PowerJob 产品手册

产品介绍 & 接入指南

PowerJob 简介

快速开始（本地IDE版）

快速开始（docker-compose版）

正式部署

- 调度中心（powerjob-server）部署
- 执行器（powerjob-worker）初始化
- 处理器（Processor）开发
- 任务与工作流配置
- 用户（账号体系）配置
- 报警配置（邮件、WebHook、钉钉、自定义）

高级特性

- 容器
- OpenAPI
- 工作流（workflow）
- 官方处理器
- 多语言支持

版本与升级

- 升级指南
- 兼容性说明

错误码

在线试用

开发者指南

- 架构与原理
- 开发计划
- 演讲稿

FAQ

常见问题

其他

鸣谢

PowerJob 简介

项目介绍

中国开源年会 & Apache 路演 – 关于 PowerJob 的介绍：
https://www.bilibili.com/video/BV1SK411A7F3/

产品特性

PowerJob（原OhMyScheduler）**是全新一代分布式任务调度与计算框架，其主要功能特性如下**：

- 使用简单：提供前端Web界面，允许开发者可视化地完成调度任务的管理（增、删、改、查）、任务运行状态监控和运行日志查看等功能。
- 定时策略完善：支持 CRON 表达式、固定频率、固定延迟和API四种定时调度策略。
- 执行模式丰富：支持单机、广播、Map、MapReduce 四种执行模式，**其中 Map/MapReduce 处理器能使开发者寥寥数行代码便获得集群分布式计算的能力**。
- **工作流支持**：支持在线配置任务依赖关系（DAG），以可视化的方式对任务进行编排，同时还支持上下游任务间的数据传递，以及多种节点类型（判断节点 & 嵌套工作流节点）。
- 执行器支持广泛：支持 Spring Bean、内置/外置 Java 类，另外可以通过引入官方提供的依赖包，一键集成 Shell、Python、HTTP、SQL 等处理器，应用范围广。
- 运维便捷：支持**在线日志**功能，执行器产生的日志可以在前端控制台页面实时显示，降低 debug 成本，极大地提高开发效率。
- 依赖精简：**最小仅依赖关系型数据库（MySQL/PostgreSQL/Oracle/MS SQLServer...）**
- 高可用 & 高性能：调度服务器经过精心设计，一改其他调度框架基于数据库锁的策略，实现了无锁化调度。部署多个调度服务器可以同时实现高可用和性能的提升（支持无限的水平扩展）。
- 故障转移与恢复：任务执行失败后，可根据配置的重试策略完成重试，只要执行器集群有足够的计算节点，任务就能顺利完成。

在线试用：https://www.yuque.com/powerjob/guidence/hnbskn

适用场景

- 有定时执行需求的业务场景：如每天凌晨全量同步数据、生成业务报表、未支付订单超时取消等。
- 有需要全部机器一同执行的业务场景：如使用广播执行模式清理集群日志。
- 有需要分布式处理的业务场景：比如需要更新一大批数据，单机执行耗时非常长，可以使用 Map/MapReduce 处理器完成任务的分发，调动整个集群加速计算。
- 有需要**延迟执行**某些任务的业务场景：比如订单过期处理等。

设计目标

PowerJob 的设计目标为**企业级的分布式任务调度平台**，即成为公司内部的**任务调度中间件**。整个公司统一部署调度中心 powerjob-server，旗下所有业务线应用只需要依赖 powerjob-worker 即可接入调度中心获取任务调度与分布式计算能力。

同类产品对比

	QuartZ	xxl-job	SchedulerX 2.0	PowerJob
定时类型	CRON	CRON	CRON、固定频率、固定延迟、OpenAPI	**CRON、固定频率、固定延迟、OpenAPI**
任务类型	内置Java	内置Java、GLUE Java、Shell、Python等脚本	内置Java、外置Java（FatJar）、Shell、Python等脚本	**内置Java、外置Java（容器）、Shell、Python等脚本**
分布式任务	无	静态分片	MapReduce 动态分片	**MapReduce 动态分片**
在线任务治理	不支持	支持	支持	**支持**
日志白屏化	不支持	支持	不支持	**支持**
调度方式及性能	基于数据库锁，有性能瓶颈	基于数据库锁，有性能瓶颈	不详	**无锁化设计，性能强劲无上限**

报警监控	无	邮件	短信	**邮件，提供接口允许开发者扩展**
系统依赖	关系型数据库（MySQL、Oracle...）	MySQL	人民币	**任意 Spring Data Jpa支持的关系型数据库（MySQL、Oracle...）**
DAG 工作流	不支持	不支持	支持	**支持**

基本概念（非常重要，请仔细阅读）

> **本节将阐述本框架所涉及的专有名词概念，帮助开发者更好的理解与使用框架。**

分组概念：

- appName：应用名称，建议与用户实际接入 PowerJob 的应用名称保持一致，用于**业务分组与隔离。一个 appName 等于一个业务集群，也就是实际的一个 Java 项目。**

核心概念：

- 任务（Job）：描述了需要被 PowerJob 调度的任务信息，包括任务名称、调度时间、处理器信息等。
- 任务实例（ JobInstance，简称 Instance）：任务（Job）被调度执行后会生成任务实例（Instance），任务实例记录了任务的运行时信息（任务与任务实例的关系类似于类与对象的关系）。
- 作业（Task）：任务实例的执行单元，一个 JobInstance 存在至少一个 Task，具体规则如下：
  - 单机任务（STANDALONE）：一个 JobInstance 对应一个 Task
  - 广播任务（BROADCAST）：一个 JobInstance 对应 N 个 Task，N为集群机器数量，即每一台机器都会生成一个 Task
  - Map/MapReduce任务：一个 JobInstance 对应若干个 Task，由开发者手动 map 产生
- 工作流（Workflow）：由 DAG（有向无环图）描述的一组任务（Job），用于任务编排。
- 工作流实例（WorkflowInstance）：工作流被调度执行后会生成工作流实例，记录了工作流的运行时信息。

扩展概念

- JVM 容器：以 Maven 工程项目的维度组织一堆 Java 文件（开发者开发的众多 Java 处理器），可以通过前端网页动态发布并被执行器加载，具有极强的扩展能力和灵活性。
- OpenAPI：允许开发者通过接口来完成手工的操作，让系统整体变得更加灵活。开发者可以基于 API 便捷地扩展 PowerJob 原有的功能。

- 轻量级任务：单机执行且不需要以固定频率或者固定延迟执行的任务 （>= v4.2.1）
- 重量级任务：非单机执行或者以固定频率/延迟执行的任务 （>= v4.2.1）

定时任务类型

- API：该任务只会由 powerjob-client 中提供的 OpenAPI 接口触发，server 不会主动调度。
- CRON：该任务的调度时间由 CRON 表达式指定。
- 固定频率：秒级任务，每隔多少毫秒运行一次，功能与 java.util.concurrent.ScheduledExecutorService#scheduleAtFixedRate 相同。
- 固定延迟：秒级任务，延迟多少毫秒运行一次，功能与 java.util.concurrent.ScheduledExecutorService#scheduleWithFixedDelay 相同。
- 工作流：该任务只会由其所属的工作流调度执行，server 不会主动调度该任务。如果该任务不属于任何一个工作流，该任务就不会被调度。

备注：固定延迟和固定频率任务统称秒级任务，这两种任务无法被停止，**只有任务被关闭或删除时才能真正停止任务**。

项目地址

> 求 Star ～OvO～

PowerJob 主项目：https://github.com/PowerJob/PowerJob
PowerJob 前端项目：https://github.com/PowerJob/PowerJob-Console
PowerJob 官网项目：https://github.com/PowerJob/Official-Website

项目结构说明

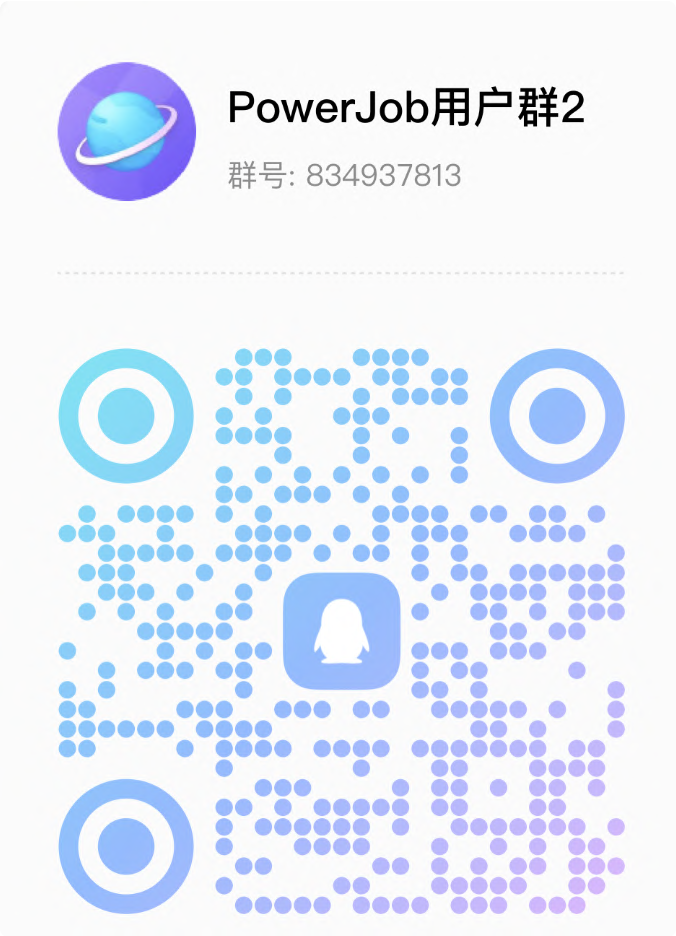
本项目由主体项目（PowerJob）和前端项目（PowerJob-Console）构成，其中，PowerJob各模块说明如下：

```
├── LICENSE
├── powerjob-client // powerjob-client，普通Jar包，提供 OpenAPI
├── powerjob-common // 各组件的公共依赖，开发者无需感知
├── powerjob-remote // 内部通讯层框架，开发者无需感知
├── powerjob-server // powerjob-server，基于SpringBoot实现的调度服务器
├── powerjob-worker // powerjob-worker，普通Jar包，接入powerjob-server的应用需要依赖该Jar包
├── powerjob-worker-agent // powerjob-agent，可执行Jar文件，可直接接入powerjob-server的代理应用
├── powerjob-worker-samples // 教程项目，包含了各种Java处理器的编写样例
├── powerjob-worker-spring-boot-starter // powerjob-worker 的 spring-boot-starter ，spring boot 应用可以通用引入该依赖一键接入 powerjob-server
├── powerjob-official-processors // 官方处理器，包含一系列常用的 Processor，依赖该 jar 包即可使用
├── others
└── pom.xml

```

用户交流与讨论

QQ1群（已满）：487453839

QQ2群：834937813


PowerJob用户群2
群号: 834937813

快速开始（本地IDE版）

> 本教程适用于熟悉项目或搭建本地试用/测试环境，正式部署请阅读**正式部署**章节。

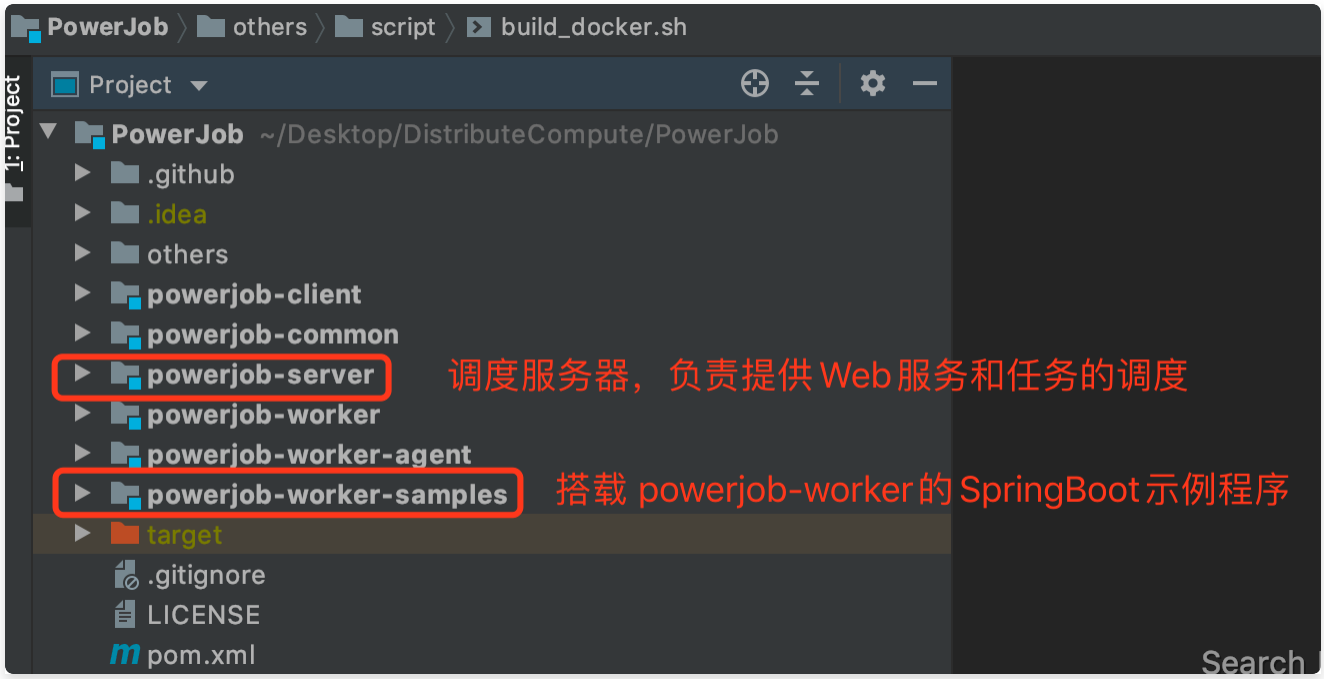
PowerJob 的设计目标为企业级的分布式任务调度平台，即成为公司内部的调度中间件。整个公司统一部署调度中心 powerjob-server，旗下所有业务线应用只需要依赖 powerjob-worker 即可接入获取任务调度与分布式计算能力，通过不同的 appName 相互隔离。

因此，PowerJob 由调度服务器（powerjob-server）和执行器(powerjob-worker)两部分组成，powerjob-server 负责提供 Web 服务和完成任务的调度，powerjob-worker 则负责执行用户所编写的任务代码，同时提供分布式计算能力。

以下为在本地开发环境快速搭建并试用 PowerJob 的教程。

STEP1: 初始化项目

1. `git clone https://github.com/PowerJob/PowerJob.git`
2. 导入 IDE，源码结构如下，我们需要启动调度服务器（powerjob-server），同时在 samples 工程中编写自己的处理器代码



STEP2: 启动调度服务器

1. 创建数据库（仅需要创建数据库）：找到你的 DB，运行 SQL `CREATE DATABASE IF NOT EXISTS \`powerjob-daily\` DEFAULT CHARSET utf8mb4`，搞定～
2. 修改配置文件，配置文件的说明官方文档写的非常详细，此处不再赘述。
   需要修改的地方(路径: PowerJob/powerjob-server/powerjob-server-starter/src/main/resources/application-daily.properties)为**数据库配置** `spring.datasource.core.jdbc-url`、`spring.datasource.core.username` 和 `spring.datasource.core.password`，当然，有 mongoDB 的同学也可以修改 `spring.data.mongodb.uri` 以获取完全版体验。

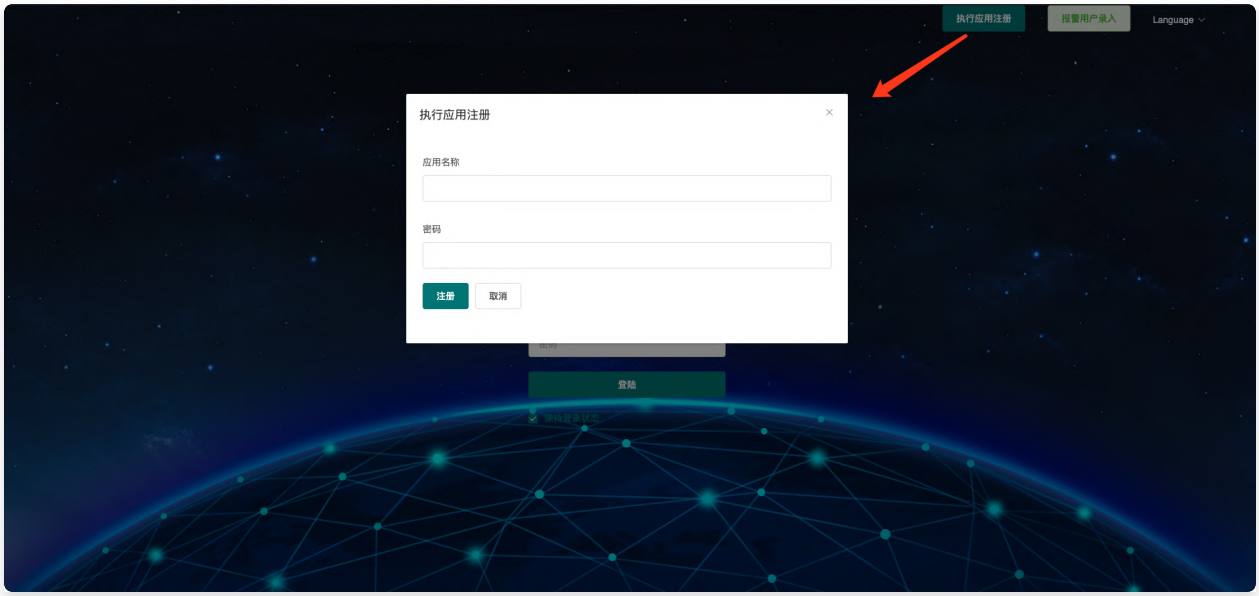
powerjob-server 日常环境配置文件：application-daily.properties

```
oms.env=DAILY
logging.config=classpath:logback-dev.xml

####### 外部数据库配置（需要用户更改为自己的数据库配置） #######
spring.datasource.core.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.core.jdbc-url=jdbc:mysql://localhost:3306/powerjob-daily?useUnicode=true&characterEncoding=UTF-8&serverTimezone=Asia/Shanghai
spring.datasource.core.username=root
spring.datasource.core.password=No1Bug2Please3!
spring.datasource.core.hikari.maximum-pool-size=20
spring.datasource.core.hikari.minimum-idle=5

####### mongoDB配置，非核心依赖，通过配置 oms.mongodb.enable=false 来关闭 ########
oms.mongodb.enable=true
spring.data.mongodb.uri=mongodb://localhost:27017/powerjob-daily

####### 邮件配置（不需要邮件报警可以删除以下配置来避免报错） #######
spring.mail.host=smtp.163.com
spring.mail.username=zqq@163.com
spring.mail.password=GOFZPNARMVKCGONV
spring.mail.properties.mail.smtp.auth=true
spring.mail.properties.mail.smtp.starttls.enable=true
spring.mail.properties.mail.smtp.starttls.required=true

####### 资源清理配置 #######
oms.instanceinfo.retention=1
oms.container.retention.local=1
oms.container.retention.remote=-1

####### 缓存配置 #######
oms.instance.metadata.cache.size=1024

####### 用户与权限体系配置 #######
oms.auth.initiliaze.admin.password=powerjob_admin
```

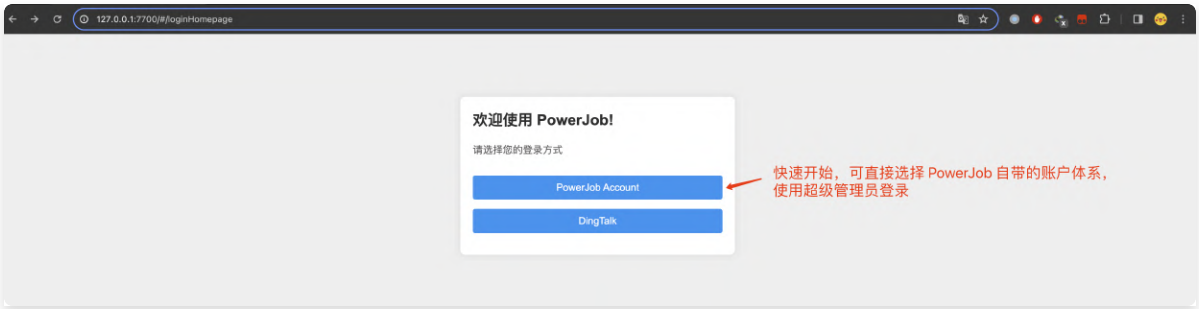
tips：如果你暂时没有可用的数据库进行测试，那么可以使用以下数据库配置来一键启动 server（仅限测试环境使用！）：

```
spring.datasource.core.driver-class-name=org.h2.Driver
spring.datasource.core.jdbc-url=jdbc:h2:file:~/h2/powerjob-daily-test
spring.datasource.core.username=sa
spring.datasource.core.password=
```

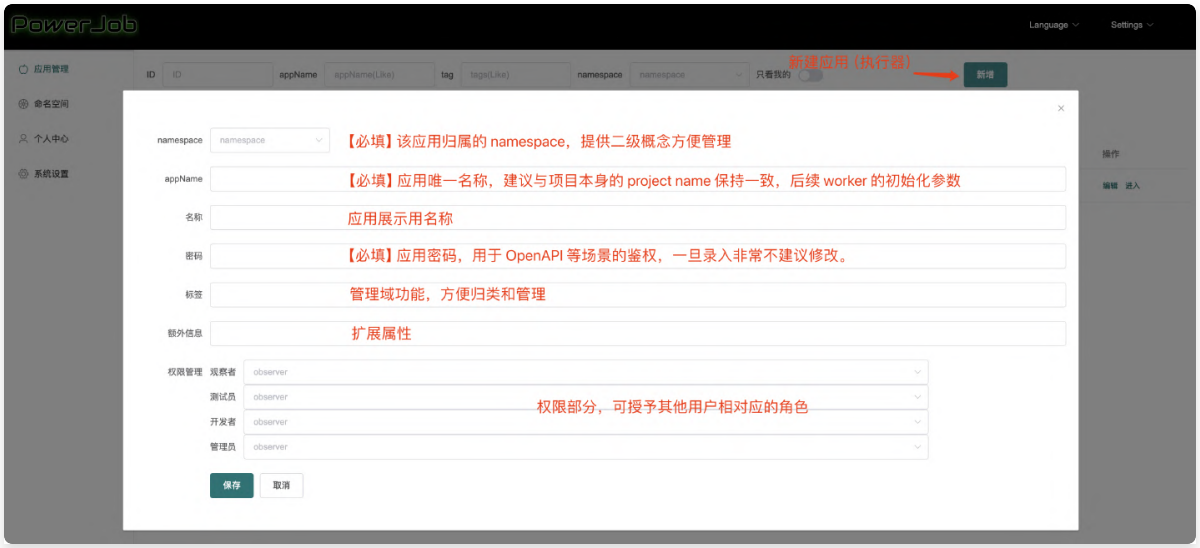
3. 完成配置文件的修改后，可以直接通过启动类 `tech.powerjob.server.PowerJobServerApplication` 启动调度服务器（**注意：需要使用 daily 配置文件启动，可自行百度搜索“SpringBoot 指定配置文件启动”**），观察启动日志，查看是否启动成功～启动成功后，访问 http://127.0.0.1:7700/ ，如果能顺利出现 Web 界面，则说明调度服务器启动成功！
4. 注册应用
   - v4.x及前序版本：点击主页**应用注册**按钮，填入 `powerjob-worker-samples` 和控制台密码（用于进入控制台），注册示例应用（当然你也可以注册其他的 appName，只是别忘记在示例程序中同步修改～）



   - v5.x 版本：
     - 选择 `PowerJob Account` 进行登录，简单起见可直接使用超级管理员登录（账号 ADMIN，密码 powerjob_admin）



- 登录后，进入 `应用管理` TAB，点击右上角 `新增` 进行应用注册即可。



STEP3: 编写示例代码

> 进入示例工程（powerjob-worker-samples），修改配置文件连接powerjob-server并编写自己的处理器代码。

1. 修改 powerjob-worker-samples 的 application.properties，将 **powerjob.worker.app-name** 改为刚刚在控制台注册的名称。

```
server.port=8081

spring.jpa.open-in-view=false

############ powerjob-worker 配置 ############
# akka 工作端口，可选，默认 27777
powerjob.worker.akka-port=27777
# 接入应用名称，用于分组隔离，推荐填写 本 Java 项目名称
powerjob.worker.app-name=powerjob-worker-samples
# 调度服务器地址，IP:Port 或 域名，多值逗号分隔
powerjob.worker.server-address=127.0.0.1:7700,127.0.0.1:7701
# 持久化方式，可选，默认 disk
powerjob.worker.store-strategy=disk
# 任务返回结果信息的最大长度，超过这个长度的信息会被截断，默认 8192
powerjob.worker.max-result-length=4096
# 单个任务追加的工作流上下文最大长度，超过这个长度的会被直接丢弃，默认 8192
powerjob.worker.max-appended-wf-context-length=4096
```

2. 编写自己的处理器：随便找个地方新建类，继承你想要使用的处理器（各个处理器的介绍可见官方文档，文档非常详细），这里为了简单演示，选择使用单机处理器 `BasicProcessor`，以下是代码示例。

```java
@Slf4j
@Component
public class StandaloneProcessorDemo implements BasicProcessor {

    @Override
    public ProcessResult process(TaskContext context) throws Exception {

        // PowerJob 在线日志功能，使用该 Logger 打印的日志可以直接在 PowerJob 控制台查看
        OmsLogger omsLogger = context.getOmsLogger();
        omsLogger.info("StandaloneProcessorDemo start process,context is {}.", context);

        return new ProcessResult(true, "process successfully~");
    }
}
```

3. 启动示例程序，即直接运行主类 `tech.powerjob.samples.SampleApplication`，观察控制台输出信息，判断是否启动成功。

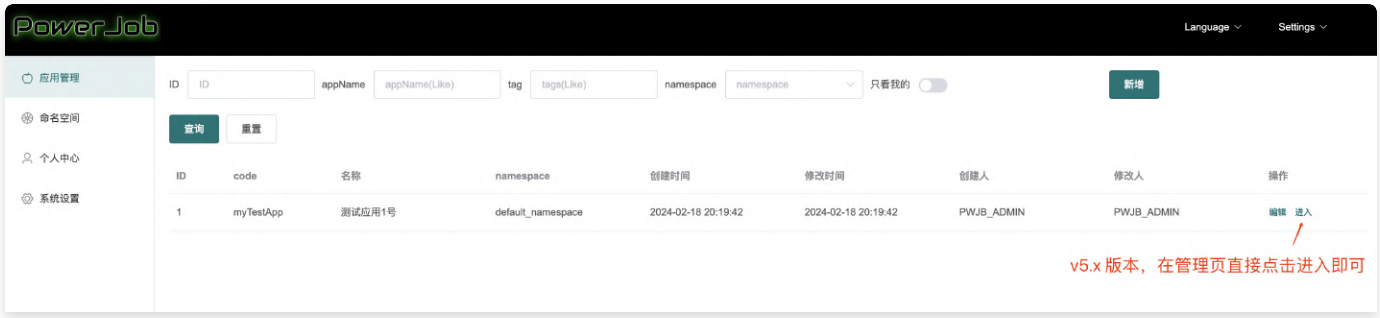
STEP4: 任务的配置与运行

调度服务器与示例工程都启动完毕后，再次前往Web页面（ http://127.0.0.1:7700/ ），进行任务的配置与运行。

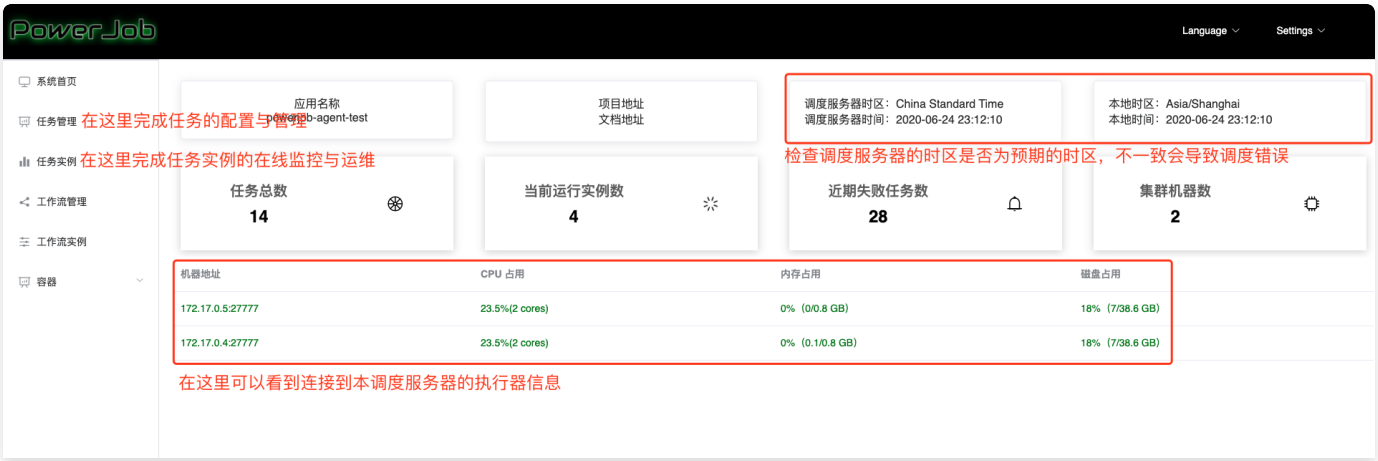
进入执行器应用的主界面

v4.x 及前序版本：在首页输入框输入配置的应用名称，成功操作后会正式进入前端管理界面

v5.x：在 `应用管理` 页面，选择对应的应用，点击右侧“进入”按钮即可。



进入注册应用首页后，出现如下图页面



新建任务

点击“任务管理”–>“新建任务”（右上角），开始创建任务。

可按下图说明填写参数

- 任务名称：名称
- 任务描述：描述
- 任务参数：任务处理时能够获取到的参数（即各个 Processor 的 process 方法入参 `TaskContext` 对象的jobParams 属性）（进行一次处理器开发就能理解了）
- 定时信息：该任务的触发方式，由下拉框和输入框组成
  - API -> 不需要填写任何参数，表明该任务由 OpenAPI 触发，不会被调度器主动调度执行
  - CRON -> 填写 CRON 表达式（在线生成网站）
  - 固定频率 -> 任务以固定的频率执行，填写整数，单位**毫秒**
  - 固定延迟 -> 任务以固定的延迟执行，填写整数，单位**毫秒**
  - 每日固定间隔 -> 哪几天的哪些时间段需要执行，比如每周二和三的10点到11点间每10分钟触发一次
  - 工作流 -> 不需要填写任何参数，表明该任务由工作流（workflow）触发
- 生命周期：任务的有效时间范围，不在生命周期的任务会自动停止。常用于任务的延迟启动和过期自动关停。
- 执行配置：由执行类型（单机、广播和 MapReduce ）、处理器类型和处理器参数组成，后两项相互关联。
  - 内置Java处理器
    - 方式一 -> 填写该处理器的**全限定类名**（eg，`tech.powerjob.samples.processors.MapReduceProcessorDemo`）
    - 方式二 -> 填写 IOC 容器的 bean 名称，比如 Spring 用户可填写 Spring Bean 名称

（eg，处理器使用注解 `@Component(value = "powerJobProcessor")`，则控制台可填写 powerJobProcessor）

- 方式三 -> 方法级注解，非 MapReduce 任务可直接使用注解 `@PowerJobHandler` 将某个方法转化为 PowerJob 任务，并设置唯一入参 `TaskContext` 注入上下文，具体参考代码

```java
@Component(value = "springMethodProcessorService")
public class SpringMethodProcessorService {

    /**
     * 处理器配置方法1： 全限定类名#方法名，比如 tech.powerjob.samples.tester.SpringMethodProcessorService#testEmptyReturn
     * 处理器配置方法2： SpringBean名称#方法名，比如 springMethodProcessorService#testEmptyReturn
     * @param context 必须要有入参 TaskContext，返回值可以是 null，也可以是其他任意类型。正常返回代表成功，抛出异常代表执行失败
     */
    @PowerJobHandler(name = "testEmptyReturn")
    public String testEmptyReturn(TaskContext context) {
        OmsLogger omsLogger = context.getOmsLogger();
        omsLogger.warn("测试日志");
        return "响应结果，正常返回视为执行成功，抛出异常视为执行失败"
    }
}
```

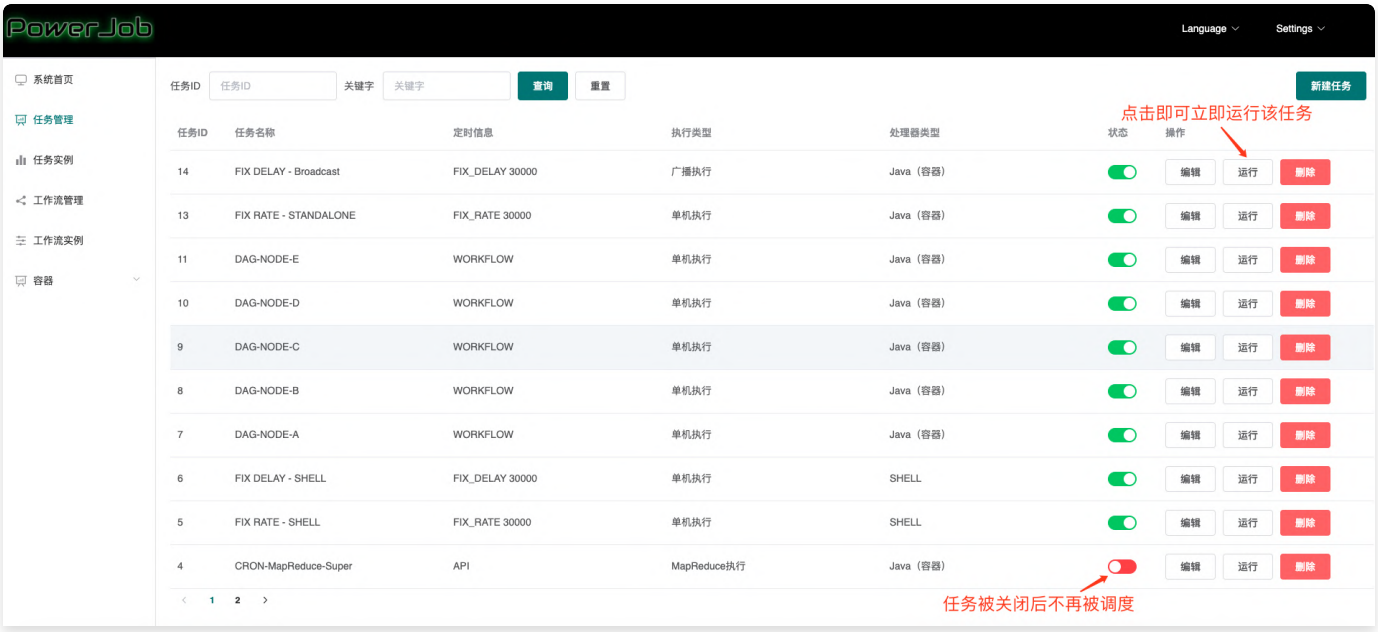
  - Java容器 -> 填写**容器ID#处理器全限定类名**（eg，`1#cn.edu.zju.oms.container.ContainerMRProcessor`）
  - SHELL、Python、SQL 、HTTP 等任务的执行：点击查看官方处理器的使用教程

- 运行配置
  - 派发策略：默认健康度优先，优先选择性能最优机器进行执行，可选随机均摊等其他派发模式
  - 最大实例数：该任务同时执行的数量，0 代表不限制实例数量
  - 单机线程并发数：该实例执行过程中每个 Worker 使用的线程数量（MapReduce 任务生效，其余无论填什么，都只会使用必要的线程数）
  - 运行时间限制：限定任务的最大运行时间，超时则视为失败，单位**毫秒**，0 代表不限制超时时间（**不建议不限制超时时间**）。
- 重试配置：
  - Instance 重试次数：实例级别，失败了整个任务实例重试，会更换 TaskTracker（本次任务实例的Master节点），代价较大，大型Map/MapReduce慎用。
  - Task 重试次数：Task 级别，每个子 Task 失败后单独重试，会更换 ProcessorTracker（本

