

《XXL-JOB分布式任务调度》

学习目标

- 1.理解分布式任务调度
- 2.理解XXL-JOB的概念
- 3.了解XXL-JOB的优势和特点
- 4.理解XXL-JOB任务调度的整体流程
- 5.掌握快速入门案例
- 6.理解XXL-JOB的架构设计
- 7.理解XXL-JOB的执行原理
- 8.掌握XXL-JOB管理界面的常用操作
- 9.掌握XXL-JOB的实战应用
- 10.掌握XXL-JOB的高级应用

一、简介

1.1 什么是任务调度

我们可以先思考一下下面业务场景的解决方案：

- 某网站为了实时展示天气，每隔5分钟就去天气服务器获取最新的天气信息。
- 某电商系统需要在每天上午10点，下午3点，晚上8点发放一批优惠券。
 - 12306网站会根据车次不同，设置几个时间点分批次放票。

类似的场景还有很多，我们该如何解决？

以上这些场景，就是任务调度所需要解决的问题。

任务调度是指系统为了自动完成特定任务，在约定的时刻去执行某个任务的过程。有了任务调度就可以解放更多的人力，交由系统去自动执行任务。

任务调度如何实现？

多线程方式实现：

学过多线程的同学，可能会想到，我们可以开启一个线程，每sleep一段时间，就去检查是否已到预期执行时间。

以下代码简单实现了任务调度的功能：

```
public static void main(String[] args) {
    //任务执行间隔时间
    final long timeInterval = 1000;
    Runnable runnable = new Runnable() {
        public void run() {
```

```

        while (true) {
            //TODO: something
            try {
                Thread.sleep(timeInterval);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
};
Thread thread = new Thread(runnable);
thread.start();
}

```

上面的代码实现了按一定的间隔时间执行任务调度的功能。

Jdk也为我们提供了相关支持，如Timer、ScheduledExecutor，下边我们了解下。

Timer方式实现：

```

public static void main(String[] args){
    Timer timer = new Timer();
    timer.schedule(new TimerTask(){
        @Override
        public void run() {
            //TODO: something
        }
    }, 1000, 2000); //1秒后开始调度，每2秒执行一次
}

```

Timer 的优点在于简单易用，每个Timer对应一个线程，因此可以同时启动多个Timer并行执行多个任务，同一个Timer中的任务是串行执行。

ScheduledExecutor方式实现：

```

public static void main(String [] agrs){
    ScheduledExecutorService service = Executors.newScheduledThreadPool(10);
    service.scheduleAtFixedRate(
        new Runnable() {
            @Override
            public void run() {
                //TODO: something
                System.out.println("todo something");
            }
        }, 1,
        2, TimeUnit.SECONDS);
}

```

Java 5 推出了基于线程池设计的 ScheduledExecutor，其设计思想是，每一个被调度的任务都会由线程池中一个线程去执行，因此任务是并发执行的，相互之间不会受到干扰。

Timer 和 ScheduledExecutor 都仅提供基于开始时间与重复间隔的任务调度，不能胜任更加复杂的调度需求。比如，设置每月第一天凌晨1点执行任务、复杂调度任务的管理、任务间传递数据等等。

Quartz 是一个功能强大的任务调度框架，它可以满足更多更复杂的调度需求，Quartz 设计的核心类包括 Scheduler, Job 以及 Trigger。其中，Job 负责定义需要执行的任务，Trigger 负责设置调度策略，Scheduler 将二者组装在一起，并触发任务开始执行。Quartz 支持简单的按时间间隔调度、还支持按日历调度方式，通过设置 CronTrigger 表达式（包括：秒、分、时、日、月、周、年）进行任务调度。

第三方 Quartz 方式实现：

```
public static void main(String [] args) throws SchedulerException {
    //创建一个Scheduler
    SchedulerFactory schedulerFactory = new StdSchedulerFactory();
    Scheduler scheduler = schedulerFactory.getScheduler();
    //创建JobDetail
    JobBuilder jobDetailBuilder = JobBuilder.newJob(MyJob.class);
    jobDetailBuilder.withIdentity("jobName", "jobGroupName");
    JobDetail jobDetail = jobDetailBuilder.build();
    //创建触发的CronTrigger 支持按日历调度
    CronTrigger trigger = TriggerBuilder.newTrigger()
        .withIdentity("triggerName", "triggerGroupName")
        .startNow()
        .withSchedule(CronScheduleBuilder.cronSchedule("0/2 * * * * ?"))
        .build();
    //创建触发的SimpleTrigger 简单的间隔调度
    /*SimpleTrigger trigger = TriggerBuilder.newTrigger()
        .withIdentity("triggerName", "triggerGroupName")
        .startNow()
        .withSchedule(SimpleScheduleBuilder
            .simpleSchedule()
            .withIntervalInSeconds(2)
            .repeatForever())
        .build();*/
    scheduler.scheduleJob(jobDetail, trigger);
    scheduler.start();
}

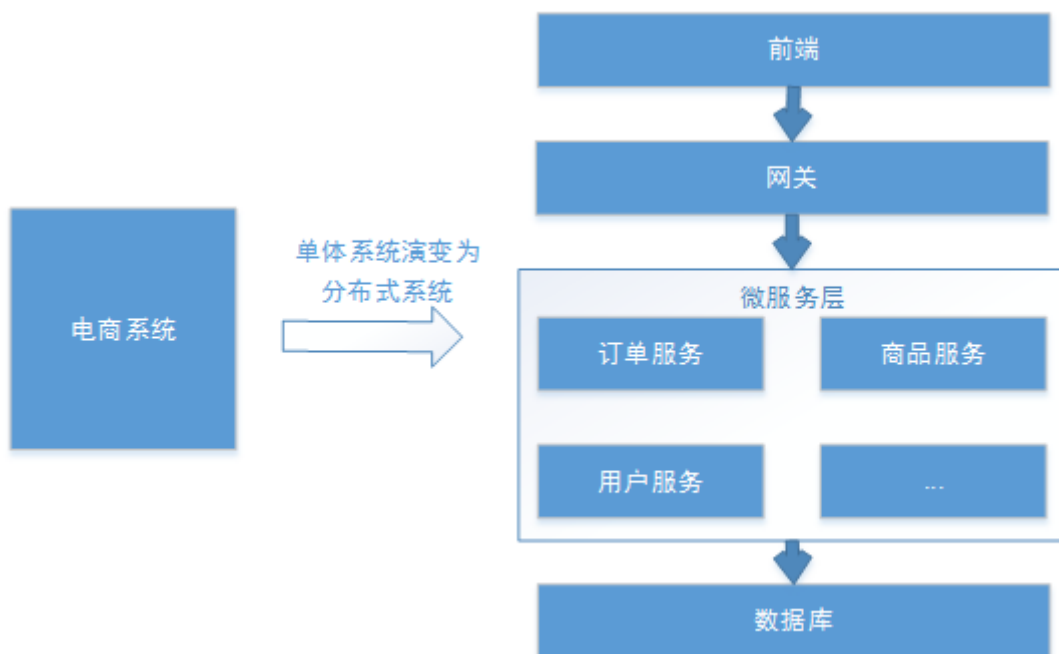
public class MyJob implements Job {
    @Override
    public void execute(JobExecutionContext jobExecutionContext){
        System.out.println("todo something");
    }
}
```

通过以上内容我们学习了什么是任务调度，任务调度所解决的问题，以及任务调度的多种实现方式。

1.2 什么是分布式的任务调度

什么是分布式？

软件架构正在逐步从单体应用转变为分布式架构，将单体结构分为若干服务，服务之间通过网络交互来完成用户的业务处理，如下图，电商系统为分布式架构，由订单服务、商品服务、用户服务等组成：

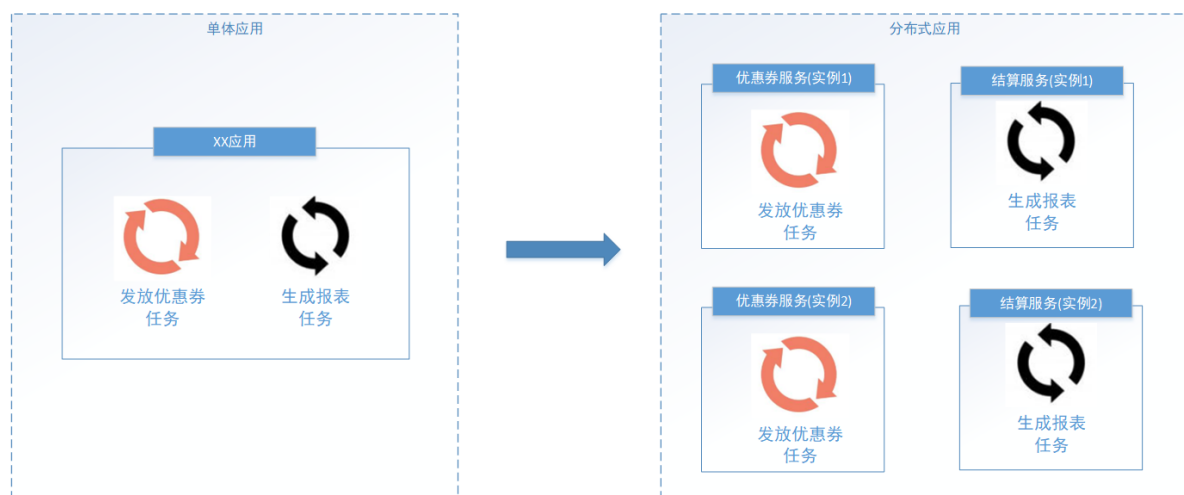


分布式系统具体如下基本特点：

- 1、分布性：每个部分都可以独立部署，服务之间交互通过网络进行通信，比如：订单服务、商品服务。
- 2、伸缩性：每个部分都可以集群方式部署，并可针对部分结点进行硬件及软件扩容，具有一定的伸缩能力。
- 3、高可用：每个部分都可以集群部署，保证高可用。

什么是分布式任务调度？

通常任务调度的程序是集成在应用中的，比如：优惠券服务中包括了定时发放优惠券的的调度程序，结算服务中包括了定期生成报表的任务调度程序，如果采用分布式架构，一个服务往往会部署多个冗余实例来运行我们的业务，在这种分布式系统环境下运行任务调度，我们称之为**分布式任务调度**，如下图：



分布式任务调度要实现的目标：

不管是集成在应用程序中，还是单独构建的任务调度系统，如果采用分布式的方式，那任务调度程序就可以具有分布式系统的特点，任务调度处理能力也会得到大大提升：

1、并行任务调度

并行任务调度实现靠多线程，如果有大量任务需要调度，此时光靠多线程就会有瓶颈了，因为一台计算机CPU的处理能力是有限的。

如果将任务调度程序分布式部署，每个结点还可以部署为集群，这样就可以让多台计算机共同去完成任务调度，我们可以将任务分割为若干个分片，由不同的实例并行执行，来提高任务调度的处理效率。

2、高可用

若某一个实例宕机，不影响其他实例来执行任务。

3、弹性扩容

当集群中增加实例就可以提高并执行任务的处理效率。

4、任务管理与监测

对系统中存在的所有定时任务进行统一的管理及监测。让开发人员及运维人员能够时刻了解任务执行情况，从而做出快速的应急处理响应。

5、避免任务重复执行

当任务调度以集群方式部署，同一个任务调度可能会执行多次，比如在上面提到的电商系统中到点发优惠券的例子，就会发放多次优惠券，对公司造成很多损失，所以我们需要控制相同的任务在多个运行实例上只执行一次。

1.3 XXL-JOB概述

XXL-JOB是一个轻量级分布式任务调度框架，其核心设计目标是开发迅速、学习简单、轻量级、易扩展。现已开放源代码并接入多家公司线上产品线，开箱即用。

1.4 XXL-JOB的优势

1.4.1 同类框架的比较

	quartz	elastic-job	LTS	xxl-job
依赖	MySQL、jdk	jdk、zookeeper	jdk、zookeeper、maven	mysql、jdk
高可用	多节点部署，通过竞争数据库锁来保证只有一个节点执行任务	通过zookeeper的注册与发现，可以动态的添加服务器	集群部署,可以动态的添加服务器。可以手动增加定时任务，启动和暂停任务。有监控	基于竞争数据库锁保证只有一个节点执行任务，支持水平扩容。可以手动增加定时任务，启动和暂停任务，有监控
任务分片	×	√	√	√
管理界面	×	√	√	√
难易程度	简单	简单	略复杂	简单

度	quartz	elastic-job	LTS	xxl-job
高级功能	-	弹性扩容，多种作业模式，失效转移，运行状态收集，多线程处理数据，幂等性，容错处理，spring命名空间支持	支持spring，spring boot，业务日志记录器，SPI扩展支持，故障转移，节点监控，多样化任务执行结果支持，FailStore容错，动态扩容。	弹性扩容，分片广播，故障转移，Rolling实时日志，GLUE（支持在线编辑代码，免发布），任务进度监控，任务依赖，数据加密，邮件报警，运行报表，国际化
版本更新	半年没更新	2年没更新	1年没更新	更新频繁

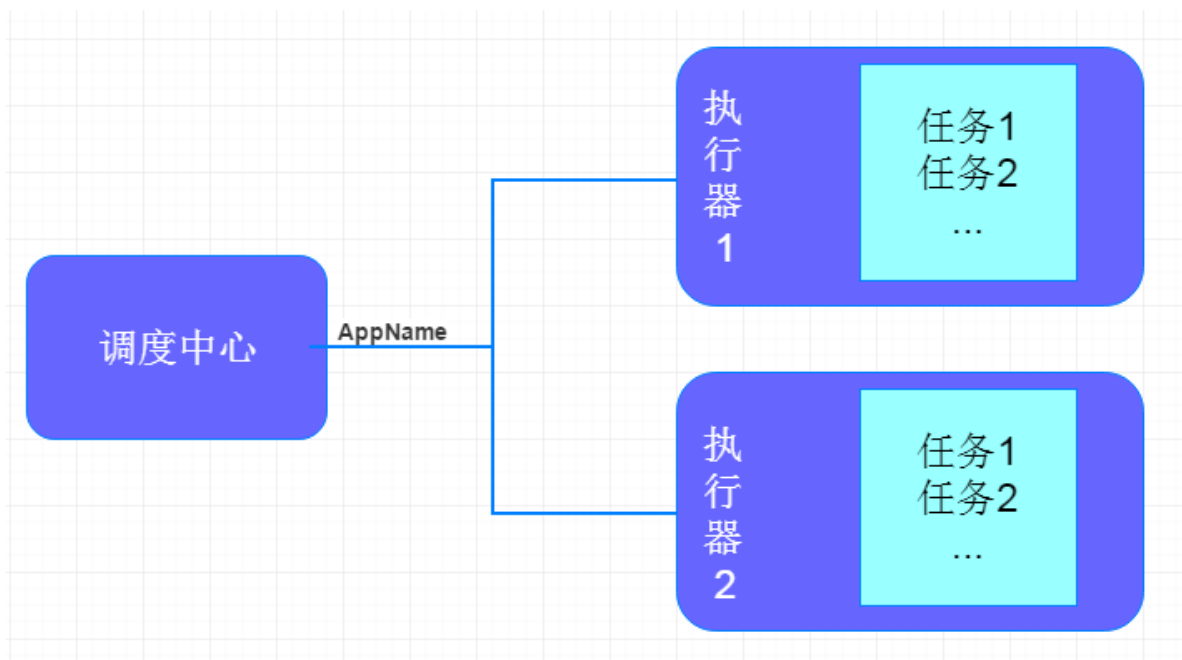
1.4.2 XXL-JOB主要功能特性

- **简单灵活** 提供Web页面对任务进行管理，管理系统支持用户管理、权限控制；支持容器部署；支持通过通用HTTP提供跨平台任务调度；
- **丰富的任务管理功能** 支持页面对任务CRUD操作；支持在页面编写脚本任务、命令行任务、Java代码任务并执行；支持任务级联编排，父任务执行结束后触发子任务执行；支持设置指定任务执行节点路由策略，包括轮询、随机、广播、故障转移、忙碌转移等；支持Cron方式、任务依赖、调度中心API接口方式触发任务执行
- **高性能** 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；
- **高可用** 任务调度中心、任务执行节点均 集群部署，支持动态扩展、故障转移 支持任务配置路由故障转移策略，执行器节点不可用是自动转移到其他节点执行 支持任务超时控制、失败重试配置 支持任务处理阻塞策略：调度当任务执行节点忙碌时来不及执行任务的处理策略，包括：串行、抛弃、覆盖策略
- **易于监控运维** 支持设置任务失败邮件告警，预留接口支持短信、钉钉告警；支持实时查看任务执行运行数据统计图表、任务进度监控数据、任务完整执行日志；

