

目录

Easyscheduler简介	1.1
前端文档	1.2
环境搭建	1.2.1
安装及配置	1.2.2
项目生产环境Nginx配置	1.2.3
前端项目发布	1.2.4
问题	1.2.5
项目目录结构	1.2.6
系统功能模块	1.2.7
路由和状态管理	1.2.8
规范	1.2.9
接口	1.2.10
扩展开发	1.2.11
后端文档	1.3
系统架构设计	1.3.1
部署文档	1.3.2
自定义任务插件文档	1.3.3
使用说明文档	1.4

Easy Scheduler

license Apache 2

Easy Scheduler for Big Data

设计特点：一个分布式易扩展的可视化DAG工作流任务调度系统。致力于解决数据处理流程中错综复杂的依赖关系，使调度系统在数据处理流程中开箱即用。其主要目标如下：

- 以DAG图的方式将Task按照任务的依赖关系关联起来，可实时可视化监控任务的运行状态
- 支持丰富的任务类型：Shell、MR、Spark、SQL(mysql、postgresql、hive、sparksql)、Python、Sub_Process、Procedure等
- 支持工作流定时调度、依赖调度、手动调度、手动暂停/停止/恢复，同时支持失败重试/告警、从指定节点恢复失败、Kill任务等操作
- 支持工作流优先级、任务优先级及任务的故障转移及任务超时告警/失败
- 支持工作流全局参数及节点自定义参数设置
- 支持资源文件的在线上传/下载，管理等，支持在线文件创建、编辑
- 支持任务日志在线查看及滚动、在线下载日志等
- 实现集群HA，通过Zookeeper实现Master集群和Worker集群去中心化
- 支持对 Master/Worker cpu load, memory, cpu在线查看
- 支持工作流运行历史树形/甘特图展示、支持任务状态统计、流程状态统计
- 支持补数
- 支持多租户
- 支持国际化
- 还有更多等待伙伴们探索

与同类调度系统的对比

	EasyScheduler	Azkaban	Airflow
稳定性			
单点故障	去中心化的多Master和多Worker	是 单个Web和调度程序组合	是 单一调度程序
HA额外要求	不需要(本身就支持HA)	DB	Celery / Dask / Mesos + Load Balancer + DB
过载处理	任务队列机制，单个机器上可调度的任务数量可以灵活配置，当任务过多时会缓存在任务队列中，不会造成机器卡死	任务太多时会卡死服务器	任务太多时会卡死服务器
易用性			
DAG监控界面	任务状态、任务类型、重试次数、任务运行机器、可视化变量等关键信息一目了然	只能看到任务状态	不能直观区分任务类型
可视化流程定义	是 所有流程定义操作都是可视化的，通过拖拽任务来绘制DAG，配置数据源及资源。	否 通过自定义DSL绘制DAG并打包上	否 通过python代码来绘制DAG，使用不便，特别是对不会

	同时对于第三方系统，提供api方式的操作。	传统	写代码的业务人员基本无法使用。
快速部署	一键部署	集群化部署复杂	集群化部署复杂
功能			
是否能暂停和恢复	支持暂停，恢复操作	否 需将工作流杀死再运行	否 需将工作流杀死再运行
是否支持多租户	支持 easyscheduler上的用户可以通过租户和hadoop用户实现多对一或一对一的映射关系，这对大数据作业的调度是非常重要的。	否	否
任务类型	支持传统的shell任务，同时支持大数据平台任务调度：MR、Spark、SQL(mysql、postgresql、hive、sparksql)、Python、Procedure、Sub_Process	shell、gobblin、hadoopJava、java、hive、pig、spark、hdfsToTeradata、teradataToHdfs	BashOperator、 DummyOperator、 MySqlOperator、 HiveOperator、 EmailOperator、 HTTPOperator、 SqlOperator
契合度	支持大数据作业 spark,hive,mr的调度，同时由于支持多租户，与大数据业务更加契合	由于不支持多租户，在大数据平台业务使用不够灵活	由于不支持多租户，在大数据平台业务使用不够灵活
扩展性			
是否支持自定义任务类型	是	是	是
是否支持集群扩展	是 调度器使用分布式调度，整体的调度能力会随集群的规模线性增长，Master和Worker支持动态上下线	是，但是复杂 Executor水平扩展	是，但是复杂 Executor水平扩展

系统部分截图

工作流实例

编号	工作流名称	运行类型	开始时间	结束时间	运行时长(s)	运行次数	host	容错标识	状态	操作
1	test_Id=0-1551787290362	重跑	2019-03-05 20:11:17	2019-03-05 20:11:37	20	2	127.0.0.1	NO	正常	编辑 重跑 恢复失败 停止 暂停 删除 甘特图
2	test_Id=0-1551787754296	从失败节点开始执行	2019-03-05 20:09:14	2019-03-05 20:10:34	80	2	127.0.0.1	NO	失败	编辑 重跑 恢复失败 停止 暂停 删除 甘特图
3	test_Id=0-1551787367803	启动工作流	2019-03-05 20:02:48	2019-03-05 20:02:59	11	1	127.0.0.1	NO	正常	编辑 重跑 恢复失败 停止 暂停 删除 甘特图
4	test_Id=0-1551787037794	启动工作流	2019-03-05 19:57:18	2019-03-05 19:57:36	18	1	127.0.0.1	NO	正常	编辑 重跑 恢复失败 停止 暂停 删除 甘特图
5	test_Id=0-1551787026739	启动工作流	2019-03-05 19:57:07	2019-03-05 19:57:25	18	1	127.0.0.1	NO	正常	编辑 重跑 恢复失败 停止 暂停 删除 甘特图
6	test_Id=0-1551787023456	启动工作流	2019-03-05 19:57:03	2019-03-05 19:57:23	20	1	127.0.0.1	NO	正常	编辑 重跑 恢复失败 停止 暂停 删除 甘特图
7	test_Id=0-1551787020007	启动工作流	2019-03-05 19:57:00	2019-03-05 19:57:18	18	1	127.0.0.1	NO	正常	编辑 重跑 恢复失败 停止 暂停 删除 甘特图
8	test_Id=0-155178701664	启动工作流	2019-03-05 19:56:52	2019-03-05 19:57:10	18	1	127.0.0.1	NO	正常	编辑 重跑 恢复失败 停止 暂停 删除 甘特图
9	test_Id=0-1551786897561	启动工作流	2019-03-05 19:54:58	2019-03-05 19:55:16	18	1	127.0.0.1	NO	正常	编辑 重跑 恢复失败 停止 暂停 删除 甘特图
10	test_Id=0-1551786873503	启动工作流	2019-03-05 19:52:15	2019-03-05 19:52:35	20	1	127.0.0.1	NO	正常	编辑 重跑 恢复失败 停止 暂停 删除 甘特图

当前节点设置

节点名称: 1
运行标志: 正常 禁止执行
描述: 2
失败重试次数: 0 (次) 失败重试间隔: 1 (分)
脚本: 1 sleep 2
资源: 请选择资源
自定义参数: +
外部依赖: +
自身依赖: 不依赖 依赖上一周期
取消 确认添加

```

graph TD
    1[1] --> 3[3]
    1[1] --> 4[4]
    3[3] --> 6[6]
    3[3] --> 5[5]
    4[4] --> 5[5]
    2[2] --> 6[6]
    2[2] --> 5[5]
  
```

树形图

节点类型: SHELL, SUB_PROCESS, PROCEDURE, SQL, SPARK, MR, PYTHON
任务状态: 提交成功, 正在执行, 准备暂停, 暂停, 准备停止, 停止, 失败, 成功, 需要容错, kill, 等待线程, 等待依赖

DAG tree structure:

- Root: DAG
 - Child: shell
 - Child: 2
 - Child: 3
 - Child: 5

文档

- 部署文档 [后端部署文档](#)

[前端部署文档](#)

[使用手册](#)

更多文档请参考 XXX

帮助

The fastest way to get response from our developers is to submit issues, or add our wechat : 510570367

前端部署文档

前端项目环境构建及编译

Escheduler项目前端技术栈

Vue + es6 + Ans-ui + d3 + jsplumb + lodash

开发环境

Node

• node安装

Node包下载 (注意版本 8.9.4) <https://nodejs.org/download/release/v8.9.4/>

• 拉取前端项目到本地

项目git仓库地址 `git@git.analysys.cn:analysys_changsha/escheduler.git`

• 前端项目构建

用命令行模式 `cd` 进入 `escheduler` 项目目录并执行 `npm install` 拉取项目依赖包

如果 `npm install` 速度非常慢

可以转淘宝镜像命令行输入 `npm install -g cnpm --registry=https://registry.npm.taobao.org`

运行 `cnpm install`

！！！ 这里特别注意 项目如果在拉取依赖包的过程中报 "node-sass error" 错误, 请在执行完后再次执行以下命令

```
npm install node-sass --unsafe-perm //单独安装node-sass依赖
```

项目根目录创建 `.env` 为后缀名的文件并输入

```
# 前端代理的接口地址  
API_BASE = http://192.168.220.204:12345  
  
# 如果您需要用ip访问项目可以把 "#" 号去掉  
#DEV_HOST = 192.168.6.132
```

运行

- `npm start` 项目开发环境 (启动后访问地址 <http://localhost:8888/#/>)
- `npm run build` 项目打包 (打包后根目录会创建一个名为dist文件夹, 用于发布线上Nginx)

安装及配置

(1-1) Nginx安装

安装 `wget http://nginx.org/download/nginx-1.10.1.tar.gz`

Nginx的配置及运行需要pcre、 zlib等软件包的支持，因此应预先安装这些软件的开发包（devel），以便提供相应的库和头文件，确保Nginx的安装顺利完成。

```
[root@nginx ~]# service iptables stop
[root@nginx ~]# setenforce 0
[root@nginx ~]# mount /dev/cdrom /mnt/
[root@nginx ~]# vim /etc/yum.repos.d/yum.repo
[base]
name=RedHat Enterprise Linux Server
baseurl=file:///mnt/Packages
gpgcheck=0
[root@nginx ~]# yum -y install pcre-devel zlib-devel openssl-devel
```

(1-2) 创建运行用户、组

Nginx服务程序默认以nobody身份运行，建议为其创建专门的用户账号，以便更准确地控制其访问权限，增加灵活性、降低安全风险。如：创建一个名为nginx的用户，不建立宿主目录，也禁止登录到shell环境。

```
[root@nginx ~]# useradd -M -s /sbin/nologin escheduler
```

(1-3) 编译安装nginx

释放nginx源码包

```
[root@nginx ~]# tar xf nginx-1.6.2.tar.gz -C /usr/src/
```

编译前配置

```
[root@nginx ~]# cd /usr/src/nginx-1.6.2/
[root@nginx nginx-1.6.2]# ./configure --prefix=/usr/local/nginx --user=escheduler --group=escheduler --with-http_stub_status_module --with-http_ssl_module --with-http_flv_module --with-http_gzip_static_module
```

注：配置前可以参考 `./configure --help` 给出说明

```
--prefix      设定Nginx的安装目录
--user和--group 指定Nginx运行用户和组
--with-http_stub_status_module    启用http_stub_status_module模块以支持状态统计
--with-http_ssl_module           启用SSL模块
```

错误

```
[root@centos nginx-1.6.2]# ./configure --prefix=/usr/local/nginx --user=escheduler --group=escheduler --with-http_stub_status_module --with-http_ssl_module --with-http_flv_module --with-http_gzip_static_module
checking for OS
+ Linux 2.6.32-431.el6.i686 i686
checking for C compiler ... not found
./configure: error: C compiler cc is not found
```

解决方法

```
yum -y install gcc gcc-c++
```

编译 安装

```
[root@nginx nginx-1.6.2]# make && make install
```

为了使Nginx服务器的运行更加方便，可以为主程序nginx创建链接文件，以便管理员直接执行nginx命令就可以调用Nginx的主程序。

```
[root@nginx nginx-1.6.2]# ln -s /usr/local/nginx/sbin/nginx /usr/local/bin/
[root@nginx nginx-1.6.2]# ll /usr/local/bin/nginx
lrwxrwxrwx 1 root root 27 12-29 07:24 /usr/local/bin/nginx -> /usr/local/nginx/sbin/nginx
```

Nginx的运行控制 与Apache的主程序httpd类似， Nginx的主程序也提供了"-t"选项用来对配置文件进行检查，以便找出不当或错误的配置。配置文件nginx.conf默认位于安装目录/usr/local/nginx/conf/目录中。若要检查位于其他位置的配置文件，可使用"-c"选项来指定路径。

```
root@nginx conf]# nginx -t
nginx: the configuration file /usr/local/nginx/conf/nginx.conf syntax is ok
nginx: configuration file /usr/local/nginx/conf/nginx.conf test is successful
```

启动、停止Nginx 直接运行nginx即可启动Nginx服务器，这种方式将使用默认的配置文件，若要改用其他配置文件，需添加"-c 配置文件路径"选项来指定路径。需要注意的是，若服务器中已安装有httpd等其他WEB服务软件，应采取措施（修改端口，停用或卸载）避免冲突。

```
[root@nginx conf]# chown -R escheduler:escheduler /usr/local/nginx
/usr/local/nginx/conf/nginx.conf

[root@nginx conf]# netstat -anpt |grep :80
[root@nginx conf]# nginx
[root@nginx conf]# netstat -anpt |grep :80
tcp        0      0 0.0.0.0:80          0.0.0.0:*              LISTEN      6810/nginx: master
```

通过检查 Nginx程序的监听状态，或者在浏览器中访问此WEB服务（默认页面将显示"Welcome to nginx!"），可以确认Nginx服务是否正常运行。

```
[root@nginx ~]# yum -y install elinks
[root@nginx ~]# elinks --dump http://localhost
Welcome to nginx!
```

主程序Nginx支持标准的进程信号，通过kill或者killall命令传送

HUP	重载配置	等同于-1
QUIT	退出进程	等同于-3
KILL	杀死进程	

```
[root@nginx ~]# killall -s HUP nginx
[root@nginx ~]# killall -s QUIT nginx
[root@nginx ~]# netstat -anpt |grep :80
```

当Nginx进程运行时，PID号默认存放在logs/目录下的nginx.pid文件中，因此若改用kill命令，也可以根据nginx.pid文件中的PID号来进行控制。为了使Nginx服务的启动、停止、重载等操作更加方便，可以编写Nginx服务脚本，并使用chkconfig和service工具来进行管理，也更加符合RHEL系统的管理习惯。

```
[root@nginx ~]# vim /etc/init.d/nginx
```

脚本一

```
#!/bin/bash
# chkconfig: 2345 99 20
# description: Nginx Server Control Script
PROG="/usr/local/nginx/sbin/nginx"
PIDF="/usr/local/nginx/logs/nginx.pid"
case "$1" in
start)
    $PROG
;;
stop)
    kill -s QUIT $(cat $PIDF)
;;
restart)
    $0 stop
    $0 start
;;
reload)
    kill -s HUP $(cat $PIDF)
;;
*)
    echo "Usage: $0 (start|stop|restart|reload)"
    exit 1
esac
exit 0

[root@nginx ~]# chmod +x /etc/init.d/nginx
[root@nginx ~]# chkconfig --add nginx
[root@nginx ~]# chkconfig nginx on
[root@nginx ~]# chkconfig --list nginx
nginx           0:关闭    1:关闭    2:启用    3:启用    4:启用    5:启用    6:关闭
```

报错的话： /usr/local/nginx/sbin/nginx -c /usr/local/nginx/conf/nginx.conf

这样就可以通过nginx脚本来启动、停止、重启、重载Nginx服务器了。

(2-1) root安装

安装epel源 yum install epel-release -y

安装Nginx yum install nginx -y

命令

- 启用 systemctl enable nginx
- 重启 systemctl restart nginx
- 状态 systemctl status nginx

项目生产环境配置

创建静态页面存放目录

```
mkdir /data2_4T/escheduler_front/escheduler/server
```

配置文件地址

```
/etc/nginx/conf.d/default.conf
```

配置信息

```
server {
    listen      8888;# 访问端口
    server_name localhost;
    #charset koi8-r;
    #access_log  /var/log/nginx/host.access.log  main;
    location / {
        root   /data2_4T/escheduler_front/escheduler/server; # 静态文件目录
        index  index.html index.html;
    }
    location /escheduler {
        proxy_pass http://192.168.220.181:12345; # 接口地址
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header x_real_ipP $remote_addr;
        proxy_set_header remote_addr $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_http_version 1.1;
        proxy_connect_timeout 4s;
        proxy_read_timeout 30s;
        proxy_send_timeout 12s;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";
    }
    #error_page  404          /404.html;
    # redirect server error pages to the static page /50x.html
    #
    error_page  500 502 503 504  /50x.html;
    location = /50x.html {
        root   /usr/share/nginx/html;
    }
}
```

重启Nginx服务

```
systemctl restart nginx
```

前端项目发布

前端在开发环境（dev）中运行 `npm run build` 命令，生成打包文件（dist）包

再拷贝到服务器 `/data2_4T/escheduler_front/escheduler/server` （服务器静态页面存放目录）

访问地址 `http://localhost:8888/#/`

问题

1. 上传文件大小限制

编辑配置文件 `vi /etc/nginx/nginx.conf`

```
# 更改上传大小
client_max_body_size 1024m
```


前端开发文档

技术选型

```
Vue mvvm框架  
Es6 ECMAScript 6.0  
Ans-ui Analysys-ui  
D3 可视化库图表库  
Jsplumb 连线插件库  
Lodash 高性能的 JavaScript 实用工具库
```