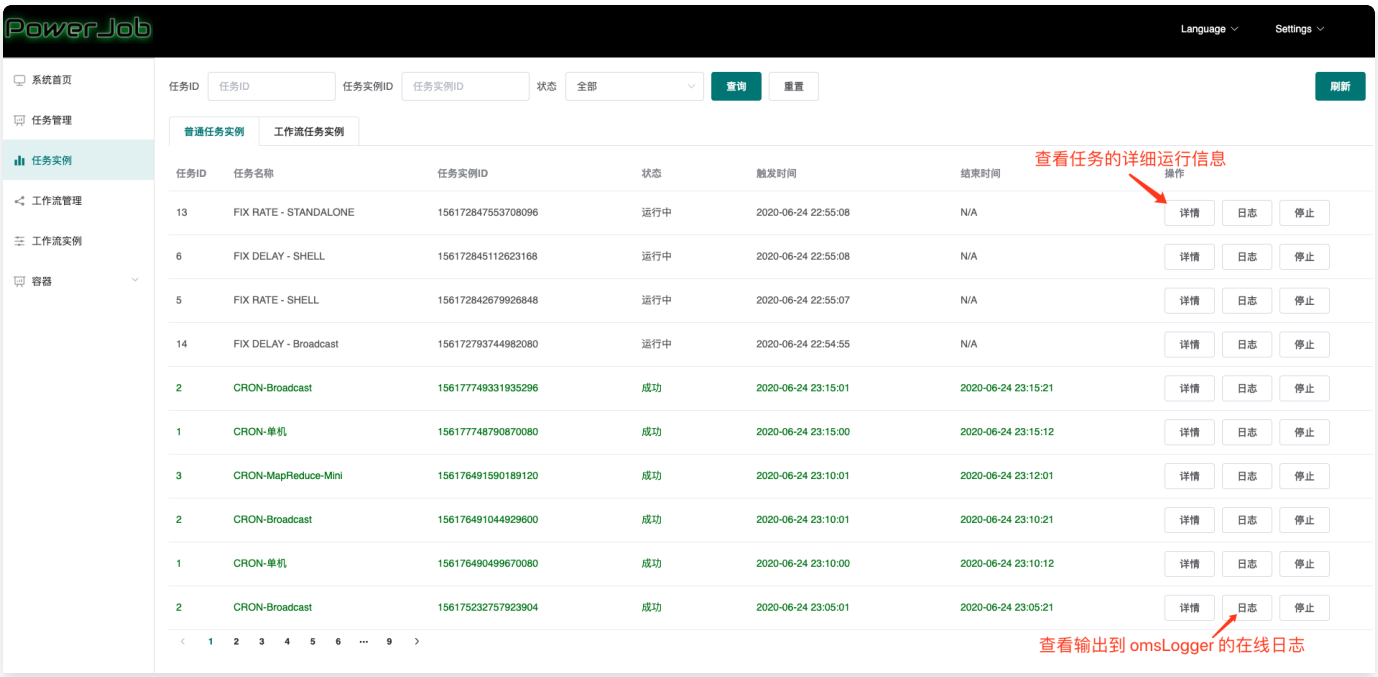
次任务实际执行的 Worker 节点），代价较小，推荐使用。
    - 注：**请注意同时配置任务重试次数和子任务重试次数之后的重试放大**，比如对于单机任务来说，假如任务重试次数和子任务重试次数都配置了 1 且都执行失败，实际执行次数会变成 4 次！推荐任务实例重试配置为 0，子任务重试次数根据实际情况配置。
- 机器配置：用来标明允许执行任务的机器状态，避开那些摇摇欲坠的机器，0 代表无任何限制。
    - 最低 CPU 核心数：填写浮点数，CPU 可用核心数小于该值的 Worker 将不会执行该任务。
    - 最低内存（GB）：填写浮点数，可用内存小于该值的 Worker 将不会执行该任务。
    - 最低磁盘（GB）：填写浮点数，可用磁盘空间小于该值的 Worker 将不会执行该任务。
- 集群配置
    - 执行机器地址，指定集群中的某几台机器执行任务
        - IP模式：多值英文逗号分割，如 `192.168.1.1:27777,192.168.1.2:27777` 。常用于 debug 等场景，需要指定特定机器运行。
        - TAG 模式：通过 `PowerJobWorkerConfig#tag` 将执行器打标分组后，可在控制台通过 tag 指定某一批机器执行。常用于分环境分单元执行的场景。如某些任务需要屏蔽安全生产环境（tag 设置为环境标），某些任务只需要在特定单元执行（tag 设置单元标）
    - 最大执行机器数量：限定调动执行的机器数量
- 报警配置：选择任务执行失败后报警通知的对象，需要事先录入。
- 日志配置：可使用控制台配置调整 Job 使用的 Logger 及 LogLevel
    - 支持 SERVER（服务端日志，默认）、LOCAL（本地日志）、STDOUT（系统输出）、NULL（空实现）4种 LogType
    - 支持 DEBUG、INFO、WARN、ERROR、OFF 5种级别控制
    - 使用建议：初期调试可使用 SERVER 日志，后续功能稳定后改为 LOCAL，并调高日志级别，降低通讯压力，消除性能瓶颈问题

任务运行&运维

完成任务创建后，即可在控制台看到刚才创建的任务，如果觉得等待调度太过于漫长，可以直接点击**运行**按钮，立即运行本任务。



前往**任务实例**边栏，查看任务的运行状态和在线日志



基础的教程到这里也就结束了～

更多功能示例可见官方文档，工作流、MapReduce、容器等高级特性等你来探索！

Others

PowerJob 部署视频教程（server + agent + 容器）：点击查看

快速开始（docker-compose版）

环境要求

本地需要安装 `docker` 和 `docker-compose`

1. 下载项目

Plain Text

```
git clone --depth=1 https://github.com/PowerJob/PowerJob.git
```

2. 运行

方式1: 一键脚本

Shell

```
# 进入 PowerJob 测试环境脚本所在目录
cd PowerJob/others/dev
# 为脚本授权
chmod 755 build_test_env.sh
# 运行脚本，会在本地自动执行构建并部署一个 server + 2个 worker 实例
./build_test_env.sh
```

方式2: 手动模式

在正式运行之前，首先需要删除低版本 powerjob 相关依赖

一键删除命令 Plain Text

```
docker rmi $(docker images | grep "powerjob" | awk '{print $3}')
```

进入到 `PowerJob` 工作目录

```
cd PowerJob

# 前台运行（初次运行时，推荐使用该方式，方便实时查看日志，排查问题）
docker-compose up

# 后台运行
docker-compose up -d
```

刚开始启动时，`powerjob-worker-samples` 会启动失败，等 `powerjob-server` 启动成功后，`powerjob-worker-samples` 才会启动成功。这大概需要几分钟。

启动成功

运行成功后，浏览器访问 http://127.0.0.1:7700/

应用名称：powerjob-worker-samples

密码：powerjob123

任务配置请参考：

此处为语雀内容卡片，点击链接查看：https://www.yuque.com/powerjob/guidence/nyio9g?view=doc_embed&inner=v8uF4

3. 停止

```
docker-compose down
Stopping powerjob-worker-samples ... done
Stopping powerjob-server         ... done
Stopping powerjob-mysql          ... done
Removing powerjob-worker-samples ... done
Removing powerjob-server         ... done
Removing powerjob-mysql          ... done

# 删除数据目录
cd PowerJob
rm -rf powerjob-data
```

4. 进阶使用

docker-compose方式运行，会创建3个容器：

1. **powerjob-mysql**：存储 PowerJob 服务端运行时数据，启动时会自动创建 `powerjob-daily` 数据库，用户名： `root` ，密码： `No1Bug2Please3!` ；
2. powerjob-server： `PowerJob` 服务端，源码路径：PowerJob/powerjob-server/powerjob-server-starter；
3. powerjob-worker-samples：PowerJob 提供的 Worker 示例，源码路径：PowerJob/powerjob-worker-samples。

相关文件路径说明：

Plain Text

```
PowerJob
├── powerjob-data
├────── powerjob-mysql                -- MySQL数据目录
├────── powerjob-server               -- server h2目录
├────── powerjob-worker-samples       -- worker h2目录
docker-compose.yml
```

调度中心（powerjob-server）部署

PowerJob 的设计目标为企业级的分布式任务调度平台，即成为某个公司的调度中间件，该公司下任意业务线的应用仅需要依赖 powerjob-worker 即可获取任务调度与分布式计算的能力。因此，**PowerJob 的理想部署模式为一个公司统一部署 powerjob-server 集群，各业务线应用直接接入使用。**

在 Spring Data JPA 和 Docker 技术的加持下，调度中心的部署极其简单，**预计耗时5分钟！**

环境要求

- JDK 8 或 JDK 11 或 JDK17（三个 LTS 版本）
  - 高版本 JDK 需要自行加入 `javax.xml.bind` 等依赖，详细文档可见LINK
- **任意 Spring Data Jpa 支持的关系型数据库**（MySQL/PostgreSQL/Oracle/MS SQLServer...）
  - PostgreSQL 启动报错可见 LINK
  - 达梦等数据库可自行百度“JPA连接达梦数据库”，简单适配即可
- 【可选】MongoDB、OSS、数据库等存储扩展，主要用于实现在线日志、容器部署等扩展功能。不会影响核心的调度能力，缺失情况下将无法使用完全版的在线日志、容器部署等扩展功能（无法做到容器与日志数据的高可用，如果 server 单实例部署则无任何影响）

STEP1: 初始化数据库

> 调度服务器（powerjob-server）的持久化层基于Spring Data Jpa实现，对于能够直连数据库的应用，开发者仅需完成数据库的创建，即运行SQL：`CREATE DATABASE IF NOT EXISTS `powerjob-product` DEFAULT CHARSET utf8mb4`

- PowerJob支持**环境隔离**，提供日常（daily）、预发（pre）和线上（product）三套环境，请根据使用的环境分别部署对应的数据库 **powerjob-daily**、**powerjob-pre** 和 **powerjob-product**。
  - 部分数据库（如 Oracle）会将 `-` 作为非法字符，因此建议使用 **`powerjob_daily`** **作为数据库名称**
- 如有手动建立数据库表结构需求，可使用 SQL 文件：下载地址
  - 注意：官方 SQL 仅基于特定版本（MySQL8）导出，不一定兼容其他数据库，也不一定兼容其他版本。由于 SpringDataJPA 自带了建表能力，推荐先在开发环境直连测试库自动建表，然后自行导出相关的 SQL 即可。

STEP2: 部署调度服务器（powerjob-server）

> 调度服务器（powerjob-server）支持任意的水平扩展，即多实例集群部署仅需要在同一个局域网内启动新的服务器实例，性能强劲无上限！

配置讲解

调度服务器（powerjob-server）为了支持环境隔离，分别采用了日常（**application-daily.properties**）、预发（**application-pre.properties**）和线上（**application-product.properties**）三套配置文件，请根据实际需求进行修改，以下为配置文件详解。

配置项	含义	可选
server.port	SpringBoot配置，HTTP端口号，默认7700	否，且不建议更改
oms.transporter.active.protocols	server 需要激活的通讯协议，建议激活全部支持的协议来支持各种 worker 连接	否，且不建议更改
oms.transporter.main.protocol	主要通讯协议，用于 server 与 server 之间的通讯，**用户必须保证该协议可用（端口开放）！**	否
oms.akka.port	PowerJob配置，Akka端口号，默认10086	否，且不建议更改
oms.http.port	PowerJob配置，多语言客户端HTTP端口号，默认10010	否，且不建议更改
oms.table-prefix	自定义数据库表名前缀	是
spring.datasource.core.xxx	关系型数据库连接配置	否
spring.mail.xxx	邮件配置	是，未配置情况下将无法使用邮件报警功能
oms.container.retention.local	本地容器保留天数，负数代表永久保留	是
oms.container.retention.remote	远程容器保留天数，负数代表永久保留	是

oms.instanceinfo.retention	任务实例和工作流实例信息的保留天数，负数代表永久保留（不建议）	是，推荐使用默认配置，生产环境保留7天
oms.auth.initiliaze.admin.password	【用户与权限】系统初始化时默认创建的超级管理员密码	是，默认值 powerjob_admin，无此配置时，随机生成密码
oms.auth.dingtalk.appkey oms.auth.dingtalk.appSecret oms.auth.dingtalk.callbackUrl	【用户与权限】钉钉用户账号体系相关配置内容。详细配置请见： 此处为语雀内容卡片，点击链接查看： https://www.yuque.com/powerjob/guidence/user?view=doc_embed&inner=ZloYU	否，仅启用钉钉账号登录体系时需要配置

端口说明：

最省事的方法：所有端口（7700 + 10086 + 10010）全打开。如果你想精细化控制端口，那么请遵循以下原则自行设置：

1. 对于任何用户，7700 为**调度服务器（powerjob-server）的 Web 服务端口，必须打开**
2. `oms.协议.port` 的端口按需打开，考虑 server-server 和 server-worker 通讯的场景
    - 比如 server-server 默认通过 HTTP 协议交互（参数 `oms.transporter.main.protocol` 控制），那必须打开 HTTP 10010 端口
    - 同时 server-worker 部分通过 HTTP，部分通过 AKKA，则仍需要打开 AKKA 的 10086 端口

注: 调度服务器部署完成后可访问 http://ip:${server.port} 检验是否部署成功!

存储扩展配置

> PowerJob 当前支持多套存储（MongoDB、AliyunOSS、MySQL等数据库），接入方可自由选择合适的存储介质。

MongoDB

> 4.3.4 版本前的主力存储，推荐使用

配置项：

- oms.storage.dfs.mongodb.uri：mongoDB 连接的 uri，如 `mongodb+srv://zqq:No1Bug2Please3!@cluster0.wie54.gcp.mongodb.net/powerjob_daily?retryWrites=true&w=majority`
- spring.data.mongodb.uri：兼容老版本，依旧支持，作用与 oms.storage.dfs.mongodb.uri 一致

AliyunOSS

> AliyunOSS 存储支持，海量、安全、低成本、高可靠的云存储服务

配置项：

- oms.storage.dfs.alioss.endpoint：阿里云 OSS 的 endpoint
- oms.storage.dfs.alioss.bucket：阿里云 OSS 的存储 bucket
- oms.storage.dfs.alioss.credential_type：密钥类型，支持以下枚举
  - PWD：账号密码，通过 ak sk 传入密钥
  - SYSTEM_PROPERTY：通过系统参数传入密钥，底层为 `com.aliyun.oss.common.auth.SystemPropertiesCredentialsProvider`
  - ENV：通过环境变量 OSS_ACCESS_KEY_ID 和 OSS_ACCESS_KEY_SECRET 读取密钥，底层为 `com.aliyun.oss.common.auth.EnvironmentVariableCredentialsProvider`
- oms.storage.dfs.alioss.ak：密钥类型为 PWD 时需要
- oms.storage.dfs.alioss.sk：密钥类型为 PWD 时需要
- oms.storage.dfs.alioss.token：密钥类型为 PWD 时需要，可选，如果你不知道这是啥，就不需要传

MySQL系列数据库

> 针对简单场景，可直接使用数据库进行日志存储，方便快捷。
> 如果使用数据库作为生产环境的存储库，至少隔离数据库（即不直接前往创建的数据库，为存储单独创建一个独立库），防止非核心场景异常影响调度主链路。

PS. 官方使用的开发环境为 MySQL8（换句话说，别的 DB 版本没测过，使用前请做好测试哦～
配置项：

- oms.storage.dfs.mysql_series.driver：驱动名称，如 `com.mysql.cj.jdbc.Driver`
- oms.storage.dfs.mysql_series.url：JDBC URL，如 `jdbc:mysql://localhost:3306/powerjob-daily?useUnicode=true&characterEncoding=UTF-8&serverTimezone=Asia/Shanghai`
- oms.storage.dfs.mysql_series.username：数据库用户名，如 root
- oms.storage.dfs.mysql_series.password：数据库密码

- oms.storage.dfs.mysql_series.auto_create_table：可选，是否自动建表
- oms.storage.dfs.mysql_series.table_name：可选，数据库表名称，默认为 `powerjob_files`

附录：数据库表 schema 要求（以 MySQL8 为例）

```SQL
CREATE TABLE IF NOT EXISTS powerjob_files (
    `id` BIGINT NOT NULL AUTO_INCREMENT COMMENT 'ID',
    `bucket` VARCHAR ( 255 ) NOT NULL COMMENT '分桶',
    `name` VARCHAR ( 255 ) NOT NULL COMMENT '文件名称',
    `version` VARCHAR ( 255 ) NOT NULL COMMENT '版本',
    `meta` VARCHAR ( 255 ) COMMENT '元数据',
    `length` BIGINT NOT NULL COMMENT '长度',
    `status` INT NOT NULL COMMENT '状态',
    `data` LONGBLOB NOT NULL COMMENT '文件内容',
    `extra` VARCHAR ( 255 ) COMMENT '其他信息',
    `gmt_create` DATETIME NOT NULL COMMENT '创建时间',
    `gmt_modified` DATETIME COMMENT '更新时间',
  PRIMARY KEY ( id )
);
```

方式一：Docker（推荐）

基础版流程：

1. 下载镜像（最新版本请参考 Docker Hub）：`docker pull powerjob/powerjob-server:latest`
2. 创建容器并运行（仅需修改行 7，即传入 SpringBoot 相关配置信息），示例如下

```Shell
docker run -d \
       --restart=always \
       --name powerjob-server \
       -p 7700:7700 -p 10086:10086 -p 10010:10010 \
       -e TZ="Asia/Shanghai" \
       -e JVMOPTIONS="" \
       -e PARAMS="--spring.profiles.active=product --spring.datasource.core.jdbc-url=jdbc:mysql://192.168.1.1:3306/powerjob-product?useUnicode=true&characterEncoding=UTF-8 --spring.datasource.core.username=root --spring.datasource.core.password=root --spring.data.mongodb.uri=mongodb://192.168.1.1:27017/powerjob-product" \
```

行1: docker run -d ： 创建 docker 容器，-d 参数指定为后台运行
行2：指定该容器随着 docker 启动而自启
行3：指定容器名称
行4：指定宿主机与容器的端口映射关系（默认使用 Bridge 网络模式）
行5：时区，默认时区为中国标准时区（Asia/Shanghai），国外用户请修改为正确的时区
行6：JVM启动参数，有需求的可以自己添加（-Xmx、-Xms、-Xmn等）
行7：通过 `-e Params=""` 传入 SpringBoot 启动参数，详细参数见上表（最小配置为示例参数，需要传入配置文件名称、JDBC_URL、数据库用户名、数据库密码和 mongoDB 的 URI）
行8：映射数据卷，**强烈建议映射/root/powerjob/server和/root/.m2这两个路径**，前者存储了powerjob-server所有的运行时文件（日志），后者可以避免使用 maven 编译容器时重复下载依赖的问题（详见容器章节）。
行9：指定使用的 docker 镜像

特殊需求（maven 私服）版流程：

> 仅需要使用Git容器功能且搭建了maven私服的场景需要进行此流程。

容器功能需要用到 Maven 来编译代码库，而公司内部往往都会搭建自己的私有仓库，为此，官方 docker 镜像所使用的 maven 无法下载编译所需要的依赖包。对于这种需求的用户，需要您自己构建 docker 镜像，别担心，我们提供了**一键构建脚本**来简化流程。具体步骤如下：

1. `git clone https://github.com/PowerJob/PowerJob.git`，下载本项目源码。
2. 修改 **powerjob-server/docker/settings.xml** 文件（加入私服配置参数等等）。
3. 运行 docker 一键构建脚本 others/script/build_docker.sh，按照提示即可完成自定义 docker 镜像的制作。

方式二：源码部署

1. 克隆：`git clone https://github.com/PowerJob/PowerJob.git`，下载本项目源码。
2. 修改对应**环境**的配置文件（application-xxx.properties）。
3. 打包：`mvn clean package -U -Pdev -DskipTests`，构建调度服务器（powerjob-server）Jar 文件。
4. 运行：`java -jar xxx.jar --spring.profiles.active=product`，指定生效的配置文件。注意，宿主机需要打开 7700（HTTP 服务）和 10086（AKKA 服务）端口。

STEP3（可选）：部署前端页面（powerjob-console）

> 每一个powerjob-server都自带了前端页面，不过 ~~Tomcat~~（为了完善的 WebSocket支持，现已切换到Undertow ）做 Web 服务器的性能就 ~~呵呵子~~（看评测好像还行，不过对于集群部署调度中心的用户还是建议单独使用源码部署）～

源码部署

1. 源码克隆：`git clone https://github.com/PowerJob/PowerJob-Console.git`
2. 替换地址：修改 `.env.product` 中的 `VUE_APP_BASE_URL` 为调度服务器地址
3. npm install && npm run build
4. 将构建结果 `dist` 文件夹拷入 Nginx 静态目录下，修改配置文件，重启 nginx，enjoy～

PS. 开发使用的 node 版本为 `v14.20.0`

Docker 部署

> 考虑到前端工程 Docker 化并不流行和实用（静态文件存 CDN 它不香嘛），目前暂未提供 Docker 镜像（有一部分原因是本人不会运行时动态替换容器的实际后端请求地址，如果有能力可以帮忙实现的话非常感谢，附前端项目地址），为此，需要开发者手动构建 Docker 镜像，当然，一键构建脚本奉上～

1. 项目克隆：`git clone https://github.com/PowerJob/PowerJob-Console.git`
2. 替换地址：修改 `.env.product` 中的 `VUE_APP_BASE_URL` 为调度服务器地址
3. `chmod 755 script/docker/build.sh && bash script/docker/build.sh`

STEP4: 验证 & 注册应用

V5.x

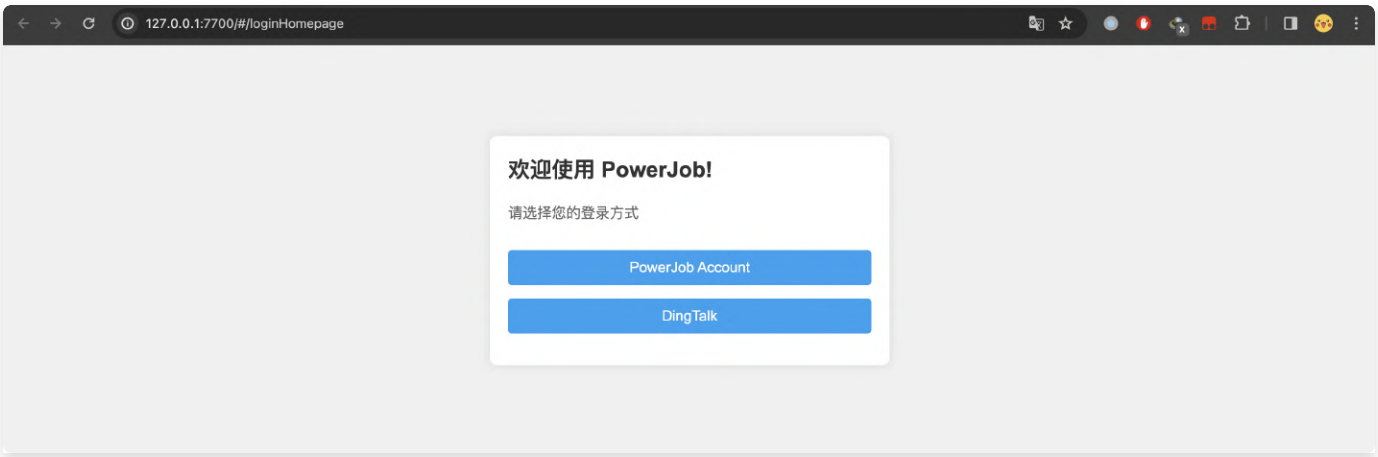
> PowerJob 5.x 后正式支持用户权限体系，通过引入用户身份、管理员、namespace 等概念，帮助接入方提升App 管理效率。

注意：如果是首次部署启动，PowerJob 默认会创建一个超级管理员账号，默认密码由配置项 `oms.auth.initiliaze.admin.password` 指定（默认为 powerjob_admin，若无此配置项则随机生成密码）。

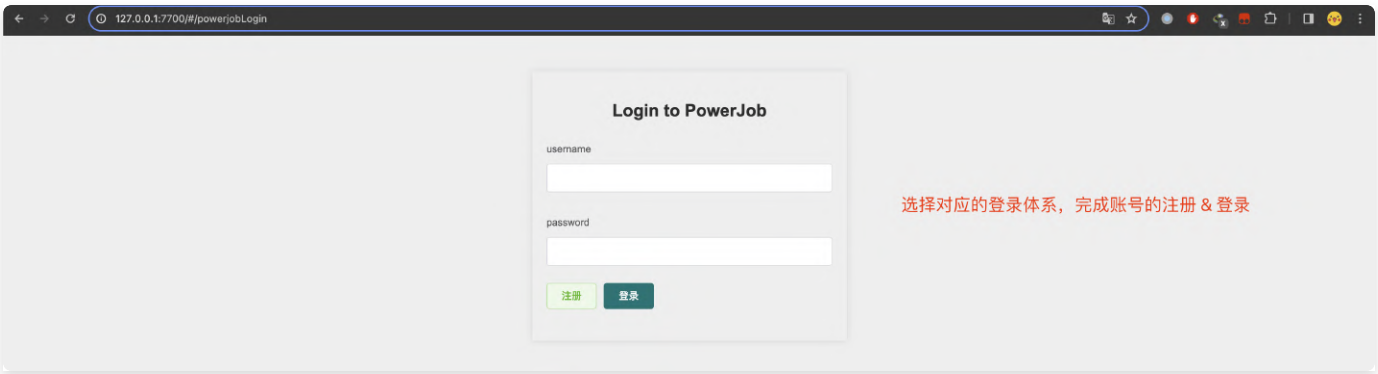
初次部署，请您务必查看 powerjob-server 的输出日志，按关键词 `SystemInitializeService` 搜索，便可看到如下日志，获取到超级管理员的账号密码。

```
[SystemInitializeService] The system has automatically created a super administrator account[username=ADMIN,password=powerjob_admin], please log in and change the password immediately!
```

自 5.x 版本开始，使用者需要登录后才可进入 PowerJob 控制台。您可选择合适的登录方式进行登录（登录方式的选择、第三方账号体系的支持可见文档）



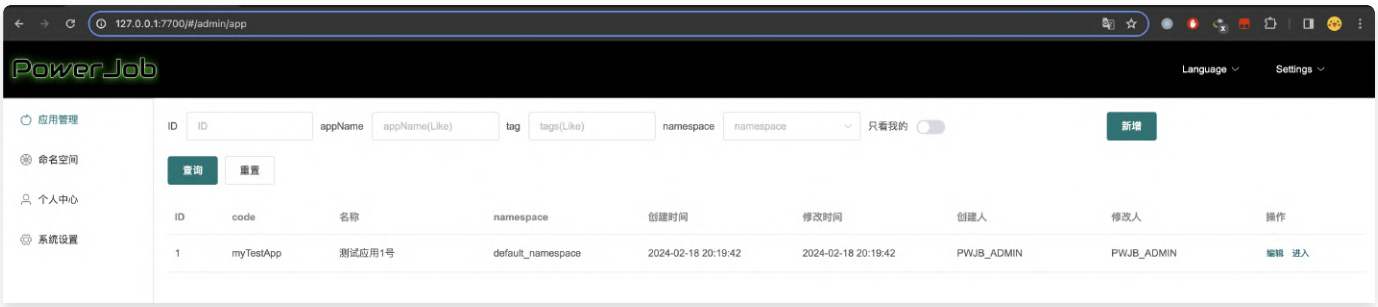
选择对应的账号体系，完成账号的注册和登录。



完成登录后，进入一级管理页面，包括以下 TAB：

- 应用管理：可在此完成执行器的创建和管理，提供全局视角查看当前所有已注册的应用。
- 命名空间：namespace，用于对应用进行分类和管理，比如按部门创建 namespace，同一个部门的应用挂载到同一个 namespace 中，方便管理。
- 个人中心：对本账号进行管理，包括基础信息的查看修改，应用权限的申请等

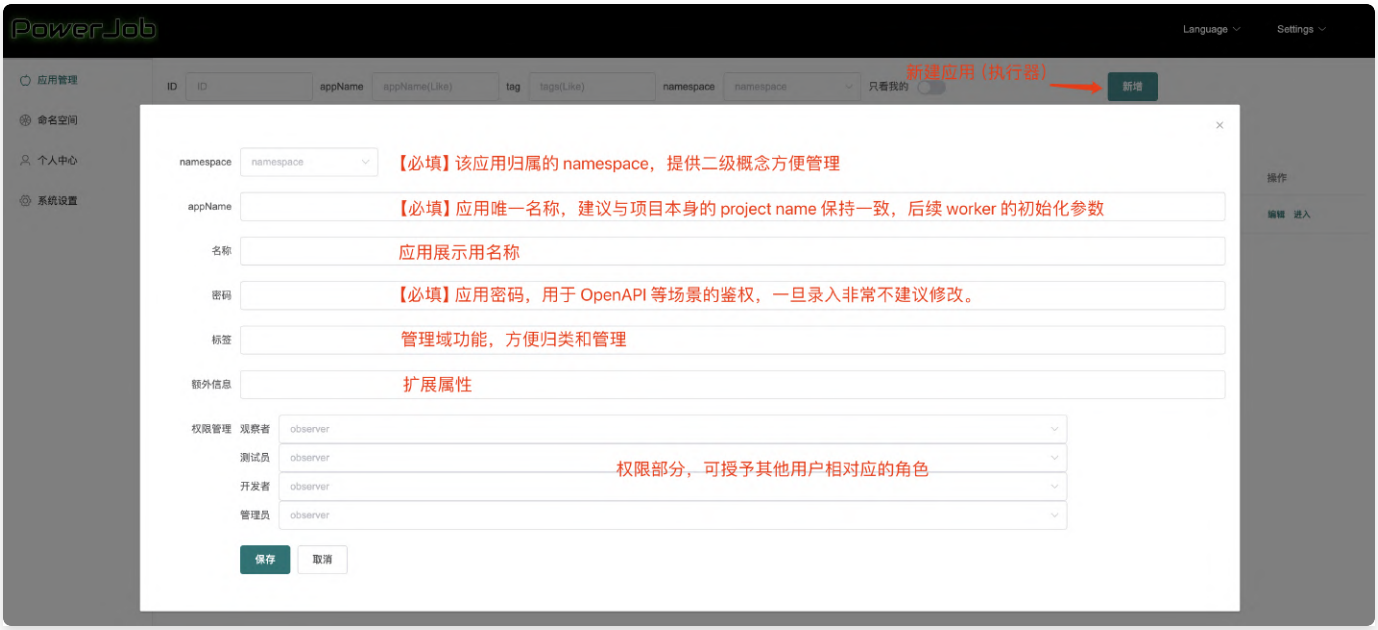
- 系统设置：提供全局超级管理员对 powerjob-server 的一些管理能力



以下为各功能的详细介绍：

应用管理：

- 应用注册：点击右上角“新增”按钮，可拉起应用注册弹窗。每一个业务系统初次接入 PowerJob 时，都需要**先完成应用注册**，即将需要接入的应用名称（appName）录入到调度中心。相关参数介绍如下：
  - namespace：【必填】命名空间，该应用归属的 namespace，提供二级概念方便管理。
  - appName：【必填】注册应用的唯一 code，推荐与项目本身的 project name 保持一致，如 `powerjob-worker-samples`
  - 名称：中文描述，仅用于展示
  - 密码：【必填】用于 OpenAPI 等场景的鉴权
  - 标签：TAG 功能，管理域功能，方便用户进行归类和管理
  - 额外信息：扩展属性，powerjob 本身不消费该字段，方便开发者二次开发预留
  - 权限管理：授予其他用户该 APP 对应的角色，使其拥有相关的权限进行操作。

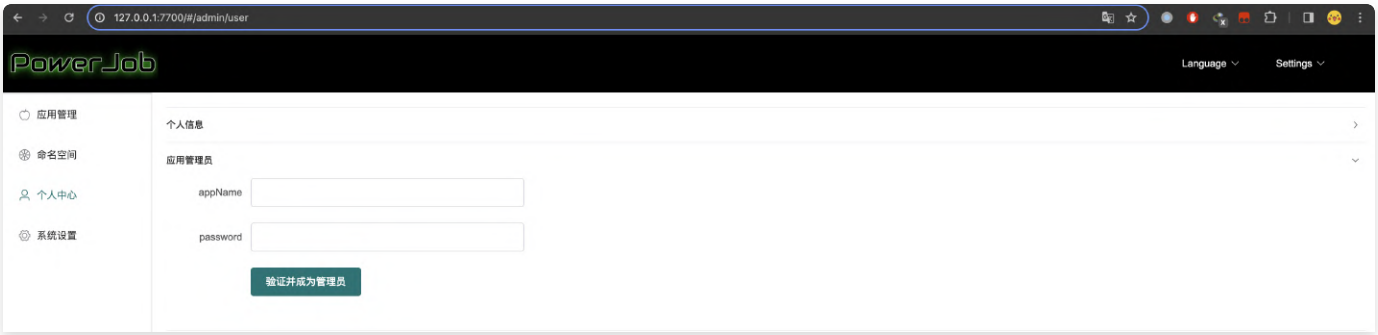


命名空间：

- 创建命名空间：点击右上角“新增”按钮，即可创建新的命名空间：
  - code：【必填】命名空间唯一标识，推荐纯英文，要求全局唯一
  - 名称：中文描述，仅用于展示
  - token：服务端自动生成的访问密钥，用于后续的 OpenAPI 场景
  - 标签：TAG 功能，管理域功能，方便用户进行归类和管理
  - 额外信息：扩展属性，powerjob 本身不消费该字段，方便开发者二次开发预留
  - 权限管理：授予其他用户该 namespace 对应的角色，使其拥有相关的权限进行操作。
    注意：**namespace 下的角色会传递到旗下所有的 APP，比如 namespace 的管理员会应用旗下全部 APP 的管理员权限。**

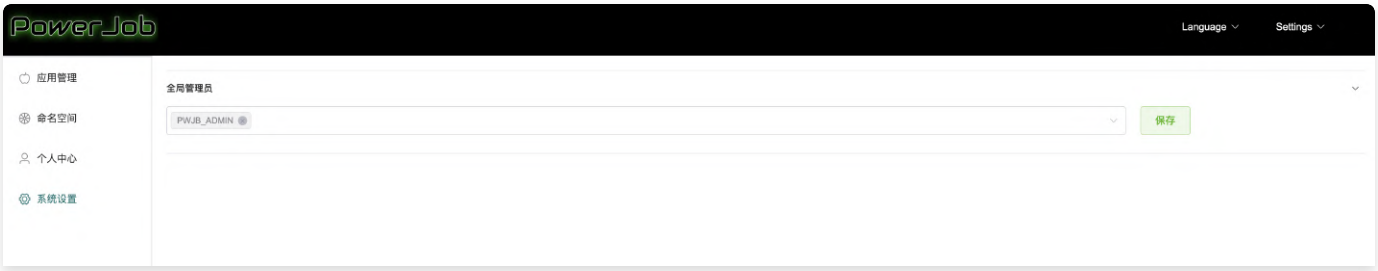
个人中心：

- 个人信息：查看并管理个人信息，可进行修改 nick、密码等操作
- 应用管理员：通过验证某个 APP 的账号密码，使当前账号获取到某个 APP 的管理员权限。



系统设置：

- 全局管理员：授予其他账号全局管理员。针对使用其他账户体系的用户，建议部署完成后，使用 PowerJob 自动生成的 ADMIN 账户将其他账户追授为全局管理员，然后禁用 PowerJob 账号登录，可保证绝对安全性（登录鉴权完全托管在自己的系统上）。



V4.x 及前序版本

每一个业务系统初次接入 PowerJob 时，都需要**先完成应用注册**，即将需要接入的应用名称（appName）录入到调度中心。

点击 **执行应用注册** 按钮，填入**应用名称（appName，重要，需要保证唯一，推荐与真实应用名称保持一致）**和**密码（进入控制台的密码）**，如果顺利完成注册，说明调度服务器和前端页面部署成功！

注册成功后，输入应用名称和密码，即可进入控制台，享受分布式调度与计算的快感～

报警用户录入 按钮用于录入用户信息（姓名、手机号、邮箱地址），收集报警信息，用户注册录入个人信息后，即可通过报警配置进行通知。

执行器（powerjob-worker）初始化

基于宿主应用的初始化

> 宿主应用即原有的业务应用，假如需要调度执行的任务与当前业务有较为紧密的联系，建议采取该方式。

SpringBoot 应用接入方式

⚠️：如果 SpringBoot 存在版本兼容性问题，可使用代码手动初始化的方式：LINK

首先，添加相关的依赖，最新依赖版本请参考 maven 中央仓库：推荐地址 & 备用地址

XML

```xml
<dependency>
    <groupId>tech.powerjob</groupId>
    <artifactId>powerjob-worker-spring-boot-starter</artifactId>
    <version>${latest.powerjob.version}</version>
</dependency>
```

随后，在 SpringBoot 配置文件（application.yml/properties）中添加相关的配置项。

```
# akka 工作端口，可选，默认 27777
powerjob.worker.akka-port=27777
# 接入应用名称，用于分组隔离，推荐填写 本 Java 项目名称
powerjob.worker.app-name=my-powerjob-worker
# 调度服务器地址，IP:Port 或 域名，多值逗号分隔
powerjob.worker.server-address=127.0.0.1:7700,127.0.0.1:7701
# 通讯协议，4.3.0 开始支持 HTTP 和 AKKA 两种协议，官方推荐使用 HTTP 协议（注意 server 和 worker 都要开放相应端口）
powerjob.worker.protocol=http
# 持久化方式，可选，默认 disk
powerjob.worker.store-strategy=disk
# 任务返回结果信息的最大长度，超过这个长度的信息会被截断，默认值 8192
powerjob.worker.max-result-length=4096
# 单个任务追加的工作流上下文最大长度，超过这个长度的会被直接丢弃，默认值 8192
powerjob.worker.max-appended-wf-context-length=4096
# 同时运行的轻量级任务数量上限
powerjob.worker.max-lightweight-task-num=1024
# 同时运行的重量级任务数量上限
powerjob.worker.max-heavy-task-num=64
```

