xxl-job

1.功能描述

XXL-JOB 是一个轻量级分布式任务调度平台，其核心设计目标是开发迅速、学习简单、轻量级、易　扩展。

特性：

- 1、简单：支持通过 Web 页面对任务进行 CRUD 操作，操作简单，一分钟上手；
- 2、动态：支持动态修改任务状态、启动/停止任务，以及终止运行中任务，即时生效；
- 3、调度中心 HA（中心式）：调度采用中心式设计，“调度中心”自研调度组件并支持集群部署，可保证调度中心 HA；
- 4、执行器 HA（分布式）：任务分布式执行，任务”执行器”支持集群部署，可保证任务执行 HA；
- 5、注册中心: 执行器会周期性自动注册任务, 调度中心将会自动发现注册的任务并触发执行。同时，也支持手动录入执行器地址；
- 6、弹性扩容缩容：一旦有新执行器机器上线或者下线，下次调度时将会重新分配任务；
- 7、路由策略：执行器集群部署时提供丰富的路由策略，包括：第一个、最后一个、轮询、随机、一致性 HASH、最不经常使用、最近最久未使用、故障转移、忙碌转移等；
- 8、故障转移：任务路由策略选择”故障转移”情况下，如果执行器集群中某一台机器故障，将会自动 Failover 切换到一台正常的执行器发送调度请求。
- 9、阻塞处理策略：调度过于密集执行器来不及处理时的处理策略，策略包括：单机串行（默认）、丢弃后续调度、覆盖之前调度；
- 10、任务超时控制：支持自定义任务超时时间，任务运行超时将会主动中断任务；
- 11、任务失败重试：支持自定义任务失败重试次数，当任务失败时将会按照预设的失败重试次数主动进行重试；其中分片任务支持分片粒度的失败重试；
- 12、任务失败告警；默认提供邮件方式失败告警，同时预留扩展接口，可方便的扩展短信、钉钉等告警方式；
- 13、分片广播任务：执行器集群部署时，任务路由策略选择”分片广播”情况下，一次任务调度将会广播触发集群中所有执行器执行一次任务，可根据分片参数开发分片任务；
- 14、动态分片：分片广播任务以执行器为维度进行分片，支持动态扩容执行器集群从而动态增加分片数量，协同进行业务处理；在进行大数据量业务操作时可显著提升任务处理能力和速度。
- 15、事件触发：除了”Cron 方式”和”任务依赖方式”触发任务执行之外，支持基于事件的触发任务方式。调度中心提供触发任务单次执行的 API 服务，可根据业务事件灵活触发。
- 16、任务进度监控：支持实时监控任务进度；
- 17、Rolling 实时日志：支持在线查看调度结果，并且支持以 Rolling 方式实时查看执行器输出的完整的执行日志；
- 18、GLUE：提供 Web IDE，支持在线开发任务逻辑代码，动态发布，实时编译生效，省略部署上线的过程。支持 30 个版本的历史版本回溯。
- 19、脚本任务：支持以 GLUE 模式开发和运行脚本任务，包括 Shell、Python、NodeJS、PHP、PowerShell 等类型脚本;