项目目录结构

```
build 打包及开发环境项目的一些webpack配置  
node_modules 开发环境node依赖包  
src 项目所需文件  
  src => combo 项目第三方资源本地化 npm run combo 具体查看 build/combo.js  
  src => font 字体图标库可访问 https://www.iconfont.cn 进行添加 注意：字体库用自己的二次开发需要重新引入自己的库 src/sass/common/_font.scss  
  src => images 公共图片存放  
  src => js js/vue  
  src => lib 公司内部组件（公司组件库开源后可删掉）  
  src => sass sass文件 一个页面对应一个sass文件  
  src => view 页面文件 一个页面对应一个html文件
```

```
> 项目采用vue单页面应用(SPA)开发  
- 所有页面入口文件在 `src/js/conf/${对应页面文件名 => home}` 的 `index.js` 入口文件  
- 对应的sass文件则在 `src/sass/conf/${对应页面文件名 => home}/index.scss`  
- 对应的html文件则在 `src/view/${对应页面文件名 => home}/index.html`
```

公共模块及util src/js/module

```
components => 内部项目公共组件  
download => 下载组件  
echarts => 图表组件  
filter => 过滤器和vue管道  
i18n => 国际化  
io => io请求封装 基于axios  
mixin => vue mixin 公共部分 用于disabled操作
```

`permissions` => 权限操作

`util` => 工具

系统功能模块

首页 => `http://localhost:8888/#/home`

项目管理 => `http://localhost:8888/#/projects/list`

```
| 项目首页  
| 工作流  
- 工作流定义  
- 工作流实例  
- 任务实例
```

资源管理 => `http://localhost:8888/#/resource/file`

```
| 文件管理  
| UDF管理  
- 资源管理  
- 函数管理
```

数据源管理 => `http://localhost:8888/#/datasource/list`

安全中心 => `http://localhost:8888/#/security/tenant`

```
| 租户管理  
| 用户管理  
| 告警组管理  
- master  
- worker
```

用户中心 => `http://localhost:8888/#/user/account`

路由和状态管理

项目 `src/js/conf/home` 下分为

`pages` => 路由指向页面目录

```
路由地址对应的页面文件
```

`router` => 路由管理

```
vue的路由器，在每个页面的入口文件index.js 都会注册进来 具体操作: https://router.vuejs.org/zh/
```

`store` => 状态管理

```
每个路由对应的页面都有一个状态管理的文件 分为:
```

```
actions => mapActions => 详情: https://vuex.vuejs.org/zh/guide/actions.html
```

```
getters => mapGetters => 详情: https://vuex.vuejs.org/zh/guide/getters.html
```

```
index => 入口
```

```
mutations => mapMutations => 详情: https://vuex.vuejs.org/zh/guide/mutations.html  
state => mapState => 详情: https://vuex.vuejs.org/zh/guide/state.html  
具体操作: https://vuex.vuejs.org/zh/
```

规范

Vue规范

1.组件名

组件名为多个单词，并且用连接线（-）连接，避免与 HTML 标签冲突，并且结构更加清晰。

```
// 正例  
export default {  
  name: 'page-article-item'  
}
```

2.组件文件

src/js/module/components 项目内部公共组件书写文件夹名与文件名同名，公共组件内部所拆分的子组件与util工具都放置组件内部 `_source` 文件夹里。

```
└── components  
    ├── header  
    │   ├── header.vue  
    │   └── _source  
    │       └── nav.vue  
    │       └── util.js  
    ├── conditions  
    │   ├── conditions.vue  
    │   └── _source  
    │       └── serach.vue  
    └── util.js
```

3.Prop

定义 Prop 的时候应该始终以驼峰格式（camelCase）命名，在父组件赋值的时候使用连接线（-）。这里遵循每个语言的特性，因为在 HTML 标记中对大小写是不敏感的，使用连接线更加友好；而在 JavaScript 中更自然的是驼峰命名。

```
// Vue  
props: {  
  articleStatus: Boolean  
}  
// HTML  
<article-item :article-status="true"></article-item>
```

Prop 的定义应该尽量详细的指定其类型、默认值和验证。

示例：

```
props: {  
  attrM: Number,  
  attrA: {  
    type: String,  
    required: true
```

```

},
attrZ: {
  type: Object,
  // 数组/对象的默认值应该由一个工厂函数返回
  default: function () {
    return {
      msg: '成就你我'
    }
  }
},
attrE: {
  type: String,
  validator: function (v) {
    return !(['success', 'fail'].indexOf(v) === -1)
  }
}
}

```

4.v-for

在执行 v-for 遍历的时候，总是应该带上 key 值使更新 DOM 时渲染效率更高。

```

<ul>
  <li v-for="item in list" :key="item.id">
    {{ item.title }}
  </li>
</ul>

```

v-for 应该避免与 v-if 在同一个元素（例如：）上使用，因为 v-for 的优先级比 v-if 更高，为了避免无效计算和渲染，应该尽量将 v-if 放到容器的父元素之上。

```

<ul v-if="showList">
  <li v-for="item in list" :key="item.id">
    {{ item.title }}
  </li>
</ul>

```

5.v-if / v-else-if / v-else

若同一组 v-if 逻辑控制中的元素逻辑相同，Vue 为了更高效的元素切换，会复用相同的部分，例如：value。为了避免复用带来的不合理效果，应该在同种元素上加上 key 做标识。

```

<div v-if="hasData" key="mazey-data">
  <span>{{ mazeyData }}</span>
</div>
<div v-else key="mazey-none">
  <span>无数据</span>
</div>

```

6.指令缩写

为了统一规范始终使用指令缩写，使用 v-bind，v-on 并没有什么不好，这里仅为了统一规范。

```

<input :value="mazeyUser" @click="verifyUser">

```

7.单文件组件的顶级元素顺序

样式后续都是打包在一个文件里，所有在单个vue文件中定义的样式，在别的文件里同类名的样式也是会生效的所有在创建一个组件前都会有个顶级类名 注意：项目内已经增加了sass插件，单个vue文件里可以直接书写sass语法 为了统一和便于阅读，应该按 <template>、<script>、<style> 的顺序放置。

```

<template>
  <div class="test-model">
    test
  </div>
</template>
<script>
  export default {
    name: "test",
    data() {
      return {}
    },
    props: {},
    methods: {},
    watch: {},
    beforeCreate() {
    },
    created() {
    },
    beforeMount() {
    },
    mounted() {
    },
    beforeUpdate() {
    },
    updated() {
    },
    beforeDestroy() {
    },
    destroyed() {
    },
    computed: {},
    components: {},
  }
</script>

<style lang="scss" rel="stylesheet/scss">
  .test-model {
  }
</style>

```

JavaScript规范

1.var / let / const

建议不再使用 var，而使用 let / const，优先使用 const。任何一个变量的使用都要提前申明，除了 function 定义的函数可以随便放在任何位置。

2.引号

```

const foo = '后除'
const bar = `${foo}, 前端工程师`

```

3.函数

匿名函数统一使用箭头函数，多个参数/返回值时优先使用对象的结构赋值。

```

function getPersonInfo ({name, sex}) {
  // ...
  return {name, gender}
}

```

函数名统一使用驼峰命名，以大写字母开头申明的都是构造函数，使用小写字母开头的都是普通函数，也不该使用 new 操作符去操作普通函数。

4. 对象

```
const foo = {a: 0, b: 1}
const bar = JSON.parse(JSON.stringify(foo))

const foo = {a: 0, b: 1}
const bar = {...foo, c: 2}

const foo = {a: 3}
Object.assign(foo, {b: 4})

const myMap = new Map([])
for (let [key, value] of myMap.entries()) {
    // ...
}
```

5. 模块

统一使用 import / export 的方式管理项目的模块。

```
// lib.js
export default {}

// app.js
import app from './lib'
```

import 统一放在文件顶部。

如果模块只有一个输出值，使用 export default，否则不用。

HTML / CSS

1. 标签

在引用外部 CSS 或 JavaScript 时不写 type 属性。HTML5 默认 type 为 text/css 和 text/javascript 属性，所以没必要指定。

```
<link rel="stylesheet" href="//www.test.com/css/test.css">
<script src="//www.test.com/js/test.js"></script>
```

2. 命名

Class 和 ID 的命名应该语义化，通过看名字就知道是干嘛的；多个单词用连接线 - 连接。

```
// 正例
.test-header{
    font-size: 20px;
}
```

3. 属性缩写

CSS 属性尽量使用缩写，提高代码的效率和方便理解。

```
// 反例
border-width: 1px;
```

```
border-style: solid;  
border-color: #ccc;  
  
// 正例  
border: 1px solid #ccc;
```

4. 文档类型

应该总是使用 HTML5 标准。

```
<!DOCTYPE html>
```

5. 注释

应该给一个模块文件写一个区块注释。

```
/**  
 * @module mazey/api  
 * @author Mazey <mazey@mazey.net>  
 * @description test.  
 * */
```

接口

所有的接口都以 **Promise** 形式返回

注意非0都为错误走 catch

```
const test = () => {  
  return new Promise((resolve, reject) => {  
    resolve({  
      a:1  
    })  
  })  
}  
  
// 调用  
test.then(res => {  
  console.log(res)  
  // {a:1}  
})
```

正常返回

```
{  
  code:0,  
  data:{}  
  msg:'成功'  
}
```

错误返回

```
{  
  code:10000,  
  data:{}  
  msg:'失败'  
}
```

相关接口路径

dag 相关接口 `src/js/conf/home/store/dag/actions.js`

数据源中心 相关接口 `src/js/conf/home/store/datasource/actions.js`

项目管理 相关接口 `src/js/conf/home/store/projects/actions.js`

资源中心 相关接口 `src/js/conf/home/store/resource/actions.js`

安全中心 相关接口 `src/js/conf/home/store/security/actions.js`

用户中心 相关接口 `src/js/conf/home/store/user/actions.js`

扩展开发

1.增加节点

(1) 先将节点的icon小图标放置 `src/js/conf/home/pages/dag/img` 文件夹内，注意 `toolbar_${后台定义的节点的英文名称 例如:SHELL}.png` (2) 找到 `src/js/conf/home/pages/dag/_source/config.js` 里的 `tasksType` 对象，往里增加

```
'DEPENDENT': { // 后台定义节点类型英文名称用作key值
  desc: 'DEPENDENT', // tooltip desc
  color: '#2FBFD8' // 代表的颜色主要用于 tree和gantt 两张图
}
```

(3) 在 `src/js/conf/home/pages/dag/_source/formModel/tasks` 增加一个 `$(节点类型 (小写)) .vue` 文件，跟当前节点相关的组件内容都在这里写。属于节点组件内的必须拥有一个函数 `_verification()` 验证成功后讲当前组件的相关数据往父组件抛。

```
/**
 * 验证
 */
_verification () {
  // datasource 子组件验证
  if (!this.$refs.refDs._verifDatasource()) {
    return false
  }

  // 验证函数
  if (!this.method) {
    this.$message.warning(`#${i18n.$t('请输入方法')}`)
    return false
  }

  // localParams 子组件验证
  if (!this.$refs.refLocalParams._verifProp()) {
    return false
  }

  // 存储
  this.$emit('on-params', {
    type: this.type,
    datasource: this.datasource,
    method: this.method,
    localParams: this.localParams
  })
  return true
}
```

(4) 节点组件内部所用到公共的组件都在 `_source` 下，`common.js` 用与配置公共数据

2.增加状态类型

(1) 找到 `src/js/conf/home/pages/dag/_source/config.js` 里的 `tasksState` 对象，往里增加

```
'WAITTING_DEPEND': { //后端定义状态类型 前端用作key值
  id: 11, // 前端定义id 后续用作排序
  desc: `${i18n.$t('等待依赖')}`, // tooltip desc
  color: '#5101be', // 代表的颜色主要用于 tree和gantt 两张图
  icoUnicode: '', // 字体图标
  isSpin: false // 是否旋转 (需代码判断)
}
```

3.增加操作栏工具

(1) 找到 `src/js/conf/home/pages/dag/_source/config.js` 里的 `toolOper` 对象，往里增加

```
{
  code: 'pointer', // 工具标识
  icon: '', // 工具图标
  disable: disable, // 是否禁用
  desc: `${i18n.$t('拖动节点和选中项')}` // tooltip desc
}
```

(2) 工具类都以一个构造函数返回 `src/js/conf/home/pages/dag/_source/plugIn`

`downChart.js => dag` 图片下载处理

`dragZoom.js => 鼠标缩放效果处理`

`jsPlumbHandle.js => 拖拽线条处理`

`util.js => 属于 plugIn 工具类`

操作则在 `src/js/conf/home/pages/dag/_source/dag.js => toolbarEvent` 事件中处理。

3.增加一个路由页面

(1) 首先在路由管理增加一个路由地址 `src/js/conf/home/router/index.js`

```
{
  path: '/test', // 路由地址
  name: 'test', // 别名
  component: resolve => require(['../pages/test/index'], resolve), // 路由对应组件入口文件
  meta: {
    title: `${i18n.$t('test')} - EasyScheduler` // title 显示
  }
},
```

(2) 在 `src/js/conf/home/pages` 建一个 `test` 文件夹，在文件夹里建一个 `index.vue` 入口文件。

```
这样就可以直接访问 `http://localhost:8888/#/test`
```

4.增加预置邮箱

找到 `src/lib/localData/email.js` 启动和定时邮箱地址输入可以自动下拉匹配。

```
export default ["test@analysys.com.cn", "test1@analysys.com.cn", "test3@analysys.com.cn"]
```

5.权限管理及disabled状态处理

权限根据后端接口 `getUserInfo` 接口给出 `userType: "ADMIN_USER/GENERAL_USER"` 权限控制页面操作按钮是否 `disabled`

具体操作: `src/js/module/permissions/index.js`

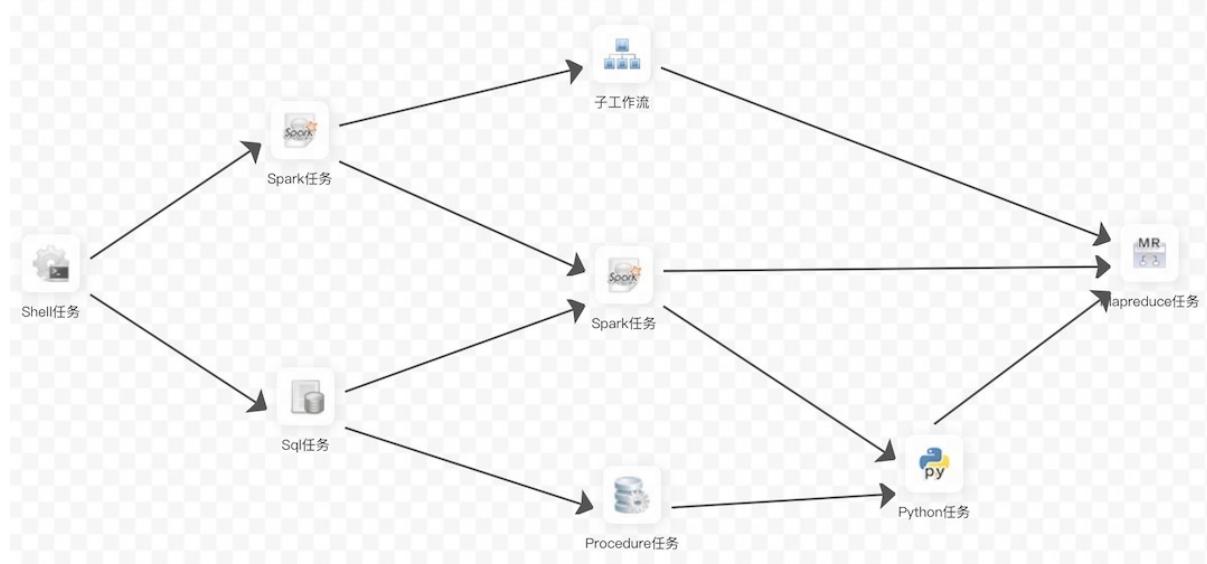
`disabled`处理: `src/js/module/mixin/disabledState.js`