最后，为了更好的观察 powerjob-worker 的运行情况，建议为 powerjob-worker 单独配置日志输出，详细教程见 日志单独配置

普通应用接入方式 （Spring 或 普通 Java 项目）

首先，添加相关的依赖，最新依赖版本请参考maven中央仓库：推荐地址 & 备用地址

```xml
<dependency>
  <groupId>tech.powerjob</groupId>
  <artifactId>powerjob-worker</artifactId>
  <version>${latest.powerjob.version}</version>
</dependency>
```

随后，手动初始化 `PowerJobWorker` 对象。PowerJobWorker 对象的初始化依赖于配置类 `PowerJobWorkerConfig`，各参数说明如下表所示：

属性名称	含义	默认值
appName	宿主应用名称，需要提前在控制台完成注册	无，必填项，否则启动报错

port	Worker 工作端口	27777，不推荐修改
serverAddress	调度中心（powerjob-server）地址列表	无，必填项，否则启动报错
storeStrategy	本地存储策略，枚举值磁盘/内存，大型 MapReduce 等会产生大量 Task 的任务推荐使用磁盘降低内存压力，否则建议使用内存加速计算	StoreStrategy.DISK（磁盘）
protocol	与 server 的通讯协议	出于兼容性考虑仍未 AKKA，但建议手动改为 HTTP
maxResultLength	每个Task返回结果的默认长度，超长将被截断，过长可能导致网络拥塞	8096
userContext	用户自定义上下文对象，该值会被透传到 TaskContext#userContext 属性（可选参数）	null
allowLazyConnectServer	是否允许延迟连接 server。 启用后无需Server也能顺利启动 PowerJobWorker，用于本地无 server 启动等场景	false
maxAppendedWfContextLength	单个任务向工作流上下文中追加数据的最大长度，超过这个长度会被直接丢弃	8192
maxLightweightTaskNum	同时运行的轻量级任务数量上限	1024
maxHeavyweightTaskNum	同时运行的重量级任务数量上限	64
healthReportInterval	worker 健康状态上报的间隔（秒）	10
processorFactoryList	处理器工厂，允许自己定制处理器加载逻辑来支持丰富的扩展需求	null

最后，初始化客户端，完成执行器的启动，代码示例如下：

- Spring 用户请使用 `PowerJobSpringWorker`
- 非 Spring 用户请使用 `PowerJobWorker`

```java
@Configuration
public class PowerJobWorkerConfiguration {

    @Bean
    public PowerJobSpringWorker initPowerJobWorker() throws Exception {

        // 1. 创建配置文件
        PowerJobWorkerConfig config = new PowerJobWorkerConfig();
        config.setPort(28888);
        config.setAppName("my-java-application");
        config.setServerAddress(Lists.newArrayList("127.0.0.1:7700", "127.0.0.1:7701"));
        // 如果没有大型 Map/MapReduce 的需求，建议使用内存来加速计算
        config.setStoreStrategy(StoreStrategy.DISK);

        // 2. 创建 Worker 对象，设置配置文件（注意 Spring 用户需要使用 PowerJobSpringWorker，而不是 PowerJobWorker）
        PowerJobSpringWorker worker = new PowerJobSpringWorker(config);
        return worker;
    }
}
```

非 Spring 应用程序在创建 PowerJobWorker 对象后手动调用 `powerJobWorker.init()` 方法完成初始化即可。

日志单独配置

目前，powerjob-worker 并没有实现自己的 LogFactory（如果有需求的话请提 ISSUE，可以考虑实现），原因如下：

1. powerjob-worker 的日志基于 `Slf4J` 输出，即采用了基于门面设计模式的日志框架，宿主应用无论如何都可以搭起 Slf4J 与实际的日志框架这座桥梁。
2. 减轻了部分开发工作量，不再需要实现自己的 LogFactory。

为此，为了顺利且友好地输出日志，请在日志配置文件（logback.xml/log4j2.xml/...）中为 `powerjob-worker` 单独进行日志配置，比如（logback 示例）：

```xml
<appender name="POWERJOB_WORKER_APPENDER" class="ch.qos.logback.core.rolling.RollingFileAppender">
    <file>${LOG_PATH}/powerjob-worker.log</file>
    <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
        <FileNamePattern>${LOG_PATH}/powerjob-worker.%d{yyyy-MM-dd}.log</FileNamePattern>
        <MaxHistory>7</MaxHistory>
    </rollingPolicy>
    <encoder class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
        <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{50} - %msg%n</pattern>
        <charset>UTF-8</charset>
    </encoder>
    <append>true</append>
</appender>

<logger name="tech.powerjob.worker" level="INFO" additivity="false">
    <appender-ref ref="POWERJOB_WORKER_APPENDER" />
</logger>
```

无论如何，powerjob-worker 启动时都会打印 Banner（如下所示），您可以通过 Banner 来判断日志配置是否成功。

基于代理应用（powerjob-agent）的初始化

> agent 是一个仅集成了官方处理器的执行器（其实就是为 worker 加了一个 main 方法），如果常见需求为脚本的调度执行，可以考虑直接部署 agent，简单快捷方便～

推荐直接使用 Docker 部署：powerjob-agent

与 server 一样，启动参数通过环境变量 `-e PARAMS` 传入，全部参数请见底部。

```shell
docker run -d
-e PARAMS="--app powerjob-agent-test --server 192.168.1.1:7700,192.168.1.2:7700"
-p 27777:27777 --name powerjob-agent
-v ~/docker/powerjob-agent:/root powerjob/powerjob-agent:$version
```

或直接启动 agent 的jar包：`java -jar powerjob-worker-agent-3.1.0.jar -a my-agent -s 127.0.0.1:7700`

```shell
Usage: PowerJobAgent [-hV] -a=<appName> [-e=<storeStrategy>] [-l=<length>]
                     [-o=<protocol>] [-p=<port>] -s=<server> [-t=<tag>]
powerjob-worker agent
  -a, --app=<appName>     worker-agent's name
  -e, --persistence=<storeStrategy>
                          storage strategy, DISK or MEMORY
  -h, --help              Show this help message and exit.
  -l, --length=<length>   ProcessResult#msg max length
  -o, --protocol=<protocol>
                          transporter protocol, AKKA or HTTP
  -p, --port=<port>       transporter working port, not recommended to change
  -s, --server=<server>   oms-server's address, IP:Port OR domain
  -t, --tag=<tag>         worker-agent's tag
  -V, --version           Print version information and exit.
```

处理器（Processor）开发

处理器概述

基本概念

> **PowerJob 支持 Python、Shell、HTTP、SQL 等众多通用任务的处理，开发者只需要引入依赖，在控制台配置好相关参数即可**，关于这部分详见 官方处理器 ，此处不再赘述。本章将重点阐述 Java 处理器开发方法与使用技巧。

- Java 处理器可根据**代码所处位置**划分为内置 Java 处理器和外置 Java 处理器，前者直接集成在宿主应用（也就是接入本系统的业务应用）中，一般用来处理业务需求；后者可以在一个独立的轻量级的 Java 工程中开发，通过 **JVM 容器技术**（详见容器章节）被 worker 集群热加载，提供 Java 的“脚本能力”，一般用于处理灵活多变的需求。
- Java 处理器可根据**对象创建者**划分为 SpringBean 处理器和普通 Java 对象处理器，前者由 Spring IOC 容器完成处理器的创建和初始化，后者则由 PowerJob 维护其生命周期。如果宿主应用支持 Spring，**强烈建议使用 SpringBean 处理器**，开发者仅需要将 Processor 注册进 Spring IOC 容器（一个 `@Component` 注解或一句 `bean` 配置）即可享受 Spring 带来的便捷之处。
- Java处理器可根据**功能**划分为单机处理器、广播处理器、Map 处理器和 MapReduce 处理器。
  - 单机处理器（`BasicProcessor`）对应了单机任务，即某个任务的某次运行只会有某一台机器的某一个线程参与运算。
  - 广播处理器（`BroadcastProcessor`）对应了广播任务，即某个任务的某次运行会**调动集群内所有机器参与运算**。
  - Map处理器（`MapProcessor`）对应了Map任务，即某个任务在运行过程中，**允许产生子任务并分发到其他机器进行运算**。
  - MapReduce 处理器（`MapReduceProcessor`）对应了 MapReduce 任务，在 Map 任务的基础上，**增加了所有任务结束后的汇总统计**。

核心方法 process

`BasicProcessor#process` 是任意 Java 处理器都需要实现的核心方法，描述了本次任务具体的工作内容，其方法签名如下：

```java
ProcessResult process(TaskContext context) throws Exception;
```

入参 TaskContext

> TaskContext 包含了本次任务的上下文信息，具体信息如下

属性列表（红色标注的为常用属性）	
属性名称	意义/用法
jobId	任务 ID，开发者一般无需关心此参数
instanceId	任务实例 ID，全局唯一，开发者一般无需关心此参数
subInstanceId	子任务实例 ID，秒级任务使用，开发者一般无需关心此参数
taskId	采用链式命名法的 ID，在某个任务实例内唯一，开发者一般无需关心此参数
taskName	task 名称，Map/MapReduce 任务的子任务的值为开发者指定，否则为系统默认值，开发者一般无需关心此参数
jobParams	任务参数 对于非**工作流中的任务**其值**等同于控制台录入的任务参数**；如果该任务为**工作流中的任务**且有配置节点参数信息，那么接收到的是**节点配置的参数信息**

instanceParams	任务实例参数 **对于非工作流中的任务** 其值 **等同于 OpenAPI 传递的实例参数，非 OpenAPI 触发的任务则一定为空。**如果该任务为**工作流中的任务**那么这里实际接收到的是**工作流上下文信息，建议使用 getWorkflowContext 方法获取上下文信息**
maxRetryTimes	Task 的最大重试次数
currentRetryTimes	Task 的当前重试次数，和 maxRetryTimes 联合起来可以判断当前是否为该 Task 的最后一次运行机会
subTask	子 Task，Map/MapReduce 处理器专属，开发者调用map方法时传递的子任务列表中的某一个
omsLogger	在线日志，用法同 Slf4J，记录的日志可以直接通过控制台查看，非常便捷和强大！不过使用过程中需要注意频率，滥用在线日志会对 Server 造成巨大的压力
userContext	用户在 PowerJobWorkerConfig 中设置的自定义上下文
workflowContext	工作流上下文，更多信息见下方说明

工作流上下文（ WorkflowContext ）

该属性是 v4.0.0 版本的重大变更之一，移除了原来的参数传递机制，提供了 API 让开发者可以更加灵活便捷地在工作流中实现信息的传递。

属性列表	
属性名称	意义/用法
wfInstanceId	工作流实例 ID
data	工作流上下文数据，键值对

appendedContextData	当前任务向工作流上下文中追加的数据。在任务执行完成后 ProcessorTracker 会将其上报给 TaskTracker，TaskTracker 在当前任务执行完成后会将这个信息上报给 server ，追加到当前的工作流上下文中，供下游任务消费

上游任务通过 `WorkflowContext#appendData2WfContext(String key,Object value)` 方法向工作流上下文中**追加数据**，下游任务便可以通过 `WorkflowContext#fetchWorkflowContext()` 方法获取到相应的数据进行**消费**。注意，当追加的上下文信息的 key **已经存在**于当前的上下文中时，新的 value 会**覆盖**之前的值。另外，每次任务实例追加的上下文数据大小也会受到 worker 的配置项 `powerjob.worker.max-appended-wf-context-length` 的限制，超过这个长度的会被**直接丢弃**。

返回值 ProcessResult

方法的返回值为 `ProcessResult`，代表了本次 Task 执行的结果，包含 `success` 和 `msg` 两个属性，分别用于传递 Task 是否执行成功和 Task 需要返回的信息。

处理器开发示例

单机处理器：BasicProcessor

单机执行的策略下，server 会在所有可用 worker 中选取健康度最佳的机器进行执行。单机执行任务需要实现接口 `BasicProcessor`，代码示例如下：

```java
// 支持 SpringBean 的形式
@Component
public class BasicProcessorDemo implements BasicProcessor {

    @Resource
    private MysteryService mysteryService;

    @Override
    public ProcessResult process(TaskContext context) throws Exception {

        // 在线日志功能，可以直接在控制台查看任务日志，非常便捷
        OmsLogger omsLogger = context.getOmsLogger();
        omsLogger.info("BasicProcessorDemo start to process, current JobParams is {}.", context.getJobParams());

        // TaskContext为任务的上下文信息，包含了在控制台录入的任务元数据，常用字段为
        // jobParams（任务参数，在控制台录入），instanceParams（任务实例参数，通过 OpenAPI 触发的任务实例才可能存在该参数）

        // 进行实际处理...
        mysteryService.hasaki();

        // 返回结果，该结果会被持久化到数据库，在前端页面直接查看，极为方便
        return new ProcessResult(true, "result is xxx");
    }
}
```

广播处理器：BroadcastProcessor

广播执行的策略下，所有机器都会被调度执行该任务。为了便于资源的准备和释放，广播处理器在 `BasicProcessor` 的基础上额外增加了 `preProcess` 和 `postProcess` 方法，分别在整个集群开始之前/结束之后**选一台机器**执行相关方法。代码示例如下：

```java
@Component
public class BroadcastProcessorDemo implements BroadcastProcessor {

    @Override
    public ProcessResult preProcess(TaskContext taskContext) throws Exception {
        // 预执行，会在所有 worker 执行 process 方法前调用
        return new ProcessResult(true, "init success");
    }

    @Override
    public ProcessResult process(TaskContext context) throws Exception {
        // 撰写整个worker集群都会执行的代码逻辑
        return new ProcessResult(true, "release resource success");
    }

    @Override
    public ProcessResult postProcess(TaskContext taskContext, List<TaskResult> taskResults) throws Exception {

        // taskResults 存储了所有worker执行的结果（包括preProcess）

        // 收尾，会在所有 worker 执行完毕 process 方法后调用，该结果将作为最终的执行结果
        return new ProcessResult(true, "process success");
    }
}
```

并行处理器：MapReduceProcessor

MapReduce 是最复杂也是最强大的一种执行器，它允许开发者完成任务的拆分，将子任务派发到集群中其他Worker 执行，是执行大批量处理任务的不二之选！实现 MapReduce 处理器需要继承 `MapReduceProcessor` 类，具体用法如下示例代码所示：

Java

```java
@Slf4j
@Component("demoMapReduceProcessor")
public class MapReduceProcessorDemo implements MapReduceProcessor {

    @Override
    public ProcessResult process(TaskContext context) throws Exception {

        // PowerJob 提供的日志 API，可支持在控制台指定多种日志模式（在线查看 / 本地打印）。最佳实践：全部使用 OmsLogger 打印日志，开发阶段控制台配置为 在线日志方便开发；上线后调整为本地日志，与直接使用 SLF4J 无异
        OmsLogger omsLogger = context.getOmsLogger();

        // 是否为根任务，一般根任务进行任务的分发
        boolean isRootTask = isRootTask();
        // Task 名称，除了 MAP 任务其他 taskName 均由开发者自己创建，某种意义上也可以按参数理解（比如多层 MAP 的情况下，taskName 可以命名为，Map_Level1, Map_Level2，最终按 taskName 判断层级进不同的执行分支）
        String taskName = context.getTaskName();
        // 任务参数，控制台任务配置中直接填写的参数
        String jobParamsStr = context.getJobParams();
        // 任务示例参数，运行任务时手动填写的参数（等同于 OpenAPI runJob 的携带的参数）
        String instanceParamsStr = context.getInstanceParams();

        omsLogger.info("[MapReduceDemo] [startExecuteNewTask] jobId:{}, instanceId:{}, taskId:{}, taskName: {}, RetryTimes: {}, isRootTask:{}, jobParams:{}, instanceParams:{}", context.getJobId(), context.getInstanceId(), context.getTaskId(), taskName, context.getCurrentRetryTimes(), isRootTask, jobParamsStr, instanceParamsStr);

        // 常见写法，优先从 InstanceParams 获取参数，取不到再从 JobParams 中获取，灵活性最佳（相当于实现了实例参数重载任务参数）
        String finalParams = StringUtils.isEmpty(instanceParamsStr) ? jobParamsStr : instanceParamsStr;
        final JSONObject params = Optional.ofNullable(finalParams).map(JSONObject::parseObject).orElse(new JSONObject());

        if (isRootTask) {

            omsLogger.info("[MapReduceDemo] [RootTask] start execute root task~");

            /*
             * rootTask 内的核心逻辑，即为按自己的业务需求拆分子任务。比如

```

```
33                  *  - 从数据库/数仓拉一批任务出来做计算，那 MAP 任务就可以 stream 读全库，每 N 个 ID 作为一个 SubTask 对外分发
34                  *  - 需要读取几千万个文件进行解析，那么 MAP 任务就可以将 N 个文件名作为一个 SubTask 对外分发，每个子任务接收到文件名称进行文件处理
35                  *
36                  * eg. 现在需要从文件中读取100W个ID，并处理数据库中这些ID对应的数据，那么步骤如下：
37                  * 1. 根任务（RootTask）读取文件，流式拉取100W个ID，并按100个一批的大小组装成子任务进行派发
38                  * 2. 非根任务获取子任务，完成业务逻辑的处理
39                  *
40                  * 以下 demo 进行该逻辑的模拟
41                  */
42
43
44                 // 构造子任务
45
46                 // 需要读取的文件总数
47                 Long num = MapUtils.getLong(params, "num", 100000L);
                   // 每个子任务携带多少个文件ID（此参数越大，每个子任务就“越大”，如果失败的重试成本就越高。参数越小，每个子任务就越轻，当相应的分片数量会提升，会让 PowerJob 计算开销增大，建议按业务需求合理调配）
48                 Long batchSize = MapUtils.getLong(params, "batchSize", 100L);
49
50
                   // 此处模拟从文件读取 num 个 ID，每个子任务携带 batchSize 个 ID 作为一个分片
51                 List<Long> ids = Lists.newArrayList();
52                 for (long i = 0; i < num; i++) {
53                     ids.add(i);
54
55                     if (ids.size() >= batchSize) {
56
57                         // 构造自己的子任务，自行传递所有需要的参数
58                         SubTask subTask = new SubTask(ThreadLocalRandom.current().nextLong(), Lists.newArrayList(ids), "extra");
59                         ids.clear();
60
61                         try {
62                             /*
63                             第一个参数：List<子任务>，map 支持批量操作以减少网络 IO 提升性能，简单起见此处不再示例，开发者可自行优化性能
64                             第二个参数：子任务名称，即后续 Task 执行时从 TaskContext#taskName 拿到的值。某种意义上也可以按参数理解（比如多层 MAP 的情况下，taskName 可以命名为，Map_Level1, Map_Level2，最终按 taskName 判断层级进不同的执行分支）
65                              */
66                             map(Lists.newArrayList(subTask), "L1_FILE_PROCESS");
67                         } catch (Exception e) {
68
```

```
                        // 注意 MAP 操作可能抛出异常，建议进行捕获并按需处理
                        omsLogger.error("[MapReduceDemo] map task failed!", e);
                        throw e;
                    }
                }
            }

            if (!ids.isEmpty()) {
                map(Lists.newArrayList(new SubTask()), "L1_FILE_PROCESS");
            }

            // map 阶段的结果，由于前置逻辑为异常直接抛出，执行到这里一定成功，所以无脑设置为 success。开发者可自行调整逻辑
            return new ProcessResult(true, "MAP_SUCCESS,totalNum:" + num);

        }

        // 如果是简单的二层结构（ROOT - SubTASK），此处一定是子 Task，无需再次判断。否则可使用 TaskContext#taskName 字符串匹配 或 TaskContext#SubTask 对象内自定义参数匹配，进入目标执行分支

        // 获取前置节点 map 传递过来的参数，进行业务处理
        SubTask subTask = (SubTask) context.getSubTask();
        log.info("[MapReduceDemo] [SubTask] taskId:{}, taskName: {}, subTask: {}", context.getTaskId(), taskName, JsonUtils.toJSONString(subTask));
        Thread.sleep(MapUtils.getLong(params, "bizProcessCost", 233L));

        // 模拟有成功有失败的情况，开发者按真实业务执行情况判断即可
        long successRate = MapUtils.getLong(params, "successRate", 80L);
        long randomNum = ThreadLocalRandom.current().nextLong(100);
        if (successRate > randomNum) {
            return new ProcessResult(true, "PROCESS_SUCCESS:" + randomNum);
        } else {
            return new ProcessResult(false, "PROCESS_FAILED:" + randomNum);
        }
    }

    @Override
    public ProcessResult reduce(TaskContext context, List<TaskResult> taskResults) {

        // 子任务结果太大，上报在线日志会有 IO 问题，直接使用本地日志打
```

```
        log.info("List<TaskResult>: {}", JSONObject.toJSONString(taskResults));

        OmsLogger omsLogger = context.getOmsLogger();
        omsLogger.info("================ MapReduceProcessorDemo#reduce ==============");

        // 所有 Task 执行结束后，reduce 将会被执行，taskResults 保存了所有子任务的执行结果。（注意 reduce 由于保存了所有子任务的执行结果，在子任务规模巨大时对内存有极大开销，超大型计算任务慎用或使用流式 reduce（开发中））

        // 用法举例：统计执行结果
        AtomicLong successCnt = new AtomicLong(0);
        AtomicLong failedCnt = new AtomicLong(0);
        taskResults.forEach(tr -> {
            if (tr.isSuccess()) {
                successCnt.incrementAndGet();
            } else {
                failedCnt.incrementAndGet();
            }
        });


        double successRate = 1.0 * successCnt.get() / (successCnt.get() + failedCnt.get());

        String resultMsg = String.format("succeedTaskNum:%d,failedTaskNum:%d,successRate:%f", successCnt.get(), failedCnt.get(), successRate);
        omsLogger.info("[MapReduceDemo] [Reduce] {}", resultMsg);

        // reduce 阶段的结果，将作为任务真正执行结果
        if (successRate > 0.8) {
            return new ProcessResult(true, resultMsg);
        } else {
            return new ProcessResult(false, resultMsg);
        }

    }


    /**
     * 自定义的子任务，按自己的业务需求定义即可
     * 注意：代表子任务参数的类：一定要有无参构造方法！一定要有无参构造方法！一定要有无参构造方法！
     * 最好把 GET / SET 方法也加上，减少序列化问题的概率
     */
    @Data
    @AllArgsConstructor
```

```
    public static class SubTask implements Serializable {

        /**
         * 再次强调，一定要有无参构造方法
         */
        public SubTask() {
        }

        private Long siteId;

        private List<Long> idList;

        private String extra;
    }
}
```

注：Map 处理器相当于 MapReduce 处理器的阉割版本（阉割了 `reduce` 方法），此处不再单独举例。

- reduce 阶段需要全量读取子任务结果，会对内存一定要求。如果只需要计算，不需要二次统计或自行完成统计，推荐使用 Map 处理器即可，对执行节点性能更友好。

最佳实践：MapReduce 实现静态分片

虽然说这有点杀鸡焉用牛刀的感觉，不过既然目前市面上同类产品都处于静态分片的阶段，我也就在这里给大家举个例子吧～

```java
@Component
public class StaticSliceProcessor implements MapReduceProcessor {

    @Override
    public ProcessResult process(TaskContext context) throws Exception {
        OmsLogger omsLogger = context.getOmsLogger();

        // root task 负责分发任务
        if (isRootTask()) {
            // 从控制台传递分片参数，假设格式为KV：1=a&2=b&3=c
            String jobParams = context.getJobParams();
            Map<String, String> paramsMap = Splitter.on("&").withKeyValueSeparator("=").split(jobParams);

            List<SubTask> subTasks = Lists.newLinkedList();
            paramsMap.forEach((k, v) -> subTasks.add(new SubTask(Integer.parseInt(k), v)));
            map(subTasks, "SLICE_TASK");
            return new ProcessResult(true, "ROOT_PROCESS_SUCCESS");
        }

        Object subTask = context.getSubTask();
        if (subTask instanceof SubTask) {
            // 实际处理
            // 当然，如果觉得 subTask 还是很大，也可以继续分发哦

            return new ProcessResult(true, "subTask:" + ((SubTask) subTask).getIndex() + " process successfully");
        }
        return new ProcessResult(false, "UNKNOWN BUG");
    }

    @Override
    public ProcessResult reduce(TaskContext context, List<TaskResult> taskResults) {
        // 按需求做一些统计工作... 不需要的话，直接使用 Map 处理器即可
        return new ProcessResult(true, "xxxx");
    }

    @Getter
    @Setter
    @NoArgsConstructor
    @AllArgsConstructor
    public static class SubTask {
        private int index;
```

```
        private String params;
    }
}
```

最佳实践：MapReduce 多级分发处理

利用 MapReduce 实现 Root -> A -> B/C -> Reduce）的 DAG 工作流。
注：PowerJob 现已提供正式的 Workflow 工作流支持（since v2.0.0）

```java
@Component
public class DAGSimulationProcessor implements MapReduceProcessor {

    @Override
    public ProcessResult process(TaskContext context) throws Exception {

        if (isRootTask()) {
            // L1. 执行根任务

            // 执行完毕后产生子任务 A，需要传递的参数可以作为 TaskA 的属性进行传递
            TaskA taskA = new TaskA();
            map(Lists.newArrayList(taskA), "LEVEL1_TASK_A");
            return new ProcessResult(true, "L1_PROCESS_SUCCESS");
        }

        if (context.getSubTask() instanceof TaskA) {
            // L2. 执行A任务

            // 执行完成后产生子任务 B，C（并行执行）
            TaskB taskB = new TaskB();
            TaskC taskC = new TaskC();
            map(Lists.newArrayList(taskB, taskC), "LEVEL2_TASK_BC");
            return new ProcessResult(true, "L2_PROCESS_SUCCESS");
        }

        if (context.getSubTask() instanceof TaskB) {
            // L3. 执行B任务
            return new ProcessResult(true, "xxx");
        }
        if (context.getSubTask() instanceof TaskC) {
            // L3. 执行C任务
            return new ProcessResult(true, "xxx");
        }

        return new ProcessResult(false, "UNKNOWN_TYPE_OF_SUB_TASK");
    }

    @Override
    public ProcessResult reduce(TaskContext context, List<TaskResult> taskResults) {
        // L4. 执行最终 Reduce 任务，taskResults保存了之前所有任务的结果
        taskResults.forEach(taskResult -> {
            // do something...
        });
        return new ProcessResult(true, "reduce success");
```

```
        }

        public static class TaskA {
        }
        public static class TaskB {
        }
        public static class TaskC {
        }
    }
```

更多示例

没看够？更多示例请见：powerjob-worker-samples

任务与工作流配置

> 调度服务器与执行器部署完成后，录入任务或工作流信息即可正式开启调度。任务与工作流的配置可在前端页面完成（当然也可以通过 OpenAPI 完成）。本章将详细详细介绍 PowerJob 的主控（前端控制台 powerjob-console）的使用！

powerjob-console 的主要功能如下

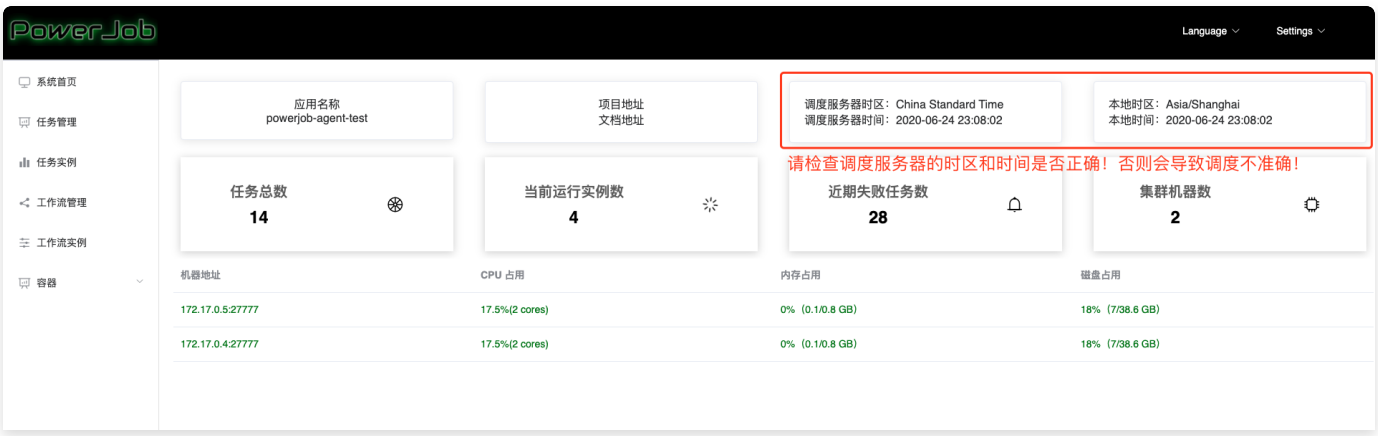
- 任务的配置与管理
- 工作流的配置与管理（提供工作流在线编辑界面，简单易用）
- 任务实例的监控，包括查看运行状态、详细运行时信息、在线日志等功能
- 工作流实例的监控，包括查看整体的运行情况（上下游关系）、某个子任务实例的运行情况等
- 任务实例的运维，允许开发者在线停止某个任务实例或运行某个任务
- 工作流实例的运维，允许开发者在线停止、重试某个工作流实例或运行某个工作流
- 容器模版的生成，一键生成容器 maven 项目，极大提升开发效率
- 容器信息的配置与管理
- 容器的在线运维，包括部署、监控等

以下为控制台的详细介绍和界面展示（由于内容较多，请善用搜索或大纲功能快速定位）。

系统首页

展示了系统整体的概览和集群 Worker 列表，同时提供服务器时间和本地时间的对比来确保调度的准确性。

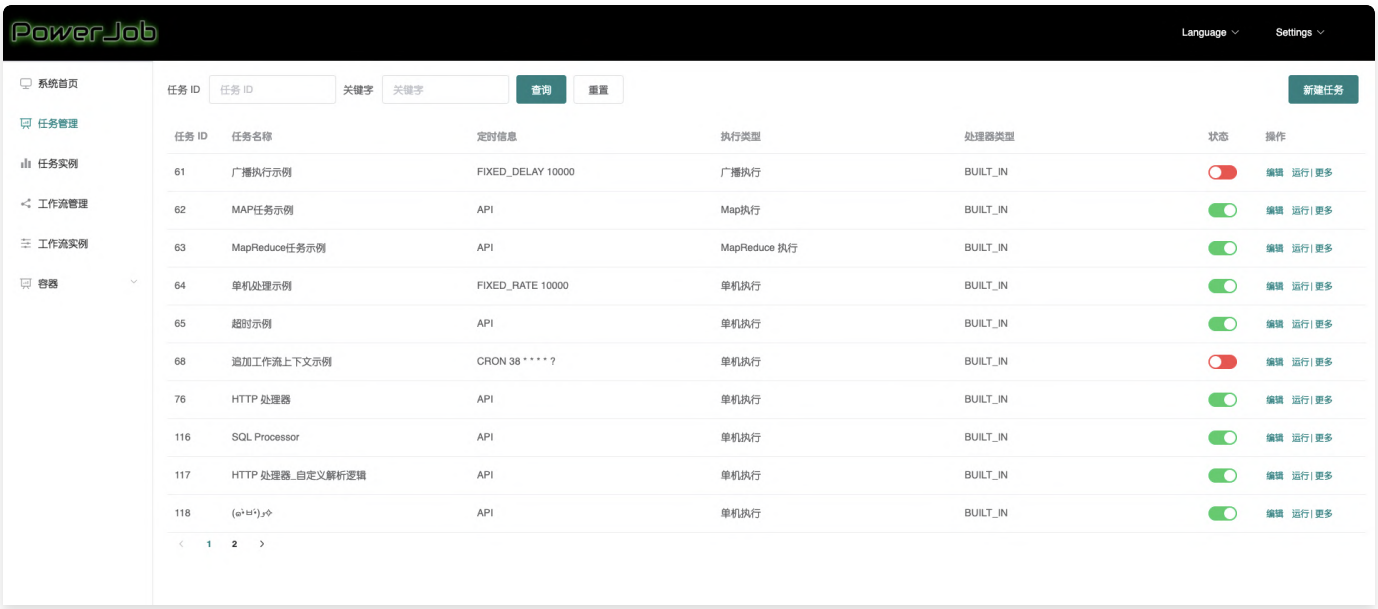
请检查调度服务器时区、时间与本地浏览器时区、时间是否一致，不一致的情况下会导致调度无法准确进行！



任务管理

主界面

直观地展示当前系统所管理的所有任务信息，并提供相应的运维方法。



新增任务界面（教程，推荐仔细阅读）

点击右上角按钮新建任务，即可录入新的任务，具体界面和说明如下所示。

- 任务名称：名称，便于记忆与搜索，无特殊用途，请尽量简短（占用数据库字段空间）
- 任务描述：描述，无特殊作用，请尽量简短（占用数据库字段空间）
- 任务参数：任务处理时能够获取到的参数（即各个Processor 的 process 方法入参 `TaskContext` 对象的 jobParams 属性）（进行一次处理器开发就能理解了）
- 定时信息：**该任务的触发方式**，由下拉框和输入框组成
  - API -> 不需要填写任何参数，表明该任务由 OpenAPI 触发
  - CRON -> 填写 CRON 表达式（在线生成网站）
  - 固定频率 -> 填写整数，单位**毫秒**
  - 固定延迟 -> 填写整数，单位**毫秒**
  - 每日固定间隔 -> 哪几天的哪些时间段需要执行，比如每周二和三的10点到11点间每10分钟触发一次
  - 工作流 -> 不需要填写任何参数，表明该任务由工作流（workflow）触发
- 生命周期：定时策略生效的时间段
- 执行配置：由执行类型（单机、广播和 MapReduce ）、处理器类型和处理器参数组成，后两项相互关联。
  - 内置Java处理器
    - 方式一 -> 填写该处理器的**全限定类名**（eg，`tech.powerjob.samples.processors.MapReduceProcessorDemo`）
    - 方式二 -> 填写 IOC 容器的 bean 名称，比如 Spring 用户可填写 Spring Bean 名称（eg，处理器使用注解 `@Component(value = "powerJobProcessor")`，则控制

台可填写 powerJobProcessor）

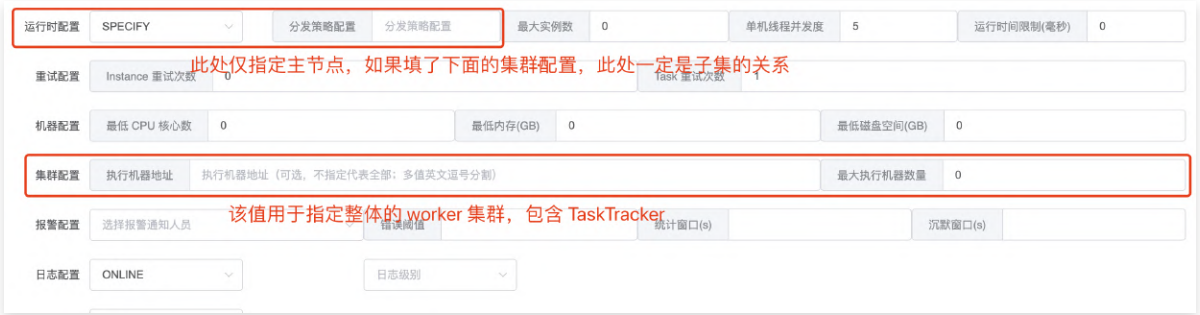
- 方式三 -> 方法级注解，非 MapReduce 任务可直接使用注解 `@PowerJobHandler` 将某个方法转化为 PowerJob 任务，并设置唯一入参 `TaskContext` 注入上下文，具体参考代码

```java
@Component(value = "springMethodProcessorService")
public class SpringMethodProcessorService {

    /**
     * 处理器配置方法1： 全限定类名#方法名，比如 tech.powerjob.samples.tester.SpringMethodProcessorService#testEmptyReturn
     * 处理器配置方法2： SpringBean名称#方法名，比如 springMethodProcessorService#testEmptyReturn
     * @param context 必须要有入参 TaskContext，返回值可以是 null，也可以是其他任意类型。正常返回代表成功，抛出异常代表执行失败
     */
    @PowerJobHandler(name = "testEmptyReturn")
    public String testEmptyReturn(TaskContext context) {
        OmsLogger omsLogger = context.getOmsLogger();
        omsLogger.warn("测试日志");
        return "响应结果，正常返回视为执行成功，抛出异常视为执行失败"
    }
}
```

  - Java容器 -> 填写**容器ID#处理器全限定类名**（eg，`1#cn.edu.zju.oms.container.ContainerMRProcessor`）
  - SHELL、Python、SQL 、HTTP 等任务的执行：点击查看官方处理器的使用教程

- 运行配置
  - 派发策略：默认健康度优先，优先选择性能最优机器进行执行，可选随机均摊等其他派发模式
    - HEALTH_FIRST：默认策略，健康度优先，会选择 worker 集群中状态最好的机器作为本次任务的主节点
    - RANDOM：随机
    - SPECIFY：指定主节点运行，常用于 Map/MapReduce 等场景，大规模计算时，主节点部署/重启会导致任务完全失败，因此可为主节点搭建一个隔离环境，通过该参数指定主节点到该隔离环境运行，使其摆脱普通 worker 节点部署带来的影响。指定语法等同于“执行