二、快速入门

2.1 XXL-JOB的整体组成

首先，我们需要了解XXL-JOB的系统组成，主要有调度中心、执行器、任务：



调度中心：负责管理调度信息，按照调度配置发出调度请求，自身不承担业务代码。

任务执行器：负责接收调度请求并执行任务逻辑。

任务：专注于任务的处理。

调度中心会发出调度请求，任务执行器接收到请求之后会去执行任务，任务则专注于任务业务的处理。

2.2 下载导入

2.2.1 源码仓库地址

- ①、GitHub: <https://github.com/xuxueli/xxl-job>
- ②、码云: <https://gitee.com/xuxueli0323/xxl-job>

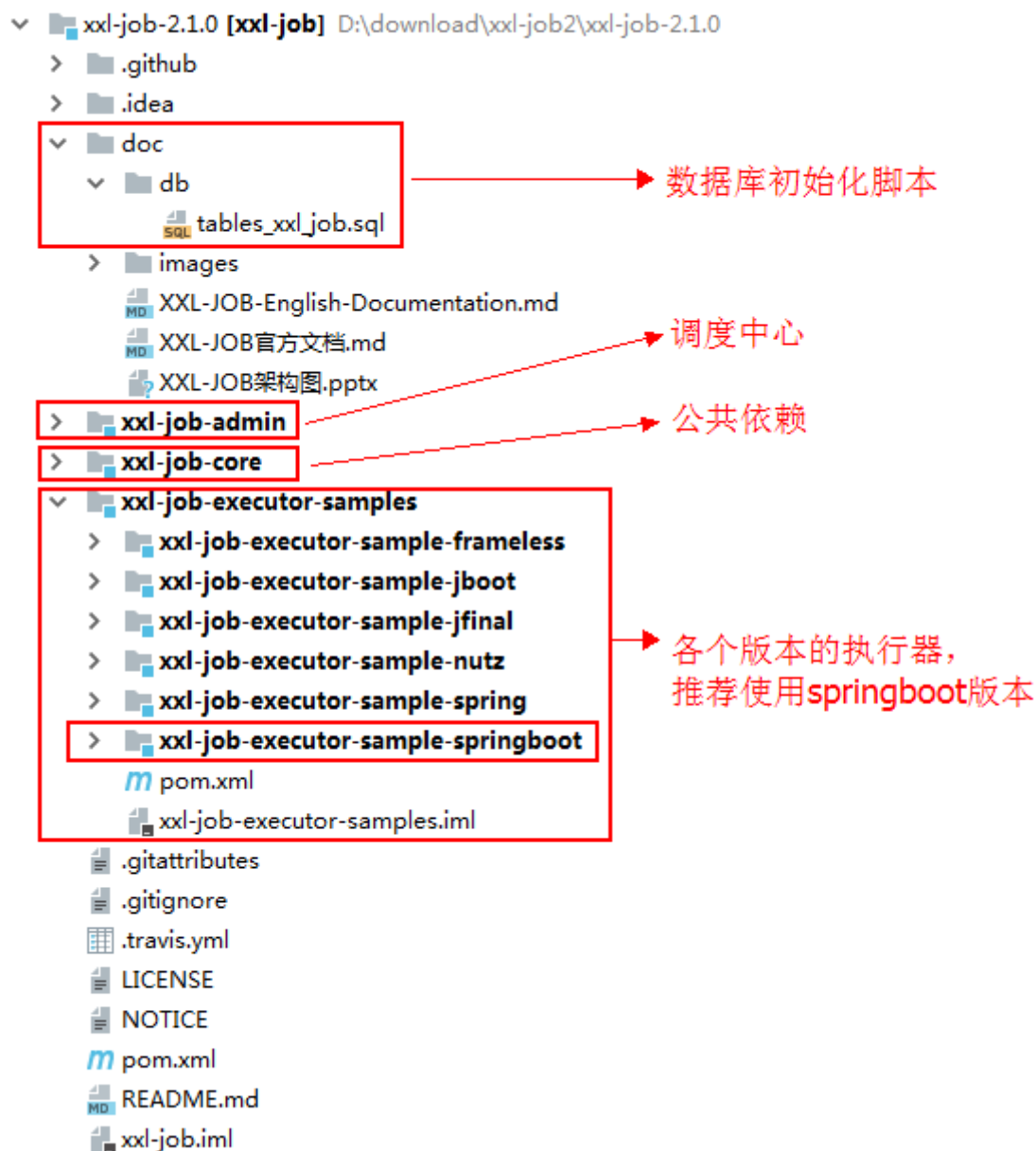
2.2.2 环境

XXL-JOB给予的环境如下：

- Maven3+
- Jdk1.7+
- Mysql5.6+

2.2.3 导入项目到idea

解压下载的源码,按照maven格式将源码导入IDE, 使用maven进行编译即可，源码结构如下：



xxl-job-admin: 调度中心 xxl-job-core: 公共依赖 xxl-job-executor-samples: 执行器Sample示例 (选择合适的版本执行器, 可直接使用) : xxl-job-executor-sample-springboot: Springboot版本, 通过Springboot管理执行器, 推荐这种方式; : xxl-job-executor-sample-spring: Spring版本, 通过Spring容器管理执行器, 比较通用; : xxl-job-executor-sample-frameless: 无框架版本; : xxl-job-executor-sample-jfinal: JFinal版本, 通过JFinal管理执行器; : xxl-job-executor-sample-nutz: Nutz版本, 通过Nutz管理执行器;

doc :文档资料, 包含数据库脚本

2.2.4 导入数据库脚本

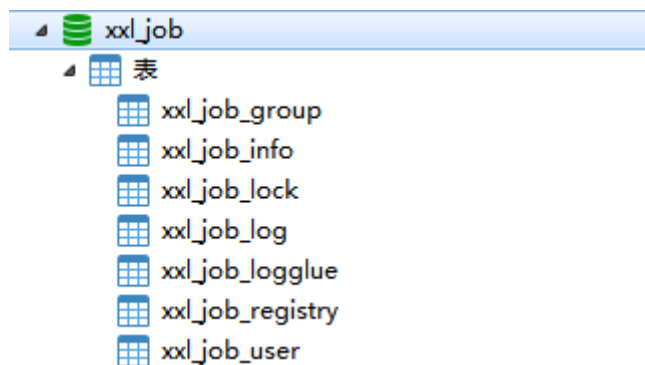
数据库脚本为XXL-JOB官方提供的初始化脚本。

(1) 脚本文件位置

SQL脚本请参考授课资料“数据库脚本”文件夹中提供的tables_xxl_job.sql文件。

(2) 导入

导入后, 正常情况下应该生成7张表。



XXL-JOB调度数据库表说明如下：

- xxl_job_lock: 任务调度锁表;
- xxl_job_group: 执行器信息表, 维护任务执行器信息;
- xxl_job_info: 调度扩展信息表: 用于保存XXL-JOB调度任务的扩展信息, 如任务分组、任务名、机器地址、执行器、执行入参和报警邮件等等;
- xxl_job_log: 调度日志表: 用于保存XXL-JOB任务调度的历史信息, 如调度结果、执行结果、调度入参、调度机器和执行器等等;
- xxl_job_logglue: 任务GLUE日志: 用于保存GLUE更新历史, 用于支持GLUE的版本回溯功能;
- xxl_job_registry: 执行器注册表, 维护在线的执行器和调度中心机器地址信息;
- xxl_job_user: 系统用户表;

2.2.5 配置调度中心项目

调度中心项目: xxl-job-admin 作用: 统一管理任务调度平台上调度任务, 负责触发调度执行, 并且提供任务管理平台。

(1) 配置文件位置

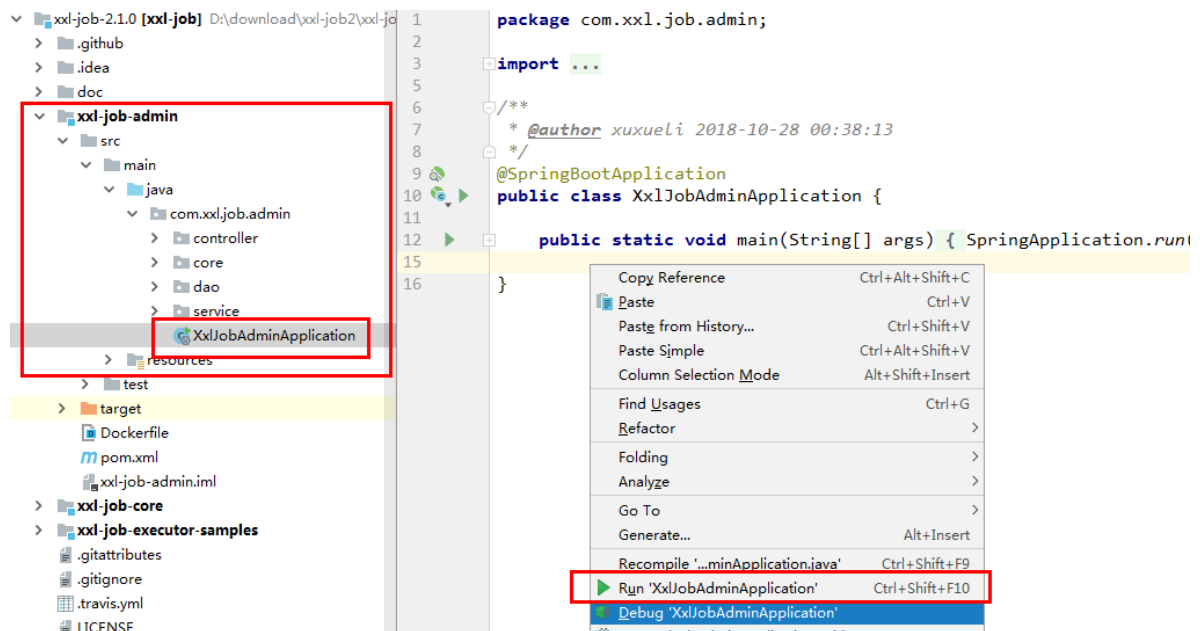
请参考授课资料中的文件夹“调度中心配置”中的application.properties。

(2) 配置内容说明

需要修改的只有数据库配置, 其他的可使用默认的。

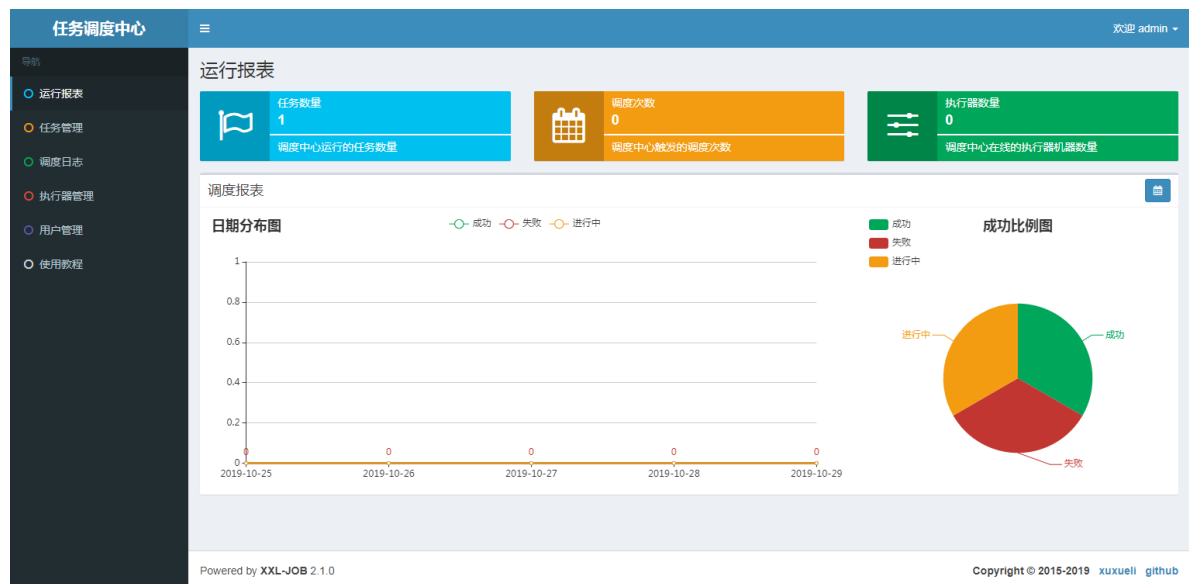
```
### xxl-job, datasource
spring.datasource.url=jdbc:mysql://127.0.0.1:3306/xxl_job?
serverTimezone=Asia/Shanghai&useUnicode=true&characterEncoding=utf8&useSSL=false
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

(3) 如果已经正确进行上述配置, 在本地启动xxl-job-admin项目



(4) 调度中心浏览器访问地址: <http://localhost:8080/xxl-job-admin>

用户名/密码: admin/123456, 登录后运行界面如下图所示:



至此“调度中心”项目已经配置启动完毕, 然后需要配置启动执行器项目。

2.2.6 配置执行器项目

执行器项目: 使用官方推荐的springboot版本 xxl-job-executor-sample-springboot 作用: 负责接收“调度中心”的调度请求并执行任务;

(1) 目标: 为了测试分布式任务调度的一些特点, 比如并行任务调度和高可用, 我们需要配置两个执行器。

此处我们采用在Idea配置启动参数的方式, 模拟两个执行器实例, 配置文件参考授课资料“执行器配置”文件夹中的application.properties文件。

执行器配置规划如下:

执行器端口不同, web端口不同, 执行器AppName保持一致, 调度中心地址一致。

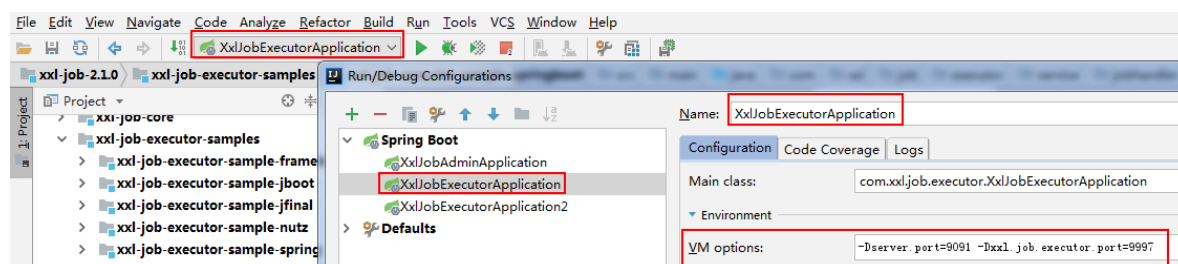
- web端口: 9091和9092
- 执行器端口: 9997和9998
- 执行器AppName: xxl-job-executor-sample
- 调度中心地址: <http://127.0.0.1:8080/xxl-job-admin>

(2) 启动参数配置如下：

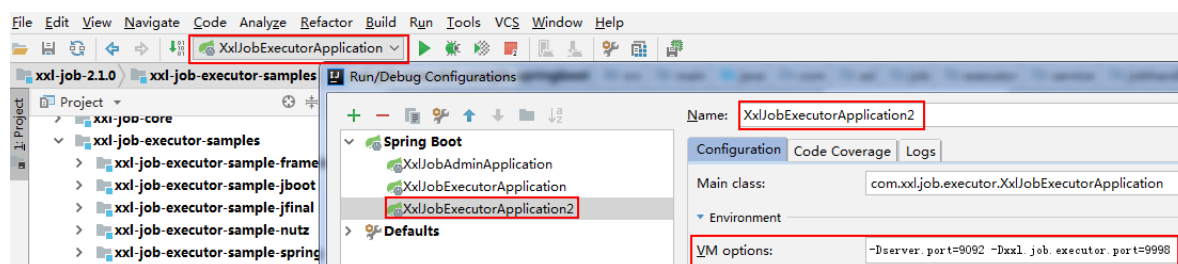
```
-Dserver.port=9091 -Dxxl.job.executor.port=9997 -Dxxl.job.executor.appname=xxl-job-executor-sample -Dxxl.job.admin.addresses=http://127.0.0.1:8080/xxl-job-admin
```