调度系统架构设计

在对系统架构说明之前，我们先来认识一下调度系统常用的名词

1. 名词解释

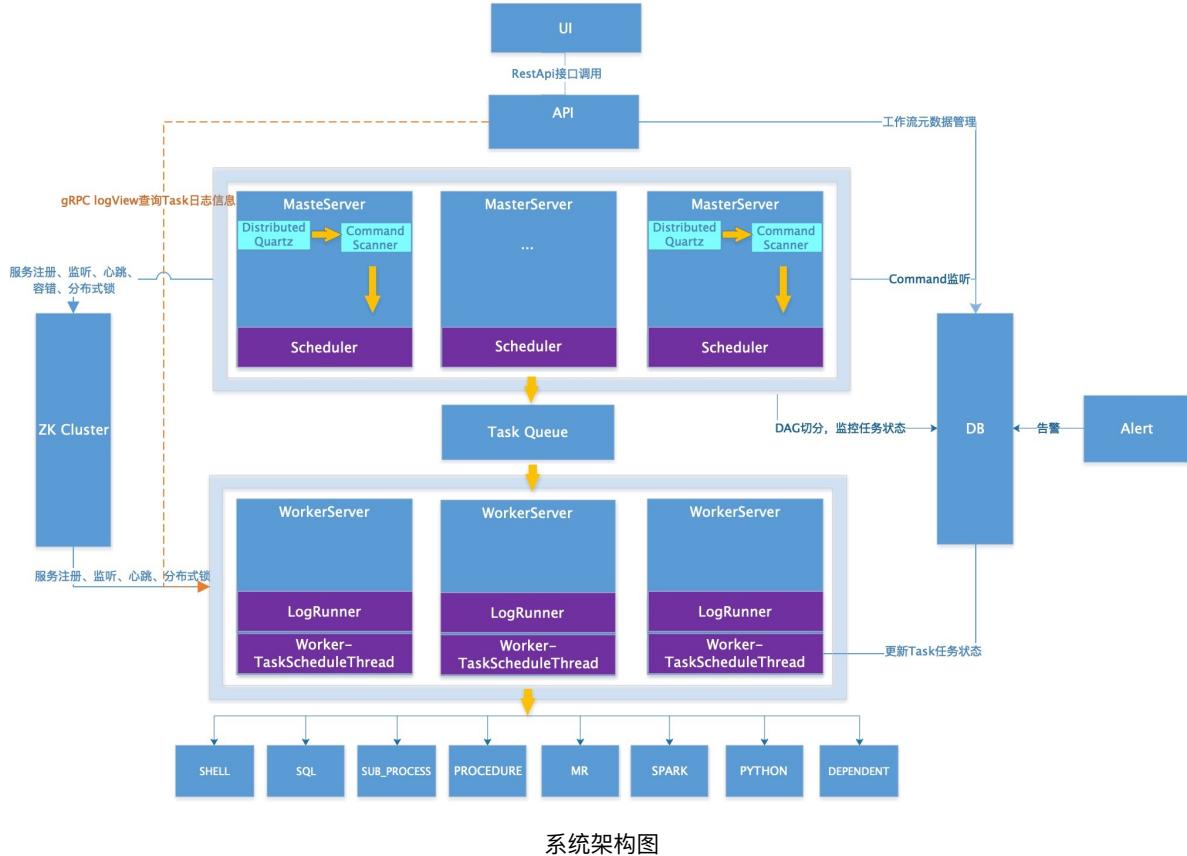
DAG: 全称Directed Acyclic Graph，简称DAG。工作流中的Task任务以有向无环图的形式组装起来，从入度为零的节点进行拓扑遍历，直到无后继节点为止。举例如下图：



dag示例

流程定义: 通过拖拽任务节点并建立任务节点的关联所形成的可视化**DAG** **流程实例**: 流程实例是流程定义的实例化，可以通过手动启动或定时调度生成 **任务实例**: 任务实例是流程定义中任务节点的实例化，标识着具体的任务执行状态 **任务类型**: 目前支持有SHELL、SQL、SUB_PROCESS、PROCEDURE、MR、SPARK、PYTHON、DEPENDENT，同时计划支持动态插件扩展，注意：其中子 **SUB_PROCESS** 也是一个单独的流程定义，是可以单独启动执行的 **调度方式**: 系统支持基于cron表达式的定时调度和手动调度。命令类型支持：启动工作流、从当前节点开始执行、恢复被容错的工作流、恢复暂停流程、从失败节点开始执行、补数、调度、重跑、暂停、停止、恢复等待线程。其中 **恢复被容错的工作流** 和 **恢复等待线程** 两种命令类型是由调度内部控制使用，外部无法调用 **定时调度**: 系统采用 **quartz** 分布式调度器，并同时支持cron表达式可视化的生成 **依赖**: 系统不单单支持 **DAG** 简单的前驱和后继节点之间的依赖，同时还提供**任务依赖**节点，支持**流程间的自定义任务依赖** **优先级**: 支持流程实例和任务实例的优先级，如果流程实例和任务实例的优先级不设置，则默认是先进先出 **邮件告警**: 支持 **SQL任务** 查询结果邮件发送，流程实例运行结果邮件告警及容错告警通知 **失败策略**: 对于并行运行的任务，如果有任务失败，提供两种失败策略处理方式，**继续**是指不管并行运行任务的状态，直到流程失败结束。**结束**是指一旦发现失败任务，则同时Kill掉正在运行的并行任务，流程失败结束 **补数**: 补历史数据，支持**区间并行和串行**两种补数方式

2. 系统架构 ## 2.1 系统架构图



系统架构图

2.2 架构说明

• MasterServer

MasterServer采用分布式无中心设计理念，MasterServer主要负责 DAG 任务切分、任务提交监控，并同时监听其它MasterServer和WorkerServer的健康状态。MasterServer服务启动时向Zookeeper注册临时节点，通过监听Zookeeper临时节点变化来进行容错处理。

该服务内主要包含：

- **Distributed Quartz**分布式调度组件，主要负责定时任务的启停操作，当quartz调起任务后，Master内部会有线程池具体负责处理任务的后续操作
- **MasterSchedulerThread**是一个扫描线程，定时扫描数据库中的 **command** 表，根据不同的命令类型进行不同的业务操作
- **MasterExecThread**主要是负责DAG任务切分、任务提交监控、各种不同命令类型的逻辑处理
- **MasterTaskExecThread**主要负责任务的持久化

• WorkerServer

WorkerServer也采用分布式无中心设计理念，WorkerServer主要负责任务的执行和提供日志服务。WorkerServer服务启动时向Zookeeper注册临时节点，并维持心跳。

该服务包含：

- **FetchTaskThread**主要负责不断从**Task Queue**中领取任务，并根据不同任务类型调用**TaskScheduleThread**对应执行器。
- **LoggerServer**是一个RPC服务，提供日志分片查看、刷新和下载等功能

- **ZooKeeper**

ZooKeeper服务，系统中的MasterServer和WorkerServer节点都通过ZooKeeper来进行集群管理和容错。另外系统还基于ZooKeeper进行事件监听和分布式锁。我们也曾经基于Redis实现过队列，不过我们希望EasyScheduler依赖到的组件尽量地少，所以最后还是去掉了Redis实现。

- **Task Queue**

提供任务队列的操作，目前队列也是基于Zookeeper来实现。由于队列中存的信息较少，不必担心队列里数据过多的情况，实际上我们压测过百万级数据存队列，对系统稳定性和性能没影响。

- **Alert**

提供告警相关接口，接口主要包括告警两种类型的告警数据的存储、查询和通知功能。其中通知功能又有邮件通知和SNMP(暂未实现)两种。

- **API**

API接口层，主要负责处理前端UI层的请求。该服务统一提供RESTful api向外部提供请求服务。接口包括工作流的创建、定义、查询、修改、发布、下线、手工启动、停止、暂停、恢复、从该节点开始执行等等。

- **UI**

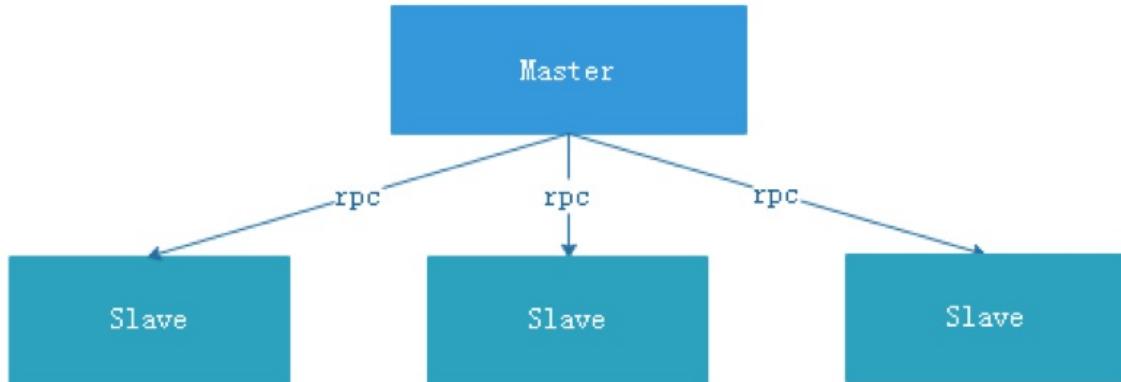
系统的前端页面，提供系统的各种可视化操作界面，详见[使用手册](#)部分。

2.3 架构设计思想

一、去中心化vs中心化

中心化思想

中心化的设计理念比较简单，分布式集群中的节点按照角色分工，大体上分为两种角色：



- Master的角色主要负责任务分发并监督Slave的健康状态，可以动态的将任务均衡到Slave上，以致Slave节点不至于“忙死”或“闲死”的状态。
- Worker的角色主要负责任务的执行工作并维护和Master的心跳，以便Master可以分配任务给Slave。
中心化思想设计存在的问题：
- 一旦Master出现了问题，则群龙无首，整个集群就会崩溃。为了解决这个问题，大多数Master/Slave架构模式都采用了主备Master的设计方案，可以是热备或者冷备，也可以是自动切换或手动切换，而且越来越多的新系统都开始具备自动选举切换Master的能力，以提升系统的可用性。
- 另外一个问题是如果Scheduler在Master上，虽然可以支持一个DAG中不同的任务运行在不同的机器上，但是会产生Master的过负载。如果Scheduler在Slave上，则一个DAG中所有的任务都只能在某一台机器上进行作业提交，则并行任务比较多的时候，Slave的压力可能会比较大。 ##### 去中心化

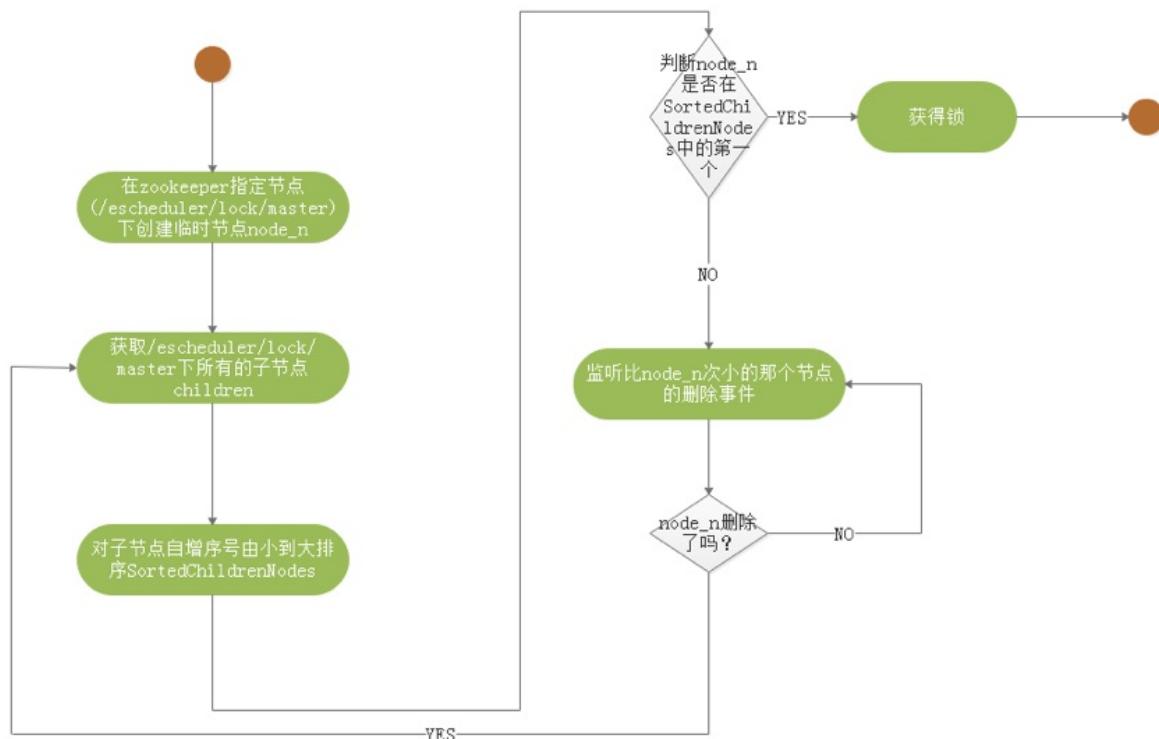
- 在去中心化设计里，通常没有Master/Slave的概念，所有的角色都是一样的，地位是平等的，全球互联网就是一个典型的去中心化的分布式系统，联网的任意节点设备down机，都只会影响很小范围的功能。
- 去中心化设计的核心设计

在于整个分布式系统中不存在一个区别于其他节点的“管理者”，因此不存在单点故障问题。但由于不存在“管理者”节点所以每个节点都需要跟其他节点通信才得到必须要的机器信息，而分布式系统通信的不可靠性，则大大增加了上述功能的实现难度。

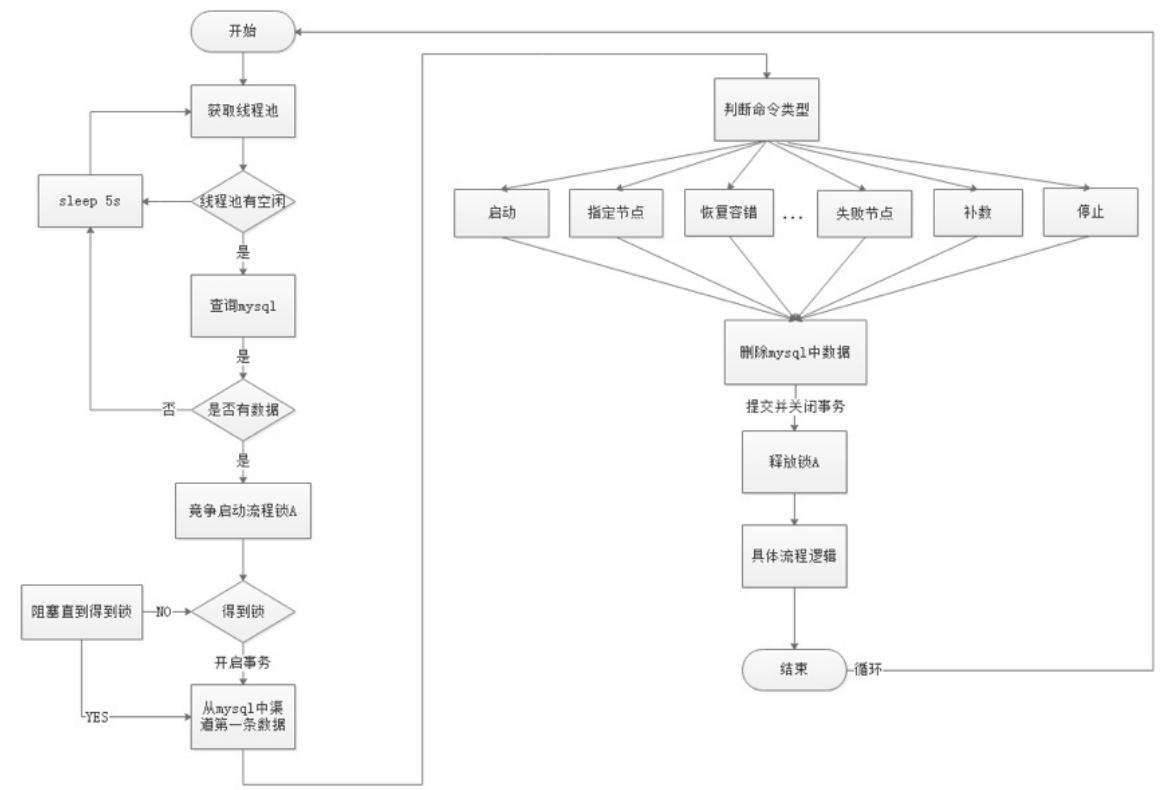
- 实际上，真正去中心化的分布式系统并不多见。反而动态中心化分布式系统正在不断涌出。在这种架构下，集群中的管理者是被动态选择出来的，而不是预置的，并且集群在发生故障的时候，集群的节点会自发的举行“会议”来选举新的“管理者”去主持工作。最典型的案例就是ZooKeeper及Go语言实现的Etcd。
- EasyScheduler的去中心化是Master/Worker注册到Zookeeper中，实现Master集群和Worker集群无中心，并使用Zookeeper分布式锁来选举其中的一台Master或Worker为“管理者”来执行任务。

二、分布式锁实践 EasyScheduler使用ZooKeeper分布式锁来实现同一时刻只有一台Master执行Scheduler，或者只有一台Worker执行任务的提交。