- 20、命令行任务：原生提供通用命令行任务 Handler（Bean 任务，”CommandJobHandler”）；业务方只需要提供命令行即可；
- 21、任务依赖：支持配置子任务依赖，当父任务执行结束且执行成功后将会主动触发一次子任务的执行, 多个子任务用逗号分隔；
- 22、一致性：“调度中心”通过 DB 锁保证集群分布式调度的一致性, 一次任务调度只会触发一次执行；
- 23、自定义任务参数：支持在线配置调度任务入参，即时生效；
- 24、调度线程池：调度系统多线程触发调度运行，确保调度精确执行，不被堵塞；
- 25、数据加密：调度中心和执行器之间的通讯进行数据加密，提升调度信息安全性；
- 26、邮件报警：任务失败时支持邮件报警，支持配置多邮件地址群发报警邮件；
- 27、推送 maven 中央仓库: 将会把最新稳定版推送到 maven 中央仓库, 方便用户接入和使用;
- 28、运行报表：支持实时查看运行数据，如任务数量、调度次数、执行器数量等；以及调度报表，如调度日期分布图，调度成功分布图等；
- 29、全异步：任务调度流程全异步化设计实现，如异步调度、异步运行、异步回调等，有效对密集调度进行流量削峰，理论上支持任意时长任务的运行；
- 30、跨语言：调度中心与执行器提供语言无关的 RESTful API 服务，第三方任意语言可据此对接调度中心或者实现执行器。除此之外，还提供了 “多任务模式”和“httpJobHandler”等其他跨语言方案；
- 31、国际化：调度中心支持国际化设置，提供中文、英文两种可选语言，默认为中文；
- 32、容器化：提供官方 docker 镜像，并实时更新推送 dockerhub，进一步实现产品开箱即用；
- 33、线程池隔离：调度线程池进行隔离拆分，慢任务自动降级进入”Slow”线程池，避免耗尽调度线程，提高系统稳定性；
- 34、用户管理：支持在线管理系统用户，存在管理员、普通用户两种角色；
- 35、权限控制：执行器维度进行权限控制，管理员拥有全量权限，普通用户需要分配执行器权限后才允许相关操作；