机器地址”的语法，填写 IP 或者 TAG。



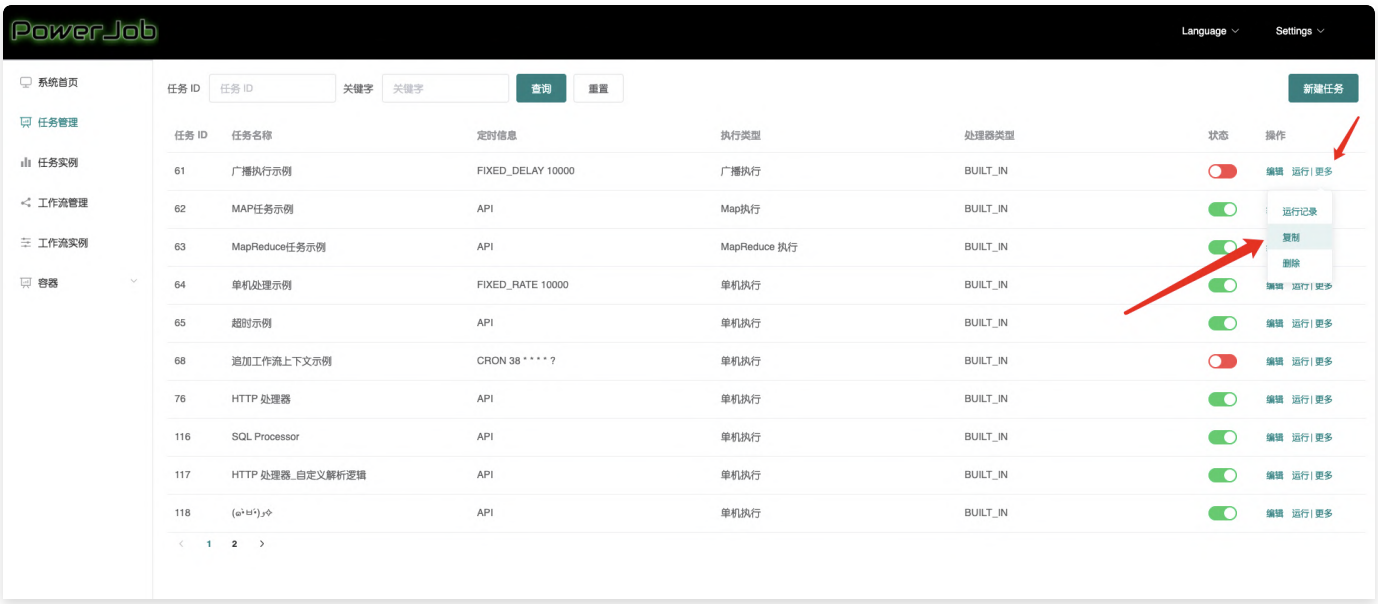
  - 最大实例数：该任务允许同时执行的数量，0代表不限（默认为 0）
  - 单机线程并发数：该实例执行过程中每个 Worker 使用的线程数量（MapReduce 任务生效，其余无论填什么，都只会使用必要的线程数...）
  - 运行时间限制：限定任务的最大运行时间，超时则视为失败，单位**毫秒**，0 代表不限制超时时间（**不建议不限制超时时间**）。
- 重试配置：
  - Instance 重试次数：实例级别，失败了整个任务实例重试，会更换 TaskTracker（本次任务实例的Master节点），代价较大，大型Map/MapReduce慎用。
  - Task 重试次数：Task 级别，每个子 Task 失败后单独重试，会更换 ProcessorTracker（本次任务实际执行的 Worker 节点），代价较小，推荐使用。
  - 注：**请注意同时配置任务重试次数和子任务重试次数之后的重试放大**，比如对于单机任务来说，假如任务重试次数和子任务重试次数都配置了 1 且都执行失败，实际执行次数会变成 4 次！推荐任务实例重试配置为 0 ，子任务重试次数根据实际情况配置。
- 机器配置：用来标明允许执行任务的机器状态，避开那些摇摇欲坠的机器，0 代表无任何限制。
  - 最低 CPU 核心数：填写浮点数，CPU 可用核心数小于该值的 Worker 将不会执行该任务。
  - 最低内存（GB）：填写浮点数，可用内存小于该值的 Worker 将不会执行该任务。
  - 最低磁盘（GB）：填写浮点数，可用磁盘空间小于该值的 Worker 将不会执行该任务。
- 集群配置
  - 执行机器地址，指定集群中的某几台机器执行任务
    - IP模式：多值英文逗号分割，如 `192.168.1.1:27777,192.168.1.2:27777` 。常用于 debug 等场景，需要指定特定机器运行。
    - TAG 模式：通过 `PowerJobWorkerConfig#tag` 将执行器打标分组后，可在控制台通过 tag 指定某一批机器执行。常用于分环境分单元执行的场景。如某些任务需要屏蔽安全生产环境（tag 设置为环境标），某些任务只需要在特定单元执行（tag 设置单元标）
  - 最大执行机器数量：限定调动执行的机器数量
- 报警配置：
  - 选择任务执行失败后报警通知的对象，需要事先录入。
  - 对于秒级任务，支持分别指定错误阈值(C)、统计窗口(S)、沉默窗口(W) 来定制更加丰富的告

警机制（表示的是在 S 秒内失败次数达到 C 次就触发告警，并且沉默 W 秒）

- 日志配置：可使用控制台配置调整 Job 使用的 Logger 及 LogLevel
  - 支持 SERVER（服务端日志，默认）、LOCAL（本地日志）、STDOUT（系统输出）、NULL（空实现）4种 LogType
  - 支持 DEBUG、INFO、WARN、ERROR、OFF 5种级别控制
  - 使用建议：初期调试可使用 SERVER 日志，后续功能稳定后改为 LOCAL，并调高日志级别，降低通讯压力，消除性能瓶颈问题
- 高级设置
  - TaskTracker 行为：
    - NORMAL：常规行为。不特殊处理，TaskTracker 正常参与集群计算，会导致其负载比常规节点高。适用于节点数不那么多，任务不那么繁重的场景。
    - PADDLING：划水：只负责管理节点，不参与计算，稳定性最优。适用于节点数量非常多的大规模计算场景，少一个计算节点来换取稳定性提升。

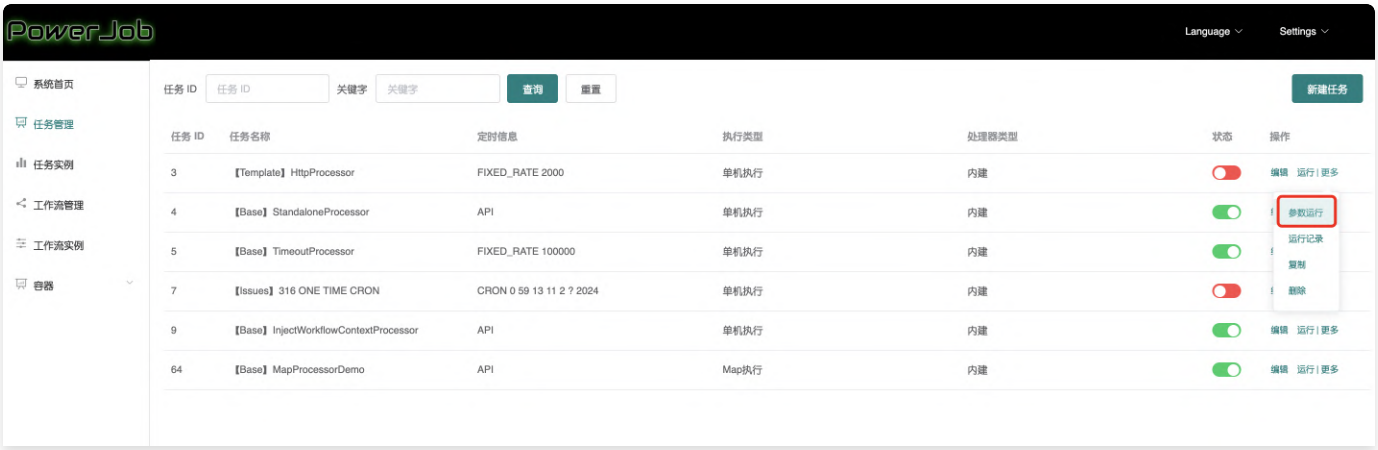
复制任务

移动到更多，点击复制即可一键复制指定任务 ~



参数运行

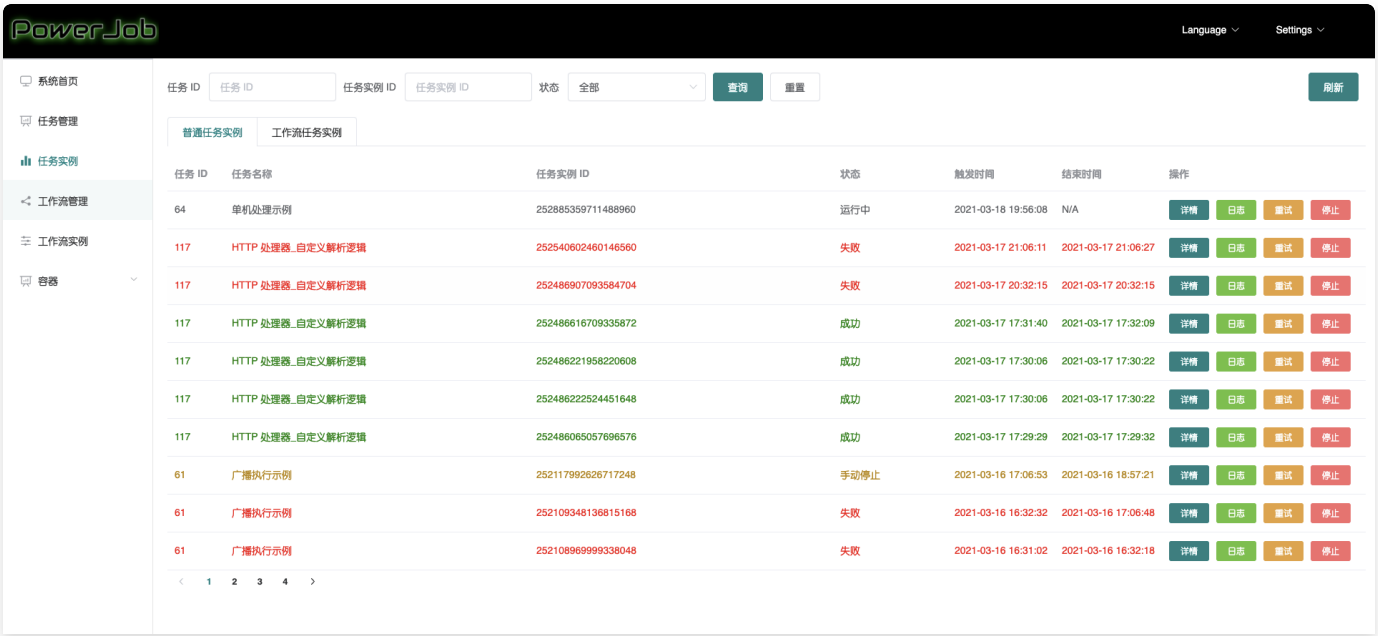
支持运行任务时手动指定实例参数



任务实例

直观地展示当前系统中运行任务实例的状态，点击详情即可获取详细的信息，点击日志可以查看通过 `OmsLogger` 上报的日志，点击停止则可以强制终止该任务。

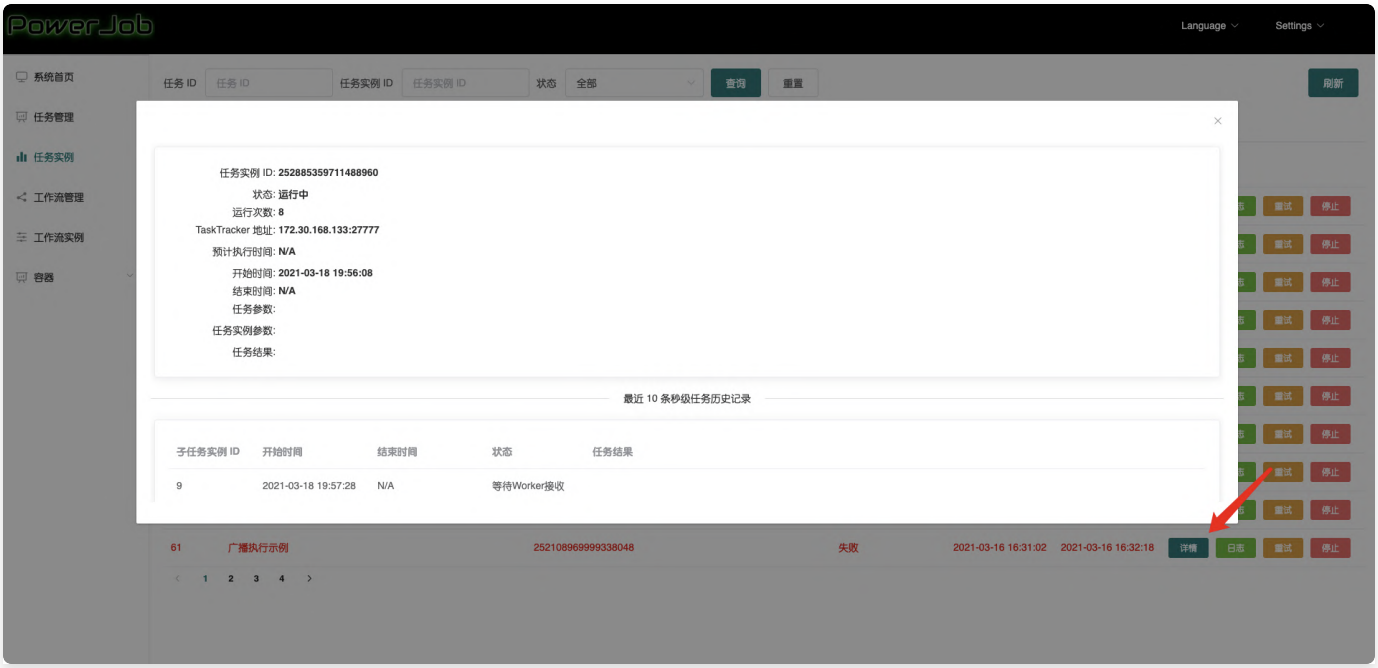
主界面



- 普通任务实例 TAB 显示由 PowerJob 直接调度的任务实例。
- 工作流任务实例 TAB 显示由 PowerJob 调度的工作流内运行的子任务实例。

任务实例详情

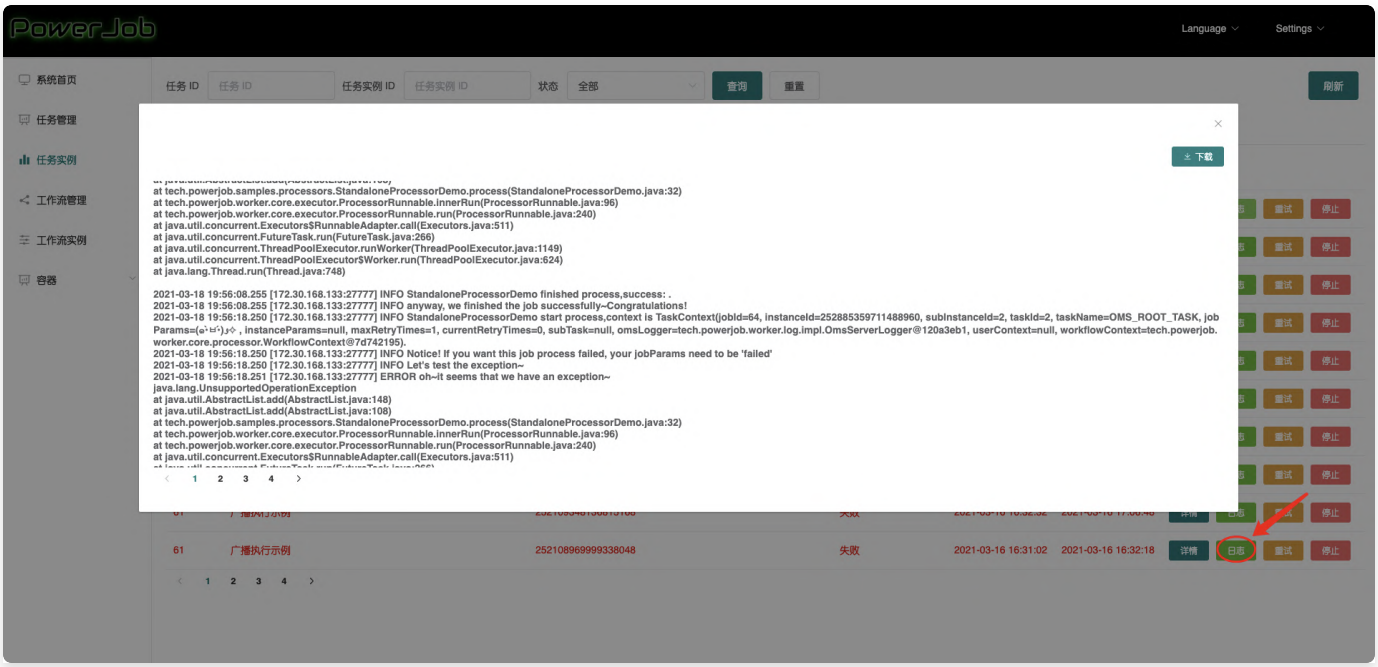
点击详情按钮，可查看任务实例的详细运行时信息。



4.3.8 及后续版本，针对 Map/MapReduce 任务，支持

在线日志

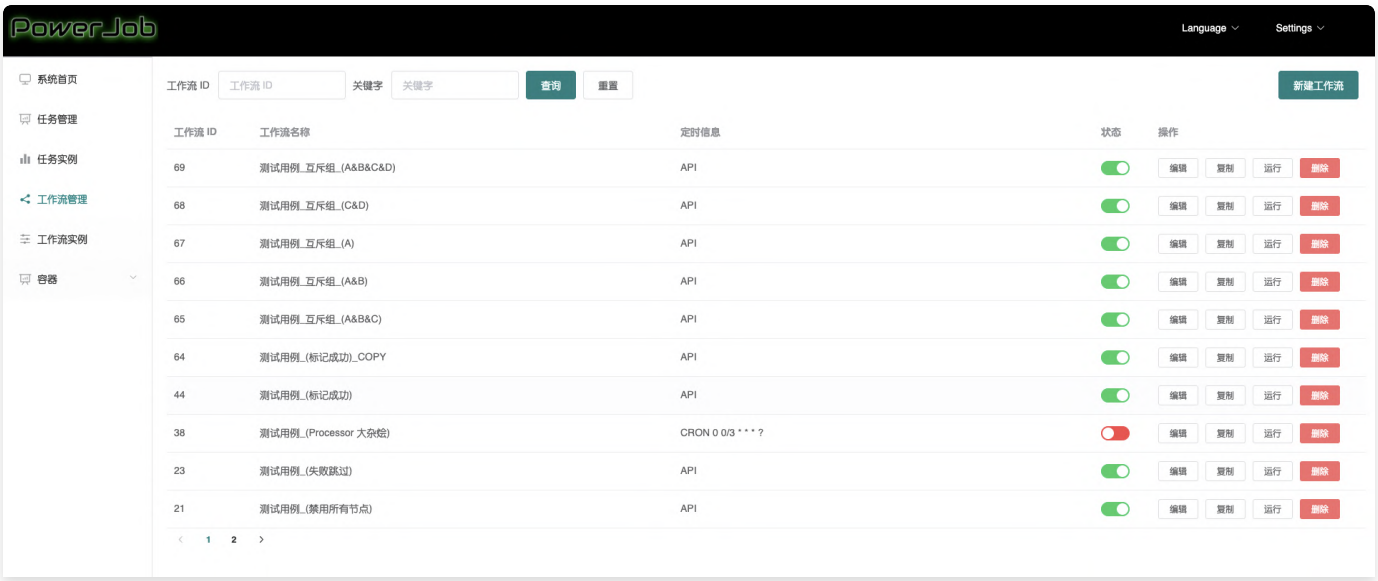
点击日志按钮，可查看任务实例的在线日志。



工作流

主界面

直观地展示当前系统所管理的所有工作流信息，并提供相应的运维方法。



新增工作流（教程，推荐仔细阅读）

workflow_edit_demo_v4.0.0.mp4

点击右上角按钮 新建工作流，即可录入新的工作流，具体界面和说明如下所示。

- 工作流名称：名称，无实际业务用途，请尽量精简字段
- 工作流描述：描述，无实际业务用途，请尽量精简字段
- 定时信息：该工作流的触发方式的触发方式，包含时间表达式类型选择框和时间表达式输入框
  - CRON –> 填写 CRON 表达式（在线生成网站）
  - API –> 不需要填写任何参数，表明该任务由 OpenAPI 触发
- 生命周期：定时策略生效的时间段
- 最大实例：该工作流同时执行的数量
- **任务依赖关系：**提供编辑界面可视化操作，绘制 DAG（有向无环图），配置工作流内各个任务的依赖关系

DAG 操作指南

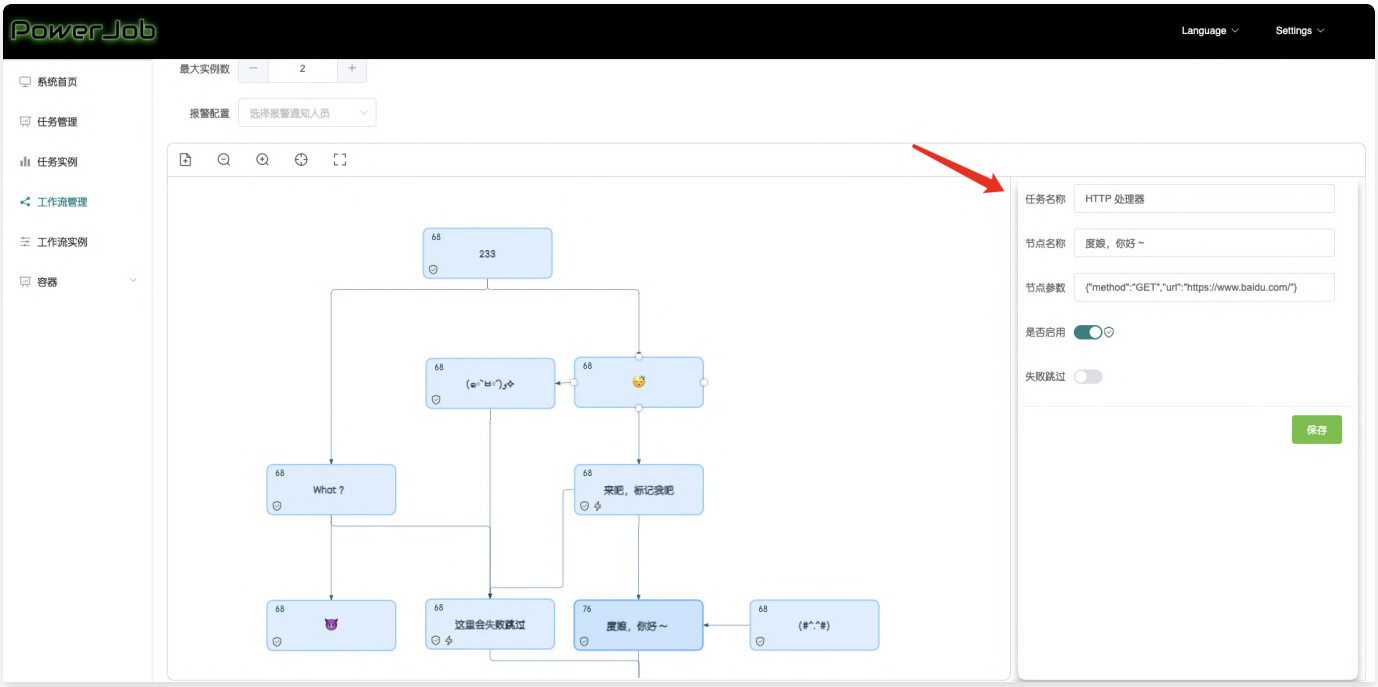
编辑依赖关系

v4.0.0 以后支持节点的自由拖拉拽，不用再点点点了，哈哈哈 ~

- **添加节点**：点击 DAG 编辑框左上方的“**导入任务**”，导入当前存在的任务（需要提前在 任务管理界面 录入任务），生成 DAG 的节点
- **连接节点**：点击起始节点的任意一个锚点摁住不放，拖动鼠标连接到另一个节点的任意一个锚点即可
- **删除节点**：选中需要删除的节点，按退格键（ **注意：windows 下使用退格键 [Backspace]，macOS 下使用删除键 [delete]** ）
- **删除边**：选中需要删除的边，按退格键（ **注意：windows 下使用退格键 [Backspace]，macOS 下使用删除键 [delete]** ）

编辑节点信息

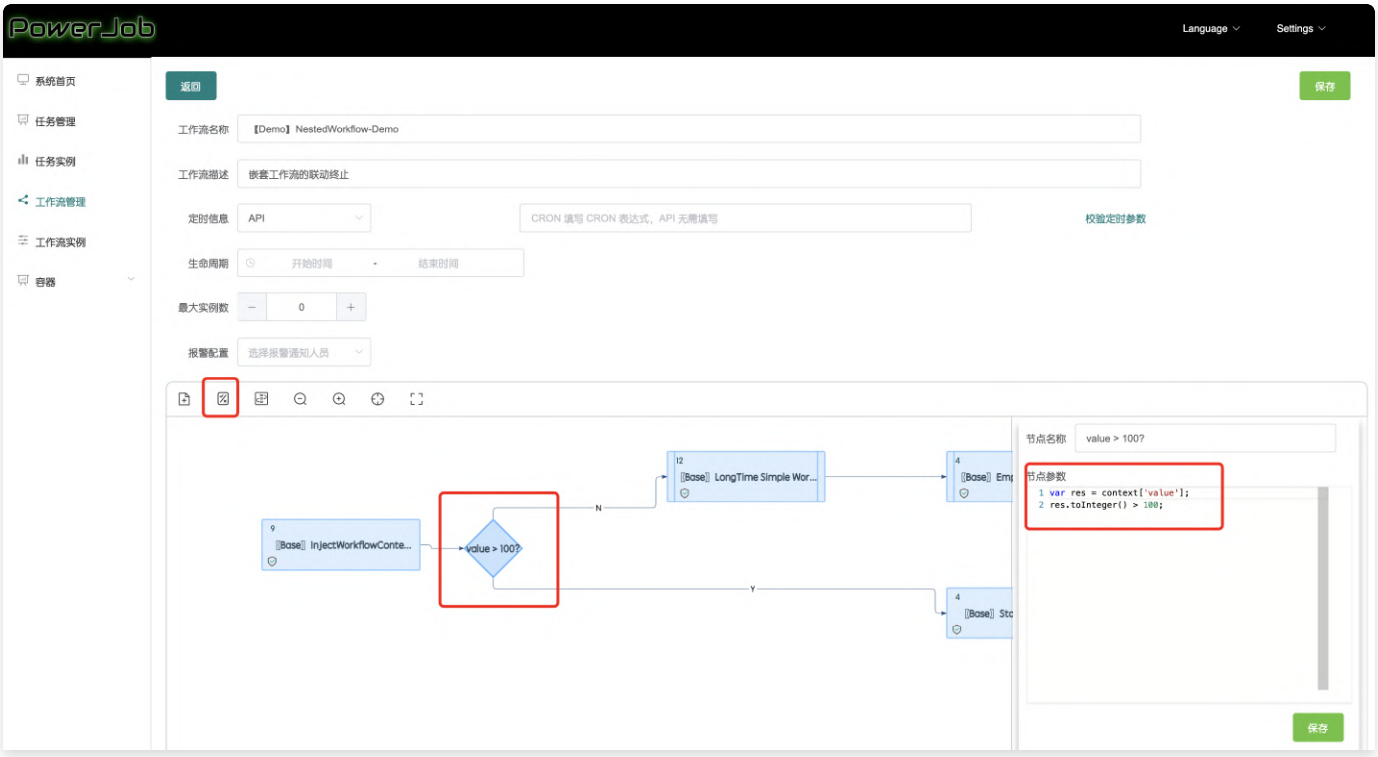
点击需要编辑的节点，在右侧会弹出一个编辑框，如下图所示



- **任务名称**：当前节点**引用**的任务名称，点击可编辑（支持输入名称进行模糊搜索）
- **节点名称**：节点的名称，无实际业务用途，在能明确表示节点背后的业务逻辑的情况下请尽量精简字段
- **节点参数**：节点的参数配置，当这个信息不为空的时候使用这个参数**覆盖**当前节点所引用的任务所配置的参数信息
- **是否启用**：未启用的节点将会直接跳过
- **失败跳过**：当这个节点执行失败的时候不会打断整个工作流的执行

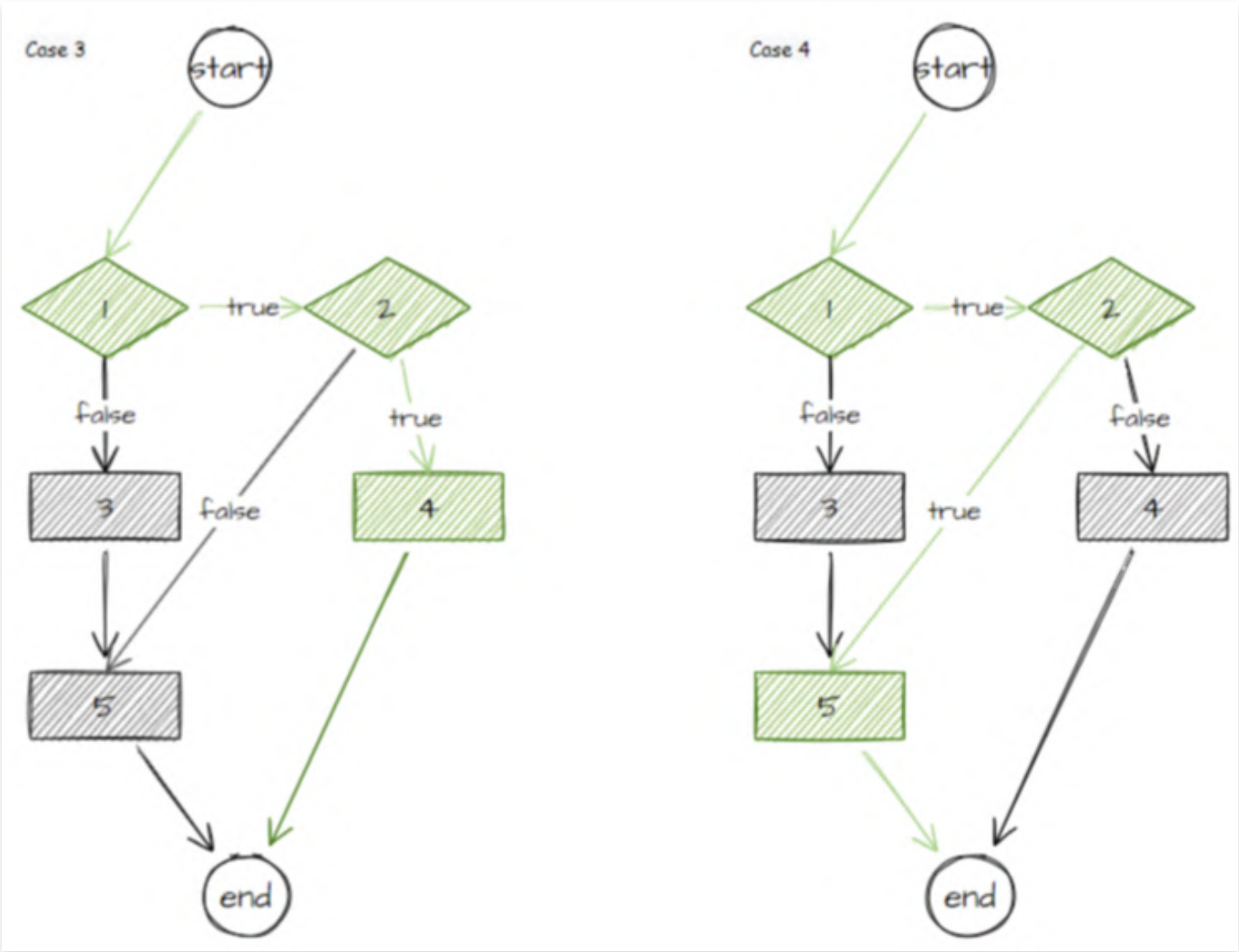
特殊节点说明

判断节点



判断节点 **不允许失败跳过以及禁用**，节点参数中存储的是 Groovy 代码（执行 Groovy 代码时会将当前工作流上下文作为 context 变量注入到代码执行的上下文中），其执行结果仅能返回 "true" 或者 "false"，同时判断节点仅有且必须有两条“输出”路径。会根据该代码的执行结果决定下游需要执行的节点。这里处理的原则是， **仅 cancel 那些只能通过被 disable 掉的边可达的节点**

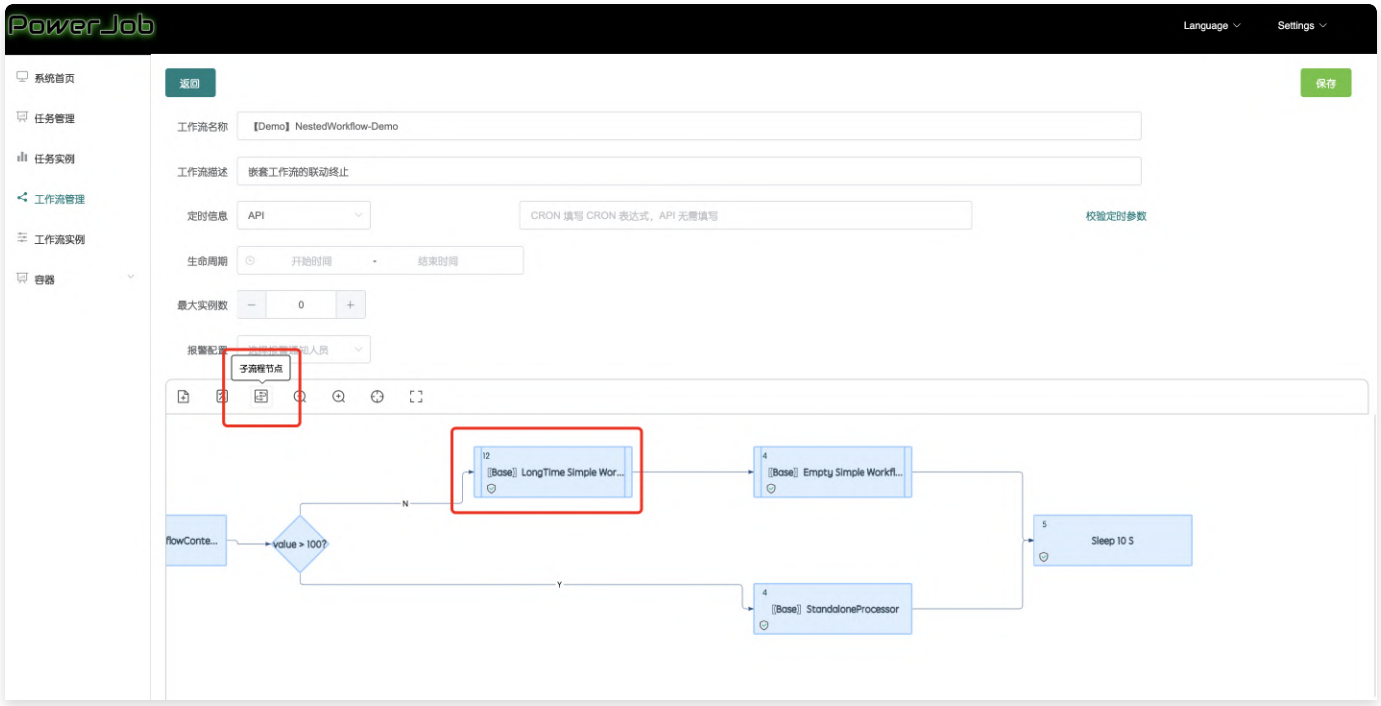
举个两个栗子，灰色代表相应的边 或者 节点被 disable 或 cancel，菱形代表判断节点，假定执行结果为 true



case 3 以及 case 4 中的节点 3 都会被 cancel ，因为它只能通过节点 1 –> 节点 3 的边可达（该边的属性为 false），但对于节点 5 而言，在 case 4 中因为判断节点 2 的执行结果为 true ，那么其可以通过节点 2 –> 节点 5 的边可达，所以不会被 disable 。

备注：如果需要根据上游节点的执行结果决定下游节点，可以将上游节点的执行结果注入上下文中，再在判断节点中做相应的判断。

工作流嵌套节点



该节点代表对某个工作流的引用，**节点的 jobId 属性存储的是工作流 id**，其他属性和普通的任务节点一致。不允许出现循环引用以及多级嵌套的情况，即**嵌套节点中指向的工作流**一定是一个**不含嵌套节点**的工作流。