1. 获取分布式锁的核心流程算法如下

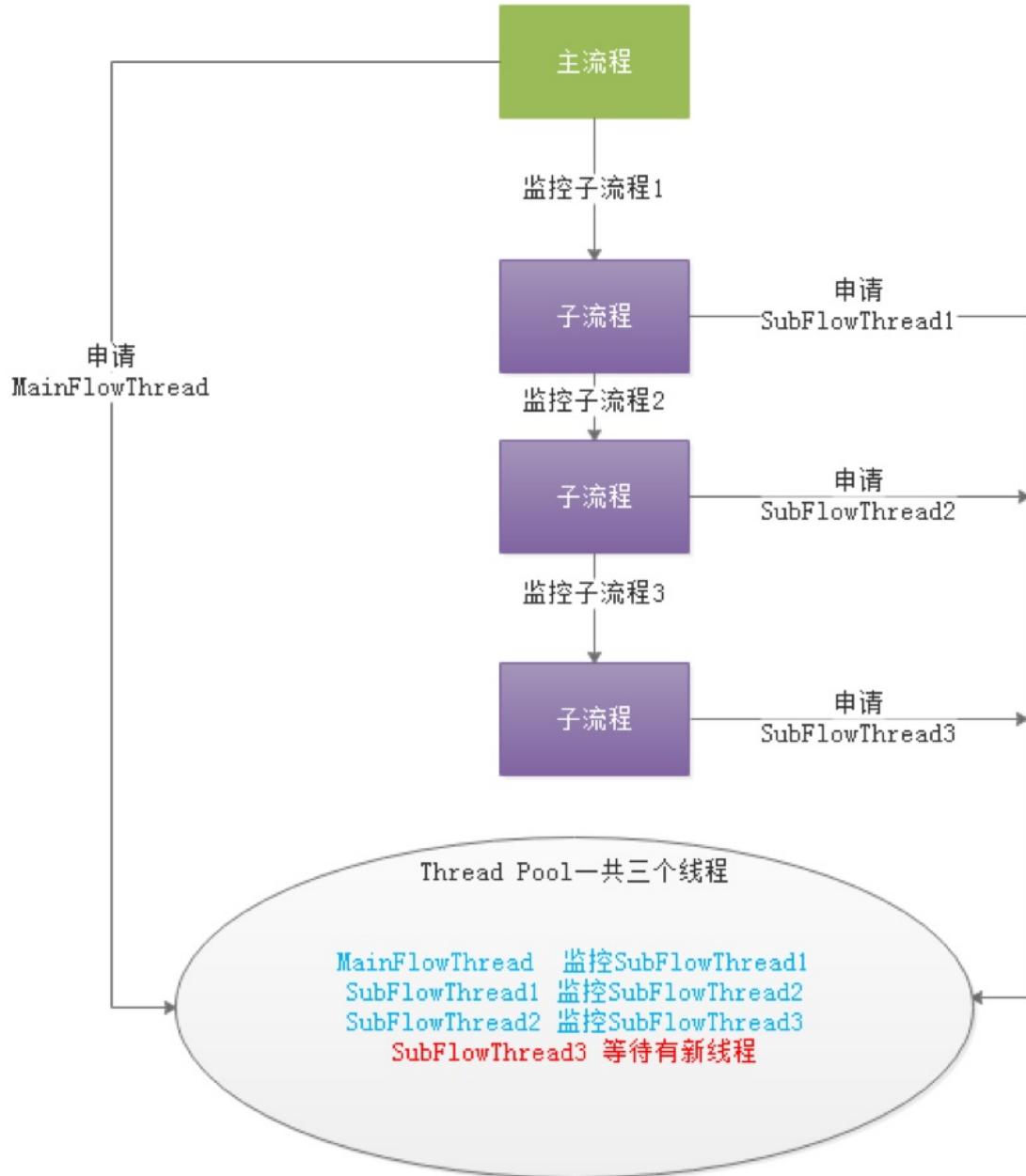


1. EasyScheduler中Scheduler线程分布式锁实现流程图：



三、线程不足循环等待问题

- 如果一个DAG中没有子流程，则如果Command中的数据条数大于线程池设置的阈值，则直接流程等待或失败。
- 如果一个大的DAG中嵌套了很多子流程，如下图则会产生“死等”状态：



上图中MainFlowThread等待SubFlowThread1结束，SubFlowThread1等待SubFlowThread2结束，SubFlowThread2等待SubFlowThread3结束，而SubFlowThread3等待线程池有新线程，则整个DAG流程不能结束，从而其中的线程也不能释放。这样就形成了子父流程循环等待的状态。此时除非启动新的Master来增加线程来打破这样的“僵局”，否则调度集群将不能再使用。

对于启动新Master来打破僵局，似乎有点差强人意，于是我们提出了以下三种方案来降低这种风险：

1. 计算所有Master的线程总和，然后对每一个DAG需要计算其需要的线程数，也就是在DAG流程执行之前做预计算。因为是多Master线程池，所以总线程数不太可能实时获取。
2. 对单Master线程池进行判断，如果线程池已经满了，则让线程直接失败。
3. 增加一种资源不足的Command类型，如果线程池不足，则将主线程挂起。这样线程池就有了新的线程，可以让资源不足挂起的流程重新唤醒执行。

注意：Master Scheduler线程在获取Command的时候是FIFO的方式执行的。

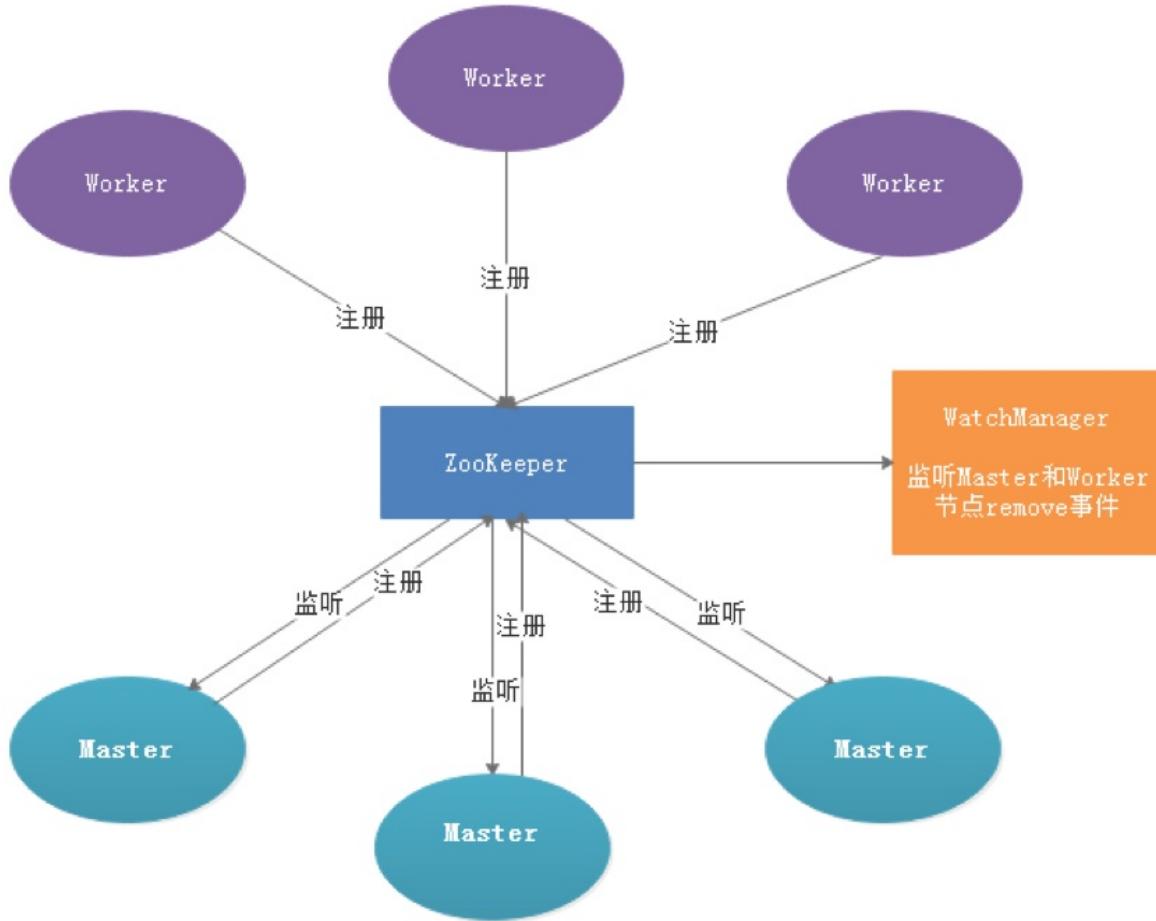
于是我们选择了第三种方式来解决线程不足的问题。

四、容错设计

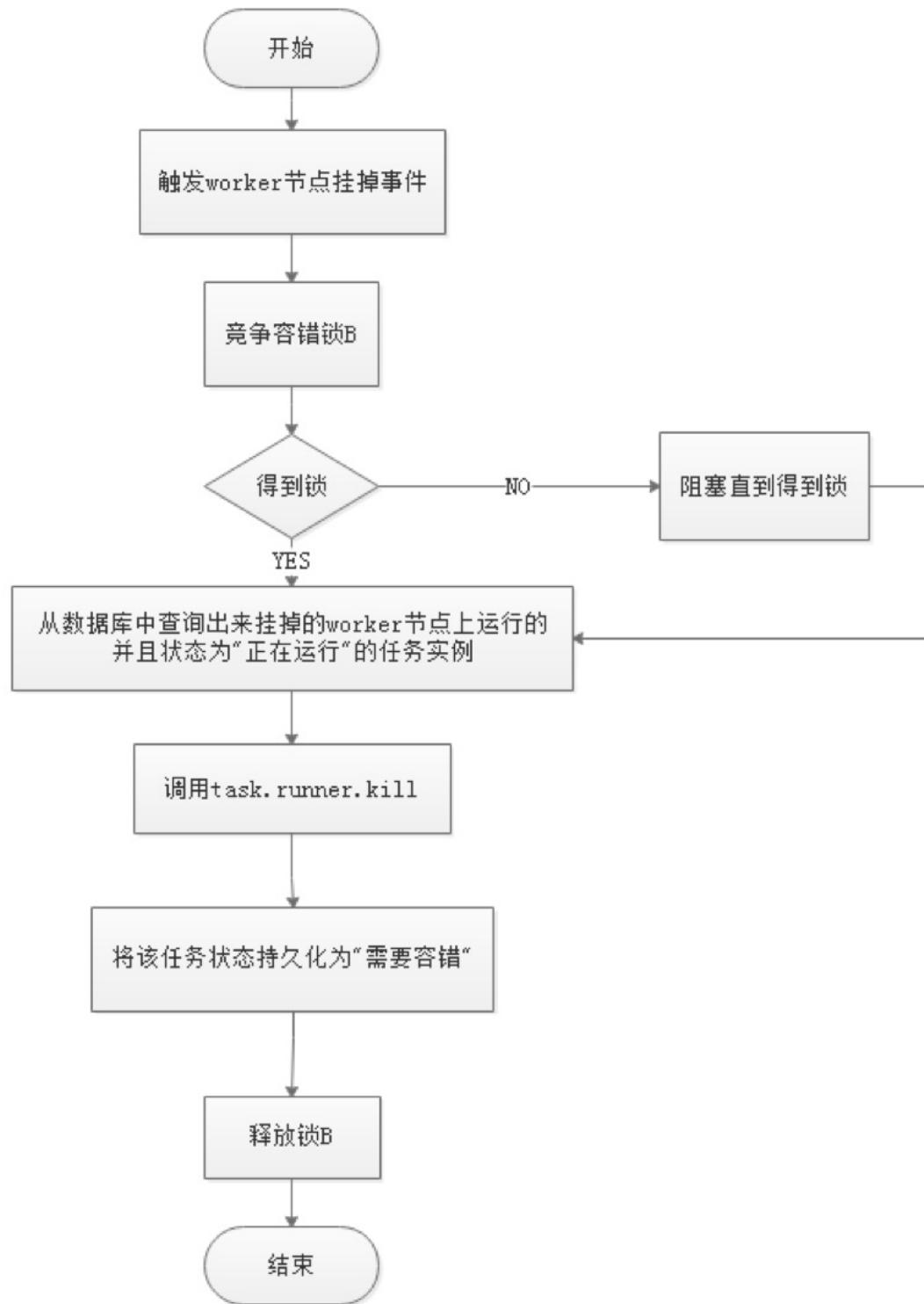
容错分为服务宕机容错和任务重试，服务宕机容错又分为Master容错和Worker容错两种情况

1. 宕机容错

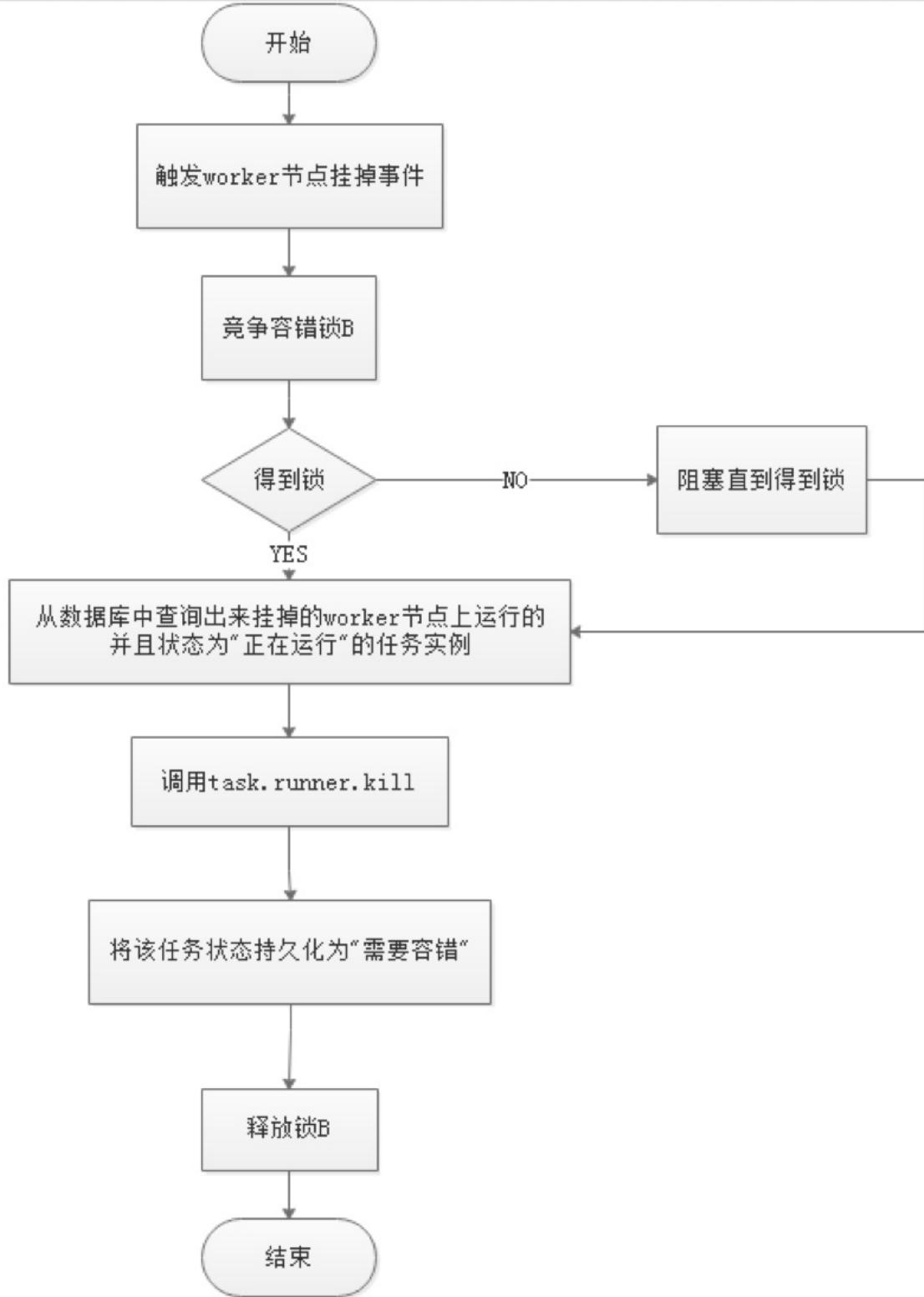
服务容错设计依赖于ZooKeeper的Watcher机制，实现原理如图：



其中Master监控其他Master和Worker的目录，如果监听到remove事件，则会根据具体的业务逻辑进行流程实例容错或者任务实例容错。 - Master容错流程图：



ZooKeeper Master容错完成之后则重新由EasyScheduler中Scheduler线程调度，遍历 DAG 找到“正在运行”和“提交成功”的任务，对“正在运行”的任务监控其任务实例的状态，对“提交成功”的任务需要判断Task Queue中是否存在，如果存在则同样监控任务实例的状态，如果不存在则重新提交任务实例。 - Worker容错流程图：



Master Scheduler线程一旦发现任务实例为“需要容错”状态，则接管任务并进行重新提交。注意：由于“网络抖动”可能会使得节点短时间内失去和ZooKeeper的心跳，从而发生节点的remove事件。对于这种情况，我们使用最简单的方式，那就是节点一旦和ZooKeeper发生超时连接，则直接将Master或Worker服务停掉。##### 2.任务失败重试 这里首先要区分任务失败重试、流程失败恢复、流程失败重跑的概念： - 任务失败重试是任务级别的，是调度系统自动进行的，比如一个Shell任务设置重试次数为3次，那么在Shell任务运行失败后会自己再最多尝试运行3次 - 流程失败恢复是流程级别的，是手动进行的，恢复是从只能**从失败的节点开始执行**或**从当前节点开始执行** - 流程失败重跑也是流程级别的，是手动进行的，重跑是从开始节点进行 接下来说正题，我们将工作流中的任务节点分了两种类型。 - 一种是

业务节点，这种节点都对应一个实际的脚本或者处理语句，比如Shell节点，MR节点、Spark节点、依赖节点等。 - 还有一种是逻辑节点，这种节点不做实际的脚本或语句处理，只是整个流程流转的逻辑处理，比如子流程节等。 每一个**业务节点**都可以配置失败重试的次数，当该任务节点失败，会自动重试，直到成功或者超过配置的重试次数。**逻辑节点**不支持失败重试。但是逻辑节点里的任务支持重试。如果工作流中有任务失败达到最大重试次数，工作流就会失败停止，失败的工作流可以手动进行重跑操作或者流程恢复操作 ##### 五、任务优先级设计 在早期调度设计中，如果没有优先级设计，采用公平调度设计的话，会遇到先行提交的任务可能会和后继提交的任务同时完成的情况，而不能做到设置流程或者任务的优先级，因此我们对此进行了重新设计，目前我们设计如下： - 按照**不同流程实例优先级**优先于**同一个流程实例优先级**优先于**同一流程内任务优先级**优先于**同一流程内任务**提交顺序依次从高到低进行任务处理。 - 具体实现是根据任务实例的json解析优先级，然后把**流程实例优先级_流程实例id_任务优先级_任务id**信息保存在ZooKeeper任务队列中，当从任务队列获取的时候，通过字符串比较即可得出最需要优先执行的任务 - 其中流程定义的优先级是考虑到有些流程需要先于其他流程进行处理，这个可以在流程启动或者定时启动时配置，共有5级，依次为HIGHEST、HIGH、MEDIUM、LOW、LOWEST。如下图

