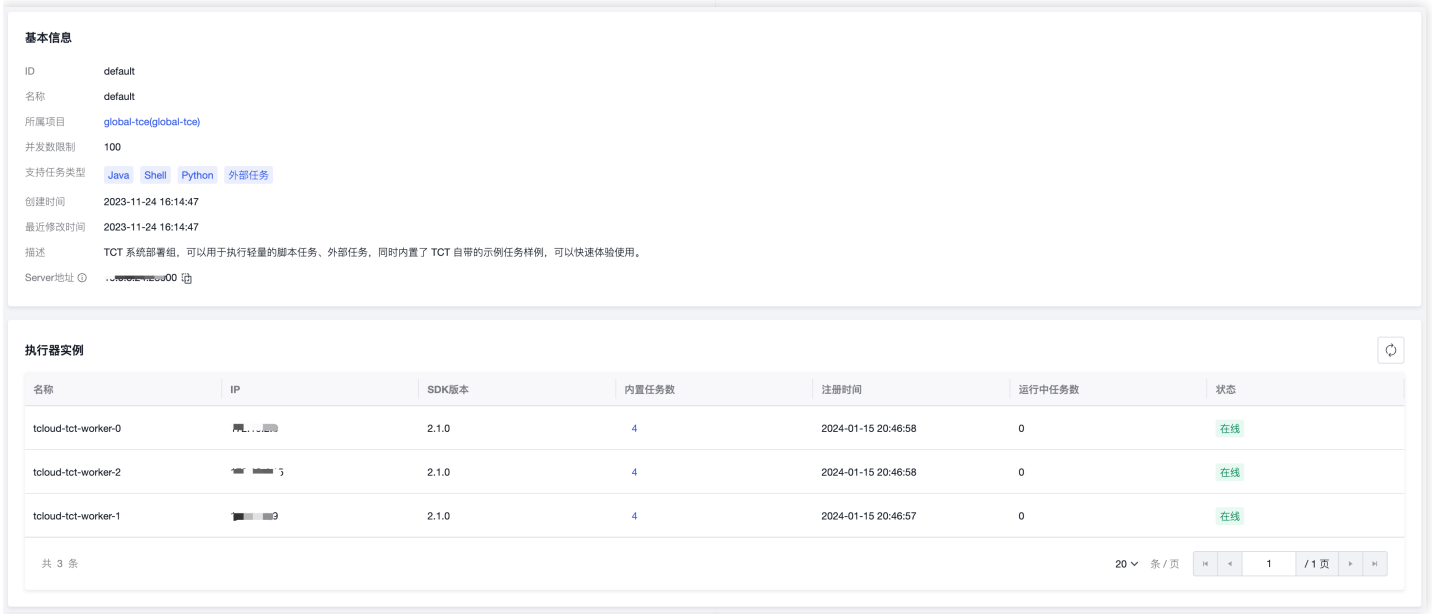
内置任务

内置任务是执行器启动时扫描到的任务，可以是在执行器代码中开发的 Java 任务，也可以是放在特定目录下的 Python、Shell 脚本。



默认部署组

TCT 自带了一个默认的部署组（default），并且自带了容器化方式部署的 3 个执行器（tct-worker），这些执行器可以执行任意类型的任务，包括内置的 Java、Shell、Python 任务以及动态下发的 Shell、Python 和外部任务。
通过它可以快速体验 TCT 的任务调度能力，该部署组是一个公共的部署组，所有用户都可见，从资源以及隔离的角度上考虑，它适用于简单的任务执行，不应作为大范围生产使用。



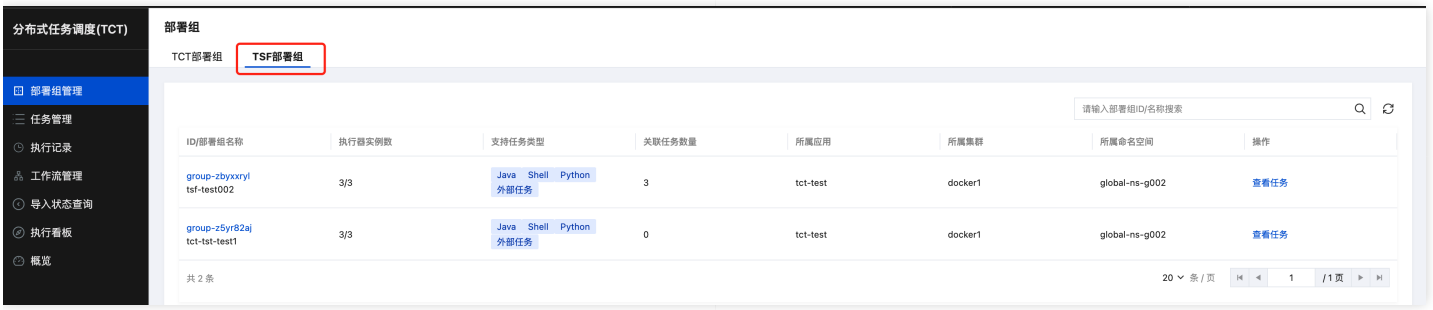
部署组类型

TCT 部署组

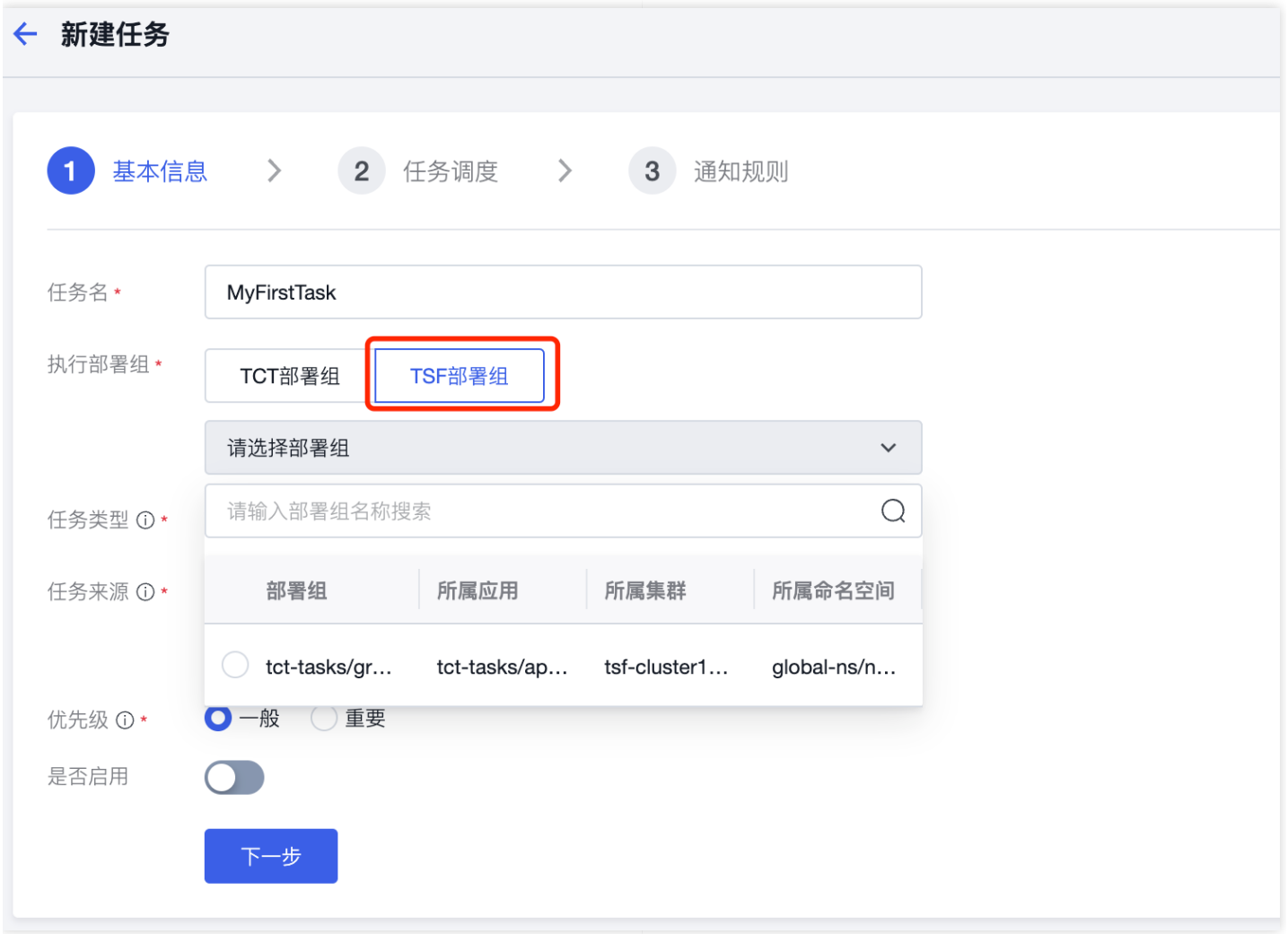
TCT 部署组是完全在 TCT 中进行管理的部署组（TCT 会负责 TCT 部署组的增删改查），在 TCT 控制台上我们可以创建一个部署组，然后将该部署组 ID 配置到执行器实例中，执行器实例启动后会自动注册到该部署组上。

TSF 部署组

除了 TCT 中管理（如增删改查）的部署组，TCT 也和 TSF 进行了打通，能直接识别 TSF 中的部署组。在部署组管理中，我们可以查看开启了 TCT 任务调度的 TSF 部署组。注意这里只有那些开启了 TCT 任务调度，并且正确配置了 TCT 参数（如 TCT 服务地址、AK/SK），成功注册到 TCT 上的 TSF 部署组才会在该列表中显示。



在我们创建任务的时候可以直接选择 TSF 部署组，例如：



注意通过 TSF 部署组部署的执行器实例也可以显式地配置一个 TCT 部署组 ID，让执行器示例注册到 TCT 部署组上，而不是使用默认的 TSF 部署组。在以下几个场景下使用 TCT 部署组会比较合适：

- 希望对部署组进行资源级别的细粒度权限控制，TCT 部署组完全由 TCT 控制，通过 CAM 和项目实现了资源级别的权限控制。
- 多个 TSF 部署组的执行器实例期望放到统一的一个部署组中进行调度，这时候需要将这些执行器示例都注册到 TCT 部署组上。例如在多 AZ 架构下，每个 AZ 下部署了一个 TSF 部署组，期望任务能够在这些跨 AZ 的执行器上调度，实现 AZ 级别容灾。

任务

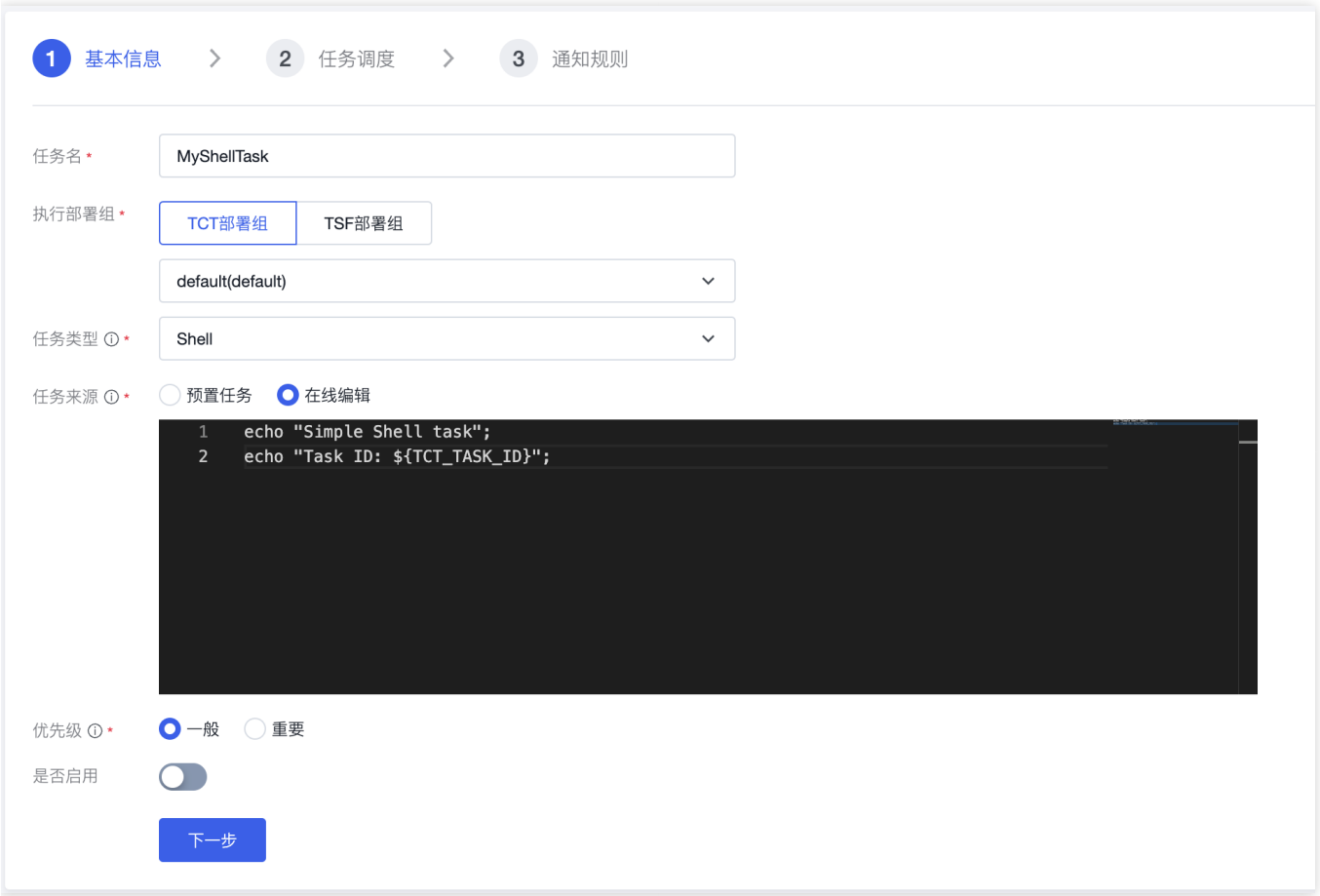
任务类型

Java 任务

TCT 执行器默认支持的任务类型，Java 任务只支持预置在执行器中的任务。预置任务既可以是声明成 Bean 的任务，也可以是没有声明成 Bean 的任务，唯一的区别是，声明成 Bean 的任务会被 TCT SDK 自动发现并上报到 TCT，创建任务时可以从下拉列表中直接选择任务类，否则需要手动输入完成的任务类路径。
Java 任务的开发请参考后续【Java 开发】章节。

Shell 任务

Shell 任务用于执行 Shell 脚本（使用 Bash），既可以将 Shell 脚本预置在执行器能访问到的特定目录中（详见后续的【内置任务】章节），也可以在创建任务时直接在线编辑脚本。脚本中使用到的工具需要用户自己正确地安装到执行器中。



任务参数

在创建任务时，我们可以配置任务参数，任务参数将会被用作脚本的命令行参数，例如以下的任务参数，在执行任务脚本时相当于：

```
bash script.sh --sleep=3 --fail
```

任务元数据

TCT 会自动将任务的元数据以环境变量的方式注入到 Shell 脚本中，用户在编写 Shell 脚本时可以直接使用以下的元数据：

环境变量	说明
TCT_TASK_ID	任务 ID
TCT_TASK_LOG_ID	任务流水 ID，可以认为是任务详情版本的 ID
TCT_BATCH_ID	任务批次 ID
TCT_BATCH_LOG_ID	任务批次流水 ID
TCT_EXECUTE_ID	执行子任务 ID
TCT_EXECUTE_LOG_ID	执行子任务流水 ID
TCT_SHARD_KEY	分片任务中该分片子任务的分片 KEY
TCT_SHARD_VALUE	分片任务中该分片子任务的分片值
TCT_SHARD_TOTAL	分片任务中总分片数

因此，在编写 Shell 脚本时，可以这种引用相关的元数据：

```
echo "Task ID: ${TCT_TASK_ID}"
```

Python 任务

Python 任务用于执行 Python 脚本，既可以将 Python 脚本预置在执行器能访问到的特定目录中（详见后续的【内置任务】章节），也可以在创建任务时直接在线编辑脚本，脚本中使用到的 Python 包需要用户提前在执行器示例中准备好。

任务参数

在创建任务时，我们可以配置任务参数，任务参数将会被用作脚本的命令行参数，例如以下的任务参数，在执行任务脚本时相当于：

```
python script.py --sleep=3 --fail
```

任务元数据

TCT 会自动将任务的元数据以环境变量的方式注入到 Python 脚本中，用户在编写 Python 脚本时可以直接使用以下的元数据：

环境变量	说明
TCT_TASK_ID	任务 ID
TCT_TASK_LOG_ID	任务流水 ID，可以认为是任务详情版本的 ID
TCT_BATCH_ID	任务批次 ID
TCT_BATCH_LOG_ID	任务批次流水 ID
TCT_EXECUTE_ID	执行子任务 ID
TCT_EXECUTE_LOG_ID	执行子任务流水 ID
TCT_SHARD_KEY	分片任务中该分片子任务的分片 KEY
TCT_SHARD_VALUE	分片任务中该分片子任务的分片值
TCT_SHARD_TOTAL	分片任务中总分片数

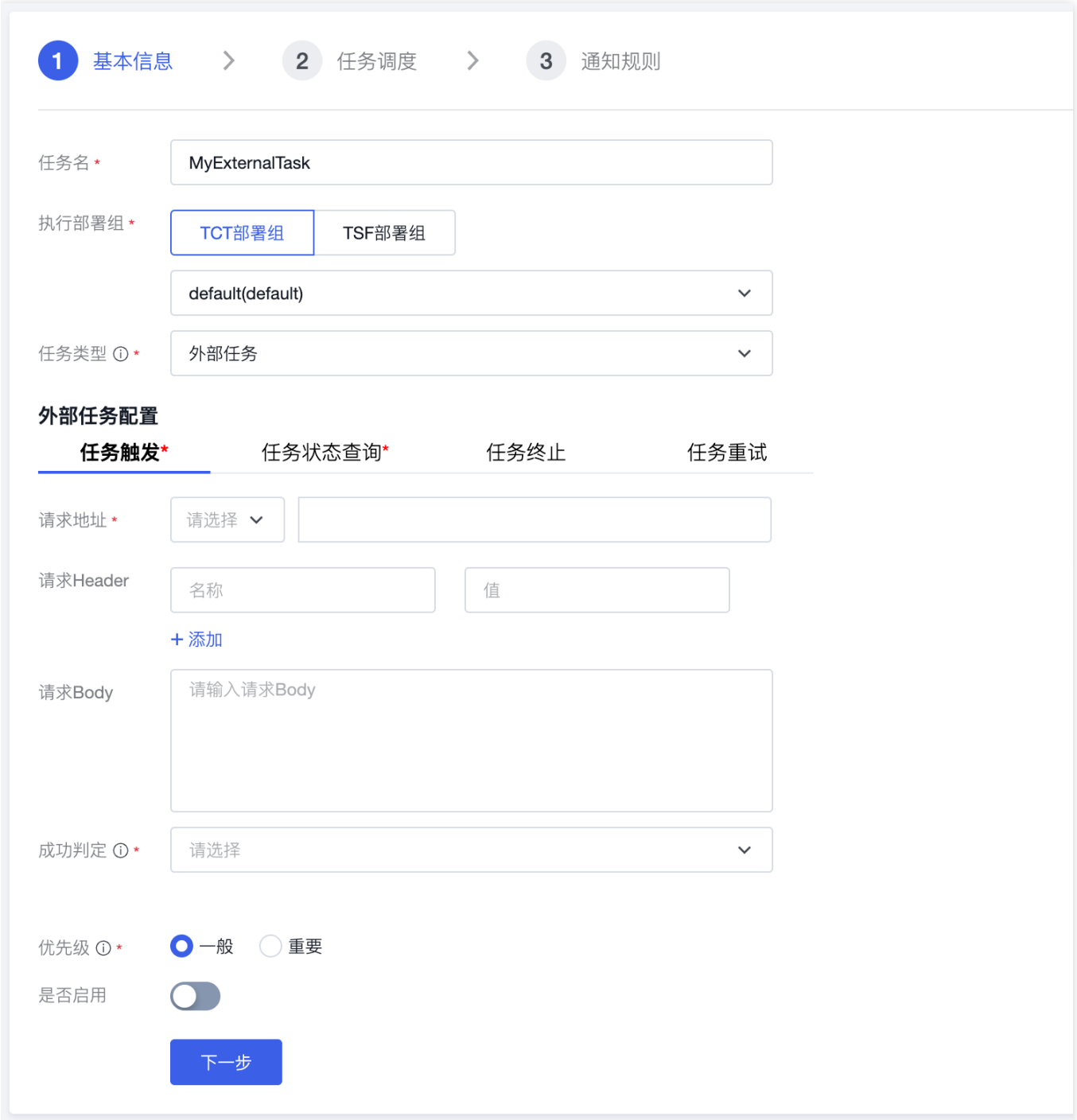
因此，在编写 Shell 脚本时，可以这种引用相关的元数据：

```
import os
envs = os.environ
print "Task ID: %s" % (envs["TCT_TASK_ID"])
```

外部任务

顾名思义，外部任务的任务执行由 TCT 外的其他任务调度系统完成，TCT 只是把外部系统任务的执行封装成 TCT 的任务。典型的使用场景是由 TCT 任务编排多个外部系统任务的执行，例如金融场景下，不同的业务系统可能都有自己的任务调度系统，TCT 可以通过工作流将不同系统的任务进行编排按照给定的顺序以及依赖关系进行执行。

外部任务创建



创建外部任务时，需要配置外部任务调度系统的 API 用于和外部系统交互，包括：

任务触发 API

该 API 用于触发外部调度系统执行给定的任务，当外部任务在 TCT 里被触发时，执行器会调用该 API 触发任务在外部系统的执行。
TCT 对该接口没有具体的限制，只要求返回 JSON 类型的返回体，同时返回体中包含字段用于标识该次触发的任务，用于后续终止或者查询任务进度。

任务状态查询 API

该 API 用于查询任务在外部系统中执行的状态，并且收集任务执行的日志。TCT 在执行外部任务时会定期通过该 API 获取任务的执行进度以及执行日志，当查询到任务执行结束时结束外部系统的执行。

请求体

该API 的请求体可以由外部系统自由设计，一般来说请求体里需要包含任务触发（或者任务重试）接口返回的某一个 ID 字段用于标识具体的一次触发（对于 TCT 来说就是任务批次 ID）。
在创建外部任务时，该接口请求体中 可以引用之前任务触发（或者任务重试）接口返回体里的字段，采用 ${...} 的格式，例如任务触发返回体为以下内容，可以通过 ${$.Response.BatchID} 来引用 BatchID 字段。

```
{
  "Response": {
    "BatchID": "282390945848967"
     . . .
  }
  . . .
}
```

因此任务查询请求体可以如下引用之前任务触发（或者任务重试） API 返回的字段：

```
{
  "BatchID": "${$.Response.BatchID}"
  . . .
}
```

返回体

TCT 对任务状态查询 API 的返回体有以下要求，以便能够正确提取任务进度以及日志信息。返回体中必须包含以下几个字段：

- State（字符串枚举值 RUNNING、SUCCESS、FAILED、TIMEOUT、TERMINATED）。
- Progress（1-100 整型）表示百分比进度，如果无法衡量可以不设置。
- ExecuteLog（字符串） 表示上报的关键日志，注意对于一个任务的执行日志总长度不能超过 64KB（超过会被截断）。
- DetailUrl（字符串）外部系统中查看任务执行详情的地址，在 TCT 任务执行记录中会显示该地址，点击跳转到外部系统中查看该任务内部执行详情。如果该字段多次上报，TCT平台只显示最新上报内容。

```
{
  "State": "RUNNING",
  "Progress": 50,
```

```
  "ExecuteLog": "Subtask A finished\n Start subtask B",
  "DetailUrl": "http://external.com/task?batchID=xxx"
}
```

任务终止 API

该接口用于终止任务在外部系统中的执行，当我们在 TCT 中终止一个外部任务时，执行器需要调用该 API 终止任务的执行。
TCT 未对该接口做具体的限制，请求体里如果需要引用之前任务触发（或者任务重试）接口返回体里的字段，请参考任务状态查询 API 里的说明。
如果外部调度系统不支持终止任务，也可以不配置该接口。

任务重试 API

该接口和【任务触发】类似，需要在接口返回体里包含新的任务执行的 ID，用于进行后续的任务状态查询或者终止等操作，并且该字段名称必须和任务触发 API 中的保持一致。
请求体里如果需要引用之前任务触发（或者任务重试）接口返回体里的字段，请参考任务状态查询 API 里的说明。

API 配置

- 请求地址：API 的地址，支持 GET/PUT/POST 方法。
- 请求 Header：可以配置一个或多个 Header，例如 token 或鉴权 key。
- 请求 Body：请求体限定使用 JSON 格式，配置 Body 时可以通过 JSON Path 方式引用之前接口返回的字段，例如编

写状态查询 API 时我们可以引用任务触发 API 返回体力的字段。JSON Path 规范可以参考后续【JSON Path 规范】章节。

- 成功判定：用于指定如何判定 API 调用成功，支持返回码的检查和返回体字段（通过 JSON Path 引用）的检查。

获取任务元数据

TCT 在调用外部 API 的时候，会自动注入一个 http header TASK-META，它包含了本次任务触发的批次历史 ID，目前外部系统一般不需要使用到该信息。具体格式如下：

```
{
  "TaskLogId": "197667124972617728",
  "BatchLogId": "198114389551595568"
}
```

具体设置到 http Header 中的值是该 JSON 对象 base64 编码后的值，例如：

```
$ echo '{
  "TaskLogId": "197667124972617728",
  "BatchLogId": "198114389551595568"
}' | jq -c | base64

eyJUYXNrTG9nSWQiOiIxOTc2NjcxMjQ5NzI2MTc3MjgiLCJCYXRjaExvZ0lkIjoiMTk4MTE0Mzg5NTUxNTk1NTY4In0K
```

JSON Path 规范

在外部任务中，编写请求体和指定成功判定条件时都可以通过 JSON Path 引用 JSON 字段，这里的 JSON Path $ 表示根对象，并且用点号 . 引用字段，例如 $.store.book[0].title。
参考链接 https://github.com/json-path/JsonPath

