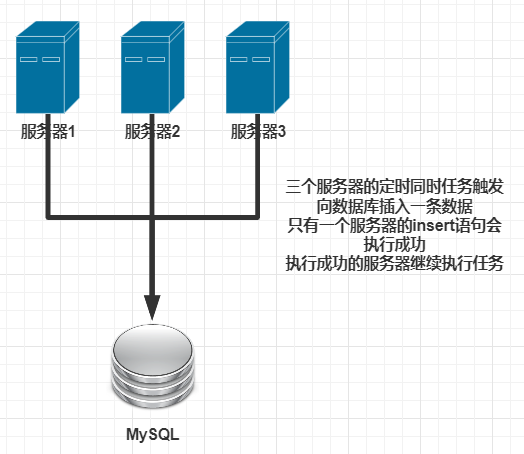
# 背景

消金电催 与 催收机器人 目前使用的是 quartz 封装的 一套分布式定时任务方案

（目前市面上的xxl-job、Elastic-Job 、saturn 等都是基于quartz 实现的分布式定时任务方案）

## 现在的定时任务,执行原理



优点：

* 操作简单

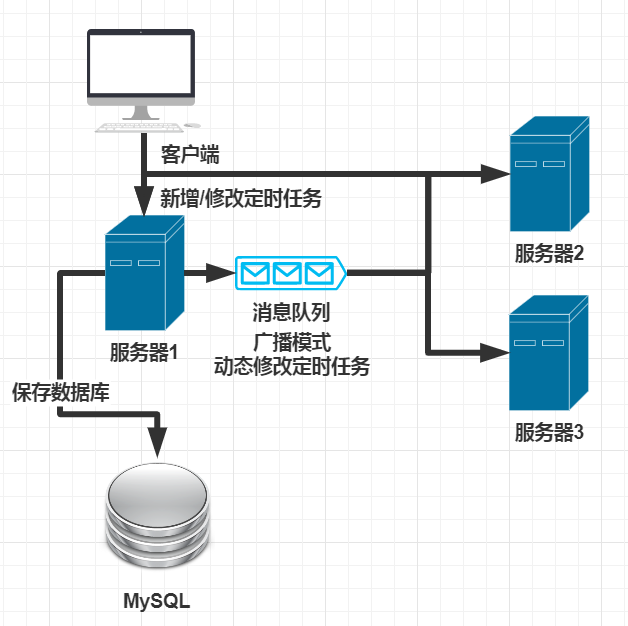
缺点：

* 一个任务一天只能执行一次，不好进行扩展
* 多次执行任务，需要删除原有的数据库记录

## 定时任务修改方案

当前的方案，用户在客户端修改/新增/停用/启用定时任务后，服务器将数据库中的定时任务数据修改成功后，通过rocketmq的广播模式通知所有的服务器，去动态修改已经创建好的定时任务实例。

确保了各个服务器上，定时任务的信息都是正确的。



# 任务修改--实现方式

## 存在的问题

通过rocketmq的广播模式通知所有的服务器，去动态修改已经创建好的定时任务实例。在普通的环境中是没有任何问题的。

但是当切换到k8s环境中的时候，一条广播模式的消息，第一次会正常的被订阅的消费者订阅。

生产中广播模式消息的过期时间为7天。7点内服务有重新上线后，k8s的pod节点的IP 或者 host发送了变化，这样的话，消息会被重新消费一次。

【重要】消费的时候，广播模式的消息是无序的，可能导致个别pod上的定时任务与实际的定时任务不一致。

## 方案

采用dubbo的集群Cluster的spi扩展机制

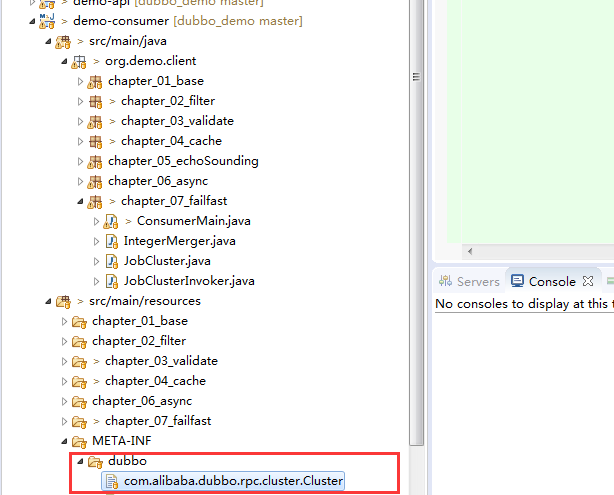
将rocketmq的广播模式通知所有的服务器，改为dubbo的广播模式通知所有的服务器。