执行到该节点时，如果该节点处于启用状态，那么将启动该节点所引用工作流的一个新实例，待该实例执行完成后再同步更新该节点的状态。

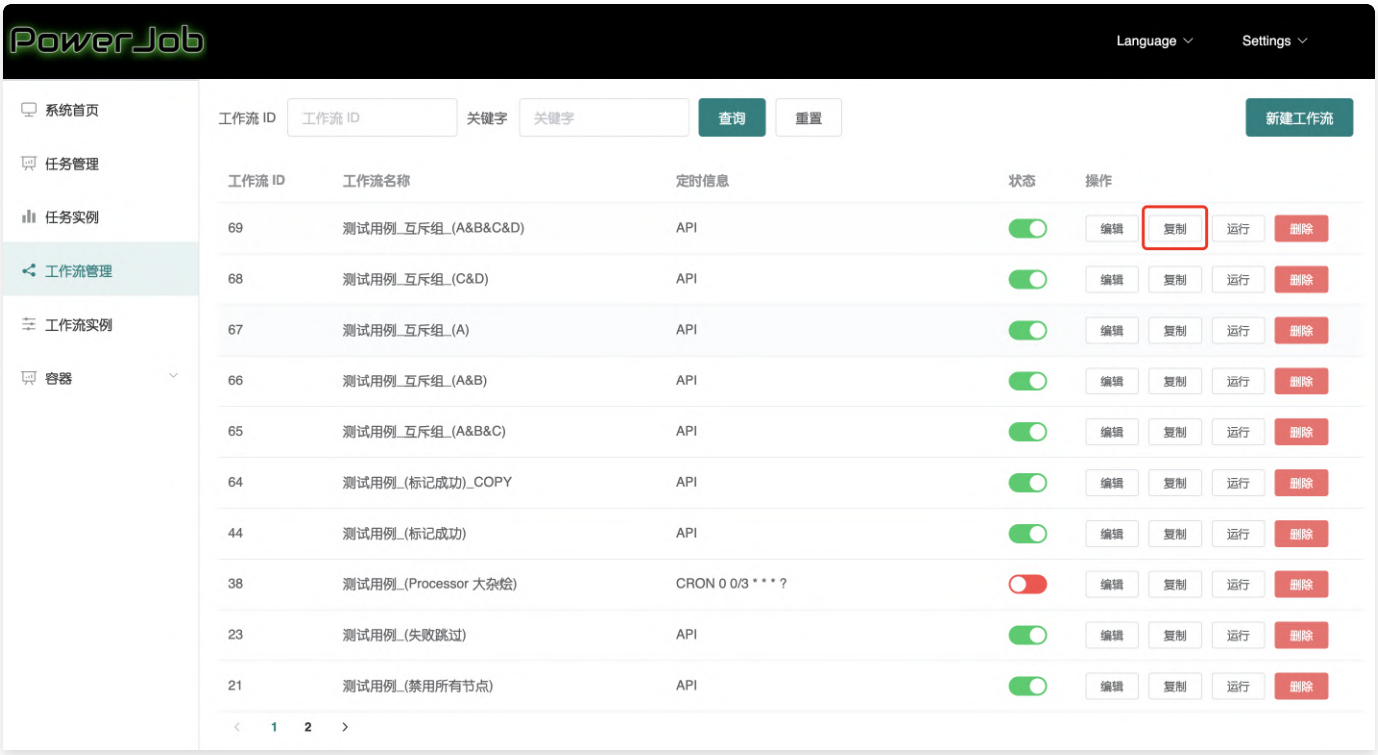
注意，创建子工作流时，会**透传**当前的上下文作为工作流的实例参数，在子工作流执行完成时会**合并**子工作流的上下文至父工作流的上下文中。

重试子工作流不会联动重试父工作流，但失败的子工作流会随着父工作流的重试而原地重试（不会生成新的实例）

复制工作流

如果需要配置一个和现有工作流**相似度很高**的工作流，墙裂建议使用工作流的复制功能，省时又省心 ~ 😎

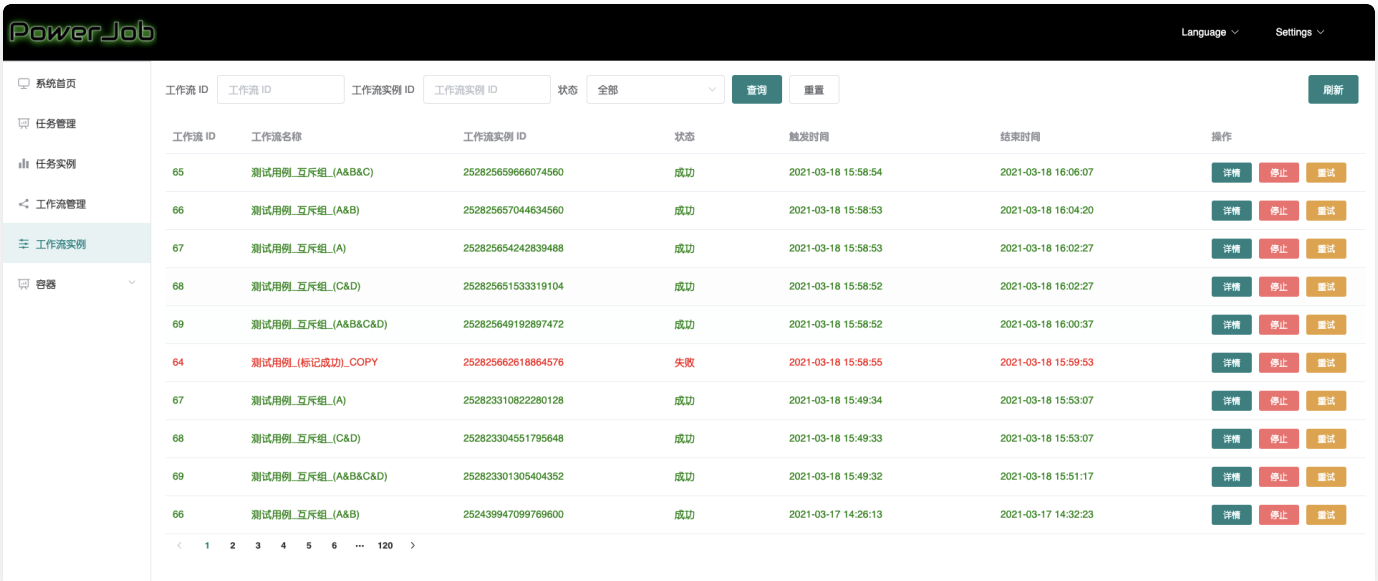
点击复制按钮即可 ~



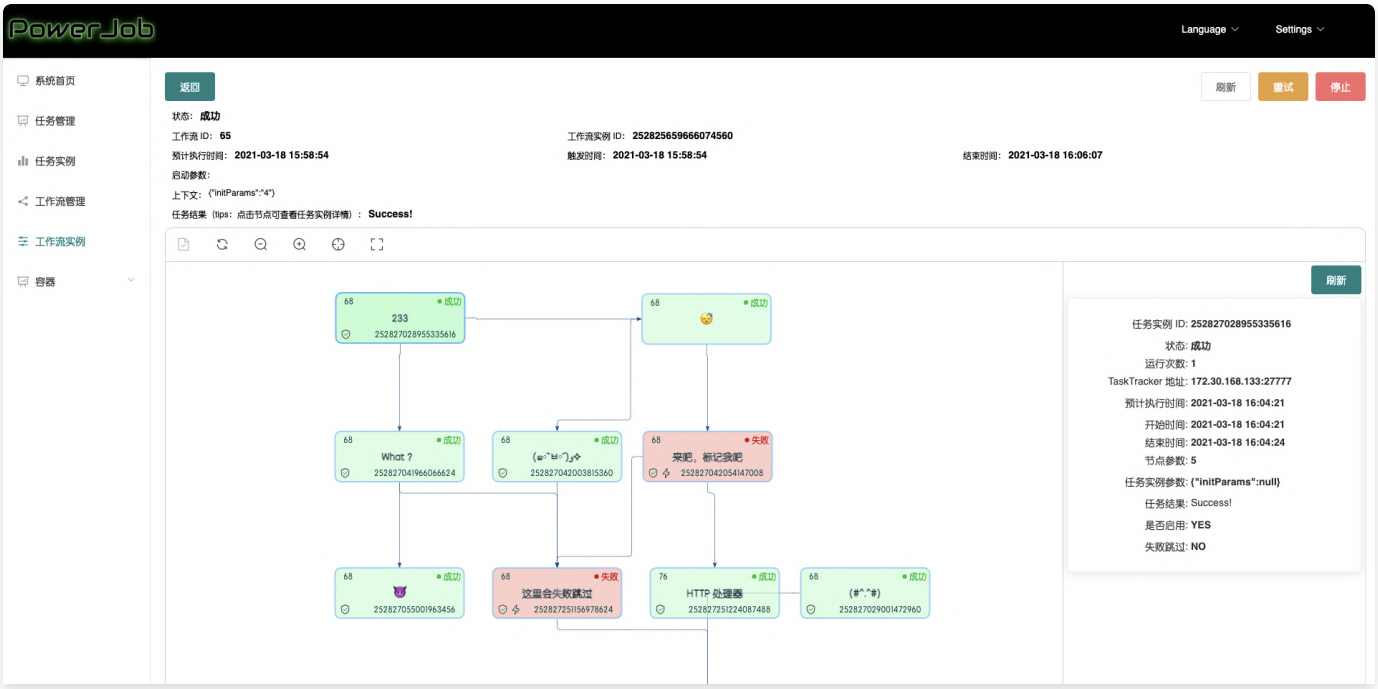
工作流实例

v4.0.0 版本大幅提升了工作流的运维能力！😎

主界面



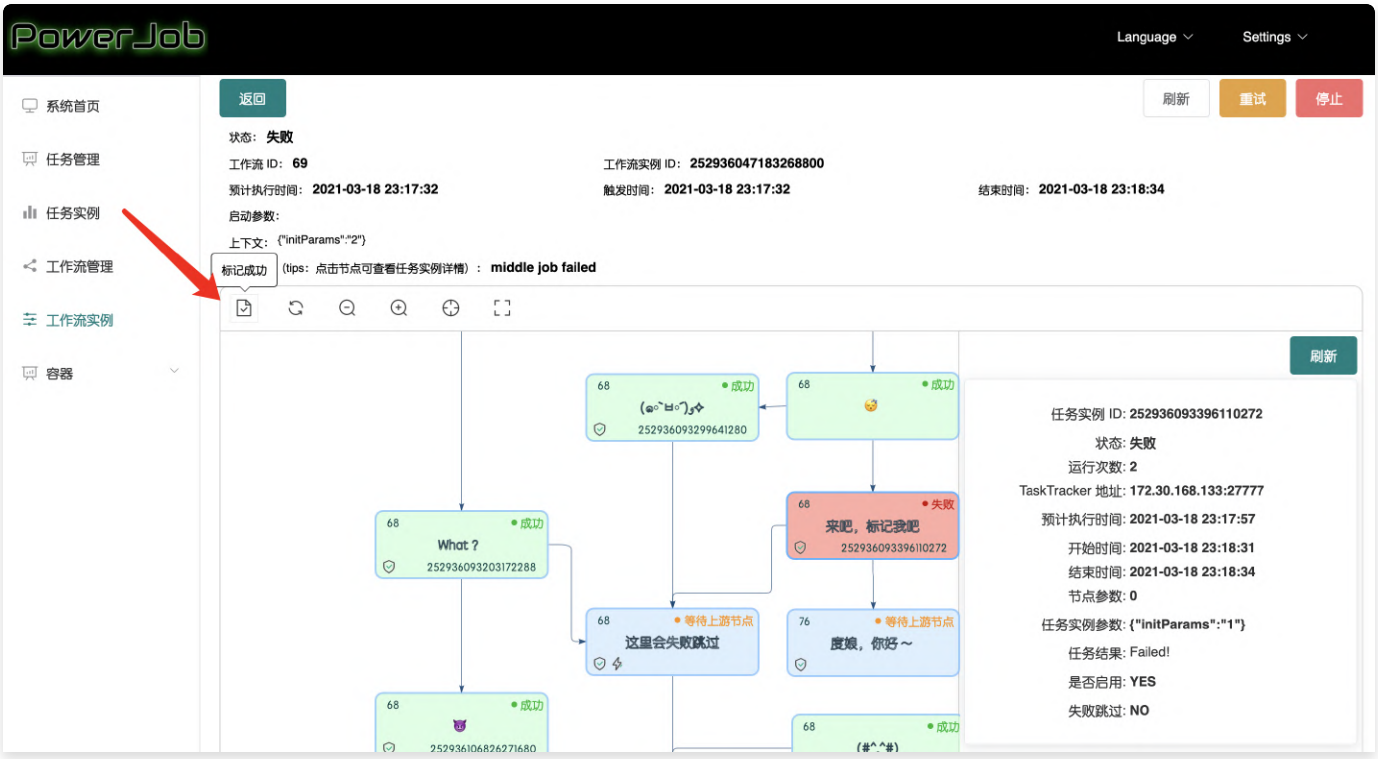
工作流实例详情



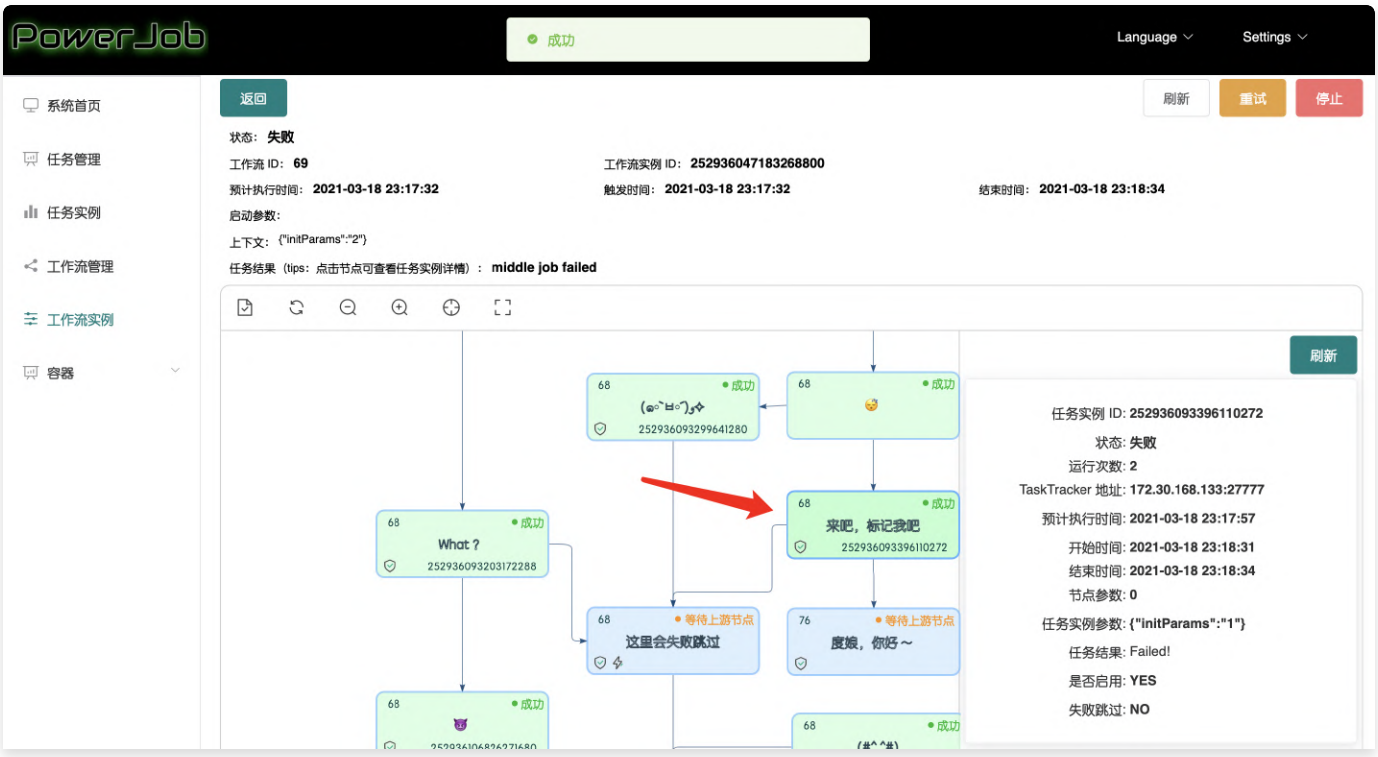
其中上方为工作流实例的基础信息，下方为实例的 DAG 信息，点击节点可以查看对应的任务实例详情信息。
右上方有三个功能按钮用于工作流的运维

刷新：重新获取当前工作流实例的信息
重试：从当前工作流实例失败（会忽略失败跳过的节点）的节点开始继续往后执行
停止：**尝试**停止当前工作流实例中所有正在执行的任务，并终止整个工作流

DAG 界面左侧上方还有个相对比较隐蔽的功能按钮 📌 **标记成功**，如下图所示



选中执行失败的节点，点击标记成功后会将其 DAG 中的节点状态置为成功（不会更改对应任务实例的状态，所以点击节点看到的任务实例详情还是失败）



该功能主要用于搭配工作流实例的重试功能实现灵活运维 （想跳过某个失败的节点进行重试）。

用户（账号体系）配置

> PowerJob 自 5.x 版本开始正式支持用户账号体系，为了方便各大开发者使用，PowerJob 原生提供了2种默认的账号体系（PowerJob账号体系、钉钉账号体系），同时基于高扩展性的设计理念，提供了允许开发者**以极低成本接入企业内部任何账号体系**的方式。

5.x 版本引入了用户与权限体系，虽说核心功能无任何变化，仅在管理端上层做了增强，但可能还是会对一些已经接入使用的用户产生一定的影响，也可能出现一些官方未及时评估到的兼容性问题。
因此 5.0.0 版本暂时以 BETA 版本的形式亮相，希望大家做到：

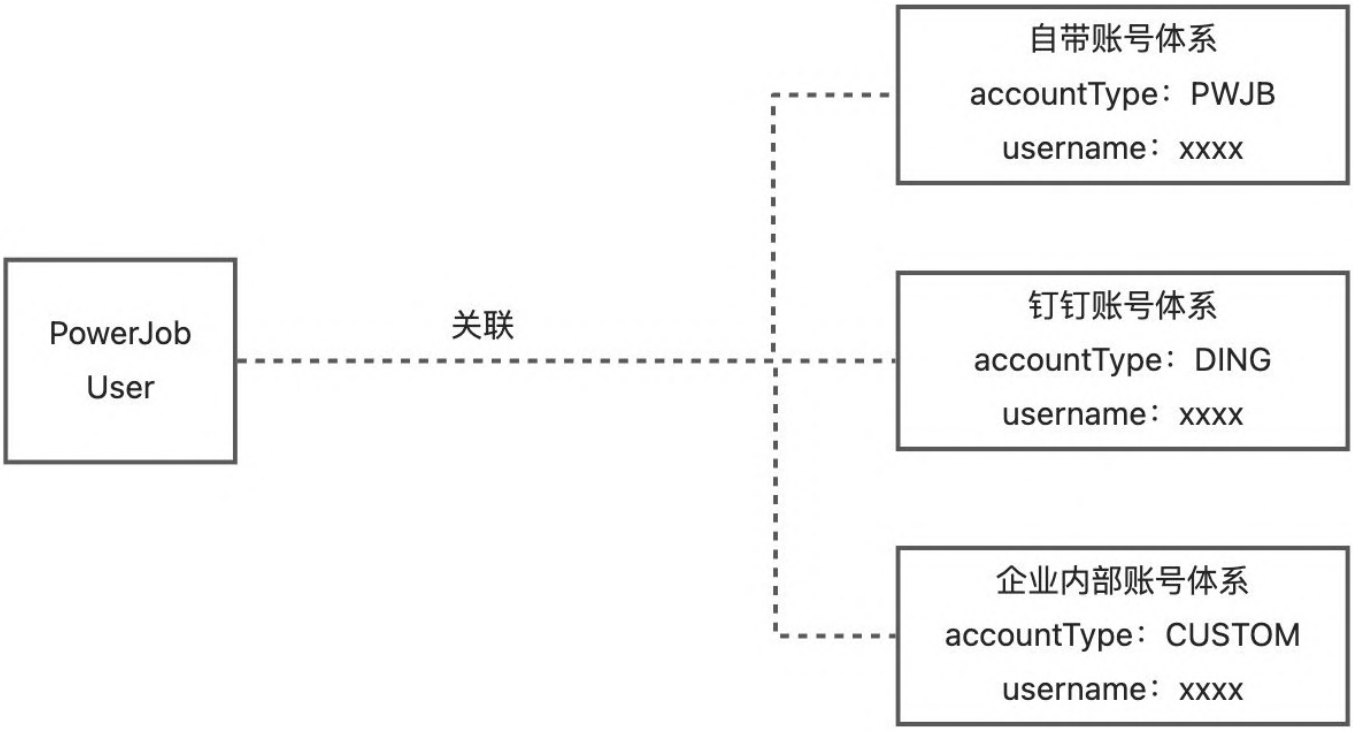
- 新用户鼓励积极尝试，做好充分的测试验证后再上生产环境。
- 老用户可阅读相关功能说明文档和迁移指南，如有条件可在不是那么核心的场景尝试升级，任何问题及时反馈到 ISSUE，避免后续版本固化后出现无法平滑升级的情况。
- BETA 期间，欢迎任何用户积极反馈问题，想法。PowerJob 致力于雕琢最极致的功能。

用户体系说明

PowerJob 用户体系的设计理念，是提供足够的开放性，让开发者能轻松接入企业已有的统一登录体系，降低使用成本，避免 N 个中间件 N 套账号密码的窘境。同时也对有安全性诉求的使用方更友好，可使用自己信赖的登录体系。

在此设计理念驱动下，PowerJob 主框架没有耦合任何一种登录方式，PowerJob 真正的 User 完全依赖第三方账号体系的同步。

举个例子，用户通过钉钉登录后，PowerJob 会感知到此回调，同步创建 accountType=DING，username=用户在钉钉内的唯一ID 的 PowerJob 账户。



权限体系说明

PowerJob 的用户权限体系，依照经典 RBAC（Role-Based Access Control，基于角色的访问控制）实现。

其中，角色类型定义如下：

- APP：权限范围收口在 APP 内部，仅拥有某个 App 的某个角色。比如拥有 App 的 DEVELOPER 权限，可对该 APP 内的任务、工作流、容器执行读、运维、写操作。
- Namespace：权限范围收口在 Namespace 纬度，包括对 namespace 的直接权限与 namespace 下所关联的全部 APP的穿透权限。比如拥有某个 namespace 的 DEVELOPER 权限，可对该 namespace 管理的全部 APP 内的任务、工作流、容器执行读、运维、写操作。
- GLOBAL：全局，目前仅开放了 GLOBAL + SU 的配置，即 PowerJob 全局超级管理员，拥有一切权限。

每个角色拥有的权限定义如下：

角色\权限	READ 读	WRITE 写	OPS 运维	SU 超级权限

OBSERVER 观察者	✅			
QA 测试人员	✅		✅	
DEVELOPER 开发者	✅	✅	✅	
Admin 管理员	✅	✅	✅	✅

支持的账号体系

PowerJob 账号体系

不过多介绍，简单的注册、登录功能。

钉钉账号体系

实现登录第三方网站 – 开放平台

本文档指导你如何实现用户登录第三方网站（扫码或账密方式）。在本场景中，第三方网站可以获取用...

钉钉开放平台

Properties

```
# 钉钉应用 AppKey
oms.auth.dingtalk.appkey=dinggzqqzqqzqqzqq
# 钉钉应用 AppSecret
oms.auth.dingtalk.appSecret=iY-FS8mzqqzqq_xEizqqzqqzqqzqqzqqzqqYEbkZOal
# 回调地址，powerjob 前端控制台地址，即 powerjob-console 地址，比如本地调试为 http://localhost:7700，部署后则为 http://try.powerjob.tech
oms.auth.dingtalk.callbackUrl=http://localhost:7700
```

自定义账号体系

> 基于 PowerJob 强大而灵活的扩展性设计，仅需要开发者自行在 powerjob-server 实现 `tech.powerjob.server.auth.login.ThirdPartyLoginService` 接口，适配企业自己的账号体系，即可接入。以下为详细说明。

接口定义

Java

```java
public interface ThirdPartyLoginService {

    /**
     * 登陆服务的类型
     * @return 登陆服务类型，比如 PowerJob / DingTalk
     */
    LoginTypeInfo loginType();

    /**
     * 生成登陆的重定向 URL
     * @param httpServletRequest http请求
     * @return 重定向地址
     */
    String generateLoginUrl(HttpServletRequest httpServletRequest);

    /**
     * 执行第三方登录
     * @param loginRequest 上下文
     * @return 登录地址
     */
    ThirdPartyUser login(ThirdPartyLoginRequest loginRequest);

    /**
     * JWT 登录的回调校验
     * @param username 用户名称
     * @param tokenLoginVerifyInfo 二次校验信息
     * @return 是否通过
     */
    default boolean tokenLoginVerify(String username, TokenLoginVerifyInfo tokenLoginVerifyInfo) {
        return true;
    }
}

```

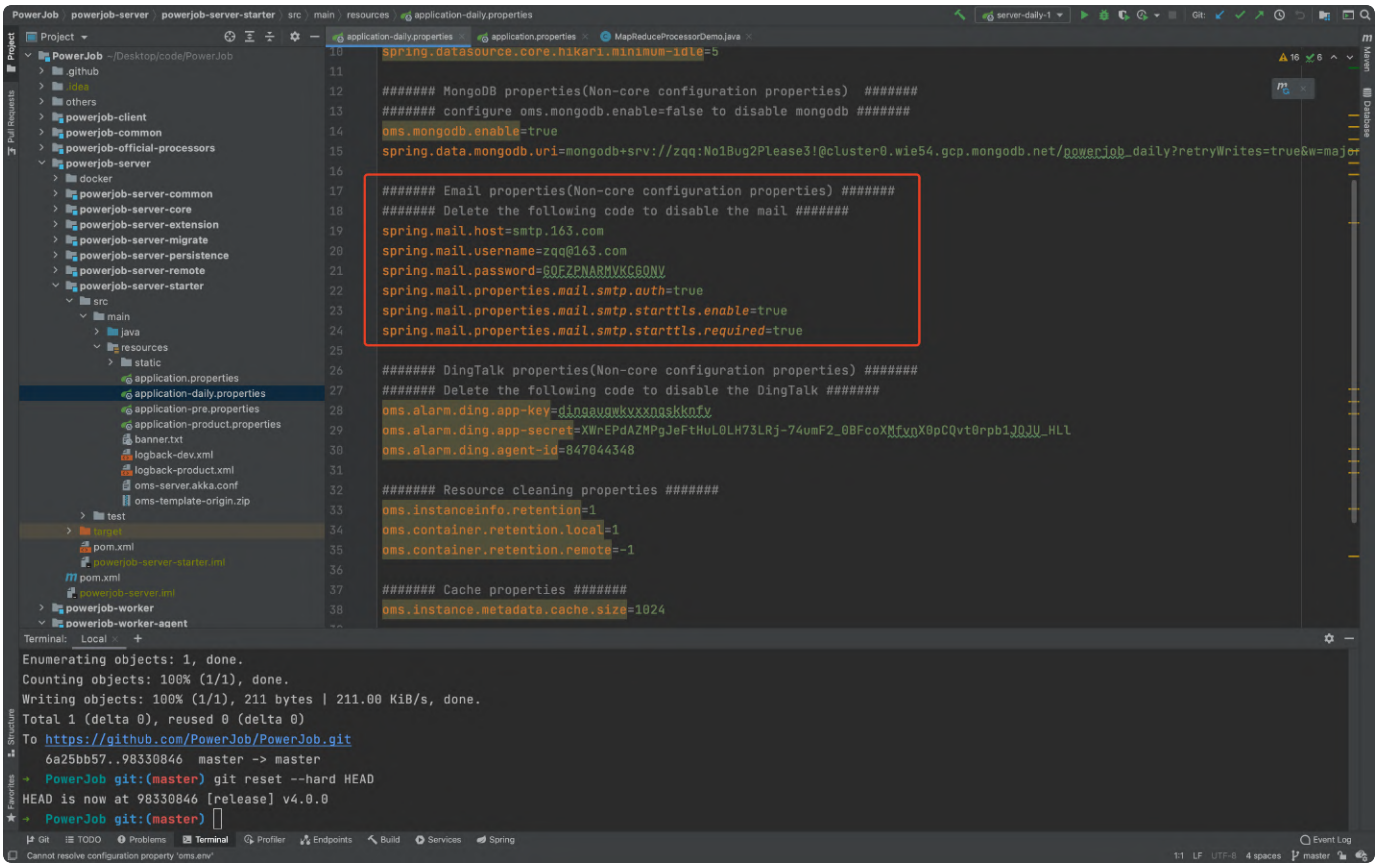
Beta 阶段文档暂时不过多描述，看官方默认提供的2个实现即可。

报警配置（邮件、WebHook、钉钉、自定义）

邮件报警

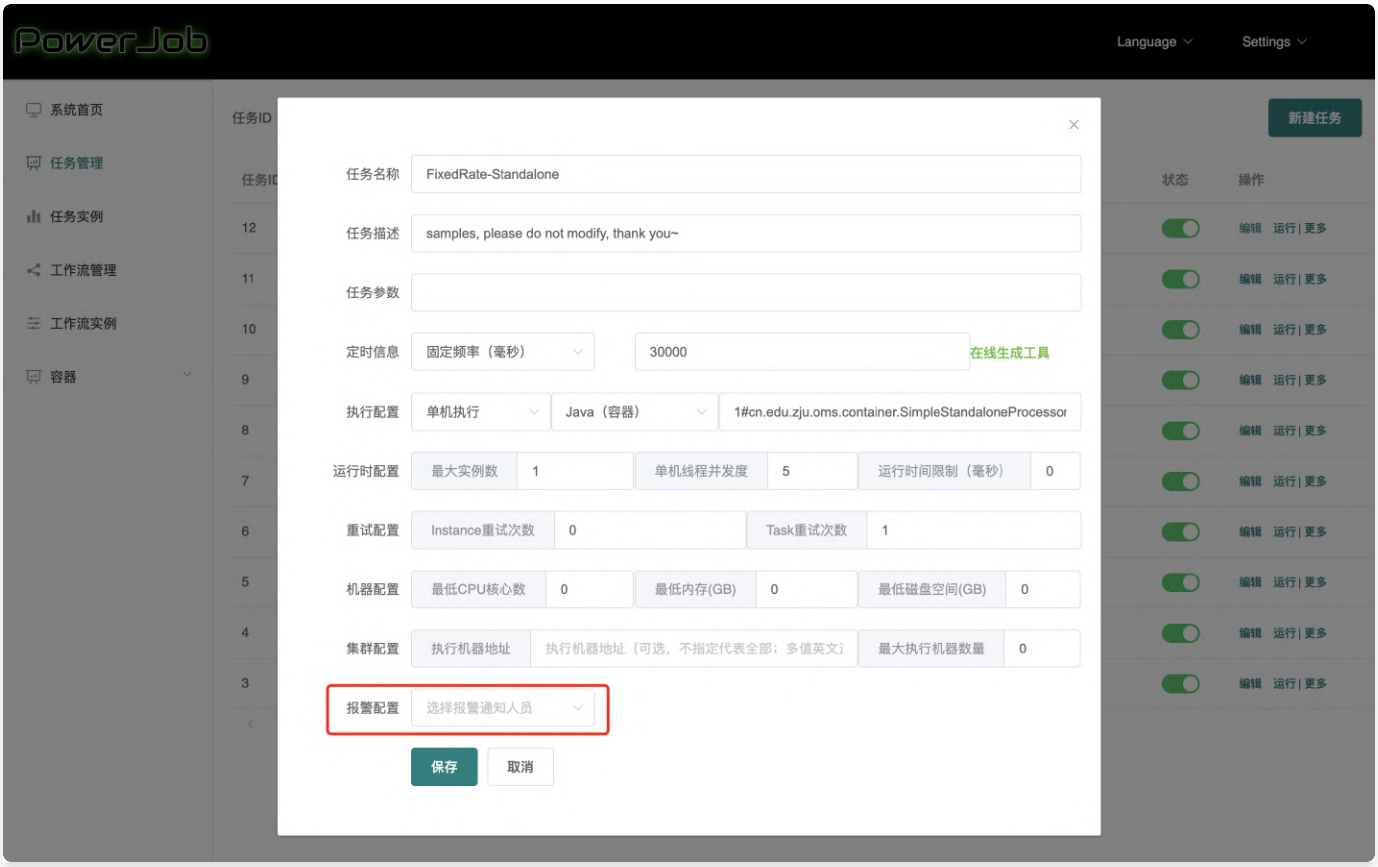
STEP1: 初始化

在 powerjob-server 中初始化邮件配置，具体教程可以搜索 SpringBoot 邮件配置。



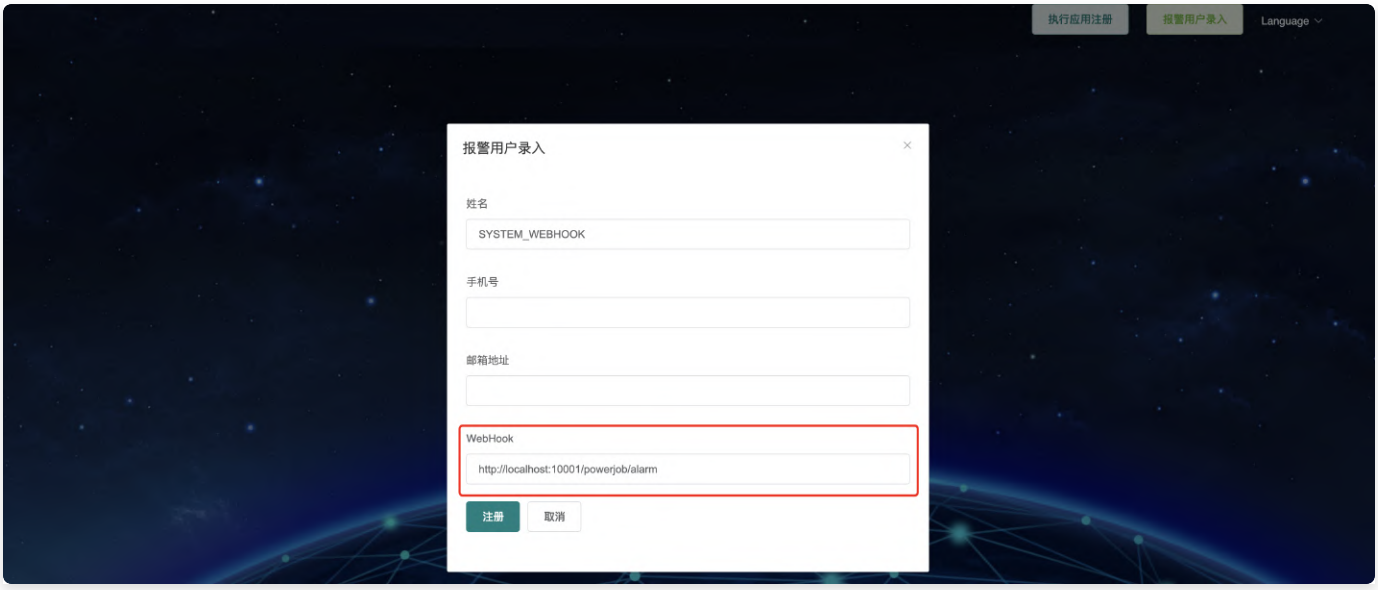
STEP2: 配置报警信息

1. 在控制台登陆界面（点击右上角 Setting – 退出 即可前往控制台登陆界面）点击 报警用户录入
2. 录入报警用户信息
3. 在任务中配置报警接收人。



WebHook

STEP1: 创建特殊角色 SYSTEM_WEBHOOK



STEP2: 开发接收 webhook 的服务端

powerjob-server 会向目标 URL 发送**包含报警任务内容**的 POST 请求，推荐使用 Map<String, Object> 接收并处理：

Java

```java
@RestController
@RequestMapping("/powerjob")
public class AlarmReceiveController {

    @PostMapping("/alarm")
    public String receiveAlarmInfo(@RequestBody Map<String, Object> params) {

        // 可以先打印下 params 看看里面具体是什么(原始对象为 JobInstanceAlarm 或 WorkflowInstanceAlarm)
        System.out.println(params);

        // 完成自定义处理

        // 返回任意值都表示调用成功
        return "success";
    }
}
```

以下为原始的参数对象：

JobInstanceAlarm（任务运行失败的告警对象）

```java
public class JobInstanceAlarm {
    // 应用ID
    private long appId;
    // 任务ID
    private long jobId;
    // 任务实例ID
    private long instanceId;
    // 任务名称
    private String jobName;
    // 任务自带的参数
    private String jobParams;
    // 时间表达式类型（CRON/API/FIX_RATE/FIX_DELAY）
    private Integer timeExpressionType;
    // 时间表达式，CRON/NULL/LONG/LONG
    private String timeExpression;
    // 执行类型，单机/广播/MR
    private Integer executeType;
    // 执行器类型，Java/Shell
    private Integer processorType;
    // 执行器信息
    private String processorInfo;

    // 任务实例参数
    private String instanceParams;
    // 执行结果
    private String result;
    // 预计触发时间
    private Long expectedTriggerTime;
    // 实际触发时间
    private Long actualTriggerTime;
    // 结束时间
    private Long finishedTime;
    // TaskTracker地址
    private String taskTrackerAddress;
}
```