启动前请先设置参数

失败策略

继续
 结束

通知策略

都不发
▼

流程优先级

↑ MEDIUM
▼

通知组

↑ HIGHEST
↑ HIGH
↑ MEDIUM
↓ LOW
↓ LOWEST

▼

收件人

抄送人

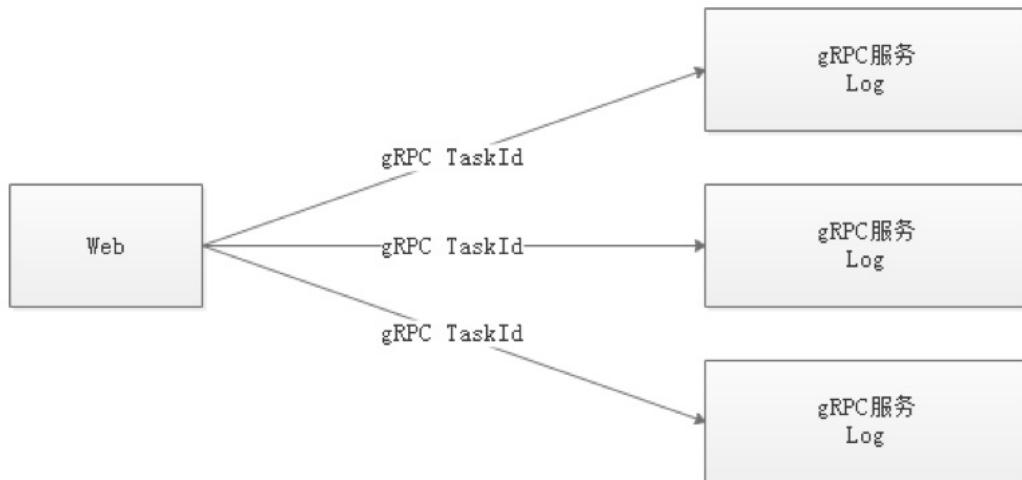
补数

```
- 任务的优先级也分为5级，依次为HIGHEST、HIGH、MEDIUM、LOW、LOWEST。如下图
<p align="center">

</p>
```

六、Logback和gRPC实现日志访问

- 由于Web(UI)和Worker不一定在同一台机器上，所以查看日志不能像查询本地文件那样。有两种方案：
 - 将日志放到ES搜索引擎上
 - 通过gRPC通信获取远程日志信息
- 介于考虑到尽可能的EasyScheduler的轻量级性，所以选择了gRPC实现远程访问日志信息。



- 我们使用自定义Logback的FileAppender和Filter功能，实现每个任务实例生成一个日志文件。
- FileAppender主要实现如下：

```

/**
 * task log appender
 */
public class TaskLogAppender extends FileAppender<ILoggingEvent> {

    ...

    @Override
    protected void append(ILoggingEvent event) {

        if (currentlyActiveFile == null){
            currentlyActiveFile = getFile();
        }
        String activeFile = currentlyActiveFile;
        // thread name: taskThreadName-processDefineId_processInstanceId_taskInstanceId
        String threadName = event.getThreadName();
        String[] threadNameArr = threadName.split("-");
        // logId = processDefineId_processInstanceId_taskInstanceId
        String logId = threadNameArr[1];
        ...
        super.subAppend(event);
    }
}
  
```

以/流程定义id/流程实例id/任务实例id.log的形式生成日志

- 过滤匹配以TaskLogInfo开始的线程名称：
- TaskLogFilter实现如下：

```
/**
```

```
* task log filter
*/
public class TaskLogFilter extends Filter<ILoggingEvent > {

    @Override
    public FilterReply decide(ILoggingEvent event) {
        if (event.getThreadName().startsWith("TaskLogInfo-")){
            return FilterReply.ACCEPT;
        }
        return FilterReply.DENY;
    }
}
```

总结

本文从调度出发，初步介绍了大数据分布式工作流调度系统--EasyScheduler的架构原理及实现思路。后续会补充

部署文档

基础软件安装

- [mysql](#) (5.5+) : 必装
- [zookeeper](#)(3.4.6) : 必装
- [hadoop](#)(2.7.3) : 选装, 资源上传, MR任务提交需要安装
- [hive](#)(1.2.1) : 选装, hive任务提交需要安装
- [spark](#)(1.x,2.x) : 选装, spark任务提交需要安装
- [postgresql](#)(8.2.15+) : 选装, postgresql sql任务和postgresql 存储过程需要安装

项目编译

- 执行编译命令:

```
mvn -U clean package assembly:assembly -Dmaven.test.skip=true
```

- 查看目录

正常编译完后, 会在当前目录生成 target/escheduler-{version}/

```
bin  
conf  
lib  
script  
sql  
install.sh
```

- 说明

```
bin : 基础服务启动脚本  
conf : 项目配置文件  
lib : 项目依赖jar包, 包括各个模块jar和第三方jar  
script : 集群启动、停止和服务监控启停脚本  
sql : 项目依赖sql文件  
install.sh : 一键部署脚本
```

数据库初始化

- 创建db和账号

```
mysql -h {host} -u {user} -p{password}
mysql> CREATE DATABASE escheduler DEFAULT CHARACTER SET utf8 DEFAULT COLLATE utf8_general_ci;
mysql> GRANT ALL PRIVILEGES ON escheduler.* TO '{user}'@'%' IDENTIFIED BY '{password}';
mysql> GRANT ALL PRIVILEGES ON escheduler.* TO '{user}'@'localhost' IDENTIFIED BY '{password}';
mysql> flush privileges;
```

- 创建表

说明: 在 target/escheduler-{version}/sql/escheduler.sql和quartz.sql

```
mysql -h {host} -u {user} -p{password} -D {db} < escheduler.sql
mysql -h {host} -u {user} -p{password} -D {db} < quartz.sql
```

创建部署用户

因为escheduler worker 都是以 sudo -u {linux-user} 方式来执行作业，所以部署用户需要有 sudo 权限，而且是免密的。

```
vi /etc/sudoers

# 部署用户是 escheduler 账号
escheduler    ALL=(ALL)      NOPASSWD: NOPASSWD: ALL

# 并且需要注释掉 Default requiretty 一行
#Default requiretty
```

配置文件说明

说明：配置文件位于 target/escheduler-{version}/conf 下面

escheduler-alert

配置邮件告警信息

- alert.properties

```
#alert type is EMAIL/SMS
alert.type=EMAIL

# mail server configuration
mail.protocol=SMTP
mail.server.host=smtp.exmail.qq.com
mail.server.port=25
mail.sender=xxxxxxx
mail.passwd=xxxxxxx

# xls file path,need create if not exist
xls.file.path=/opt/xls
```

escheduler-common

通用配置文件配置，队列选择及地址配置，通用文件目录配置

- common/common.properties

```
#task queue implementation, default "zookeeper"
escheduler.queue.impl=zookeeper

# user data directory path, self configuration, please make sure the directory exists and have read write permissions
data.basedir.path=/tmp/escheduler

# directory path for user data download. self configuration, please make sure the directory exists and have read write permissions
data.download.basedir.path=/tmp/escheduler/download
```

```

# process execute directory. self configuration, please make sure the directory exists and have read write permissions
process.exec.basepath=/tmp/escheduler/exec

# data base dir, resource file will store to this hadoop hdfs path, self configuration, please make sure the directory exists on hdfs and have read write permissions. "/escheduler" is recommended
data.store2hdfs.basepath=/escheduler

# whether hdfs starts
hdfs.startup.state=true

# system env path. self configuration, please make sure the directory and file exists and have read write execute permissions
escheduler.env.path=/opt/.escheduler_env.sh
escheduler.env.py=/opt/escheduler_env.py

#resource.view.suffixs
resource.view.suffixs=txt,log,sh,conf,cfg,py,java,sql,hql,xml

# is development state? default "false"
development.state=false

```

SHELL任务 环境变量配置

说明：配置文件位于 target/escheduler-{version}/conf/env 下面

.escheduler_env.sh

```

export HADOOP_HOME=/opt/soft/hadoop
export HADOOP_CONF_DIR=/opt/soft/hadoop/etc/hadoop
export SPARK_HOME1=/opt/soft/spark1
export SPARK_HOME2=/opt/soft/spark2
export PYTHON_HOME=/opt/soft/python
export JAVA_HOME=/opt/soft/java
export HIVE_HOME=/opt/soft/hive

export PATH=$HADOOP_HOME/bin:$SPARK_HOME1/bin:$SPARK_HOME2/bin:$PYTHON_HOME/bin:$JAVA_HOME/bin:$HIVE_HOME/bin:$PATH

```

Python任务 环境变量配置

说明：配置文件位于 target/escheduler-{version}/conf/env 下面

escheduler_env.py

```

import os

HADOOP_HOME="/opt/soft/hadoop"
SPARK_HOME1="/opt/soft/spark1"
SPARK_HOME2="/opt/soft/spark2"
PYTHON_HOME="/opt/soft/python"
JAVA_HOME="/opt/soft/java"
HIVE_HOME="/opt/soft/hive"
PATH=os.environ['PATH']
PATH="%s/bin:%s/bin:%s/bin:%s/bin:%s/bin:%s%"%(HIVE_HOME,HADOOP_HOME,SPARK_HOME1,SPARK_HOME2,JAVA_HOME,PYTHON_HOME,PATH)

os.putenv('PATH', '%s'%PATH)

```

hadoop 配置文件

- common/hadoop/hadoop.properties

```
# ha or single namenode, If namenode ha needs to copy core-site.xml and hdfs-site.xml to the conf directory
fs.defaultFS=hdfs://mycluster:8020

#resourcemanager ha note this need ips , this empty if single
yarn.resourcemanager.ha.rm.ids=192.168.xx.xx,192.168.xx.xx

# If it is a single resourcemanager, you only need to configure one host name. If it is resourcemanager HA, the
# default configuration is fine
yarn.application.status.address=http://ark1:8088/ws/v1/cluster/apps/%s
```

定时器配置文件

- quartz.properties

```
#=====
# Configure Main Scheduler Properties
#=====

org.quartz.scheduler.instanceName = EasyScheduler
org.quartz.scheduler.instanceId = AUTO
org.quartz.scheduler.makeSchedulerThreadDaemon = true
org.quartz.jobStore.useProperties = false

#=====
# Configure ThreadPool
#=====

org.quartz.threadPool.class = org.quartz.impl.SimpleThreadPool
org.quartz.threadPool.makeThreadsDaemons = true
org.quartz.threadPool.threadCount = 25
org.quartz.threadPool.threadPriority = 5

#=====
# Configure JobStore
#=====

org.quartz.jobStore.class = org.quartz.impl.jdbcjobstore.JobStoreTX
org.quartz.jobStore.driverDelegateClass = org.quartz.impl.jdbcjobstore.StdJDBCDelegate
org.quartz.jobStore.tablePrefix = QRTZ_
org.quartz.jobStore.isClustered = true
org.quartz.jobStore.misfireThreshold = 60000
org.quartz.jobStore.clusterCheckinInterval = 5000
org.quartz.jobStore.dataSource = myDs

#=====
# Configure Datasources
#=====

org.quartz.dataSource.myDs.driver = com.mysql.jdbc.Driver
org.quartz.dataSource.myDs.URL = jdbc:mysql://192.168.xx.xx:3306/escheduler?characterEncoding=utf8&useSSL=false
org.quartz.dataSource.myDs.user = xx
org.quartz.dataSource.myDs.password = xx
org.quartz.dataSource.myDs.maxConnections = 10
org.quartz.dataSource.myDs.validationQuery = select 1
```

zookeeper 配置文件

- zookeeper.properties

```
#zookeeper cluster
zookeeper.quorum=192.168.xx.xx:2181,192.168.xx.xx:2181,192.168.xx.xx:2181

#escheduler root directory
zookeeper.escheduler.root=/escheduler
```

```

#zookeeper server directory
zookeeper.escheduler.dead.servers=/escheduler/dead-servers
zookeeper.escheduler.masters=/escheduler/masters
zookeeper.escheduler.workers=/escheduler/workers

#zookeeper lock directory
zookeeper.escheduler.lock.masters=/escheduler/lock/masters
zookeeper.escheduler.lock.workers=/escheduler/lock/workers

#escheduler failover directory
zookeeper.escheduler.lock.masters.failover=/escheduler/lock/failover/masters
zookeeper.escheduler.lock.workers.failover=/escheduler/lock/failover/workers

#escheduler failover directory
zookeeper.session.timeout=300
zookeeper.connection.timeout=300
zookeeper.retry.sleep=1000
zookeeper.retry.maxtime=5

```

escheduler-dao

dao数据源配置

- dao/data_source.properties

```

# base spring data source configuration
spring.datasource.type=com.alibaba.druid.pool.DruidDataSource
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://192.168.xx.xx:3306/escheduler?characterEncoding=UTF-8
spring.datasource.username=xx
spring.datasource.password=xx

# connection configuration
spring.datasource.initialSize=5
# min connection number
spring.datasource.minIdle=5
# max connection number
spring.datasource.maxActive=50

# max wait time for get a connection in milliseconds. if configuring maxWait, fair locks are enabled by default
# and concurrency efficiency decreases.
# If necessary, unfair locks can be used by configuring the useUnfairLock attribute to true.
spring.datasource.maxWait=60000

# milliseconds for check to close free connections
spring.datasource.timeBetweenEvictionRunsMillis=60000

# the Destroy thread detects the connection interval and closes the physical connection in milliseconds if the
# connection idle time is greater than or equal to minEvictableIdleTimeMillis.
spring.datasource.timeBetweenConnectErrorMillis=60000

# the longest time a connection remains idle without being evicted, in milliseconds
spring.datasource.minEvictableIdleTimeMillis=300000

#the SQL used to check whether the connection is valid requires a query statement. If validation Query is null,
# testOnBorrow, testOnReturn, and testWhileIdle will not work.
spring.datasource.validationQuery=SELECT 1
#check whether the connection is valid for timeout, in seconds
spring.datasource.validationQueryTimeout=3

# when applying for a connection, if it is detected that the connection is idle longer than time Between Evicti
on Runs Millis,
# validation Query is performed to check whether the connection is valid
spring.datasource.testWhileIdle=true

```

```
#execute validation to check if the connection is valid when applying for a connection
spring.datasource.testOnBorrow=true
#execute validation to check if the connection is valid when the connection is returned
spring.datasource.testOnReturn=false
spring.datasource.defaultAutoCommit=true
spring.datasource.keepAlive=true

# open PSCache, specify count PSCache for every connection
spring.datasource.poolPreparedStatements=true
spring.datasource.maxPoolPreparedStatementPerConnectionSize=20
```

escheduler-server

master配置文件

- master.properties

```
# master execute thread num
master.exec.threads=100

# master execute task number in parallel
master.exec.task.number=20

# master heartbeat interval
master.heartbeat.interval=10

# master commit task retry times
master.task.commit.retryTimes=5

# master commit task interval
master.task.commit.interval=100

# only less than cpu avg load, master server can work. default value : the number of cpu cores * 2
master.max.cpupload.avg=10

# only larger than reserved memory, master server can work. default value : physical memory * 1/10, unit is G.
master.reserved.memory=1
```

worker配置文件

- worker.properties

```
# worker execute thread num
worker.exec.threads=100

# worker heartbeat interval
worker.heartbeat.interval=10

# submit the number of tasks at a time
worker.fetch.task.num = 10

# only less than cpu avg load, worker server can work. default value : the number of cpu cores * 2
worker.max.cpupload.avg=10

# only larger than reserved memory, worker server can work. default value : physical memory * 1/6, unit is G.
worker.reserved.memory=1
```

escheduler-api

web配置文件

- application.properties

```
# server port
server.port=12345

# session config
server.session.timeout=7200

server.context-path=/escheduler/

# file size limit for upload
spring.http.multipart.max-file-size=1024MB
spring.http.multipart.max-request-size=1024MB

# post content
server.max-http-post-size=5000000
```

伪分布式部署

1, 创建部署用户

如上 创建部署用户

2, 根据实际需求来创建HDFS根路径

根据 common/common.properties 中 hdfs.startup.state 的配置来判断是否启动HDFS，如果启动，则需要创建 HDFS根路径，并将 owner 修改为部署用户，否则忽略此步骤

3, 项目编译

如上进行 项目编译

4, 修改配置文件

根据 配置文件说明 修改配置文件和 环境变量 文件

5, 创建目录并将环境变量文件复制到指定目录

- 创建 common/common.properties 下的data.basedir.path、data.download.basedir.path 和 process.exec.basepath 路径
- 将 escheduler_env.sh 和 escheduler_env.py 两个环境变量文件复制到 common/common.properties 配置的 escheduler.env.path 和 escheduler.env.py 的目录下，并将 owner 修改为部署用户

6, 启停服务

- 启停Master

```
sh ./bin/arklifter-daemon.sh start master-server
sh ./bin/arklifter-daemon.sh stop master-server
```

- 启停Worker

```
sh ./bin/arklifter-daemon.sh start worker-server
```

```
sh ./bin/arklifter-daemon.sh stop worker-server
```

- 启停Api

```
sh ./bin/arklifter-daemon.sh start api-server  
sh ./bin/arklifter-daemon.sh stop api-server
```

- 启停Logger

```
sh ./bin/arklifter-daemon.sh start logger-server  
sh ./bin/arklifter-daemon.sh stop logger-server
```