2.源码下载

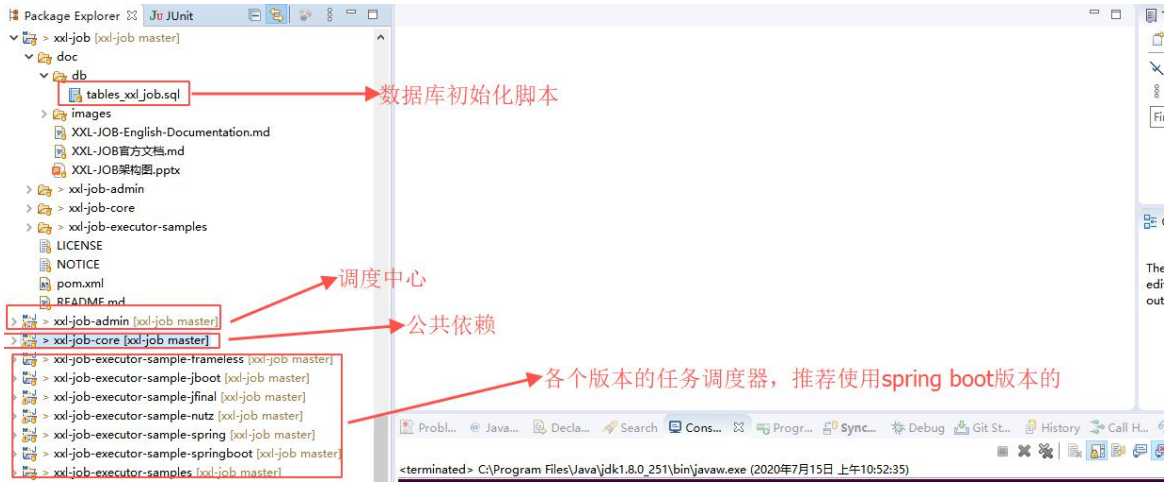
https://github.com/xuxueli/xxl-job

3.文档地址

http://www.xuxueli.com/xxl-job/#/

4.源码结构

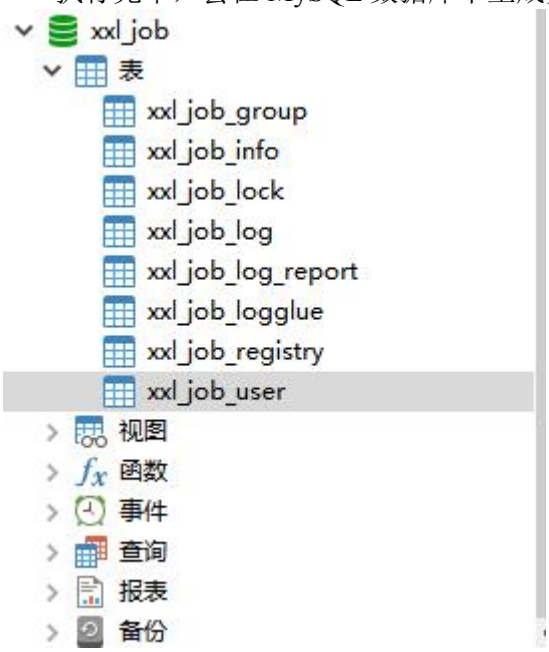
通过上面给出的源码下载地址，我们将源码 clone 到 IDE 中，如下：



5.初始化数据库

初始化脚本在上面源码目录的 /doc/db/tables_xxl_job.sql ，将此脚本在 MySQL 数据库中执行一遍。

执行完毕，会在 MySQL 数据库中生成如下 8 张表：



6.配置调度中心

调度中心就是源码中的 xxl-job-admin 工程，我们需要将其配置成自己需要的调度中心，通过该工程我们能够以图形化的方式统一管理任务调度平台上调度任务，负责触发调度执行。

a.**修改调度中心配置文件**

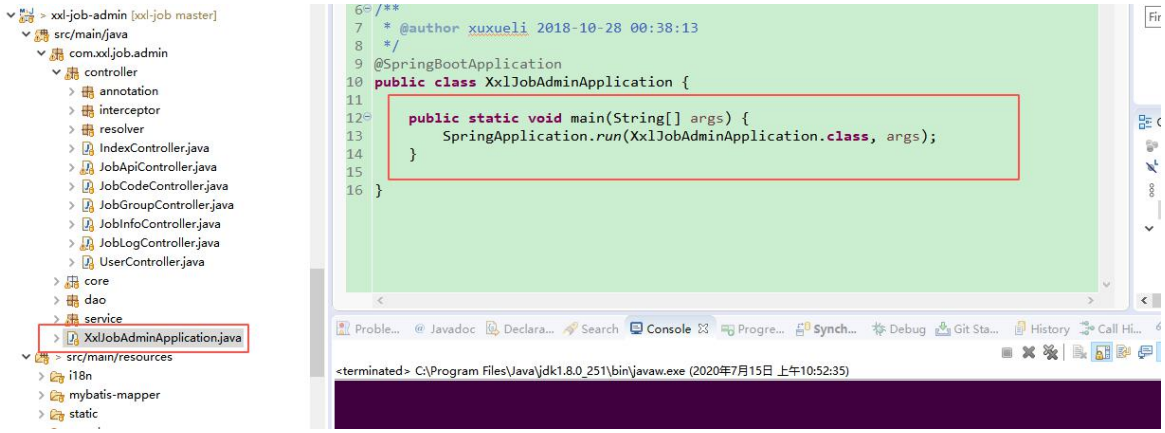
文件地址：/xxl-job/xxl-job-admin/src/main/resources/xxl-job-admin.properties

```
24
25 ### xxl-job, datasource
26 spring.datasource.url=
27 spring.datasource.username=
28 spring.datasource.password=
29 spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
30
31 xxl.job.login.username=admin
32 xxl.job.login.password=123456
33
```