```
-Dserver.port=9092 -Dxxl.job.executor.port=9998 -Dxxl.job.executor.appname=xxl-job-executor-sample -Dxxl.job.admin.addresses=http://127.0.0.1:8080/xxl-job-admin
```

执行器1启动参数配置：



执行器2启动参数配置：



至此“执行器”项目已经配置完毕，接下来可以创建任务了。

2.3 开发第一个任务

目标：实现一个简单的分布式任务调度的开发

说明：如果完成了数据库的初始化，那么在任务调度中心的执行器管理列表中，默认有一个创建好的示例执行器：xxl-job-executor-sample，我们测试时就是用这个执行器。



2.3.1 新增任务

本示例新建一个“Bean模式任务”，需要在执行器项目中开发业务代码。

(1) 直接使用springboot项目提供的示例代码DemoJobHandler，然后启动执行器项目。



(2) 登录调度中心，进入任务管理，点击下图所示“新增”按钮，新增任务。



新建一个BEAN运行模式的任务，参考下面截图中任务的参数配置：

执行器选择示例执行器，**JobHandler**指定为demoJobHandler（这个值和执行器代码中 @JobHandler(value="demoJobHandler")这个注解的值是对应的）。

新增

执行器*	示例执行器	任务描述*	测试任务2
路由策略*	第一个	Cron*	0/5 * * * * ?
运行模式*	BEAN	JobHandler*	demoJobHandler
阻塞处理策略*	单机串行	子任务ID*	请输入子任务的任务ID,如存在多个则逗号分隔
任务超时时间*	任务超时时间, 单位秒, 大于零时生效	失败重试次数*	失败重试次数, 大于零时生效
负责人*	XXL	报警邮件*	请输入报警邮件, 多个邮件地址则逗号分隔
任务参数*	请输入任务参数		

保存

取消

(3) 点击保存，即完成任务创建。

任务管理						
执行器	示例执行器	全	请输入任务描述	请输入JobHandler	请输入负责人	搜索 新增
每页	10	条记录				
任务ID	任务描述	运行模式	Cron	负责人	状态	操作
3	测试任务2	BEAN: demoJobHandler	0/5 * * * * ?	XXL	OSTOP	操作
1	测试任务1	BEAN: demoJobHandler	0 0 0 * * ? *	XXL	OSTOP	操作

2.3.2 启动任务

启动刚才创建的“BEAN模式(Java)”任务。直接点击任务右侧操作下拉框，选择启动即可，会通过配置Cron表达式进行任务调度触发。

任务管理						
执行器	示例执行器	全	请输入任务描述	请输入JobHandler	请输入负责人	搜索 新增
每页	10	条记录				
任务ID	任务描述	运行模式	Cron	负责人	状态	操作
3	测试任务2	BEAN: demoJobHandler	0/5 * * * * ?	XXL	OSTOP	操作
1	测试任务1	BEAN: demoJobHandler	0 0 0 * * ? *	XXL	OSTOP	操作

第 1 页 (总共 1 页, 2 条记录)

上页 日志

接下来，我们就可以测试xxl-job开发的任务是否具有分布式任务调度的特点。

2.3.3 任务调度高可用

目标：测试出分布式任务调度高可用的特点。

方案分析：通过模拟执行器集群的方式，进行任务调度，测试当集群中一个节点挂掉后，任务是否还可以成功完成调度，如果依然可以完成调度，说明实现了高可用。

XXL-JOB提供了如下路由策略保证任务调度高可用，调度中心基于路由策略路由选择一个执行器节点执行任务：

- 忙碌转移策略：下发任务前向执行器节点发起rpc心跳请求查询是否忙碌，如果执行器节点返回忙碌则转移到其他执行器节点执行
- 故障转移策略：下发任务前向执行器节点发起rpc心跳请求查询是否在线，如果执行器节点没返回或者返回不可用则转移到其他执行器节点执行

另外，XXL-JOB还提供了阻塞处理策略，来保证任务的执行效率：

当执行器节点存在多个相同任务id的任务未执行完成，则需要基于阻塞策略对任务进行取舍：

- 串行策略：默认策略，任务进行排队
- 丢弃后续调度
- 覆盖之前调度

案例演示：

上边我们已经配置好两个执行器的启动项并启动，下面就可以直接修改任务了。

- (1) 在调度中心对任务进行修改，修改任务的路由策略为故障转移，然后保存启动。

更新任务

执行器*	示例执行器 ▼	任务描述*	测试任务1
路由策略*	故障转移 ▼	Cron*	0/5 * * * * ?
运行模式*	BEAN ▼	JobHandler*	demo.JobHandler
阻塞处理策略*	单机串行 ▼	子任务ID*	请输入子任务的任务ID,如存在多个则逗号分隔
任务超时时间*	0	失败重试次数*	0
负责人*	XXL	报警邮件*	请输入报警邮件, 多个邮件地址则逗号分隔
任务参数*	请输入任务参数		

保存取消

(2) 查看调度日志，此时调度备注可以看到，调度备注中执行器列表为9997和9998，心跳检测为9997在线，所以9997被选中触发调度。

任务触发类型：Cron触发	
调度机器：192.168.99.146	
执行器-注册方式：自动注册	
执行器地址列表：[192.168.99.146:9997, 192.168.99.146:9998]	
路由策略：故障转移	
阻塞处理策略：单机串行	
任务超时时间：0	
失败重试次数：0	
调度结果	调度备注
>>>>>>>>>触发调度<<<<<<<<<<<	查看
心跳检测：	
address: 192.168.99.146:9997	
code: 200 成功	查看
msg: null	
触发调度： 成功	查看
address: 192.168.99.146:9997	
code: 200	
0:05 成功	查看
msg: null	
9:21 成功	查看

(3) 停掉9997的服务，继续查看日志，此时可以看到，心跳检测9997已经宕机，触发调度的已经转移到9998，任务依然调度成功了。

任务触发类型: Cron触发			
调度机器: 192.168.99.146			
执行器 注册方式: 自动注册			
执行器 地址列表: [192.168.99.146:9997, 192.168.99.146:9998]			
路由策略: 故障转移			
阻塞处理策略: 单机串行			
任务超时时间: 0			
失败重试次数: 0			
	调度结果	调度备注	执行时间
触发调度			
心跳检测:			
address: 192.168.99.146:9997			
code: 500	成功	查看	
msg: com.xxl.rpc.util.XxlRpcException:			
io.netty.channel.AbstractChannel\$AnnotatedConnectException: Connection refused: no			
further information: /192.168.99.146:9997			
心跳检测:			
address: 192.168.99.146:9998			
code: 200	成功	查看	
msg: null			
触发调度:			
address: 192.168.99.146:9998			
code: 200	成功	查看	
msg: null			

结论: 当执行器集群部署时, 其中一个节点挂掉时, 任务被其他节点接管, 继续完成了调度, 系统的功能并未受到影响, 所以我们用XXL-JOB通过执行器集群+故障转移路由策略实现了分布式任务调度的高可用。

2.3.4 避免任务重复执行

目标: 测试出分布式任务调度避免任务重复执行的特点。

方案分析: 模拟执行器集群, 多个节点运行时, 针对同一个任务是否只有一节点在调度, 通过执行日志和执行器控制台观察任务的执行情况, 是否每次任务调度只有一个节点在执行, 而不会被多个节点重复执行, 那说明XXL-JOB可以实现避免任务重复执行, 体现了分布式任务调度的特点。

XXL-JOB提供了如下路由策略来避免任务重复执行, 可以通过配置 "单机路由策略 (如: 第一台、一致性哈希)" + "阻塞策略 (如: 单机串行、丢弃后续调度)" 来规避。

案例演示:

(1) 首先就是在调度中心修改任务的属性, 路由策略改为“第一个”, 阻塞处理策略改为“单机串行”

执行器*	示例执行器	任务描述*	测试任务1
路由策略*	第一个	Cron*	0/5 * * * * ?
运行模式*	BEAN	JobHandler*	demoJobHandler
阻塞处理策略*	单机串行	子任务ID*	请输入子任务的任务ID,如存在多个则逗号分隔
任务超时时间*	0	失败重试次数*	0
负责人*	XXL	报警邮件*	请输入报警邮件, 多个邮件地址则逗号分隔
任务参数*	请输入任务参数		

(2) 观察调度日志的执行结果，通过调度备注可以看到，每次任务调度都是选择集群中在线的9997执行器执行，并且为串行执行，所以不会有重复调度情况的发生。



结论：我们通过使用XXL-JOB的单机路由策略和串行执行阻塞策略可以避免集群情况下任务的重复执行，体现了分布式任务的的特点。

至此，我们已经完成了XXL-JOB的快速入门，实现第一个Hello World分布式任务调度！

2.4 小结

整个分布式任务调度，核心就是通过调度中心+执行器+任务共同完成的，主要的开发流程如下：

- 1.配置好xxl-job的数据库环境
- 2.配置启动调度中心项目和执行器项目
- 3.执行器项目中创建和任务对应的JobHandler类，负责实际业务逻辑的处理

4. 执行器支持集群配置，可以通过执行器项目配置多个启动参数实现多实例启动
5. 在调度中心创建执行器
6. 在调度中心创建任务，任务绑定执行器，并且对应项目中创建的JobHandler
7. 在调度中心启动任务
8. 日志查看

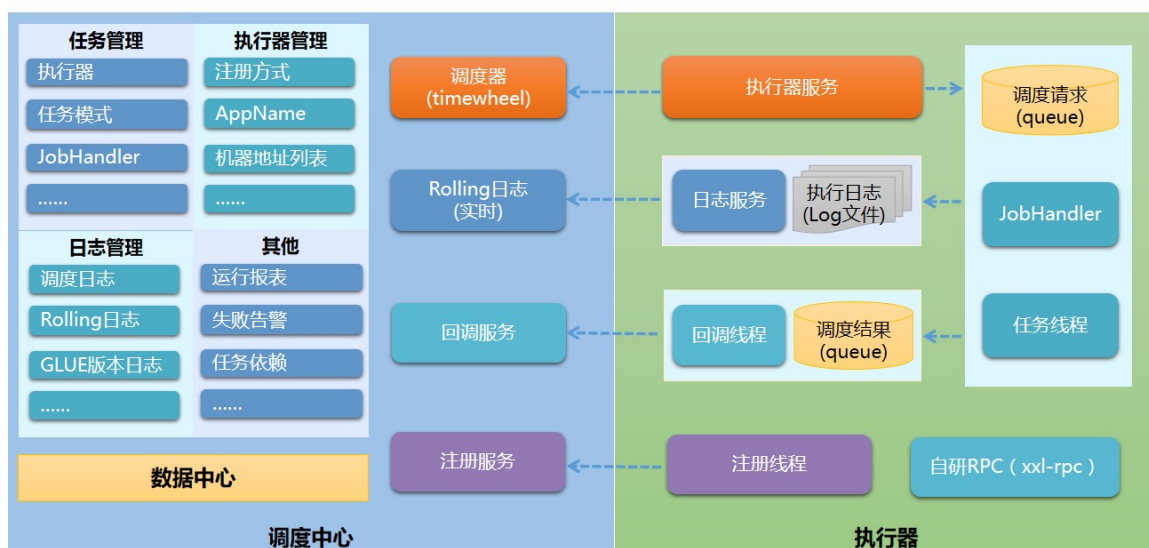
总之：

- 将调度行为抽象形成“调度中心”公共平台，而平台自身并不承担业务逻辑，“调度中心”负责发起调度请求；
- 将任务抽象成分散的JobHandler，交由“执行器”统一管理；
- 执行器负责接收调度请求，并执行对应的任务逻辑；

三、XXL-JOB架构设计及原理

3.1 架构设计

下面是XXL-JOB2.1.0架构设计图：



XXL-JOB架构图 v2.1.0

通过架构图我们可以看出，**系统组成**主要包含两大模块，调度模块和执行器模块：**调度模块：**

- 负责管理调度信息，支持可视化，包括任务的增删改查以及GLUE开发和任务报警，同时支持监控调度日志和执行日志，支持执行器故障转移等；
- 负责按照调度配置发出调度请求，支持集群部署，自身不承担业务代码；
- 调度中心包含：任务管理、执行器管理、日志管理、调度器、回调服务、注册服务等

执行器模块：

- 负责接收调度请求并执行任务逻辑，记录执行日志，上报执行结果，支持集群部署。
 - 任务模块专注于任务的执行等操作。
 - 执行器包含：执行器服务、调度请求队列、任务线程、日志服务、回调线程、注册线程等

总得来说，XXL-JOB的**设计思想**就是：

- 将调度行为抽象形成“调度中心”公共平台，而平台自身并不承担业务逻辑，“调度中心”负责发起调度请求。

- 将任务抽象成分散的JobHandler，交由“执行器”统一管理
- 执行器负责接收调度请求并执行对应的JobHandler中业务逻辑。

“调度”和“任务”两部分可以相互解耦，提高系统整体稳定性和扩展性，功能丰富的可视化管理界面，使开发和维护更加简单和高效；

3.2 XXL-JOB的工作原理

调度中心：

主要职责是执行器管理、任务管理、日志管理、监控运维和发起调度请求等

任务执行器：

主要职责是注册服务、接受调度请求并执行任务、执行结果上报、提供日志服务等

任务：

主要职责是具体的业务处理。

执行流程：

- 1.任务执行器根据配置的调度中心的地址，自动注册到调度中心
- 2.达到任务触发条件，调度中心下发任务调度
- 3.执行器基于线程池执行任务，并把执行结果放入内存队列中、把执行日志写入日志文件中
- 4.执行器的回调线程消费内存队列中的执行结果，主动异步上报给调度中心
- 5.当用户在调度中心查看任务日志，调度中心请求任务执行器，任务执行器读取任务日志文件并返回日志详情

四、任务调度中心操作指南

XXL-JOB提供了人性化的管理界面，即任务调度中心，通过浏览器登录即可使用。

浏览器访问地址：<http://localhost:8080/xxl-job-admin> 登录用户名：admin，密码：123456。

任务调度中心包含执行器管理（属性说明、配置执行器）、任务管理（任务详解，各种运行模式）、日志管理、用户管理和运行报表。

4.1 执行器管理

4.1.1 配置执行器

在任务调度中心，点击进入“执行器管理”界面, 如下图:



任务调度中心					
执行器管理					
执行器列表 新增执行器					
排序	AppName	名称	注册方式	OnLine 机器地址	操作
1	xxl-job-executor-sample	示例执行器	自动注册		编辑 删除
1	executor2	执行器2	自动注册		编辑 删除

1、此处的AppName,会在创建任务时被选择，每个任务必然要选择一个执行器。 2、“执行器列表”中显示在线的执行器列表, 支持编辑删除。

以下是执行器的属性说明：

属性名称	说明
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;
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;

具体操作:

(1) 新增执行器:

新增执行器

AppName*

名称*

排序*

注册方式*

☒ 自动注册
☐ 手动录入

机器地址*

请输入执行器地址列表, 多地址逗号分隔

保存

取消

(2) 自动注册和手动注册的区别和配置

注册方式*

☐ 自动注册
☒ 手动录入

机器地址*

127.0.0.1:9997,127.0.0.1:9998

 多个地址中间用英文的逗号隔开

4.2 任务管理

任务管理界面如下：



4.2.1 任务的基本操作

在调度中心操作台的任务管理中，任务的基本操作主要有：

新增、执行、启动/停止、GLUE IDE、编辑、删除、日志；

4.2.1.1 任务的新增

登录调度中心，进入任务管理界面，点击“新增”按钮，在弹出的“新增”界面配置任务属性后保存即可。

任务配置属性介绍

执行器*	示例执行器	任务描述*	请输入任务描述
路由策略*	第一个	Cron*	请输入Cron
运行模式*	BEAN	JobHandler*	请输入JobHandler
阻塞处理策略*	单机串行	子任务ID*	请输入子任务的任务ID,如存在多个则逗号分隔
任务超时时间*	任务超时时间，单位秒，大于零时生效	失败重试次数*	失败重试次数，大于零时生效
负责人*	请输入负责人	报警邮件*	请输入报警邮件，多个邮件地址则逗号分隔
任务参数*	请输入任务参数		

一、路由策略

路由策略：当执行器集群部署时，提供丰富的路由策略，包括

策略名称	描述
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)	在线的执行器轮流来执行;
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)	最久未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;

二、Cron表达式

(1) 基本介绍:

Cron表达式是一个字符串, 是由空格隔开的6或7个域组成, 每一个域对应一个含义 (秒 分 时 每月第几天 月 星期 年) 其中年是可选字段。

cron表达式格式: {秒数} {分钟} {小时} {日期} {月份} {星期} {年份(可为空)}

cron表达式由七部分组成, 中间由空格分隔, 这七部分从左往右依次是:

英文字段名	中文字段名	合法值	允许的特殊字符
Seconds	秒	0 - 59	, - * /
Minutes	分钟	0 - 59	, - * /
Hours	小时	0 - 23	, - * /
Day-of-Month	日期	1 - 月最后一天	, - * / ? L W
Month	月份	1 - 12 或 JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC	, - * /
Day-of-Week	星期几	1 - 7(1表示星期日,即从星期日开始) 或 SUN, MON, TUE, WED, THU, FRI, SAT	, - * / ? L C #
Year (optional field)	年(可选项)	1970-2099, 一般该项不设置, 直接忽略掉, 即可为空值	, - * /

*表示所有值；?表示未说明的值，即不关心它为何值；-表示一个指定的范围；,表示附加一个可能值；/符号前表示开始时间，符号后表示每次递增的值；

(2) 常用表达式例子

- 0/5 * * * * ? 每隔5秒执行一次
- 0 0/30 9-17 * * ? 朝九晚五时间内每半小时执行一次
- 0 15 10,14 ? * MON-FRI 表示周一到周五每天上午10:15和14:15执行作业

(3) 效果演示

表达式

表达式字段:

0/5

秒

*

分钟

*

小时

*

日

*

月

?

星期

年

Cron 表达式:

0/5 * * * * ?

反解析到UI

运行

最近10次运行时间

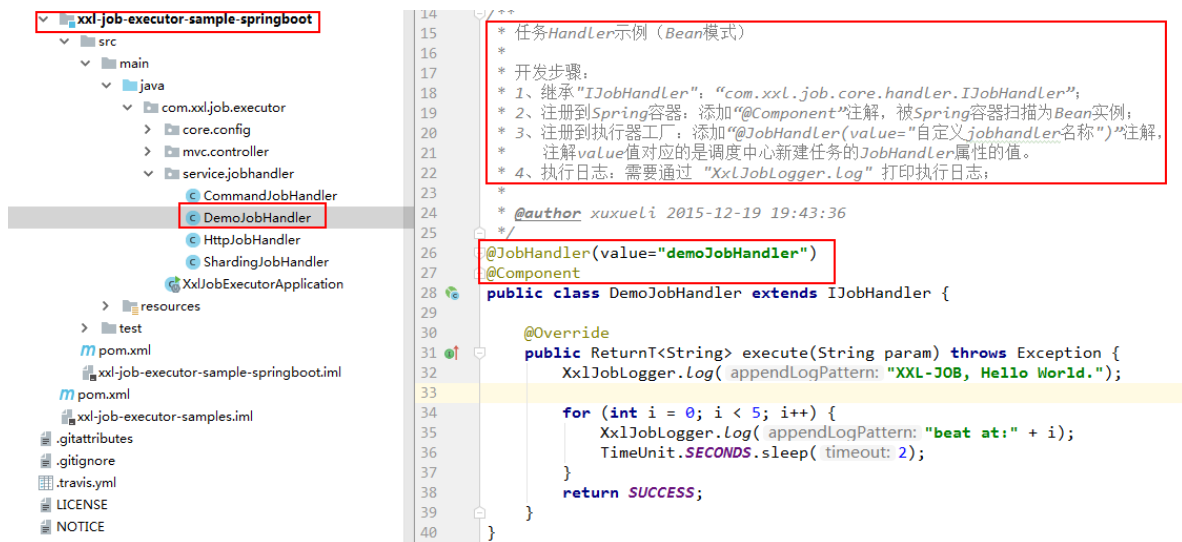
三、任务的运行模式

任务模式共提供有7种，目前项目中实际使用的Bean模式较多，在这里我们只介绍其中有代表性的两种：Bean模式 和GLUE(Java)模式。

(1) Bean模式

Bean模式任务提供多个示例Handler，可以直接配置使用，任务逻辑以JobHandler的形式存在于“执行器”所在项目中，开发流程如下：

- 执行器项目中，开发JobHandler
- 代码逻辑根据业务需求编写：



- 调度中心，新建调度任务

新建一个任务，运行模式选中 "BEAN模式", JobHandler属性填写任务注解"@JobHandler"中定义的值;

- 启动任务，查看执行日志

(2) GLUE(Java)模式

- 新增一个GLUE(Java)模式的任务

新增

执行器*	示例执行器	任务描述*	GLUE (Java) 模式测试
路由策略*	第一个	Cron*	0/3 * * * * ?
运行模式*	GLUE(Java)	JobHandler*	请输入JobHandler
阻塞处理策略*	单机串行	子任务ID*	请输入子任务的任务ID,如存在多个则逗
任务超时时间*	任务超时时间,单位秒,大于零时生效	失败重试次数*	失败重试次数,大于零时生效
负责人*	XXL	报警邮件*	请输入报警邮件,多个邮件地址则逗号
任务参数*	请输入任务参数		

保存

取消

- 保存后，点击任务后边操作下拉框，选择GLUE IDE可以在线编辑代码：



- 回到任务管理列表，启动任务，查看执行日志

执行日志 Console