外部任务执行

外部任务的调度在 TCT 里看来和其他类型任务没有任何区别，区别只在执行器如何执行外部任务，外部任务执行的过程可以简单的理解为：

- 开始执行：调用任务触发 API。
- 执行中：定期调用任务状态查询 API，获取执行进度，收集执行日志。
- 执行结束：任务状态查询 API 返回任务结束状态，任务执行结束。
  外部任务执行信息里会包含一个链接，链接到外部系统中执行详情页中，如下图所示：

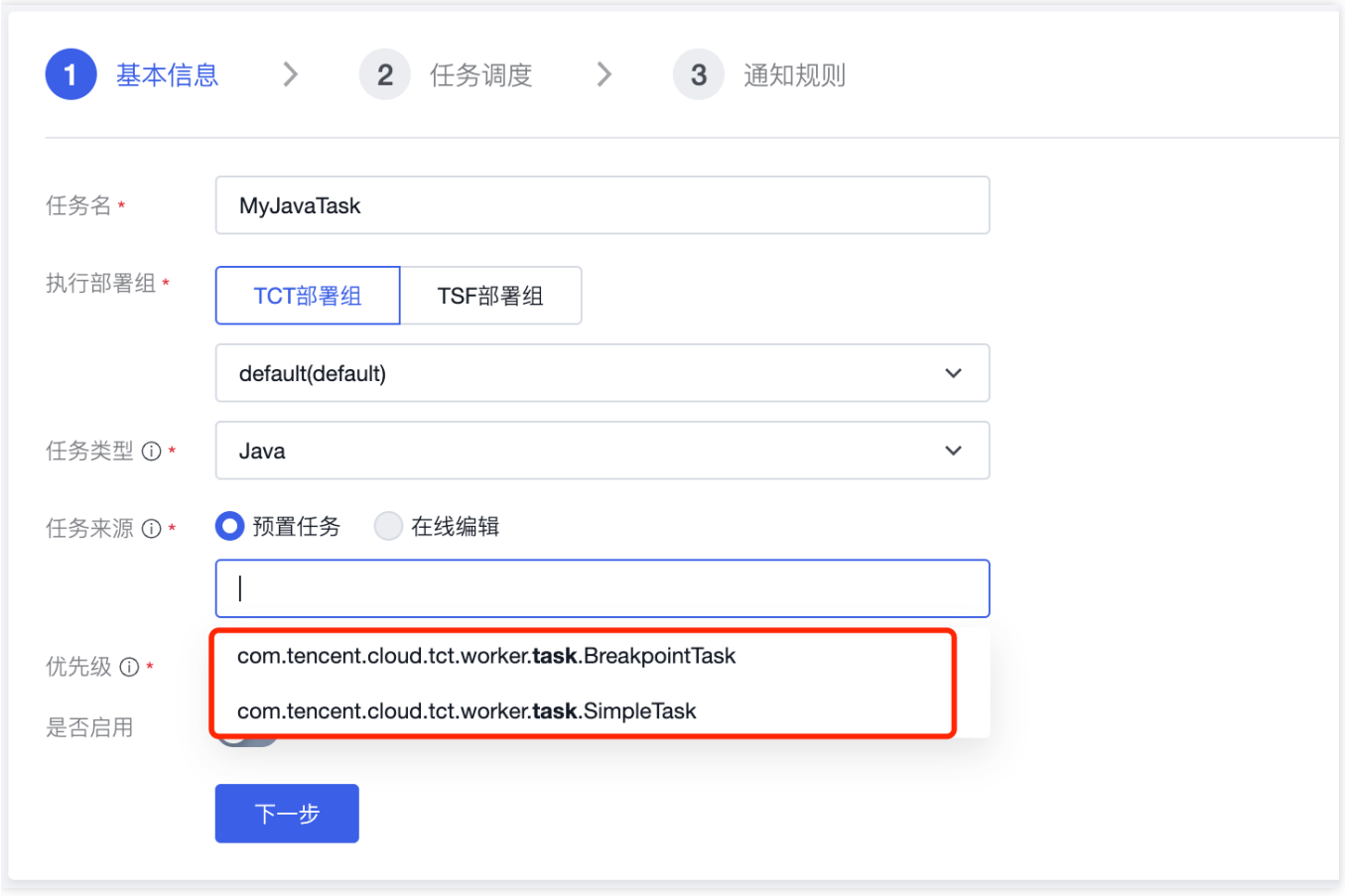
任务来源

预置任务

预置任务是已经包含在执行器是中的任务，Java、Shell、Python 任务类型都支持预置任务，预置任务时 TCT 任务最常用的形态。

Java 预置任务发现

Java 预置任务即那些实现了 ExecutableTask 接口的 Java 类，这些都是预置任务，也都可以被 TCT 调度执行。
但是只有注册成 Bean 的那些预置任务会被 TCT SDK 自动发现并上报到 TCT 中，其他的预置任务需要在创建的时候手动输入完成的类名。在下图中，下拉列表显示的任务是注册成 Bean 的两个预置任务，可以直接选择，其他的任务需要在输入框中输入完整的类名来使用。



Shell/Python 预置任务发现

预置的脚本任务以脚本文件的方式放在执行器特定的目录中（称为TCT任务目录），任务目录下的文件结构如下所示：

```
<任务目录>
|__ Shell
    |__ SimpleTask.sh
    |__ ...
```

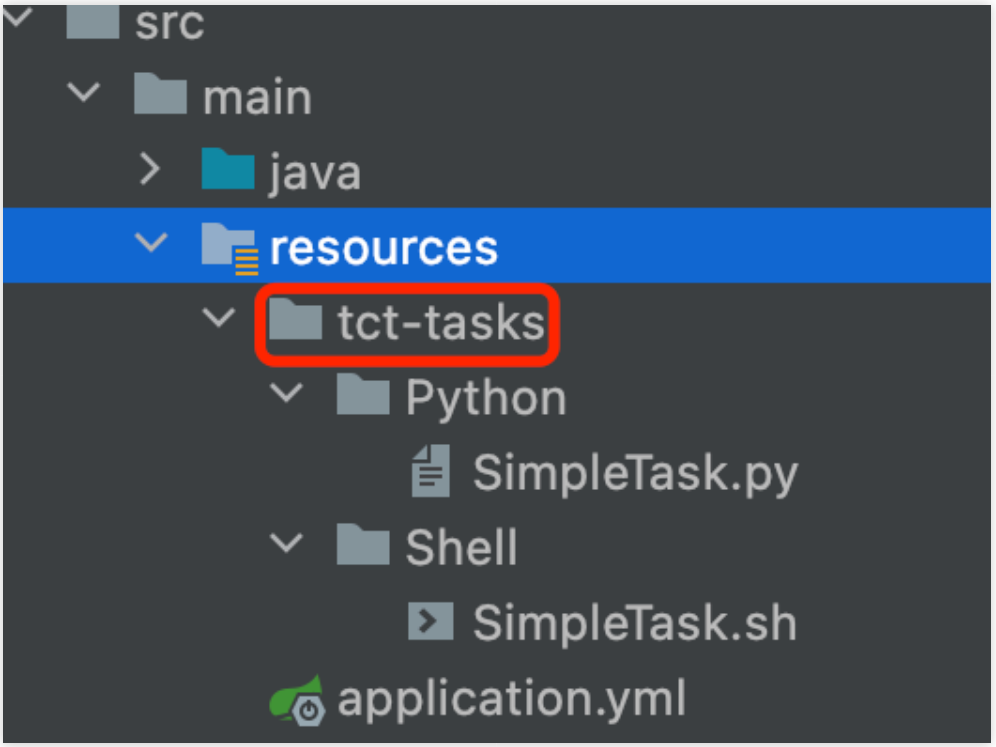
```
|__ Python
    |__ SimpleTask.py
    |__ ...
```

TCT SDK 启动的时候会扫描 TCT 任务目录来收集执行器上的预置任务，任务目录可以在执行器启动参数中配置：

```
tct:
  taskDirs:
    - /tct/tasks
```

TCT 会扫描以下的任务目录：

- 扫描 Jar 包里的 tct-tasks 这个 resources 目录，即执行器代码中 resources 目录下的 tct-tasks 目录：



- 扫描工作目录下的 tct-tasks 目录
  - System.getProperty("user.dir") + "/tct-tasks"
- 扫描执行器启动参数中的目录
  - tct.taskDirs

在线编辑任务

Shell、Python 任务类型支持在线编辑任务内容，创建任务时可以打开在线编辑器编辑 Shell 或者 Python 脚本。

任务优先级

TCT 中任务目前支持一般和重要两个不同的优先级级别，不同优先级的任务在以下几方面体现出不同：

- 任务调度和执行时，如果存在任务的排队现象，重要的任务会进行"插队"，即插队到一般任务的前面，但是同优先级的任务还是先到先出的顺序。
- 如果系统健康度下降，处理能力不足，这时候一般的任务触发后会被概率性地限制调度，而直接进入被限流的终态（详见后续【调度限流】章节），而重要的任务不会被限流。

触发规则

定时触发

通过 Cron 表达式指定的定时触发，Cron 采用标准的 Quartz 语法，例如 "0 * * * * ?" 表示每分钟触发一次。

周期触发

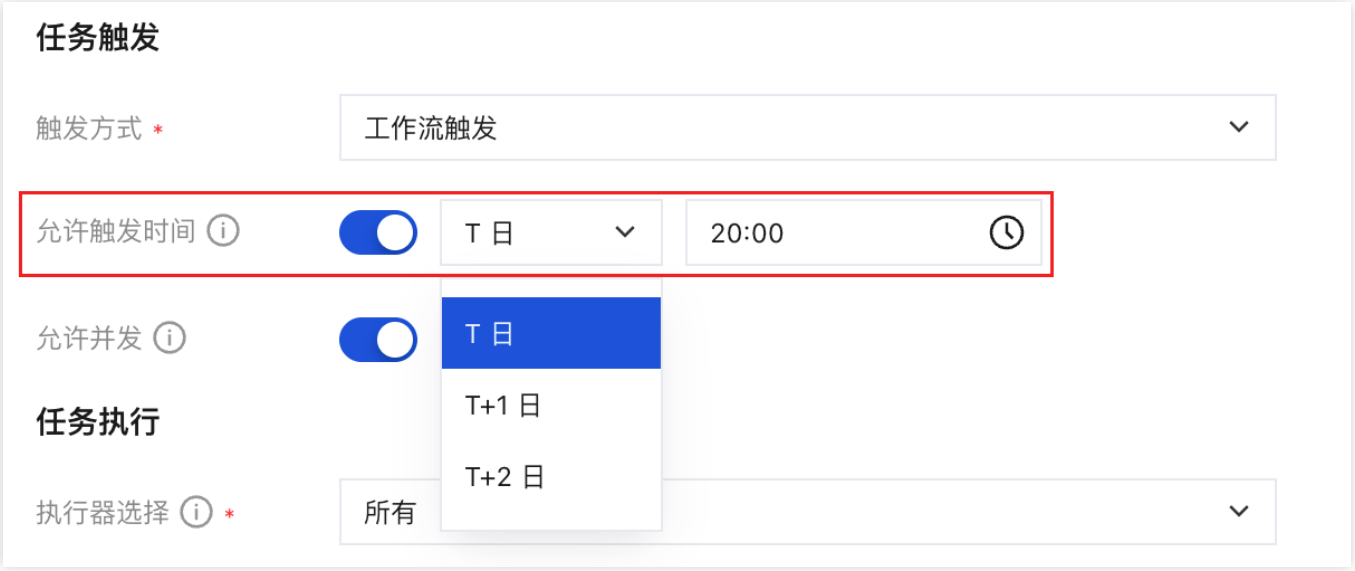
周期触发指定固定的触发周期，例如每隔 5 分钟触发一次。注意这里的触发间隔衡量的是上次触发的时间到这次触发的时间，而不是上次任务执行结束的时间到这次触发的时间。

工作流触发

工作流触发表示该任务将用于构建工作流，由工作流执行去触发。

对于工作流触发的任务，我们还可以选择配置一个允许触发的时间，指定自然日的几点之后才允许触发该任务，该参数在金融场景中比较常用，这里的 T 日表示的是工作流开始执行的日期，T+1 日表示第二天。如下图所示我们配置了 T 日的 20:00，那么该任务在工作流中要触发执行需要同时满足：

- 该任务的前置依赖任务都完成。
- 时间是 T 日的 20:00 之后，否则要等到 20:00 才允许触发。



特定时间触发

特定时间触发用于指定任务只在特定的时间点触发一次，例如 2024/01/01 00:00:00。

手动触发

手动触发用于手动触发任务执行一次，无论任务配置的何种触发规则以及是否启用，都可以手动触发执行。

触发限制

对于定时触发和周期触发的任务，可以限制任务的触发，包括：

- 开始触发时间：只有该时间后才允许按照触发规则自动触发任务执行。
- 结束触发时间：该时间后就不再触发任务执行，该时间之后任务会自动进入禁用状态。
- 最多触发次数：任务最多触发的次数，注意手动触发任务也统计在次数中。

  > **注意：**这里最多触发次数是以当前能够查询到的执行记录数来统计的，所以会受执行记录清理的影响。如果时间超过结束触发时间或者触发次数超过最多触发次数，任务会自动被设置为停用。

执行方式

TCT 支持以下三种执行方式：

- 单点执行：在可用的执行器实例中选择一个实例用于执行该任务，选择的策略目前为随机选择。
- 广播执行：将任务调度到所有的执行器实例中执行，因此在任务调度的时候有多少个可用的执行器实例就会产生多少

个子任务。

- 分片执行：分片执行是将任务拆分成多个分片（子任务），然后在所有可用执行器实例上调度这些分片子任务。分片数量由用户在创建任务的时候指定，每个分片都会有一个自己的分片编号（1、2、3...），一般来说用户需要设置每个分片的分片参数，TCT 在调度分片的时候会将分片参数传递给执行器。

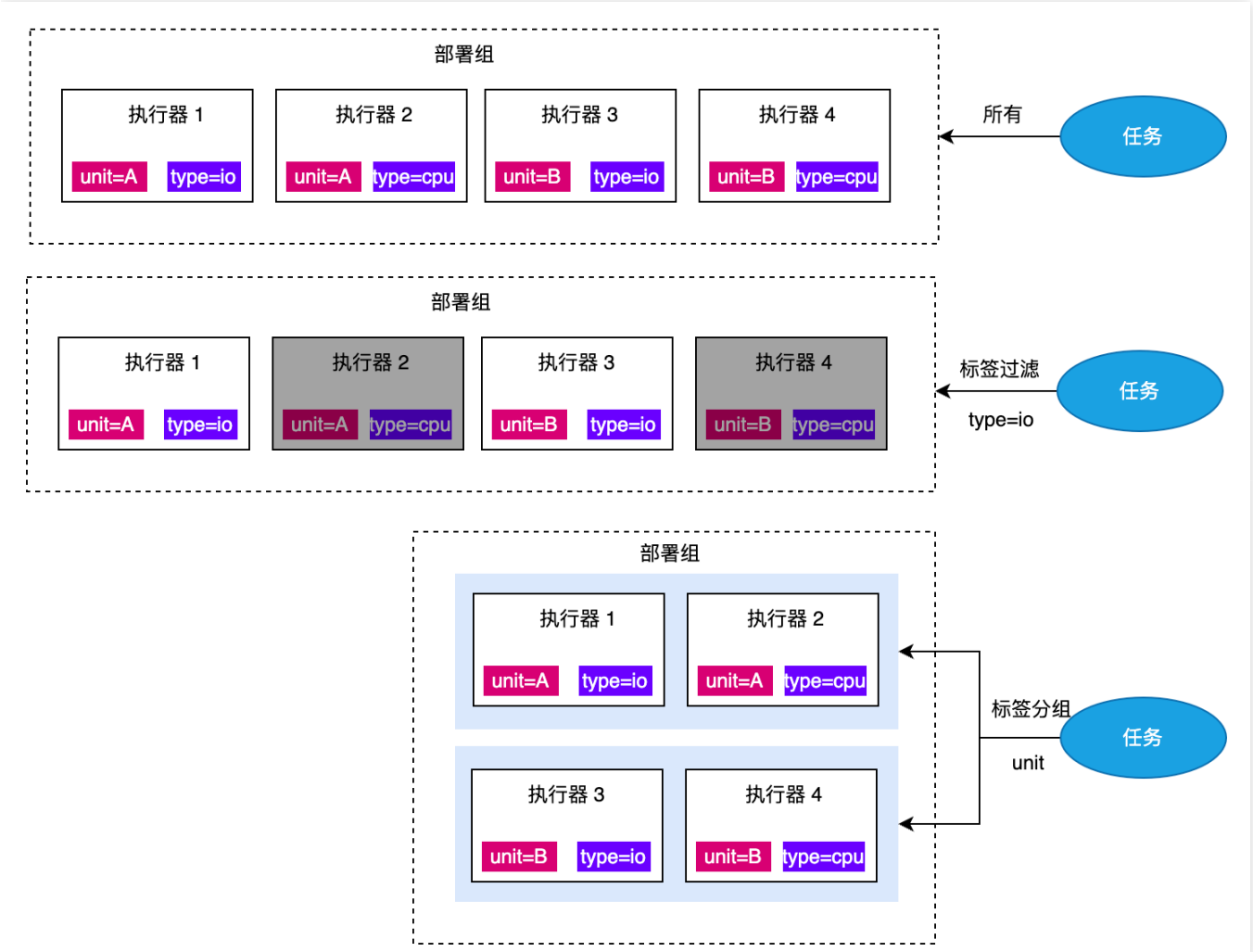
调度策略

调度策略用于控制如何将任务调度到执行器上，例如：

- 选择哪些执行器实例用来调度给定的任务。
- 任务 A 和任务 B 不允许同时在同一个执行器上执行。
- 任务 C 和任务 D 尽量在同一个执行器上执行。

TCT 提供了以下两方面的调度控制：

执行器选择



所有

为默认行为，任务会按照指定的执行方式（单点执行、广播执行、分片执行）在部署组内所有可用的执行器实例中调度执行。

标签过滤

通过指定标签选择器筛选匹配的执行器实例执行，而不是部署组下所有可用的执行器实例。

标签分组

将执行器实例按照指定的标签进行分组，标签值相同的放在一组，如果执行器没有给定的标签，则排除在外。执行器分组后任务会按照指定的执行方式（单点执行、广播执行、分片执行）在每个分组上进行调度执行。

任务启用

自动触发（定时触发、周期触发、特定时间触发）的任务只有启用了才会被自动触发执行，在创建任务的时候可以选择是否启用任务。
任务创建好后也可以在控制台上修改任务启用状态，注意工作流触发的任务由工作流触发，任务本身没有启用、禁用一说，也因此工作流触发的任务在控制台上无法进行启用或禁用的操作。

任务超时

创建任务的时候可以给任务配置超时时间，任务调度和执行超过配置的超时时间后，TCT 会将触发的任务批次设置为超时状态。超时的时间会传递到执行器上，由执行器控制任务执行时的超时，超时后终止任务的执行。

任务重试

创建任务的时候可以给任务配置任务运行失败自动重试的次数，以及每次失败重试的间隔，单位：秒。

任务参数

TCT 可以给任务配置执行参数，执行参数的格式 TCT 不做限制，只负责按原样字符串透传，参数的解析由任务逻辑自己负责。

Java 任务参数

例如下图所示，我们设置了 "--sleep=30 --fail" 任务参数，在 Java 中，可以通过 ExecutableTaskData::getTaskArgument 获取到原始参数字符串，然后进行参数的解析，例如使用 org.apache.commons.cli.CommandLineParser。


```
public ProcessResult execute(ExecutableTaskData taskData) {
    // Output: Raw task argument: --sleep=30 --fail
    LOG.debug("Raw task argument: {}", taskData.getTaskArgument());
    . . .
}
```

Shell 任务参数

Shell 脚本类型任务执行的时候会将任务参数作为脚本执行时的命令行参数进行传递，因此上图中的任务参数 "--sleep=3 --fail" 相当于执行脚本时的如下参数：

```
bash script.sh --sleep=3 --fail
```

Python 任务参数

Python 脚本类型任务执行的时候会将任务参数作为脚本执行时的命令行参数进行传递，因此上图中的任务参数 "--sleep=3

--fail" 相当于执行脚本时的如下参数：

```
python script.py --sleep=3 --fail
```

分片参数

对于分片执行的任务，我们可以在高级设置里设置任务的分片参数，分片参数以每个分片的 Key（即分片编号 1,2,3....）为 Key，参数值类型不做限制，具体参数值的解析由任务逻辑自己负责，TCT 会将参数值作为字符串进行透传。
如下图所示，任务包含 3 个分片，我们可以给每个分片（1,2,3）设置分片参数：

- 1：0
- 2：100
- 3：200

任务执行

执行器选择 ⓘ *：所有

执行方式 *：分片执行

部署组每个节点获取控制台指定的分片参数信息并发执行，提升执行效率。

任务参数：--sleep=30 --fail

17 / 10000

超时时间：1 分钟

最大超时时间为24小时

分片数：3

成功判定：分片成功率 >= 100 %

高级设置 ▲

重试次数：0 次

重试次数的范围为[0,10]

重试间隔：0 秒

重试间隔的范围为[0,600]

子任务单机并发数：— 0 +

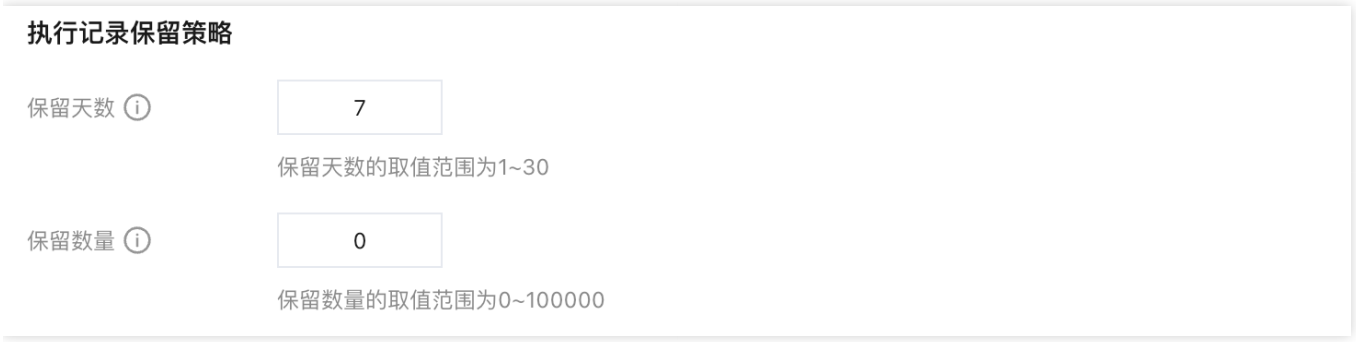
分片参数：1=0,2=100,3=200

在 Java 中，可以通过 ExecutableTaskData::getShardingArgs 获取到分片参数（包括分片 Key、分片参数值、分片总数），而在 Python 和 Shell 任务类型中，分片参数通过环境变量注入到脚本中：TCT_SHARD_KEY、TCT_SHARD_VALUE、TCT_SHARD_TOTAL。

执行记录保留策略

TCT 支持给每个任务指定单独的执行记录保留策略，用户可以指定执行记录保留的天数和执行记录保留的数量，其中保留天数最多支持 30 天，保留数量 0 代表不限制（相当于无限大），最多支持保留 100000 条。

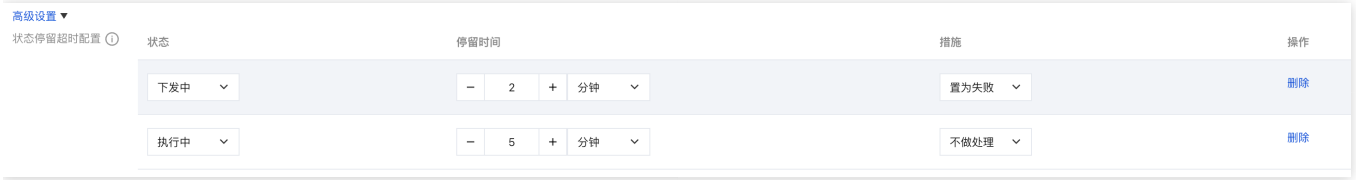
如果配置了两个条件，那么 TCT 会按照每个条件分别去清理执行记录，假设配置保留 3 天，最多保留 1000 条记录，那么 TCT 会清理掉该任务 3 天前的所有执行记录，同时判断执行记录数是否超过了 1000，超过的话保留最新的 1000 条清理掉其他的记录。



状态停留超时配置

在任务的高级配置中，有一个状态停留超时配置，该配置用于指定任务执行时在某个状态停留超过指定时间后的处理措施。例如任务一直卡在下发中，我们可以选择将它直接置为失败，避免一直重试，同时也可以发出告警通知用户。

在这里配置的状态，触发停留超时后都会在监控指标中体现停留超时，用户可以通过配置告警规则用于告警发现。如下图所示虽然我们对执行中停留操过 5 分钟的情况没有采取额外措施，但是执行超过 5 分钟会有监控告警体现这个情况。



并发控制

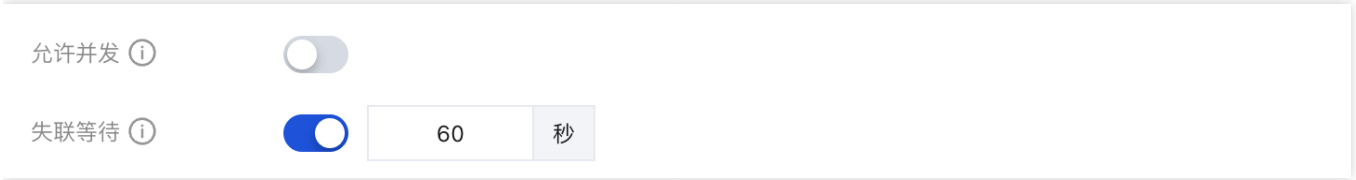
TCT 中涉及三个方面的并发控制：

- 部署组中配置的执行器实例的并发限制，允许多少个子任务同时在一个执行器上执行。
- 分片任务配置的子任务单机并发数，同时允许该任务的多少分片子任务在同一个执行器上执行。