- 启停Alert

```
sh ./bin/arklifter-daemon.sh start alert-server  
sh ./bin/arklifter-daemon.sh stop alert-server
```

分布式部署

1, 创建部署用户

- 在需要部署调度的机器上如上 [创建部署用户](#)
- [将主机器和各个其它机器SSH打通](#)

2, 根据实际需求来创建HDFS根路径

根据 **common/common.properties** 中 **hdfs.startup.state** 的配置来判断是否启动HDFS，如果启动，则需要创建HDFS根路径，并将 **owner** 修改为部署用户，否则忽略此步骤

3, 项目编译

如上进行 [项目编译](#)

4, 将环境变量文件复制到指定目录

将 **escheduler_env.sh** 和 **escheduler_env.py** 两个环境变量文件复制到 **common/common.properties** 配置的 **escheduler.env.path** 和 **escheduler.env.py** 的目录下，并将 **owner** 修改为部署用户

5, 修改 **install.sh**

修改 **install.sh** 中变量的值，替换成自身业务所需的值

6, 一键部署

- 安装 `pip install kazoo`
- 使用部署用户 `sh install.sh` 一键部署

服务监控

`monitor_server.py` 脚本是监听，master和worker服务挂掉重启的脚本

注意：在全部服务都启动之后启动

```
nohup python -u monitor_server.py > nohup.out 2>&1 &
```

日志查看

日志统一存放于指定文件夹内

```
logs/
├── escheduler-alert-server.log
├── escheduler-master-server.log
├── escheduler-worker-server.log
├── escheduler-api-server.log
└── escheduler-logger-server.log
```

任务插件开发

提醒:目前任务插件开发暂不支持热部署

基于SHELL的任务

基于YARN的计算 (参见 MapReduceTask)

- 需要在 `cn.escheduler.server.worker.task` 下的 `TaskManager` 类中创建自定义任务(也需在 `TaskType` 注册对应的任务类型)
- 需要继承 `cn.escheduler.server.worker.task` 下的 `AbstractYarnTask`
- 构造方法调度 `AbstractYarnTask` 构造方法
- 继承 `AbstractParameters` 自定义任务参数实体
- 重写 `AbstractTask` 的 `init` 方法中解析自定义任务参数
- 重写 `buildCommand` 封装command

基于非YARN的计算 (参见 ShellTask)

- 需要在 `cn.escheduler.server.worker.task` 下的 `TaskManager` 中创建自定义任务
- 需要继承 `cn.escheduler.server.worker.task` 下的 `AbstractTask`
- 构造方法中实例化 `ShellCommandExecutor`

```
public ShellTask(TaskProps props, Logger logger) {
    super(props, logger);

    this.taskDir = props.getTaskDir();

    this.processTask = new ShellCommandExecutor(this::logHandle,
        props.getTaskDir(), props.getTaskAppId(),
        props.getTenantCode(), props.getEnvFile(), props.getTaskStartTime(),
        props.getTaskTimeout(), logger);
    this.processDao = DaoFactory.getDaoInstance(ProcessDao.class);
}
```

传入自定义任务的 `TaskProps` 和自定义 `Logger`, `TaskProps` 封装了任务的信息, `Logger` 分装了自定义日志信息

- 继承 `AbstractParameters` 自定义任务参数实体
- 重写 `AbstractTask` 的 `init` 方法中解析自定义任务参数实体
- 重写 `handle` 方法, 调用 `ShellCommandExecutor` 的 `run` 方法, 第一个参数传入自己的 `command`, 第二个参数传入 `ProcessDao`, 设置相应的 `exitStatusCode`

基于非SHELL的任务 (参见 SqlTask)

- 需要在 `cn.escheduler.server.worker.task` 下的 `TaskManager` 中创建自定义任务
- 需要继承 `cn.escheduler.server.worker.task` 下的 `AbstractTask`
- 继承 `AbstractParameters` 自定义任务参数实体
- 构造方法或者重写 `AbstractTask` 的 `init` 方法中, 解析自定义任务参数实体
- 重写 `handle` 方法实现业务逻辑并设置相应的 `exitStatusCode`

使用手册

登录

- 输入<http://192.168.xx.xx:8888/view/login/index.html>网址，输入用户名：admin，密码：escheduler123 登录



- 登录之后每个页面的右上角都有用户的身份标识。点击下拉箭头包含用户信息和退出两个按钮



- 点击“用户信息”按钮，如下图：

用户信息

用户名称 admin

邮箱 825193156@qq.com

手机 15001335629

权限 管理员

创建时间 2018-03-27 15:48:50

更新时间 2018-10-24 17:40:22

修改

- 点击“修改”按钮，修改用户信息

编辑用户

* 用户名称

* 邮箱

手机

[取消](#) [确认编辑](#)

- 点击退出按钮则退出系统，返回登录页面

安全中心

- 只有管理员才有安全中心，安全中心的主要功能是给管理员提供管理普通用户的功能。
- 管理员可以有多个，管理员是功能上的管理，不参与具体的业务。也就是说管理员是不能执行具体任务的。

租户管理

租户是Linux上的用户，用于作业的提交。

- 创建、编辑租户



- 租户编码：租户编码是Linux上的用户，唯一，不能重复
- 租户名称：租户的名称
- 队列：租户对应的YARN上的队列，在数据库 t_escheduler_queue 中设置
- 描述：租户的描述信息

用户管理

用户是EasyScheduler上的用户，用于EasyScheduler上的功能操作。

- 创建、编辑用户

编辑用户

* 用户名称

* 密码

* 租户

* 邮箱

手机

[取消](#) [确认编辑](#)

- 用户名称：用户的名称，唯一，不能重复
- 租户：设置该用户所属的租户
- 邮箱：输入用户的邮箱，用来邮件发送和任务告警
- 手机：输入用户的手机号

注意：如果该用户切换了租户，则该用户所在租户下所有资源将复制到切换的新租户下

- 授权

管理员可以对普通用户进行非其创建的项目、资源、数据源和UDF函数进行授权。因为项目、资源、数据源和UDF函数授权方式都是一样的，所以以项目授权为例介绍。

- 1.点击指定人的授权按钮，如下图：



- 2.选中项目按钮，进行项目授权



- 项目列表：是该用户未授权的项目
- 已选项目：是该用户已授权的项目。
- 特别注意：对于用户自己创建的项目，该用户拥有所有的权限。则项目列表和已选项目列表中不会体现。

告警组管理

告警组是告警用户抽象出来的组，使用告警组来管理用户。

- 新建、编辑邮件组



- 组名称: 输入组的名称
- 组类型: 支持邮件/短信两种
- 备注: 输入告警组的备注信息
- 管理用户



- 管理用户列表：是未添加到该组的用户列表
- 已选管理用户：是已添加到该组的用户列表

服务管理

服务管理是对EasyScheduler的Master、Worker的任务监控

Master

服务管理-Master							
编号	ip	进程pid	zk注册目录	cpuUsage	memoryUsage	创建时间	最后心跳时间
1	192.168.220.204	22716	/arklifter/masters/192.168.220.204_0000000000	0.45 %	7.57 %	2018-11-28 19:09:19	2018-12-14 10:53:39

Worker

服务管理-Worker							
编号	ip	进程pid	zk注册目录	cpuUsage	memoryUsage	创建时间	最后心跳时间
1	192.168.220.155	23281	/arklifter/workers/192.168.220.155_0000000000	2.40 %	12.80 %	2018-11-28 19:09:29	2018-12-14 10:54:52
2	192.168.220.156	31231	/arklifter/workers/192.168.220.156_0000000001	26.62 %	25.03 %	2018-11-28 19:09:37	2018-12-14 10:54:59

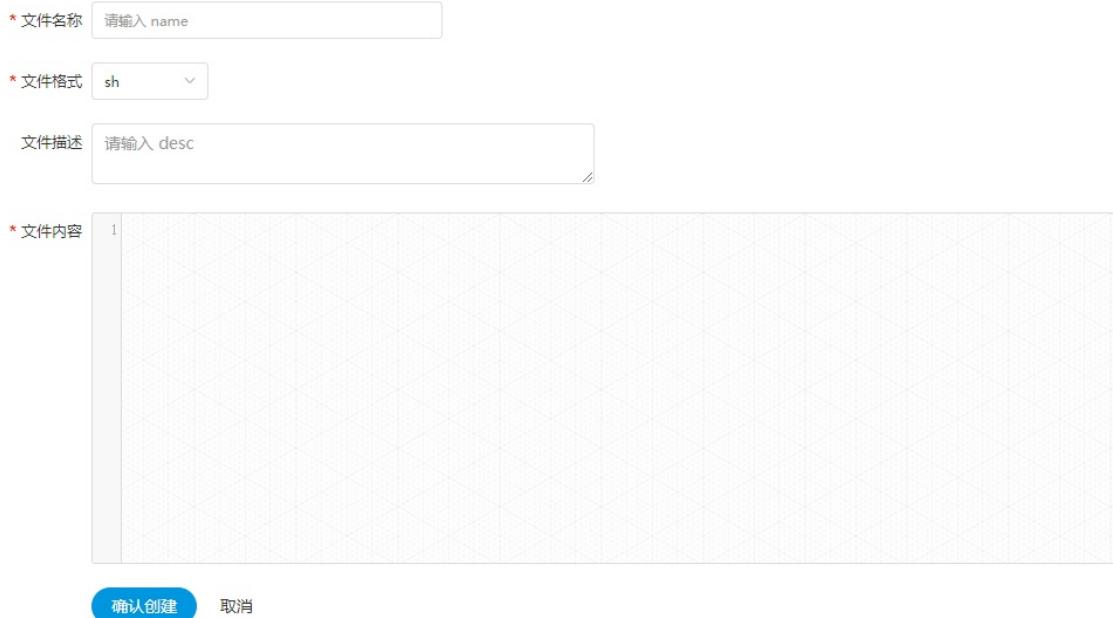
资源中心

资源中心主要分为文件管理和UDF函数管理。文件管理：主要是用户的程序，脚本和配置文件需要上传到HDFS进行统一管理 UDF函数管理：对用户创建的UDF进行管理

文件管理

创建文件

创建文件



A screenshot of a 'Create File' form. It includes fields for file name, file format, file description, and file content. The file content area is a large text input field with a grid background. Buttons at the bottom are 'Confirm Create' (highlighted) and 'Cancel'.

* 文件名称 请输入 name

* 文件格式 sh

文件描述 请输入 desc

* 文件内容

确认创建 取消

- 文件格式支持以下几种类型: txt、log、sh、conf、cfg、py、java、sql、xml、hql

上传文件



A screenshot of a 'File Upload' form. It has fields for file name, description, and a file upload area. A note says to drag files into the upload window. Buttons at the bottom are 'Cancel' and 'Confirm Upload' (highlighted).

* 文件名 请输入 name

描述 请输入 desc

* 上传文件 点击上传

请将文件拖拽到当前上传窗口内！ 取消 确认上传

- 文件名: 输入文件的名称
- 描述: 输入文件的描述信息
- 上传文件: 点击上传按钮进行上传, 将文件拖拽到上传区域, 文件名会自动以上传的文件名称补全

文件查看

对可查看的文件类型, 点击 文件名称 可以查看文件详情

文件详情

1525_new

127 MB

```

1 [INFO][42823460794580] 2019-01-03 14:36:51.207 cn.analysys.arklifter.server.worker.log.JobLogger:[138] - [jobId:FLOW_25_1401_1628] (stdout, stderr) -> ++ pd
2 + v_shells=/bin/sh /opt/soft/program/arklifter/exec/flow/5/25/1401/1628
3 + source ./mysql_pub_func_new.sh
4 ++ COMBINE_MYSQL='mysql -h 192.168.220.124 -uel -pEl12345'
5 ++ export MLS_LANG=AMERICAN_AMERICA_EU.UTF8
6 ++ MLS_LANG=AMERICAN_AMERICA_EU.UTF8
7 ++ basename p_bds_dev_app_installation_list_d.sh
8 ++ v_shells=/bin/sh /opt/soft/program/arklifter/exec/flow/5/25/1401/1628
9 ++ echo p_bds_dev_app_installation_list_d.sh
10 ++ awk -F. '{print $1}'
11 ++ v_shells=/bin/sh /opt/soft/program/arklifter/exec/flow/5/25/1401/1628
12 ++ date +%Y%m%d%H%M%S
13 ++ v_filedate=20190103143644
14 ++ v_proc_date=20170903
15 ++ v_proc_date=20170903
16 ++ v_proc_name=p_bds_dev_app_installation_list_d
17 ++ v_source_dbs=
18 ++ v_target_dbs=
19 ++ v_source_tableods=ods_dev_installed_app_d
20 ++ v_target_tablehive=bds.bds_dev_app_installation_list_d
21 ++ v_logfile=/opt/soft/program/arklifter/tmp/p_bds_dev_app_installation_list_d_20170903_20190103143644.log
22 ++ v_application_name=hive.bds.bds_dev_app_installation_list_d_20170903
23 ++ insertLog p_bds_dev_app_installation_list_d_20170903 ods.ods_dev_installed_app_d hive.bds.bds_dev_app_installation_list_d
24 ++ V_PROC_NAME=p_20170903
25 ++ V_PROC_DATE=20170903
26 ++ V_SOURCE_TABLE=ods.ods_dev_installed_app_d
27 ++ V_TARGET_TABLE=hive.bds.bds_dev_app_installation_list_d
28 ++ V_SCHEMA=
29 ++ date +%Y-%m-%d %H:%M:%S"
30 ++ V_STAMP='2019-01-03 14:36:44'
31 ++ v_file='call etl.HIVE_INSERT_LOG(''`p_bds_dev_app_installation_list_d`'', `''20170903`', `''ods.ods_dev_installed_app_d`'', `''hive.bds.bds_dev_app_installation_list_d`'', `''`'')'
32 ++ mysql -h 192.168.220.124 -uel -pEl12345
33 ++ getLogNumber 20170903 ods.ods_dev_installed_app_d
34 [INFO][428234623225047] 2019-01-03 14:36:52.370 cn.analysys.arklifter.server.worker.log.JobLogger:[138] - [jobId:FLOW_25_1401_1628] (stdout, stderr) -> ++ V_PROC_DATE=20170903
35 ..v_file='call etl.HIVE_INSERT_LOG(''`p_bds_dev_app_installation_list_d`'', `''20170903`', `''ods.ods_dev_installed_app_d`'', `''hive.bds.bds_dev_app_installation_list_d`'', `''`'')'

```

下载文件

可以在文件详情中点击右上角下载按钮下载文件，或者在文件列表后的下载按钮下载文件

文件重命名



删除

- 文件列表，点击删除按钮，删除文件

UDF管理

资源管理

资源管理和文件管理功能类似，不同之处是资源管理是上传的UDF函数，文件管理上传的是用户程序，脚本及配置文件

函数管理

创建、编辑UDF函数

编辑UDF函数

* 类型 HIVE UDF

* UDF函数名称

* 包名类名

参数

数据库名

* UDF资源

使用说明

目前只支持HIVE的临时UDF函数

- UDF函数名称：输入UDF函数时的名称
- 包名类名：输入UDF函数的全路径
- 参数：用来标注函数的输入参数
- 数据库名：预留字段，用于创建永久UDF函数
- UDF资源：设置创建的UDF对应的资源文件

数据源中心

数据源中心支持MySQL、POSTGRESQL、HIVE及Spark数据源

创建、编辑MySQL数据源

编辑数据源

* 数据源 MYSQL POSTGRESQL HIVE SPARK

* 数据源名称

描述

* IP/主机名

* 端口

* 用户名

密码

* 数据库名

jdbc连接参数

取消 测试连接 确认编辑

- 数据源：选择MYSQL
- 数据源名称：输入数据源的名称
- 描述：输入数据源的描述
- IP/主机名：输入连接MySQL的IP
- 端口：输入连接MySQL的端口
- 用户名：设置连接MySQL的用户名
- 密码：设置连接MySQL的密码
- 数据库名：输入连接MySQL的数据库名称
- Jdbc连接参数：用于MySQL连接的参数设置，以JSON形式填写

创建、编辑POSTGRESQL数据源

编辑数据源

* 数据源 MYSQL POSTGRESQL HIVE SPARK

* 数据源名称

描述

* IP/主机名

* 端口

* 用户名

密码

* 数据库名

jdbc连接参数

取消 测试连接 确认编辑

- 数据源：选择POSTGRESQL
- 数据源名称：输入数据源的名称
- 描述：输入数据源的描述
- IP/主机名：输入连接POSTGRESQL的IP
- 端口：输入连接POSTGRESQL的端口
- 用户名：设置连接POSTGRESQL的用户名
- 密码：设置连接POSTGRESQL的密码
- 数据库名：输入连接POSTGRESQL的数据库名称
- Jdbc连接参数：用于POSTGRESQL连接的参数设置，以JSON形式填写

创建、编辑HIVE数据源

1. 使用HiveServer2方式连接

编辑数据源

* 数据源 MYSQL POSTGRESQL HIVE SPARK

* 数据源名称

描述

* IP/主机名

* 端口

* 用户名

密码

* 数据库名

Jdbc连接参数

取消 测试连接 确认编辑

- 数据源：选择HIVE
- 数据源名称：输入数据源的名称
- 描述：输入数据源的描述
- IP/主机名：输入连接HIVE的IP
- 端口：输入连接HIVE的端口
- 用户名：设置连接HIVE的用户名
- 密码：设置连接HIVE的密码
- 数据库名：输入连接HIVE的数据库名称
- Jdbc连接参数：用于HIVE连接的参数设置，以JSON形式填写

