

|  |  |
| --- | --- |
| 密级： | 公开 |
| 版本： | V1.0 |

高并发网状调度系统白皮书

2019年10月24日

**修订记录**

|  |  |  |  |
| --- | --- | --- | --- |
| 版 本 | 修订说明 | 编制人 | 编制日期 |
| V0.1 | 建立文档 | 龙大帅 | 2019/10/22 |
| V1.0 | 定稿输出 | 龙大帅 | 2019/10/24 |

目 录

[一、 背景 2](#_Toc22829643)

[二、 定位与价值 3](#_Toc22829644)

[2.1 分布式作业调度平台 4](#_Toc22829645)

[2.2 灵活易用的调度平台 4](#_Toc22829646)

[2.3 实时全面的监控平台 4](#_Toc22829647)

[2.4 企业运维自动化 5](#_Toc22829648)

[三、 核心模块 5](#_Toc22829649)

[3.1 基础配置 6](#_Toc22829650)

[3.1.1 调度服务器配置 6](#_Toc22829651)

[3.1.2 作业参数配置 7](#_Toc22829652)

[3.1.3 预警配置 8](#_Toc22829653)

[3.2 调度管理 8](#_Toc22829654)

[3.2.1 调度作业 9](#_Toc22829655)

[3.2.2 作业流 10](#_Toc22829656)

[3.3 作业监控 12](#_Toc22829657)

[3.3.1 作业执行详情 13](#_Toc22829658)

[四、 产品优势 15](#_Toc22829659)

[4.1 分布式架构 15](#_Toc22829660)

[4.2 高可用性 15](#_Toc22829661)

[4.3 高容错性 15](#_Toc22829662)

[4.4 支持任意类型的作业 16](#_Toc22829663)

[4.5 交互友好性 16](#_Toc22829664)

[4.6 作业之间的灵活依赖 16](#_Toc22829665)

[4.7 强大的作业调度方式 16](#_Toc22829666)

[4.8 灵活多样的告警规则 16](#_Toc22829667)

[4.9 完整的日志记录 17](#_Toc22829668)

[4.10 丰富的应用场景 17](#_Toc22829669)

# 背景

大数据时代随着企业业务的发展与信息化建设的成熟，企业内部积累了海量的业务数据。在大数据平台，每天承载着成千上万的ETL任务调度，随着ETL同步数据任务数量日益增加，迫切需要一个更加强大的调度系统来管理平台中的定时任务。另外，支付系统的定期跑批清算、电子商城优惠活动的自动上下架、营销活动的通知下发、订单支付管理等等场景都需要一个强大的调度系统来支撑日益复杂的业务需求。

在传统的调度系统中，作业运行时不能暂停、不能支持参数传递、不能补数、调度系统不高可用、配置作业流不能可视化、任务的运行状态不能实时在线查看、各个作业之间的依赖仅靠简单的串行来实现等等问题都在困扰着我们。同时大数据平台的开发语言多种多样，开发人员背景各异，很多不同类型的程序（任务）运行在大数据平台上，也为调度任务的管理带来了极大的挑战。

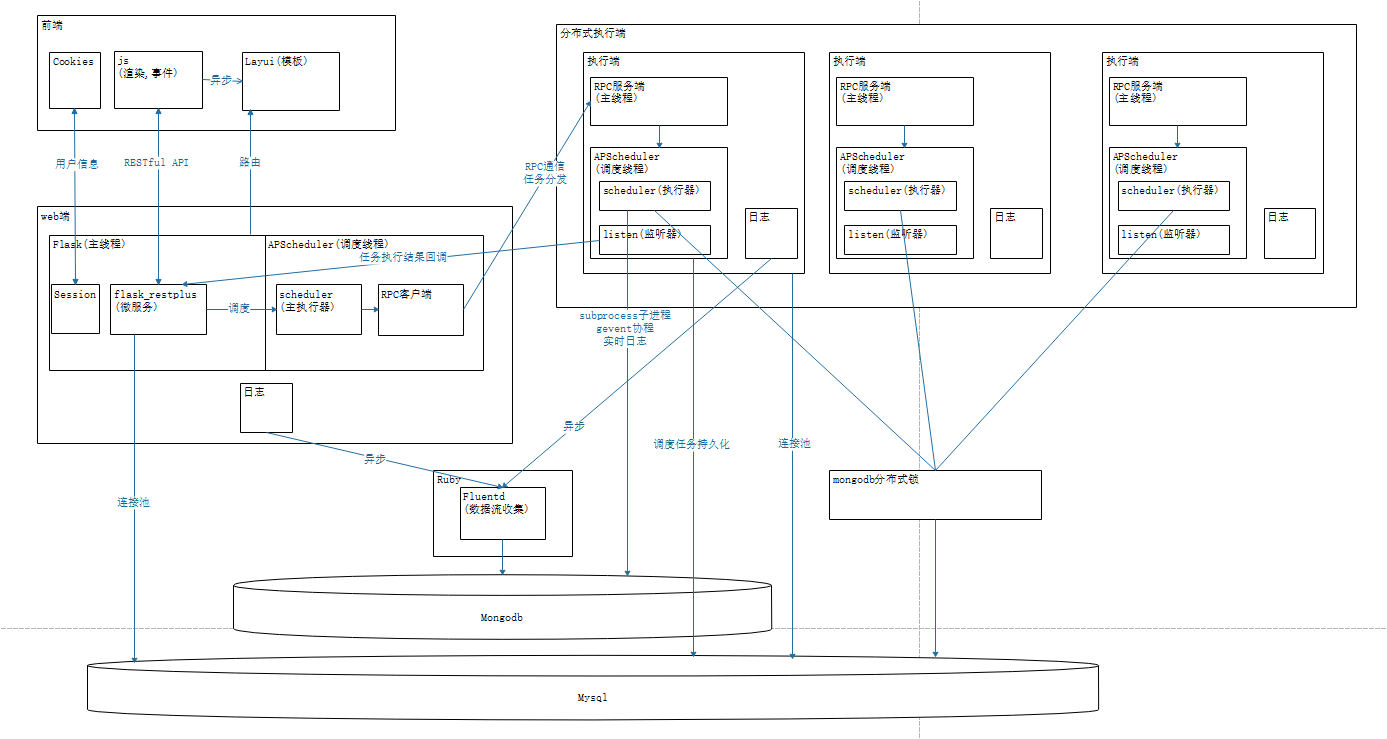
如何保证大量的ETL作业准确的完成调度而不出现问题？如何在调度执行中出现错误的情况下，调度能够完成自我恢复甚至执行错误告警与完整的日志查询？

高并发网状调度系统就是在这种背景下衍生的一款分布式海量作业调度和监控平台。本项目使用网状模型重构ETL架构，解决调度维护和监控工作逐渐繁重，机器资源消耗过多的问题。同时提供完善易用的web用户界面，可以快速的可视化实现作业之间复杂的依赖关系配置、灵活配置多种调度方式和调度的维护管理。另外系统会收集并记录作业运行过程中产生的日志，用户可以方便的访问作业运行日志，查看作业状态和运行详情，对调度的执行过程进行监控，并提供实时预警服务。

调度系统是大数据平台的基础，一个简单易用、性能优异的调度系统不仅是支撑业务稳定运行的基石，更是业务不断发展壮大的不可或缺的基础服务平台。

# 定位与价值

高并发网状调度中心主要用于集中处理企业大数据平台中大量的ETL作业，按业务逻辑梳理作业，自动高效地完成作业调度。系统自动化监控各个调度的运行情况，作业执行出现异常时自动发送预警通知，帮助运维人员快速反馈和定位作业执行错误。



## 分布式作业调度平台

调度引擎采用可配置的代理负载加权算法，可以在执行作业时支持多机集群，根据负载情况合理分担作业执行的服务器，从而承担大规模数据的高负荷运行。

## 灵活易用的调度平台

高并发网状调度系统提供简单易用的web交互界面，同时支持按照业务需求灵活配置多种作业调度方式，提供日历、频度、作业依赖、手动触发等按需的作业调度，支持作业复杂依赖关系的配置、展示、运行监控。

利用高并发网状调度系统灵活的配置功能和完善的告警处理机制，大大降低了因为故障处理不及时而带来的损失。

## 实时全面的监控平台

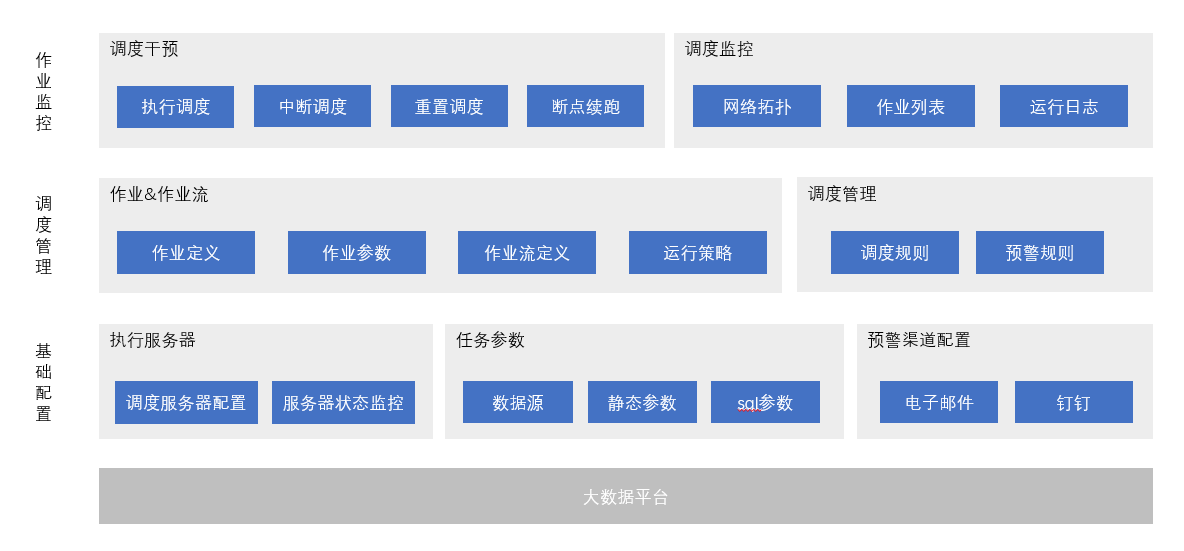
作业监控模块对作业运行状况进行全面的分析，通过自动采集作业运行状况、时间，分析故障分布、作业运行时长等业务运行指标，整体把握系统运行健康度。