- 任务是否允许并发执行，控制一个任务上一次执行还未结束，是否允许触发新的执行。

如果任务配置为不允许并发执行，那么上一个任务执行没有结束，任务不允许再次执行，到了触发时间后，新的触发记录会直接被置为被拒绝状态。设想一下，如果上一次执行因为异常的原因一直没有结束，那么会一直影响后续的触发执行。例如，任务调度到执行器上执行，执行过程中执行器实例突然下线了，那么 TCT 就无法知道该任务的执行状态，因为无法判断执行器实例是网络隔离了还是已经下线了，这时候任务在 TCT 里会一直处于执行中状态，导致后续的触发执行直接被拒绝。

为了解决这个问题，TCT 做了两方面的优化，一方面如果执行器实例恢复了（例如网络隔离解除、执行器重启上线），TCT 能够知道任务是否还在该执行器上执行，如果已经不存在了，那么 TCT 就可以将任务状态置为一个未知的状态，它是一个终态。之所以不是成功或者失败的状态，是因为我们无法判断任务在业务层面上是否已经完成了。另一方面我们给任务增加了一个失联等待的功能，如果任务不允许并发执行，那么可以配置一个失联等待的时间，如果任务调度到执行器上执行后失联了（例如上面的执行器突然下线了，导致我们无法判断任务状态），失联超过配置的时间后 TCT 会将任务状态置为未知状态。



> 注意：任务配置了失联等待后，如果任务失联被置为未知状态，任务的下一次触发会正常执行，由于上一次执行是未知状态，可能会出现任务并行执行的情况，因此需要严格串行执行的任务需要根据自己的情况谨慎使用这个功能。

聚合输出

TCT 任务执行完后可以以键值对的方式输出执行结果，例如对于 Java 使用 ProcessResult.withOutput(String key, String value) 输出执行结果。
在任务中，我们可以给输出结果配置聚合规则，用于聚合分片任务、广播任务的多个输出值，如求取平均值、求和。

这里待聚合字段必须是任务输出的一个字段，而聚合结果字段是我们为聚合后的值指定的字段名称。

调度限流

调度限流是 TCT 在系统健康性出现问题或者处理能力不足时采用的一种服务降级的策略。它根据系统当前的健康程度，按照任务的优先级，对低优先级的任务进行选择性地放弃，从而确保高优先级任务的正常调度。触发限流的条件是比较苛刻的，不用担心一般优先级的任务经常性地被限流，并且限流发生时也会触发 TCT 的告警。

健康度

系统健康度用来评估系统当前的健康状况，取值 0-1，值越高表示系统越健康，处理能力越强。低于 1 表示系统出现不同程度的拥塞，正常情况下健康度都应该是 1。
健康度的计算考虑了任务调度队列、任务分发队列的当前排队长度以及队列当前的增长趋势。

限流算法

对于重要级别的任务，不会进行限流，限流只针对一般的任务。下述示例代码中，score.get() 用于获取当前系统实时的健康度，正常情况下返回 1，rand.nextDouble 用于获取一个 0-1 范围内的随机小数，因此该方法正常情况下永远返回 false，即不限流。

当系统健康度下降，系统出现一定程度的拥塞，例如 score.get() 返回 0.9，这时候 rand.nextDouble 随机返回 0-1 中的小数，只有返回 0.9-1.0 区间的小数时该方法才返回 true，即限制该任务的调度，而 rand.nextDouble 返回的随机数落在 0.9-1.0 范围内的概率为 0.1，即该任务的此次调度有 10% 的概率会被限流放弃，直接进入被限流状态。

换句话说，如果系统健康度为 x，则一般任务的一次调度有 1.0-x 的概率会被限流。

```
public boolean throttling(int priority) {
    if (priority == Priority.IMPORTANT.getValue()) {
        return false;
    }

    return rand.nextDouble() > score.get();
}
```

执行日志

TCT 支持收集和显示任务的执行日志，如下图所示：

对于 Java 类任务，TCT 收集的日志是通过 SDK 中的 LogReporter::log 上报的日志，而对于 Python/Shell 脚本类任务，TCT 收集的是脚本执行产生的标准输出和错误输出日志，对于外部任务，日志收集的是外部系统通过任务状态 API 上报的执行日志。

断点续跑

断点续跑是针对分片任务的续跑功能，分片任务的一个分片子任务需要处理多个数据，可能出现部分数据处理成功部分处理失败或者未处理的情况，这时候任务可以将任务执行的断点信息（哪些数据处理了，哪些没处理）上报给 TCT，后续从 TCT 控制台触发断点续跑的时候，TCT 会将断点信息下发到执行器，任务执行的时候根据断点信息只执行未处理的数据。

任务停止

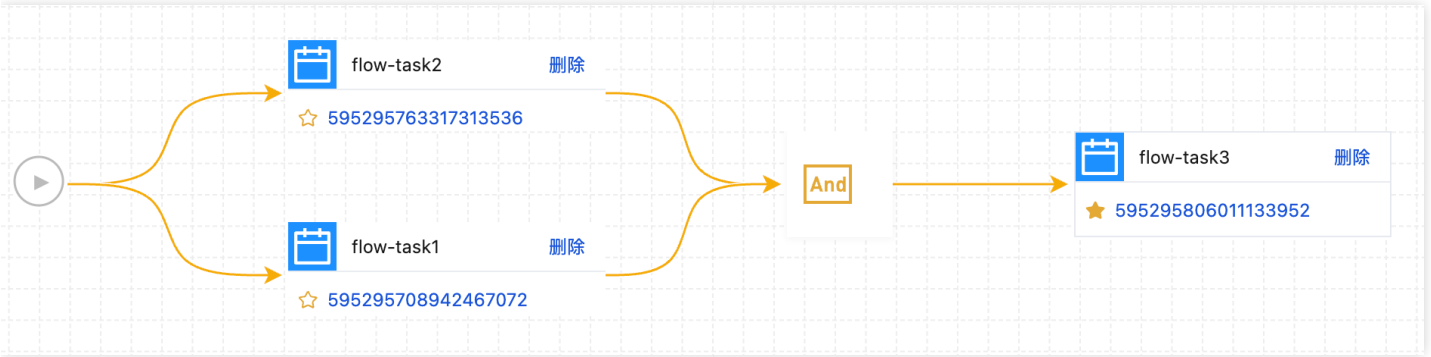
执行中的任务可以手动停止，任务停止时 TCT 会将任务批次状态设置为终止中，并通知执行器终止任务的执行，所有子任务终止后会将任务批次状态置为已终止。

任务删除

TCT 支持单个任务的删除以及批量任务的删除，注意已经用于构建工作流的任务不允许删除。任务删除后会立刻进入删除中状态，删除中状态的任务在 TCT 控制台上不可见，TCT 通过 GC（Garbage Collector 垃圾回收）处理任务的删除，GC 时会清理任务相关的历史版本、任务执行批次（包括批次流水）、子任务执行记录（Execute）等。如果任务未完成清理，后端服务故障重启了，GC 机制在启动的时候也会扫描数据库中处于删除中的任务，尝试继续清理任务相关资源，确保任务最终被清理干净。

工作流

DAG 编排



TCT 中任务和工作流可以按照 DAG（Directed Acyclic Graph，有向无环图）的方式任意编排成工作流，并且支持以下功能：

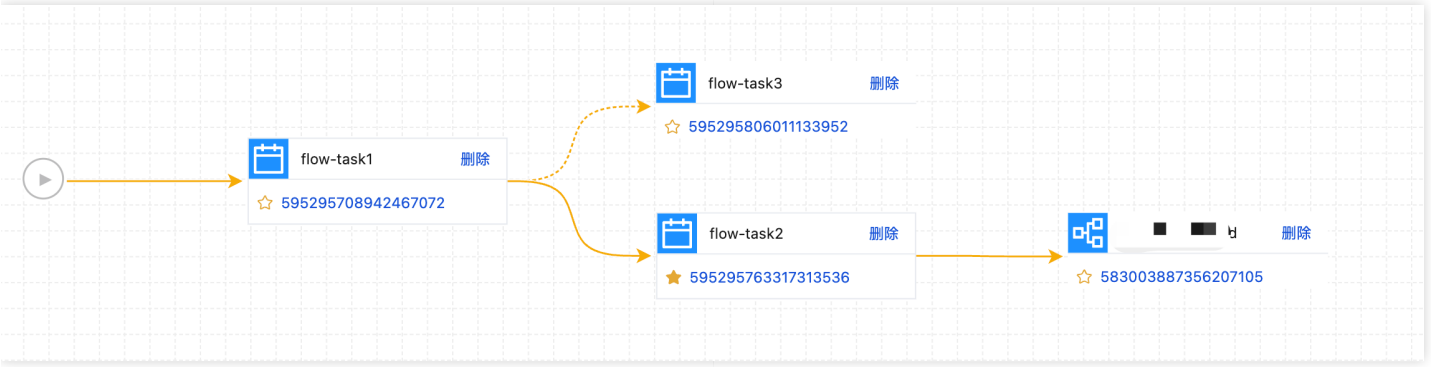
节点类型

工作流中支持任务和工作流两种节点类型，即任务和工作流可以混合、嵌套进行编排。在工作流图中，任务和工作流节点使用不同的图标进行表示。

节点类型	图标
任务节点	
工作流节点	

下图的工作流中，我们包含了一个工作流节点。工作流可以这样进行多层嵌套，TCT 限制最多嵌套 5 层（不包含工作

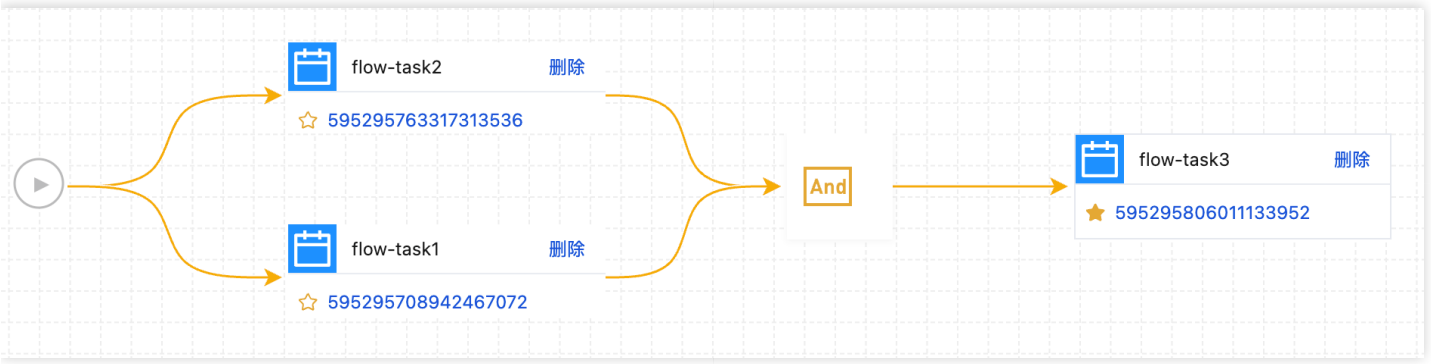
流节点的工作流嵌套深度为 1），一般建议嵌套深度不要超过 3 层。

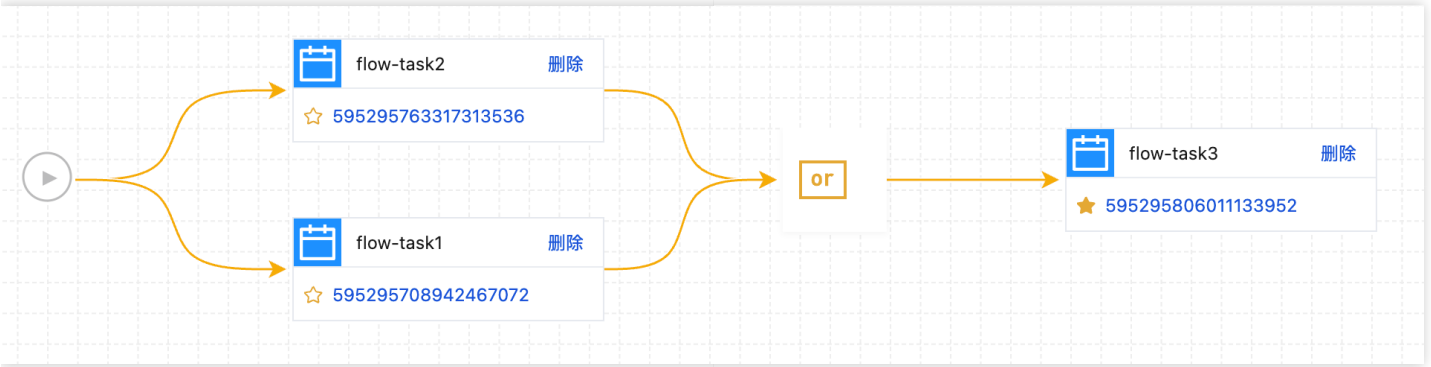


用于构建工作流的节点，无论是任务节点还是工作流节点，触发方式必须选择为工作流触发，并且一个任务或者工作流只能被一个工作流使用，被工作流使用后不允许删除、不允许修改触发方式，删除工作流后才可以删除或者修改触发方式。

逻辑 AND/OR

如果一个任务有多个依赖，可以声明这些依赖关系是 AND（表示所有依赖任务必须满足）还是 OR（表示所有依赖任务只需满足一个）方式，例如：





成功触发/失败触发

工作流中通过图中的边来声明依赖关系，依赖关系既可以是成功触发（以实线表示，依赖的任务执行成功后触发该任务），也可以是失败触发（以虚线表示，依赖的任务执行失败后触发该任务）。如下图所示，task1 执行成功则触发 task2，task1 执行失败则触发 task3。

关键任务

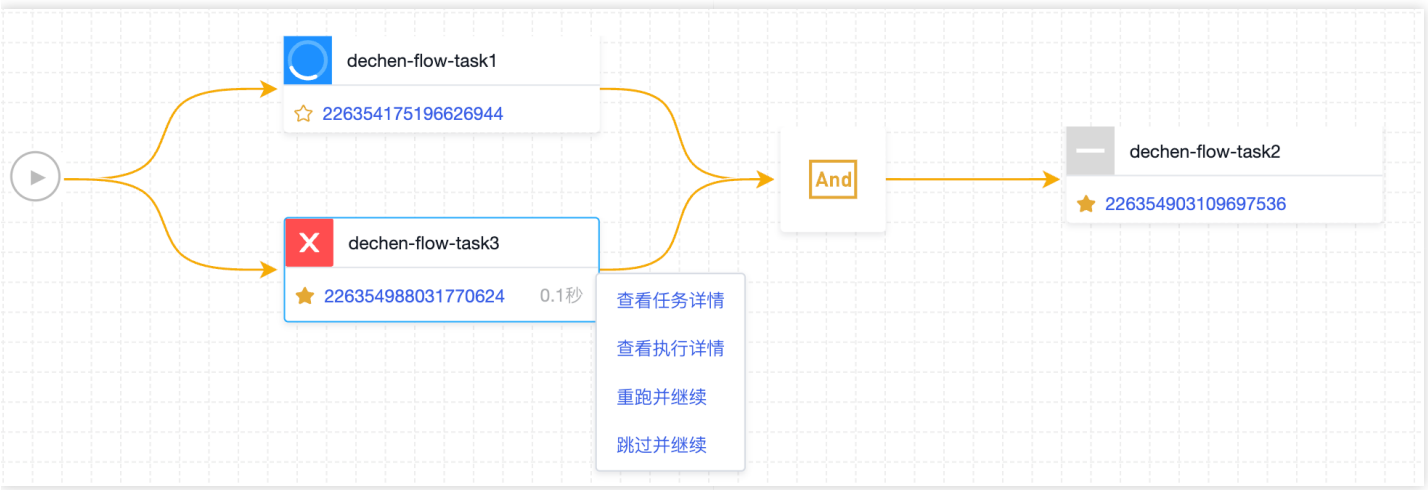
关键是工作流中必须要执行成功的任务，工作流中必须至少有一个关键任务，只有所有关键任务执行成功整个工作流才能算执行成功。例如上图中，只有 task2 执行成功了整个工作流的执行状态才能设置为成功。

触发规则

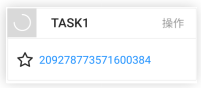
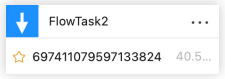
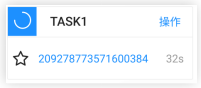
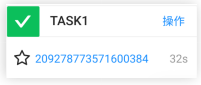
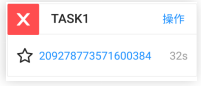
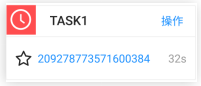
工作流的触发规则以及触发限制和任务里的一致，请参考前面任务的【触发规则】章节。

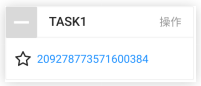
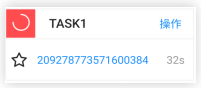
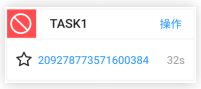
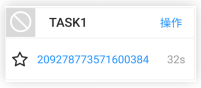
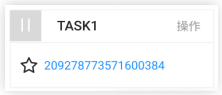
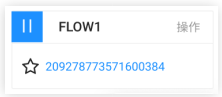
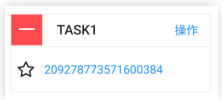
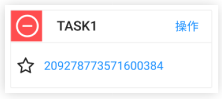
工作流执行

工作流触发后会按照编排好的依赖关系一步步向后执行，执行过程中我们可以查看任意任务节点的执行状态，以及任务的详细信息。



执行过程中，任务节点可能存在以下的状态：

任务执行状态	说明
	等待执行
	下发中
	执行中
	执行成功
	执行失败
	执行超时

	未执行，工作流执行已经结束，那些未被触发的任务，例如未达到触发条件的任务节点。
	终止中，任务被手动终止，进入终止中状态。
	已终止，被手动终止的任务进入已终止状态。
	已跳过，任务失败后手动执行跳过的任务状态。
	挂起，工作流中未开始执行的节点可以被挂起，挂起后工作流执行到该节点时会等待，直到该节点的挂起被解除。
	暂停，只有在节点是子工作流时才支持暂停状态。
	已拒绝，该节点配置了不允许并发执行，当该节点对应的任务或子工作流上一次执行还未结束时，重新触发执行会进入拒绝状态。
	被限流，当系统达到性能瓶颈时，会对优先级为一般的任务进行限流，这样的节点触发后会概率性地被放弃调度执行，直接进入被限流状态。

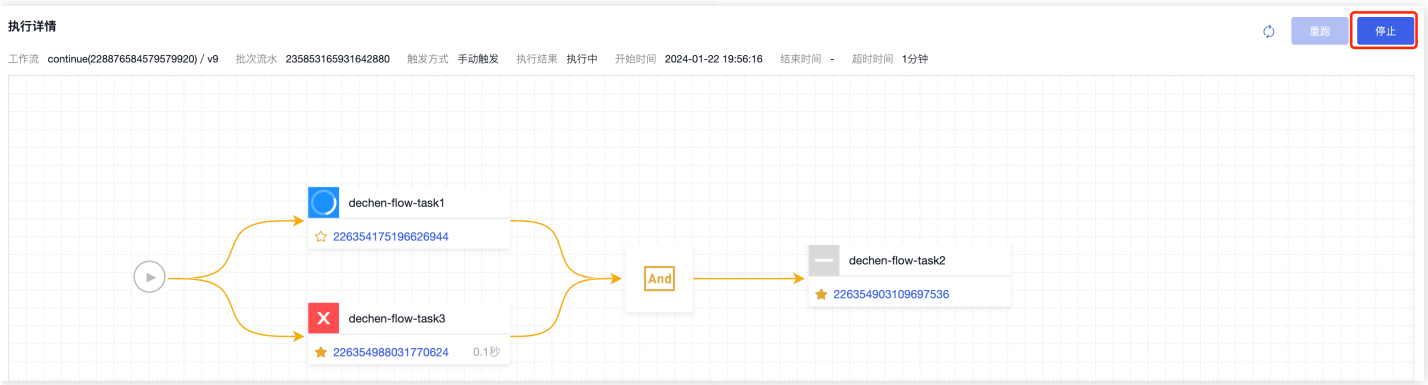
工作流状态判定

状态	说明

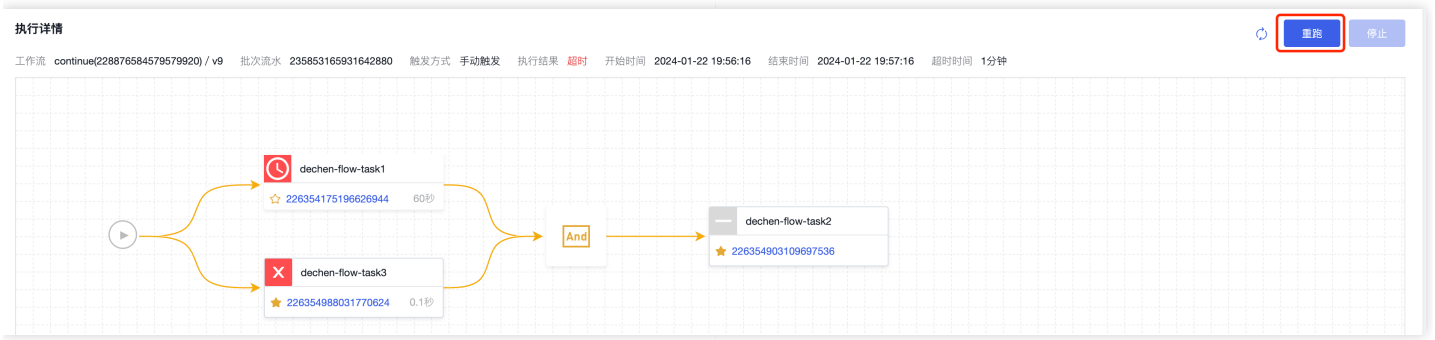
状态	说明
执行中	执行中表示工作流尚未执行结束，可以是有节点尚在执行中，或者有节点被挂起等待，或者有可以触发的节点（触发条件满足）等待被触发执行。
成功	工作流被判定为执行成功，只需要工作流不在执行中，并且工作流中所有核心节点执行成功。
超时	工作流超时，表示整个工作流的执行时间超过了工作流上配置的超时时间，只有这一种情况下工作流才会被判定为超时状态。以下情况都不会判定为超时： - 工作流只有一个任务节点，节点执行超时（超过任务自己的超时配置）进入超时状态，但是整个工作流执行没有超过工作流的超时时间。
已终止	只有手动停止整个工作流的执行，工作流才有可能进入已终止状态。即使工作流节点被单独停止进入已终止状态，整个工作流也不会进入已停止状态。
已拒绝	如果工作流不允许并发执行，工作流上一次执行尚未结束，新的触发会进入已拒绝状态。注意工作流节点因为并发限制进入了已拒绝状态，并不会导致整个工作流进入已拒绝状态，工作流进入已拒绝一定是工作流自身的并发控制。
失败	工作流执行结束，并且有核心节点没有执行成功，则工作流进入失败状态，这里并不需要去关心核心节点为什么没有执行成功（是依赖条件未满足没有触发，还是执行超时，亦或者被手动终止）。

重跑/停止

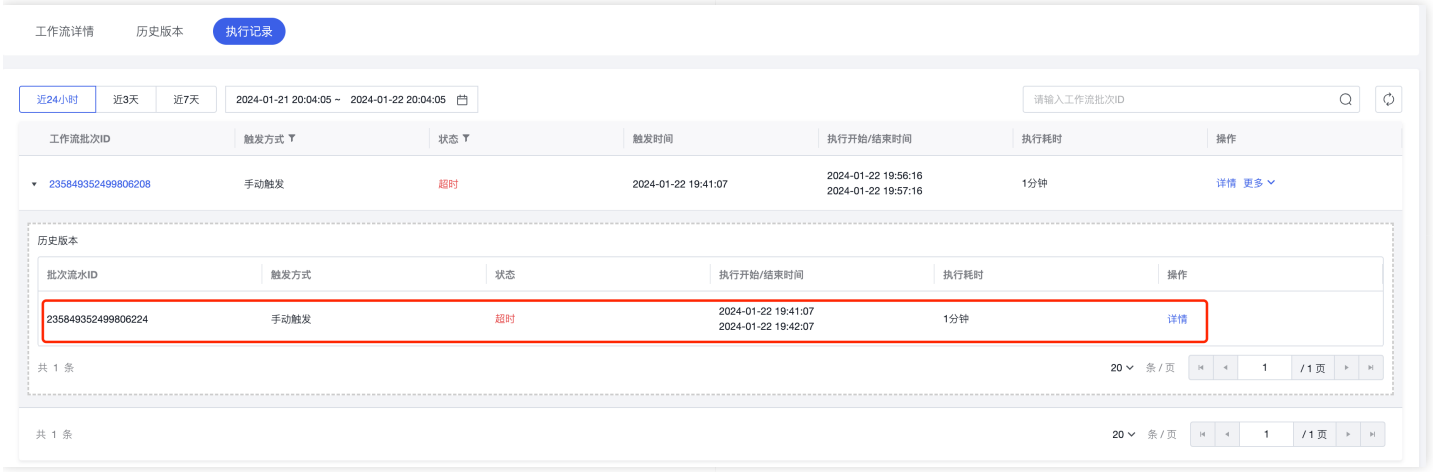
如果一个工作流处于执行中，我们可以在控制台上停止它，工作流停止后工作流批次状态会进入停止中状态，并且通知所有执行中任务节点执行停止操作，任务节点的停止请参照前文中任务的停止章节。所有任务都停止完成时（进入已终止状态）工作流批次状态进入已终止状态。



对于一个执行结束（成功、失败、超时、已终止都属于结束的状态）的工作流批次，我们可以在控制台重跑它，重跑时会采用和该任务批次（即执行记录）当时触发时相同的工作流详情版本进行触发。



触发重跑后会在该工作流批次下生成一个新的工作流批次流水，如下所示:



强制执行

工作流执行时，节点的触发要等到前置依赖的节点执行成功才能触发，如果节点是任务节点并且配置了允许触发的时间（如 20:00 之后）还需要等待触发时间到达才能触发。为了应对一些特殊的情况，TCT 也支持在工作流中强制一个节点开始执行，而忽略它的触发条件（如前置依赖、允许触发时间）。节点强制执行后同样会向后正常触发它后续的节点（但需要触发条件满足）。