2. 使用HiveServer2 HA Zookeeper方式连接

创建数据源

* 数据源 MYSQL POSTGRESQL HIVE SPARK

* 数据源名称

描述

* IP/主机名

* 端口

* 用户名

密码

* 数据库名

jdbc连接参数

取消 测试连接 确认提交

- 数据源：选择HIVE
- 数据源名称：输入数据源的名称
- 描述：输入数据源的描述
- IP/主机名：输入连接Zookeeper的集群
- 端口：输入连接Zookeeper的端口
- 用户名：设置连接HIVE的用户名
- 密码：设置连接HIVE的密码
- 数据库名：输入连接HIVE的数据库名称
- Jdbc连接参数：用于Zookeeper连接的参数设置，以JSON形式填写

创建、编辑Spark数据源

编辑数据源

* 数据源 MYSQL POSTGRESQL HIVE SPARK

* 数据源名称

描述

* IP/主机名

* 端口

* 用户名

密码

* 数据库名

Jdbc连接参数

[取消](#) [测试连接](#) [确认编辑](#)

- 数据源：选择Spark
- 数据源名称：输入数据源的名称
- 描述：输入数据源的描述
- IP/主机名：输入连接Spark的IP
- 端口：输入连接Spark的端口
- 用户名：设置连接Spark的用户名
- 密码：设置连接Spark的密码
- 数据库名：输入连接Spark的数据库名称
- Jdbc连接参数：用于Spark连接的参数设置，以JSON形式填写

首页



首页是对所有项目在指定时间范围内的任务状态、流程状态和流程定义的统计。

首页和项目首页的主要区别在于：

- 首页中的图表是没有链接的，项目首页中图表是有链接的
- 首页统计的是所有的项目，项目首页统计的是某一个项目

项目管理

项目是调度对用户流程定义DAG分组的一个抽象

创建、编辑项目

编辑项目

* 项目名称	test
描述	这是一个测试的项目

取消 确认编辑

- 项目名称：输入项目的名称
- 描述：输入项目的描述

项目首页

点击项目列表中的项目名称，可以跳转到指定的项目首页，如下图：



- 项目首页其中包含四个部分，任务状态统计，流程状态统计、流程定义统计及统计的时间范围
- 任务状态统计：是指在指定时间范围内，统计任务实例中的待运行、失败、运行中、完成、成功的个数
- 流程状态统计：是指在指定时间范围内，统计流程实例中的待运行、失败、运行中、完成、成功的个数
- 流程定义统计：是统计该用户创建的流程定义及管理员授予该用户的流程定义

注意：可以点击图，或者数量跳转到相应的任务实例，流程实例和流程定义列表

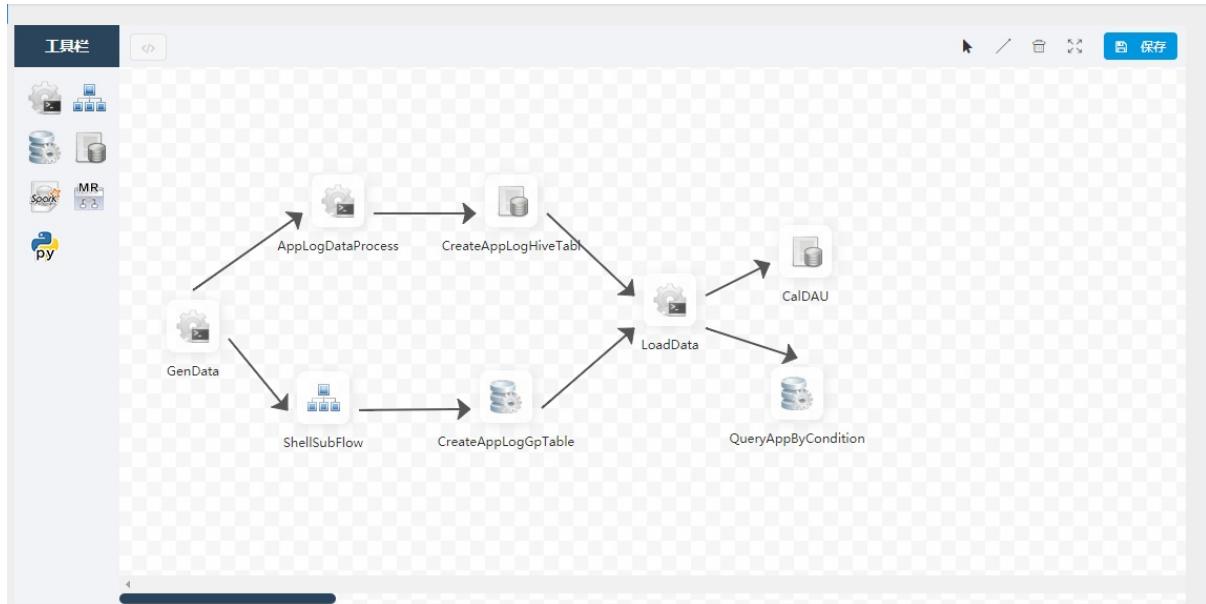
工作流

工作流分为流程定义、流程实例和任务实例三个功能模块

- 流程定义：是可视化拖拽成的DAG的统称，它是静态的，没有状态
- 流程实例：对流程定义的每次实例化会生成一个流程实例，是动态的，是有状态的
- 任务实例：流程实例DAG中每个Task称为任务实例，是动态的，是有状态的

流程定义

创建工作流



- 左侧工具栏 => 是目前调度支持的任务类型，当前调度支持SHELL、子流程、存储过程、SQL、MR、Spark和Python七种任务类型
- 右上角图标 => 分别是拖动节点和选中项、选择线条连线、删除选中的线或节点、全屏和流程定义保持，其主要功能是DAG的绘制所用

创建 SHELL 节点

拖动工具栏中的 任务节点到画板中，双击任务节点，如下图：

当前节点设置

节点名称

运行标志 正常 禁止执行

描述

任务优先级

失败重试次数 (次) 失败重试间隔 (分)

任务超时告警

脚本

资源

自定义参数

取消

确认添加

- 节点名称：一个流程定义中的节点名称是唯一的
- 运行标志：标识这个节点是否能正常调度
- 描述信息：描述该节点的功能
- 失败重试次数：任务失败重新提交的次数，支持下拉和手填
- 失败重试间隔：任务失败重新提交任务的时间间隔，支持下拉和手填
- 脚本：用户开发的SHELL程序
- 资源：是指脚本中需要调用的资源文件列表
- 自定义参数：是SHELL局部的用户自定义参数，会替换脚本中以\${变量}的内容

创建子流程节点

 拖动工具栏中的  任务节点到画板中，双击任务节点，如下图：

当前节点设置

 进入该子节点节点名称 运行标志 正常 禁止执行描述 任务优先级  MEDIUM 任务超时告警 子节点 

取消

确认添加

- 节点名称：一个流程定义中的节点名称是唯一的
- 运行标志：标识这个节点是否能正常调度
- 描述信息：描述该节点的功能
- 子节点：是选择子流程的流程定义，右上角进入该子节点可以跳转到所选子流程的流程定义

创建 存储过程 节点

 拖动工具栏中的  任务节点到画板中，双击任务节点，如下图：

当前节点设置

节点名称

请输入name(必填)

运行标志

正常
 禁止执行

描述

请输入desc

任务优先级

↑ MEDIUM ↓

失败重试次数

(次)

失败重试间隔

 (分)

任务超时告警

数据源

MySQL

188 mysql数据源

方法

请输入method(选填)

自定义参数

+

- 节点名称：一个流程定义中的节点名称是唯一的
- 运行标志：标识这个节点是否能正常调度
- 描述信息：描述该节点的功能
- 失败重试次数：任务失败重新提交的次数，支持下拉和手填
- 失败重试间隔：任务失败重新提交任务的时间间隔，支持下拉和手填
- 数据源：存储过程的数据源类型支持MySQL和POSTGRESQL两种，选择对应的数据源
- 方法：是存储过程的方法名称
- 自定义参数：存储过程的自定义参数类型支持IN、OUT两种，数据类型支持VARCHAR、INTEGER、LONG、FLOAT、DOUBLE、DATE、TIME、TIMESTAMP、BOOLEAN九种数据类型

创建 SQL 节点

拖动工具栏中的  任务节点到画板中，双击任务节点，如下图：

当前节点设置

节点名称

运行标志 正常 禁止执行

描述

任务优先级

失败重试次数 (次) 失败重试间隔 (分)

任务超时告警

数据源

sql类型 表格 附件

sql语句

自定义参数

- 节点名称：一个流程定义中的节点名称是唯一的
- 运行标志：标识这个节点是否能正常调度
- 描述信息：描述该节点的功能
- 失败重试次数：任务失败重新提交的次数，支持下拉和手填
- 失败重试间隔：任务失败重新提交任务的时间间隔，支持下拉和手填
- 数据源：SQL数据源支持MySQL、POSTGRESQL、HIVE和Spark四中数据源类型，选择对应的数据源
- sql类型：支持查询和非查询两种，查询是select类型的查询，是有结果集返回的，可以指定邮件通知为表格、附件或表格附件三种模板。非查询是没有结果集返回的，是针对update、delete、insert三种类型的操作
- sql参数：输入参数格式为key1=value1;key2=value2...
- sql语句：SQL语句
- UDF函数：对于HIVE类型的数据源，可以引用资源中心中创建的UDF函数，其他类型的数据源暂不支持UDF函数
- 自定义参数：SQL任务类型自定义参数类型和数据类型同存储过程任务类型一样。区别在于SQL任务类型自定义参数会替换sql语句中\${变量}，而存储过程是自定义参数顺序的给方法设置值

创建 MR 节点

 拖动工具栏中的  任务节点到画板中，双击任务节点，如下图：

(1) JAVA程序

当前节点设置

节点名称

运行标志 正常 禁止执行

描述

任务优先级

失败重试次数 (次) 失败重试间隔 (分)

任务超时告警

程序类型

主函数的class

主jar包

命令行参数

其他参数

资源

自定义参数

- 节点名称：一个流程定义中的节点名称是唯一的
- 运行标志：标识这个节点是否能正常调度
- 描述信息：描述该节点的功能
- 失败重试次数：任务失败重新提交的次数，支持下拉和手填
- 失败重试间隔：任务失败重新提交任务的时间间隔，支持下拉和手填
- 主函数的class：是MR程序的入口Main Class的全路径
- 程序类型：选择JAVA语言
- 主jar包：是MR的jar包
- 命令行参数：是设置MR程序的输入参数，支持自定义参数变量的替换
- 其他参数：支持 -D、-files、-libjars、-archives格式
- 资源：如果其他参数中引用了资源文件，需要在资源中选择指定
- 自定义参数：是MR局部的用户自定义参数，会替换脚本中以\${变量}的内容

(2) Python程序

当前节点设置

节点名称

MRPythonWC

运行标志

正常 禁止执行

描述

请输入desc

任务优先级

↑ MEDIUM ↓

失败重试次数

(次) 失败重试间隔 (分)

任务超时告警

程序类型

PYT... ↓

主jar包

hadoop-streaming-2.7.3.jar

命令行参数

请输入命令行参数

其他参数

-mapper mapper.py -file mapper.py -reducer reducer.py -file reducer.py -input /journey/words.txt -output /journey/out/mr/\${outpath}

资源

mapper.py ×
reducer.py ×

自定义参数

IN VARCHAR \${system.datetime} - +

[取消](#)[确认添加](#)

- 节点名称：一个流程定义中的节点名称是唯一的
- 运行标志：标识这个节点是否能正常调度
- 描述信息：描述该节点的功能
- 失败重试次数：任务失败重新提交的次数，支持下拉和手填
- 失败重试间隔：任务失败重新提交任务的时间间隔，支持下拉和手填
- 程序类型：选择Python语言
- 主jar包：是运行MR的Python jar包
- 其他参数：支持 -D、-mapper、-reducer、-input -output格式，这里可以设置用户自定义参数的输入，比如：
- -mapper "mapper.py 1" -file mapper.py -reducer reducer.py -file reducer.py -input /journey/words.txt -output /journey/out/mr/\${currentTimeMillis}
- 其中 -mapper 后的 mapper.py 1是两个参数，第一个参数是mapper.py，第二个参数是1
- 资源：如果其他参数中引用了资源文件，需要在资源中选择指定
- 自定义参数：是MR局部的用户自定义参数，会替换脚本中以\${变量}的内容

[创建 Spark 节点](#)

拖动工具栏中的  任务节点到画板中，双击任务节点，如下图：

当前节点设置



The screenshot shows the configuration interface for a 'SparkWC' task node. The fields are as follows:

- 节点名称:** SparkWC
- 运行标志:** 正常 (Normal) (selected)
- 描述:** 请输入desc
- 任务优先级:** MEDIUM (可上下调整)
- 失败重试次数:** 0 (次)
- 失败重试间隔:** 1 (分)
- 任务超时告警:** (开关状态为开启)
- 程序类型:** SCALA
- 主函数的class:** com.journey.wordcount.SparkWordCount
- 主jar包:** journey-1.0-SNAPSHOT.jar
- 部署方式:** cluster (selected)
- Driver内核数:** 1
- Driver内存数:** 512M
- Executor数量:** 2
- Executor内存数:** 2G
- Executor内核数:** 2

At the bottom right are two buttons: 取消 (Cancel) and 确认添加 (Confirm Add).

- 节点名称：一个流程定义中的节点名称是唯一的
- 运行标志：标识这个节点是否能正常调度
- 描述信息：描述该节点的功能
- 失败重试次数：任务失败重新提交的次数，支持下拉和手填
- 失败重试间隔：任务失败重新提交任务的时间间隔，支持下拉和手填
- 程序类型：支持JAVA、Scala和Python三种语言
- 主函数的class：是Spark程序的入口Main Class的全路径
- 主jar包：是Spark的jar包
- 部署方式：支持yarn-cluster、yarn-client、和local三种模式
- Driver内核数：可以设置Driver内核数及内存数
- Executor数量：可以设置Executor数量、Executor内存数和Executor内核数
- 命令行参数：是设置Spark程序的输入参数，支持自定义参数变量的替换。

- 其他参数：支持 --jars、--files、--archives、--conf格式
- 资源：如果其他参数中引用了资源文件，需要在资源中选择指定
- 自定义参数：是MR局部的用户自定义参数，会替换脚本中以\${变量}的内容

注意：JAVA和Scala只是用来标识，没有区别，如果是Python开发的Spark则没有主函数的class，其他都是一样

创建 Python 节点

拖动工具栏中的  任务节点到画板中，双击任务节点，如下图：

当前节点设置



节点名称: PythonTest

运行标志: 正常 禁止执行

描述: 请输入desc

任务优先级: MEDIUM

失败重试次数: 0 (次) 失败重试间隔: 1 (分)

任务超时告警:

脚本:

```

1 #!/usr/bin/env python
2 # -*- coding:utf-8 -*-
3 # Author:qiaozhanwei
4
5 import numpy as np
6 '''
7 numpy.loadtxt 加载考试成绩
8 '''
9 def loaddata(file, delimiter):
10     data = np.loadtxt('exam_score.txt', delimiter=',', skiprows=1)
11     print('Dimensions: ', data.shape)
12     '''
13     注意：data[起始行:结束行, 起始列:结束列]
14     包含头，但不包含尾部
15     '''

```

- 节点名称：一个流程定义中的节点名称是唯一的
- 运行标志：标识这个节点是否能正常调度
- 描述信息：描述该节点的功能
- 失败重试次数：任务失败重新提交的次数，支持下拉和手填
- 失败重试间隔：任务失败重新提交任务的时间间隔，支持下拉和手填
- 脚本：用户开发的Python程序

- 资源：是指脚本中需要调用的资源文件列表
- 自定义参数：是Python局部的用户自定义参数，会替换脚本中以\${变量}的内容

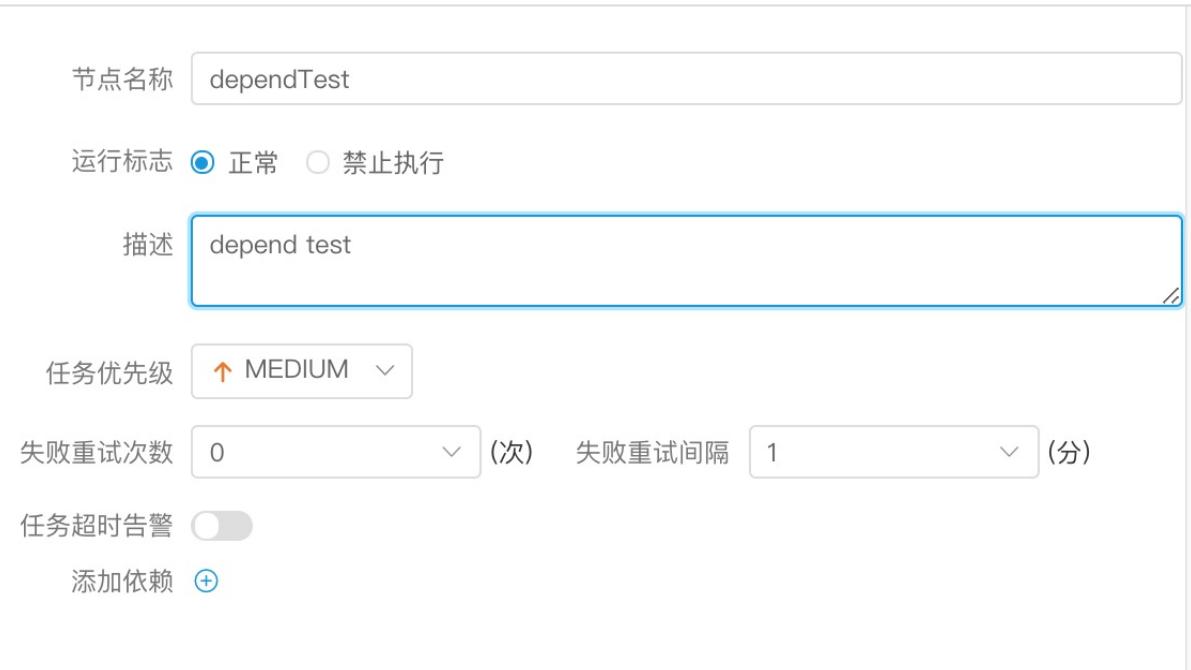
创建 依赖 节点

任务依赖分为水平依赖和垂直依赖

- 水平依赖就是指DAG图的有向依赖，是同一个流程实例任务节点的前驱，后继之间的依赖关系
- 垂直依赖是流程实例之间的任务依赖，基于定时的依赖。

 拖动工具栏中的任务节点到画板中，双击任务节点，如下图：

当前节点设置



节点名称

运行标志 正常 禁止执行

描述

任务优先级

失败重试次数 (次) 失败重试间隔 (分)