```
2019-11-21 15:59:51 [com.xx1.job.core.thread.JobThread#run]-[124]-[Thread-19]
----- xxl-job job execute start -----
----- Param:
2019-11-21 15:59:51 [com.xx1.job.core.handler.impl.GlueJobHandler#execute]-[25]-[Thread-19] ----- glue.version:1574321133000 -----
2019-11-21 15:59:51 [sun.reflect.NativeMethodAccessorImpl#invoke0]-[-2]-[Thread-19] XXL-JOB, Hello World.测试GLUE(JAVA)模式
2019-11-21 15:59:51 [com.xx1.job.core.thread.JobThread#run]-[164]-[Thread-19]
----- xxl-job job execute end(finish) -----
----- ReturnT:ReturnT [code=200, msg=null, content=null]
2019-11-21 15:59:51 [com.xx1.job.core.thread.TriggerCallbackThread#callbackLog]-[190]-[xxl-job, executor TriggerCallbackThread]
----- xxl-job job callback finish.

[Load Log Finish]
```

4.2.1.2 任务的执行启动和停止

以下是具体操作：

执行，需要手动输入执行参数。

任务管理

执行器

示例执行器

每页 10 条记录

任务ID	任务描述
1	测试任务1

执行

任务参数*

请输入任务参数

保存

取消

启动，手动点击开启任务，等待cron表达式触发。

停止，手动关闭正在执行的任务。

4.2.1.3 GLUE IDE编辑删除

以下是具体操作：

GLUE IDE，GLUE运行模式特有的选项，即在线编辑任务代码或者脚本。

WebIDE【GLUE(Java)】GLUE任务

```
package com.xx1.job.service.handler;

import com.xx1.job.core.log.XxlJobLogger;
import com.xx1.job.core.biz.model.ReturnT;
import com.xx1.job.core.handler.IJobHandler;

public class DemoGlueJobHandler extends IJobHandler {

    @Override
    public ReturnT<String> execute(String param) throws Exception {
        XxlJobLogger.log("XXL-JOB, Hello World.");
        return ReturnT.SUCCESS;
    }

}
```


编辑，就是修改任务的基本属性，但是任务的运行模式一旦创建不可修改。

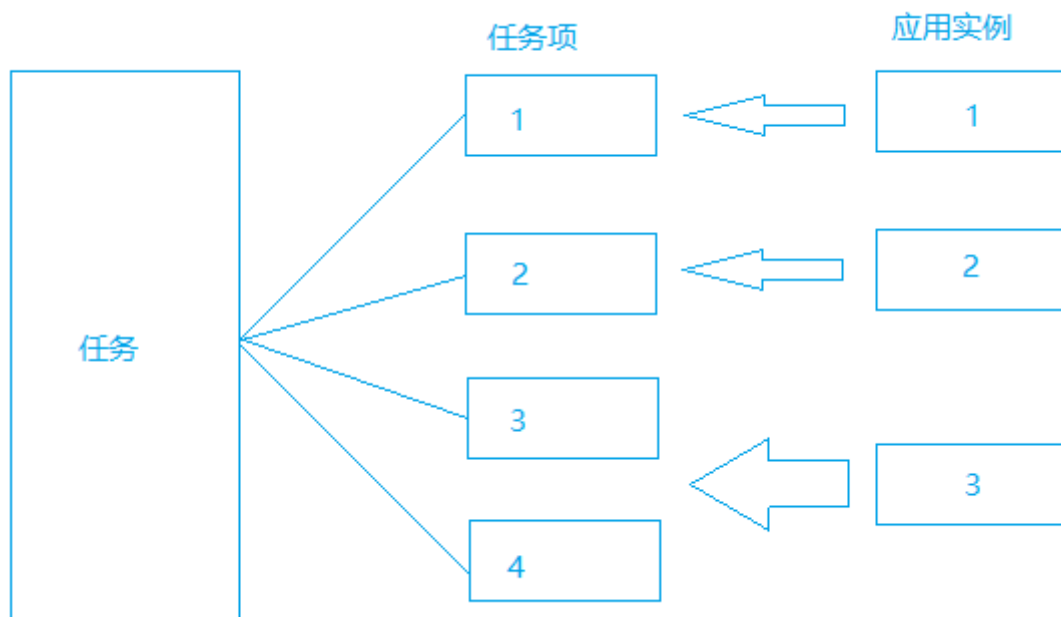
删除，手动删除任务。

4.2.2 广播任务和动态分片

4.2.2.1 概述

什么是作业分片：

作业分片是指任务的分布式执行，需要将一个任务拆分为多个独立的任务项，然后由分布式的应用实例分别执行某一个或几个分片项。



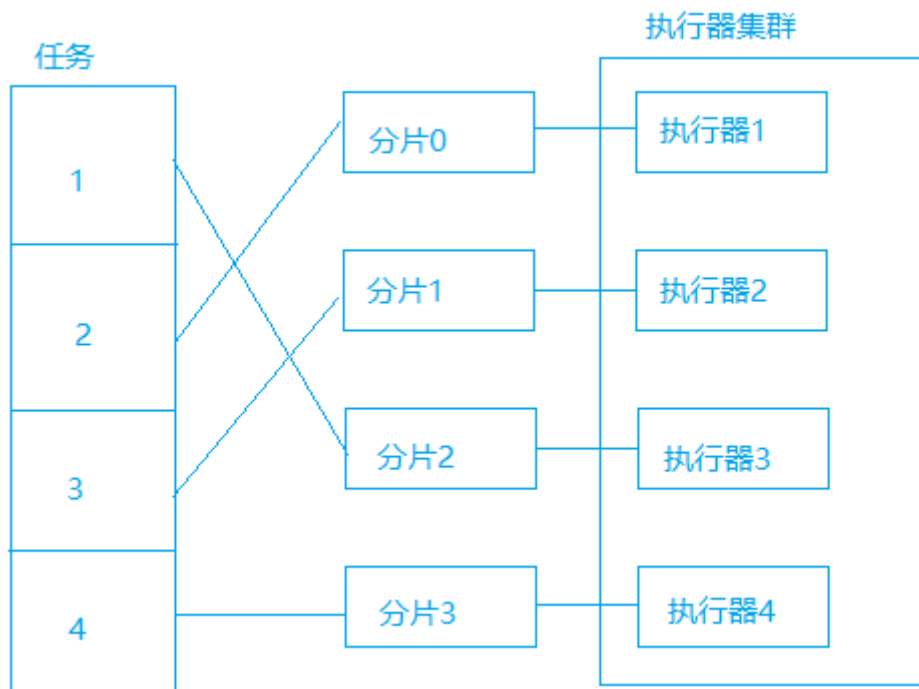
4.2.2.2 XXL-JOB分片

广播任务：

执行器集群部署，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，系统会自动传递分片参数，可根据分片参数开发分片任务；

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。



(1) XXL-JOB支持分片的好处

- 分片项与业务处理解耦

XXL-JOB并不直接提供数据处理的功能，框架只会将分片项分配至各个运行中的作业服务器，开发者需要自行处理分片项与真实数据的对应关系。

- 最大限度利用资源

基于业务需求配置合理数量的执行器服务，合理设置分片，作业将会最大限度合理的利用分布式资源。

(2) 使用说明：

"分片广播" 和普通任务开发流程一致，不同之处在于可以获取分片参数进行分片业务处理。

- Java语言任务获取分片参数方式：

BEAN、GLUE模式(Java)，可参考Sample示例执行器中的示例任务"ShardingJobHandler":
`ShardingUtil.ShardingVO shardingVO = ShardingUtil.getShardingVo();`

- 分片参数属性说明：

index：当前分片序号(从0开始)，执行器集群列表中当前执行器的序号； total：总分片数，执行器集群的总机器数量；

4.2.2.3 分片广播案例演示

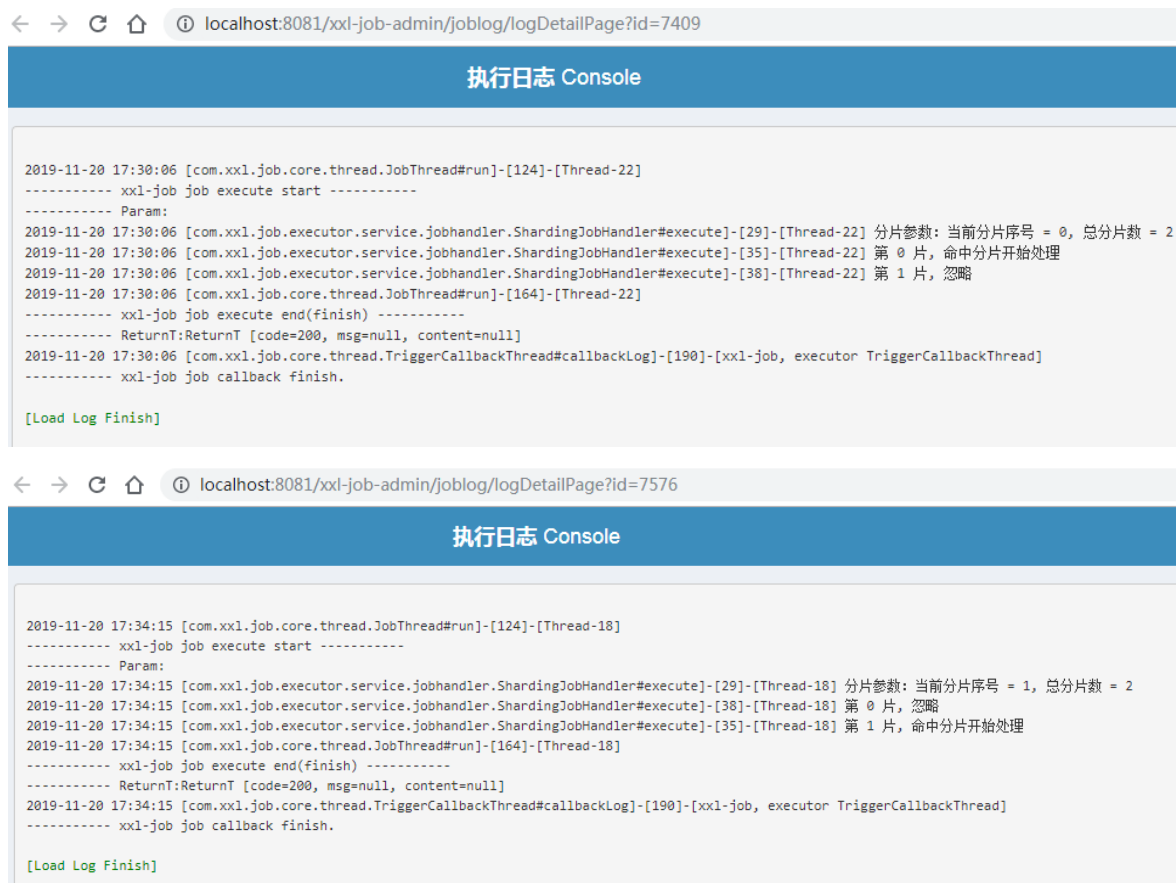
目标：实现XXL-JOB作业分片的演示

方案分析：规划一个任务，两个分片，对应两个执行器，每个分片处理一部分任务。

实现步骤：

- (1)：执行器项目中，开发ShardingJobHandler：

分片示例：

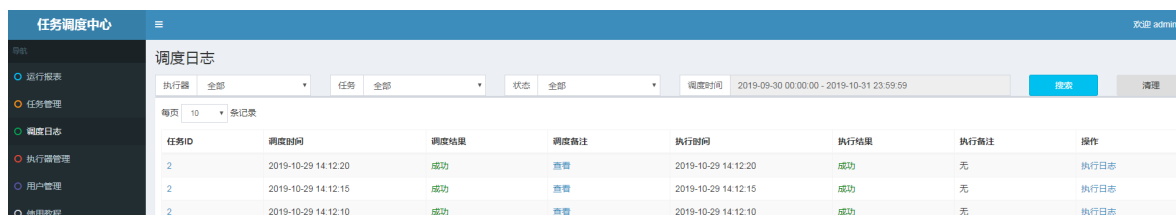


结论：通过调度日志可以看出，同一个任务在给定的时间点、给定的时间间隔被多个进程不断调用，实现了任务的分片，体现了分布式任务调度的并行任务调度的特点。

4.3 调度日志

4.3.1 调度日志列表展示

进入调度日志界面，可以查看任务调度的记录列表，支持条件搜索和清理。



4.3.2 调度备注查看

调度备注展示调度相关的基本配置属性内容。

任务触发类型：Cron触发		
调度机器：169.254.100.246		
执行器-注册方式：自动注册	状态：全部	调度时间：2019-09-30 00
执行器-地址列表：[169.254.100.246:9999]		
路由策略：第一个		
阻塞处理策略：单机串行		
任务超时时间：0		
失败重试次数：0	调度备注	执行时间
成功	查看	2019-10-29 14:12:20
成功	查看	2019-10-29 14:12:15
成功	查看	2019-10-29 14:12:10
成功	查看	2019-10-29 14:12:05
成功	查看	2019-10-29 14:12:01
失败	查看	

4.3.3 查看执行日志

点击调度日志右侧的“执行日志”按钮，可跳转至执行日志界面，可以查看业务代码中打印的完整日志，支持动态刷新，如下图：

执行日志 Console

刷新