先配置数据库信息，然后再添加用户名*密码（登录控制台使用）

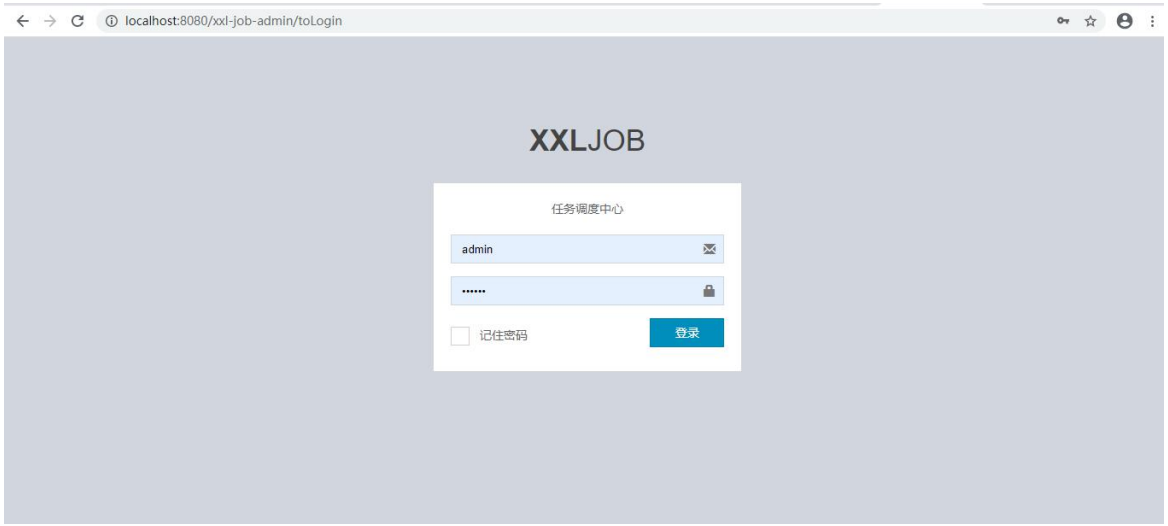
b.部署调度中心

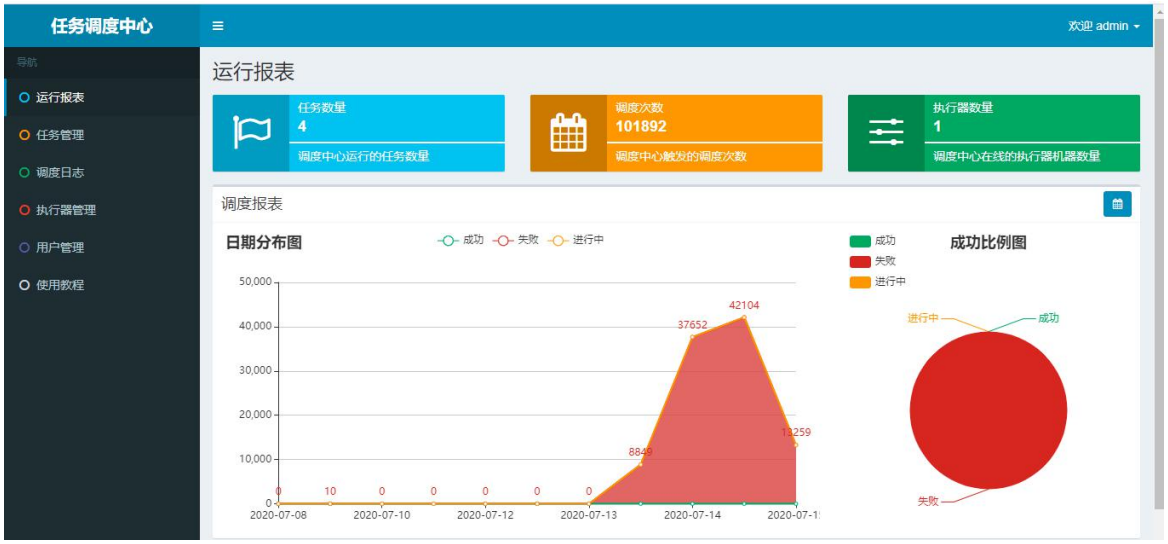
该工程是一个 springboot 项目，我们只需要在 IDEA 中执行 XxlJobAdminApplication 类即可运行该工程



c.访问调度中心管理界面

在浏览器输入 http://localhost:8080/xxl-job-admin 然后输入用户名和密码（前面配置文件中配置的），即可看到如下管理界面。





7.项目（自己的项目）改造

a.添加 pom

```
1.    <!-- xxl-job-->
2.        <dependency>
3.            <groupId>com.xuxueli</groupId>
4.            <artifactId>xxl-job-core</artifactId>
5.            <version>2.2.0</version>
6.        </dependency>
```

b.**配置执行器**

```
xxl:
  job:
    accessToken: ''
    admin:
      addresses: http://172.16.181.31:18899/xxl-job-admin
    executor:
      address: ''
      appname: edu-course-boot
      ip: ''
      logpath: /data/applogs/xxl-job/jobhandler
      logretentiondays: 30
      port: 19999
```