通过使用高并发网状调度系统对作业进行自动化管理和监控。当系统出现异常的情况时，以邮件、钉钉等多种方式通知相应的管理员进行人工干预，大大减轻了现有管理员的工作压力，提高了IT系统的管理效率，避免人力成本的虚耗。

## 企业运维自动化

高并发网状调度系统可以为企业提供专业的作业调度服务，全面的调度监控服务，规范企业运维作业执行流程，降低运维复杂度和成本，增强作业执行的安全性，快速发现问题、处理问题。既能高效、快速的解决企业剧增的ETL作业调度需求，也能实现调度作业的全方位监控，有效的提高了企业运维自动化水平。

# 核心模块

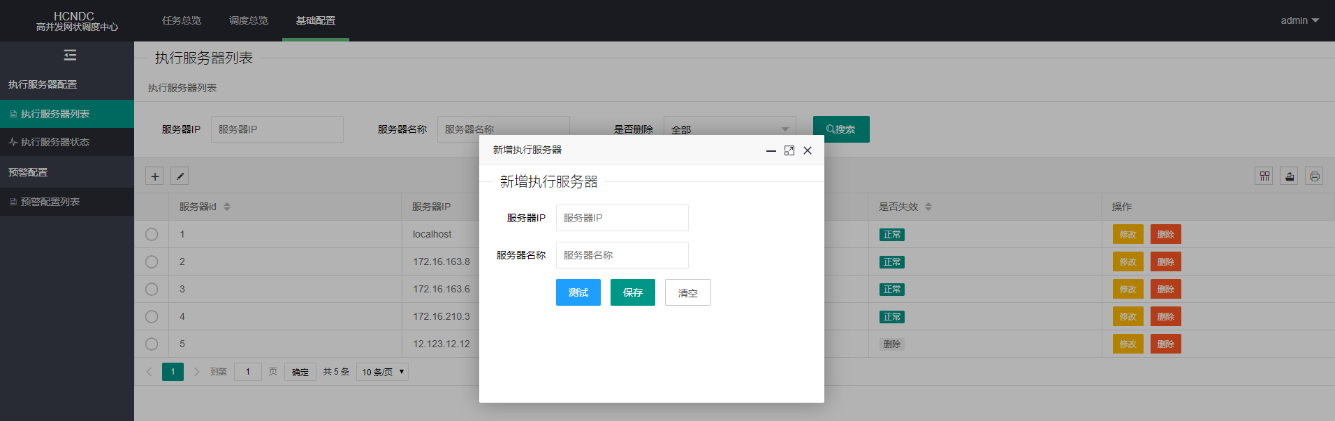


## 基础配置

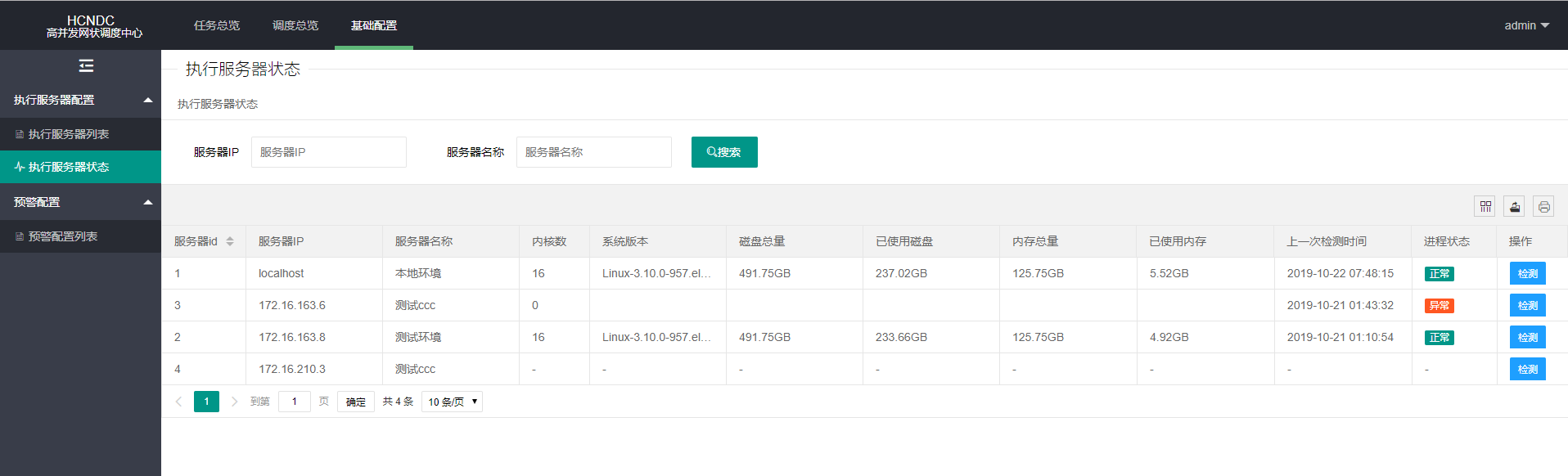
在基础配置模块可以对调度作业相关的执行服务器、调度执行过程中预警下发的渠道以及调度作业涉及的参数进行配置。为后续作业的配置、运行做好前置工作。

### 调度服务器配置

在执行服务器列表中可以对所有服务器进行管理。新增服务器，只需输入IP地址(也可填写服务器域名, 确保web所在服务器有域名映射即可)和自定义服务器名称即可。输入服务器IP地址后，还可以通过“测试”对服务器连接状态进行验证。

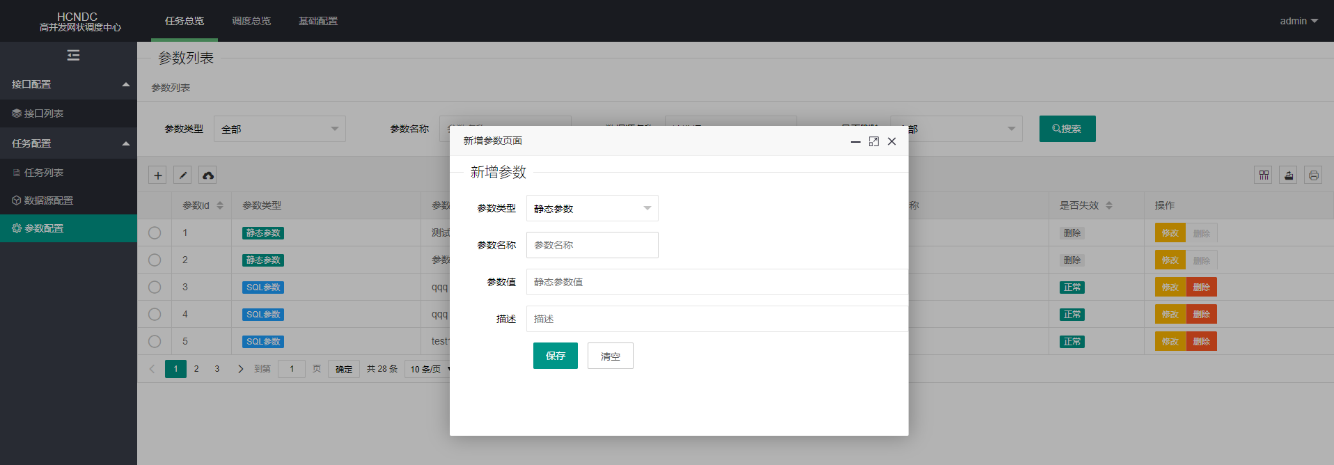


对于已添加的服务器，可在执行服务器状态中统一查看服务器信息，如服务器IP、服务器名称、内核数、系统版本、磁盘总量、已使用磁盘、内存总量、已使用内存、进程状态等。根据服务器的状态信息及时对异常做出反应。