```
2019-10-29 14:12:00 [com.xx1.job.core.thread.JobThread#run]-[124]-[Thread-17]
----- xx1-job job execute start -----
----- Param:
2019-10-29 14:12:00 [com.xx1.job.core.handler.impl.GlueJobHandler#execute]-[25]-[Thread-17] ----- glue.version:1572329099000 -----
2019-10-29 14:12:00 [sun.reflect.NativeMethodAccessorImpl#invoke0]-[-2]-[Thread-17] -----XXL-JOB, Hello World.
2019-10-29 14:12:00 [com.xx1.job.core.thread.JobThread#run]-[164]-[Thread-17] -----
----- xx1-job job execute end(finish) -----
----- ReturnT:ReturnT [code=200, msg=null, content=null]
2019-10-29 14:12:00 [com.xx1.job.core.thread.TriggerCallbackThread#callbackLog]-[190]-[xx1-job, executor TriggerCallbackThread]
----- xx1-job job callback finish.

[Load Log Finish]
```

Powered by XXL-JOB 2.1.0

Copyright © 2015-2019 xuxuelli github

4.4 用户管理

4.4.1 用户管理

进入 "用户管理" 界面，可查看和管理用户信息：

任务调度中心

用户管理

角色：全部 账号： 搜索 新增用户

每页：10 条记录

账号	密码	角色	操作
admin	*****	管理员	编辑 删除

第 1 页 (总共 1 页, 1 条记录)

上页 1 下页

目前用户分为两种角色：

- 管理员：拥有全量权限，支持在线管理用户信息，为用户分配权限，权限分配粒度为执行器；
- 普通用户：仅拥有被分配权限的执行器，及相关任务的操作权限；

新增用户

账号*

请输入账号

密码*

请输入密码

角色*

☒ 普通用户 ☐ 管理员

权限*

☐ 示例执行器(xxl-job-executor-sample)

☐ 测试执行器2(xxl-job-executor-test)

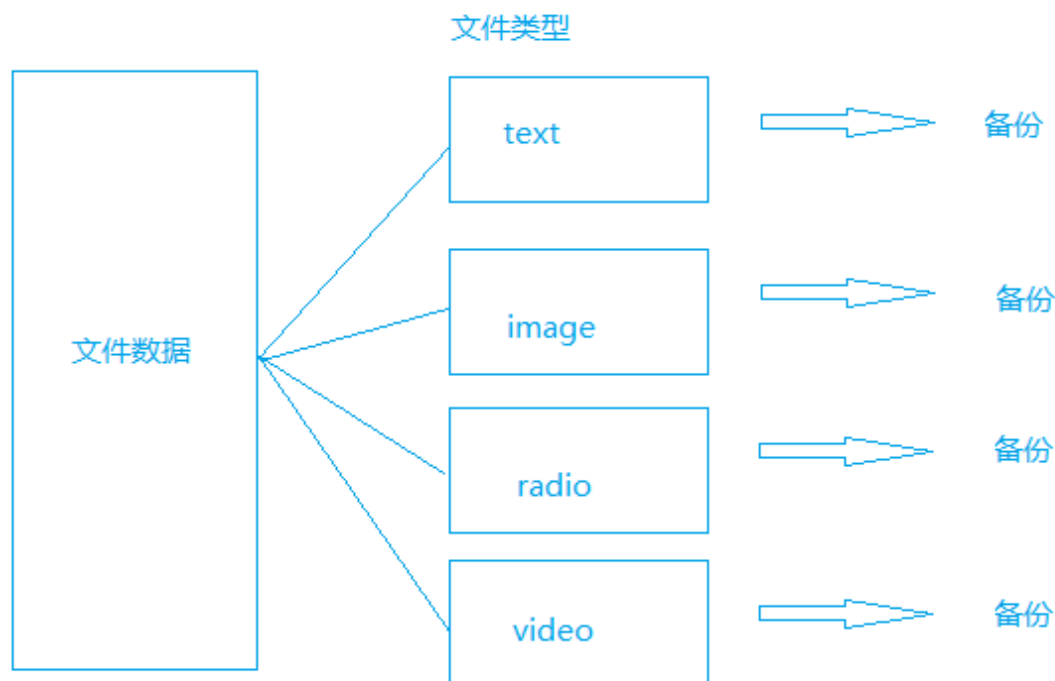
保存

取消

五、XXL-JOB实战案例

5.1 需求描述

有四种类型的文件，分别为text、image、radio、vedio，我们需要基于Spring boot集成xxl-job方式的而产出的工程代码，采取更接近真实项目的数据库存取方式，使用分布式任务调度完成对这几种类型的文件的备份。



5.2 解决方案分析

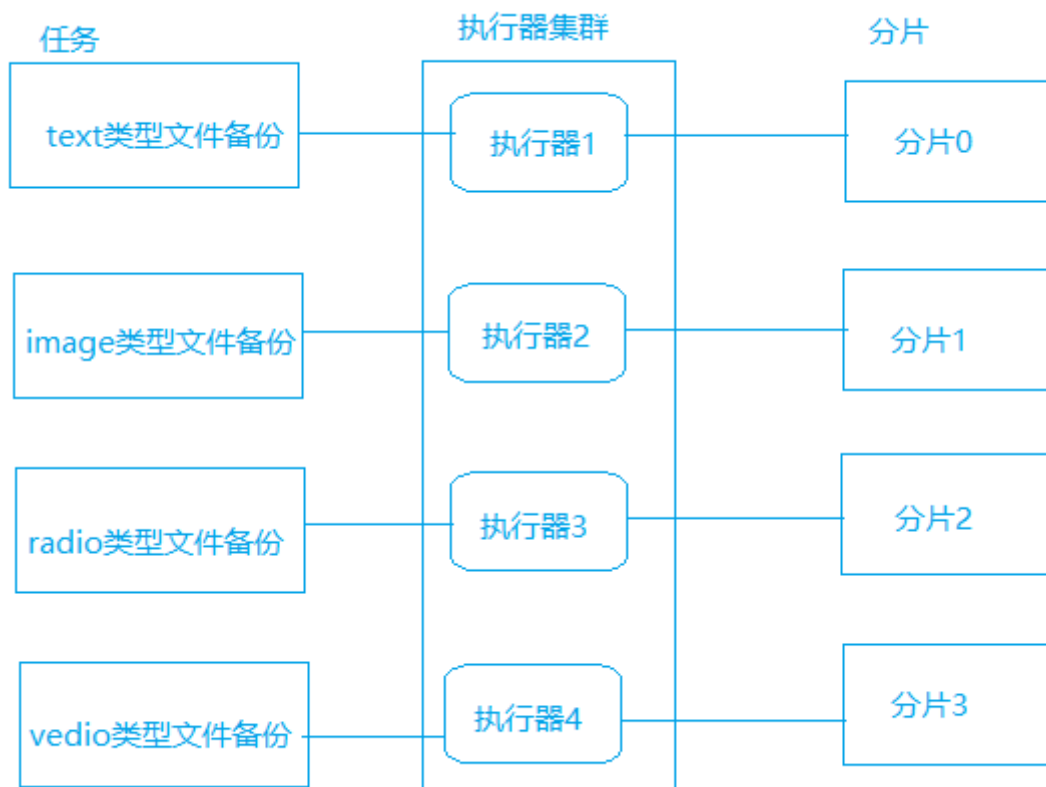
使用XXL-JOB提供的对作业分片的支持，我们用分片广播模式实现对文件备份的分布式任务调度。

(1) 文件类型有4种：text、image、radio、video类型，考虑一个实例来执行一种类型的文件备份，所以分片数设定为4；

(2) 由于XXL-JOB是总分片数是基于执行器集群的总机器数量的，所以要配置4个执行器服务（可使用相同执行器名称，不同端口模拟集群），从而达到任务并行处理的效果，最大限度的提高执行作业的吞吐量。

分片策略如下：

开启4个执行器的服务，分片数为4片，每个实例对应处理一个分片，分片序号默认从0开始，那么每个实例处理的分片为：



5.3 环境说明

数据库：MySQL-5.7.25

JDK：64位 jdk1.8.0_201

框架：SpringBoot-1.5.21.RELEASE

XXL-JOB: xxl-job-core 2.1.0

5.4 作业分片的实现

5.4.1 初始化数据库

创建xxl_job_demo数据库：

```
CREATE DATABASE `xxl_job_demo` CHARACTER SET 'utf8' COLLATE 'utf8_general_ci';
```

在xxl_job_demo库中创建t_file表：

```

DROP TABLE IF EXISTS `t_file`;
CREATE TABLE `t_file` (
  `id` varchar(11) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL,
  `name` varchar(255) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT
  NULL,
  `type` varchar(255) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT
  NULL,
  `content` varchar(255) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT
  NULL,
  `backedup` tinyint(1) NULL DEFAULT NULL,
  PRIMARY KEY (`id`) USING BTREE
) ENGINE = InnoDB CHARACTER SET = utf8 COLLATE = utf8_general_ci ROW_FORMAT =
Dynamic;

```

5.4.2 新增maven依赖

在xxl-job-executor-sample-springboot项目的pom文件中，加入以下依赖：

```

<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <version>1.18.0</version>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<!-- mysql -->
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>5.1.47</version>
</dependency>

```

5.4.3 配置属性文件

在工程的application.properties中加入数据库的相关配置：

```

### datasource
spring.datasource.url=jdbc:mysql://127.0.0.1:3306/xxl_job_demo?
serverTimezone=Asia/Shanghai&useUnicode=true&characterEncoding=utf8&useSSL=false
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.jdbc.Driver

```

5.4.4 编写实体类

```

package com.xxl.job.executor.entity;

import lombok.Data;

import java.io.Serializable;

@Data
public class FileCustom implements Serializable {

```



```

/**
 * 标识
 */
private String id;

/**
 * 文件名
 */
private String name;

/**
 * 文件类型，如text、image、radio、vedio
 */
private String type;

/**
 * 文件内容
 */
private String content;

/**
 * 是否已备份
 */
private Boolean backedUp = false;

public FileCustom() {
}

public FileCustom(String id, String name, String type, String conten) {
    this.id = id;
    this.name = name;
    this.type = type;
    this.content = content;
}
}

```

5.4.5 编写文件服务类

```

package com.xx1.job.executor.service;

import com.xx1.job.executor.entity.FileCustom;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.BeanPropertyRowMapper;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Service;

import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.List;