c.在项目中创建 XxlJobConfig.java 文件：

```
1.  package com.xxl.job.executor.core.config;
2.
3. import com.xxl.job.core.executor.impl.XxlJobSpringExecutor;
4. import org.slf4j.Logger;
5. import org.slf4j.LoggerFactory;
6. import org.springframework.beans.factory.annotation.Value;
7. import org.springframework.context.annotation.Bean;
8. import org.springframework.context.annotation.Configuration;
```

```
9.
10. /**
11.  * xxl-job config
12.  *
13.  * @author xuxueli 2017-04-28
14.  */
15. @Configuration
16. public class XxlJobConfig {
17.     private Logger logger = LoggerFactory.getLogger(XxlJobConfig.class);
18.
19.     @Value("${xxl.job.admin.addresses}")
20.     private String adminAddresses;
21.
22.     @Value("${xxl.job.accessToken}")
23.     private String accessToken;
24.
25.     @Value("${xxl.job.executor.appname}")
26.     private String appname;
27.
28.     @Value("${xxl.job.executor.address}")
29.     private String address;
30.
31.     @Value("${xxl.job.executor.ip}")
32.     private String ip;
33.
34.     @Value("${xxl.job.executor.port}")
35.     private int port;
36.
37.     @Value("${xxl.job.executor.logpath}")
38.     private String logPath;
39.
40.     @Value("${xxl.job.executor.logretentiondays}")
41.     private int logRetentionDays;
42.
43.
44.     @Bean
45.     public XxlJobSpringExecutor xxlJobExecutor() {
46.         logger.info(">>>>>>>>>>> xxl-job config init.");
47.         XxlJobSpringExecutor xxlJobSpringExecutor = new XxlJobSpringExecutor();
48.         xxlJobSpringExecutor.setAdminAddresses(adminAddresses);
49.         xxlJobSpringExecutor.setAppname(appname);
50.         xxlJobSpringExecutor.setAddress(address);
51.         xxlJobSpringExecutor.setIp(ip);
52.         xxlJobSpringExecutor.setPort(port);
53.         xxlJobSpringExecutor.setAccessToken(accessToken);
54.         xxlJobSpringExecutor.setLogPath(logPath);
55.         xxlJobSpringExecutor.setLogRetentionDays(logRetentionDays);
56.
57.         return xxlJobSpringExecutor;
58.     }
59.
```

```
60.     /**
61.      * 针对多网卡、容器内部署等情况，可借助 "spring-cloud-commons" 提供的
"InetUtils" 组件灵活定制注册 IP；
62.      *
63.      *     1、引入依赖：
64.      *         <dependency>
65.      *             <groupId>org.springframework.cloud</groupId>
66.      *             <artifactId>spring-cloud-commons</artifactId>
67.      *             <version>${version}</version>
68.      *         </dependency>
69.      *
70.      *     2、配置文件，或者容器启动变量
71.      *         spring.cloud.inetutils.preferred-networks: 'xxx.xxx.xxx.'
72.      *
73.      *     3、获取 IP
74.      *         String ip_ = inetUtils.findFirstNonLoopbackHostInfo().getIpAddress();
75.      */
76.
77.
78. }
```

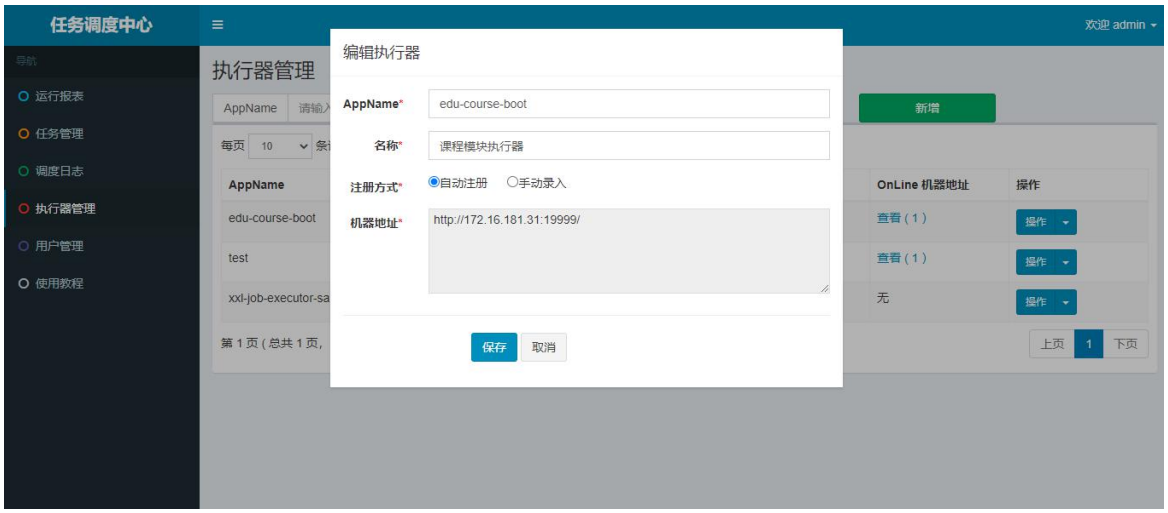
d.在项目中创建一个 Handler，用于执行我们想要执行的东西，这里我只是简单的打印一行日志：

```
1. @Component
2. public class CourseAutoOnlineJob {
3.
4.     @Autowired
5.     private ICourseService courseService;
6.
7.     /**
8.      * 1、简单任务示例（Bean 模式）
9.      */
10.     @XxlJob("courseAutoOnlineJobHandler")
11.     public ReturnT<String> execute(String param) throws Exception {
12.         XxlJobLogger.log("Execute course auto online.");
13.         courseService.courseAutoOnline();
14.         return ReturnT.SUCCESS;
15.     }
16. }
```