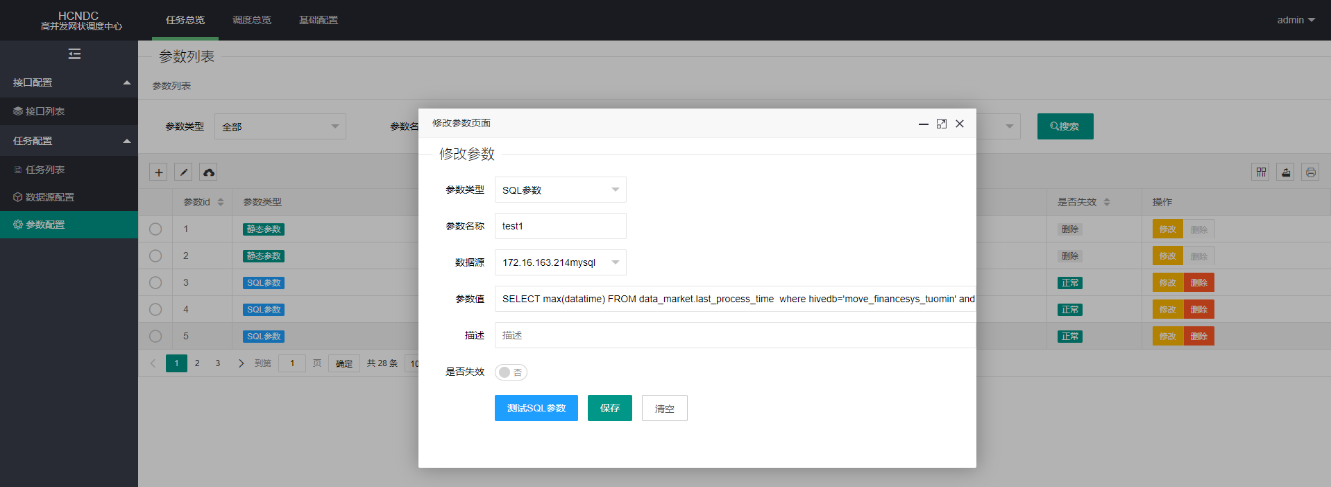


### 作业参数配置

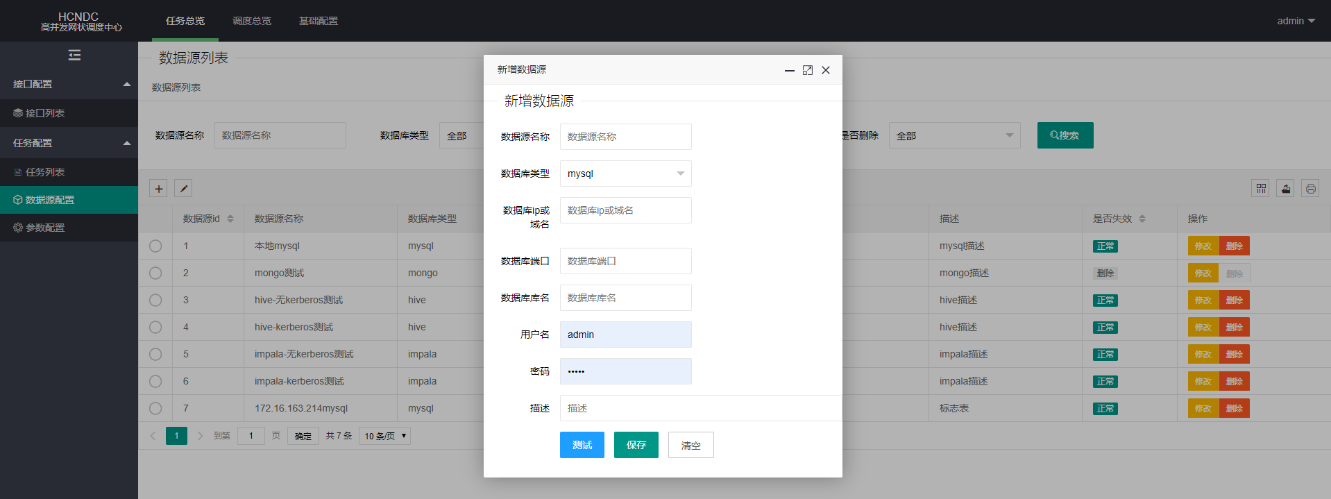
参数配置用来配置全局变量, 为作业配置中作业参数字段使用, 参数配置分为两种, 静态参数和SQL参数。其中静态参数拥有固定的参数值，简单配置参数名称、参数值即可。



SQL参数为动态的参数，这个参数的取值由不同的数据源决定。配置参数名称、数据源、参数值（通过SQL查询标识）即可。

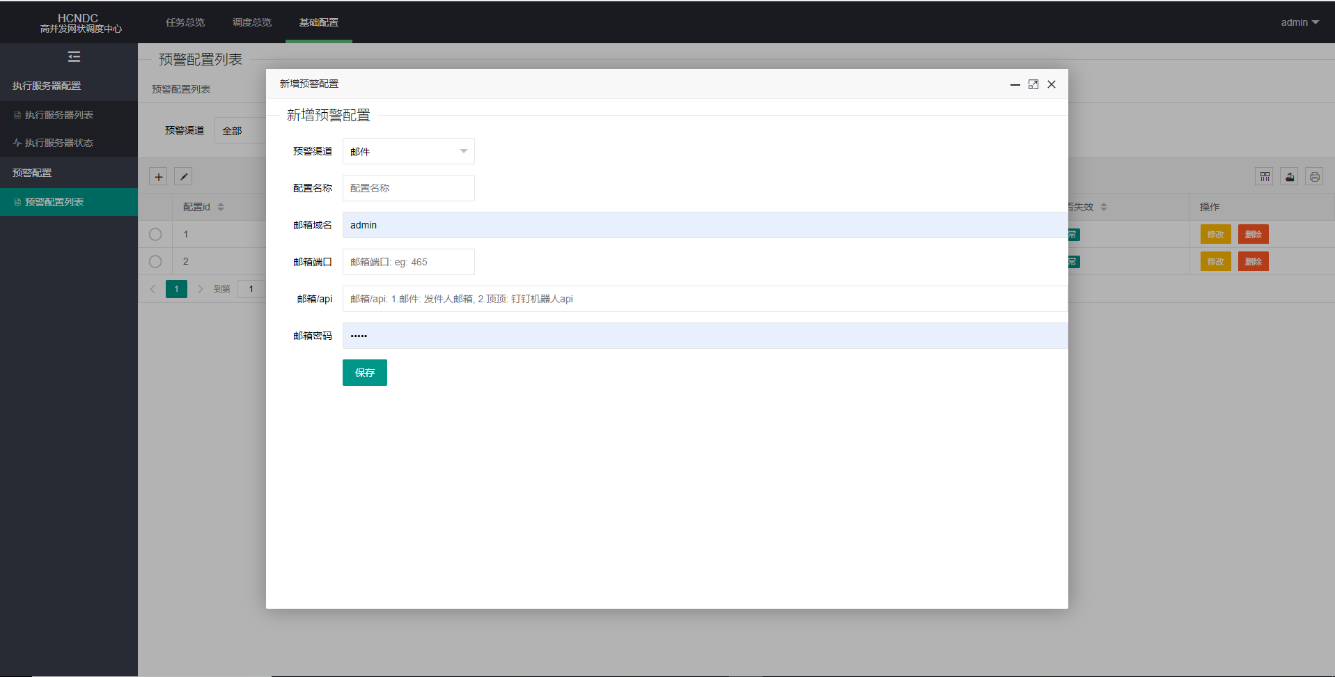


数据源的接入。填写数据库的相关信息即可连接到数据库，进而使用相关的SQL参数。



### 预警配置

在调度运行过程中，当出现了异常情况，第一时间进行预警并进而做出响应，是非常关键的。在预警配置中可以配置不同的预警渠道，在调度作业中只需选择预警渠道和预警规则即可。支持设置的预警渠道类型包括电子邮件、短信、钉钉等。



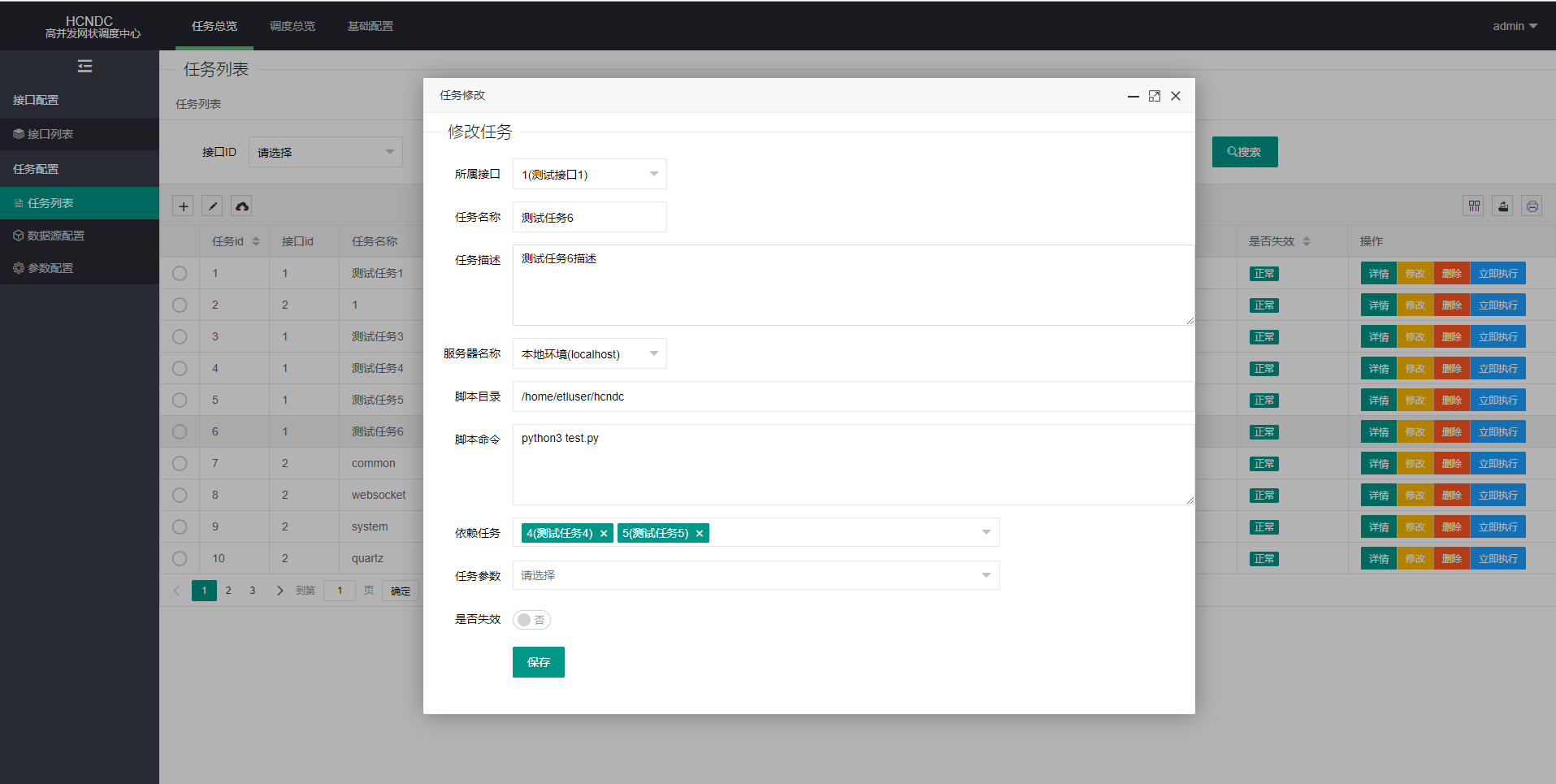
## 调度管理

在调度管理中可以对调度作业进行配置和管理，并按照调度作业之间的依赖关系组合为作业流。调度作业是指一个程序运行单元，比如，一个MapReduce程序、一个Shell脚本等。作业流是指由相互依赖的多个作业组成的一组作业流。

### 调度作业

在调度作业中可以新增、修改、执行、上传、查看、删除作业。在配置作业时可对作业的所属作业流、名称、描述、服务器 （运行环境）、脚本目录、脚本命令、依赖作业、作业参数进行相应设置。