状态标记

TCT 允许直接在工作流中将一个非终态（包括等待中、下发中、已挂起、执行中、暂停中、已暂停、终止中）的任务节点状态标记为某一个终态（如成功、失败、已终止、超时、未知），状态标记后会相应触发工作流中的后续节点。状态标记仅仅是在 TCT 中将任务标记为指定的状态，并不会去终止执行器上的任务（如果还在执行）。

> 注意：
>
> 标记任务批次的状态实际上是批量标记任务批次下的子任务的状态，而整个任务批次的状态还是通过子任务状态统计计算得到，因此会存在标记任务批次为某一个状态但是最终任务批次却是另一个状态的情况。例如对于个分片任务，假设有 100 个分片，我们设置任务成功判定条件为分片成功率大于 90%，假设这时候已经有 90 个分片执行成功，剩余 10 个还在执行中，我们尝试将它标记为超时，最终 10 个执行中的任务会是超时状态，但是任务批次整体的状态会是成功状态。

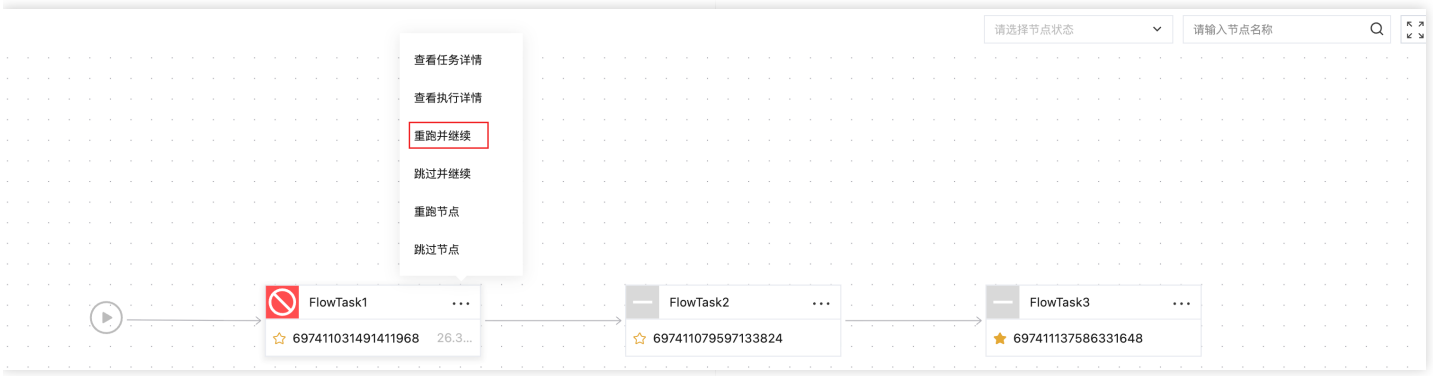
任务跳过/重试

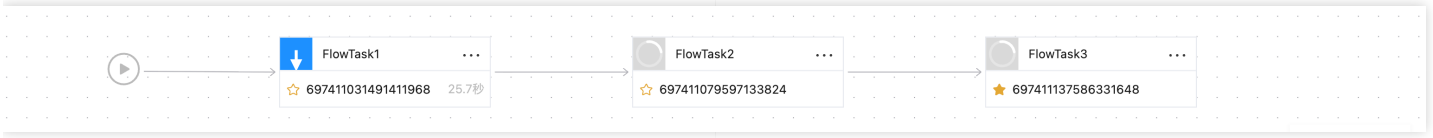
如果一个工作流已经执行结束（成功、失败、超时、已终止都属于结束的状态），对于其中失败、已终止、超时、未知的任务，我们可以在控制台上执行重跑并继续当前工作流的执行，或者执行跳过并继续当前的工作流执行。

重跑并继续

执行重跑并继续，当前的任务节点会被重新执行，同时依赖它的所有后续任务节点也会重新进入等待执行的状态。注意重跑并继续会对节点产生新的批次流水记录，但是整个工作流批次流水记录还是前的，这个和整个工作流批次的重跑不同。

> 警告：
>
> 重跑并继续表示的是从当前节点开始重跑后续的整个工作流分支，该操作会导致该节点后续已经执行过的节点（例如通过强制执行已经执行过的节点）重新执行，如果这个不符合业务场景需要请慎重使用该操作。如果想要仅仅重跑当前节点，而后续已经执行过的节点不再重新执行，请使用重跑节点功能。

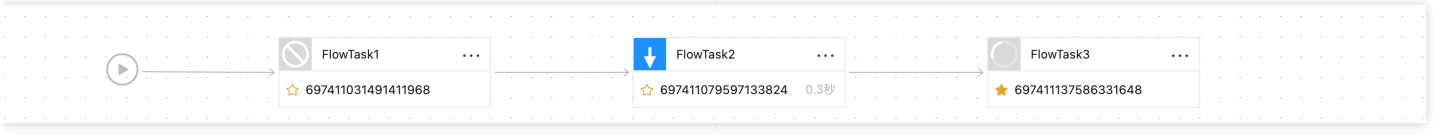




跳过并继续

执行跳过并继续，会将当前的任务节点直接跳过（节点的运行状态被置为已跳过）并继续后续的任务，跳过的任务节点相当于执行成功，即使该节点是关键节点也不影响工作流最终的结果判断。

> 警告：
>
> 跳过并继续表示的是跳过当前节点并从当前节点开始重跑后续的整个工作流分支，该操作会导致该节点后续已经执行过的节点（例如通过强制执行已经执行过的节点）重新执行，如果这个不符合业务场景需要请慎重使用该操作。如果想要仅仅跳过当前节点，而后续已经执行过的节点不再重新执行，请使用重跑节点功能。



重跑节点

重跑节点仅仅重跑当前节点，而不会重跑该节点后续已经跑过的节点，该节点执行完成后会正常触发后续还没有执行过的节点。

跳过节点

跳过节点仅仅跳过当前节点，而不会重跑该节点后续已经跑过的节点，跳过该节点相当于该节点执行成功，会正常触发后续还没有执行过的节点。

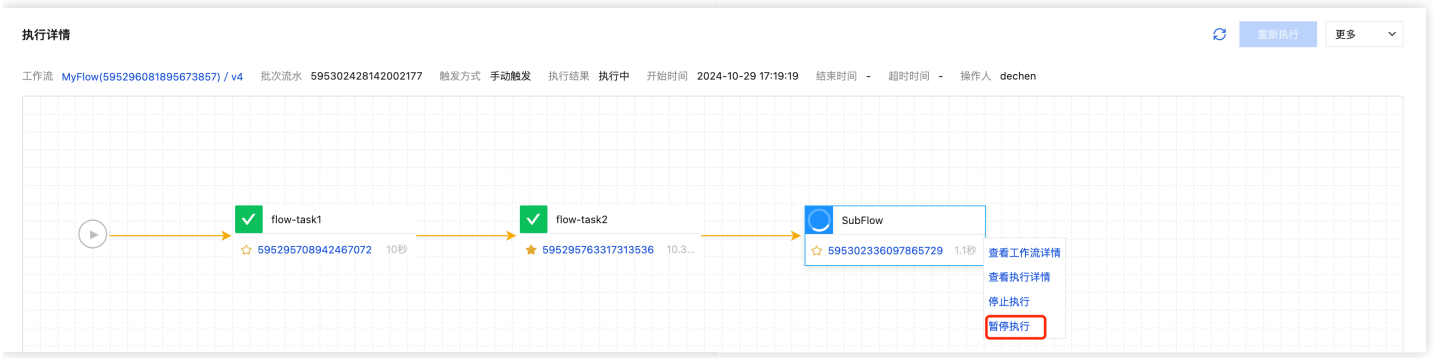
暂停/继续

如果工作流处于执行中，我们可以将工作流暂停，注意暂停时如果一个任务节点已经开始执行，那么该任务节点会继续执行，待已经开始执行的任务节点执行完成后整个工作流将停止执行。

下图中，flow-task1 已经开始执行，此时如果暂停执行，工作流将会在 flow-task1 执行结束后进入等待状态。

对于暂停的工作流，可以继续执行。

如果某一个节点是工作流节点，那么我们也可以对执行中的该节点进行暂停、继续操作。



挂起/解除挂起

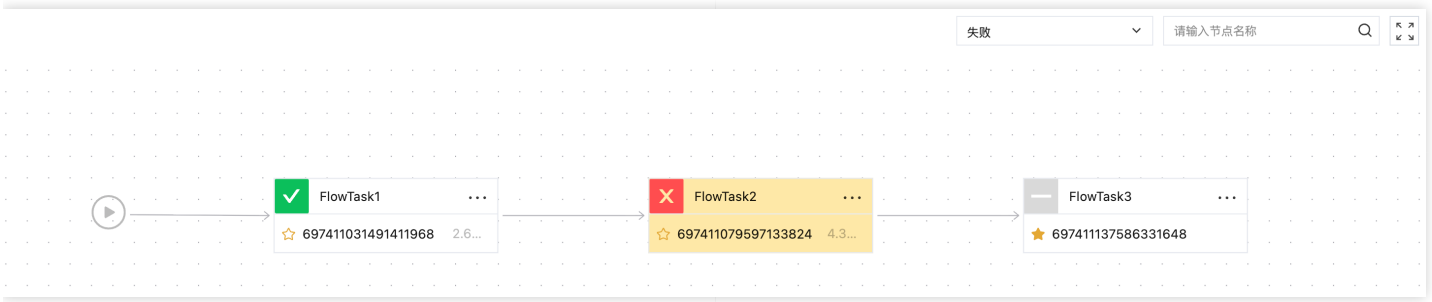
对于工作流中一个还未开始执行的节点，我们可以选择将其挂起，挂起后工作流执行到该节点时会停在该节点等待，直到手动解除挂起后该节点才会开始执行。

工作流删除

工作流删除和任务删除类似，删除有工作流进入删除中状态，删除中的工作流在 TCT 控制台上不可见。删除后工作流会被 GC（Garbage Collector，垃圾回收器）回收，和该工作流相关的历史版本，执行批次（包括流水），任务节点的执行批次（流水）等都会被回收。注意工作流中的任务节点不会被回收。

工作流节点搜索

在工作流执行记录中，我们可以通过节点状态或者节点名称来搜索图中的节点，搜索命中的节点会高亮显示并且自动缩放和移动画布来展示这些节点。状态搜索支持多选，节点名称搜索支持模糊匹配。


失败
请输入节点名称
FlowTask1
697411031491411968
2.6...
FlowTask2
697411079597133824
4.3...
FlowTask3
697411137586331648

认证鉴权

基于部署组的权限管理

TCT 中对资源（部署组、任务、工作流、执行记录）的权限管理是基于部署组的。TCT 中的任务、工作流、执行记录等资源的权限直接继承对应部署组的权限。也就是说用户对部署组 A 有权限，那么他对部署组 A 上创建的任务、工作流、任务的执行记录、工作流的执行记录也有权限。

对于工作流的情况稍微复杂一些，一个工作流可能会涉及多个部署组（例如工作流节点 A 使用部署组 A，节点 B 使用部署组 B），用户只有对所有涉及的部署组都有权限才对该工作流有权限。

TCT 中的权限分为以下三种：

- r：可读，如果用户对部署组有可读权限，那么他拥有该部署组对应任务、执行记录等资源的查看权限，但是没有操作权限。
- w：可写或者全读写权限，是三个权限中最大的权限，他包含了所有操作的权限。如果用户对部署组有全读写权限，那么它对部署组或者部署组相关的任务有所有操作的权限，包括执行任务、删除/编辑任务。
- x：可执行权限，拥有该权限的用户对任务或者工作流有执行的权限，这里的执行包括执行相关的所有操作，例如启用/停用任务、执行一次任务、重跑、挂起、暂停工作流等操作。和全读写权限（w）相比，它只少了任务/工作流的编辑、删除、设置当前版本，执行看板的创建、编辑、删除操作。

系统级别权限管理

对应于 TCT 中的三种权限，TCT 定义了三种系统级别权限策略。用户如果绑定了这些权限策略，那么就拥有了租户下所有资源的该权限，例如用户绑定 TCTExecuteAccess 权限策略，那么用户可以执行租户下所有的任务、工作流。

- TCTReadOnlyAccess： TCT 系统级别权限策略，拥有 TCT 的只读权限（r），可以查看 TCT 里所有的部署组、任务、工作流、执行看板等，但是没有操作的权限（创建、删除、编辑、运行等）。
- TCTExecuteAccess：TCT 系统级别权限策略，该权限允许用户执行任务/工作流，包括暂停、继续、终止、强制终止、重跑、挂起、解除挂起、跳过等操作。
- TCTFullAccess：TCT 系统级别权限策略，拥有 TCT 的全部操作权限， 能够进行增删改查以及执行相关的所有操作。

系统级别权限策略可以直接在【访问管理】中的用户下进行绑定。

资源级别权限管理

如果希望对 TCT 中的资源进行细粒度的权限控制，例如对任务 A 有执行权限、对任务 B 有只读权限，那么就需要进行资源级别的权限管理。

TCT 中我们通过部署组来实现权限控制，部署组资源的权限是通过项目来实现的，TCT 的部署组作为资源加入到项目中，用户绑定项目级别权限策略加入到项目中，从而拥有项目中对应资源的权限。

权限策略

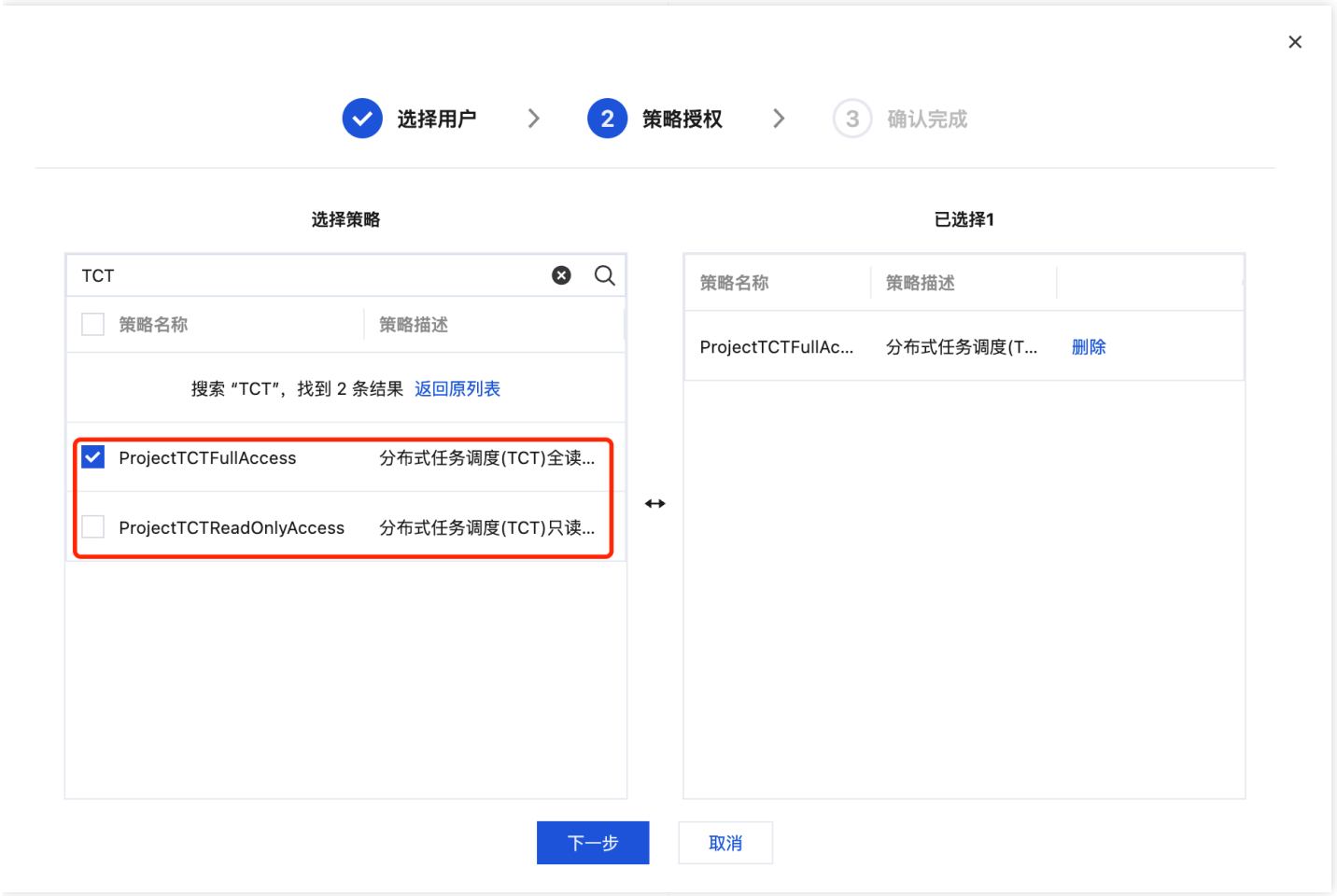
TCT 中项目级别权限策略同样对应于 r、x、w 三个级别：

- ProjectTCTReadOnlyAccess：TCT 项目级别权限策略，绑定该策略的用户拥有对应项目下所有 TCT 部署组的只读权限（r），从而拥有部署组对应任务、工作流、执行记录的只读权限。
- ProjectTCTExecuteAccess：TCT 项目级别权限策略，绑定该策略的用户拥有对应项目下所有 TCT 部署组的可执行权限（x），从而拥有部署组对应任务、工作流、执行记录的可执行权限。
- ProjectTCTFullAccess：TCT 项目级别权限策略，绑定该策略的用户拥有对应项目下所有 TCT 部署组的全读写权限（w），从而拥有部署组对应任务、工作流、执行记录的全读写权限。

如何绑定权限

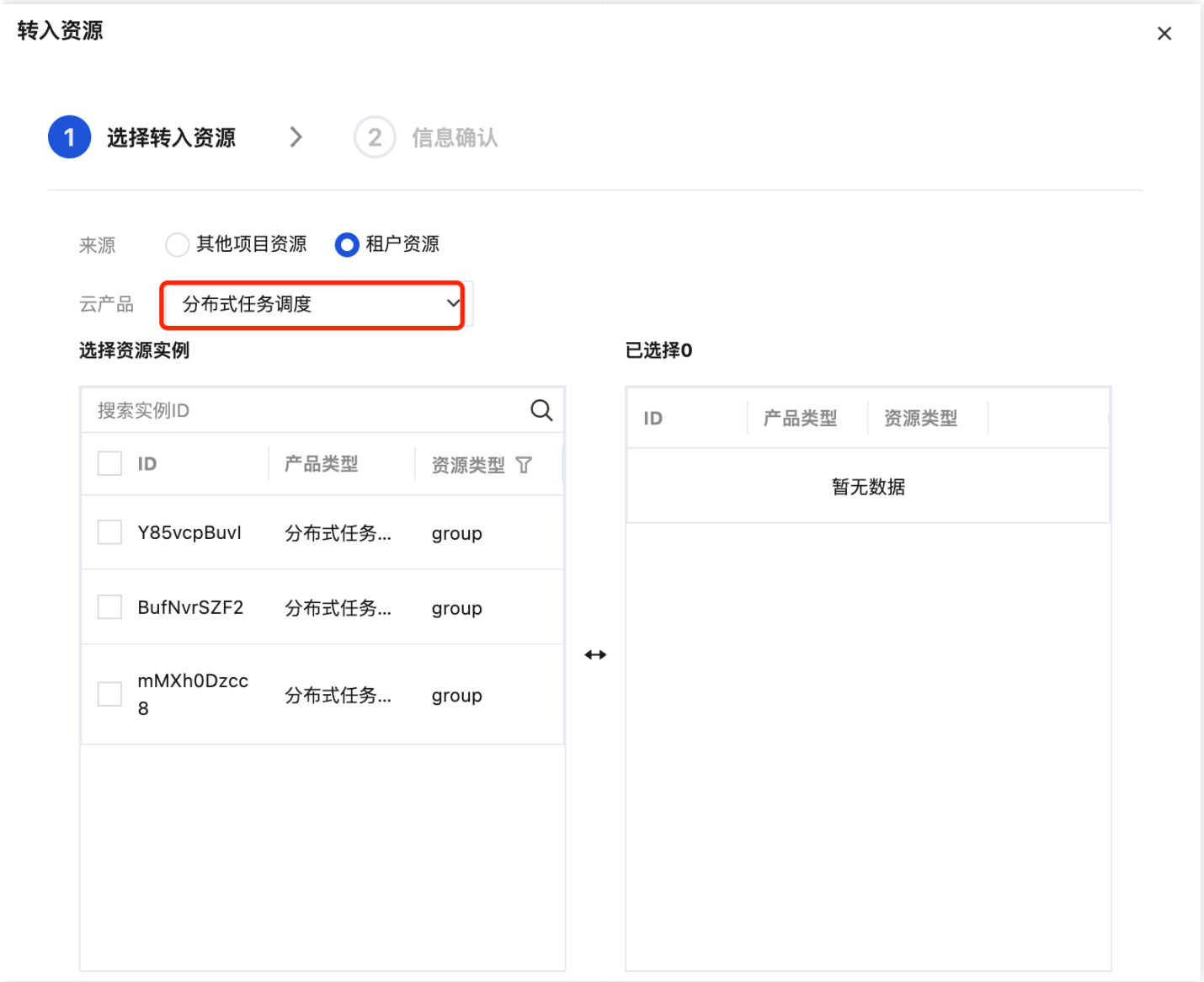
接下来我们看看如何通过项目来实现资源级别的权限控制。

首先我们可以在 云平台的 【资源管理】中创建一个项目。
创建项目后，我们需要将用户添加到项目中，并为之授权，例如用户以 ProjectTCTFullAccess 权限策略加入到项目中。



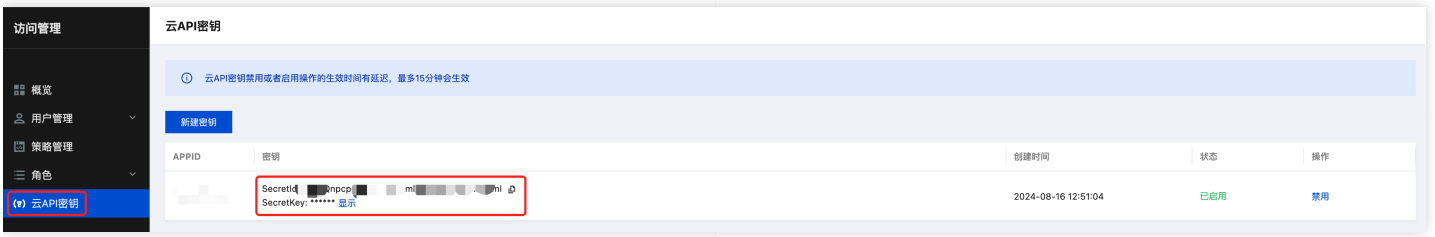
在 TCT 中创建部署组时，我们选择这个项目，这样该部署组会作为资源归属于该项目，该项目中的成员则会拥有该部署组对应的权限。

对于存量的部署组，我们也可以在【资源管理】中直接将部署组作为资源转入到项目中，如下图所示。



执行器注册认证鉴权

执行器（客户端实例）通过 SDK 注册到 TCT 上，接收 TCT 的任务调度，执行器实例在注册的时候是通过 AK/SK 来进行认证鉴权的。AK/SK 可以在云平台的账号信息中获取。



TCT 收到客户端发来的注册请求后，会先进行认证，即校验 AK/SK 的有效性（并同时获取对应的用户信息），认证通过后再进行鉴权，验证用户对注册的目标部署组是否有写权限（w），如果验证通过则注册成功。这里需要注意的是，由于 TSF 未接入平台项目管理，因此我们对 TSF 类型的部署组只做认证而不做鉴权，如果希望进行鉴权，请使用 TCT 部署组（通过在客户端配置中配置 TCT 部署组 ID 实现，详情请参考开发指南）。