8.调度中心配置执行器

a.配置执行器

点击 执行器管理----》新增执行器---》，如下如下界面，然后填充此表格，点击**保存**即可。



下面是对这几个参数的介绍：

AppName：是每个执行器集群的唯一标示 AppName, 执行器会周期性以 AppName 为对象进行自动注册。可通过该配置自动发现注册成功的执行器, 供任务调度时使用;

名称：执行器的名称, 因为 AppName 限制字母数字等组成,可读性不强, 名称为了提高执行器的可读性;

排序: 执行器的排序, 系统中需要执行器的地方,如任务新增, 将会按照该排序读取可用的执行器列表;

注册方式：调度中心获取执行器地址的方式，

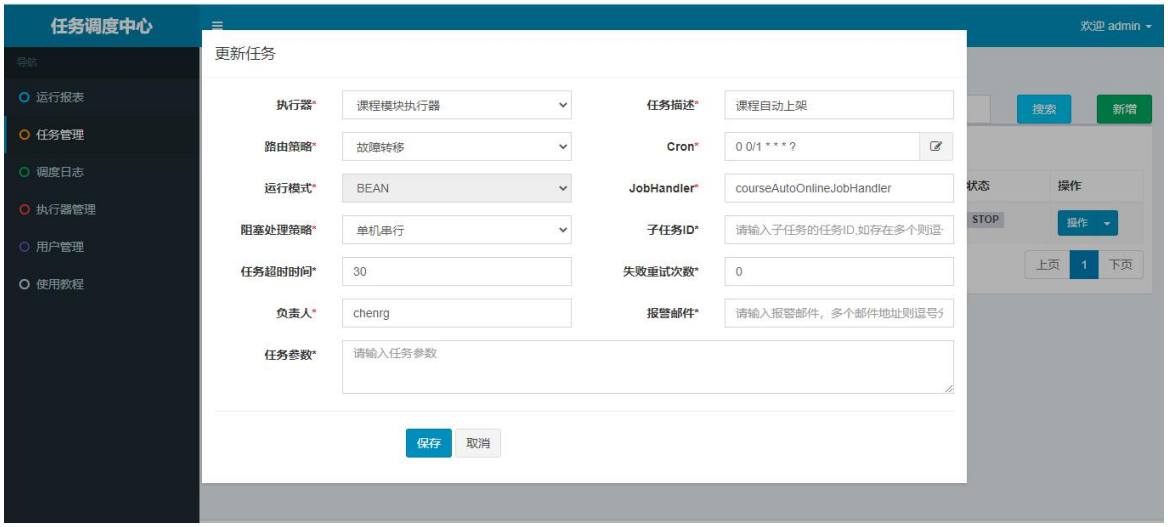
自动注册：执行器自动进行执行器注册，调度中心通过底层注册表可以动态发现执行器机器地址；