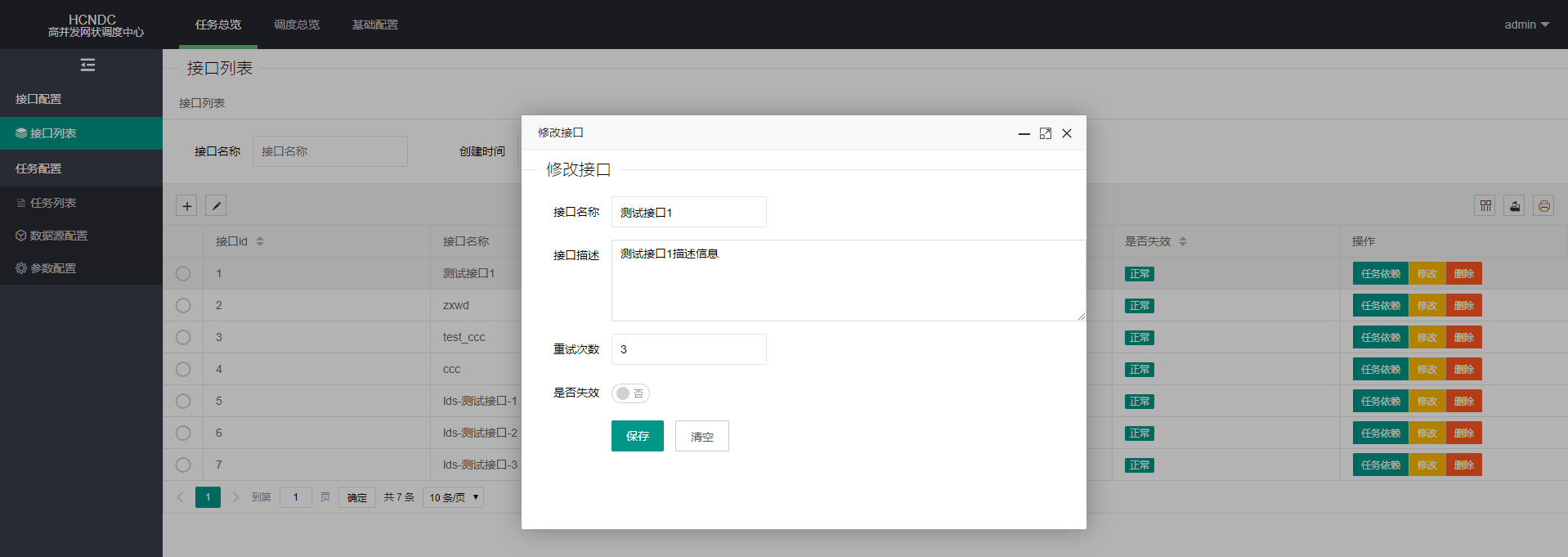
* 作业配置需要选择其所属的作业流
* 一个作业只能属于一个作业流
* 不同作业流之间的任务可以互相依赖
* 任务执行方式为命令行执行
* 环境变量为启动该服务的用户所属的环境变量
* 依赖作业为执行该作业前的前置作业, 可以多选
* 在作业列表可以手动单独触发作业, 不用考虑前置作业是否完成



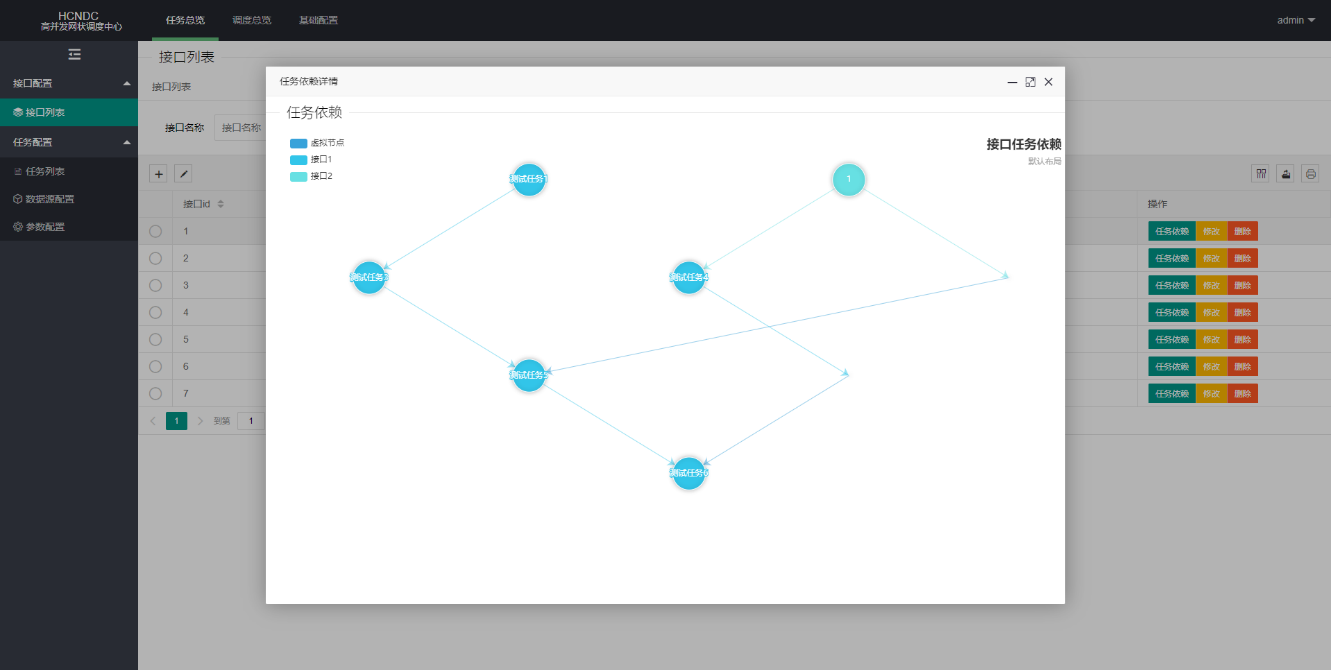
### 作业流

#### 作业依赖和重跑机制

在作业流管理中可以新增、修改、查看、删除作业流。新增作业流时需要设置作业流的名称、描述、重试次数，其中重试次数是指当Master重新启动后，作业流是否需要重新运行，以及可重新运行的次数。



也可以查看调度作业流中作业之间的依赖关系，依赖关系在作业配置中已定义，依赖关系以作业流的形式直观展现。



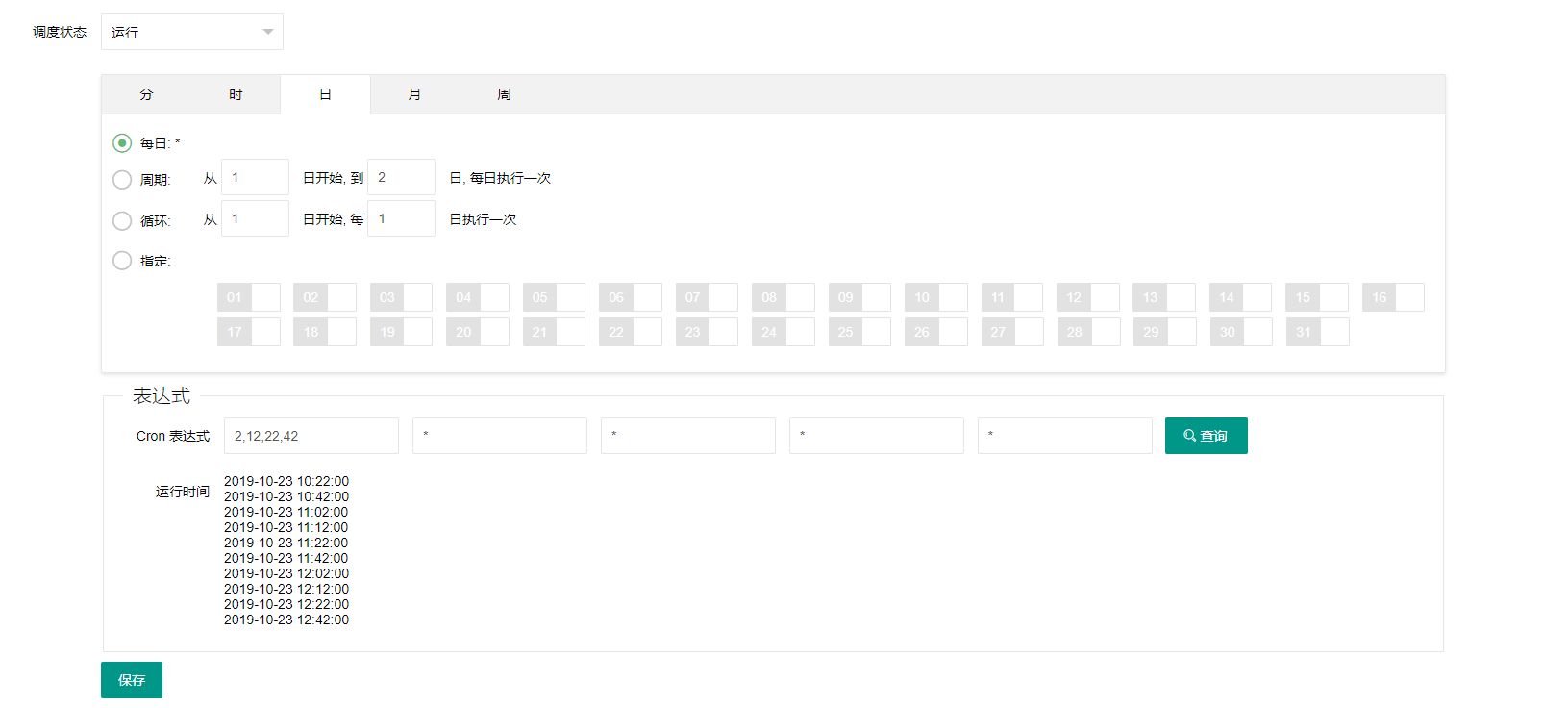
#### 自动调度配置

在配置完作业流中相关的作业和作业依赖关系之后可以根据实际需求设置调度自动执行的规则。

自动执行时需先设定调度执行的规则。其中调度状态可为运行、暂停 、删除。调度周期可以通过两种方式控制，分别是可视化配置方式和Cron表达式。其中可视化方式支持按月、周、日、时、分设置调度日历，同时支持以下多种方式调度方式：