@Service
public class FileService {

    @Autowired
    private JdbcTemplate jdbcTemplate;

```

```

/**
 * 获取某类型的未备份的文件
 * @param i 分片参数
 * @param fileType 文件类型
 * @param count 获取文件条数
 * @return
 */
public List<FileCustom> fetchUnBackupFiles(int i,String fileType, Integer
count){
    List<FileCustom> files = jdbcTemplate.query(
        "select * from t_file t where t.type = ? and t.backedUp = 0
order by id limit 0,? "
        ,new Object[]{fileType, count}
        ,new BeanPropertyRowMapper(FileCustom.class));
    System.out.println(String.format("Time: %s | 线程 %d | 分片 %s | 已获取文件
数据 %d 条"
        ,new SimpleDateFormat("HH:mm:ss").format(new Date())
        ,Thread.currentThread().getId()
        ,i
        ,files.size()));
    return files;
}

/**
 * 备份文件
 * @param files 要备份的文件
 */
public void backupFiles(List<FileCustom> files){
    for(FileCustom file : files){
        jdbcTemplate.update("update t_file set backedUp = 1 where id =
?",new Object[]{file.getId()});
        System.out.println(String.format("线程 %d | 已备份文件:%s 文件类型:%s"
            ,Thread.currentThread().getId()
            ,file.getName()
            ,file.getType()));
    }
}
}

```

5.4.6 创建JobHandler

```

package com.xx1.job.executor.service.jobhandler;

import com.xx1.job.core.biz.model.ReturnT;
import com.xx1.job.core.handler.IJobHandler;
import com.xx1.job.core.handler.annotation.JobHandler;
import com.xx1.job.core.log.XxlJobLogger;
import com.xx1.job.core.util.ShardingUtil;
import com.xx1.job.executor.entity.FileCustom;
import com.xx1.job.executor.service.FileService;
import org.springframework.stereotype.Service;

import javax.annotation.Resource;
import java.util.List;

@JobHandler("fileBackupJobHandler")

```

```

@Service
public class FileBackupJobHandler extends IJobHandler {

    @Resource
    private FileService fileService;

    @Override
    public ReturnT<String> execute(String param) throws Exception {

        // 分片参数
        ShardingUtil.ShardingVO shardingVO = ShardingUtil.getShardingVo();
        XxlJobLogger.log("分片参数: 当前分片序号 = {}, 总分片数 = {}",
            shardingVO.getIndex(), shardingVO.getTotal());

        // 业务逻辑
        for (int i = 0; i < shardingVO.getTotal(); i++) {
            //分片处理
            if (0 == shardingVO.getIndex()) {
                XxlJobLogger.log("第 {} 片, 命中分片开始处理", i);
                executeFile(0,"text",10);
            } else if(1 == shardingVO.getIndex()){
                XxlJobLogger.log("第 {} 片, 命中分片开始处理", i);
                executeFile(1,"image",10);
            } else if(2 == shardingVO.getIndex()){
                XxlJobLogger.log("第 {} 片, 命中分片开始处理", i);
                executeFile(2,"radio",10);
            } else if(3 == shardingVO.getIndex()){
                XxlJobLogger.log("第 {} 片, 命中分片开始处理", i);
                executeFile(3,"vedio",10);
            }else {
                XxlJobLogger.log("忽略", i);
            }
        }
        return SUCCESS;
    }

    /**
     * 文件获取和备份的具体处理逻辑
     * @param i
     * @param type
     * @param count
     */
    private void executeFile(int i,String type,int count){
        System.out.println("第"+i+"片执行, 处理"+type+"类型的文件");
        //获取某种类型未备份的文件
        List<FileCustom> files = fileService.fetchUnBackupFiles(i,type, 10);
        //备份文件
        fileService.backupFiles(files);
    }
}

```

5.4.7 配置启动项

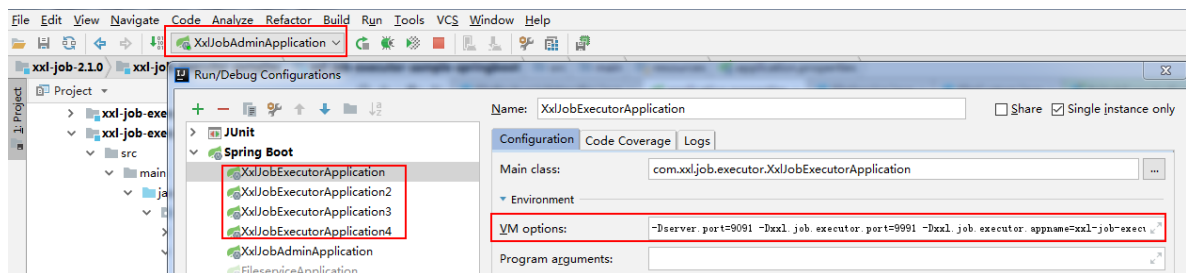
在IDEA工具栏打开编辑配置，配置4个执行器启动项，启动参数VM options依次加入如下配置：

```
-Dserver.port=9091 -Dxxl.job.executor.port=9991 -Dxxl.job.executor.appname=xxl-job-executor-sample -Dxxl.job.admin.addresses=http://127.0.0.1:8080/xxl-job-admin
```

```
-Dserver.port=9092 -Dxxl.job.executor.port=9992 -Dxxl.job.executor.appname=xxl-job-executor-sample -Dxxl.job.admin.addresses=http://127.0.0.1:8080/xxl-job-admin
```

```
-Dserver.port=9093 -Dxxl.job.executor.port=9993 -Dxxl.job.executor.appname=xxl-job-executor-sample -Dxxl.job.admin.addresses=http://127.0.0.1:8080/xxl-job-admin
```

```
-Dserver.port=9094 -Dxxl.job.executor.port=9994 -Dxxl.job.executor.appname=xxl-job-executor-sample -Dxxl.job.admin.addresses=http://127.0.0.1:8080/xxl-job-admin
```



依次启动xxl-job-admin和xxl-job-executor-sample-springboot项目，总共是5个启动项。

5.4.8 在XXL-JOB的任务调度中心操作台完成执行器和任务创建

1.登录到XXL平台，用户名：admin 密码:123456

2.在执行器管理中新增执行器

此处我们不再新建，使用初始数据中自带的示例执行器：

执行器管理					
执行器列表 新增执行器					
排序	AppName	名称	注册方式	OnLine 机器地址	操作
1	xxl-job-executor-sample	示例执行器	自动注册		编辑 删除

3.新建文件备份任务，保存后启动：

执行器*	示例执行器	任务描述*	文件备份任务
路由策略*	分片广播	Cron*	0/5 * * * * ?
运行模式*	BEAN	JobHandler*	fileBackupJobHandler
阻塞处理策略*	单机串行	子任务ID*	请输入子任务的任务ID,如存在多个则逗号分隔
任务超时时间*	0	失败重试次数*	0
负责人*	XXL	报警邮件*	请输入报警邮件，多个邮件地址则逗号分隔
任务参数*	请输入任务参数		

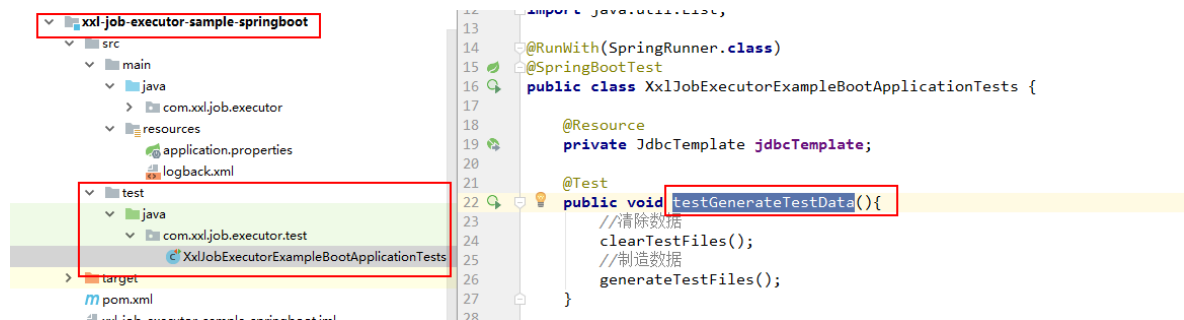
[保存](#) [取消](#)

新增完成后，启动任务：

任务管理						
执行器	示例执行器	全部	请输入任务描述	请输入JobHandler	请输入负责人	搜索 新增
每页 10	条记录					
任务ID	任务描述	运行模式	Cron	负责人	状态	操作
12	文件备份任务	BEAN : fileBackupJobHandler	0/5 * * * * ?	XXL	OSTOP	操作
11	GLUE (Java) 模式测试	GLUE(Java)	0/3 * * * * ?	XXL	OSTOP	执行
1	测试任务1	BEAN : demoJobHandler	0/5 * * * * ?	XXL	OSTOP	启动
第 1 页 (总共 1 页 , 3 条记录)						日志
						编辑
						删除

5.5 测试

按照XXL调度中心创建的执行器和任务，启动任务后，在给定的时间点和时间间隔，会执行对不同文件类型的文件进行备份操作，此时我们自己模拟一些测试数据，执行一下创建数据的测试类：



以下是调度中心的日志记录和控制台的日志：

任务调度中心执行日志：

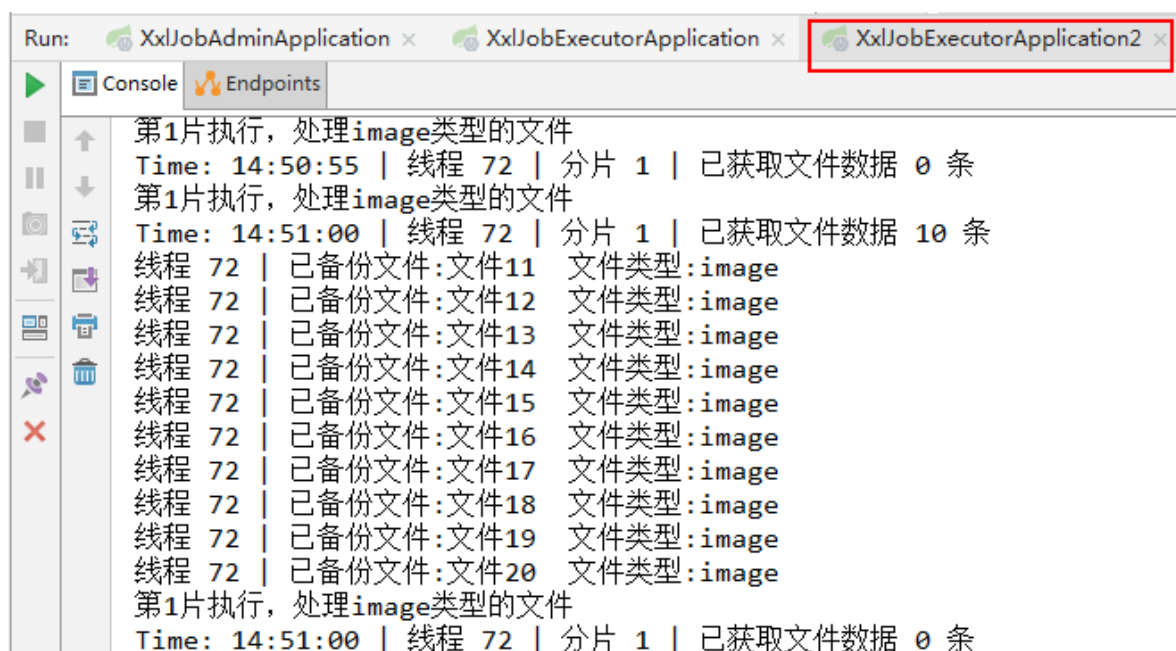
执行日志 Console
<pre>2019-11-22 14:49:15 [com.xx1.job.core.thread.JobThread#run]-[124]-[Thread-23] ----- xx1-job job execute start ----- ----- Param: 2019-11-22 14:49:15 [com.xx1.job.executor.service.jobhandler.FileBackupJobHandler#execute]-[27]-[Thread-23] 分片参数: 当前分片序号 = 0, 总分片数 = 4 2019-11-22 14:49:15 [com.xx1.job.executor.service.jobhandler.FileBackupJobHandler#execute]-[33]-[Thread-23] 第 0 片, 命中分片开始处理 2019-11-22 14:49:15 [com.xx1.job.executor.service.jobhandler.FileBackupJobHandler#execute]-[33]-[Thread-23] 第 1 片, 命中分片开始处理 2019-11-22 14:49:15 [com.xx1.job.executor.service.jobhandler.FileBackupJobHandler#execute]-[33]-[Thread-23] 第 2 片, 命中分片开始处理 2019-11-22 14:49:15 [com.xx1.job.executor.service.jobhandler.FileBackupJobHandler#execute]-[33]-[Thread-23] 第 3 片, 命中分片开始处理 2019-11-22 14:49:15 [com.xx1.job.core.thread.JobThread#run]-[164]-[Thread-23] ----- xx1-job job execute end(finish) ----- ----- ReturnT:ReturnT [code=200, msg=null, content=null] 2019-11-22 14:49:15 [com.xx1.job.core.thread.TriggerCallbackThread#callbackLog]-[190]-[xx1-job, executor TriggerCallbackThread] ----- xx1-job job callback finish. [Load Log Finish]</pre>

IDEA控制台日志，查看4个执行器项目的控制台：

执行器1：



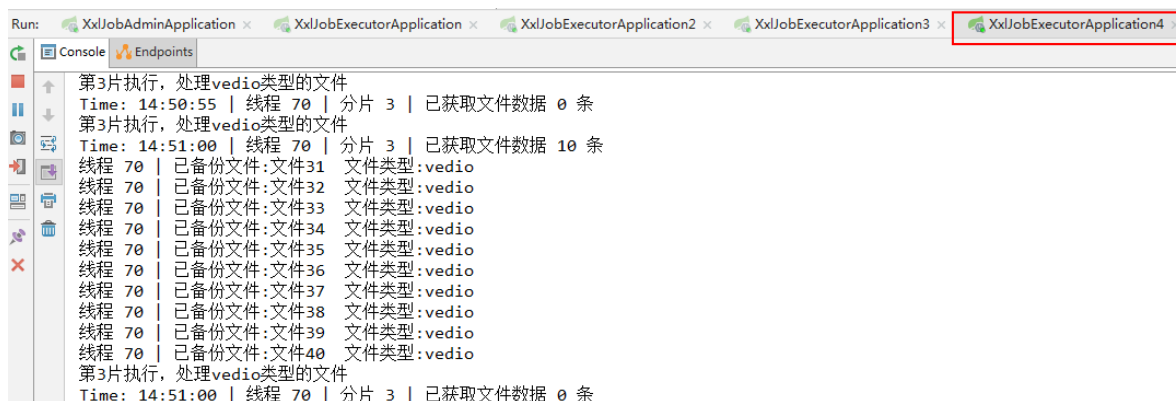
执行器2:



执行器3:



执行器4:



至此，我们就使用分片广播模式，完成了XXL-JOB的分布式任务调度实战案例！

六、XXL-JOB高级

6.1 HA设计

6.1.1 调度中心高可用

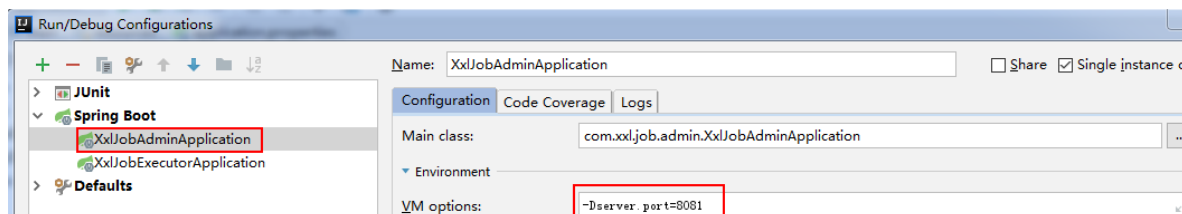
6.1.1.1 概述

调度中心支持多节点部署，基于数据库行锁保证同时只有一个调度中心节点触发任务调度，当正在运行的节点挂掉，其他可用节点会继续之前节点的工作。

6.1.1.2 案例演示

我们把调度中心改造为高可用，具体操作如下：

(1) 调度中心的修改，我们通过修改启动参数，模拟两个调度中心实例，web端口分别为为8081和8082



(2) 执行器（此处使用上面创建的xxl-job-executor-sample示例执行器）的配置，执行器端口指定为9997（默认9999，可不改）；web端口指定为9091；调度中心集群地址配置时，配置两个调度中心的地址用逗号隔开。



(3)：启动任务执行

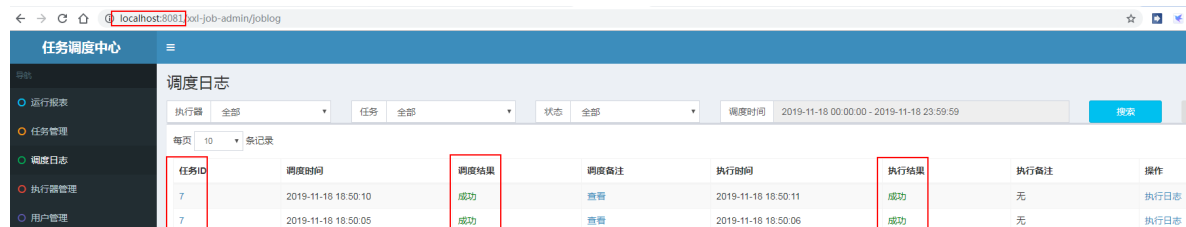
启动创建任务。

通常情况下，点击任务右侧启动即可，通过配置Cron表达式进行任务调度触发；也可点击任务右侧“执行”按钮，需要配置执行参数（url），立即手动触发一次任务执行。

（4）：查看任务调度日志

在调度日志页面，可以看到每条任务调度的调度时间、调度结果和调度备注等，调度成功的每条后有“执行日志”链接，点击可以进入执行日志控制台。

下面是调度日志列表：



任务ID	调度时间	调度结果	调度备注	执行时间	执行结果	执行备注	操作
7	2019-11-18 18:50:10	成功	查看	2019-11-18 18:50:11	成功	无	执行日志
7	2019-11-18 18:50:05	成功	查看	2019-11-18 18:50:06	成功	无	执行日志
7	2019-11-18 18:50:00	成功	查看	2019-11-18 18:50:01	成功	无	执行日志



任务ID	调度时间	调度结果	调度备注	执行时间	执行结果	执行备注	操作
7	2019-11-18 18:50:10	成功	查看	2019-11-18 18:50:11	成功	无	执行日志
7	2019-11-18 18:50:05	成功	查看	2019-11-18 18:50:06	成功	无	执行日志
7	2019-11-18 18:50:00	成功	查看	2019-11-18 18:50:01	成功	无	执行日志

下面是执行日志记录：