任务超时告警

添加依赖

- 节点名称：一个流程定义中的节点名称是唯一的
- 运行标志：标识这个节点是否能正常调度
- 描述信息：描述该节点的功能
- 失败重试次数：任务失败重新提交的次数，支持下拉和手填
- 失败重试间隔：任务失败重新提交任务的时间间隔，支持下拉和手填
- 任务依赖：增加依赖条件，选择依赖流程定义、节点名称（默认为全部节点）、依赖周期、依赖时间点

当前节点设置

节点名称

运行标志 正常 禁止执行

描述

任务优先级

失败重试次数 (次) 失败重试间隔 (分)

任务超时告警

添加依赖

且 且 或 或

- 选择多个依赖条件之间的关系：或、且

当前节点设置

节点名称

运行标志 正常 禁止执行

描述

任务优先级

失败重试次数 (次) 失败重试间隔 (分)

任务超时告警

添加依赖

且 ALL 日 昨天

或 ALL 日 昨天

流程实例列表

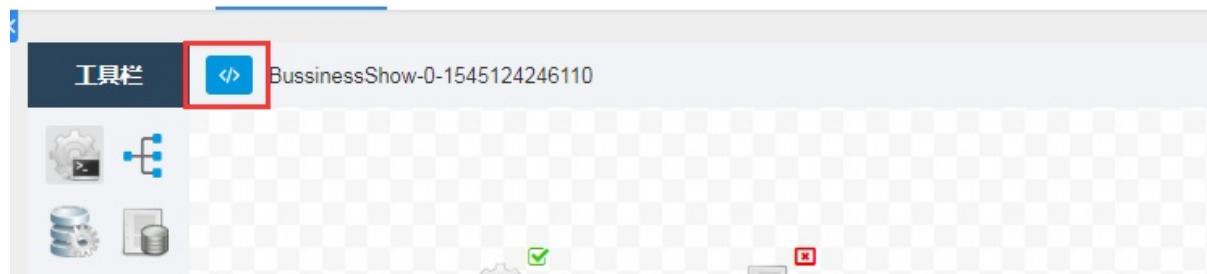
流程实例列表页是可以显示所有本项目下所有流程实例的列表，并有对流程实例进行名称、状态、时间等字段的筛选功能。通过列表页可以直接对某一个流程实例进行编辑、重跑、恢复失败、暂停、停止、恢复暂停、删除、查看甘特图等操作。



- 编辑功能：对已经完成的流程实例，点击编辑按钮，可以对其进行编辑，如图：



- 查看流程实例运行变量



- 点击隐藏按钮，查看流程实例运行变量。如下图：



- 点击变量是对变量的复制
- 点击"重跑",可以对已经完成的流程实例进行重新运行操作, 如图:



- 点击"恢复失败", 可以对失败的流程进行恢复, 直接从失败的任务节点开始运行。如图:



- 点击"暂停", 可以对正在运行的流程进行暂停操作,如图:



- 点击"停止", 可以对正在运行的流程进行停止操作, 如图:



- 点击"恢复暂停",可以对暂停的流程恢复, 直接从暂停的节点开始运行, 如图:



- **删除**

删除流程实例及流程实例下的任务实例

- **Gantt**

Gantt图纵轴是某个流程实例下的任务实例的拓扑排序, 横轴是任务实例的运行时间



任务实例列表页

当前节点设置

[查看历史](#) [查看日志](#)

节点名称	日期测试				
运行标志	<input checked="" type="radio"/> 正常	<input type="radio"/> 禁止执行			
描述	请输入desc				
任务优先级	↑ MEDIUM ↓				
失败重试次数	0	(次)	失败重试间隔	1	(分)
任务超时告警	<input type="checkbox"/>				
脚本	<pre> 1 echo "biz_date ---> : "\${biz_date} 2 echo "biz_curdate ---> : "\${biz_curdate} 3 echo "system_datetime ---> : "\${system_datetime} 4 echo "local_param --> : "\${local_param} 5 echo "local_param_bizdate --> : "\${local_param_bizdate}</pre>				
资源	请选择资源				
自定义参数	biz_date	IN	VARCHAR	`\${system.biz.date}`	删除
	biz_curdate	IN	VARCHAR	`\${system.biz.curda}`	删除
	system_datetime	IN	VARCHAR	`\${system.datetime}`	删除
	local_param	IN	VARCHAR	`\${local_param}`	删除
	local_param_bizdate	IN	VARCHAR	`\${global_bizdate}`	删除 +

- 点击任务实例节点，点击 查看历史，可以查看该流程实例运行的该任务实例列表

[查看日志](#)

当前节点设置

查看历史 **查看日志**

节点名称 日期测试

运行标志 正常 禁止执行

描述 请输入desc

任务优先级 ↑ MEDIUM ↓

失败重试次数 0 (次) 失败重试间隔 1 (分)

任务超时告警

脚本

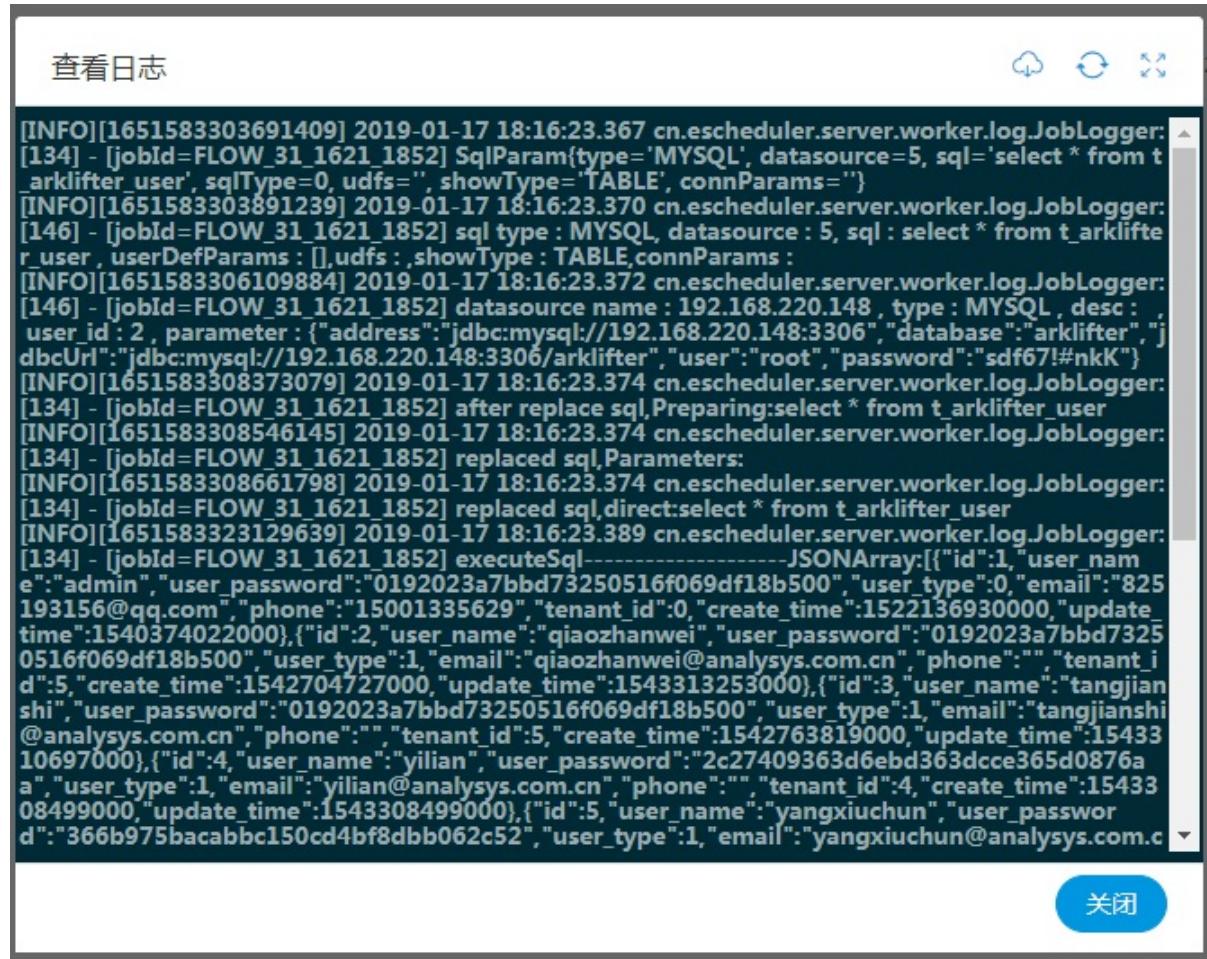
```
1 echo "biz_date ---> : "${biz_date}
2 echo "biz_curdate ---> : "${biz_curdate}
3 echo "system_datetime ---> : "${system_datetime}
4 echo "local_param --> : "${local_param}
5 echo "local_param_bizdate --> : "${local_param_bizdate}
```

资源 请选择资源

自定义参数

biz_date	IN	VARCHAR	`\${system.biz.date}`	
biz_curdate	IN	VARCHAR	`\${system.biz.curda}`	
system_datetime	IN	VARCHAR	`\${system.datetime}`	
local_param	IN	VARCHAR	`\${local_param}`	
local_param_bizdate	IN	VARCHAR	`\${global_bizdate}`	

- 点击任务实例节点，点击 查看日志，可以查看该任务实例运行的日志，如下图：



- 右上角是下载日志、刷新日志和放大/缩小按钮
- 注意：日志查看是分片的查看，上下滚动查看

任务实例

任务实例是流程实例任务节点的列表

任务实例											
编号	名称	工作流实例	节点类型	状态	提交时间	开始时间	结束时间	host	运行时长(秒)	重试次数	操作
1	sqlTest	sqlTest-0-15477720183217	SQL	●	2019-01-17 18:16:23	2019-01-17 18:16:23	2019-01-17 18:16:26	192.168.220.156	3	0	<button>查看日志</button>
2	nodeTest	testLog-0-1547638746316	SHELL	●	2019-01-16 19:39:07	2019-01-16 19:39:07	2019-01-16 19:39:08	192.168.220.156	1	0	<button>查看日志</button>
3	nodeTest	testLog-0-1547638512096	SHELL	○	2019-01-16 19:35:13	2019-01-16 19:35:13	2019-01-16 19:35:15	192.168.220.156	2	0	<button>查看日志</button>
4	nodeTest	testLog-0-1547638489670	SHELL	○	2019-01-16 19:34:50	2019-01-16 19:34:50	2019-01-16 19:34:51	192.168.220.156	1	0	<button>查看日志</button>
5	nodeTest	testLog-0-1547638487596	SHELL	○	2019-01-16 19:34:48	2019-01-16 19:34:48	2019-01-16 19:34:49	192.168.220.155	1	0	<button>查看日志</button>
6	nodeTest	testLog-0-1547638485524	SHELL	○	2019-01-16 19:34:46	2019-01-16 19:34:46	2019-01-16 19:34:47	192.168.220.156	1	0	<button>查看日志</button>
7	nodeTest	testLog-0-1547638483459	SHELL	○	2019-01-16 19:34:44	2019-01-16 19:34:44	2019-01-16 19:34:45	192.168.220.156	1	0	<button>查看日志</button>
8	CalDAU	BusinessShow-0-1547637917760	SQL	●	2019-01-16 19:26:14	2019-01-16 19:26:13	2019-01-16 19:29:52	192.168.220.156	339	0	<button>查看日志</button>
9	QueryAppByCondition	BusinessShow-0-1547637917760	PROCEDURE	○	2019-01-16 19:26:14	2019-01-16 19:26:13	2019-01-16 19:26:20	192.168.220.156	7	0	<button>查看日志</button>
10	LoadData	BusinessShow-0-1547637917760	SHELL	●	2019-01-16 19:25:48	2019-01-16 19:25:48	2019-01-16 19:26:13	192.168.220.156	65	0	<button>查看日志</button>

两种方式查看任务实例：

- 第一种是通过流程实例任务节点 查看历史，这时查看的是此流程实例的任务实例 重跑的列表
- 第二种是通过点击 流程实例 导航栏，调转到流程实例列表，这时查看的是所有流程实例的任务实例列表

查看日志：点击 查看日志 按钮，可下载和查看日志

系统参数

系统参数

变量	含义
<code> \${system.biz.date}</code>	日常调度实例定时的定时时间前一天，格式为 yyyyMMdd，补数据时，该日期 +1
<code> \${system.biz.curdate}</code>	日常调度实例定时的定时时间，格式为 yyyyMMdd，补数据时，该日期 +1
<code> \${system.datetime}</code>	日常调度实例定时的定时时间，格式为 yyyyMMddHHmmss，补数据时，该日期 +1

时间自定义参数

支持代码中自定义变量名，声明方式： `${变量名}`。可以是引用 "系统参数" 或指定 "常量"。

我们定义这种基准变量为 `[...]` 格式的， `${yyyyMMddHHmmss}` 是可以任意分解组合的，比如： `${yyyyMMdd}`, `${HHmmss}`, `${yyyy-MM-dd}` 等

也可以这样：

- 后 N 年： `${add_months(yyyyMMdd,12*N)}`
- 前 N 年： `${add_months(yyyyMMdd,-12*N)}`
- 后 N 月： `${add_months(yyyyMMdd,N)}`
- 前 N 月： `${add_months(yyyyMMdd,-N)}`
- 后 N 周： `${yyyyMMdd+7*N}`
- 前 N 周： `${yyyyMMdd-7*N}`
- 后 N 天： `${yyyyMMdd+N}`
- 前 N 天： `${yyyyMMdd-N}`
- 后 N 小时： `${HHmmss+N/24}`
- 前 N 小时： `${HHmmss-N/24}`
- 后 N 分钟： `${HHmmss+N/24/60}`
- 前 N 分钟： `${HHmmss-N/24/60}`

用户自定义参数

用户自定义参数分为全局参数和局部参数。全局参数是保存流程定义和流程实例的时候传递的全局参数，全局参数可以在整个流程中的任何一个任务节点的局部参数引用。

例如：

设置DAG图名称

日期测试-补数

请输入desc(选填)

设置全局

local_param	IN	VARCHAR	这是全局参数	删除
global_bizdate	IN	VARCHAR	\${system.biz.date}	删除

+

取消 添加

The screenshot shows a configuration interface for a DAG. At the top, there's a field labeled "设置DAG图名称" (Set DAG Graph Name) containing the value "日期测试-补数". Below it is a note "请输入desc(选填)" (Please enter desc (optional)). The next section, "设置全局" (Set Global), contains two parameter entries. Each entry has four fields: a key, a type, a value, and a delete button. The first entry is "local_param" with type "IN" and value "VARCHAR", and the note "这是全局参数" (This is a global parameter). The second entry is "global_bizdate" with type "IN" and value "VARCHAR", and the value is "\${system.biz.date}". There is also a plus sign (+) button to add more parameters. At the bottom right are "取消" (Cancel) and "添加" (Add) buttons.

- `global_bizdate`为全局参数，引用的是系统参数。

当前节点设置

节点名称

运行标志 正常 禁止执行

描述

任务优先级

失败重试次数 (次) 失败重试间隔 (分)

任务超时告警

脚本

```

1 echo "biz_date ---> : "${biz_date}
2 echo "biz_curdate ---> : "${biz_curdate}
3 echo "system_datetime ---> : "${system_datetime}
4 echo "local_param --> : "${local_param}
5 echo "local_param_bizdate --> : "${local_param_bizdate}

```

资源

自定义参数

biz_date	IN	VARCHAR	<input type="text" value="\${system.biz.date}"/>	<input type="button" value="删除"/>
biz_curdate	IN	VARCHAR	<input type="text" value="\${system.biz.curda}"/>	<input type="button" value="删除"/>
system_datetime	IN	VARCHAR	<input type="text" value="\${system.datetime}"/>	<input type="button" value="删除"/>
local_param	IN	VARCHAR	<input type="text" value="\${local_param}"/>	<input type="button" value="删除"/>
local_param_bizdate	IN	VARCHAR	<input type="text" value="\${global_bizdate}"/>	<input type="button" value="删除"/> <input type="button" value="添加"/>

- 任务中local_param_bizdate通过\${global_bizdate}来引用全局参数，对于脚本可以通过\${local_param_bizdate}来引用变量local_param_bizdate的值，或通过JDBC直接将local_param_bizdate的值set进去