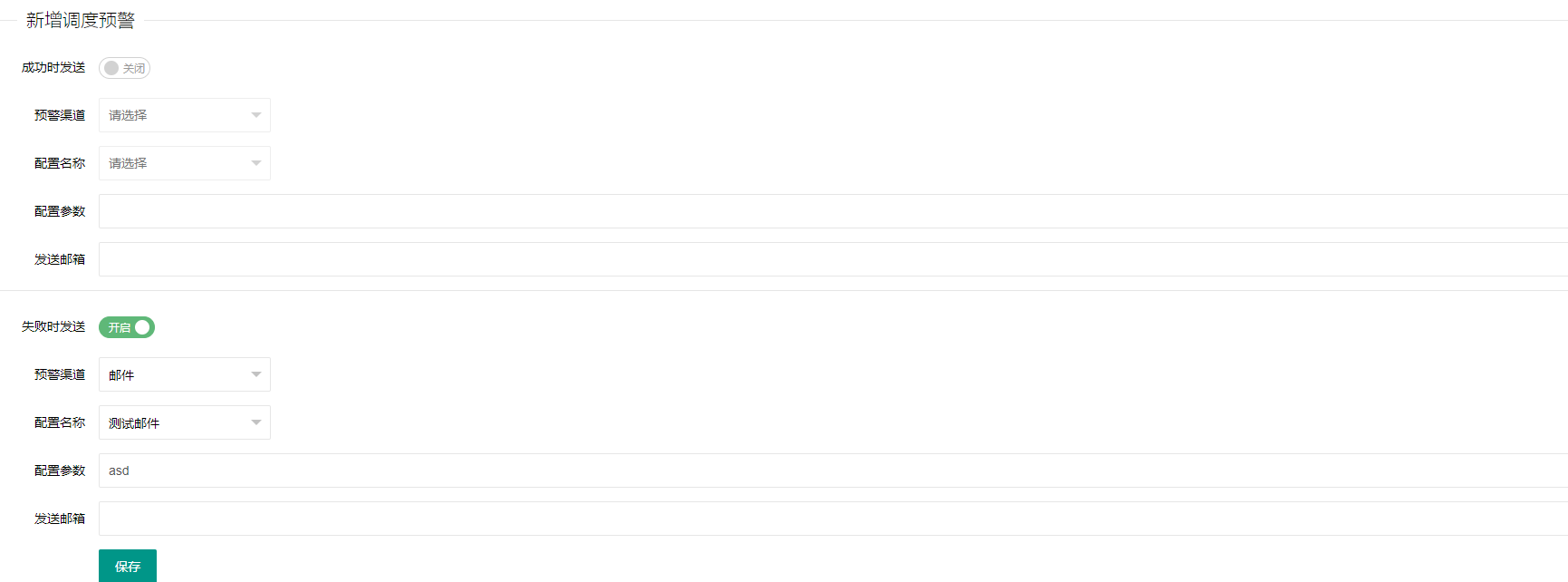
* 固定每个时间周期。如每月、每周、每日等；
* 特定周期执行。如从XX分到XX分，每分执行一次；
* 循环执行。如从XX分开始，每XX分执行一次；
* 指定。直接勾选对应的月份、日期、周、小时、分钟。

也可以通过编辑Cron表达式来设置调度周期。



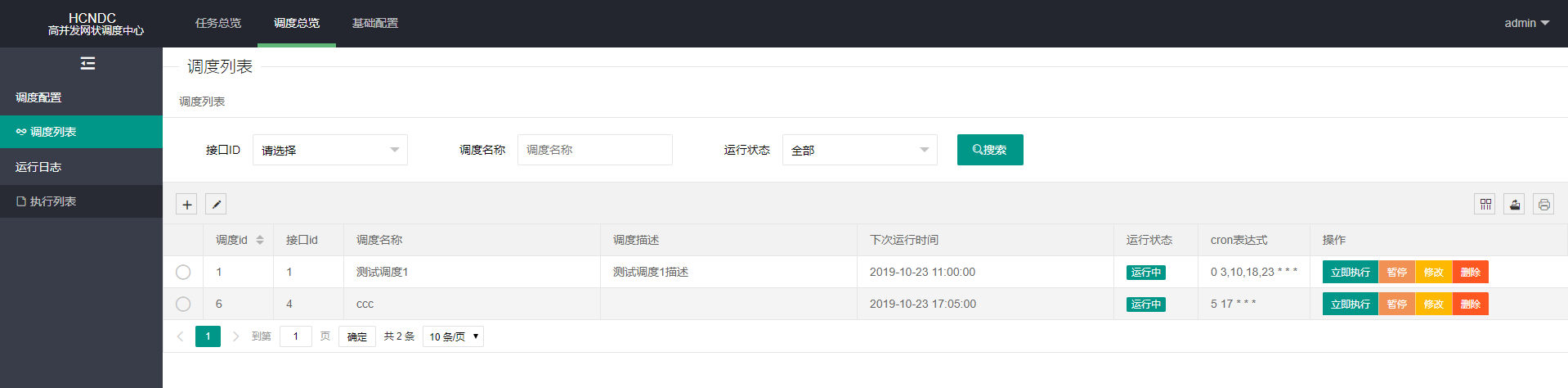
#### 调度预警

可以在调度管理中配置调度执行的预警规则，只需要选择发送预警通知的触发条件、预警渠道、预警配置、接收人即可，在调度执行过程中如果触发了预警规则，将会根据配置向对应的接收人实时发送预警通知。其中预警配置是此前在基础配置模块已添加的预警渠道。