手动录入：人工手动录入执行器的地址信息，多地址逗号分隔，供调度中心使用；

机器地址："注册方式"为"手动录入"时有效，支持人工维护执行器的地址信息；

b.**创建任务**

点击 任务管理---》新增任务---》



执行器：任务的绑定的执行器，任务触发调度时将会自动发现注册成功的执行器, 实现任务自动发现功能; 另一方面也可以方便的进行任务分组。每个任务必须绑定一个执行器, 可

在 "执行器管理" 进行设置。

任务描述：任务的描述信息，便于任务管理；

路由策略：当执行器集群部署时，提供丰富的路由策略，包括；

FIRST（第一个）：固定选择第一个机器；

LAST（最后一个）：固定选择最后一个机器；

ROUND（轮询）:；

RANDOM（随机）：随机选择在线的机器；

CONSISTENT_HASH（一致性 HASH）：每个任务按照 Hash 算法固定选择某一台机器，且所有任务均匀散列在不同机器上。

LEAST_FREQUENTLY_USED（最不经常使用）：使用频率最低的机器优先被选举；

LEAST_RECENTLY_USED（最近最久未使用）：最久为使用的机器优先被选举；

FAILOVER（故障转移）：按照顺序依次进行心跳检测，第一个心跳检测成功的机器选定为目标执行器并发起调度；

BUSYOVER（忙碌转移）：按照顺序依次进行空闲检测，第一个空闲检测成功的机器选定为目标执行器并发起调度；

SHARDING_BROADCAST(分片广播)：广播触发对应集群中所有机器执行一次任务，同时系统自动传递分片参数；可根据分片参数开发分片任务；

Cron：触发任务执行的 Cron 表达式；

运行模式：

BEAN 模式：任务以 JobHandler 方式维护在执行器端；需要结合 "JobHandler" 属性匹配执行器中任务；

GLUE 模式(Java)：任务以源码方式维护在调度中心；该模式的任务实际上是一段继承自 IJobHandler 的 Java 类代码并 "groovy" 源码方式维护，它在执行器项目中运行，可使用@Resource/@Autowire 注入执行器里中的其他服务；

GLUE 模式(Shell)：任务以源码方式维护在调度中心；该模式的任务实际上是一段 "shell" 脚本；

GLUE 模式(Python)：任务以源码方式维护在调度中心；该模式的任务实际上是一段 "python" 脚本；

GLUE 模式(PHP)：任务以源码方式维护在调度中心；该模式的任务实际上是一段 "php" 脚本；

GLUE 模式(NodeJS)：任务以源码方式维护在调度中心；该模式的任务实际上是一段 "nodejs" 脚本；

GLUE 模式(PowerShell)：任务以源码方式维护在调度中心；该模式的任务实际上是一段 "PowerShell" 脚本；

JobHandler：运行模式为 "BEAN 模式" 时生效，对应执行器中新开发的 JobHandler 类“@JobHandler”注解自定义的 value 值；

阻塞处理策略：调度过于密集执行器来不及处理时的处理策略；

单机串行（默认）：调度请求进入单机执行器后，调度请求进入 FIFO 队列并以串行方式运行；

丢弃后续调度：调度请求进入单机执行器后，发现执行器存在运行的调度任务，本次请求将会被丢弃并标记为失败；

覆盖之前调度：调度请求进入单机执行器后，发现执行器存在运行的调度任务，将会终止运行中的调度任务并清空队列，然后运行本地调度任务；

子任务：每个任务都拥有一个唯一的任务 ID(任务 ID 可以从任务列表获取)，当本任务执行结束并且执行成功时，将会触发子任务 ID 所对应的任务的一次主动调度。

任务超时时间：支持自定义任务超时时间，任务运行超时将会主动中断任务；

失败重试次数；支持自定义任务失败重试次数，当任务失败时将会按照预设的失败重试次数主动进行重试；

报警邮件：任务调度失败时邮件通知的邮箱地址，支持配置多邮箱地址，配置多个邮箱地址时用逗号分隔；

负责人：任务的负责人；

执行参数：任务执行所需的参数，多个参数时用逗号分隔，任务执行时将会把多个参数转换成数组传入；

9.启动任务

配置完执行器以及任务，我们只需要启动该任务，便可以运行了。