版本管理

版本管理

TCT 中任务和工作流都支持版本管理，每次编辑时都会产生一个新的版本，在 TCT 工作台上可以将任务和工作流回滚到任意的一个版本上。

版本号

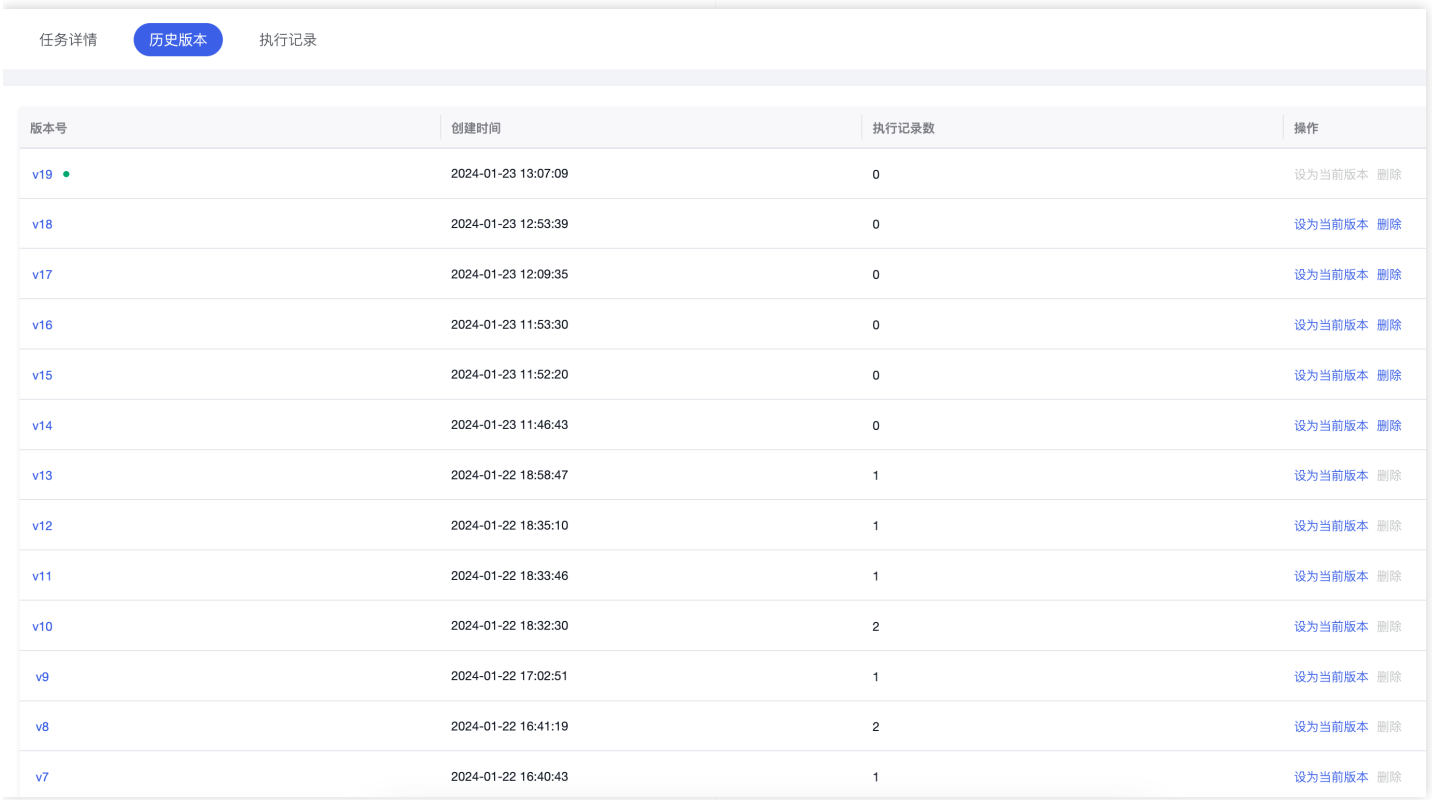
TCT 中采用 v1, v2, v3 ... 递增的整数型版本，任务或工作流的每一次改动都会递增一下版本号，注意版本号不可复用，即使前面的某个版本被删除了，后续也不会重新使用这个版本号。
对于任务，还有一个继承自之前版本的 TaskLogId（任务流水ID） 的概念，它其实也是一个版本号的概念，数据类型是 UUID，不是用户可读的，可以将它理解为版本 ID，而 v1, v2 这种是用户友好的版本代号，从 TCT 2.1.0 版本开始直接使用版本号而不再使用 TaskLogId，实际上内部实现中还是用到了 TaskLogId。

当前版本

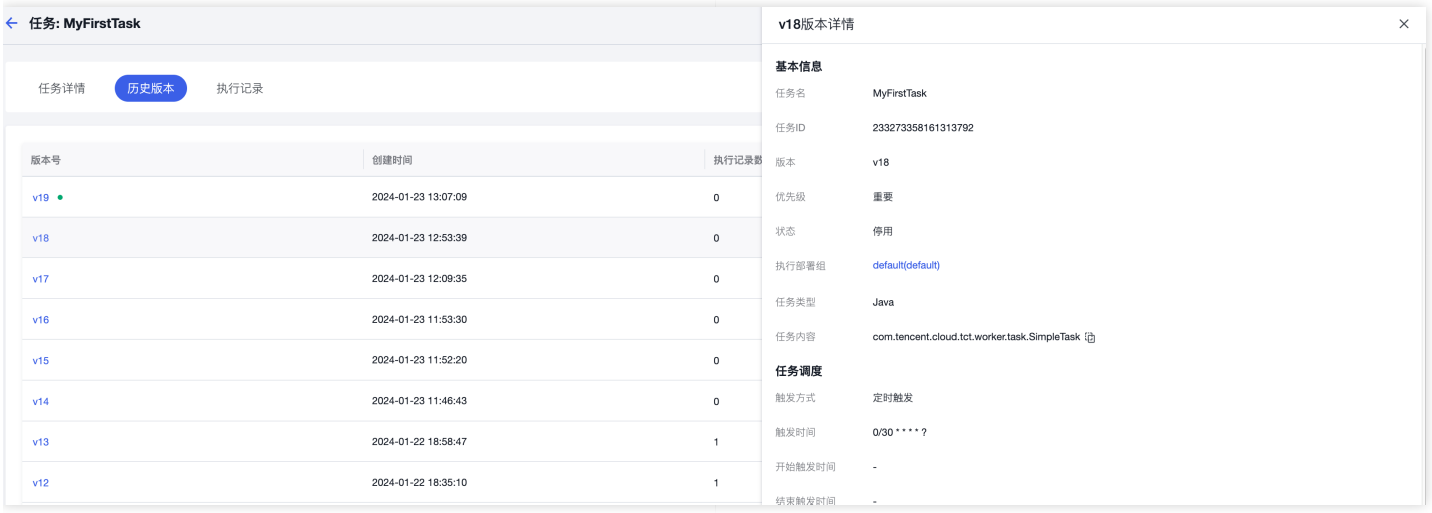
当前版本为任务或工作流当前在使用的版本，TCT 触发任务都是使用的任务或工作流当前的版本。用户每次在控制台编辑任务或者工作流时，新产生的版本都会默认被设置成当前版本。

历史版本

TCT 中会保留任务或者工作流所有的版本，并可以随时将某个历史版本切换为当前版本进行使用。历史版本也可以删除，但是需要满足两个条件：a. 不能是当前版本，b. 不能有执行记录。



在历史版本管理中我们也可以随时查看各个历史版本的详细信息：



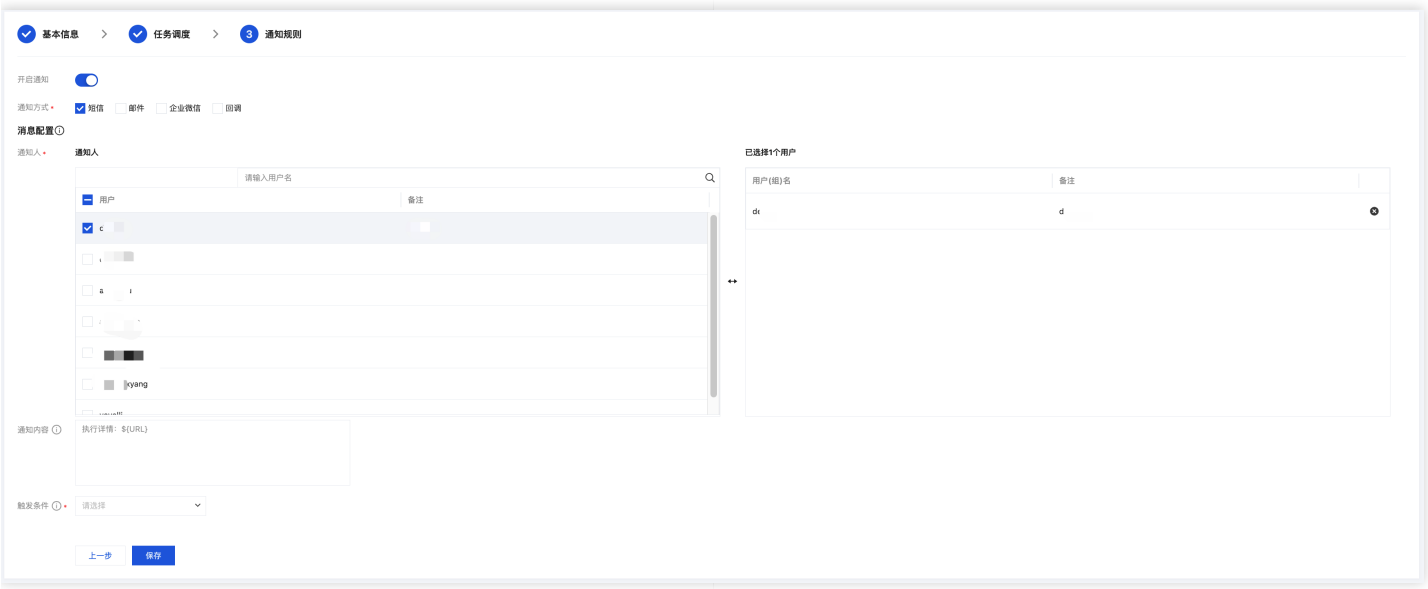
通知订阅

在创建任务或者工作流时我们都可以配置消息通知，目前支持短信、邮件、企业微信、回调的方式发送通知。通知人是云平台上的注册用户（也可以选择用户组），用户注册时已经提供了手机号和邮箱地址。触发通知的条件可以是任务或工作流执行成功进行通知，也可以是执行失败、超时等状态进行通知，支持多选。通知的内容可以由用户自定义，一般直接采用默认的通知内容模板即可。

通知方式

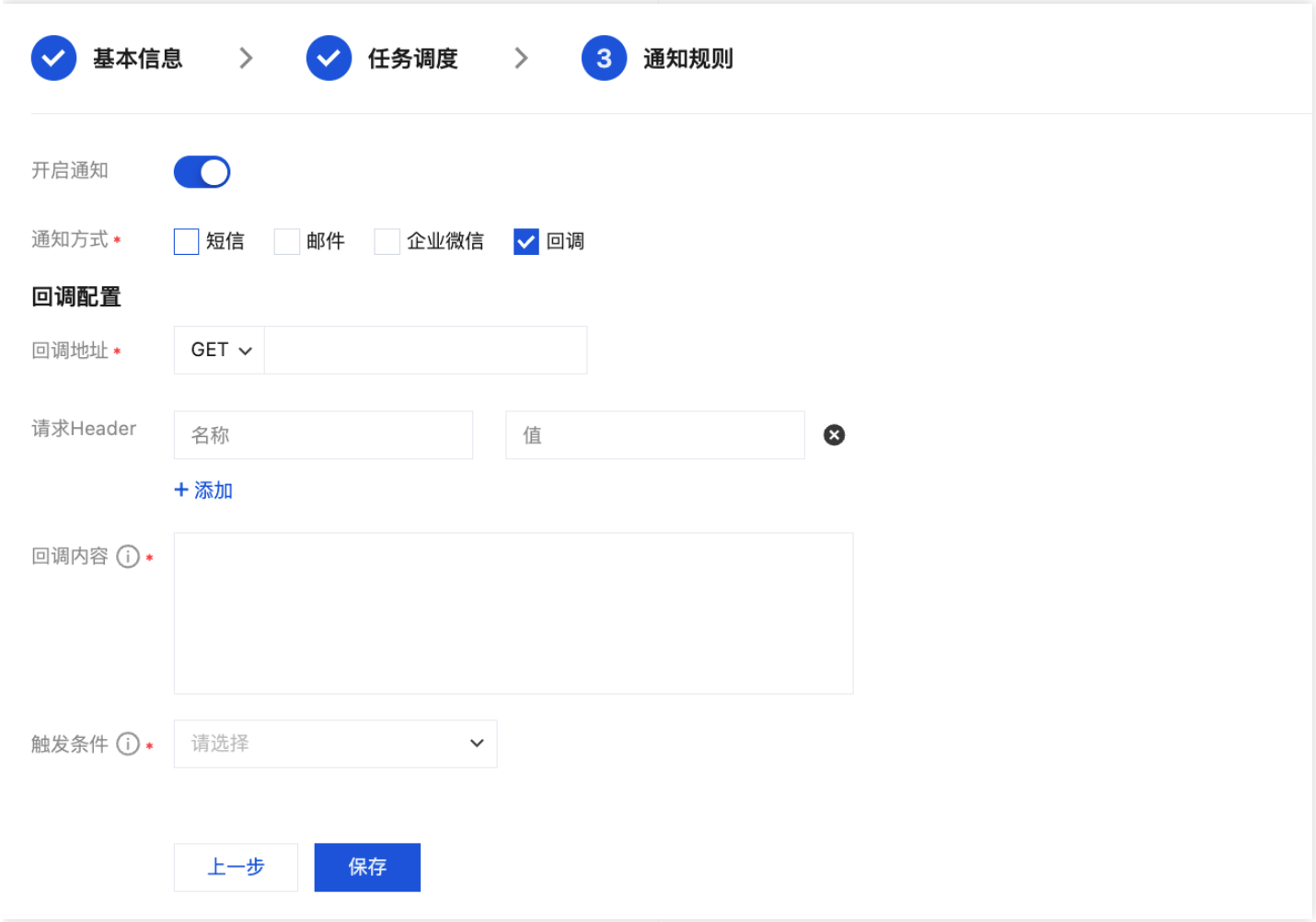
短信/邮件/企业微信

这三种通知方式使用的是云平台的消息管理模块，TCT 调用消息管理的 API 将要发送的消息推送到消息管理，由消息管理将消息投递出去。通知发送的情况可以在运营端的【客户消息管理】中查看。



回调方式

TCT 支持回调方式发送通知，用户可以提供一个回调地址用于接收消息，详细配置如下，其中的"回调内容"和短信等通知方式里的"通知内容"内容一样支持模版参数（详见下文的【通知内容模版】）。



通知内容

通知发送的的内容，用户可以自行编写，TCT 提供一系列的模版参数可以使用。以任务为例，默认的通知内容如下：

执行详情：${URL}

模版参数 ${URL} 会被替换为实际的执行详情地址，当前支持的模版参数如下表所示：

任务相关模版参数

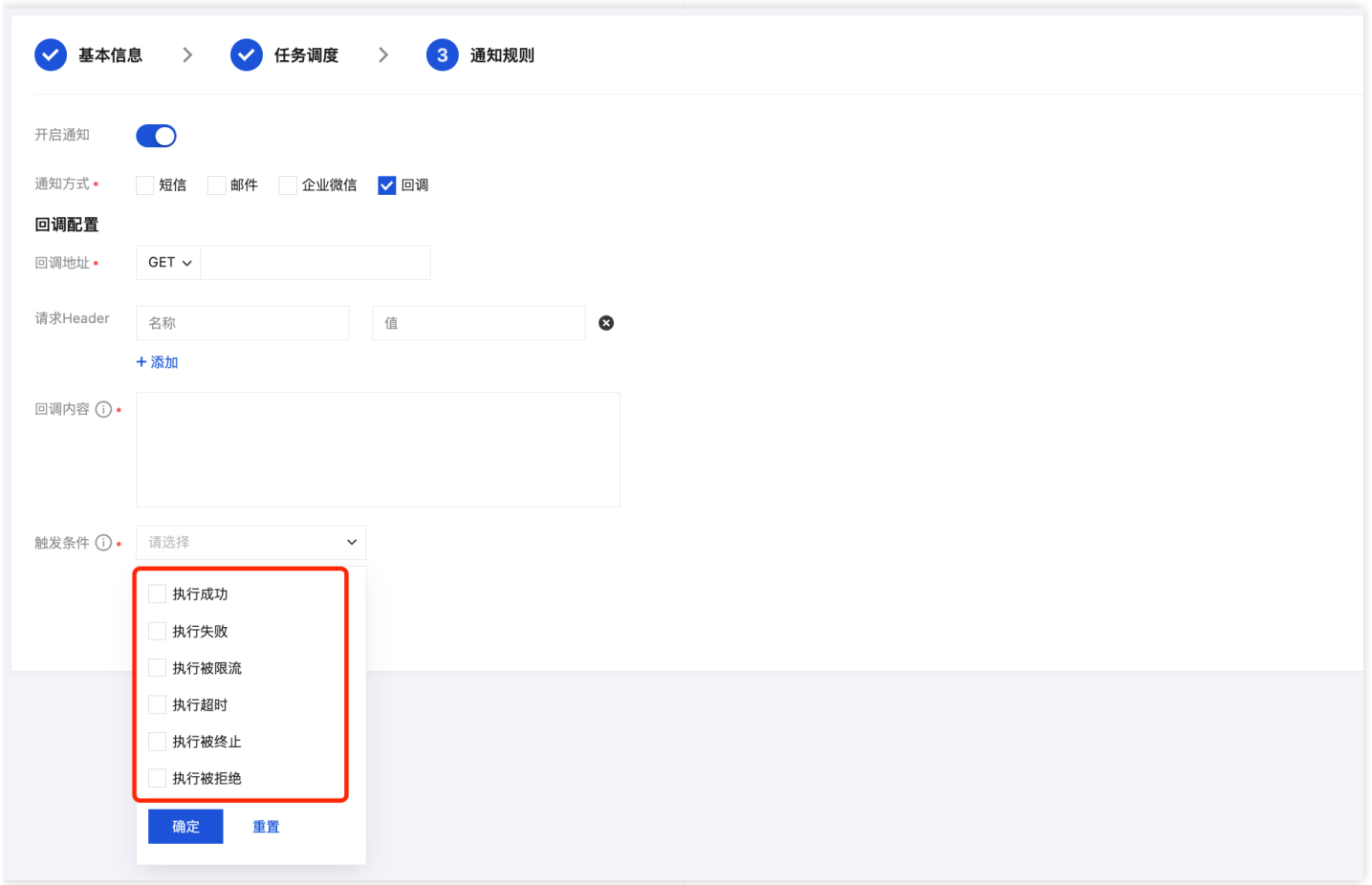
参数名称	说明
ID	任务ID
VERSION	版本
NAME	任务名称
STATE	执行状态
URL	执行详情链接
START_TIME	开始执行时间，格式形如 2024-03-07T11:11:37.931
END_TIME	结束执行时间，格式形如 2024-03-07T11:11:37.931
TASK_TYPE	任务类型，枚举值：Java,External,Shell,Python
BATCH_TYPE	批次类型，枚举值： N-普通任务, F-工作流任务
PRIORITY	任务优先级，枚举值： 1-NORMAL, 2-IMPORTANT;
TRIGGER_TYPE	任务触发方式，枚举值： FixRate-周期触发, WorkFlow-工作流触发， Cron-定时触发， Static-指定时间触发
EXECUTE_TYPE	执行方式，枚举值： UNICAST-随机单节点, BROADCAST-广播, SHARD-分片
BATCH_ID	批次ID
BATCH_LOG_ID	批次历史ID
BELONG_FLOW_BATCH_ID	所属工作流批次ID 注：仅当该任务归属某一个工作流才会渲染该变量
BELONG_FLOW_BATCH_LOG_ID	所属工作流批次历史ID 注：仅当该任务归属某一个工作流才会渲染该变量

工作流相关模版参数

变量名称	含义
ID	工作流ID
VERSION	版本
NAME	工作流名称
STATE	执行状态
URL	执行详情链接
START_TIME	开始执行时间，格式形如 2024-03-07T11:11:37.931
END_TIME	结束执行时间，格式形如 2024-03-07T11:11:37.931
TRIGGER_TYPE	工作流触发方式，枚举值： FixRate-周期触发, Cron-定时触发， Static-指定时间触发
BATCH_ID	批次ID
BATCH_LOG_ID	批次历史ID

触发条件

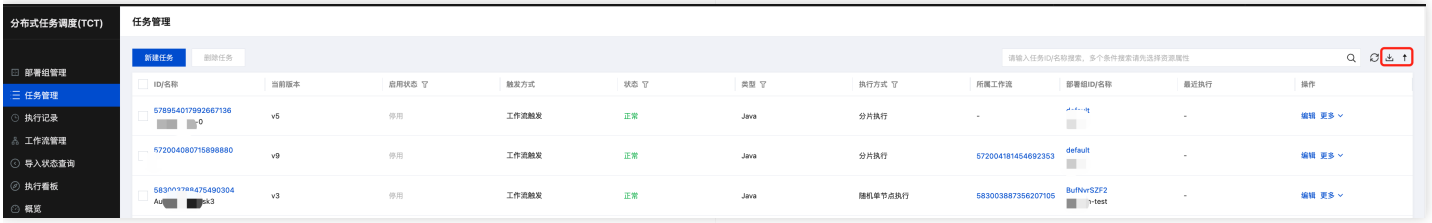
触发通知的条件包括任务或工作流执行成功、执行失败等，允许多选。


基本信息
任务调度
3
通知规则
开启通知
通知方式 *
短信
邮件
企业微信
回调
回调配置
回调地址 *
GET
请求Header
名称
值
+ 添加
回调内容 *
触发条件 *
请选择
执行成功
执行失败
执行被限流
执行超时
执行被终止
执行被拒绝
确定
重置

导入导出

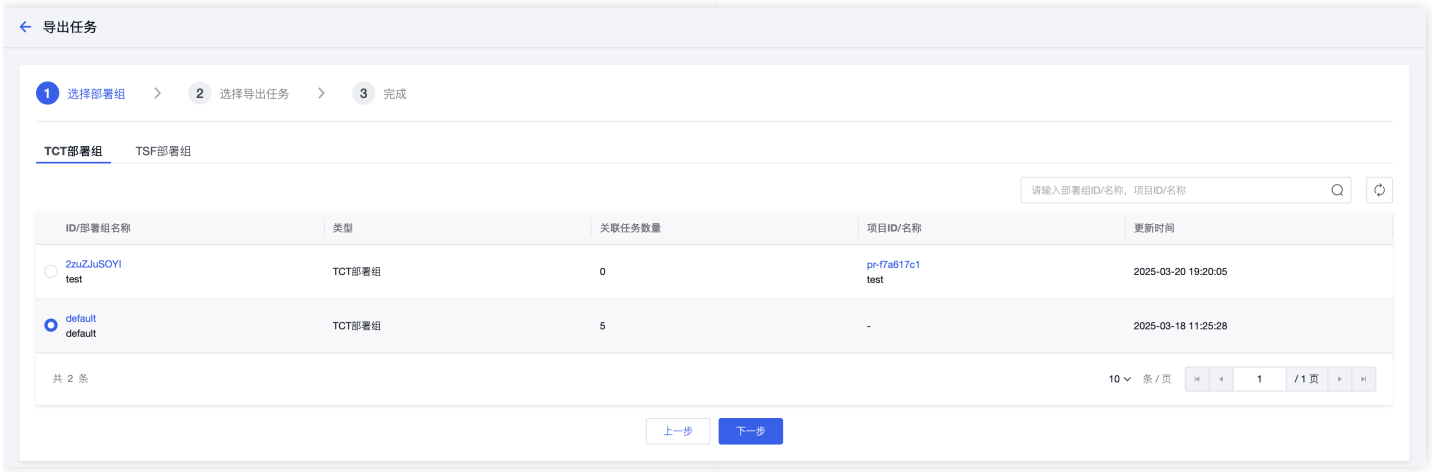
任务导入导出

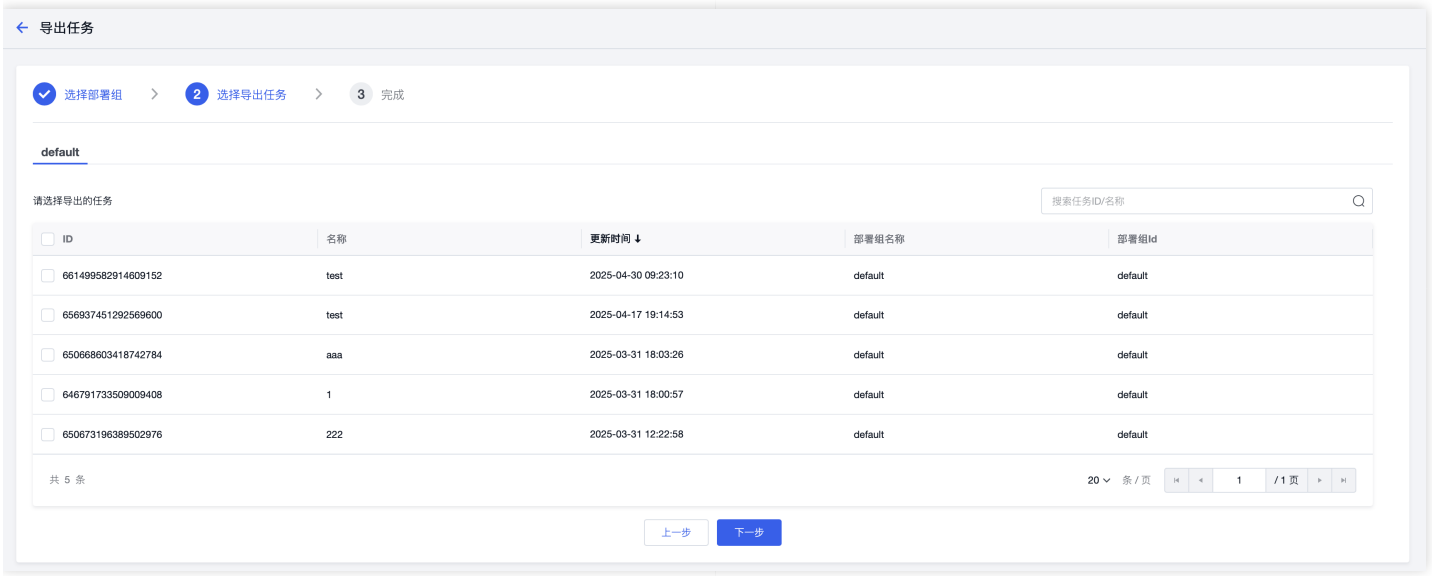
在任务管理页面，可以通过右上角的任务导入导出按钮进行任务的导入导出。



任务导出

在任务导出页面，依次选择部署组、任务进行导出。任务导出后会以一个 JSON 文件的形式存在，注意任务导出只会导出任务当前的版本，其他历史版本不会导出。





任务导入

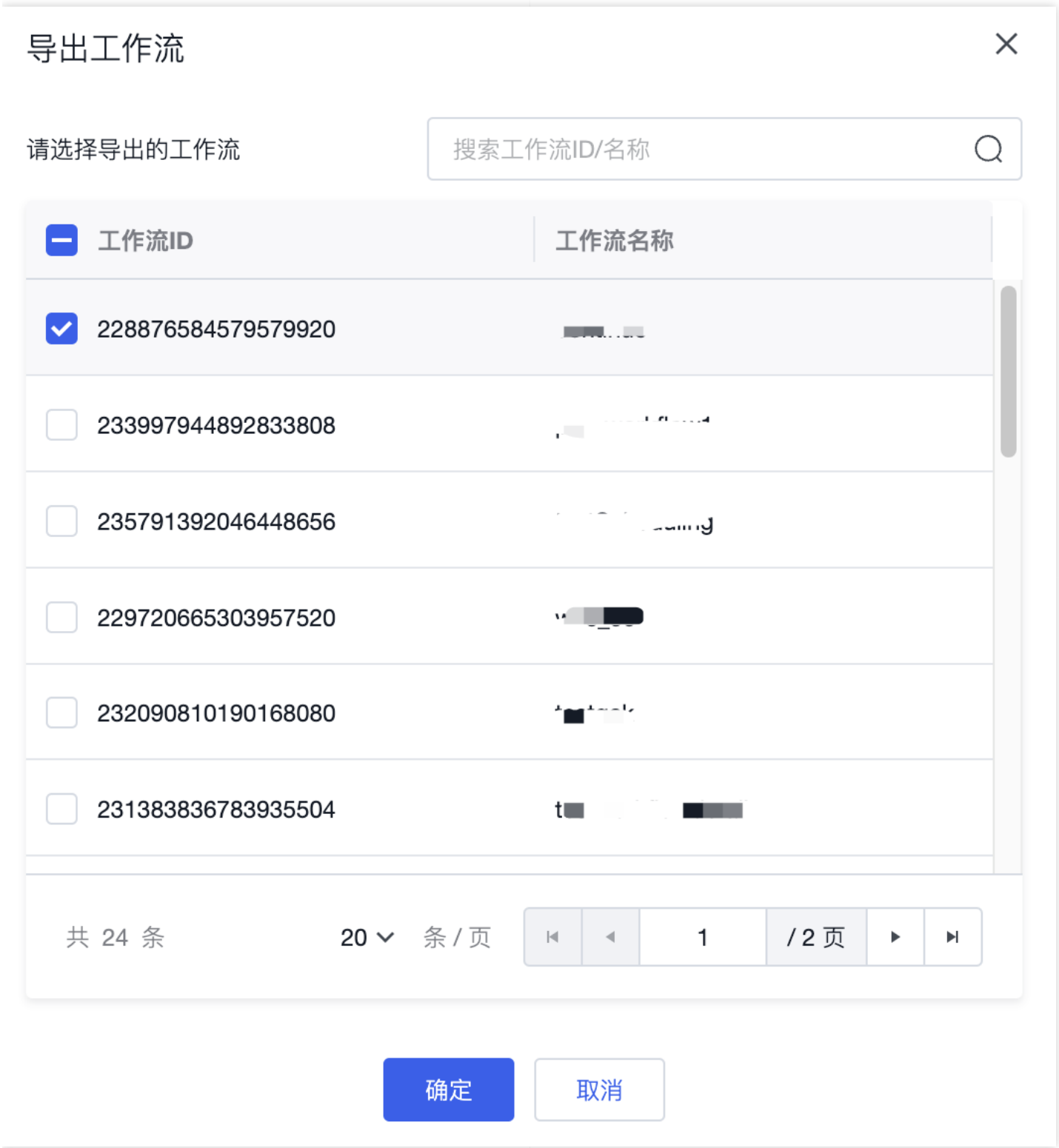
在任务导入页面导入之前导出的任务文件，文件导入后会预览文件中的任务，并且检查任务所属的部署组是否存在（表格的部署组ID列），如果部署组不存在不允许导入。导入时允许配置是否覆盖同名任务，以及是否直接启用导入的任务。