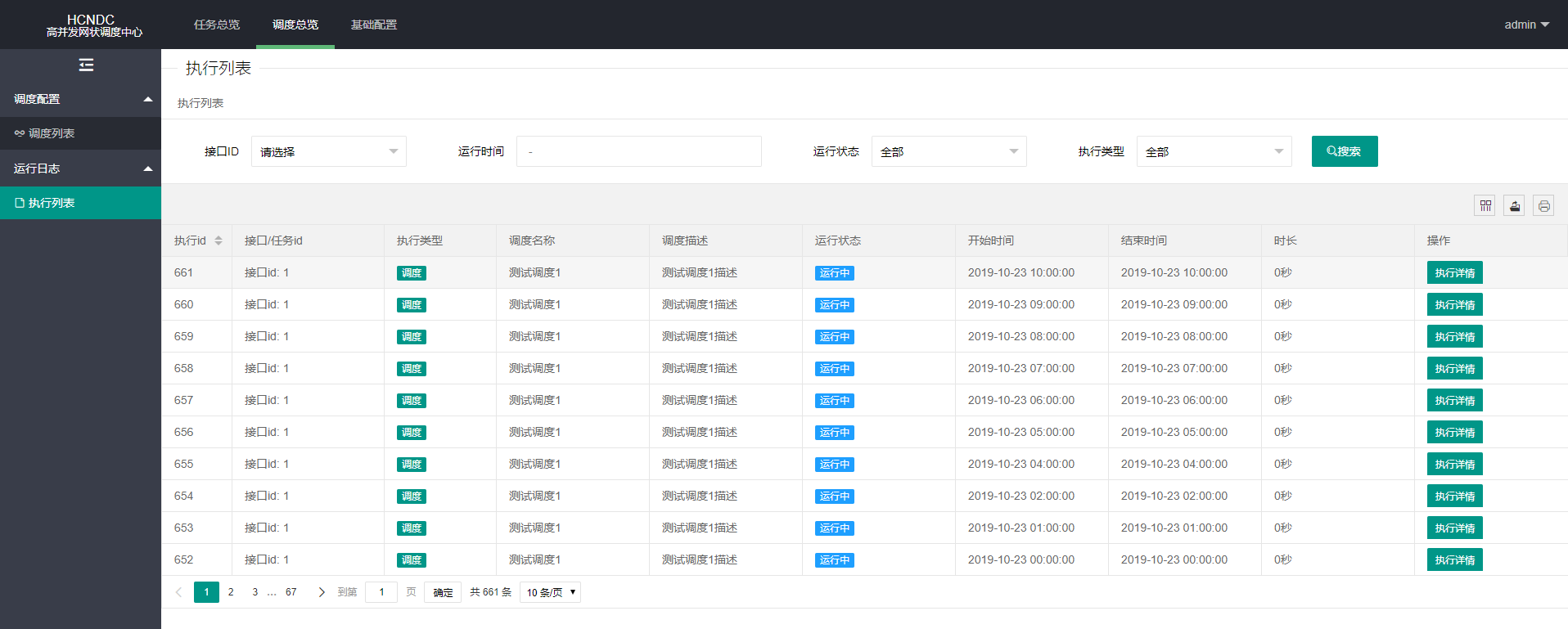
#### 调度手动执行

对于已配置好的调度，可以按照调度规则自动化地执行调度，特殊情况下也可以通过手动进行干预，通过立即执行、暂停、恢复的方式控制调度作业。



## 作业监控

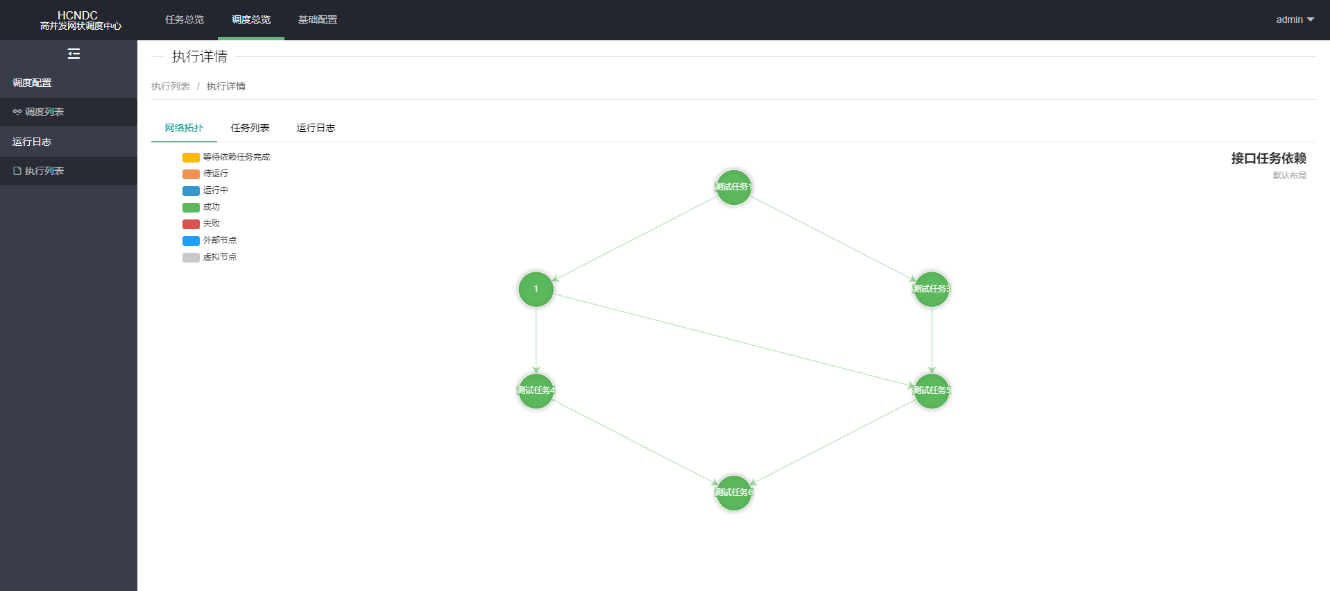
在作业监控中，可以查看每一次执行作业的日志, 包括手动执行和调度执行。作业的执行详情中包括有网络拓扑图、任务列表、运行日志等功能。作业监控模块对作业运行状况进行全面的分析，通过自动采集作业运行状况、时间，分析故障分布、作业运行时长等业务运行指标，整体把握系统运行健康度。