```
2019-11-18 18:56:30 [com.xx1.job.core.thread.JobThread#run]-[124]-[Thread-19]
----- xxl-job job execute start -----
----- Param:
2019-11-18 18:56:30 [com.xx1.job.core.handler.impl.GlueJobHandler#execute]-[25]-[Thread-19] ----- glue.version:1574045422000 -----
2019-11-18 18:56:30 [sun.reflect.GeneratedMethodAccessor33#invoke]-[1]-[Thread-19] XXL-JOB, Hello World.
2019-11-18 18:56:30 [com.xx1.job.core.thread.JobThread#run]-[164]-[Thread-19]
----- xxl-job job execute_end(finish) -----
----- ReturnT:Return [code=200, msg=null, content=null]
2019-11-18 18:56:30 [com.xx1.job.core.thread.TriggerCallbackThread#callbackLog]-[190]-[xxl-job, executor TriggerCallbackThread]
----- xxl-job job callback finish.

[Load Log Finish]
```



```
2019-11-18 18:56:30 [com.xx1.job.core.thread.JobThread#run]-[124]-[Thread-19]
----- xxl-job job execute start -----
----- Param:
2019-11-18 18:56:30 [com.xx1.job.core.handler.impl.GlueJobHandler#execute]-[25]-[Thread-19] ----- glue.version:1574045422000 -----
2019-11-18 18:56:30 [sun.reflect.GeneratedMethodAccessor33#invoke]-[1]-[Thread-19] XXL-JOB, Hello World.
2019-11-18 18:56:30 [com.xx1.job.core.thread.JobThread#run]-[164]-[Thread-19]
----- xxl-job job execute_end(finish) -----
----- ReturnT:Return [code=200, msg=null, content=null]
2019-11-18 18:56:30 [com.xx1.job.core.thread.TriggerCallbackThread#callbackLog]-[190]-[xxl-job, executor TriggerCallbackThread]
----- xxl-job job callback finish.

[Load Log Finish]
```

（5）：测试

此时，我们将8081端口的调度中心服务停掉，现在8081端口的调度中心已经不能访问，访问8082端口的调度中心，查看调度日志，看任务是否可以正常调度：

The screenshot displays the XXL-JOB web interface. The top section, titled '任务调度中心' (Task Scheduling Center), includes a sidebar with navigation options like '运行报表', '任务管理', '调度日志', '执行器管理', '用户管理', and '使用教程'. The main area shows a '调度日志' (Scheduling Log) table with columns for '任务ID', '调度时间', '调度结果', '调度备注', '执行时间', '执行结果', '执行备注', and '操作'. Three rows of logs are visible, all with a '成功' (Success) status. Below this, the '执行日志 Console' (Execution Log Console) shows a detailed log of a task execution. A red box highlights a specific log entry: '2019-11-18 18:59:26 [com.xx1.job.core.thread.TriggerCallbackThread#callbackLog]-[190]-[xx1-job, executor TriggerCallbackThread] ----- xx1-job job callback error, errorMsg:io.netty.channel.AbstractChannel\$AnnotatedConnectException: Connection refused: no further information: /127.0.0.1:8081 ----- xx1-job job callback finish.'

通过日志，可以看到，任务Id为“7”的同一个任务，单机集群中两个调度中心的其中一个8081的调度中心服务挂掉了拒绝连接，但是任务依然可以正常调度，状态为200，说明调度中心实现了高可用，一个宕机之后，另外一个服务依然可以正常维持任务的调度。

6.1.2 任务调度高可用

6.1.2.1 概述

执行器支持集群部署，调度中心会感知到在线的所有执行器并进行心跳检测，会根据路由策略和阻塞处理策略进行任务调度，保障任务调度的高可用。

总的来说：

执行器如若集群部署，调度中心将会感知到在线的所有执行器，比如

“127.0.0.1:9997, 127.0.0.1:9998, 127.0.0.1:9999”。

当任务"路由策略"选择"故障转移(FAILOVER)"时，当调度中心每次发起调度请求时，会按照顺序对执行器发出心跳检测请求，第一个检测为存活状态的执行器将会被选定并发送调度请求，如果心跳检测失败则自动跳过，第二个依然心跳检测失败.....直至心跳检测第三个地址“127.0.0.1:9999”成功，选定为“目标执行器”，然后对“目标执行器”发送调度请求，调度流程结束，等待执行器回调执行结果。

调度成功后，可在日志监控界面查看“调度备注”，“调度备注”可以看出本地调度运行轨迹，执行器的“注册方式”、“地址列表”和任务的“路由策略”。

6.1.2.2 案例演示

说明：执行器集群的高可用已经在快速入门进行过演示，可参考2.3.3，此处不再重复演示。

6.2 安全性设计

6.2.1 概述

访问令牌 (AccessToken)

为提升系统安全性，调度中心和执行器进行安全性校验，双方AccessToken匹配才允许通讯；

调度中心和执行器，可通过配置项 "xxl.job.accessToken" 进行AccessToken的设置。

调度中心和执行器，如果需要正常通讯，只有两种设置：

- 设置一：调度中心和执行器，均不设置AccessToken；关闭安全性校验；

- 设置二：调度中心和执行器，设置了相同的AccessToken;

```
xxl-job-executor-sample-springboot\...\application.properties ×
1  # web port
2  server.port=9091
3
4  # log config
5  logging.config=classpath:logback.xml
6
7  ### 调度中心地址 [选填]：如调度中心集群部署存在多个地址则用逗号分隔。执行器将会使用该地址
8  xxl.job.admin.addresses=http://127.0.0.1:8081/xxl-job-admin
9
10 ### 执行器AppName [选填]：执行器心跳注册分组依据；为空则关闭自动注册
11 xxl.job.executor.appname=xxl-job-executor-sample
12
13 ### 执行器IP [选填]：默认为空表示自动获取IP，多网卡时可手动设置指定IP，该IP不会绑定Host
14 xxl.job.executor.ip=
15
16 ### 执行器端口号 [选填]：小于等于0则自动获取；默认端口为9999，单机部署多个执行器时，注意不要冲突
17 xxl.job.executor.port=9997
18
19 ### 执行器通讯TOKEN [选填]：非空时启用；
20 xxl.job.accessToken=
21
22 ### 执行器运行日志文件存储磁盘路径 [选填]：需要对该路径拥有读写权限；为空则使用默认路径
23 xxl.job.executor.logpath=/data/applogs/xxl-job/jobhandler
24
25 ### 执行器日志保存天数 [选填]：值大于3时生效，启用执行器Log文件定期清理功能，否则不生效
26 xxl.job.executor.logretentiondays=-1

xxl-job-admin\...\application.properties ×
13 spring.freemarker.request-context-attribute=request
14 spring.freemarker.settings.number_format=0.#####
15
16 ### mybatis
17 mybatis.mapper-locations=classpath:/mybatis-mapper/*Mapper.xml
18
19 ### xxl-job, datasource
20 spring.datasource.url=jdbc:mysql://127.0.0.1:3306/xxl_job?serverTimezone=
21 spring.datasource.username=root
22 spring.datasource.password=root
23 spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
24
25 spring.datasource.type=org.apache.tomcat.jdbc.pool.DataSource
26 spring.datasource.tomcat.max-wait=10000
27 spring.datasource.tomcat.max-active=30
28 spring.datasource.tomcat.test-on-borrow=true
29 spring.datasource.tomcat.validation-query=SELECT 1
30 spring.datasource.tomcat.validation-interval=30000
31
32 ### xxl-job email
33 spring.mail.host=smt.qq.com
34 spring.mail.port=25
35 spring.mail.username=xxx@qq.com
36 spring.mail.password=xxx
37 spring.mail.properties.mail.smtp.auth=true
38 spring.mail.properties.mail.smtp.starttls.enable=true
39 spring.mail.properties.mail.smtp.starttls.required=true
40 spring.mail.properties.mail.smtp.socketFactory.class=javax.net.ssl.SSLSo
41
42 ### xxl-job, access token
43 xxl.job.accessToken=
44 ### xxl-job, i18n (default empty as chinese, "en" as english)
45 xxl.job.i18n=
46
```

6.2.2 当调度中心和执行器的AccessToken都配置了，如果不一样，就会出现启动异常：“com.xxl.rpc.util.XxlRpcException: The access token[*] is wrong.”。

```
xxl-job-admin\...\application.properties ×
34 spring.mail.port=25
35 spring.mail.username=xxx@qq.com
36 spring.mail.password=xxx
37 spring.mail.properties.mail.smtp.auth=true
38 spring.mail.properties.mail.smtp.starttls.enable=true
39 spring.mail.properties.mail.smtp.starttls.required=true
40 spring.mail.properties.mail.smtp.socketFactory.class=javax.net.s
41
42 ### xxl-job, access token
43 xxl.job.accessToken=itcas
44
45 ### xxl-job, i18n (default empty as chinese, "en" as english)
46 xxl.job.i18n=
47

xxl-job-executor-sample-springboot\...\application.properties ×
7
8 ### xxl-job admin address list, such as "http://address" or
9 xxl.job.admin.addresses=http://127.0.0.1:8080/xxl-job-admin
10
11 ### xxl-job executor address
12 xxl.job.executor.appname=xxl-job-executor-sample
13 xxl.job.executor.ip=
14 xxl.job.executor.port=9999
15
16 ### xxl-job, access token
17 xxl.job.accessToken=itheima
18
19 ### xxl-job Log path
20 xxl.job.executor.logpath=/data/applogs/xxl-job/jobhandler
21 ### xxl-job Log retention days
22 xxl.job.executor.logretentiondays=-1
23

Endpoints
til.XxlRpcException: The access token[itheima] is wrong.
1.rpc.remoting.invoker.reference.XxlRpcReferenceBean$1.invoke(XxlRpcReferenceBean.java:221) <1 internal call>
1.job.core.thread.ExecutorRegistryThread$1.run(ExecutorRegistryThread.java:48)
ang.Thread.run(Thread.java:745)
logback [xxl-job, executor ExecutorRegistryThread] INFO c.x.p.r.XxlRpcReferenceBean - >>>>>>>> xxl-rpc, invoke error, address:http://127.0.0.1:8080/xxl-job-admin/api, XxlR
logback [xxl-job, executor ExecutorRegistryThread] INFO c.x.j.c.t.ExecutorRegistryThread - >>>>>>>> xxl-job registry error, registryParam:RegistryParam{registGroup='EXECUTOR',
til.XxlRpcException: The access token[itheima] is wrong.
1.rpc.remoting.invoker.reference.XxlRpcReferenceBean$1.invoke(XxlRpcReferenceBean.java:221) <1 internal call>
1.job.core.thread.ExecutorRegistryThread$1.run(ExecutorRegistryThread.java:48)
ang.Thread.run(Thread.java:745)
```

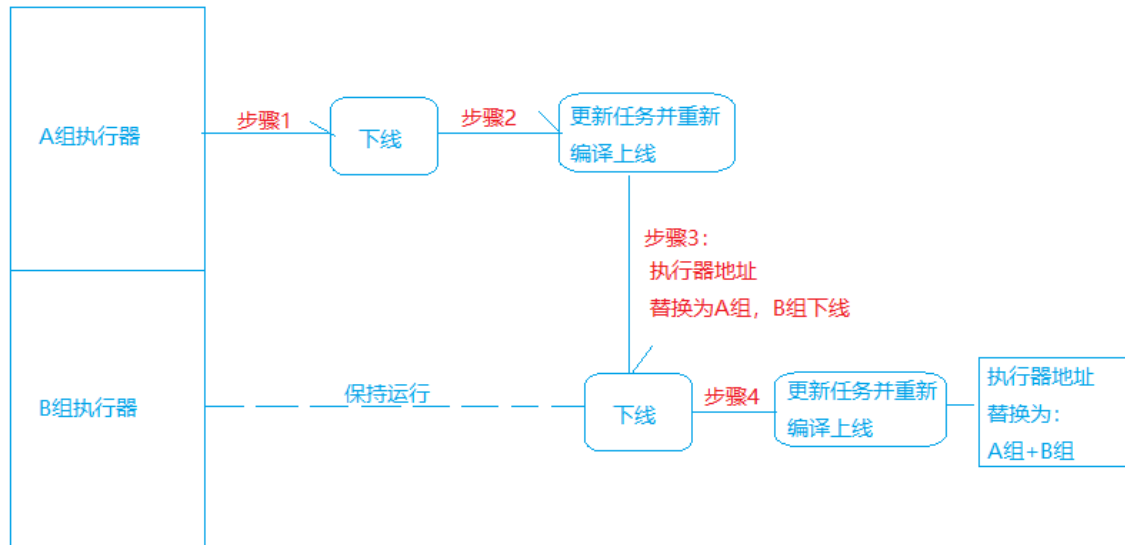
6.3 执行器灰度上线

6.3.1 概述

执行器灰度上线 执行器中托管运行着业务作业，作业上线和变更需要重启执行器，尤其是Bean模式任务，执行器重启可能会中断运行中的任务，但是，XXL-JOB得益于自建执行器与自建注册中心，可以通过灰度上线的方式，避免因重启导致的任务中断的问题。

步骤如下：

执行器集群 (手动注册)



- 1、执行器改为手动注册，下线一半机器列表（A组），线上运行另一半机器列表（B组）；
- 2、等待A组机器任务运行结束并更新后重新编译上线；执行器注册地址替换为A组；
- 3、下线B组，等待B组机器任务运行结束并更新后重新编译上线；执行器注册地址替换为A组+B组；

6.3.2 案例演示

(1) 在IDEA中配置启动项，并启动

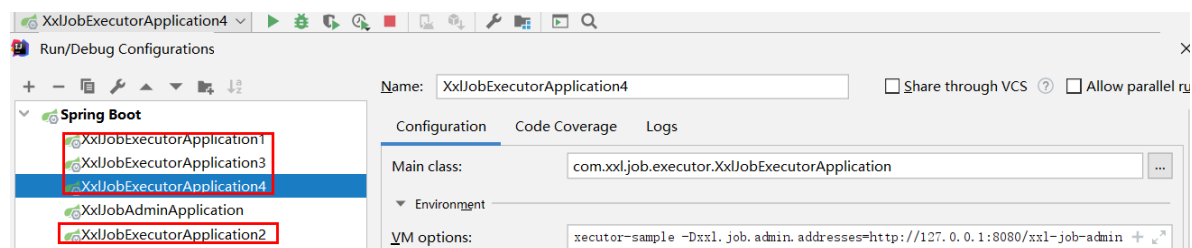
在IDEA工具栏打开编辑配置，配置4个执行器启动项，启动参数VM options依次加入如下配置：