优点是：

* 消息没有延迟；
* dubbo天然支持服务的广播模式（本次采用自定义广播模式，消除了原生方案的弊端）；
* 采用dubbo提供的结果合并方案，实现多服务节点的实时监控（现在使用Redis保存数据，有垃圾数据存在的可能）

## 扩展SPI

在classpath下添加spi扩展注册文件。用于注册扩展的类

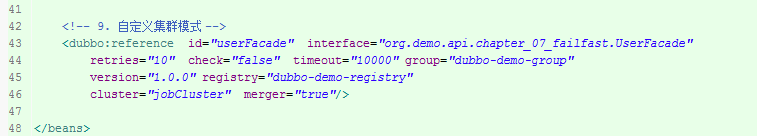


文件的内容如下

jobCluster=org.demo.client.chapter\_07\_failfast.JobCluster

## 配置集群策略

上面定义的集群策略名称为jobCluster。我们这样写就行：cluster=*"jobCluster"*

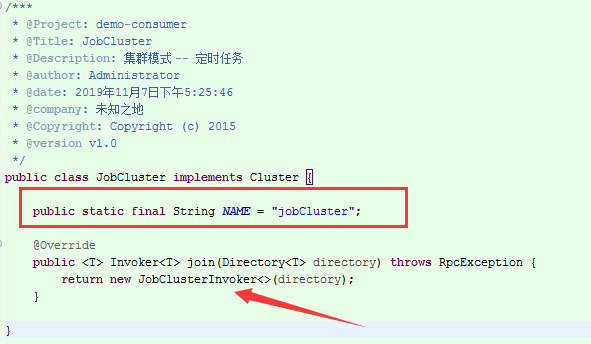


注意：配置之后，这个facade下的所有方法都使用配置的集群策略。

配置merger=*"true"* ，主要是为了将各个节点触发后的结果返回给客户端。

## JobCluster类

这个类很简单，主要是调用集群策略的代理类实现具体的方法



## JobClusterInvoker

* 第一段代码

这里为了方式错误使用集群策略。我们强行使用merger = "true"。没有配置这个属性的时候，走的是普通的dubbo容错的集群策略



* 第二段代码

轮询调用服务所有的，将各个服务的结果保存到map中，并把对象存储到 results 中。

采用异步的方式等待各个节点返回数据（这个地方的连接超时时间尽量调的大点）



* 第三段代码

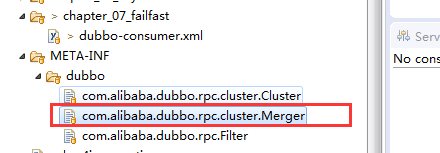
将多个服务器的结果进行合并



## 扩展合并处理类

* 扩展spi

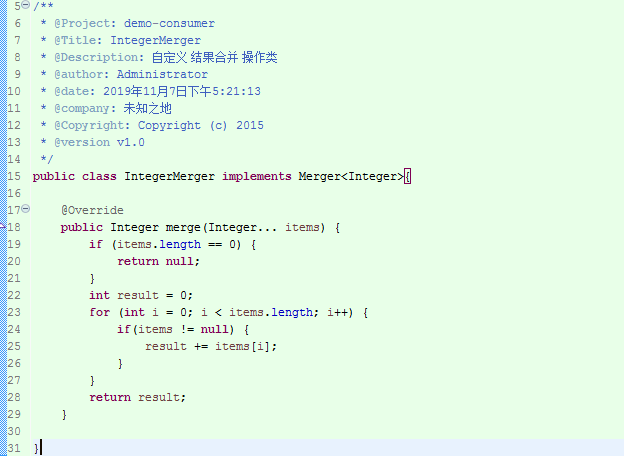
integer=org.demo.client.chapter\_07\_failfast.IntegerMerger