### 作业执行详情

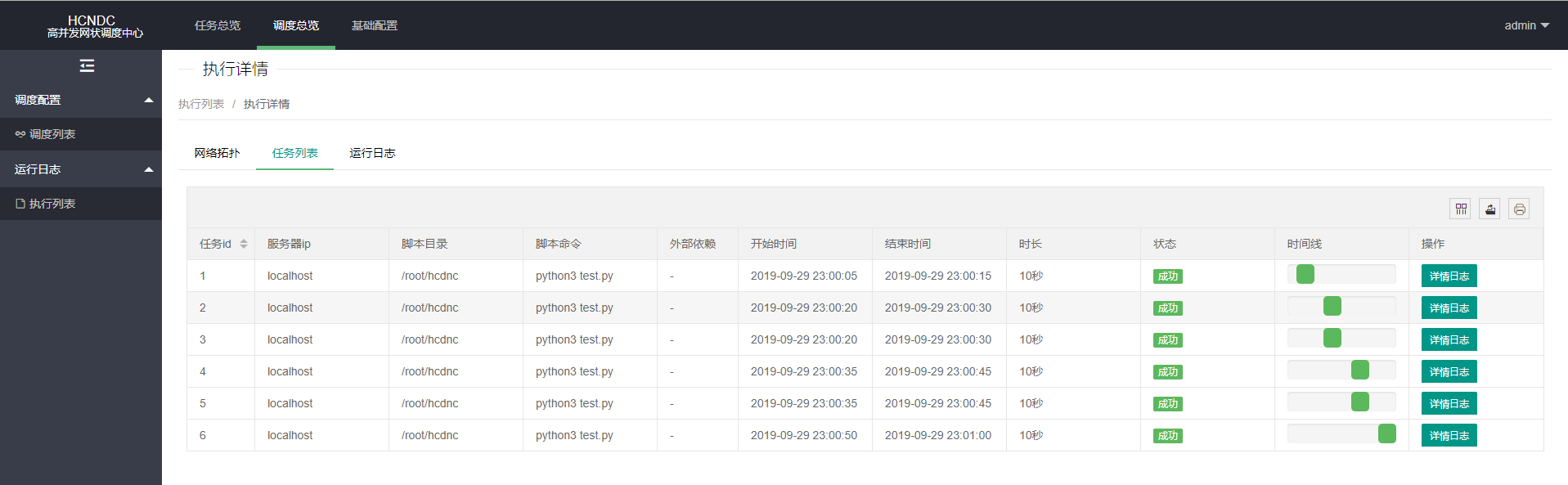
#### 网络拓扑

网络拓扑图对应于调度中的作业依赖关系图（作业流）。通过拓扑图我们可以直观清楚的判断调度中各节点（作业）的执行状态，快速做出判断和反映。

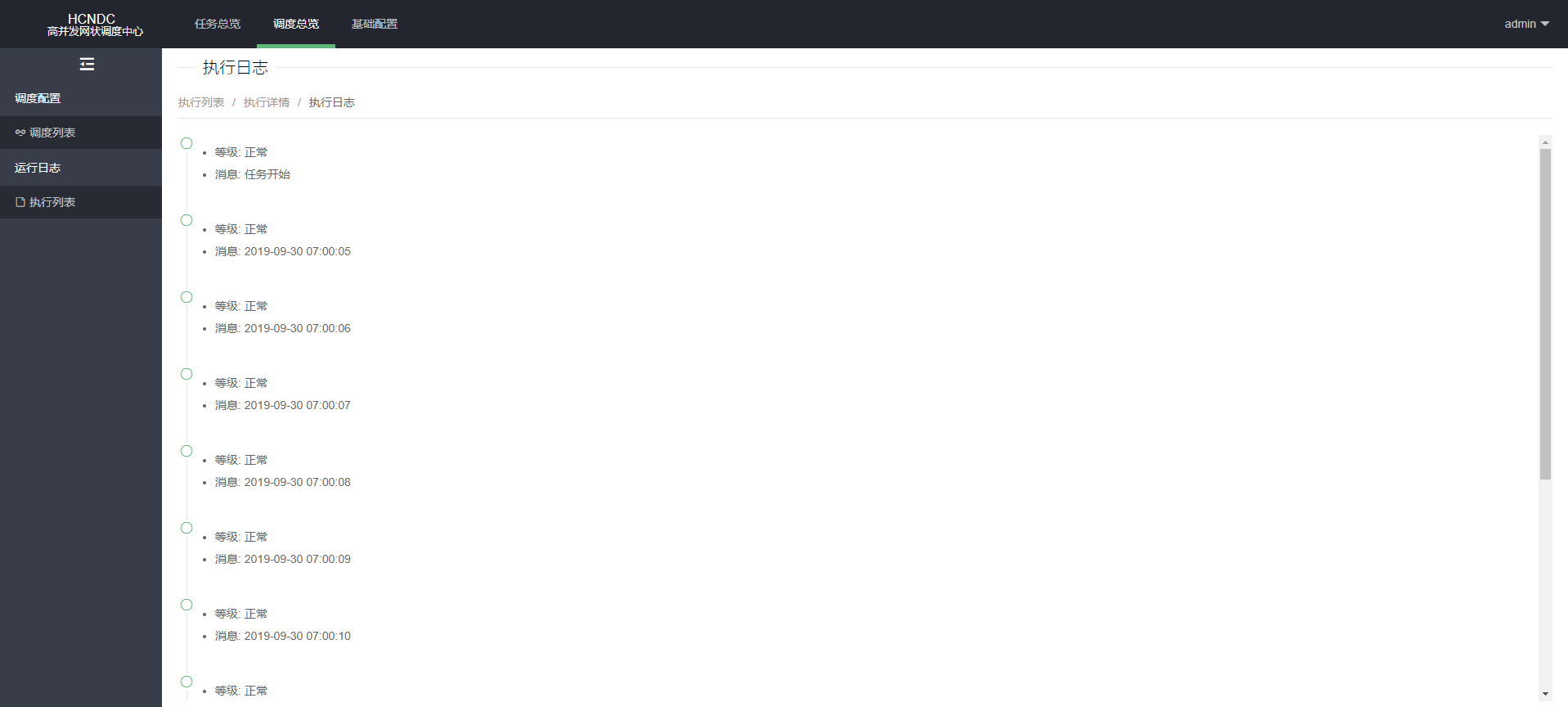


#### 作业列表

作业列表中记录的是当前调度最近一次执行时各节点的执行情况。各个作业按执行的时间顺序依次排列，展示任务执行状态等信息，并且以甘特图的形式展现作业执行的时间线。

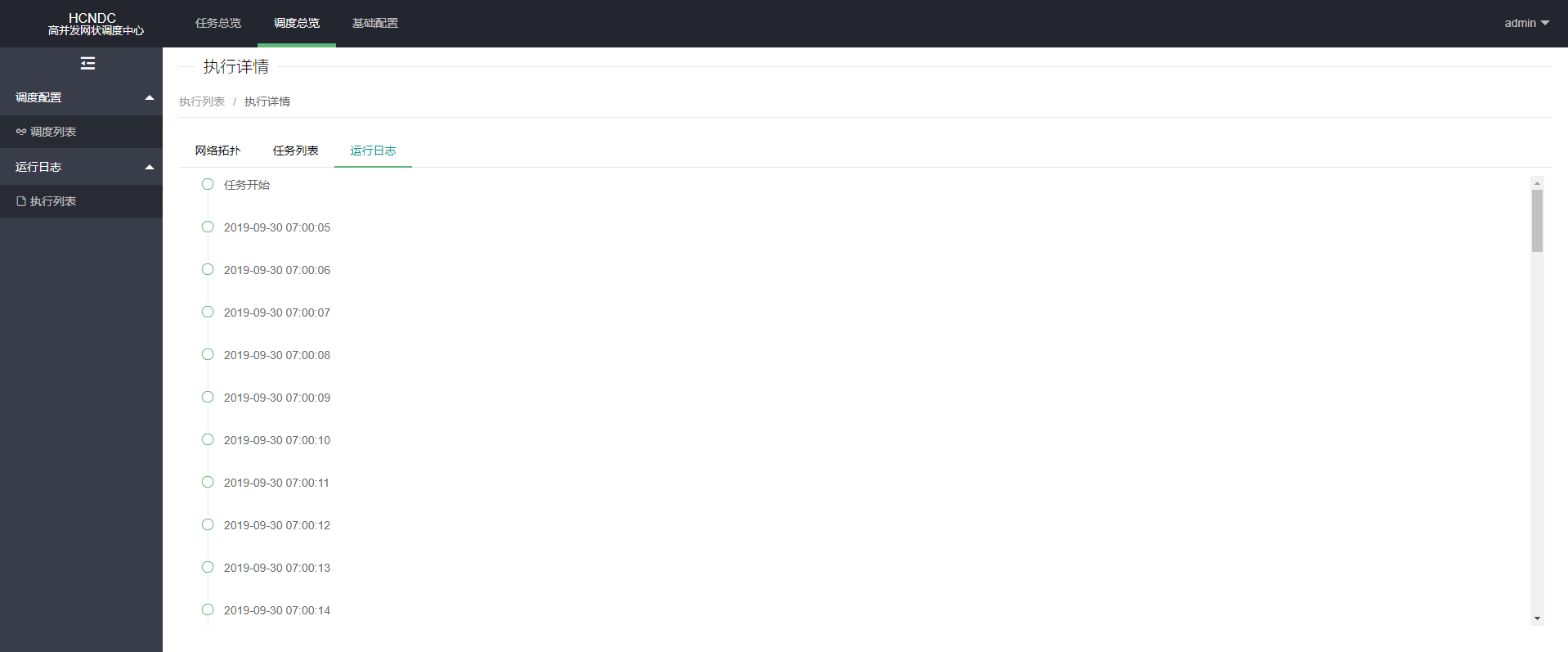


同时还可以查看调度中各作业的详细的执行日志，通过点击作业对应的详情日志按钮即可查看。正常执行的作业将显示执行成功时间，异常任务将给出对应的异常提示。管理员可以通过日志快速锁定问题，并及时处理，减少损失。



#### 运行日志

运行日志中记录了调度历史上每一次执行的执行。正常执行的作业将显示执行成功时间，异常任务将给出对应的异常提示。管理员可以通过日志快速锁定问题，并及时处理，减少损失。



# 产品优势

## 分布式架构

容量和负载能力(JobWorker)可线性扩充，并可以跨外网部署（只需能访问元数据库即可）；开放系统任务调度触发接口。

## 高可用性

拥有主备Master，一旦主Master异常，备Master会接替主Master提供服务。

## 高容错性

Master重新启动后，会将之前未完成的任务重新调度运行；任务自动恢复。支持更加高级的任务恢复机制，比如根据数据血缘分析恢复与表，字段相关的任务。

## 支持任意类型的作业

除了Hadoop生态圈的MapReduce、Hive、Pig等，还支持其他任何语言开发的作业，如Java、Shell、Python、Perl、Spark等。

## 交互友好性

完善易用的Web用户界面，用于用户配置、提交、查询、监控任务及任务的依赖关系；可视化实现任务之间复杂的依赖。

## 作业之间的灵活依赖

可将任意一个任务作为自己的父任务进行依赖触发。作业依赖配置完成后自动生成作业流，可视化展现任务之间复杂的依赖关系。

## 强大的作业调度方式

按照业务需求灵活配置多种作业调度方式，提供日历、频度、表达式等方式进行调度规则配置。 支持调度作业定时调度、依赖调度、手动调度、手动暂停/停止/恢复等操作。

## 灵活多样的告警规则

支持接入邮件、钉钉等多种告警渠道。支持多种告警规则配置，除了失败告警，也支持调度超时未完成、调度超时未开始等告警规则。可以大大解放人力，自动化监控系统的各种运行状态。

## 完整的日志记录

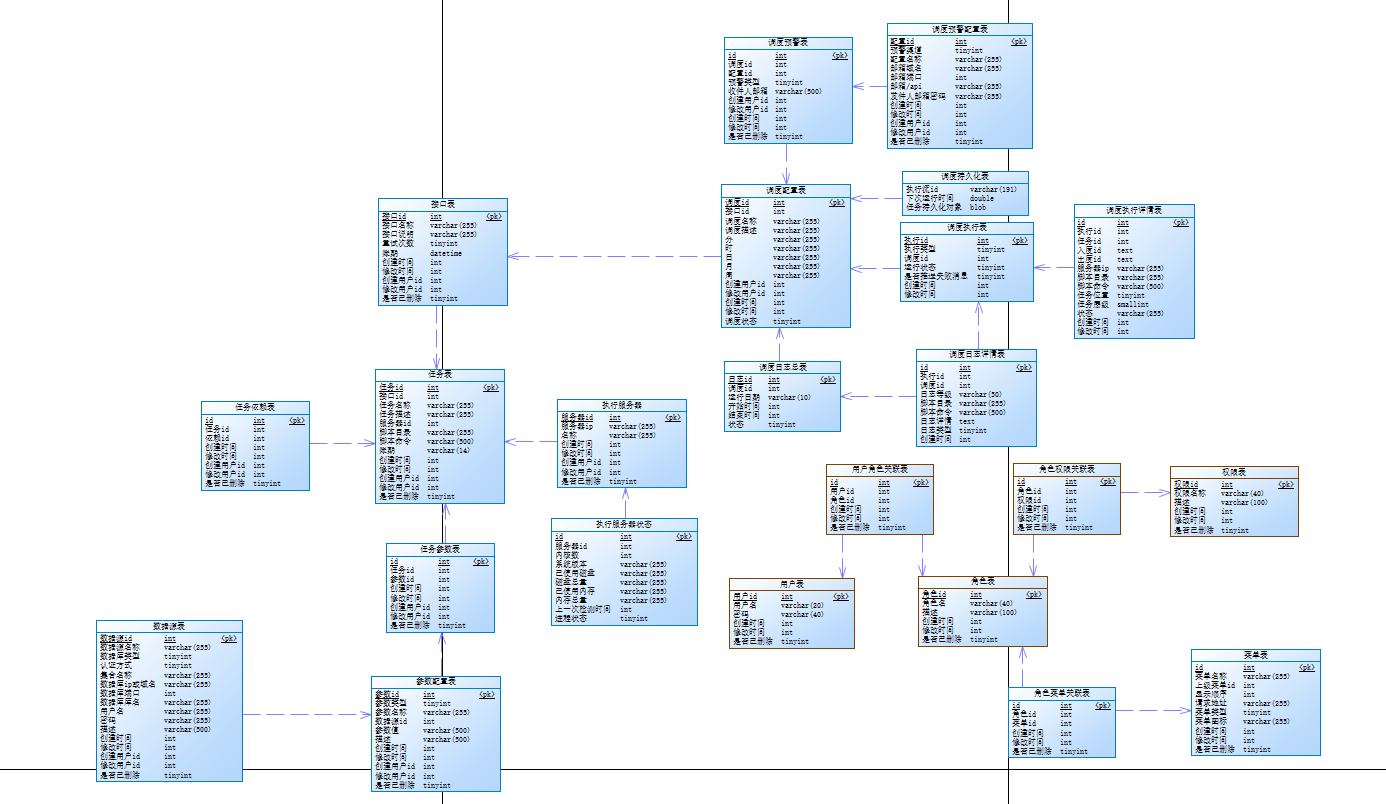
收集并记录调度运行过程中产生的标准输出和标准错误，用户可通过访问对应调度和作业的监控日志来方便的访问调度运行的网络拓扑图、作业执行明细、调度运行日志。

## 丰富的应用场景

除了大数据平台 ETL调度外，本系统还可以支持支付系统定期清算、电子商城优惠活动上下架、营销通知下发等丰富的作业调度场景。

# 程序架构

## 数据库设计



主要模块：

* 调度模块：调度、持久化、调度预警表
* 用户模块：用户、角色、权限表
* 执行模块：执行、详情表
* 任务模块：接口、任务表

## 环境配置

* 系统要求：windows或Linux
* 语言：python3.6
* 框架：后端-Flask、前端-Echarts、UI-layui

## 微服务架构

微服务架构（Microservice Architecture）是一种架构概念，旨在通过将功能分解到各个离散的服务中以实现对解决方案的解耦。你可以将其看作是在架构层次而非获取服务的

类上应用很多SOLID原则。它的主要作用是将功能分解到离散的各个服务当中，从而降低系统的耦合性，并提供更加灵活的服务支持。

概念：把一个大型的单个应用程序和服务拆分为数个甚至数十个的支持微服务，它可扩展单个组件而不是整个的应用程序堆栈，从而满足服务等级协议。

定义：围绕业务领域组件来创建应用，这些应用可独立地进行开发、管理和迭代。在分散的组件中使用云架构和平台式部署、管理和服务功能，使产品交付变得更加简单。

本质：用一些功能比较明确、业务比较精练的服务去解决更大、更实际的问题。

前后端分离，使用Swagger描述后端API，后端只完成API部分代码，与前端进行解耦，方便测试和调试。



## docker部署

Docker 是一个开源的应用容器引擎，基于 Go 语言 并遵从 Apache2.0 协议开源。Docker 可以让开发者打包他们的应用以及依赖包到一个轻量级、可移植的容器中，然后发布到任何流行的 Linux 机器上，也可以实现虚拟化。

容器是完全使用沙箱机制，相互之间不会有任何接口（类似 iPhone 的 app）,更重要的是容器性能开销极低。

持续部署与测试：Docker在开发与运维的世界中具有极大的吸引力，因为它能保持跨环境的一致性。在开发与发布的生命周期中，不同的环境具有细微的不同，这些差异可能是由于不同安装包的版本和依赖关系引起的。然而，Docker可以通过确保从开发到产品发布整个过程环境的一致性来解决这个问题\*Docker容器通过相关配置，保持容器内部所有的配置和依赖关系始终不变。最终，你可以在开发到产品发布的整个过程中使用相同的容器来确保没有任何差异或者人工干预。