工作流导入导出

在工作流管理页面，可以通过右上角的导入导出按钮进行工作流的导入导出。

工作流导出

在工作流导出页面，选择需要导出的工作流，工作流导出后会以一个 JSON 文件形式存在。注意工作流导出只导出工作流的当前版本，不会导出其他的历史版本，并且工作流中的任务不会自动导出，因此导出工作流前需要先导出任务。



工作流导入

在工作流导入页面，上传之前导出的 JSON 文件，文件上传后会预览文件中的工作流，并且检查工作流涉及的任务是否已经存在（表格的任务列），如果有任务不存在不允许。

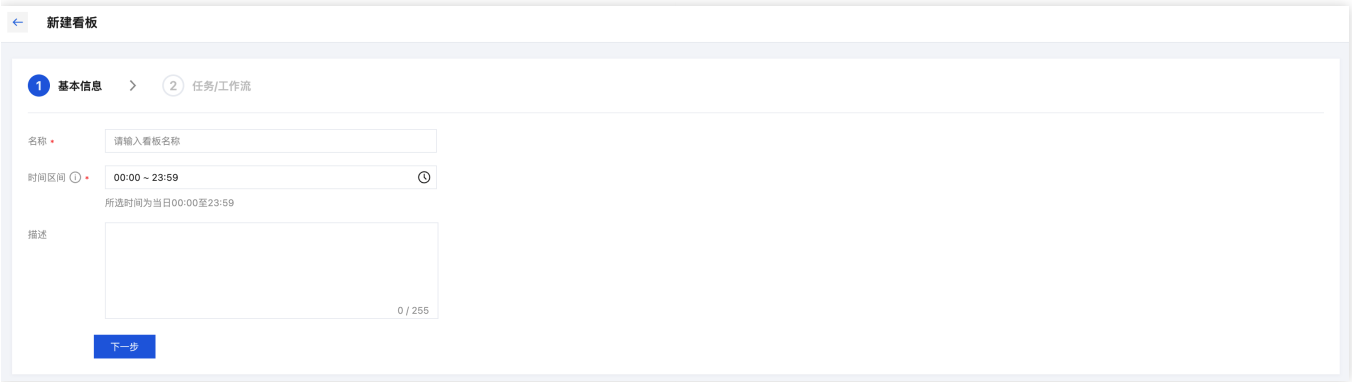

导入工作流
导入文件
tct_workflow_2024-01-23 17_19_19.json
相同工作流
覆盖
跳过
是否启用
是
否
工作流信息
ID
名称
任务
228876584579579920
确定
取消

执行看板

用户可以通过执行看板观察感兴趣的任务和工作流当天的执行情况，只需要创建一个执行看板，并将想要观察的任务和工作流添加到看板中。就可以通过看板观察这些任务和工作流当天的执行情况以及执行计划。

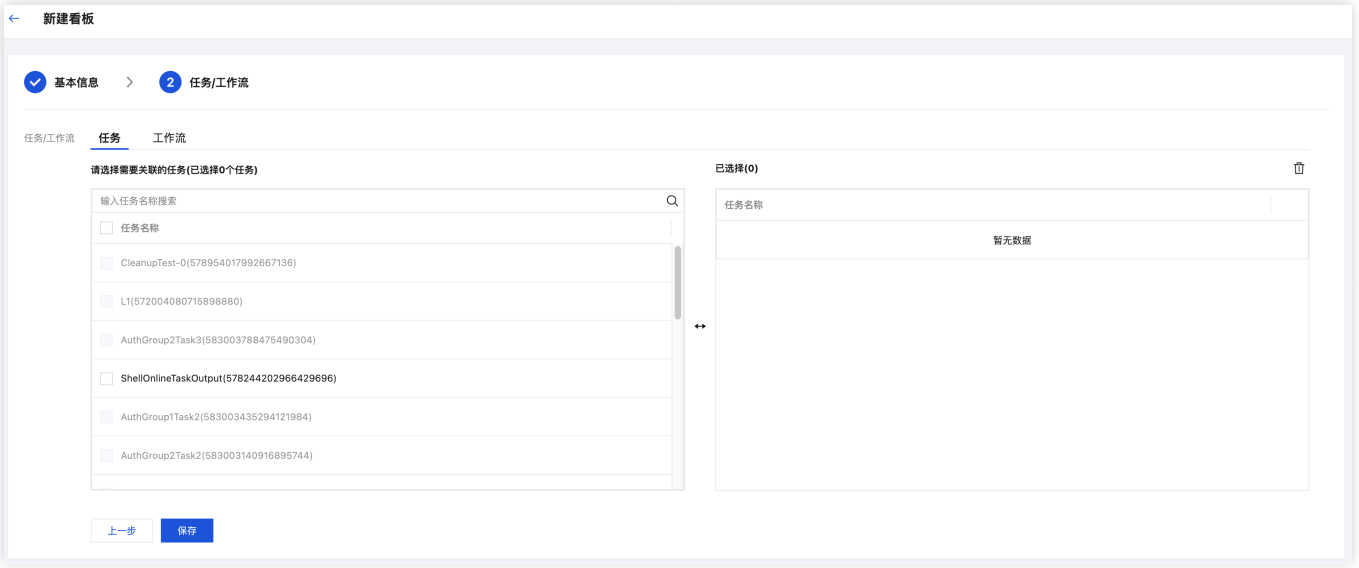
新建看板

新建看板需要指定一个每日观察的时间区间，看板只会观察当天这个时间区间里任务和工作流的执行情况和执行计划。时间区间默认为 00:00-23:59。
这里时间区间也可以配置为跨天的区间，例如 18:00 ~ 06:00 表示的时间为当天的 18:00 到次日的 06:00。



配置好基本信息后，我们可以将期望观察的任务和工作流添加到该看板中，注意能添加到看板中的任务和工作流必须满足以下几个条件：

- 任务和工作流不属于另外的工作流，即他们不是其他工作流的节点。如果作为其他工作流的节点，他们的触发受工作流控制，无法计算他们执行的计划，需要添加他们的父工作流到看板中。
- 任务和工作流触发的时间间隔不能太短，默认不短于 5 分钟。



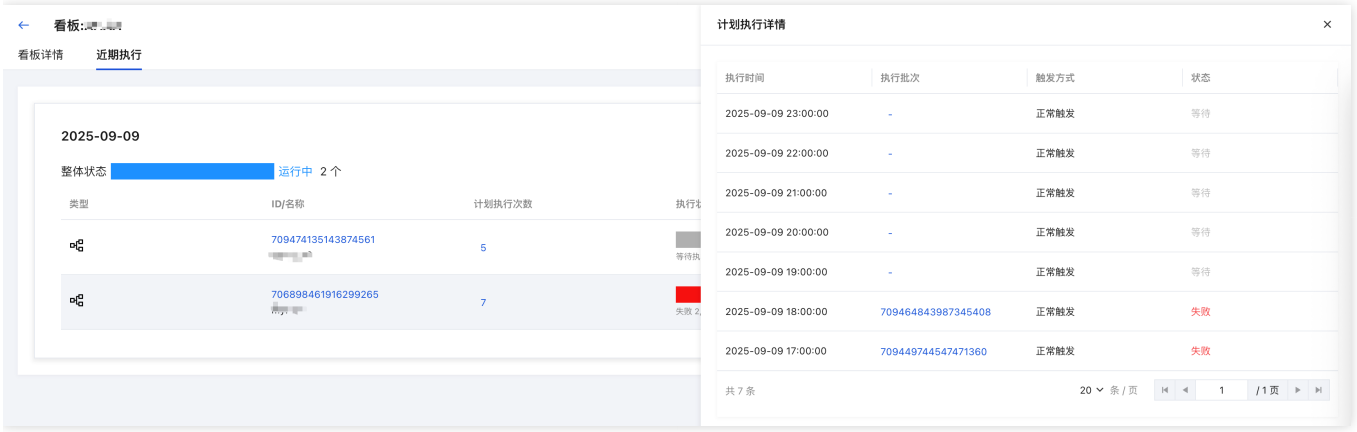
> 注意：只有具有 TCTFullAccess 系统级别权限或者租户管理员权限的用户才有执行看板的写操作（包括创建、修改、删除）。而只有具有 TCTReadOnlyAccess 级别权限以上的用户才能查看执行看板。简单说用户需要对租户下所有资源都有写权限才能操作执行看板，而只有对租户下所有资源都有读权限才能查看执行看板。

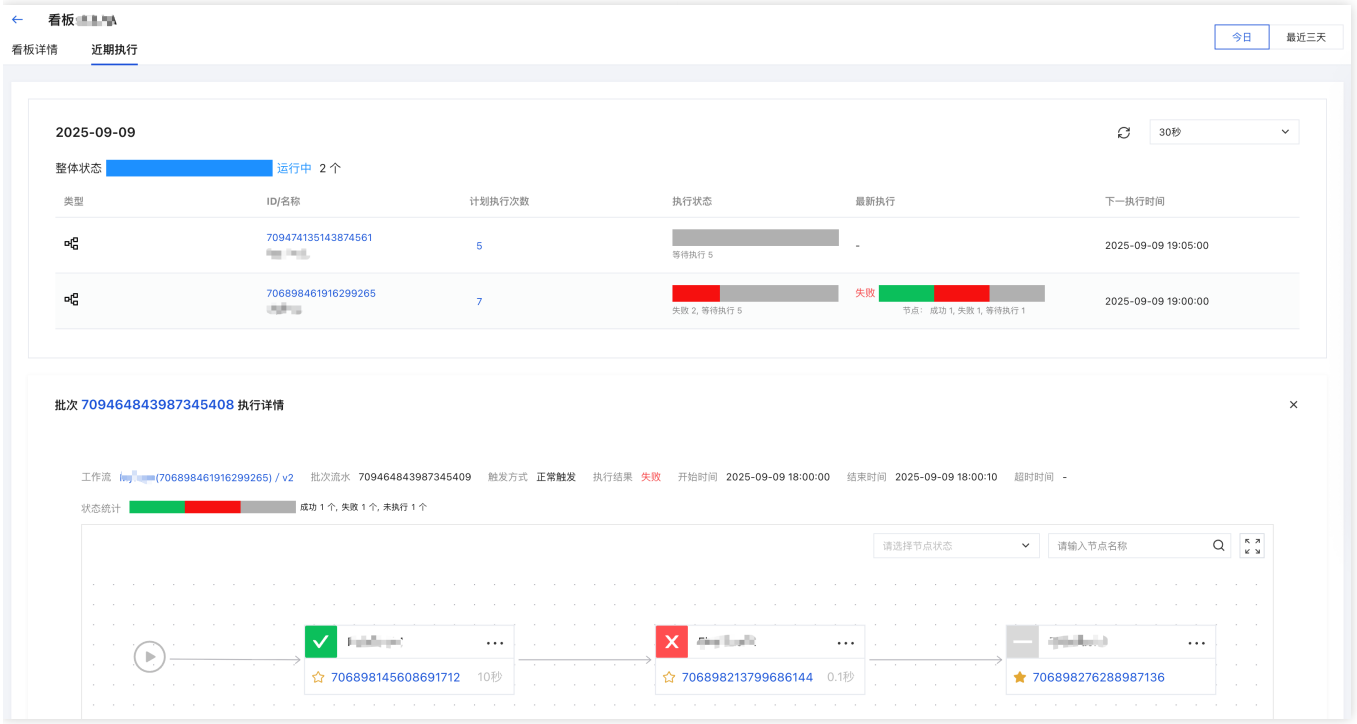
看板状态

执行看板创建好后，我们可以在执行看板列表中直观地查看看板今日的执行状态，有多少个还未执行完，多少个已经成功，多少个已经结束。

点开某个执行看板，我们可以看到看板最近的执行状态，可以选择今日（默认）或者最近三天的执行状态。再执行状态中详细列出了每个添加进看板的任务/工作流的执行状态，包括计划中总共有多少次执行，当前执行成功多少次、失败多少次、等待执行多少次，同时给出了最近一次的执行以及下一次执行的时间。对于工作流。看板的执行状态支持手动刷新和自动刷新（默认关闭），自动刷新允许配置每 15 秒或者 30 秒自动刷新看板的执行状态。

我们可以点开某个任务或者工作流的执行计划，查看已经执行的和等待执行的执行记录，对于已经执行的执行记录可以点击展开执行详情。


看板:
看板详情
近期执行
2025-09-09
整体状态
运行中 2 个
类型
ID/名称
计划执行次数
709474135143874561
5
706898461916299265
7
计划执行详情
执行时间
执行批次
触发方式
状态
2025-09-09 23:00:00
正常触发
等待
2025-09-09 22:00:00
正常触发
等待
2025-09-09 21:00:00
正常触发
等待
2025-09-09 20:00:00
正常触发
等待
2025-09-09 19:00:00
正常触发
等待
2025-09-09 18:00:00
709464843987345408
正常触发
失败
2025-09-09 17:00:00
709449744547471360
正常触发
失败
共 7 条
20 条 / 页
1
/ 1 页


看板
看板详情
近期执行
今日
最近三天
2025-09-09
30秒
整体状态
运行中 2 个
类型
ID/名称
计划执行次数
执行状态
最新执行
下一执行时间
709474135143874561
5
等待执行 5
-
2025-09-09 19:05:00
706898461916299265
7
失败 2, 等待执行 5
失败
节点：成功 1, 失败 1, 等待执行 1
2025-09-09 19:00:00
批次 709464843987345408 执行详情
工作流
批次流水 709464843987345409
触发方式 正常触发
执行结果 失败
开始时间 2025-09-09 18:00:00
结束时间 2025-09-09 18:00:10
超时时间 -
状态统计
成功 1 个, 失败 1 个, 未执行 1 个
请选择节点状态
请输入节点名称
706898145608691712
10秒
706898213799686144
0.1秒
706898276288987136

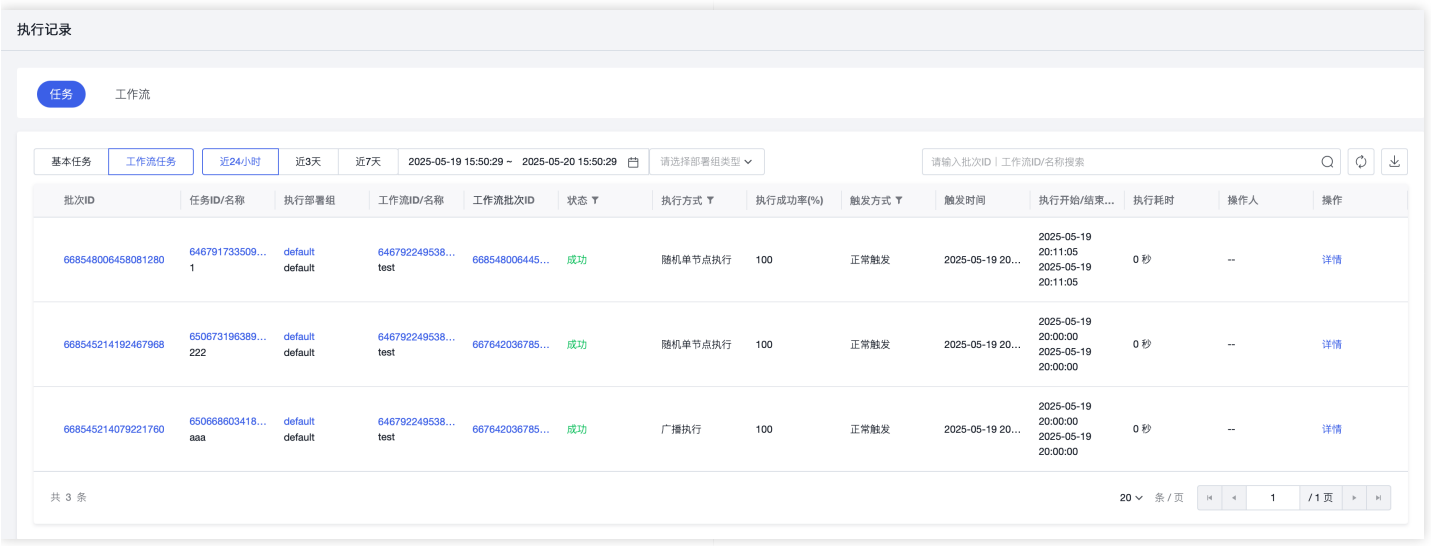
执行记录

执行记录查看

任务执行记录查看

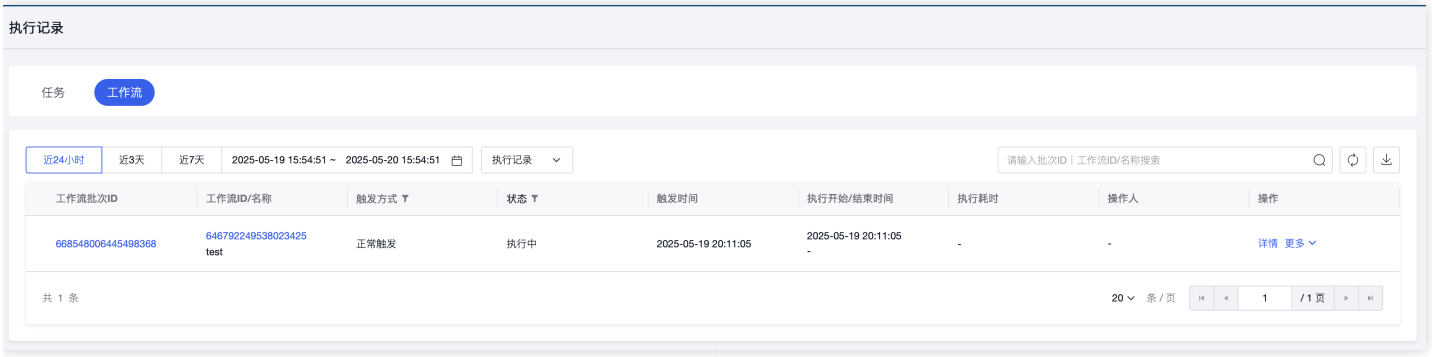
支持查看基本任务的执行部署组、状态、执行方式、执行成功率、触发方式、触发时间、执行开始/结束时间、执行耗时等，支持输出结果、停止执行、重新执行、断点续跑操作。

支持查看工作流任务的执行部署组、工作流ID/名称、工作流批次ID、状态、执行方式、执行成功率、触发方式、触发时间、执行开始/结束时间、执行耗时。



工作流执行记录查看

支持查看工作流的执行触发方式、状态、触发时间、执行开始/结束时间、执行耗时等，支持对工作流进行停止执行、重新执行和强制执行操作。



执行记录下载

支持以 csv 格式下载当前页面的执行记录。

执行记录保留与清理

TCT 中，任务和工作流的每次触发都会生成一条执行记录，一条执行记录涉及到数据库中多条数据记录，如果长时间不清理执行记录，会导致执行记录量非常庞大，大大降低数据库查询效率，从而严重影响任务的调度性能，因此 TCT 提供了执行记录自动清理的功能。

执行记录保留策略

TCT 根据保留策略来清理执行记录，并且允许用户给任务和工作流单独配置执行记录保留策略。保留策略支持保留天数、保留数量配置，保留天数和保留数量只要有一个达到限制就会触发清理。

保留天数

保留天数用于指定任务或工作流保留多少天的执行记录，默认为 7 天，最多支持保留 30 天。需要注意：对于历史数据，任务和工作流没有保留策略配置，会生成默认保留 7 天的配置，TCT 会将超过 7 天的执行记录进行清理。

保留数量

保留数量用于指定任务或工作流需要保留多少条执行记录，默认为 0 表示不限制，最多支持保留 100000 条。

清理时间

TCT 对执行记录的清理不是实时的，默认每 30 分钟执行一次清理，因此出现执行记录数超过配置的限额属于正常现象。在执行记录持续产生的情况下，可能超过限额是常态，但是最多超过 30 分钟产生的执行记录数。

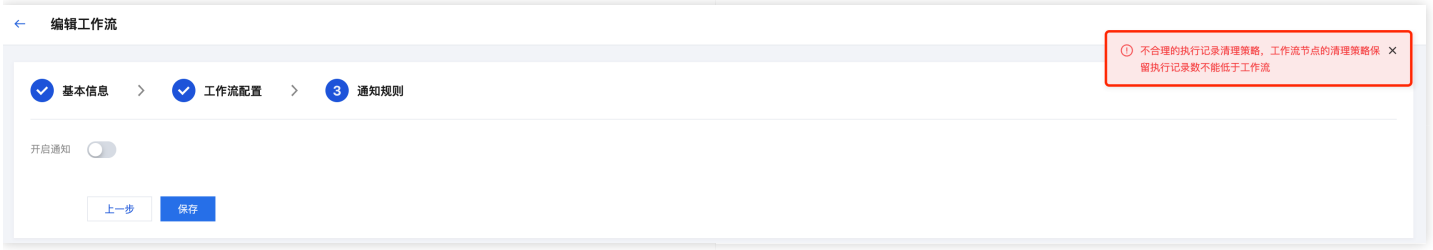
工作流保留策略

任务和工作流都可以配置执行记录保留策略，对于工作流需要额外注意的是：如果工作流和工作流里的节点（任务节点或者子工作流节点）都配置了保留策略，那么工作流节点的保留策略应该“不小于”工作流的保留策略。这样做的目的是为了防止工作流的执行记录还在，但是节点的执行记录被清理了。

这里保留策略的比较，我们说配置 A 不小于 配置 B，表示的是配置 A 能够保留的执行记录条数不小于配置 B 的，具体来说需要同时满足两个条件：

- 配置 A 的保留天数不小于配置 B 的保留天数。
- 配置 A 的保留数量不小于配置 B 的保留数量。注意不配置或者配置为 0 表示不限制，即表示无穷大。

如果工作流配置的保留策略部满足上述的约束，将会产生如下所示的报错信息：



输入输出

任务的输出指的是任务执行产生的输出结果，以 Key-Value 的形式体现，TCT SDK 从 2.1.7 版本开始支持输出结果。

> 注意：
>
> 由于 SDK 在 2.1.7 后续的几个版本中修复了几个重要的问题，建议使用 2.1.12 或之后的 SDK 版本。

上报输出结果

不同类型的任务（如 Java、Shell）上报输出结果的方式略有不同，我们将分别进行介绍。

Java 类型任务

在 TCT SDK 中，我们在 ProcessResult 类中新增了 output 字段（Map<String, String> 类型）用于收集任务执行后的输出结果。用户可以调用 ProcesResult 提供的以下方法添加输出结果：

```
public ProcessResult withOutput(String key, String value);
public ProcessResult withOutputs(Map<String, String> values);
```

例如：

```
public ProcessResult execute(ExecutableTaskData taskData) {
  ProcessResult result = ProcessResult.newSuccessResult();
  return result.withOutput("code", "OK").withOutput("score", "100");
}
```

Python/Shell 脚本任务

对于 Python 和 Shell 脚本任务，TCT 在执行的时候会自动注入上报日志的工具方法，业务脚本里可以直接调用。
Python 示例：

```
reportOutput("score", 100)
reportOutput("code", "OK")
```