* 自定义处理类

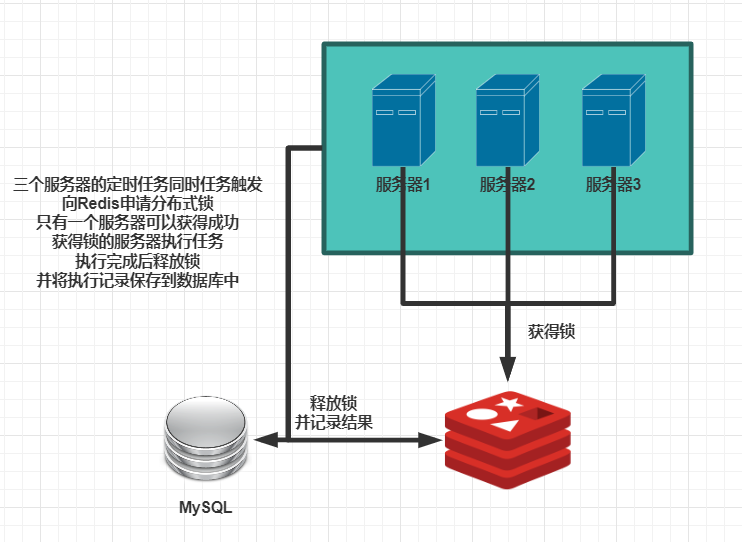
dubbo Merger 的实现类只支持 数组/集合类型

如果是其他的类型，比如 Integer,Double,Long, User等等不同的类型，需要自己实现Merger接口以及对其进行SPI扩展



# 任务执行--实现方式

## 定时任务--执行原理



优点：

* 一个任务一天可以执行多次
* 每次执行任务的记录都可以记录到数据库里面
* 当一个任务在执行中的时候，不能再次执行

缺点：

* 如果两个服务器存在时间差，可能出现一个任务执行多次的问题

如何预防上面的时间差：

这个问题：通过程序控制，确保服务器之间时间差在20s内（这个时间不包括跑批的时间），100%不行执行多次。本次优化，Redis分布式锁的过期时间为5 \* 60 = 300秒。

这里加上允许的20s时间差，以及任务跑批使用的时间，锁生效的时间为 ( 20, 300]。基本可以保证定时任务不会重复跑批。

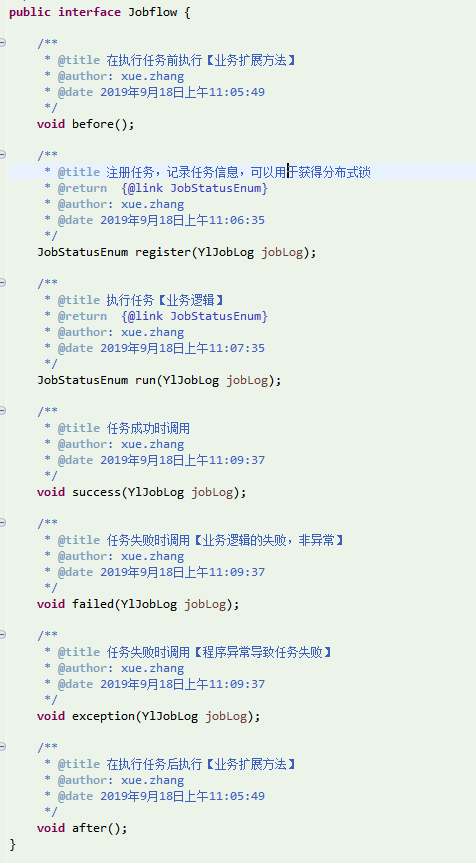
如果担心定时任务一天会执行多次，可以单独实现Jobflow#before()方法，进行这类需求的扩展。

新增一个定时任务执行的job超类，优化定时任务入口类的繁琐操作。目前只提供两种简单的实现方式。

* Jobflow
  + 是定时任务要实现的接口类。
* AbstractDistributedLockJob （接口实现的超类1）
  + 实现了Jobflow，负责主要的定时任务流程。
  + 使用分布式锁的定时任务流程（多实例的时候，只有一个机器会执行任务）
  + 业务跑批，继承AbstractDistributedLockJob类，实现 Jobflow#run(YlJobLog jobLog) 方法即可。
* AbstractNoLockJob（接口实现的超类2）
  + 实现了Jobflow，负责主要的定时任务流程。
  + 无锁任务（多实例的时候，所有的机器同时都会执行）
  + 业务跑批，继承AbstractNoLockJob类，实现 Jobflow#run(YlJobLog jobLog) 方法即可。

## 接口介绍

* Jobflow的主要为定时任务流程，方便后期不同类型的定时任务进行扩展



## 定时任务的状态

以枚举的方式列举状态，下面的 YL\_JOB\_LOG 表里面保存的执行状态