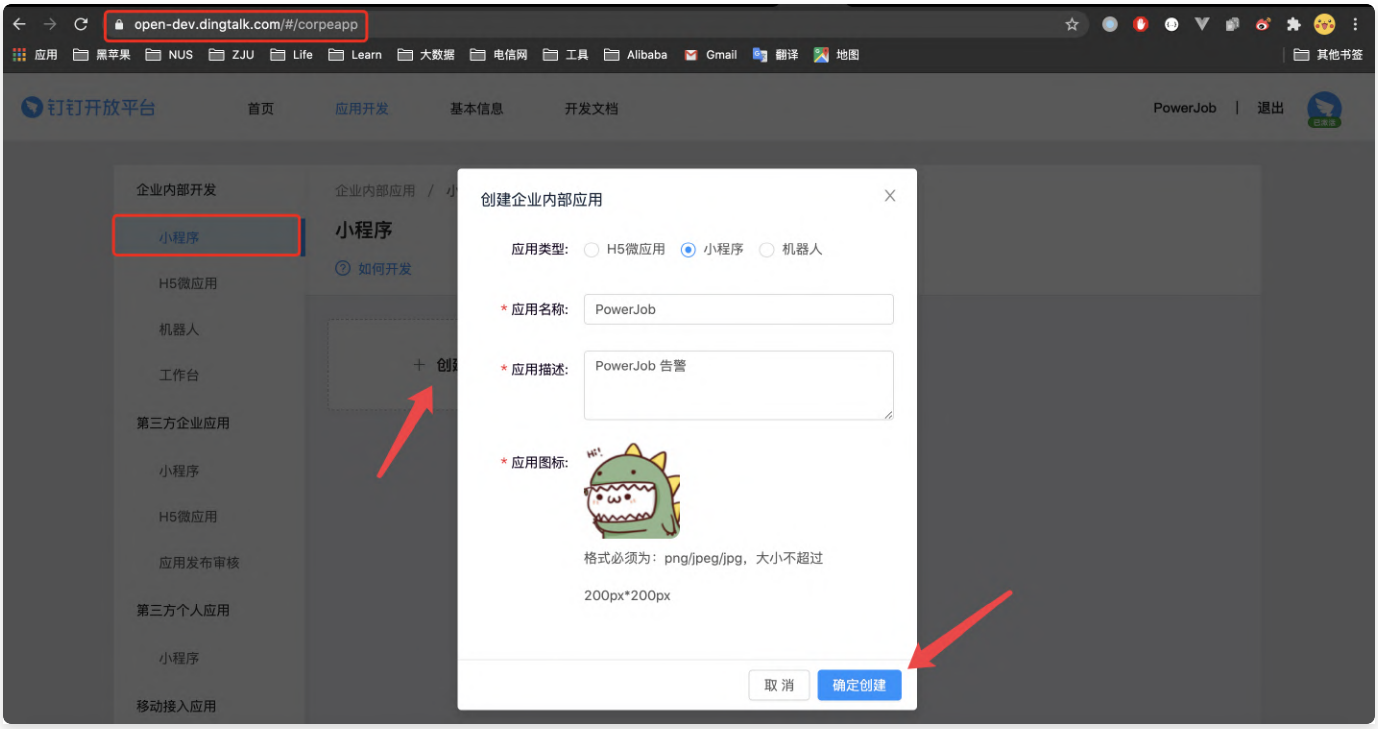
WorkflowInstanceAlarm（工作流执行失败告警对象）

```java
public class WorkflowInstanceAlarm {

    private String workflowName;

    // 任务所属应用的ID，冗余提高查询效率
    private Long appId;
    private Long workflowId;
    // workflowInstanceId（任务实例表都使用单独的ID作为主键以支持潜在的分表需求）
    private Long wfInstanceId;
    // workflow 状态（WorkflowInstanceStatus）
    private Integer status;

    private PEWorkflowDAG peWorkflowDAG;
    private String result;

    // 实际触发时间
    private Long actualTriggerTime;
    // 结束时间
    private Long finishedTime;

    // 时间表达式类型（CRON/API/FIX_RATE/FIX_DELAY）
    private Integer timeExpressionType;
    // 时间表达式，CRON/NULL/LONG/LONG
    private String timeExpression;
}

```

钉钉报警

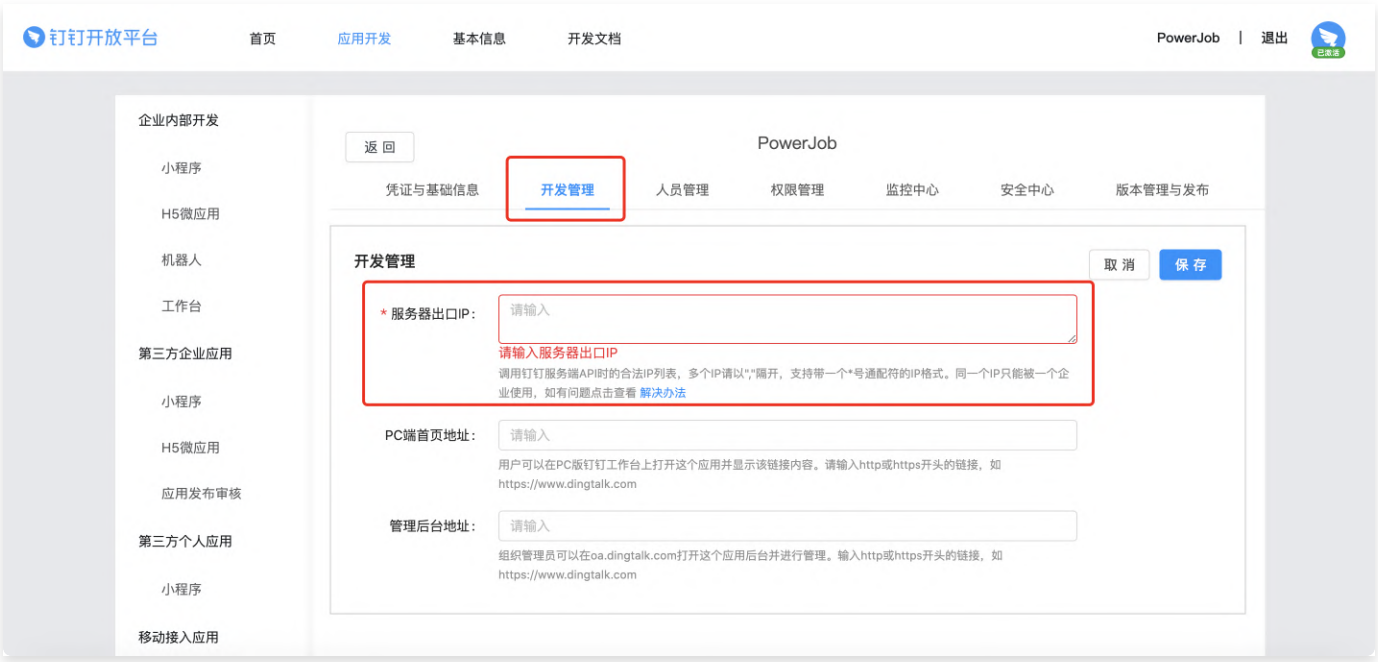
STEP1: 创建企业内部应用：PowerJob

1. 前往 钉钉开放平台，登陆企业账号，创建小程序，填入应用名称（推荐使用 PowerJob）、描述和图标。具体可参考下图。



2. 点击创建的小程序，进入小程序管理界面，查看应用凭证，将相关的参数复制到 powerjob-server 的配置文件中（oms.alarm.ding.agent-id、oms.alarm.ding.app-key 和 oms.alarm.ding.app-secret）。

3. 配置服务器出口 IP。根据钉钉限制，只有指定 IP 的服务器发送的消息才能被用户接收。因此需要前往 开发管理 tab 页录入 **服务器出口IP**。（注意，是服务器出口 IP 而不是 服务器本机 IP，也就是要 **获得服务器的公网 IP**）（一句话教程：登陆服务器执行 curl ip.sb，不会或者命令失效请自行百度）



4. 开通小程序权限。进入权限管理 tab 页，点击添加接口权限，找到“**手机号获取userid**”并添加。

STEP2: 配置报警信息

1. 在控制台登陆界面（点击右上角 Setting – 退出 即可前往控制台登陆界面）点击 报警用户录入
2. 录入报警用户信息，其中 手机号需要关联企业钉钉账户。
3. 在任务中配置报警接收人。

定制开发其他的报警能力

powerjob-server 是一个“简单”的 SpringBoot 应用，因此如果系统自带的报警能力无法满足您的需求，您可以自己动手完成想要的报警能力开发（比如短信报警、电话报警、企业微信等）。您需要做的只有一件事：

实现 `tech.powerjob.server.extension.Alarmable` 接口，并将实现类对象的创建交由 Spring 管理（添加 @Service 或 @Component 注解）

容器

什么是容器?

介绍

PowerJob 的容器技术允许开发者开发**独立于 Worker 项目之外 Java 处理器**，简单来说，就是以 Maven 工程项目的维度去组织一堆 Java 文件（开发者开发的众多脚本处理器），进而兼具开发效率和可维护性。

该容器为 **JVM 级容器**，而不是操作系统级容器（Docker）。

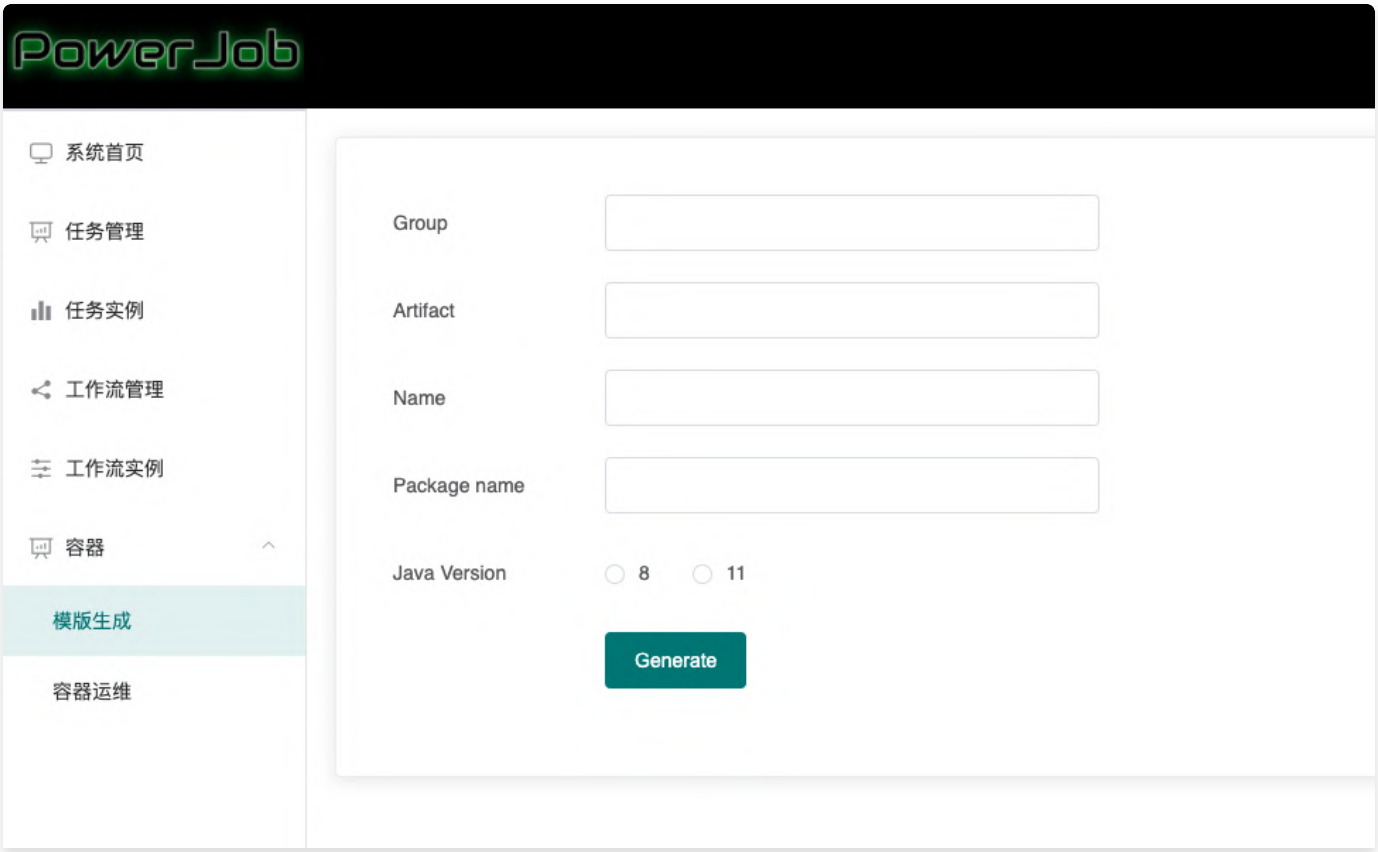
用途举例

- 比如，突然出现了某个数据库数据清理任务，与主业务无关，写进原本的项目工程中不太优雅，这时候就可以单独创建一个用于数据操作的容器，在里面完成处理器的开发，通过 PowerJob 的容器部署技术在 Worker 集群上被加载执行。
- 比如，常见的日志清理啊，机器状态上报啊，对于广大 Java 程序员来说，也许并不是很会写 shell 脚本，此时也可以借用 **agent+容器** 技术，利用 Java 完成各项原本需要通过脚本进行的操作。

（感觉例子举的都不是很好...这个东西嘛，只可意会不可言传，大家努力理解一下吧～超好用哦～）

生成容器模版

为了方便开发者使用，最新版本的前端页面已经支持容器工程模版的自动生成，开发者仅需要填入相关信息即可下载容器模版开始开发。



- Group：对应 Maven 的 `<groupId>` 标签，一般填入倒写的公司域名。
- Artifact：对应 Maven 的 `<artifactId>` 标签，填入代表该容器的唯一标示。
- Name：对应 Maven 的 `<name>` 标签，填入该容器名称。
- Package Name：包名，代表了的容器工程内部所使用的包名，**警告：包名一旦生成后，请勿更改！否则会导致运行时容器加载错误**（当然，如有必须修改包名的需求，可以尝试替换 `/resource` 下以 `oms-worker-container` 开头的所有文件相关的值）。
- Java Version：容器工程的 Java 版本，**请务必与容器目标部署 Worker 平台的 Java 版本保持一致**。

开发容器工程

完成容器模版创建后，下载，解压，会得到如下结构的 Java 工程：

```
oms-template-origin // 工程名称，可以自由更改
├── pom.xml
└── src
    ├── main
    │   ├── java
    │   │   └── cn
    │   │       └── edu
    │   │           └── zju
    │   │               └── tjq
    │   │                   └── container
    │   │                       └── samples // 所有处理器代码必须位于该目录下，其余类随意
    │   └── resources // 严禁随意更改以下两个配置文件（允许添加，不允许更改现有内容）
    │       ├── oms-worker-container-spring-context.xml
    │       └── oms-worker-container.properties
    └── test
        └── java
```

之后便可以愉快地在**指定包（packageName 对应路径）**下编写处理器代码啦～

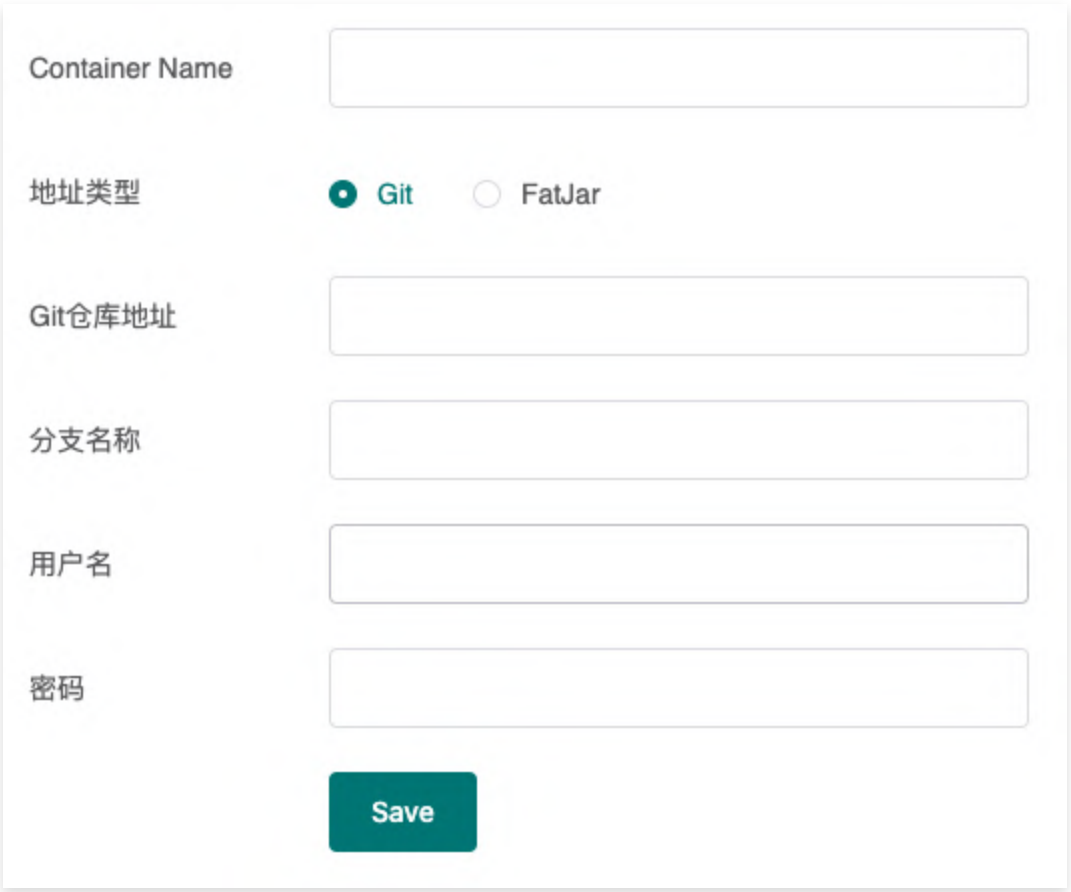
注：开发容器工程前，请检查 pom.xml 文件，确保 powerjob.worker.version 与实际部署应用的 worker 版本一致！

创建容器

>目前，PowerJob 支持使用 **Git 代码库**和 **FatJar** 来创建容器。创建路径：**容器运维 -> 容器管理 -> 新建容器。**

当使用 **Git 代码库**创建容器时，powerjob-server 需要完成代码库的下载、编译、构建和上传，**因此需要其运行环境包含可用的 Git 和 Maven 环境（包括私服的访问权限）**。

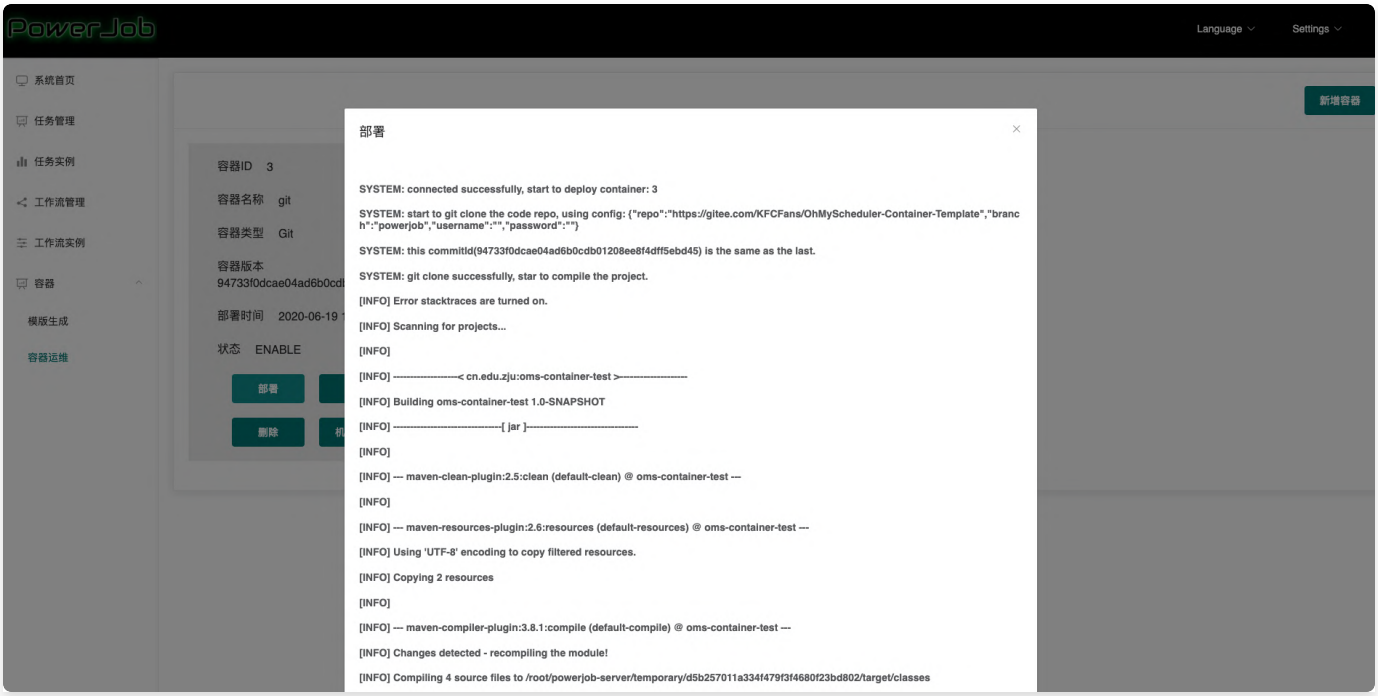
下图为使用**Git代码库**创建容器的示例，需要填入容器名称和代码库信息等参数：



下图为使用 **FatJar** 创建容器的示例，需要上传可用的 **FatJar**（注：FatJar 内必须包含了所有依赖的 Jar 文件）

部署容器

每次创建和修改容器后，都需要进行一次部署（相当于发布该容器的最新版本），使 powerjob-worker 动态加载容器内的 Processor。点击**部署**，可以看到详细的部署信息。



部署完成后，可以点击**机器列表**查看已部署该容器的机器信息。

OpenAPI

OpenAPI 允许开发者通过接口来完成手工的操作，让系统整体变得更加灵活。开发者可以基于 API 便捷地扩展PowerJob 原有的功能，比如，全面定制自己的任务调度策略。

换句话说，通过 OpenAPI，可以让接入方自己实现 PowerJob 的整个任务管理与调度模块。

依赖

最新依赖版本请参考 Maven 中央仓库：推荐地址 & 备用地址。

XML

```xml
<dependency>
  <groupId>tech.powerjob</groupId>
  <artifactId>powerjob-client</artifactId>
  <version>${latest.powerjob.version}</version>
</dependency>
```

简单示例

通过 OpenAPI 停止某个任务实例。

XML

```
// 初始化 client，需要server地址和应用名称作为参数
PowerJobClient client = new PowerJobClient("127.0.0.1:7700", "oms-test", "password");
// 调用相关的API
client.stopInstance(1586855173043L)
```

API列表

任务（Job）相关

创建/修改任务

修改（更新）时也需要填写全部参数，不能只填写修改字段！

接口签名：`ResultDTO<Long> saveJob(SaveJobInfoRequest request)`

入参：任务信息（详细说明见下表，也可以参考 前端任务创建各参数的正确填法）

返回值：ResultDTO，根据 success 判断操作是否成功。若操作成功，data 字段返回任务ID

属性	说明
jobId	任务 ID，可选，null 代表创建任务，否则填写需要修改的任务 ID
jobName	任务名称
jobDescription	任务描述
jobParams	任务参数，Processor#process 方法入参 `TaskContext` 对象的 jobParams 字段
timeExpressionType	时间表达式类型，枚举值
timeExpression	时间表达式，填写类型由 timeExpressionType 决定，比如 CRON 需要填写 CRON 表达式
executeType	执行类型，枚举值
processorType	处理器类型，枚举值
processorInfo	处理器参数，填写类型由 processorType 决定，如Java 处理器需要填写全限定类名，如：com.github.kfcfans.oms.processors.demo.MapReduceProcessorDemo
maxInstanceNum	最大实例数，该任务同时执行的数量（任务和实例就像是类和对象的关系，任务被调度执行后被称为实例）
concurrency	单机线程并发数，表示该实例执行过程中每个 Worker 使用的线程数量
instanceTimeLimit	任务实例运行时间限制，0 代表无任何限制，超时会被打断并判定为执行失败

instanceRetryNum	任务实例重试次数，整个任务失败时重试，代价大，不推荐使用
taskRetryNum	Task 重试次数，每个子 Task 失败后单独重试，代价小，推荐使用
minCpuCores	最小可用 CPU 核心数，CPU 可用核心数小于该值的 Worker 将不会执行该任务，0 代表无任何限制
minMemorySpace	最小内存大小（GB），可用内存小于该值的 Worker 将不会执行该任务，0 代表无任何限制
minDiskSpace	最小磁盘大小（GB），可用磁盘空间小于该值的 Worker 将不会执行该任务，0 代表无任何限制
designatedWorkers	指定机器执行，设置该参数后只有列表中的机器允许执行该任务，空代表不指定机器
maxWorkerCount	最大执行机器数量，限定调动执行的机器数量，0代表无限制
notifyUserIds	接收报警的用户 ID 列表
enable	是否启用该任务，未启用的任务不会被调度
dispatchStrategy	调度策略，枚举，目前支持随机（RAMDOM）和 健康度优先（HEALTH_FIRST）
lifecycle	生命周期（预留，用于指定定时调度任务的生效时间范围）
extra	扩展字段（供开发者使用，用于功能扩展，powerjob 自身不会使用该字段）

查找任务

接口签名：`ResultDTO<JobInfoDTO> fetchJob(Long jobId)`

入参：任务ID

返回值：根据 success 判断操作是否成功，若请求成功则返回任务的详细信息

禁用某个任务

接口签名：`ResultDTO<Void> disableJob(Long jobId)`

入参：任务 ID

返回值：根据 success 判断操作是否成功

启用某个任务

接口签名：`ResultDTO<Void> enableJob(Long jobId)`

入参：任务 ID

返回值：根据 success 判断操作是否成功

删除某个任务

接口签名：`ResultDTO<Void> deleteJob(Long jobId)`

入参：任务 ID

返回值：根据 success 判断操作是否成功

运行某个任务（支持延迟执行）

接口签名：`ResultDTO<Long> runJob(Long jobId, String instanceParams, long delayMS)`

入参：任务 ID + **任务实例参数**（Processor#process 方法入参 `TaskContext` 对象的 instanceParams 字段）+ **延迟执行时间**

返回值：根据 success 判断操作是否成功，操作成功返回对应的任务实例 ID (instanceId)

任务实例（Instance）相关

取消某个定时任务实例

接口签名：`ResultDTO<Void> cancelInstance(Long instanceId)`

入参：任务实例 ID

返回值：根据 success 判断操作是否成功

停止某个任务实例

接口签名：`ResultDTO<Void> stopInstance(Long instanceId)`

入参：任务实例 ID

返回值：根据 success 判断操作是否成功

查询某个任务实例

接口签名：`ResultDTO<InstanceInfoDTO> fetchInstanceInfo(Long instanceId)`

入参：任务实例 ID

返回值：根据 success 判断操作是否成功，操作成功返回任务实例的详细信息

查询某个任务实例的状态

接口签名：`ResultDTO<Integer> fetchInstanceStatus(Long instanceId)`

入参：任务实例 ID

返回值：根据 success 判断操作是否成功，操作成功返回任务实例的状态码，对应的枚举为：InstanceStatus

工作流（Workflow）相关

创建/修改工作流

修改（更新）时也需要填写全部参数，不能只填写修改字段！

接口签名：`ResultDTO<Long> saveWorkflow(SaveWorkflowRequest workflowInfo)`

入参：工作流信息

返回值：ResultDTO，根据 success 判断操作是否成功。若操作成功，data 字段返回工作流 ID

属性	说明
id	工作流 ID，可选，仅更新时需要（ id 为 null 代表新增数据，id 不为 null 代表更新）
wfName	工作流名称
wfDescription	工作流描述
dag	工作流中的任务依赖配置，点+线构成的 DAG 描述
timeExpressionType	时间表达式类型，枚举值，仅支持 CRON 和 API

timeExpression	时间表达式，填写类型由 timeExpressionType 决定，比如 CRON 需要填写 CRON 表达式
maxWfInstanceNum	最大工作流实例数
notifyUserIds	接收报警的用户 ID 列表
enable	是否启用该工作流，未启用的工作流不会被调度

查找工作流

接口签名：`ResultDTO<JobInfoDTO> fetchWorkflow(Long workflowId)`

入参：工作流 ID

返回值：根据 success 判断操作是否成功，若请求成功则返回工作流的详细信息

禁用某个工作流

接口签名：`ResultDTO<Void> disableWorkflow(Long workflowId)`

入参：工作流 ID

返回值：根据 success 判断操作是否成功

启用某个工作流

接口签名：`ResultDTO<Void> enableWorkflow(Long workflowId)`

入参：工作流 ID

返回值：根据 success 判断操作是否成功

删除某个工作流

接口签名：`ResultDTO<Void> deleteWorkflow(Long workflowId)`

入参：工作流 ID

返回值：根据 success 判断操作是否成功

立即运行某个工作流

接口签名：`ResultDTO<Long> runWorkflow(Long workflowId)`

入参：工作流 ID

返回值：根据 success 判断操作是否成功，操作成功返回对应的工作流实例 ID (wfInstanceId)

运行某个工作流（指定初始参数、延迟）

接口签名：`ResultDTO<Long> runWorkflow(Long workflowId, String initParams, long delayMS)`

入参：工作流 ID（workflowId），初始参数（initParams），延迟（delayMS）

返回值：根据 success 判断操作是否成功，操作成功返回对应的工作流实例 ID (wfInstanceId)

工作流实例（WorkflowInstance）相关

停止某个工作流实例

接口签名：`ResultDTO<Void> stopWorkflowInstance(Long wfInstanceId)`

入参：工作流实例 ID (wfInstanceId)

返回值：根据 success 判断操作是否成功

查询某个工作流任务实例

接口签名：`ResultDTO<WorkflowInstanceInfoDTO> fetchWorkflowInstanceInfo(Long wfInstanceId)`

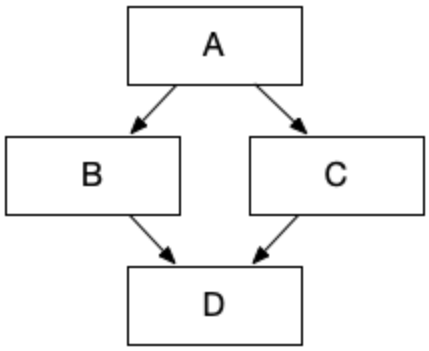
入参：工作流实例 ID (wfInstanceId)

返回值：根据 success 判断操作是否成功，操作成功返回工作流实例的详细信息

工作流（workflow）

什么是工作流?

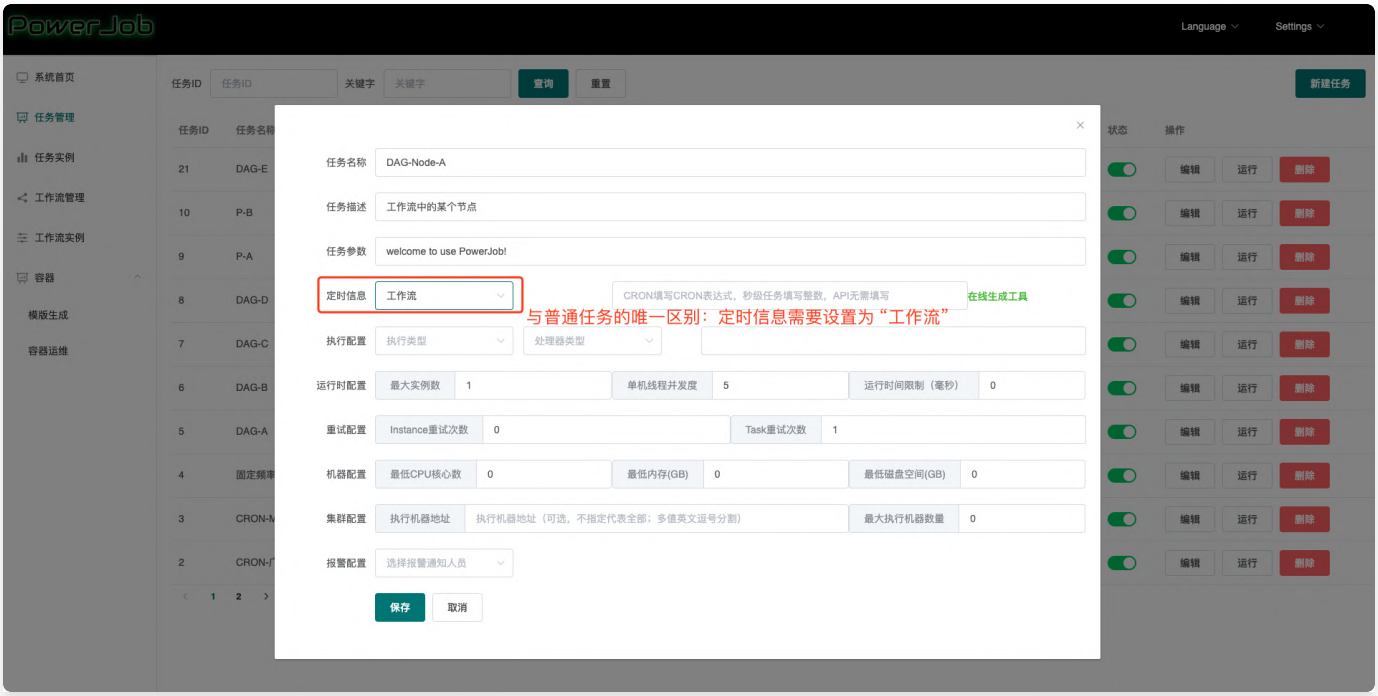
工作流描述了任务与任务之间的依赖关系，比如我现在有 A、B、C、D 四个任务，我希望 A 任务运行完毕后才开始运行 B、C 任务，最后再运行 D 任务。这就形成了一个依赖关系，可以通过有向无环图（DAG）来描述这个关系，如下图所示。



如何使用工作流?