使用Docker，你还可以确保开发者不需要配置完全相同的产品环境，他们可以在他们自己的系统上通过VirtualBox建立虚拟机来运行Docker容器。Docker的魅力在于它同样可以让你在亚马逊EC2实例上运行相同的容器。如果你需要在一个产品发布周期中完成一次升级，你可以很容易地将需要变更的东西放到Docker容器中，测试它们，并且使你已经存在的容器执行相同的变更。这种灵活性就是使用Docker的一个主要好处。和标准部署与集成过程一样，Docker可以让你构建、测试和发布镜像，这个镜像可以跨多个服务器进行部署。哪怕安装一个新的安全补丁，整个过程也是一样的。你可以安装补丁，然后测试它，并且将这个补丁发布到产品中。

多云平台：Docker最大的好处之一就是可移植性。在过去的几年里，所有主流的云计算提供商，包括亚马逊AWS和谷歌的GCP，都将Docker融入到他们的平台并增加了各自的支持。Docker容器能运行在亚马逊的EC2实例、谷歌的GCP实例、Rackspace服务器或者VirtualBox这些提供主机操作系统的平台上。举例来说，如果运行在亚马逊EC2实例上的Docker容器能够很容易地移植到其他几个平台上，比如说VirtualBox，并且达到类似的一致性和功能性，那这将允许你从基础设施层中抽象出来。除了AWS和GCP，Docker在其他不同的IaaS提供商也运行的非常好，例如微软的Azure、OpenStack和可以被具有不同配置的管理者所使用的Chef、Puppet、Ansible等。

环境标准化和版本控制：通过上面的讨论，Docker容器可以在不同的开发与产品发布生命周期中确保一致性，进而标准化你的环境。除此之外，Docker容器还可以像git仓库一样，可以让你提交变更到Docker镜像中并通过不同的版本来管理它们。设想如果你因为完成了一个组件的升级而导致你整个环境都损坏了，Docker可以让你轻松地回滚到这个镜像的前一个版本。这整个过程可以在几分钟内完成，如果和虚拟机的备份或者镜像创建流程对比，那Docker算相当快的，它可以让你快速地进行复制和实现冗余。此外，启动Docker就和运行一个进程一样快。

隔离性：Docker可以确保你的应用程序与资源是分隔开的。几个月前，Gartner发表了一篇报告，这份报告说明了运行Docker 容器进行资源隔离的效果和虚拟机（VM）管理程序一样的好，但是管理与控制方面还需要进行完善。

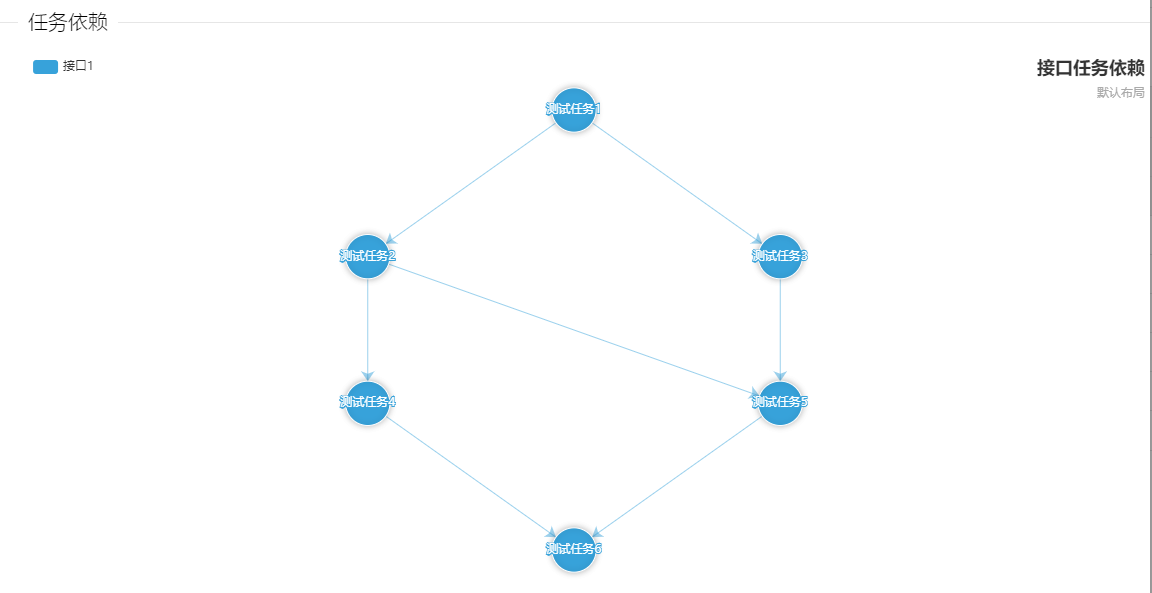
安全性：如上所述，Gartner也承认Docker正在快速地发展。从安全角度来看，Docker确保运行在容器中的应用程序和其他容器中的应用程序是完全分隔与隔离的，在通信流量和管理上赋予你完全的控制权。Docker容器不能窥视运行在其他容器中的进程。从体系结构角度来看，每个容器只使用着自己的资源（从进程到网络堆栈）。

作为紧固安全的一种手段，Docker将宿主机操作系统上的敏感挂载点（例如/proc和/sys）作为只读挂载点，并且使用一种写时复制系统来确保容器不能读取其他容器的数据。Docker也限制了宿主机操作系统上的一些系统调用，并且和SELinux与AppArmor一起运行的很好。此外，在Docker Hub上可以使用的Docker镜像都通过数字签名来确保其可靠性。由于Docker容器是隔离的，并且资源是受限制的，所以即使你其中一个应用程序被黑，也不会影响运行在其它Docker容器上的应用程序。



## 前端设计

前端模板为静态文件，所有动态资源异步请求获得，后端API响应时间均在0.5秒之内。其中，任务流数据结构渲染使用DAG有向无环图。DAG原本是计算机领域一种常用数据结构，因为独特的拓扑结构所带来的优异特性，经常被用于处理动态规划、导航中寻求最短路径、数据压缩等多种算法场景。本模块使用广度优先遍历，层级间使用虚拟节点，保证可视化效果，3000复杂节点构造数据结构所需时间在10秒之内。



## 后端设计

RESTFUL API设计：

客户端-服务器：通过将用户UI与数据存储分开，我们可以简化服务器组件来提高跨多个平台的用户界面的可移植性并提高可伸缩性。 它可以比表现成前后端分离的思想。

无状态：从客户端到服务器的每个请求都必须包含理解请求所需的所有信息，并且不能利用服务器上任何存储的上下文。 这表示你应该尽可能的避免使用session，由客户端自己标识会话状态。（token）

规范接口：REST接口约束定义：资源识别; 请求动作; 响应信息; 它表示通过uri标出你要操作的资源，通过请求动作（http method）标识要执行的操作，通过返回的状态码来表示这次请求的执行结果。

可缓存： 缓存约束要求将对请求的响应中的数据隐式或显式标记为可缓存或不可缓存。如果响应是可缓存的，则客户端缓存有权重用该响应数据以用于以后的等效请求。 它表示get请求响应头中应该表示有是否可缓存的头（Cache-Control)

函数装饰器设计：

装饰器本质上是一个 Python 函数或类，它可以让其他函数或类在不需要做任何代码修改的前提下增加额外功能，装饰器的返回值也是一个函数/类对象。它经常用于有切面需求的场景，比如：插入日志、性能测试、事务处理、缓存、权限校验等场景，装饰器是解决这类问题的绝佳设计。有了装饰器，我们就可以抽离出大量与函数功能本身无关的雷同代码到装饰器中并继续重用，@ 符号就是装饰器的语法糖，它放在函数开始定义的地方，这样就可以省略最后一步再次赋值的操作。项目中使用了4层装饰器，使得代码可读性大幅提高，并且使用起来更加方便。

## 调度持久化和序列化

持久化（Persistence），即把数据（如内存中的对象）保存到可永久保存的存储设备中（如磁盘）。持久化的主要应用是将内存中的对象存储在关系型的数据库中，当然也可以存储在磁盘文件中、XML数据文件中等等。持久化是将程序数据在持久状态和瞬时状态间转换的机制。JDBC就是一种持久化机制。文件IO也是一种持久化机制。

持久化是一种对象服务，就是把内存中的对象保存到外存中，让以后能够取回。需要实现至少3个接口：

* 把一个对象保存到外存中
* 通过对象标识从外存中取回对象
* 检查外存中是否存在某个对象

序列化也是一种对象服务，就是把内存中的对象序列化成流、或者把流反序列化成对象。需要实现2个接口：

* 把对象序列化到流中
* 把流反序列化成对象

序列化和持久化很相似，有些人甚至混为一谈，其实还是有区别的，序列化是为了解决对象的传输问题，传输可以在线程之间、进程之间、内存外存之间、主机之间进行。我之所以在这里提到序列化，是因为我们可以利用序列化来辅助持久化，可以说凡是可以持久化的对象都可以序列化，因为序列化相对容易一些（也不是很容易），所以主流的软件基础设施，比如.net和java，已经把序列化的框架完成了。

持久化方案可以分为关系数据库方案、文件方案、对象数据库方案、xml数据库方案，现今主流的持久化方案是关系数据库方案，关系数据库方案不仅解决了并发的问题，更重要的是，关系数据库还提供了持久化服务之外的价值：统计分析功能。

调度任务使用持久化存储，遇到机器宕机或服务异常，重启项目之后，调度任务会自动序列化，保证调度任务的正常运行。

## RPC通信

WEB端和执行端之间使用RPC通信，RPC是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的协议。主要是基于TCP/IP协议的，而HTTP服务主要是基于HTTP协议的， HTTP协议是在传输层协议TCP之上的，所以效率来看的话，在调用速度上，RPC更胜一筹。

RPC 主要特质：

RPC 是协议：协议意味着规范。目前典型的 RPC 实现包括Hetty 等。但这些实现往往都会附加其他重要功能，例如 Du理、访问权限管理等功能。

网络协议和网络 IO 模型对其透明：既然 RPC 的客户端认地对象。那么传输层使用的是 TCP/UDP 还是 HTTP 协议，又或络协议它就不需要关心了。既然网络对其透明，那么调哪一种网络 IO 模型调用者也不需要关心。

信息格式对其透明：远程调用过程中，需要传递一些参调用结果。至于这些参数会以某种信息格式传递给网络上的个信息格式是怎样构成的，调用方是不需要关心的。