```
-Dserver.port=9091 -Dxxl.job.executor.port=9991 -Dxxl.job.executor.appname=xxl-job-executor-sample -Dxxl.job.admin.addresses=http://127.0.0.1:8081/xxl-job-admin
```

```
-Dserver.port=9092 -Dxxl.job.executor.port=9992 -Dxxl.job.executor.appname=xxl-job-executor-sample -Dxxl.job.admin.addresses=http://127.0.0.1:8081/xxl-job-admin
```

```
-Dserver.port=9093 -Dxxl.job.executor.port=9993 -Dxxl.job.executor.appname=xxl-job-executor-sample -Dxxl.job.admin.addresses=http://127.0.0.1:8081/xxl-job-admin
```

```
-Dserver.port=9094 -Dxxl.job.executor.port=9994 -Dxxl.job.executor.appname=xxl-job-executor-sample -Dxxl.job.admin.addresses=http://127.0.0.1:8081/xxl-job-admin
```



(2) 调度中心的执行器管理，找到当前对应的执行器，将注册方式修改为手动注册，并录入上面4个执行器地址

编辑执行器

AppName*	xxl-job-executor-sample
名称*	示例执行器
排序*	1
注册方式*	<input type="radio"/> 自动注册 <input checked="" type="radio"/> 手动录入
机器地址*	127.0.0.1:9991,127.0.0.1:9992,127.0.0.1:9993,127.0.0.1:9994

保存

取消

(3) 启动一个任务，查看日志，任务正常执行成功，我们假定把9991和9992分为A组，9993和9994分为B组，如果需要重启，我们需要做如下操作：

先将执行器手动录入的列表删掉B组，然后B组下线；

修改完毕后，B组启动上线，手动在执行器列表录入B组，并删除A组，A组下线；

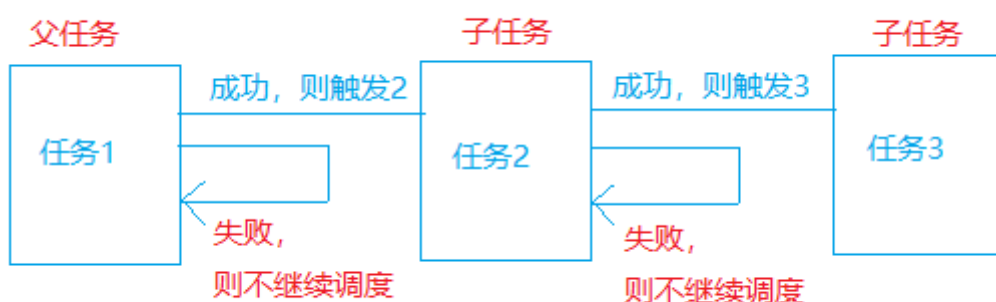
修改完毕后，A组上线，手动录入执行器列表；

上下线执行器小组时，我们可以观察任务执行日志，发现任务是成功执行的，并未中断，至此，我们的执行器灰度上线完成。

6.4 任务依赖

6.4.1 概述

任务依赖 原理：XXL-JOB中每个任务都对应有一个任务ID，同时，每个任务支持设置属性“子任务ID”，因此，通过“任务ID”可以匹配任务依赖关系。当父任务执行结束并且执行成功时，将会根据“子任务ID”匹配子任务依赖，如果匹配到子任务，将会主动触发一次子任务的执行。



实际的业务场景：

比如：用户注册成功后，发送一封激活邮件到用户邮箱，激活成功后发送一条短信到用户手机，中间任何一步失败，后续步骤就不会执行；

6.4.2 案例演示

(1) 先创建一个任务，作为子任务绑定到另外一个任务

更新任务

执行器*

示例执行器

路由策略*

第一个

运行模式*

BEAN

阻塞处理策略*

单机串行

任务超时时间*

0

负责人*

XXL

任务参数*

请输入任务参数

任务描述*

子任务1

Cron*

0/3 * * * * ?

JobHandler*

demo2JobHandler

子任务ID*

请输入子任务的任务ID,如存在多个则逗号分隔

失败重试次数*

0

报警邮件*

请输入报警邮件，多个邮件地址则逗号分隔

保存

取消

保存之后，注意新建任务的ID。

任务管理							
执行器	示例执行器	全部	请输入任务描述	请输入JobHandler	请输入负责人	搜索	新增
每页	10	条记录					
任务ID	任务描述	运行模式	Cron	负责人	状态	操作	
3	子任务1	BEAN: demo2JobHandler	0/3 * * * * ?	XXL	OSTOP	操作	

(2) 将子任务绑定到另外一个任务上，我们绑定到测试任务一

更新任务

执行器*

示例执行器

路由策略*

第一个

运行模式*

BEAN

阻塞处理策略*

单机串行

任务超时时间*

0

负责人*

XXL

任务参数*

请输入任务参数

任务描述*

测试任务1

Cron*

0/5 * * * * ?

JobHandler*

demoJobHandler

子任务ID*

3

失败重试次数*

0

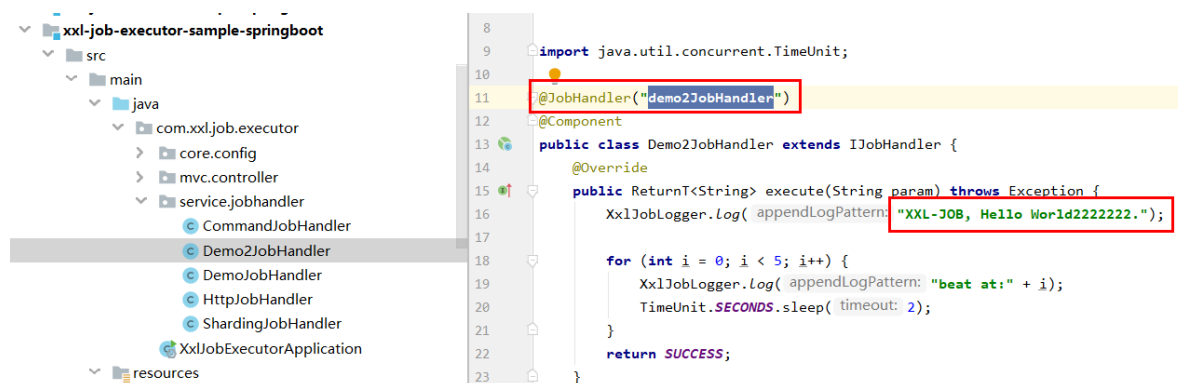
报警邮件*

请输入报警邮件，多个邮件地址则逗号分隔

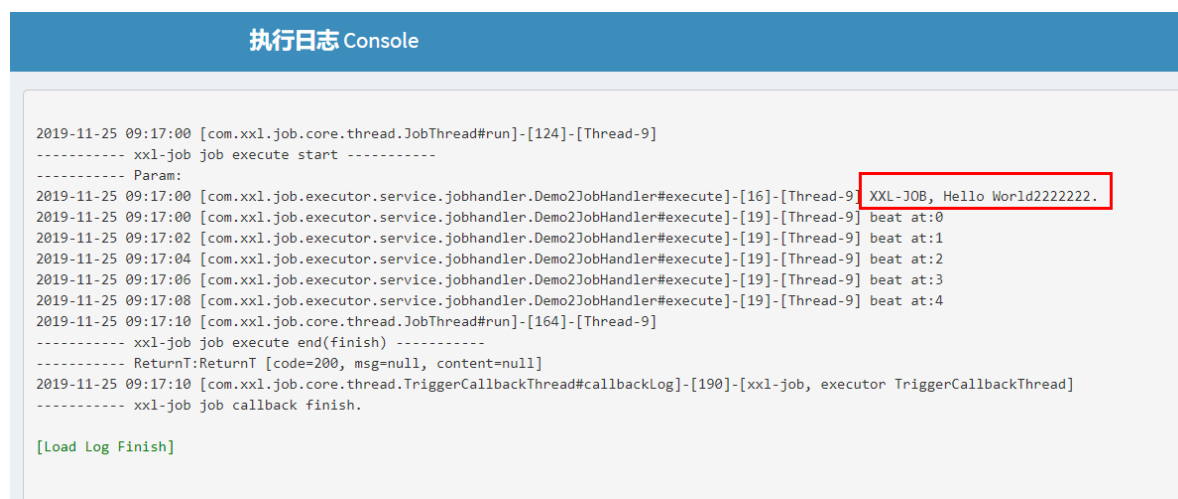
保存

取消

(3)执行测试，观察调度日志，查看测试任务1执行成功后，子任务1是否可以触发



执行日志:



结论：通过观察调度日志执行结果，发现父任务执行成功后，子任务也触发并执行，实现了XXL-JOB任务的依赖。

七、课程总结

通过以上的学习，我们也对XXL-JOB有了比较深入的了解，我们可以看出XXL-JOB有诸多优势，下面我们对XXL-JOB做一个回顾。

7.1 什么是任务调度

任务调度是指系统为了自动完成特定任务，在约定的时刻去执行某个任务的过程。有了任务调度就可以解放更多的人力，交由系统去自动执行任务。

7.2 什么是分布式的任务调度

软件架构正在逐步从单体应用转变为分布式架构，将单体结构分为若干服务，服务之间通过网络交互来完成用户的业务处理，这就是分布式；那么在分布式系统环境下运行任务调度，我们称之为分布式任务调度。

7.3 分布式任务调度解决了哪些问题

(1) 并行任务调度

并行任务调度实现靠多线程，如果有大量任务需要调度，此时光靠多线程就会有瓶颈了，因为一台计算机CPU的处理能力是有限的。

如果将任务调度程序分布式部署，每个结点还可以部署为集群，这样就可以让多台计算机共同去完成任务调度，我们可以将任务分割为若干个分片，由不同的实例并行执行，来提高任务调度的处理效率。

(2) 高可用

若某一个实例宕机，不影响其他实例来执行任务。

(3) 弹性扩容

当集群中增加实例就可以提高并执行任务的处理效率。

(4) 任务管理与监测

对系统中存在的所有定时任务进行统一的管理及监测。让开发人员及运维人员能够时刻了解任务执行情况，从而做出快速的应急处理响应。

(5) 避免任务重复执行

当任务调度以集群方式部署，同一个任务调度可能会执行多次，比如在上面提到的电商系统中到点发优惠券的例子，就会发放多次优惠券，对公司造成很多损失，所以我们需要控制相同的任务在多个运行实例上只执行一次。

7.4 XXL-JOB的概念和主要特性

概念：XXL-JOB是一个轻量级分布式任务调度框架，其核心设计目标是开发迅速、学习简单、轻量级、易扩展。现已开放源代码并接入多家公司线上产品线，开箱即用。

特性：

- **简单灵活** 提供Web页面对任务进行管理，管理系统支持用户管理、权限控制；支持容器部署；支持通过通用HTTP提供跨平台任务调度；
- **丰富的任务管理功能** 支持页面对任务CRUD操作；支持在页面编写脚本任务、命令行任务、Java代码任务并执行；支持任务级联编排，父任务执行结束后触发子任务执行；支持设置指定任务执行节点路由策略，包括轮询、随机、广播、故障转移、忙碌转移等；支持Cron方式、任务依赖、调度中心API接口方式触发任务执行
- **高性能** 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；
- **高可用** 任务调度中心、任务执行节点均 集群部署，支持动态扩展、故障转移 支持任务配置路由故障转移策略，执行器节点不可用是自动转移到其他节点执行 支持任务超时控制、失败重试配置 支持任务处理阻塞策略：调度当任务执行节点忙碌时来不及执行任务的处理策略，包括：串行、抛弃、覆盖策略
- **易于监控运维** 支持设置任务失败邮件告警，预留接口支持短信、钉钉告警；支持实时查看任务执行运行数据统计图表、任务进度监控数据、任务完整执行日志；

7.5 XXL-JOB的架构设计和执行原理是什么

XXL-JOB系统组成主要包含两大模块，调度模块和执行器模块：**调度模块：**

- 负责管理调度信息，支持可视化，包括任务的增删改查以及GLUE开发和任务报警，同时支持监控调度日志和执行日志，支持执行器故障转移等；
- 负责按照调度配置发出调度请求，支持集群部署，自身不承担业务代码；
- 调度中心包含：任务管理、执行器管理、日志管理、调度器、回调服务、注册服务等

执行器模块：

- 负责接收调度请求并执行任务逻辑，记录执行日志，上报执行结果，支持集群部署。

- 任务模块专注于任务的执行等操作。
- 执行器包含：执行器服务、调度请求队列、任务线程、日志服务、回调线程、注册线程等

总得来说，XXL-JOB的**设计思想**就是：

- 将调度行为抽象形成“调度中心”公共平台，而平台自身并不承担业务逻辑，“调度中心”负责发起调度请求。
- 将任务抽象成分散的JobHandler，交由“执行器”统一管理
- 执行器负责接收调度请求并执行对应的JobHandler中业务逻辑。

“调度”和“任务”两部分可以相互解耦，提高系统整体稳定性和扩展性，功能丰富的可视化管理界面，使开发和维护更加简单和高效；

7.6 XXL-JOB分片的概念-分片解决了什么问题

概念：作业分片是指任务的分布式执行，需要将一个任务拆分为多个独立的任务项，然后由分布式的应用实例分别执行某一个或几个分片项。

解决的问题：

- 分片项与业务处理解耦

XXL-JOB并不直接提供数据处理的功能，框架只会将分片项分配至各个运行中的作业服务器，开发者需要自行处理分片项与真实数据的对应关系。

- 最大限度利用资源

基于业务需求配置合理数量的执行器服务，合理设置分片，作业将会最大限度合理的利用分布式资源。