STEP1：录入任务

配置依赖关系的前提是要有任务可以配置，所以首先需要前往任务管理界面，录入相关的任务（详细教程）。



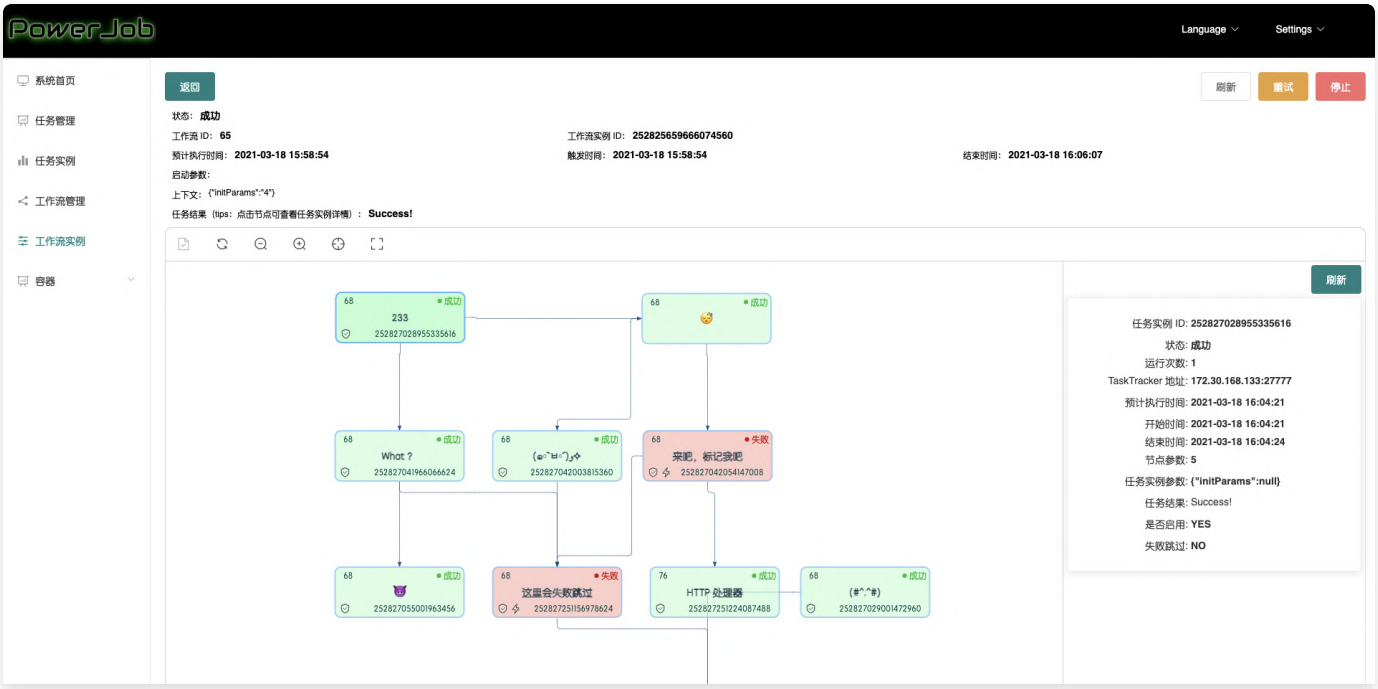
STEP2：配置工作流

前往**工作流管理**页面，点击“**新建工作流**”按钮，开始录入工作流（详细教程）。

workflow_edit_demo_v4.0.0.mp4

STEP3：查看工作流运行状态

前往**工作流实例**页面，即可查看各个工作流的运行状态，点击**详情**还能查看每个节点的具体运行信息（详细教程）。


PowerJob
Language
Settings
系统首页
任务管理
任务实例
工作流管理
工作流实例
容器
返回
刷新
重试
停止
状态：成功
工作流 ID：65
工作流实例 ID：252825659666074560
预计执行时间：2021-03-18 15:58:54
触发时间：2021-03-18 15:58:54
结束时间：2021-03-18 16:06:07
启动参数：
上下文：{"initParams":"4"}
任务结果（tips：点击节点可查看任务实例详情）：Success!
刷新
任务实例 ID: 252827028955335616
状态: 成功
运行次数: 1
TaskTracker 地址: 172.30.168.133:27777
预计执行时间: 2021-03-18 16:04:21
开始时间: 2021-03-18 16:04:21
结束时间: 2021-03-18 16:04:24
节点参数: 5
任务实例参数: {"initParams":null}
任务结果: Success!
是否启用: YES
失败跳过: NO
233
What ?
(๑•̀ㅂ•́)و✧
呆吧，标记我吧
这里会失败跳过
HTTP 处理器
(#^.^#)

官方处理器

> 对于一些通用的任务，PowerJob 官方编写了可开箱即用的 Processor 来方便各位使用！您只需要引入以下依赖即可享受所有现成的强大的官方处理器！

最新版本请自行从中央仓库获取：点击直达

```xml
<dependency>
    <groupId>tech.powerjob</groupId>
    <artifactId>powerjob-official-processors</artifactId>
    <version>${latest.version}</version>
</dependency>
```

每个官方处理器的详细使用方法请仔细阅读文档，有任何疑惑建议直接阅读源码！

由于 JSON 内传递许多参数涉及到转义，强烈建议先用 Java 代码生成配置（JSONObject#put），再调用 toJSONString 方法生成参数。

Shell 处理器

全限定类名 `tech.powerjob.official.processors.impl.script.ShellProcessor`

任务参数

填写需要处理的 Shell 脚本（直接复制文件内容）或脚本下载链接（http://xxx）

示例

Python 处理器

全限定类名 `tech.powerjob.official.processors.impl.script.PythonProcessor`

注意：Python 处理器会使用机器的 python 命令执行，因此 python 版本需要与本机 python 环境保持一致！

任务参数

填写需要处理的 Python 脚本（直接复制文件内容）或脚本下载链接（http://xxx）

示例

HTTP 处理器

全限定类名 `tech.powerjob.official.processors.impl.HttpProcessor`

任务参数（JSON）

- **method【必填字段】**：GET / POST / DELETE / PUT
- **url【必填字段】**：请求地址
- timeout【可选字段】：超时时间，单位为秒
- mediaType【可选字段】：使用非 GET 请求时，需要传递的数据类型，如 `*application/json*`
- body【可选字段】：使用非 GET 请求时的 body 内容，后端使用 String 接收，**如果为 JSON 请注意转义**
- headers【可选字段】：请求头，后端使用 Map<String, String> 接收

最简 GET 请求示例

文件清理处理器

> **注意：文件删除是高危操作，请慎用该处理器。**默认情况下该处理器不可用，需要传入 JVM 参数 `-Dpowerjob.official-processor.file-cleanup.enable=true` 开启

全限定类名 `tech.powerjob.official.processors.impl.FileCleanupProcessor`

任务参数（JSONArray）

整体参数为 array，array 中的每个元素为 JSON，描述需要清理的资源，每个节点的参数如下：

- dirPath：待删除文件的文件夹目录（会递归查找该目录下所有符合要求的文件）

- filePattern：待删除文件名称的 Java 版正则表达式
- retentionTime：待删除文件的保留时间，单位为小时（当前时间 – 待删除文件上次编辑时间 > retentionTime 的文件才会被删除），用于保留某些滚动日志，0 代表忽略该规则

由于 JSON 内传递正则表达式需要转义，强烈建议先用 Java 代码生成配置（JSONObject#put, JSONArray#add），再调用 toJSONString 方法生成参数。

示例

脚本清理配置

```
    @Test
    void testCleanWorkerScript() throws Exception {
        JSONObject params = new JSONObject();
        params.put("dirPath", "/Users/tjq/poweriob/script");
        params.put("filePattern", "(shell|python)_[0-9]*\\.(sh|py)");
        params.put("retentionTime", 24);
        JSONArray array = new JSONArray();
        array.add(params);

        TaskContext taskContext = TestUtils.genTaskContext(array.toJSONString());
        System.out.println(new FileCleanupProcessor().process(taskContext));
    }
}
```

PowerJob 脚本清理参数

日志清理配置

```
@Test
void testCleanLogs() throws Exception {
    JSONObject params = new JSONObject();
    params.put("dirPath", "/Users/tjq/logs");
    params.put("filePattern", "[\\s\\S]*log");
    params.put("retentionTime", 72);
    JSONArray array = new JSONArray();
    array.add(params);

    String paramsStr = array.toJSONString();
    System.out.println(paramsStr);

    TaskContext taskContext = TestUtils.genTaskContext(paramsStr);
    System.out.println(new FileCleanupProcessor().process(taskContext));
}
```

SQL 处理器

目前内置了两款 SQL 处理器，均支持自定义 SQL 的校验、解析逻辑，主要区别在于数据源连接的获取方式不同。

任务参数（JSON）

- dataSourceName：数据源名称，仅对 `SpringDatasourceSqlProcesssor` 生效，非必填，默认使用 `default` 数据源
- **sql**：需要执行的 SQL 语句，必填
- timeout：SQL 超时时间（秒），非必填，默认值 60
- jdbcUrl：jdbc 数据库连接，仅对 `DynamicDatasourceSqlProcessor` 生效，必填
- showResult：布尔值，是否在实例日志中展示 SQL 执行结果，非必填，默认值 false

建议生产环境使用 `AbstractSqlProcessor#registerSqlValidator` 方法至少注册一个 **SQL 校验器**拦截掉非法 SQL，比如 truncate、drop 此类危险操作，或者在数据库账号的权限上做管控。如果需要**自定义 SQL 解析逻辑**，比如 **宏变量替换，参数替换** 等，则可以通过指定 `AbstractSqlProcessor.SqlParser` 来实现。

SpringDatasourceSqlProcessor

全限定类名 `tech.powerjob.official.processors.impl.sql.SpringDatasourceSqlProcessor`

默认情况下在初始化的时候需要至少注入一个数据源，所以必须提前手动初始化并注册到 Spring IOC 容器中，以 SpringBean 的方式进行加载。

允许使用 `SpringDatasourceSqlProcessor#registerDataSource` 方法注册多个数据源

建议：最好将该 SQL Processor 用的数据库连接池和其他业务模块用的数据库连接池**隔离**开，**不要共用一个连接池！**

初始化 SpringDatasourceSqlProcessor 示例代码

```java
@Configuration
public class SqlProcessorConfiguration {


    @Bean
    @DependsOn({"initPowerJob"})
    public DataSource sqlProcessorDataSource() {
        String path = System.getProperty("user.home") + "/test/h2/" + CommonUtils.genUUID() + "/";
        String jdbcUrl = String.format("jdbc:h2:file:%spowerjob_sql_processor_db;DB_CLOSE_DELAY=-1;DATABASE_TO_UPPER=false", path);
        HikariConfig config = new HikariConfig();
        config.setDriverClassName(Driver.class.getName());
        config.setJdbcUrl(jdbcUrl);
        config.setAutoCommit(true);
        // 池中最小空闲连接数量
        config.setMinimumIdle(1);
        // 池中最大连接数量
        config.setMaximumPoolSize(10);
        return new HikariDataSource(config);
    }


    @Bean
    public SpringDatasourceSqlProcessor simpleSpringSqlProcessor(@Qualifier("sqlProcessorDataSource") DataSource dataSource) {
        SpringDatasourceSqlProcessor springDatasourceSqlProcessor = new SpringDatasourceSqlProcessor(dataSource);
        // do nothing
        springDatasourceSqlProcessor.registerSqlValidator("fakeSqlValidator", sql -> true);
        // 排除掉包含 drop 的 SQL
        springDatasourceSqlProcessor.registerSqlValidator("interceptDropValidator", sql -> sql.matches("^(?i)((?!drop).)*$"));
        // do nothing
        springDatasourceSqlProcessor.setSqlParser((sql, taskContext) -> sql);
        return springDatasourceSqlProcessor;
    }

}
```

参数配置示例

使用默认数据源，执行 SQL： `select 'x' from t_example` ，限定超时时间为 10 秒，并且在实例日志中展示结果

```JSON
{
    "dataSourceName": "default",
    "sql": "select 'x' from t_example",
    "timeout": 10,
    "showResult": true
}
```

DynamicDatasourceSqlProcessor

> 默认情况下该处理器不可用，需要传入 JVM 参数 `-Dpowerjob.official-processor.dynamic-datasource.enable=true` 开启

全限定类名 `tech.powerjob.official.processors.impl.sql.DynamicDatasourceSqlProcessor`

支持通过参数动态指定数据源连接，在指定的数据库执行 SQL。

参数配置示例

```JSON
{
    "sql": "select 'x' from t_example",
    "timeout": 10,
    "showResult": true,
    "jdbcUrl":"jdbc:mysql://myhost1:3306/db_name?user=root&password=mypass"
,
}
```

工作流上下文注入处理器

全限定类名 `tech.powerjob.official.processors.impl.context.InjectWorkflowContextProcessor`（ since v1.2.0 ）

该处理器会从任务参数中加载数据，尝试将其解析成 Map ，如果解析成功，则会将其注入到工作流上下文中。
注意，参数必须是一个 `HashMap<String, Object>` 的 JSON 串形式，否则会解析失败。

注意：该 Processor 主要用于一些需要注入固定上下文的工作流场景，作为单个任务执行是没有任何意义的

参数配置示例

JSON

```json
{
    "uid": "powerjob_001",
    "createTime": 1662210950000,
}
```

动态配置处理器

> 简易版配置中心，适用于没有 nacos 等配置中心时临时需要动态下发配置的场景。

该处理器原理非常简单，将控制台的 jobParams 写入类本身的 static 变量中，这样开发者可直接通过静态方法获取到 Job 的配置（ `ConfigProcessor#fetchConfig` ），起到伪配置中心的效果。

全限定类名：tech.powerjob.official.processors.impl.ConfigProcessor

执行模式：**广播！！！（需要保证每台机器都下发配置，必须选广播执行！！！）**

参数说明：

```java
public static class Config implements Serializable {
    /**
     * 原始配置，通过 ConfigProcessor#fetchConfig 真正能获取到的部分
     */
    private Map<String, Object> config;

    /**
     * 持久到本地的全路径名称，空代表不持久化（不需要则留空即可）
     */
    private String persistentFileName;
}
```

最简参数示例：

```json
{
    "config":{
        "keyA":"valueA",
        "keyB":"valueB",
        "tips":"config内可放置任意层级的 JSON 数据，完全透传"
    }
}
```

以下为控制台完整配置，再次强调

- **执行模式必须选择为：广播执行！！！**
- **定时信息：建议选择 1 分钟的 CRON（参数1分钟更新一次），表达式：`0 * * * * ?`**

使用代码示例（非常简单，通过 `ConfigProcessor.fetchConfig()` 获取到 config）：

Java

```java
// 测试配置中心获取数据
Map<String, Object> dynamicConfig = ConfigProcessor.fetchConfig();
Object valueA = dynamicConfig.get("keyA");
logger.info("[Test] dynamicConfig: {}, fetchByKeyA: {}", dynamicConfig, valueA);
```

多语言支持

> server 与 worker 之间通过 HTTP+JSON 进行通讯，开发者基于 Http 实现每个语言具体的 client。

PowerJob 在设计中，任务的执行是 worker 高度自治的，server 只做一开始的任务派发和后续的状态检测，因此需要实现的规范很少。
推荐先阅读官方的 powerjob-worker Java 版，无论是与 server 的交互还是 worker 内部任务的执行，思路大体上都是一致的。
没有特别约定返回值的请求，可随意返回结果字段，比如 "success"

postman: LINK

简单版：HTTP

复活：HttpProcessor

- 主要是考虑到开发一个 client 还是有一定成本的（开发 + 维护）
- 官方仅提供 Java & Python 双 client

完全体：Client

初始化阶段

初始化阶段的所有请求用的都是配置文件中配置的服务器地址（默认为 7700）端口
也只有这两个请求用的是配置文件中的服务器地址

App 校验

> 校验该 App 是否已在 server 端注册

请求地址：/server/assert?appName=
请求方式：GET
返回值：

```JSON
{
    "success":true, // true 代表校验通过，false 代表校验不通过，此时应该阻止应用继续启动
    "data":1, // 该 appName 对应的 appId，需要保存该值，后续所有请求都依赖 appId
    "message":"" // success 为 false 时存在，代表错误信息
}
```

Java 版代码路径：tech.powerjob.worker.PowerJobWorker#assertAppName

服务发现

> 定时请求 server 获取当前 app 真正的调度服务器地址，所有 worker 的请求都通过该返回地址发送。

请求地址：/server/acquire?appId=%d&currentServer=%s&protocol=HTTP

- appId 处填入之前返回的 appId
- currentServer 填当前使用的调度服务器地址，初次请求时可不填
- clientVersion：客户端版本

请求方式：GET

返回值

```JSON
{
    "success":true,
    "data":"127.0.0.1:10010", // 当前该 app 所使用的 server 地址，需要保存下来，后续所有请求都通过该地址发送给 server
    "message":""
}
```

Java 版代码路径：tech.powerjob.worker.background.ServerDiscoveryService#acquire

注意：

- 该接口极其重要，为了保证高可用性，需要**定时请求**以获取 server 的最新地址
- **该接口返回的地址才是 worker 真正需要交互的 server address**，因此该返回值必须保存下来且可被 worker 读取并使用

处理 Server 请求

任务执行请求

> 当 server 调度任务执行时，会发送该请求到 worker，worker 需要接收并解析该请求，并完成任务的运行。

worker 需要**提供**符合预期的 HTTP 接口来接收 server 的请求：

- 接口地址：/worker/runJob
- 请求方式：POST
- 请求参数
  - 官方文档更新可能不及时，建议直接参考对象 `tech.powerjob.common.request.ServerScheduleJobReq`

JSON

```
{
  "allWorkerAddress": ["192.168.1.9:27777", "192.168.1.9:27778"],
  "jobId": 202,
  "wfInstanceId": null,
  "instanceId": 241521139652755520,
  "executeType": "STANDALONE",
  "processorType": "EMBEDDED_JAVA",
  "processorInfo": "cn.edu.zju.oms.container.SimpleStandaloneProcessor",
  "instanceTimeoutMS": 0,
  "jobParams": "{\"batchSize\": 10, \"batchNum\": 10}",
  "instanceParams": null,
  "threadConcurrency": 5,
  "taskRetryNum": 1,
  "timeExpressionType": "API",
  "timeExpression": null,
  "maxInstanceNum": 0
}
```

- allWorkerAddress：Array类型，代表可参与本次任务运行的所有 worker 地址
- executeType：执行类型，*STANDALONE / BROADCAST / MAP_REDUCE / MAP*
- instanceId：任务实例ID
- instanceTimeoutMS：任务超时时间
- jobId：任务ID
- jobParams：任务的控制台参数
- maxInstanceNum：任务的最大实例数
- processorInfo：任务的处理器信息
- processorType：任务的处理器类（其他语言的客户端可以直接忽略这个属性）
- taskRetryNum：最大 Task 重试次数
- threadConcurrency：线程并发数

- timeExpression：时间表达式内容，由 timeExpressionType 决定具体的值类型，可能是数字（秒级任务）
- timeExpressionType：时间表达式类型，*API / CRON / FIXED_RATE / FIXED_DELAY / WORKFLOW*

任务实例停止执行请求

> 用户手动停止某个任务时，需要停止该任务实例

worker 需要**提供**符合预期的 HTTP 接口来接收 server 的请求：

- 接口地址：/worker/stopInstance
- 请求类型：POST
- 请求参数

JSON

```
{
  "instanceId": 239140752452485440
}
```

任务实例运行状态查询请求

> 用户查看任务实例详情时，需要从 TaskTracker 查询具体的运行时信息

worker 需要**提供**符合预期的 HTTP 接口来接收 server 的请求：

- 接口地址：/worker/queryInstanceStatus
- 请求类型：POST
- 请求参数：`tech.powerjob.common.request.ServerQueryInstanceStatusReq`

JSON

```
{
  "instanceId": 239140752452485440
}
```

- **返回值**

参考 Java ： tech.powerjob.common.model.InstanceDetail

该接口非核心功能，可选择实现

向 server 发送请求

心跳

> worker 需要定期向 server 发送心跳包，**建议每 15S 发送一次请求**

- 请求地址：/server/workerHeartbeat
- 请求方式：POST
- 请求参数：`tech.powerjob.common.request.WorkerHeartbeat`

JSON

```
{
    "appId":1,
    "appName":"powerjob-agent-test",
    "workerAddress":"192.168.124.29:27778",
    "heartbeatTime":1612792419121,
    "protocol":"HTTP",
    "systemMetrics":{
        "cpuLoad":2.4258,
        "cpuProcessors":8,
        "diskTotal":233.4691,
        "diskUsage":0.0446,
        "diskUsed":10.4179,
        "jvmMaxMemory":3.5557,
        "jvmMemoryUsage":0.1083,
        "jvmUsedMemory":0.385,
        "extra": "",
        "score":11
    }
}
```

- workerAddress：对 powerjob-server 提供 HTTP 的服务地址
- heartbeatTime：当前时间，13位时间戳
- protocol：写死 HTTP
- systemMetrics：当前系统的指标
  - cpuLoad：load1 数据
  - cpuProcessors：CPU 核心数
  - diskTotal：磁盘剩余空间大小，单位 GB
  - diskUsage：磁盘使用率
  - diskUsed：磁盘已使用空间
  - jvmMaxMemory：该进程当前最大可用内存，单位 GB（忽略 JVM）
  - jvmMemoryUsage：该进程当前的内存使用率（忽略 JVM）

- jvmUsedMemory：该进程当前所使用的内存大小，单位 GB（忽略 JVM）
- score：按当前各项参数计算得到的分数，分数越高健康度排名越高，会被优先调度

任务实例状态上报

> 每个运行中的任务都要定时发送状态信息到 server，**建议 10S 左右上报一次**。

请求地址：/server/reportInstanceStatus

请求方式：POST

请求参数：`tech.powerjob.common.request.TaskTrackerReportInstanceStatusReq`

JSON

```
{
    "jobId":14,
    "instanceId":239146443485479232,
    "reportTime":1612793022034,
    "sourceAddress":"192.168.124.29:27778",
    "instanceStatus":3,
}
```

- jobId：任务ID
- instanceId：当前任务实例ID
- reportTime：上报时间，直接取当前时间
- sourceAddress：本机地址
- instanceStatus：任务状态

```java
 */
@Getter
@AllArgsConstructor
public enum InstanceStatus {

    WAITING_DISPATCH( v: 1,  des: "等待派发"),
    WAITING_WORKER_RECEIVE( v: 2,  des: "等待Worker接收"),
    RUNNING( v: 3,  des: "运行中"),
    FAILED( v: 4,  des: "失败"),
    SUCCEED( v: 5,  des: "成功"),
    CANCELED( v: 9,  des: "取消"),
    STOPPED( v: 10,  des: "手动停止");

    private final int v;
    private final String des;
```

返回值：根据 success 判断 server 是否接收并更新成功，若任务结束请求被 server 成功接收，则代表该任务执行完毕，需要关闭任务实例处理的单元并释放相关的资源。

JSON

```json
{
  "success": true,
  "message": ""
}
```

在线日志

> 可选接口，建议实现，可在控制台直接看到日志

请求路径：/server/reportLog

请求方式：POST

请求参数：`tech.powerjob.common.request.WorkerLogReportReq`

JSON


```
{
    "instanceLogContents":[
        {
            "instanceId":239148708413833536,
            "logContent":"[DemoMRProcessor] process subTask: {"age":24,"name":"name24"}.",
            "logLevel":2,
            "logTime":1612793526037
        },
        {
            "instanceId":239148708413833536,
            "logContent":"[DemoMRProcessor] process subTask: {"age":25,"name":"name25"}.",
            "logLevel":2,
            "logTime":1612793526037
        }
    ],
    "workerAddress":"192.168.124.29:27778"
}
```


- instanceLogContents：数组，包含了若干条日志
  - instanceId：任务实例ID
  - logContent：日志的具体内容
  - logLevel：日志级别，1 ~ 4 代表 DEBUG、INFO、WARN 和 ERROR
  - logTime：日志的记录时间
- workerAddress：worker 对 server 暴露的地址

Java 参考实现：tech.powerjob.worker.log 包下所有代码

升级指南

规范：语义化版本

为了避免后期维护困难，本框架需要时刻遵守如下准则：

版本格式：主版本号.次版本号.修订号
递增规则：

1. 主版本号：当做了不兼容的 API 修改
2. 次版本号：当做了向下兼容的功能性新增
3. 修订号：当做了向下兼容的问题修正

非兼容版本升级

> 如何升级到不兼容的 PowerJob 版本？一句话描述：多版本并存。

1. 独立部署新版本的 powerjob-server，也就是版本升级阶段新、旧调度中心并存
2. 推动搭载 powerjob-worker 的执行器应用进行升级，分别升级依赖（jar 版本）和修改配置（连接到新版本的调度中心）。
3. 测试、回归、上线。
4. 完成所有升级后，下线旧版本调度中心。

V3.X.X 升级 V4.X.X

v4.x 相较 v3.x 变更很大，属于完全不兼容的两个版本，在数据层面上也存在不兼容的改动（工作流的图信息 以及 任务类型信息）。需要完成数据的转换（官方已提供相应的程序） 和 Worker 的升级（需开发自行处理，有一定工作量）才行 ~

升级方案

这里给出一种不停机升级方案，请**根据实际情况评估可行性**，做出相应的**调整**。如果目前 v3.x 版本能满足需求，不建议立即升级 ～

1. 克隆 DB 数据至新的库（如果用的库没有和业务库隔离开的话，可以用**更改表前缀**的方式，重点是将旧的数据**迁移**到一个**新的“环境”**，使其**不会影响原有业务**）

2. 升级新“环境”的表结构
3. **关闭**新库的所有定时调度任务（SQL 操作数据，更改对应任务的状态即可）
4. 起一个新的 Server 集群连接到新“环境”
5. 使用官方提供的 API 转换数据（见下方说明）
6. 挑选某个 APP 下的 Worker 进行升级（**升级 Worker 版本，重构 Processor**）
7. Worker 升级完成后起一个新的 Worker 集群连接新的 Server 集群，**充分测试验证**
8. **验证通过**后关闭该 APP 下旧集群的定时调度任务，同时关闭该 APP 下的旧 Worker 集群，再开启新“环境”的定时调度任务
9. 至此，一个 APP 下的所有任务迁移完成，再按照此步骤**逐步迁移**其他 APP 下的任务

数据转换 API

修复 Job 的处理类型信息

接口路径：/migrate/v4/job

请求参数：appId

示例请求（修复 appId = 1 的应用下的 Job 信息）

```
curl http://localhost:7700/migrate/v4/job?appId=1
```

返回数据示例

JSON


```json
{
    "success": true,
    "data": {
        "docs": "https://www.yuque.com/powerjob/guidence/official_processor",
        "scriptJobsNum": 3,
        "convertedJobIds": [1,2,3],
        "tips": "please add the maven dependency of 'powerjob-official-processors'"
    },
    "message": null
}
```


scriptJobsNum：脚本任务数量（处理器类型为 Python、Shell）

convertedJobIds：转换的任务 ID 列表

注意：如果 scriptJobsNum > 0，那么说明原来存在处理器类型为 Python 或者 Shell 的脚本任务，在 V4.0.0 版本中 Python 和 Shell 不再作为一种内嵌的处理器类型，而是以插件的形式进行支持，**需引入官方处理器的依赖！**详情见文档 官方处理器

转换工作流 DAG 信息

接口路径：/migrate/v4/workflow

请求参数：appId

示例请求（转换 appId = 1 的应用下的工作流 DAG 信息）

```
curl http://localhost:7700/migrate/v4/workflow?appId=1
```

返回数据示例

JSON


```
{
    "success": true,
    "data": {
        "failureWorkflowInfo": {
            "1": "tech.powerjob.common.exception.PowerJobException: sorry, we can't fix this workflow info automatically whose node info is wrong! you need to fix them by yourself."
        },
        "totalNum": 75,
        "fixedWorkflowIds": [1,2,3]
    },
    "message": null
}
```


failureWorkflowInfo：未能自动转换成功的数据，工作流 ID -> 失败原因的映射，此类数据需要开发者**手动**进行修复。（**一般情况下不会出现此类数据，除非开发者手工改过 DB 中的 DAG 信息**）

totalNum：工作流总数

fixedWorkflowIds：自动转换成功的工作流 ID 列表

兼容版本升级

V4.0.0 升级 V4.1.0

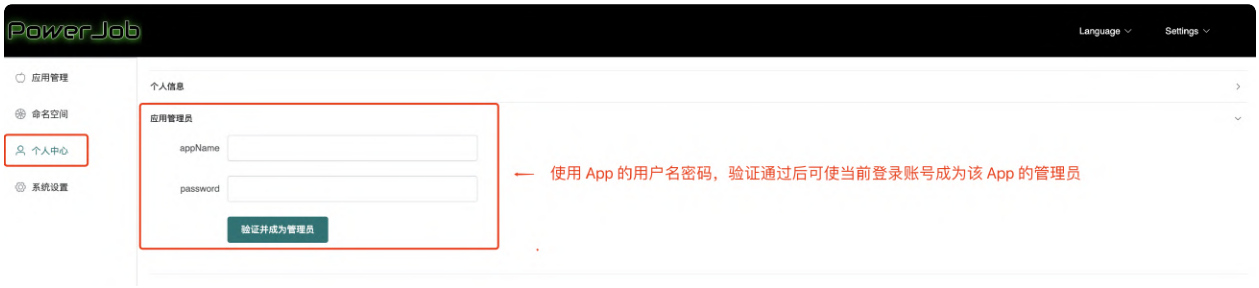
执行数据库表结构变更脚本，注意按需更改表前缀

```sql
-- ----------------------------
-- Table change for workflow_instance_info
-- ----------------------------
alter table workflow_instance_info
    add parent_wf_instance_id bigint default null null comment '上层工作流实例ID';
-- ----------------------------
-- Table change for job_info
-- ----------------------------
alter table job_info add alarm_config varchar(512) comment '告警配置' default null;
```

V4.3.x 升级 V5.0.0

> 5.x 版本的核心改动为引入了用户账号体系与权限体系，主要在 server 端，worker 部分兼容（即 powerjob-worker 5.x 与 4.x 行为一致，均可被 5.x 版本的 server 调度），因此升级成本相对较低。以下为详细升级步骤：

1. 追平数据库层面变更，方法有很多，比如
    - 【官方脚本】参考官方的数据库升级脚本，如果数据库是 MySQL8.x 应该可以直接用，否则可以参考脚本依样画葫芦。官方脚本目录（others/sql/upgrade/vx.x.x-vx.x.x.sql），本次为：LINK
    - 【自己动手版】导出当前您的 powerjob 数据库表结构，同时创建一个测试库，让 5.x 版本的 server 直连该测试库，自动建表。分别拿到 4.x 和 5.x 的表结构 SQL 后，借用工具生产 update SQL 即可（navigate 等数据库管理软件均支持结构对比）
2. 授权：5.x 版本开始，任何权限携带在用户账号纬度，废弃了原先直接拿 app 名称与密码登录的方式，当完成 5.x 版本部署后，系统中只有一个 PowerJob 自动创建的超级管理员账号，此时您可自由发挥，选择您喜欢的方式进行新用户的创建和权限的授予。
    - 方式一：手动创建相应的用户，利用超级管理员账号分别进行授权。后续登录对应的账号即可使用。
    - 方式二：手动创建相应的用户，自行使用原 App 名称和密码验证，验证通过后系统会对当前账号授予该 App 的管理员权限。



更新记录

见：GitHub Release

兼容性说明

必读

1. powerjob-worker 版本建议 低于或等于 powerjob-server ，一般可跨越2个中版本（中间的那个版本号）
2. powerjob-worker 的大版本（比如 v3.x 和 v2.x 完全不兼容）不能与 powerjob-server 不同。

迁移指南

v1.x.x -> v2.x.x

> 主要是数据库层面的变动

1. instance_info 表变动
    a. 新增字段：wf_instance_id (bigint)
    b. 新增字段：type (int)
    c. 修改字段类型：instance_params (varchar -> text)
    d. 修改字段类型：result (varchar -> text)
2. 新增 workflow_info 表
3. 新增 workflow_instance_info 表

v2.x.x -> v3.x.x

> 项目重命名，从 OhMyScheduler 改为了 PowerJob，为此需要修改相关包名：

1. com.github.kfcfans.oms.xxx -> com.github.kfcfans.powerjob.xxx

v3.x.x -> v4.x.x

> 变更较大，当然，也更强大！

1. maven 坐标变更：从 `com.github.kfcfans` 切换为 `tech.powerjob`

2. 包名变更 com.github.kfcfans.powerjob.xxx -> tech.powerjob.xxx，开发者需要感知的有
    a. OhMyWorker -> PowerJobWorker
    b. OhMyConfig -> PowerJobWorkerConfig
    c. OhMyClient -> PowerJobClient
3. 模型变更：废弃‘SHELL’和‘PYTHON’两种处理器类型，只保留“内建”和“外置”两种处理器类型。
    a. 内建：原 'Java' 类型，表示该处理器是项目中的原生代码，只要没有用到 JVM 容器，无脑选这个！
    b. 外置：原 'Java 容器'类型，表示该处理器是通过 JVM 容器热加载的。
    c. 至于 SHELL 和 PYTHON 处理器，统一由官方处理器进行承接：LINK。
4. 数据库表结构变更
    a. 新增数据库表：workflow_node_info（工作流节点信息表）
    b. 修改数据库表：job_info（任务信息表）
        i. 新增字段：dispatch_strategy（int，调度策略）
        ii. 新增字段：lifecycle（varchar，生命周期）
        iii. 新增字段：extra（varchar，用户自定义扩展参数）
    c. 修改数据库表：instance_info（任务实例信息表）
        i. 新增字段：job_params（varchar，冗余存储任务的参数）
    d. 修改数据库表：workflow_info（工作流信息表）
        i. 新增字段：lifecycle（varchar，生命周期）
        ii. 新增字段：extra（varchar，用户自定义扩展参数）
    e. 修改数据库表：workflow_instance_info（工作流实例信息表）
        i. 新增字段：wf_context（text，工作流实例上下文信息）

错误码

任务实例（Instance）错误码

too many instance

cancel dispatch job due to too many instance is running

出现原因：当前运行的任务实例数量大于任务配置的最大实例数

解决方法：前往任务管理界面修改该任务的**最大实例数**，或手动停止之前运行中的任务实例

no worker available

出现原因：没有可用的执行器，请检查worker与server的连接情况，一般主要有以下两种可能：

- 由于网络原因worker未成功连接到server
- worker成功连接到server，但由于**时间问题（包括时区、时间不准确等一系列问题）**导致server与worker存在着较大的时间差，server会丢弃该worker的连接信息

解决方法：查看日志分析原因并解决

instance execute timeout

出现原因：任务执行超时，被强制终止后失败

解决方法：优化代码缩短运行时间 或 增大超时时间

worker report timeout, maybe TaskTracker down

出现原因：任务执行集群的主节点（TaskTracker）宕机，任务直接失败

解决方法：无，一般由机器重启部署等原因导致

create root task failed

出现原因：创建根任务失败

解决方法：理论上不存在该错误，如果出现该错误，请提取相关错误的日志堆栈联系开发者@KFCFans解决

unknown bug

出现原因：不详

解决方法：同上，联系开发者解决。

工作流实例（WorkflowInstance）错误码

middle job failed

出现原因：整个工作流中的某个任务执行失败，工作流被打断，整体失败

解决方法：查看失败任务的详细信息，分析失败原因并解决

middle job stopped by user

出现原因：整个工作流中的某个任务被用户手动停止，工作流被打断，整体失败

can't find some job

出现原因：工作流所需要的任务不存在或被禁用

解决方法：前往任务管理页面检查该工作流所需要的任务是否存在且启用

在线试用

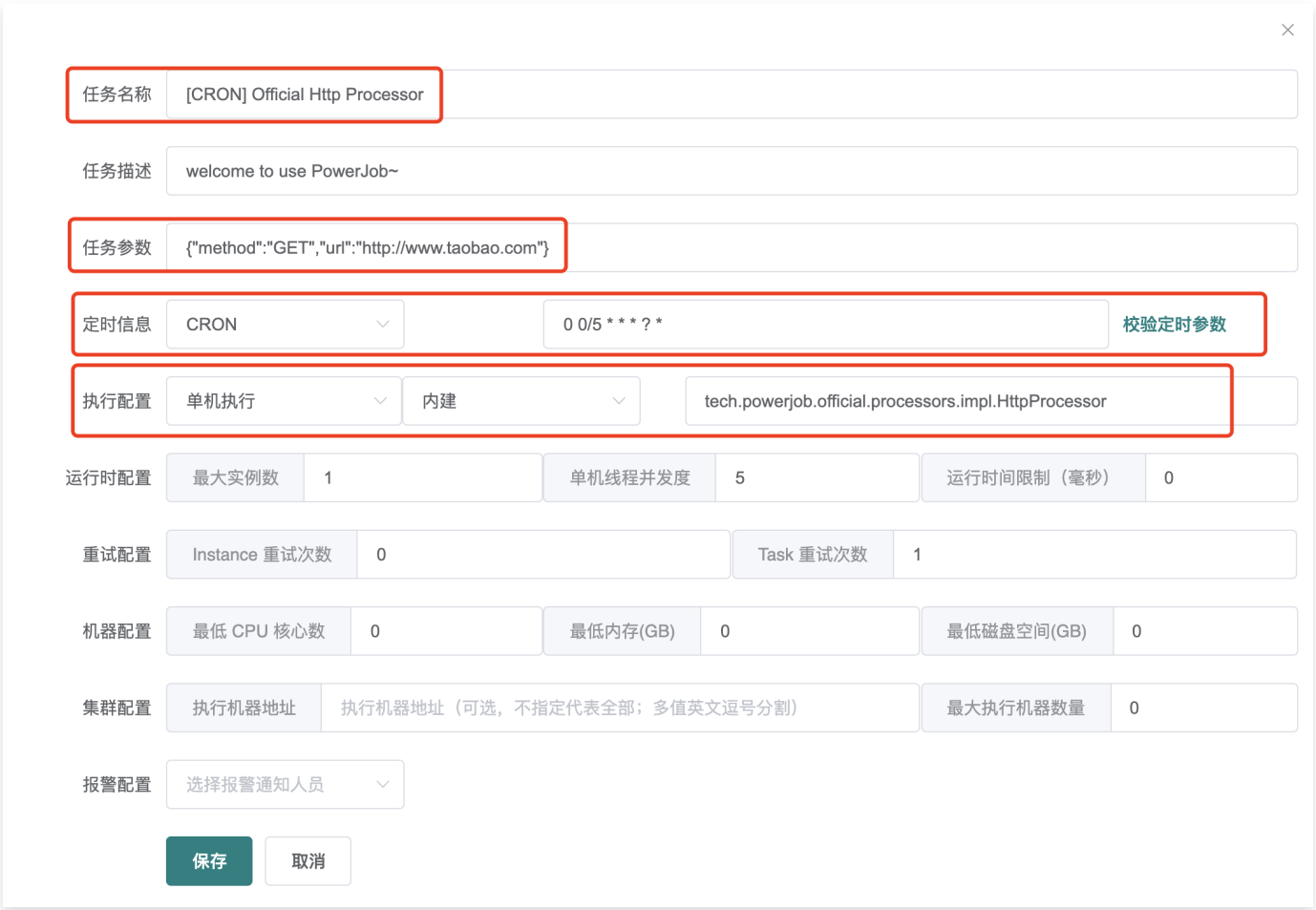
试用地址1：http://try.powerjob.tech/#/welcome?appName=powerjob-worker-samples&password=powerjob123
应用名称：powerjob-worker-samples
密码：powerjob123

说明：试用的 PowerJob 系统由 powerjob-server 和 powerjob-agent 构成，powerjob-agent 中携带了所有的官方处理器。同时，为了展示 PowerJob 强大的分布式计算、广播处理能力，agent 也动态加载了一个简单的 JVM容器，源码地址：LINK