Shell 示例：

```
reportOutput score 100
```

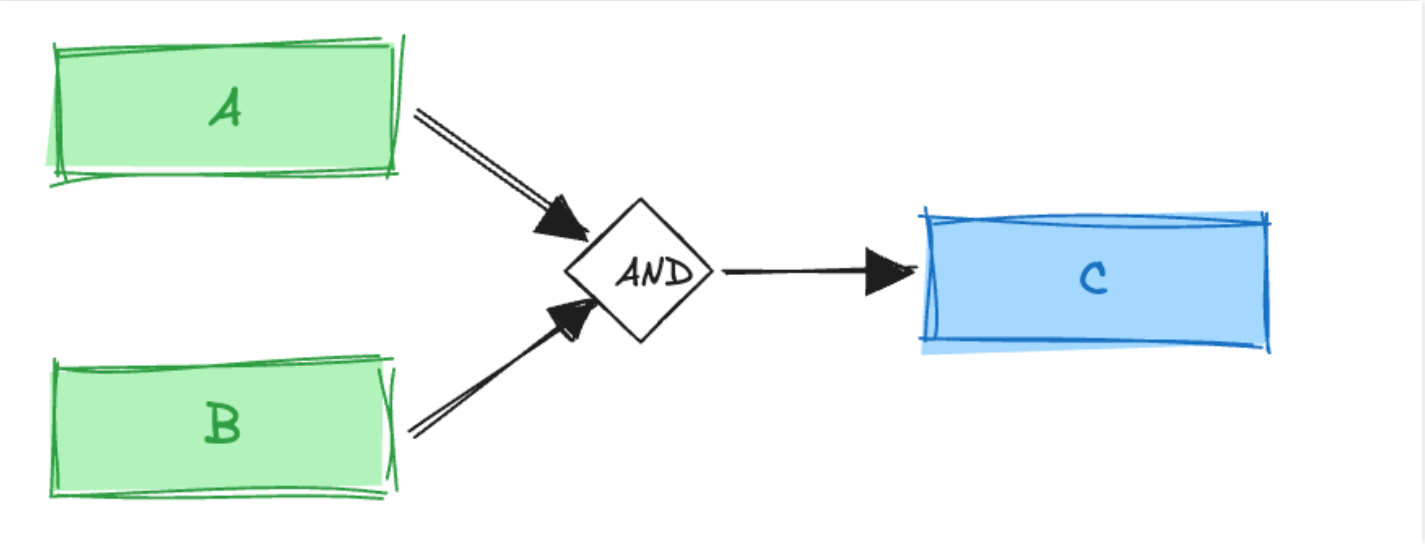
reportOutput code OK

外部任务

目前外部任务不支持上报输出结果。

获取输入数据

任务在工作流中执行的时候，可以获取该任务依赖的前置任务的输出结果，以作为该任务执行时的输入，TCT 会自动将这部分数据作为输入数据传递到任务执行中。



对于上图中的工作流，任务 A、B 已经执行成功，任务 C 正在执行，这时候任务 C 将会获得类似于以下的输入。其中 A、B 为任务 C 依赖的任务的 ID，每个 ID 下以 Key-Value 的形式列举出该任务执行的输出结果。

```
{
  "A": {
    "key1": "value1",
    "key2": "value2"
  },
  "B": {
    "key3": "value3"
  }
}
```

Java 类型任务

在任务开发中，我们可以通过 ExecutableTaskData 的 getInput 方法获取任务的输入数据：

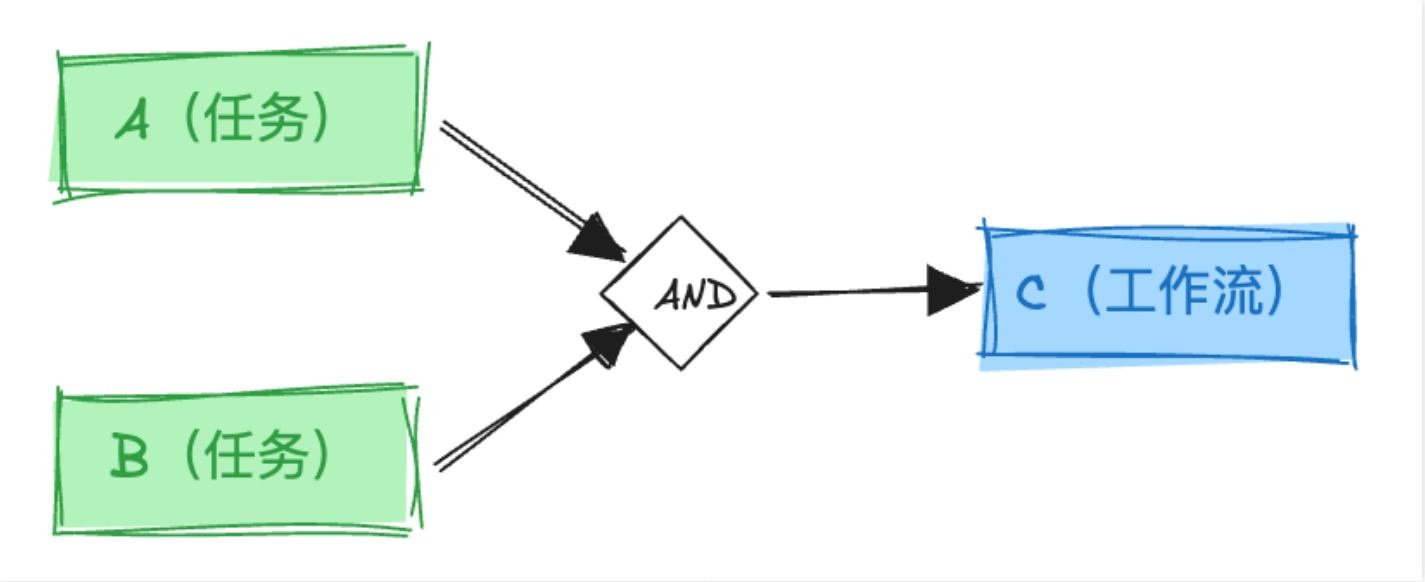
```
public Map<String, Map<String, Object>> getInput();
```

例如，在以下示例中我们将任务的输入数据打印到执行日志中：

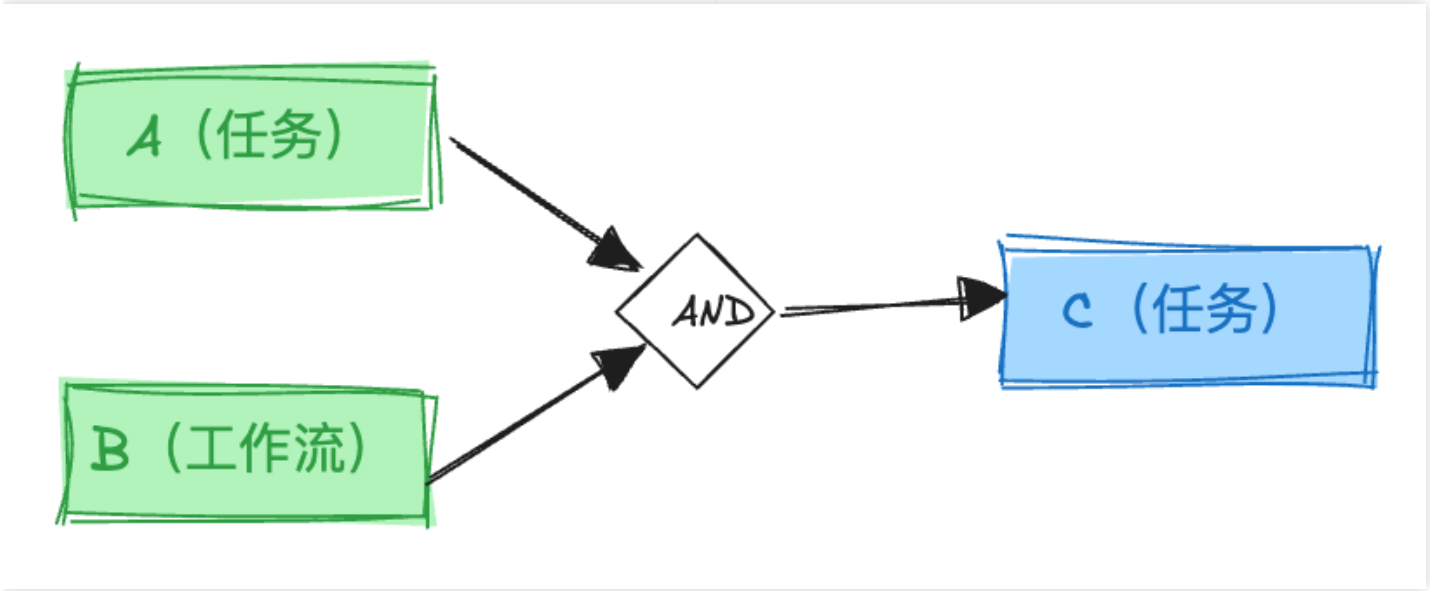
```
public ProcessResult execute(ExecutableTaskData taskData) {
  if (taskData.getInput() != null && taskData.getInput().size() > 0) {
    LogReporter.log(taskData, JsonJackson.writeAsString(taskData.getInput()));
  }

  ProcessResult result = ProcessResult.newSuccessResult();
  return result.withOutput("code", "OK").withOutput("score", "100");
}
```

需要特别注意的事，在工作流执行中以下情况并不会有数据传递到后续节点中：

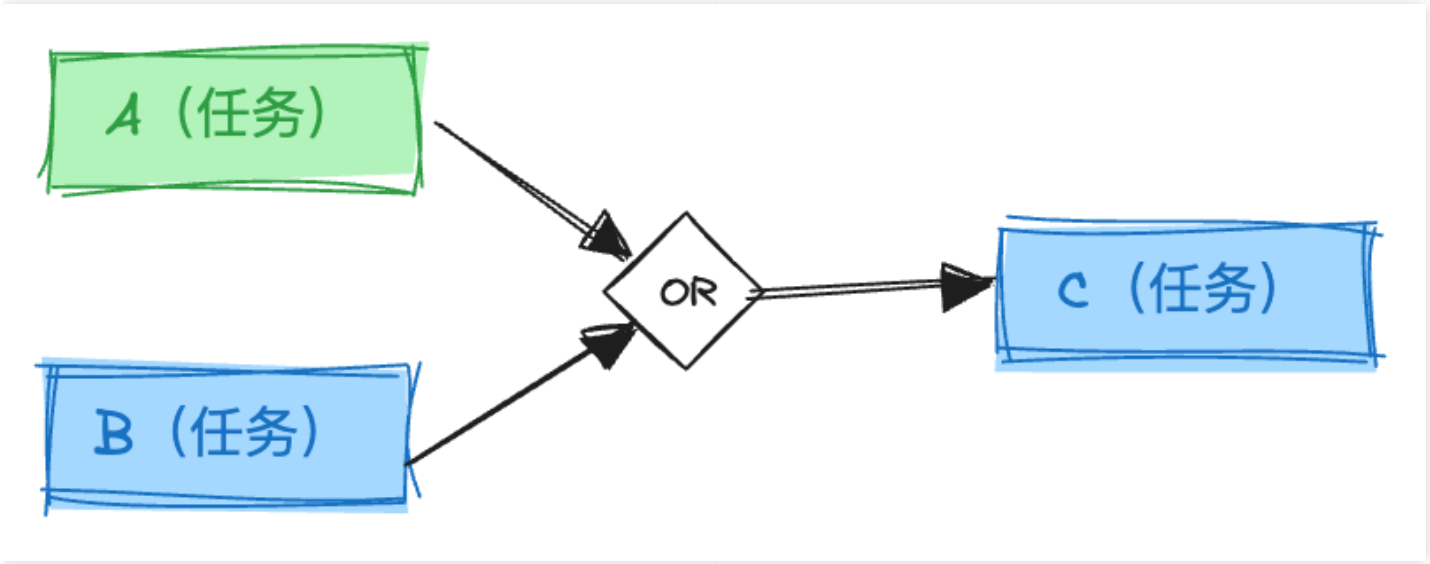
场景一：子工作流执行的时候不会有输入数据，下图中，节点 C 是一个子工作流，那么在 C 执行的时候我们不会传递输入数据。



场景二：工作流节点不会有输出结果，下图中由于节点 B 是工作流，那么任务 C 执行的时候只会有 A 的数据输入，节点 B 并不会有数据输入到 C。



场景三：只有执行结束的任务节点才会有数据传递到后续节点中，执行中的任务节点不会传递数据。下图中的任务 B 节点仍处于执行中，所以节点 C 执行的时候只会有任务 A 的数据传递过来。



Python/Shell 类型任务

Python 和 Shell 任务，可以在脚本里通过环境变量 TCT_INPUT 获取工作流上游传递来的输入数据。例如：

```
echo "Input data: ${TCT_INPUT:-{}}"
echo "Shell task finished."
```

外部任务

对于外部任务，TCT 会在调用外部任务接口时，将输入数据以 HTTP Header 方式传递，Header 名为 TASK-INPUT，输入数据为 JSON 格式字符串 Base64 编码后的结果。

> 注意：
>
> HTTP 的 Header 支持的数据量大小不是无限的，取决于具体的服务器、代理和服务端，一般都被限制在 4K-16K，因此需要关注外部任务输入数据的数据量。

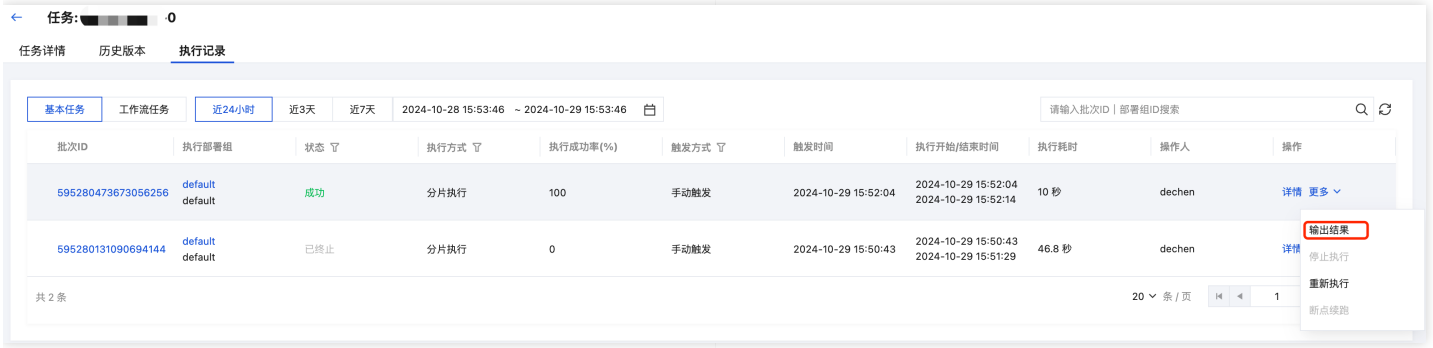
任务批次输出结果

对于分片和广播任务，一次执行（一个任务执行批次）会产生多个子任务，每个子任务都会上报输出结果到 TCT 中。这样对于一个任务执行批次，同样的 Key 会有多个值（每个子任务一个）。对于这样的输出结果我们可以配置聚合规则，对这些值进行聚合。

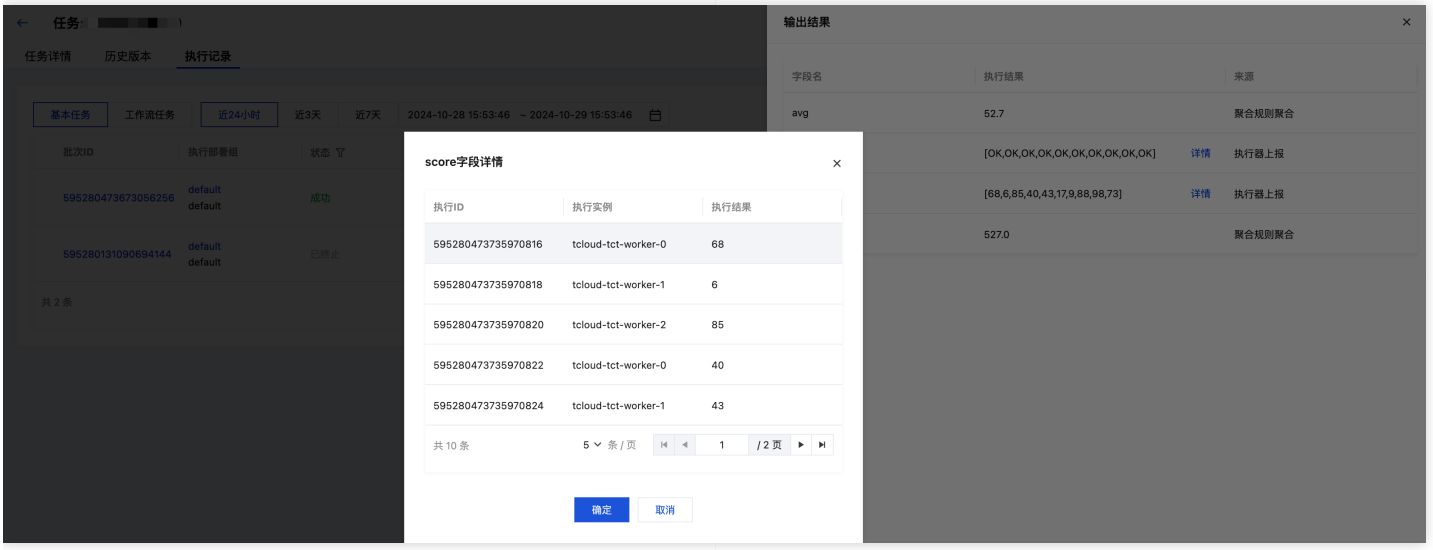
在创建任务的时候，我们可以在高级设置中对输出结果进行聚合（注意只有分片执行和广播执行的时候才会有聚合输出配置）。这里的聚合规则会将原始的字段通过求和、求平均等操作聚合得到一个新的字段，例如下图中，对子任务上报的 score 字段，进行求和，得到新的 sum 字段。

> 注意：
>
> 需要用户确保这里配置的待聚合字段存在，并且数据的类型支持聚合，当前聚合操作只支持数值类型。

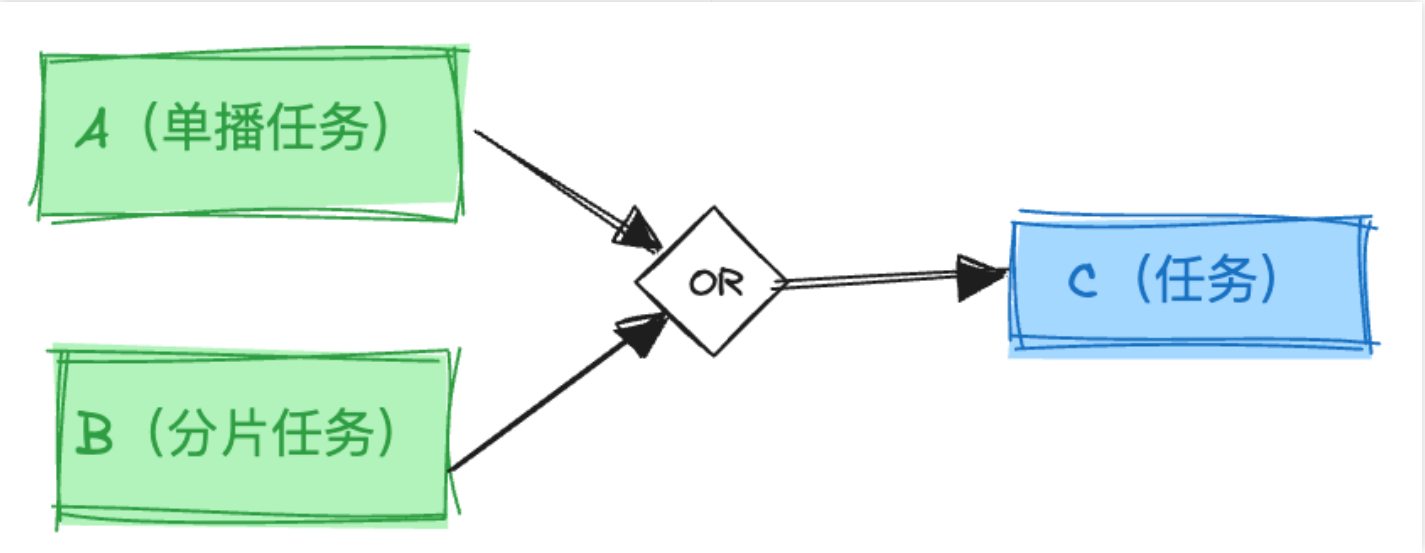
聚合的输出结果会和原始数据一起作为任务执行批次的输出结果，例如一下执行批次的输出结果包含原始的结果以及聚合后的结果：



对于原始的输出结果，可以展开查看是哪个子任务上报的：



> 注意：
>
> 控制台上，我们只对执行状态为成功的任务批次显示输出结果。



在工作流中，传递给任务节点的输入数据既包含原始的输出数据（数组形式）也包含聚合后的数据，假设上图中节点 B 是分片任务，而节点 A 是单播任务，那么任务 C 接收到的输入数据将是以下形式：

```
{
  "A": {
    "code": "ok",
    "score": "84"
  },
  "B": {
    "code": ["ok", "ok", "ok"],
    "score": ["99", "100", "98"],
    "aggregate_avg": 99,
    "aggregate_max": 100
  }
}
```

开发指南
Java开发

JDK 版本

推荐使用 JDK 1.8 或者以上版本，对于高于 1.8 的 JDK 版本，需要额外添加以下依赖包：

```
<dependency>
   <groupId>javax.xml.bind</groupId>
   <artifactId>jaxb-api</artifactId>
   <version>2.3.0</version>
</dependency>
```

添加依赖

找到 Maven 所使用的配置文件 settings.xml，一般为 ~/.m2/settings.xml，添加 TCT Maven 地址。

```
<?xml version="1.0" encoding="UTF-8"?>
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 http://maven.apache.org/xsd/set
tings-1.0.0.xsd">

  <pluginGroups></pluginGroups>
  <proxies></proxies>
  <servers></servers>
  <mirrors></mirrors>

  <profiles>
     <profile>
      <id>nexus</id>
      <repositories>
         <repository>
            <id>central</id>
            <url>http://repo1.maven.org/maven2</url>
            <releases>
               <enabled>true</enabled>
```

```
            </releases>
            <snapshots>
              <enabled>true</enabled>
            </snapshots>
          </repository>
        </repositories>
        <pluginRepositories>
          <pluginRepository>
            <id>central</id>
            <url>http://repo1.maven.org/maven2</url>
            <releases>
              <enabled>true</enabled>
            </releases>
            <snapshots>
              <enabled>true</enabled>
            </snapshots>
          </pluginRepository>
        </pluginRepositories>
      </profile>
      <profile>
        <id>tct</id>
        <repositories>
          <repository>
            <id>tct</id>
            <name>tct</name>
            <url>https://mirrors.cloud.tencent.com/nexus/repository/maven-public/</url>
            <releases>
              <enabled>true</enabled>
            </releases>
            <snapshots>
              <enabled>true</enabled>
            </snapshots>
          </repository>
        </repositories>
      </profile>
    </profiles>

    <activeProfiles>
      <activeProfile>nexus</activeProfile>
      <activeProfile>tct</activeProfile>
    </activeProfiles>

  </settings>
```

然后在 pom.xml 文件中添加 TCT spring boot starter 依赖。

```
  <dependency>
```

```
    <groupId>com.tencent.cloud </groupId>
    <artifactId>tct-spring-boot-starter </artifactId>
    <version>2.1.0-rc8</version>
</dependency>
```

任务开发

简单任务

编写 TCT 任务，只需要实现 TCT 提供的 com.tencent.cloud.task.sdk.client.spi.ExecutableTask 接口，在 execute 方法中实现任务执行逻辑，SDK 内部通过反射机制，生成任务对象实例，并执行 execute 方法。如下所示，SleepTask 是一个 sleep 10 秒的简单任务。
注意，将任务申明为 Bean 才能被 SDK 自动发现并作为预置任务上报给 TCT，如果不申明为 Bean 任务也可以使用但是在 TCT 控制台上创建任务时需要手动输入完整的类名。

```
import com.tencent.cloud.task.sdk.client.LogReporter;
import com.tencent.cloud.task.sdk.client.model.ExecutableTaskData;
import com.tencent.cloud.task.sdk.client.model.ProcessResult;
import com.tencent.cloud.task.sdk.client.spi.ExecutableTask;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Component;

import java.lang.invoke.MethodHandles;

@Component
public class SleepTask implements ExecutableTask {
    private static final Logger LOG = LoggerFactory.getLogger(MethodHandles.lookup().lookupClass());

    @Override
    public ProcessResult execute(ExecutableTaskData taskData) {
        LOG.info("Run sleep task, taskMeta: {}", taskData.getTaskMeta().toString());

        try {
            Thread.sleep(10 * 1000L);
            return ProcessResult.newSuccessResult();
        } catch (InterruptedException e) {
            LogReporter.log(taskData, "Task is terminated.");
            return ProcessResult.newCancelledResult();
```

```
        } catch (Throwable e) {
            LogReporter.log(taskData, String.format("Exception when sleep: %s", e.getMessage()));
            return ProcessResult.newFailResult();
        }
    }
}
```