简单试用

四处红框是 PowerJob 任务的最小配置，其中：

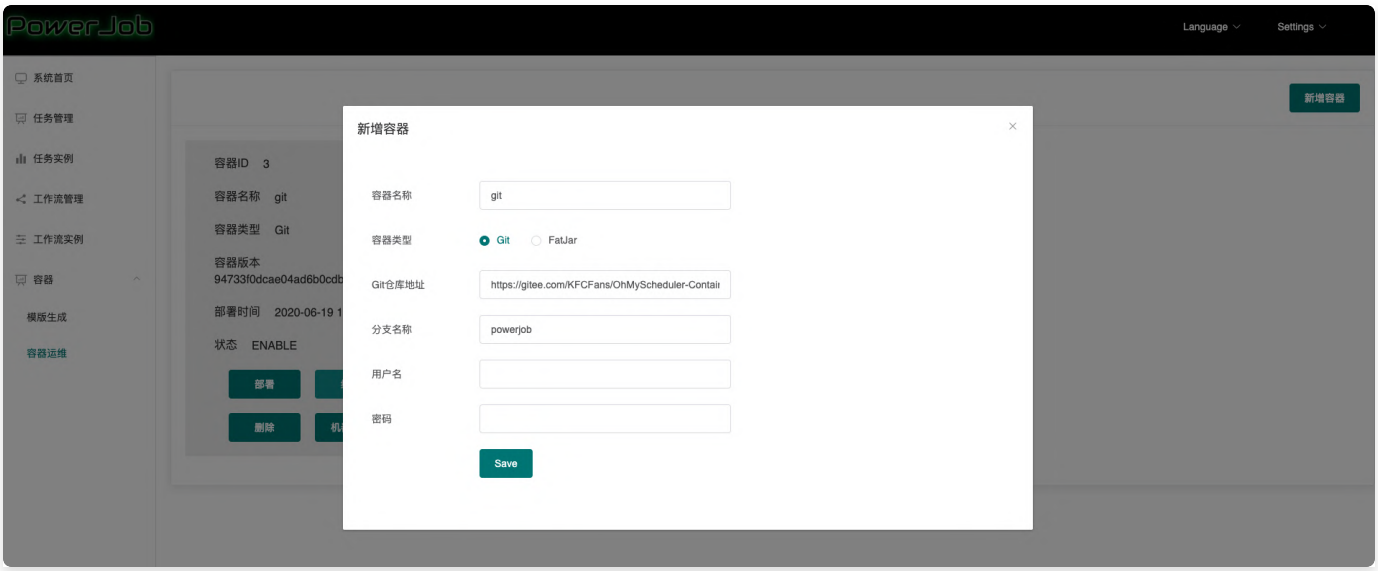
- 任务参数：官方处理器要求有特定的参数，可先仔细阅读官方处理器章节填入
- 定时信息：支持 API、CRON、固定频率、固定延迟 4种调度策略
- 执行配置
  - 试用官方处理器请选择“内建”，并填入处理器的全限定类名
  - 试用 JVM容器 请选择“外置”，并填入 容器ID#处理器全限定类名，详见下一小节



Java处理器的试用

容器技术

在试用容器之前，建议阅读开发文档：容器



您可以将自己的容器上传到 GitHub/Gitee，也可以选择官方编写的试用 Git 容器。

Git仓库地址：https://gitee.com/KFCFans/PowerJob-Container-Template/tree/master/
分支：master
用户名：空（不需要填写）
密码：空（不需要填写）

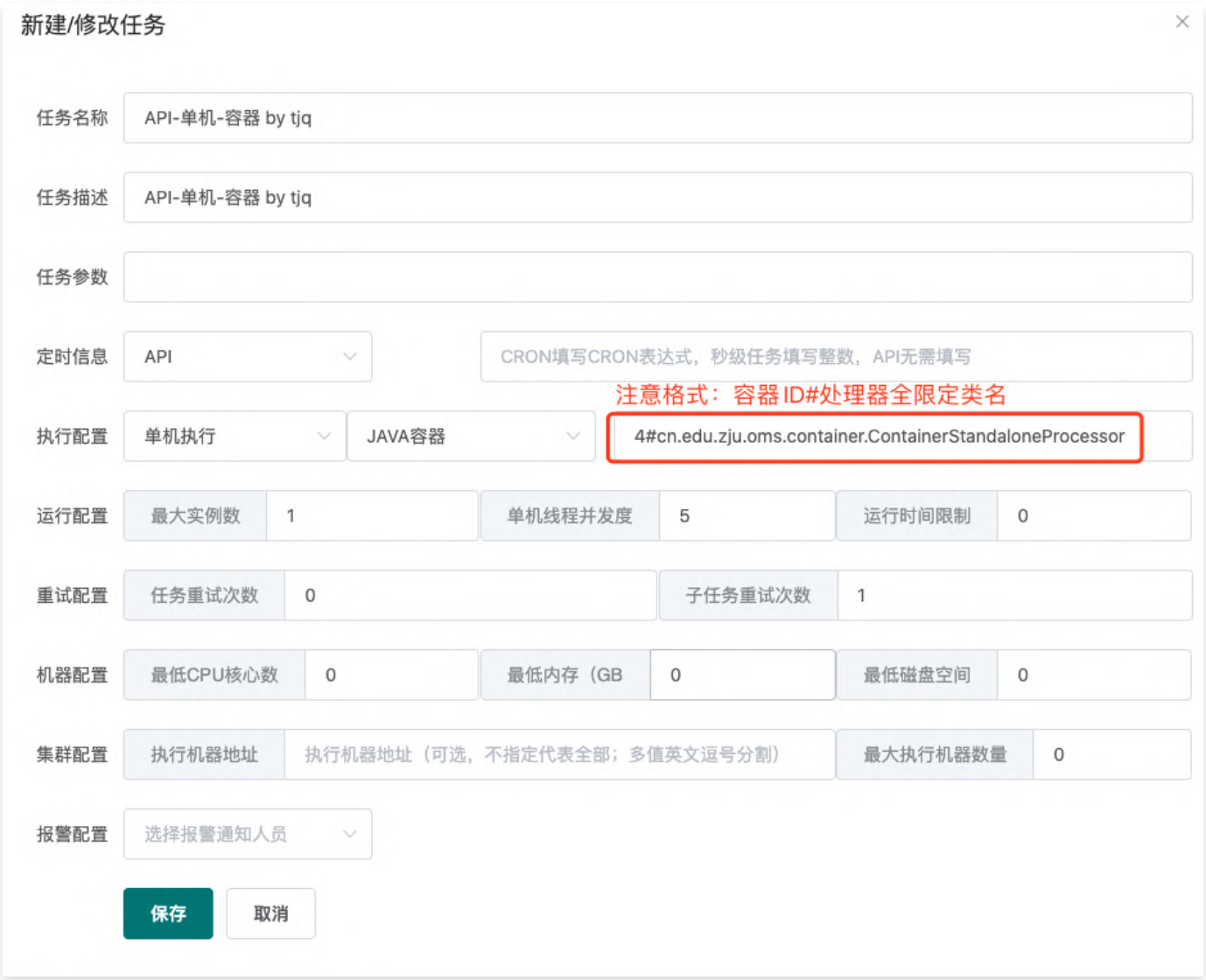
该容器包括以下三个处理器：
单机处理器：cn.edu.zju.oms.container.SimpleStandaloneProcessor
广播处理器：cn.edu.zju.oms.container.ContainerBroadcastProcessor
MR处理器：cn.edu.zju.oms.container.ContainerMRProcessor

- MR处理器需要填入自定义的任务参数：{"batchSize": 100, "batchNum": 10}

当然，也可以上传自己的 FatJar 并完成部署，详细教程请参考：容器
注：生成模版工程后需要更改 pom.xml 中的 **oms.worker.version**，**最新当前版本为 4.2.1。**

创建任务

任务录入示例如下，注意：处理器信息的填写格式为 容器ID#全限定类名



工作流试用

需要先录入任务，任务录入完毕后，请参考 官方教程 进行工作流的创建与使用。

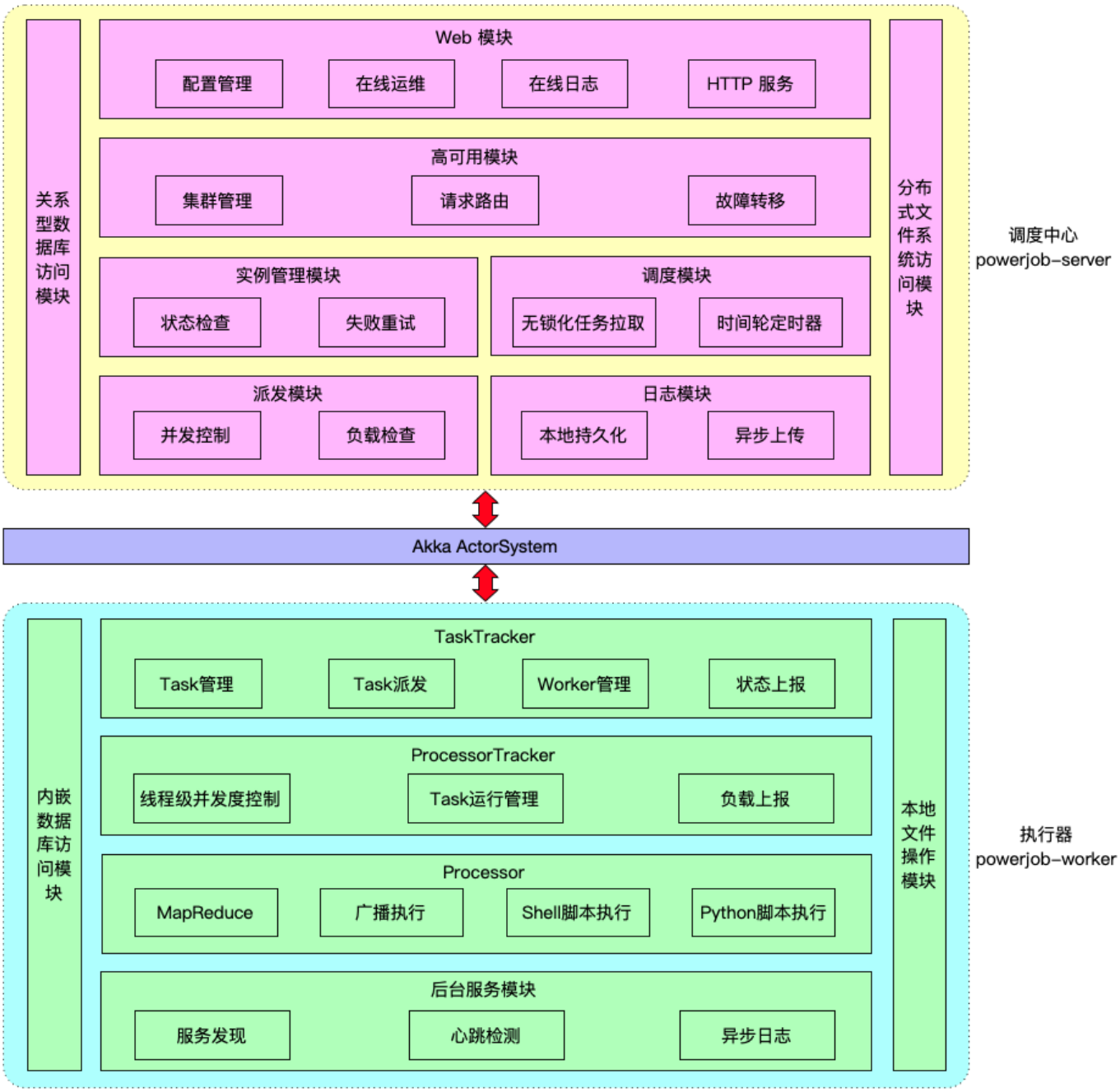
workflow_edit_demo_v4.0.0.mp4

架构与原理

原理剖析系列文章

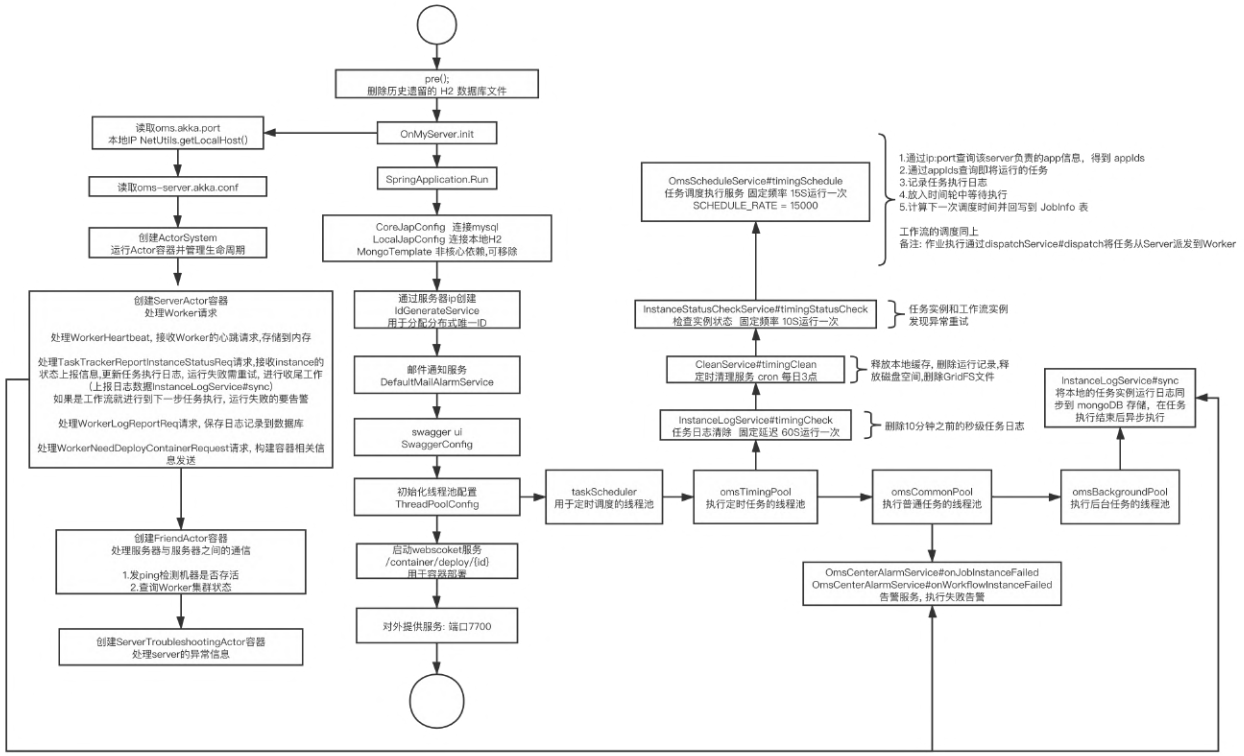
点击查看

架构图（模块视图）



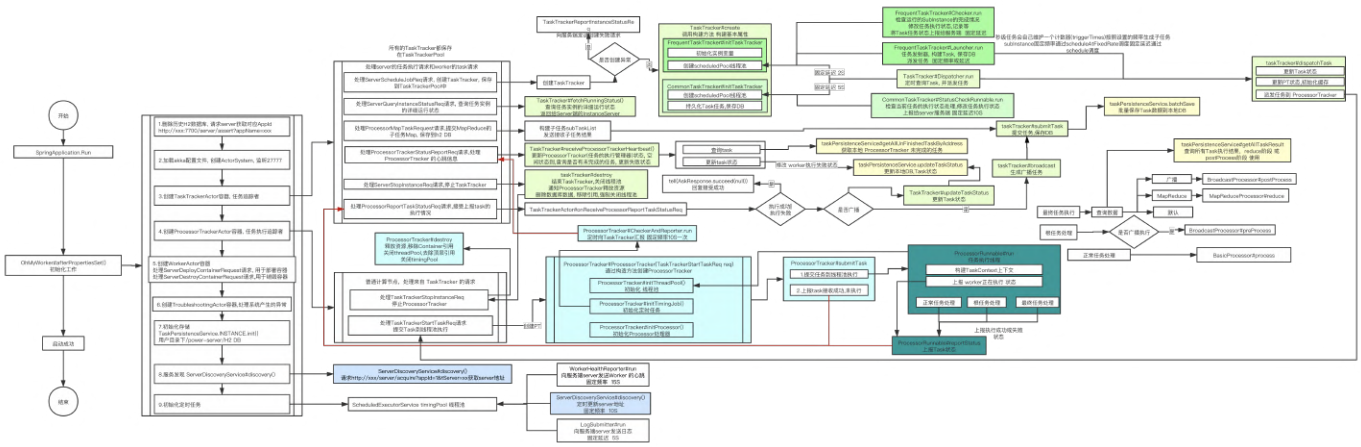
Server 启动流程分析

> server 功能全景图，感谢 **群友@大海** 的贡献

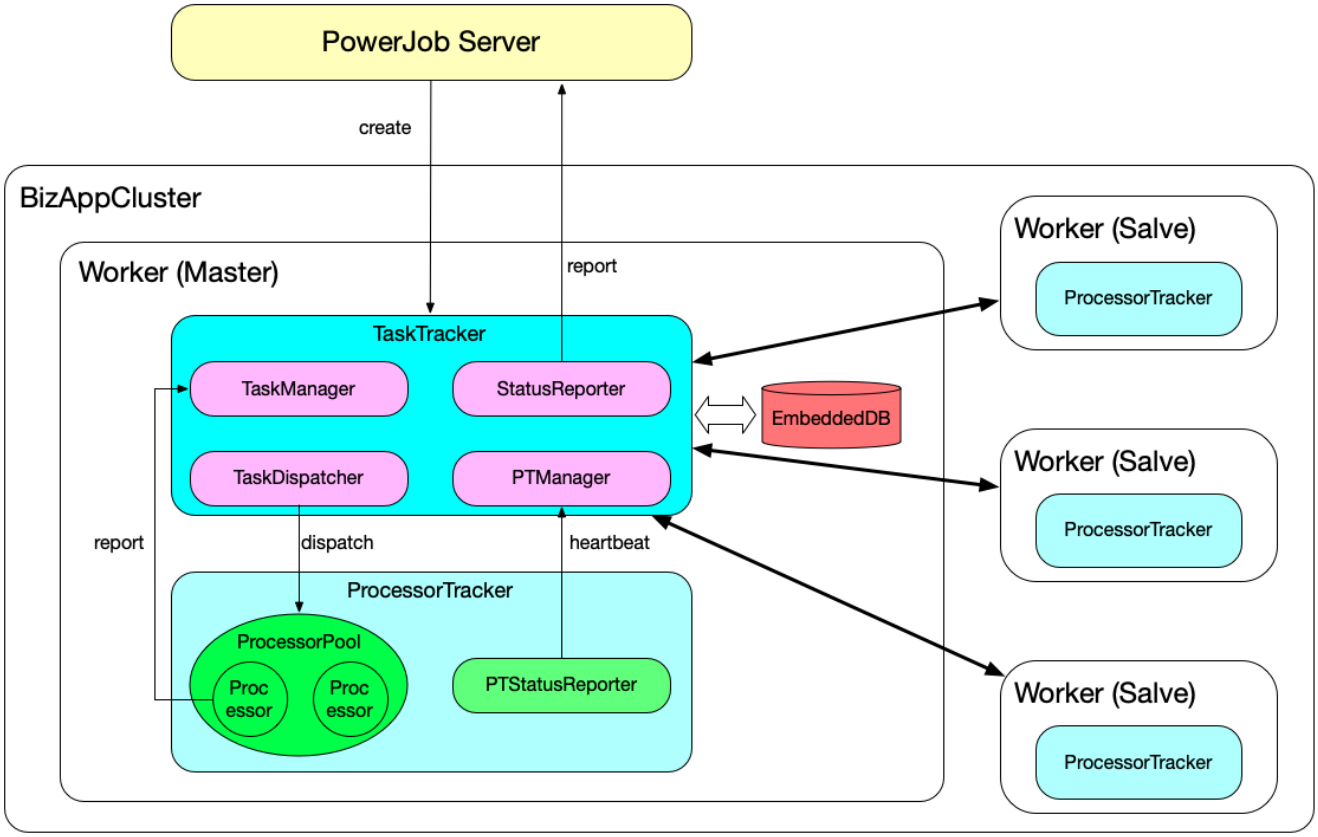


Worker 启动流程分析

> worker 功能全景图，感谢 **群友@大海** 的贡献



MapReduce 原理


PowerJob Server
create
report
BizAppCluster
Worker (Master)
TaskTracker
TaskManager
StatusReporter
TaskDispatcher
PTManager
EmbeddedDB
report
dispatch
heartbeat
ProcessorTracker
ProcessorPool
Proc essor
Proc essor
PTStatusReporter
Worker (Salve)
ProcessorTracker
Worker (Salve)
ProcessorTracker
Worker (Salve)
ProcessorTracker

#

开发计划

截止目前（v4.1.0），经过多个版本的迭代，框架的基本功能（调度和分布式计算）已经非常稳定，大家可以放心接入使用。不过为了使框架更为成熟易用，仍有需要不断改进和开发的地方。

下阶段规划

详见 GitHub看板

共同建设邀请（❁´◡`❁)*✲ﾟ

> **前路昭然，你我共进～**

都看到这里了，想必您已经对本项目产生了**浓厚**的兴趣，欢迎加入PowerJob开源社区，为框架贡献自己的力量～

PowerJob_Issues：https://github.com/PowerJob/PowerJob/issues

PowerJob 用户群：487453839

（PR、Issue求求了～）

如果有什么想法、建议或意见，欢迎联系作者：tengjiqi@gmail.com

演讲稿

COSCon'20 & Apache Roadshow – China

FAQ

Q：PowerJob 与 DolphinScheduler 比有什么不同吗？或者优势?

A: 立足点和面向的领域不同。
像 airflow 、dolphinscheduler 这类框架主要面向大数据处理领域，其核心需求是按照规定流程（DAG ）跑一堆脚本去完成一些数据任务。因此这类框架提供的处理器一般都很简单（比如 Airflow 其实就是用 bash 去执行 shell 、python 脚本或者用 HTTP 去触发），其主要功能或者亮点在于如何生成并发布数据处理方法（像 dolphinscheduler 内置了 MySQL 、Spark 、Hive 、Hadoop 等等各种数据源客户端，用户可以直接在线写 SQL 、HQL 等完成数据操作）。
而 PowerJob 最初是面向业务应用设计的，核心需求是解决那些需要通过代码做一些复杂的数据处理的任务的调度，同时在计算量很大单机难以支撑的情况下提供分布式计算能力。也就是说 PowerJob 面向的是需要你自己写代码干活的场景。
简单概括，airflow 系的主战场是跑脚本跑 SQL 类任务，而 PowerJob 的主战场是业务相关的需要写代码完成的计算任务，当然，也顺手支持脚本的运行。

Q：请问，有没有通过 API 增删改查执行的操作?

A：OpenAPI 就是为此而生的。OpenAPI 在 HTTP 的基础上进行了封装，提供规范的接口完成任务的管理与运维。

Q：Ignite 也支持分布式计算，请问这个项目有什么优势?

A：从本质上讲，PowerJob是一个具有分布式计算能力的任务调度框架，而 Ignite 是一个分布式计算框架，前者立足于任务调度（虽然本项目的亮点是分布式计算没错啦...），后者立足于大数据计算，两者立足点不同。
从分布式计算的角度来讲，Ignite 确实具备全部 PowerJob 的功能（毕竟人家是 Apache 顶级项目...），PowerJob-Worker 集群可以看成嵌入式的 Ignite 集群，整体对外提供服务。两者虽然表面上功能有所重合，但背后的设计理念是截然不同的。
Ignite 本质上是由分布式内存 SQL 数据库发展而来的分布式计算平台，它解决的问题更偏向于大数据处理（Spark、Hadoop 之类），因此对于传统的 Java 项目并不是非常友好，比如官方推荐的部署模式是建立独立的Ignite 集群负责计算，业务应用只负责提交代码。再比如获取各种资源（Spring Bean）都需要先注入 Ignite 中，这对于依赖繁杂的业务来说是非常痛苦的。
而 PowerJob 就是面向业务应用设计的，从示例代码中也能看出，开发 PowerJob 的处理器是没有任何额外的成本的，想要某个 SpringBean，直接注入即可。想要分发任务，调用 map 方法即可，开发者

的学习和使用成本会低很多。
一句话总结就是：Ignite 的分布式计算偏向于数据侧，适用于大数据处理。而 PowerJob 的分布式计算偏向于业务侧，适用于传统 Java 应用的业务处理。

此外，高效开发运维一直是 PowerJob 的设计理念，像在线查看任务运行情况、在线查看任务运行日志等功能，在实际开发中将会非常实用。

常见问题

当 powerjob-server 或 powerjob-worker 无法启动时，请您先尝试自主排查问题，可靠的途径有：

1. 在本文档中查找是否有和您遇到的问题类似的 case
2. 去 GitHub Issue 搜索相关的问题
3. 使用搜索引擎（Google、百度）搜索相关的问题
4. 问问 ChatGPT

如果您无法解决遇到的问题，需要相关的帮助，请前往 GitHub 新建问题（new Issuce -> Question），我们将稍后为您解答。

1. 无法启动

1.1 编译报错

编译报错：不兼容的类型：推断类型不符合上限

-> 请升级最新版本的 OpenJDK 8 或 11

1.2 Worker 无法启动

NoClassDefFoundError

典型的依赖冲突问题，建议使用 `mvn dependency:tree` 打印依赖树进行问题的排查。

ClassNotFoundException

解决方案同上

1.3 Server 无法启动

powerjob-server 就是一个普通的 SpringBoot 应用程序，**无法启动的原因一般是数据库配置问题**、JDK兼容性问题（JDK 8 & 11 测试通过，其余未测试）和操作系统兼容性问题（Linux & macOS & Windows 10 测试通过，Windows 7 可能无法启动），**建议自行根据异常堆栈信息进行排查**。

powerjob-server 日志路径：

Docker 启动：在 docker run 命令中通过 -v 指定，默认为 ~/docker/powerjob-server/logs

Jar 启动：~/powerjob-server/logs

serverTimezone 问题

```
java.sql.SQLException: The server time zone value '�й��� ʰ��' is unrecognized or represents more than one time zone. You must configure either the server or JDBC driver (via the 'serverTimezone' configuration property) to use a more specifc time zone value if you want to utilize time zone support.
```

解决方案：**在 JDBC URL 后加上 serverTimezone 信息，如**

jdbc:mysql://127.0.0.1:3306/powerjob-daily?useUnicode=true&characterEncoding=UTF-8&serverTimezone=Asia/Shanghai

高版本JDK问题

> PowerJob 兼容 JDK8～17 的版本，高版本 JDK 由于移除了某些包，需要自己手动引入

类似于：java.lang.NoClassDefFoundError: javax/xml/bind/JAXBException

高版本JDK移除内置模块导致，参考以下教程添加相关包依赖即可。

https://stackoverflow.com/questions/52502189/java-11-package-javax-xml-bind-does-not-exist

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

3. Server/Worker 网络问题（执行器无法连接到 server）

PowerJob 中的所有组件（server 和 worker）必须处于 **同一个局域网** 中，这是系统运行所需要的必须条件。

2. 无法连接（控制台无worker列表）

- 请检查 worker 是否启动成功
- 请检查 worker 和 server 是否存在时差，为了保证调度准确性，**server 和 worker 最大时差为 60S**
- 请检查 worker 与 server 是否在一个局域网中，如果不在（公网 / NAT / 容器 等场景），则需要特殊处理。

2.1 非局域网通讯

可使用 PowerJob 提供的启动参数绑定外部实际通讯 IP（该参数 worker 和 server 共享，均可配置，按需配置）：

- `powerjob.network.external.address`：外部IP地址，比如公网IP、NAT入口地址
- `powerjob.network.external.port`：外部地址的端口
  - worker 侧由于只涉及一个接口，直接使用该参数指定外部地址端口即可
  - server 侧由于涉及多协议，需要单独为每个协议指定映射端口，规则为 `powerjob.network.external.port.${协议}`，比如 HTTP 协议为 `powerjob.network.external.port.http`。

举例，现有

- server：内网地址为 server-innerIp，公网地址为 server-externalIp
- worker：内网地址为 worker-innerIp，公网地址为 worker-externalIp。
- server 与 worker 基于 http 协议通讯，server 的 http 通讯协议端口为 10010，worker 的工作端口为 27777

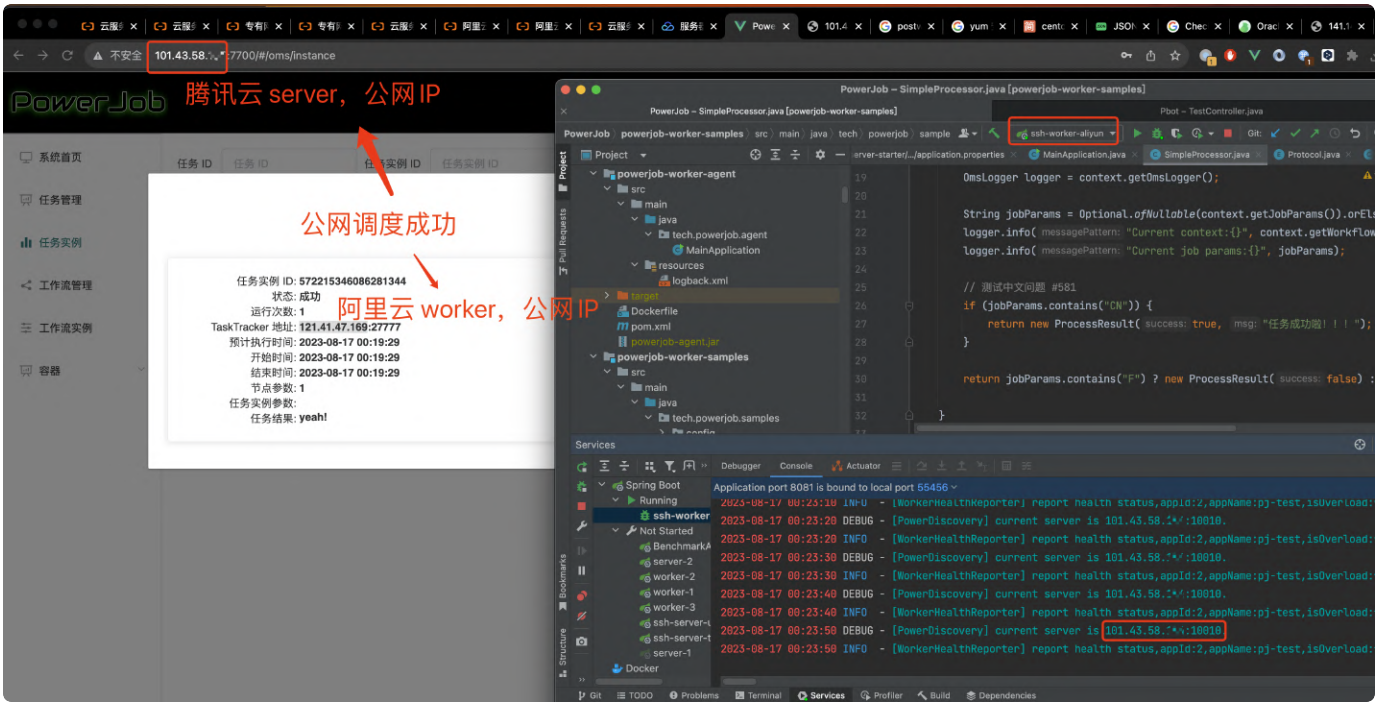
基于上述场景：

- server 需要的启动参数：
  - 绑定 server 的外部地址 `-Dpowerjob.network.external.address=server-externalIp`
  - 绑定 server **某个通讯协议**的外部地址端口：`-Dpowerjob.network.external.port.http=10010`
- worker 需要的启动参数：
  - 绑定 worker 的外部地址 `-Dpowerjob.network.external.address=worker-externalIp`

- 绑定 worker 的外部地址端口（worker 只有一个端口，因此不需要协议） `-Dpowerjob.network.external.port=27777`

注意点：

1. powerjob.network.external.address 绑定的是本机的外部地址，而不是请求对象的地址（比如 server 部署在 121.41.47.169 这台机器上，内部绑定 127.0.0.1，external 则绑定 121.41.47.169）
2. **external 的作用是向另一端下发真正可访问的IP地址和端口，这个功能并不是万能的。要实现通讯，依旧需要保证 server 和 worker 最终的 external address 可互通**
3. 注意绑定 external address 后反而破坏局域网条件的场景。比如 docker 内的 server 和 worker，本身互通，但一旦 server 绑定 external address 为宿主地址，那 docker 内的 worker 就无法访问这个地址了，破坏了局域网条件。
4. HTTP 协议确认可用，akka 协议未测试。
5. 当前 PowerJob 并没有做过多的安全设计，支持非局域网调度主要是为了解决 docker + 本地 ide 等场景。强烈不建议在 PowerJob 支持用户和权限系统前使用公网调度。**（敬请期待权限版本）**
6. 最后放一张图证明功能可用性，不要怀疑！可用！



2.2 绑定网卡/获取 IP 不对

如果发现绑定的网卡/获取的 IP 不对，可以在worker启动时，添加 JVM 启动参数 `-Dpowerjob.network.interface.preferred=xxx` 来指定绑定的网卡（该参数需要填入需要绑定的网卡名称），

或通过 `-Dpowerjob.network.interface.ignored=xxx` 来忽略错误的网卡（该参数支持正则表达式，匹配到的网卡会被忽略）。
终极大招：可通过 JVM 启动参数 `-Dpowerjob.network.local.address` 来**直接绑定本地使用的某个IP**

3. 常见运行时问题

3.1. 负载均衡问题

- worker 连接 server 默认用 IP 列表的第一个值，多应用接入情况下更换 IP 顺序即可实现各个 server 都提供服务。当然，最推荐的还是用域名的形式做负载均衡。
- server 调度 worker 执行任务默认采用的是**最优选择策略**（可选择为随机），单机任务只会选取 CPU/内存状态最佳的机器执行！
- 当某个节点故障时，PowerJob 有强大的高可用机制进行故障转移，只要系统中有组件存活，调度和任务就能正常进行！

3.2. 精确时间调度/只调度N次问题

CRON表达式支持精确时间调度
比如 0 0 13 8 10 ? 2020-2020 代表 2020年10月8日13:00:00秒

如果确定 CRON 无法支持，请自行通过 OpenAPI 实现。

4. 多数据库支持

> 牢记一点，`powerjob-server` 只是一个用了 SpringDataJPA 的普通应用。遇到数据库相关的错误，网上以 SpringDataJPA + 相关堆栈关键字，大概率有解决方案。

4.1. PostgreSQL

- Caused by: org.postgresql.util.PSQLException: 大型对象无法被使用在自动确认事物交易模式
- JpaSystemException: Unable to access lob stream

如有类似错误可参考以下 ISSUE
https://github.com/PowerJob/PowerJob/issues/153#issuecomment-812771783

1. 如果 JPA 已经自动建表，请先删除 powerjob 相关的全部表
2. 在配置文件中添加 `spring.datasource.remote.hibernate.properties.hibernate.dialect=tech.powerjob.server.persistence.config.dialect.PowerJobPGDialect` 来启用 postgreSQL 专用方言处理器，重新启动 server & 自动建表。

4.2. Oracle

表名称不合法问题（oracle 数据库账号名称如（powerjob-product）不要出现字符 -，请使用 powerjob_product）：

https://github.com/PowerJob/PowerJob/issues/765

表名称不合法问题 Plain Text

```
org.springframework.beans.factory.BeanCreationException: Error creating bean with name 'remoteEntityManagerFactory' defined in class path resource [tech/powerjob/server/persistence/config/RemoteJpaConfig.class]: Invocation of init method failed; nested exception is javax.persistence.PersistenceException: [PersistenceUnit: remotePersistenceUnit] Unable to build Hibernate SessionFactory; nested exception is org.hibernate.exception.GenericJDBCException: Error accessing index information: POWERJOB-PRODUCT.APP_INFO
at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.initializeBean(AbstractAutowireCapableBeanFactory.java:1804)
at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.doCreateBean(AbstractAutowireCapableBeanFactory.java:620)
at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.createBean(AbstractAutowireCapableBeanFactory.java:542)
at org.springframework.beans.factory.support.AbstractBeanFactory.lambda$doGetBean$0(AbstractBeanFactory.java:335)
at org.springframework.beans.factory.support.DefaultSingletonBeanRegistry.getSingleton(DefaultSingletonBeanRegistry.java:234)
```

- Caused by: java.sql.SQLSyntaxErrorException: ORA-02289: 序列不存在

Oracle 数据库需要自己创建序列去处理主键自增，网上搜一搜，教程非常多。

SQL

```sql
create sequence native
minvalue 1
maxvalue 999999999999999999999999999
start with 1
increment by 1
cache 20;
```

其他

离线文档

📎PowerJob 产品手册_v4.3.6.pdf

鸣谢

感谢所有项目的贡献者！！！

https://github.com/PowerJob/PowerJob/graphs/contributors

赞助者

- @群友 BelAir
- @群友 yzpniceboy
- 苏州镁氪智能设备有限公司
- 深圳富龙小额贷款有限公司

项目贡献者

- 某著名上市电商公司前端工程师
- hxuanyu
- Linfly
- dudiao
- 宁远
- ocean23
- Echo from CVTE
- Max from CVTE

Alibaba SchedulerX2.0

本框架的设计思想来源。

- Akka 框架：不得不说，akka-remote 简化了相当大一部分的网络通讯代码。
- 执行器架构设计：这篇文章反而不太认同，感觉我个人的设计更符合 Yarn 的“架构”。
- MapReduce模型：想法很 Cool，大数据处理框架都是处理器向数据移动，但对于传统 Java 应用来说，数据向处理器移动也未尝不可，这样还能使框架的实现变得简单很多。
- 广播执行：运行清理日志脚本什么的，也太实用了 8 ～