可停止任务

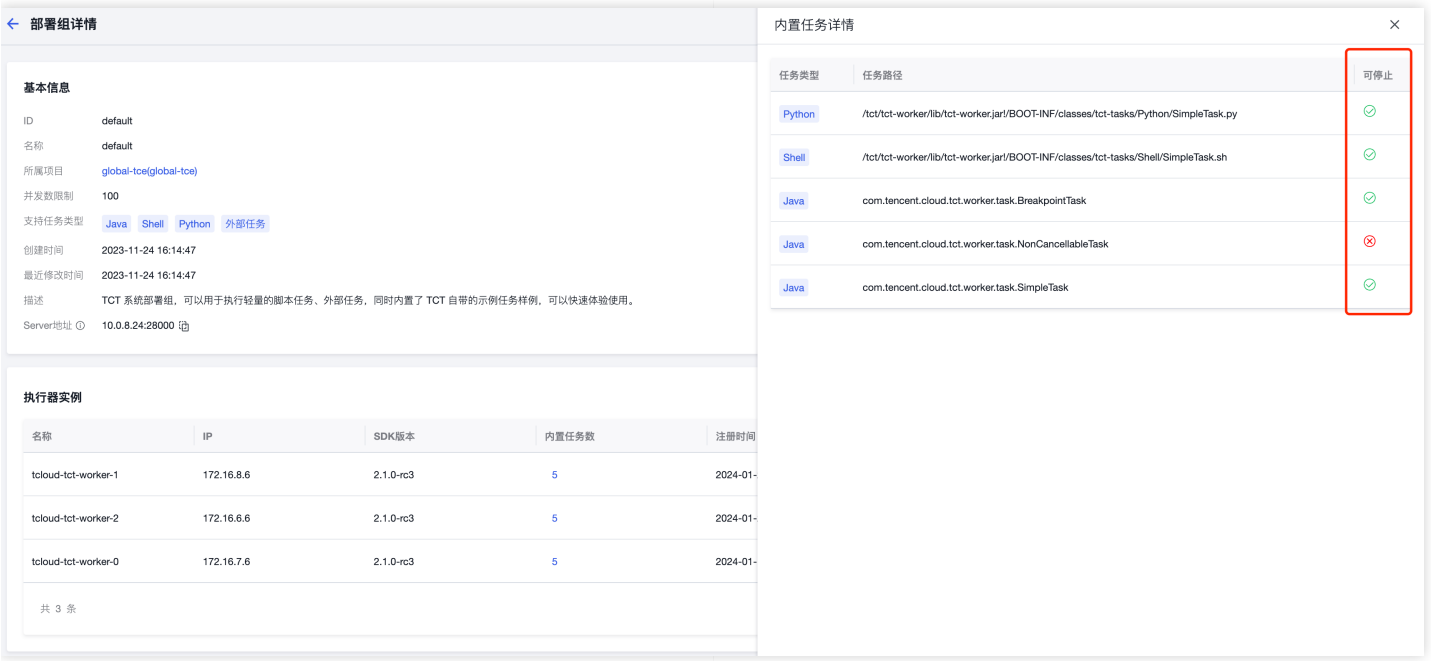
在 TCT 控制台中，我们可以停止一个执行中的任务，为了使任务可停止，编写任务逻辑的时候，需要实现 com.tencent.cloud.task.sdk.client.spi.TerminableTask 接口，在 cancel 方法中实现任务的停止逻辑，并返回停止结果。

如下所示，SleepTask 中我们通过 Future 的 cancel 方法停止 sleep 任务，这时候任务执行线程会收到中断信号，抛出中断异常 InterruptedException，示例代码中捕获了 InterruptedException 异常并返回任务终止成功。

```
import com.tencent.cloud.task.sdk.client.LogReporter;
import com.tencent.cloud.task.sdk.client.model.ExecutableTaskData;
import com.tencent.cloud.task.sdk.client.model.ProcessResult;
import com.tencent.cloud.task.sdk.client.model.TerminateResult;
import com.tencent.cloud.task.sdk.client.remoting.TaskExecuteFuture;
import com.tencent.cloud.task.sdk.client.spi.ExecutableTask;
import com.tencent.cloud.task.sdk.client.spi.TerminableTask;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Component;

import java.lang.invoke.MethodHandles;

@Component
public class SleepTask implements ExecutableTask, TerminableTask {
    private static final Logger LOG = LoggerFactory.getLogger(MethodHandles.lookup().lookupClass());

    @Override
```

```
    public ProcessResult execute(ExecutableTaskData taskData) {
        LOG.info("Run sleep task, taskMeta: {}", taskData.getTaskMeta().toString());

        try {
            Thread.sleep(10 * 1000L);
            return ProcessResult.newSuccessResult();
        } catch (InterruptedException e) {
            LogReporter.log(taskData, "Task is terminated.");
            return ProcessResult.newCancelledResult();
        } catch (Throwable e) {
            LogReporter.log(taskData, String.format("Exception when sleep: %s", e.getMessage()));
            return ProcessResult.newFailResult();
        }
    }

    @Override
    public TerminateResult cancel(TaskExecuteFuture taskExecuteFuture, ExecutableTaskData executable
TaskData) {
        taskExecuteFuture.cancel(true);
        return TerminateResult.newTerminateSuccessResult();
    }
}
```

TCT SDK 扫描预置任务的时候会通过判断任务类是否实现了 TerminableTask 接口判断任务是否支持停止，TCT 控制台中只有支持停止的任务允许执行停止操作。



任务停止原理

首先，我们需要了解 Java 体系内，如何停止一个执行中的任务。向一个 Alive 状态的 Thread 发送中断信号，不一定

能中断线程的执行。中断信号只是向运行的线程一个建议，告诉它有外界希望中断它，至于线程接受到信号后要做出何种反应，完全由线程及运行状态自身决定。
Java 提供的 API 中，对于中断信号，通常存在两种响应形态：

- 设置中断状态标志位，通过 Thread.isInterrupted() 进行判断。
- 当执行任务的线程处于 BLOCKED 状态（例如调用了 wait、sleep、join 等方法）时，向线程发送中断信号，通常会抛出中断异常，如 java.lang.InterruptedException。Java 中常见的中断异常有 java.lang.InterruptedException、 java.io.InterruptedIOException、 java.nio.channels.ClosedByInterruptException。

TCT 对用户侧提供了中断任务执行线程的 API 来向执行线程发送中断信号：

```
cancel(TaskExecuteFuture future, ExecutableTaskData tasData)
```

此方法中暴露 Future 对象，底层是对当前任务提交执行后返回的java.util.concurrent.future的封装。 通过调用 future.cancel(boolean) 向执行任务的线程发送中断信号。 当在控制台操作停止任务时，cancel 方法将会被调用。

因此，我们在实现任务执行逻辑的时候，需要判断中断标志位和捕获中断异常来实现可停止的任务，如下所示，加入我们的任务需要循环处理一批数据，在每一次循环的时候我们都判断：

```
@Override
public ProcessResult execute(ExecutableTaskData taskData) {
    try {
        List<String> dataset = Arrays.asList("id1", "id2", "id3");
        for(String data : dataset) {
            if (Thread.currentThread().isInterrupted()) {
                return ProcessResult.newCancelledResult();
            }

            // 数据处理逻辑。。。
        }
    } catch (InterruptedException e) {
        LogReporter.log(taskData, "Task is terminated.");
        return ProcessResult.newCancelledResult();
    } catch (Throwable e) {
        LogReporter.log(taskData, String.format("Exception when sleep: %s", e.getMessage()));
        return ProcessResult.newFailResult();
    }
}
```

除了通过中断信号实现可停止任务外，我们也可以通过 execute 和 cancel 两个接口配合实现业务的逻辑终止，例如通过一个 isCancelled 字段标识业务逻辑是否被终止，在 cancel 方法中设置它，在 execute 中执行业务逻辑时检查它。

```
import com.tencent.cloud.task.sdk.client.model.ExecutableTaskData;
import com.tencent.cloud.task.sdk.client.model.ProcessResult;
import com.tencent.cloud.task.sdk.client.model.TerminateResult;
import com.tencent.cloud.task.sdk.client.remoting.TaskExecuteFuture;
import com.tencent.cloud.task.sdk.client.spi.ExecutableTask;
import com.tencent.cloud.task.sdk.client.spi.TerminableTask;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Component;

import java.lang.invoke.MethodHandles;
import java.util.Arrays;
import java.util.List;
import java.util.concurrent.atomic.AtomicBoolean;

@Component
public class SleepTask implements ExecutableTask, TerminableTask {
    private static final Logger LOG = LoggerFactory.getLogger(MethodHandles.lookup().lookupClass());
    private final AtomicBoolean isCancelled = new AtomicBoolean(false);

    @Override
    public ProcessResult execute(ExecutableTaskData taskData) {
        List<String> dataset = Arrays.asList("id1", "id2", "id3");
        for (String data : dataset) {
            if (isCancelled.get()) {
                return ProcessResult.newCancelledResult();
            }

            // 数据处理逻辑。。。
        }
        return ProcessResult.newSuccessResult();
    }

    @Override
    public TerminateResult cancel(TaskExecuteFuture taskExecuteFuture, ExecutableTaskData executable
TaskData) {
        // 设置终止状态，终止成功
        isCancelled.set(true);
        // 返回终止成功
        return TerminateResult.newTerminateSuccessResult();
    }
}
```

任务工厂

任务工厂是 TCT 用来生成任务实例的工厂类，TCT 提供了默认的工厂类 com.tencent.cloud.task.sdk.client.DefaultTaskFactory，也支持用户自定义任务工厂类。

默认工厂

默认工厂 DefaultTaskFactory 通过 Java 的反射机制来生成任务对象的实例，如下代码所示：

```
public class DefaultTaskFactory implements ExecutableTaskFactory {
    private static final Logger LOG = LoggerFactory.getLogger(MethodHandles.lookup().lookupClass());
    private final ClassLoader classLoader;

    public DefaultTaskFactory(ClassLoader classLoader) {
        this.classLoader = classLoader;
    }

    @Override
    public ExecutableTask newExecutableTask(ExecutableTaskData taskData) throws InstancingException
{
        String taskName = taskData.getTaskContent();
        if (LOG.isDebugEnabled()) {
            LOG.debug("producing instance of ExecutableTask: '" + taskName + "'");
        }
        try {
            Class<?> taskClass = Class.forName(taskName, true, classLoader);
            if (!ExecutableTask.class.isAssignableFrom(taskClass)) {
                throw new InstancingException("Problem instancing ExecutableTask, "
                        + "Caused by task Class name '" + ExecutableTask.class.getName()
                        + "' is not AssignableFrom Class '" + taskName + "'");
            }
            return (ExecutableTask) taskClass.newInstance();
        } catch (ClassNotFoundException t) {
            throw new InstancingException("Class '" + taskName + "' is not found", t);
        } catch (Exception e) {
            if (e instanceof InstancingException) {
                throw (InstancingException) e;
            }
            throw new InstancingException("Problem instancing ExecutableTask,"
                    + " task Class is '" + taskName + "'", e);
        }
    }
}
```

自定义任务工厂

普通 Java 工厂

创建自定义任务工厂，只需要实现 com.tencent.cloud.task.sdk.client.spi.ExecutableTaskFactory 接口，如下示例所示，我们创建了 SimpleExecuteTaskFactory 类，它扩展了默认的 DefaultTaskFactory 工厂类，DefaultTaskFactory 实现了 ExecutableTaskFactory 接口。

```
package com.tencent.cloud.task.factory

public class SimpleExecuteTaskFactory extends DefaultTaskFactory {
    private static final Logger LOG = LoggerFactory.getLogger(MethodHandles.lookup().lookupClass());

    public SimpleExecuteTaskFactory() {
        super(Thread.currentThread().getContextClassLoader());
    }

    public SimpleExecuteTaskFactory(ClassLoader classLoader) {
        super(classLoader);
    }

    @Override
    public ExecutableTask newExecutableTask(ExecutableTaskData taskData) throws InstancingException
{
        LOG.info("generate task: {}", taskData.getTaskContent());
        return super.newExecutableTask(taskData);
    }
}
```

创建好工厂类后，我们需要修改任务应用（即执行器）的启动配置，在 application.yml 里配置任务工厂类：

```
tct:
  client:
    properties:
      "task.factory.name": "com.tencent.cloud.task.factory.SimpleExecuteTask
```

Spring 框架工厂

在 Spring 框架中，我们可以将任务注册成 bean，然后自定义任务工厂从应用上下文中获取这些任务 bean。

```
@Component
public class SpringExecuteTaskFactory implements ExecutableTaskFactory, ApplicationContextAware {
    private static final Logger LOG = LoggerFactory.getLogger(MethodHandles.lookup().lookupClass());
    private ApplicationContext applicationContext;
```

```
    private final ExecutableTaskFactory defaultFactory = new DefaultTaskFactory(Thread.currentThread
().getContextClassLoader());
    @Override
    public ExecutableTask newExecutableTask(ExecutableTaskData executableTaskData) throws Instancin
gException {
        try {
            ExecutableTask executableTask = (ExecutableTask)applicationContext.getBean(Class.forName(ex
ecutableTaskData.getTaskContent()));
            LOG.info("generate executableTask bean SpringExecutableTaskFactory. taskName: {}", executabl
eTaskData.getTaskContent());
            return executableTask;
        } catch (Throwable t) {
            return defaultFactory.newExecutableTask(executableTaskData);
        }
    }
    @Override
    public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {
        this.applicationContext = applicationContext;
    }
```

同样需要修改配置指定工厂类：

```
tct:
  client:
    properties:
      "task.factory.name": "com.tencent.cloud.task.factory.SpringExecuteTaskFactory"
```

任务配置

TCT 任务应用（即执行器）提供以下配置项供用户配置：

配置项	说明
tct.enabled	是否开启 TCT 任务调度，只有该配置项为 true 才会启用 TCT 功能。
tct.server.host	TCT服务端地址，可以在 TCT 部署组详情中获取。
tct.server.port	
tct.client.groupId	部署组 ID，如果任务应用通过 TSF 部署，该配置可以不填，TSF 会自动注入部署组 ID。其他情况下需要手动填充在 TCT 控制台创建好的部署组的 ID，例如默认的部署组填 default。

tct.client.instanceId	任务应用实例（执行器实例）的 ID，如果任务应用通过 TSF 部署，在部署时会自动注入，该配置可不填。其他情况下需要自行配置，注意这里的 ID 当前需要在部署组范围内唯一。
tct.client.accessKey	用于认证和鉴权，在TCS 控制台获取，当前登录用户 - 账号信息 - API密钥管理。
tct.client.secretKey	
tct.client.environments	指定该任务应用（执行器）具备怎样的执行环境，即支持执行什么类型的任务，例如 Java、Python、Shell、External。

可以通过以下几种方式配置这些配置项，并且几种方式的优先级如下，高优先级的配置会覆盖低优先级的配置：环境变量 > 命令行参数 > application.yml。

application.yml

```
tct:
  enabled: true
  server:
    host: server.chongqing.tct
    port: 28000
  client:
    groupId: BjFnVcXkVw
    instanceId: tct-demo-ins1
    accessKey: xxx
    secretKey: xxx
    environments:
      - Java
      - Shell
      - Python
```

命令行参数

```
java \
-Dtct.server.host=10.0.8.24 \
-Dtct.server.port=28000 \
-Dtct.client.groupId=BjFnVcXkVw \
-Dtct.client.instanceId=tct-demo-ins1 \
-Dtct.client.accessKey=xxx \
-Dtct.client.secretKey=xxx \
-jar tct-demo.jar
```

环境变量

可以通过其他方式注入环境变量（例如 k8s Pod 中指定 env），也可以通过启动参数注定环境变量，例如：

```
java \
-Dtct_server_host=10.0.8.24 \
-Dtct_server_port=28000 \
-Dtct_group_id=BjFnVcXkVw \
-Dtct_instance_id=tct-demo-ins1 \
-Dtct_access_key=xxx \
-Dtct_secret_key=xxx \
-Dtct_environments=Java,Python,Shell \
-jar tct-demo.jar
```

最佳实践
最佳实践

任务/工作流触发周期

如果任务或者工作流采用定时触发或周期触发，触发时间的间隔不允许小于 30 秒。

触发太过频繁的任务并不适用分布式任务调度系统，因为分布式任务调度系统主要解决任务的可靠执行（例如通过失败重试、故障转移），对任务执行的实时性没有严格的保证。分布式任务调度系统本身在进行可靠调度上就可能存在秒级的延迟（例如在性能较差的环境中），对于秒级触发周期的任务可能会产生任务积压以及乱序。对于实时性要求较高的任务，建议采用简单高效的方式实现，例如在应用中启用定时任务（如 Spring 中通过 @Scheduled 注解创建定时任务）。

任务/工作流触发时间

如果没有特殊需求，尽量不要让任务和工作流集中在同一个时间点触发。

例如我们创建了一堆任务，期望凌晨 00:00:00 进行触发，如果对时间没有严格的要求，我们可以将触发时间分散在一个时间周期内，如 00:00:00 - 00:10:00。这样做既可以避免 TCT 在某一个时候任务调度量激增加重系统负担，也可以避免执行器实例（客户端）同时出现大量并发任务出现性能问题。

任务/工作流优先级

TCT 支持任务/工作流的优先级配置，分为一般和重要两个优先级。使用时推荐根据任务/工作流的性质，合理设置优先级，简单地全部设置为重要或者一般并不是好的方式。一般优先级的任务/工作流在系统出现性能瓶颈的时候会被概率性地限流（直接标注某次触发为被限流，不进行调度），以确保重要优先级的任务/工作流能正确地执行，同时保护系统能够继续平稳运行不至于被拖垮。

默认部署组

TCT 提供了一个默认的部署组（名称和 ID 均为 default）以及 3 副本的执行器实例。该部署组是一个公共的部署组，所有的用户都可以查看该部署组以及该部署组下的任务和工作流，它的作用主要是为了执行一些简单轻量的 TCT 任务以及快速体验 TCT。对于复杂的生产任务以及需要做权限隔离的使用场景，不建议使用默认的部署组，用户需要

创建自己的部署组并设置权限。

任务发现

用户在基于 SDK 开发任务时，建议将任务注册成 Bean 以便能被 SDK 自动发现并注册到 TCT 服务端，这样用户在创建任务时能够直接获取到任务以及任务相关的信息。

执行器实例 ID

TCT 是根据执行器实例注册时提供的执行器实例 ID 来标识该执行器实例的，以便执行器实例发生下线重新注册时 TCT 服务端能够识别出来，在执行器实例重启或者重建的时候，需要尽量保证执行器实例 ID 不变。

尤其需要注意的是在 Kubernetes 中容器化部署执行器的时候，通常我们会将 Pod 名称作为执行器实例 ID，如果我们采用 StatefulSet 方式部署，那么各个 Pod 的名称是固定的（格式为 xxx-0, xxx-1, xxx-2），即使 Pod 删除重建名称也不会变。但是如果采用的是 Deployment 方式部署，每次重建 Pod，Pod 名称就会改变，这样执行器实例 ID 也会发生变化。这里强烈建议采用 StatefulSet 方式部署执行器。

如果执行器实例 ID 发生变化，影响是什么呢？执行器 ID 变化主要会导致 TCT 无法下发指令（如停止执行）干预任务的执行，例如 TCT 下发了任务到执行器实例 A 中执行，如果这时候执行器实例 A 被重建了，并且实例 ID 变成了 A1，因为执行器实例重建了任务就不存在了，后续不会上报任何该任务相关进度信息到 TCT 中，而 TCT 也无法对执行的任务进行干预（如停止执行），因为 TCT 是将任务下发给了 A，而现在找不到 A 的连接，TCT 无法判断是 A 彻底下线了还是因为网络隔离暂时联系不上了。这样最终会导致任务在 TCT 中一直处于执行中的状态直到超时，并且正常的手动停止也不会成功，需要通过强制停止来扭转任务状态。

通用参考
性能

性能指标 TPS

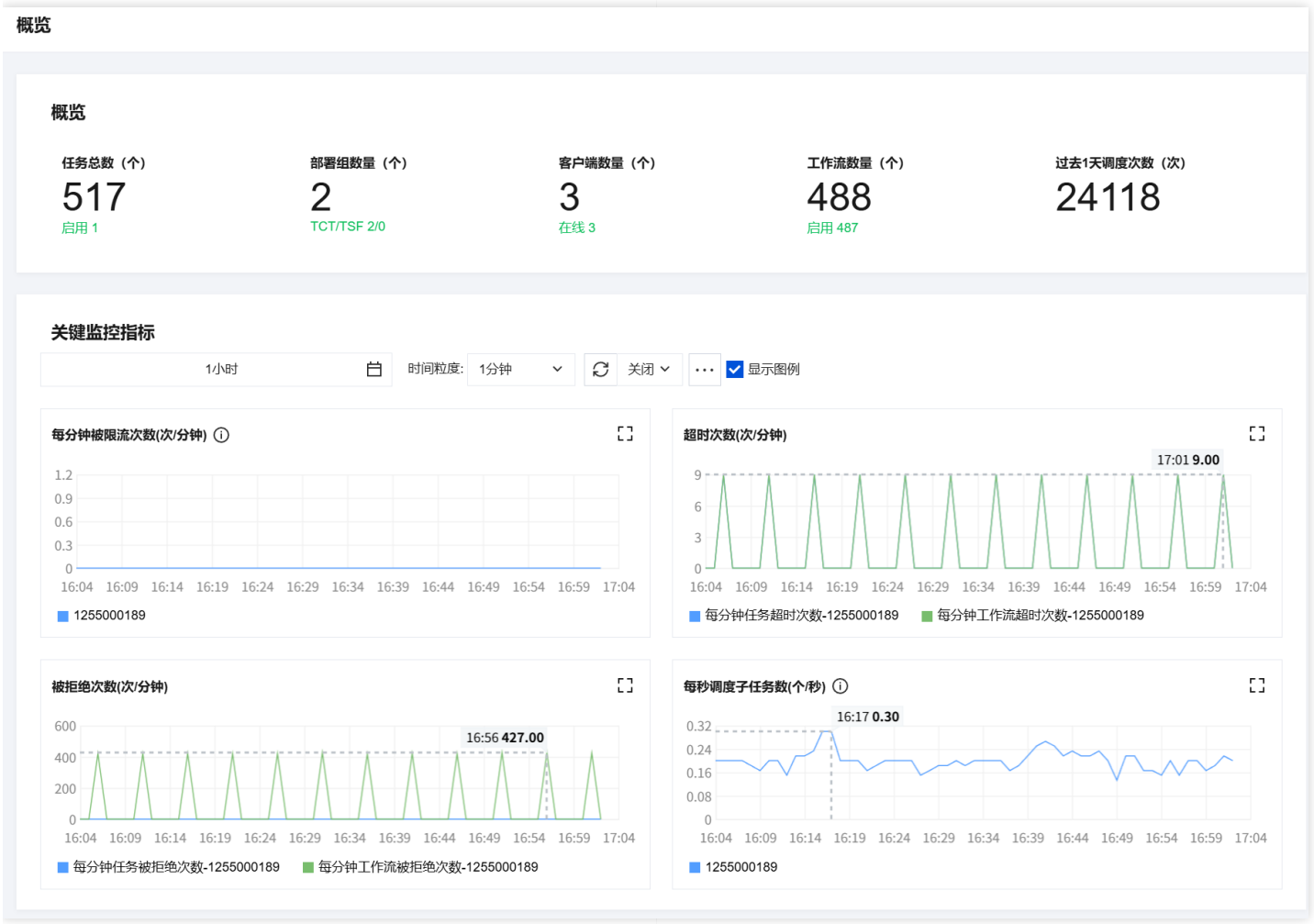
在 TCT 中，任务调度的单位是子任务（即 Execute），例如一个 100 分片的分片任务，最终会拆分成 100 个子任务进行调度，一个广播任务在 10 个执行器实例上调度，会生成 10 个子任务进行调度，因此单位时间内调度的子任务数能够正确反应当前系统的负载情况。而任务数量、工作流数量因为任务/工作流是否启用以及触发频率的原因并不能很好反应系统的负载。
我们在 TCT 中定义 TPS 为每秒调度的子任务数量，例如假设有 10000 个分片任务（分片数 12），任务每分钟触发一次，那么每分钟会产生 120000 个子任务，每秒产生 2000 个子任务，那么 TPS 为 2000，如果这些任务一天触发一次（并且假设任务触发时间平均分布在一天范围内），那么 TPS 为 0.14。

测试结果

TCT 使用容器化 MariaDB 12C8G 规格下，最大 TPS 达到 3200，详细的性能评估请参考性能测试报告。

监控指标

TCT 采集了丰富的监控指标，重要的指标会在租户概览页展示：



关键监控指标说明如下：

指标名	单位	说明
tct_throttling	个	最近一分钟发生限流的次数。
tct_tps	个/秒	当前 TCT 每秒调度的任务数，即 TPS。
tct_trigger_count_task_success	次/分钟	最近一分钟成功触发的任务数量。
tct_trigger_count_flow_success	次/分钟	最近一分钟成功触发的工作流数量。
tct_task_timeout	次/分钟	最近一分钟任务批次执行超时次数。

tct_flow_timeout	次/分钟	最近一分钟工作流批次执行超时次数。
tct_mean_turnaround_time	秒	调度平均流转时间，流转时间指从计划触发到完成调度耗费时间。

词汇表

基本概念

概念	说明
执行器	执行器是用于执行任务的进程，可以将它理解为执行任务的 Agent，它通过 TCT SDK 和 TCT 进行交互，接受 TCT 任务的调度并执行，目前 TCT SDK 只有 Java 版本，因此执行器也只支持 Java。 有时候我们也会将执行器称之为任务应用。
部署组	部署组是执行器的一个逻辑集合，不同的执行器实例通过 TCT SDK 注册到 TCT 的一个部署组中（通过执行器的配置参数），一个部署组中的执行器实例是完全对等的。例如在 k8s 中，一个 Deployment 对应一个部署组，其中的一个 Pod 即对应于一个执行器实例。 在 TCT 里，任务的调度是在一个部署组内进行的，例如广播执行、分片执行都是在部署组中的执行器上进行调度。
批次	在 TCT 里，我们将任务或者工作流的一次触发执行称之为批次（Batch），因此会有任务批次（TaskBatch）和工作流批次（FlowBatch）的概念。一个批次可以被任意手动重试，一次手动重试会产生一次新的执行（流水），所有的这些执行流水统一构成一个批次，这也是批次概念的由来。
流水	流水（Log）是和批次紧密相关的概念，在 TCT 中任务或者工作流的一次执行（Batch）可以被手动重试，一次重试会产生该批次的一次新的流水（BatchLog），因此有任务批次流水（TaskBatchLog）和工作流批次流水（FlowBatchLog）的概念。 TCT 中任务和工作流的每一次具体的执行都对应一个批次流水，而每个批次流水都属于一个批次。任务或工作流被触发一次，会产生一个批次（Batch），同时产生该批次下的一个批次流水（BatchLog）对应于此次执行，如果后续该批次被重新执行，会在该批次下产生新的批次流水。
Execute	在 TCT 中，一个任务最终会被拆成一个个子任务调度到执行器中执行。随机单点执行的任务会产生一个子任务调度到随机选定的一个执行器中执行；广播执行的任务会产生 N（部署组中执行器实例个数）个子任务分别调度到各个执行器中执行；分片执行（分片数 M）的任务会产生 M 个子任务调度到各个执行器中执行。 在 TCT 中，因为历史原因我们将子任务称之为一个 Execute，因为一个子任务也可以被重试，因此也有 ExecuteLog 即子任务流水的概念，一个 Execute 可以包含多个 ExecuteLog。