

虚拟商品交易架构设计及源码实现

1.虚拟商品交易架构体系深入剖析

1.1.什么是虚拟电商

所谓虚拟电商，跟传统电商最大的特点就是没有实体物流配送，像充话费，充流量。比如对于传统电商而言买个笔记本电脑，支付完成后，后台需要走物流配送流程，而像话费充值这种虚拟商品，购买后不需要物流配送，但是需要和第三方接口（例如中国移动）对接，第三方充值成功或者失败后需要回调异步通知。

这类虚拟电商业务除了话费充值外还包括很多还包括加油卡，礼品卡，游戏充值，保险等，我们把这些虚拟商品的交易统称电商虚拟交易。

虚拟电商交易是对传统电商业务的一种补充，占据着非常重要的位置，随着社会技术的进步，未来会发展的更好，像某东/某宝上都有虚拟业务交易对应的板块。



两家企业在虚拟业务上都投入了大量的人力物力，足见虚拟业务的重要地位。

因此：虚拟电商交易类型的项目有一定业务价值和技术价值，值得探讨。

1.2.虚拟电商业务特点

商业上，需要业务人员提前和第三方签订合作协议，申请账号信息（调用第三方系统接口需要的身份凭据），充值（调用第三方接口需要付流量费用，涉及到对账问题）等，供应商的差额是虚拟业务的赢利点所在。

聚合平台对接：<https://www.juhe.cn/docs/api/id/85>

极速平台对接：<https://www.jisuapi.com/api/9>

技术上，需要根据不同的虚拟场景产生出不同策略组件，另外还涉及到一些第三方接口，在和第三方做接口对接的过程中，由于网络，双方系统的各种问题，会有延迟，重试，重试次数限制，轮转及最后订单同步成功失败等一系列问题。

2.交易策略系统架构

要设计什么样的交易策略才能保证能够满足虚拟交易业务的完整性呢？

2.1.虚拟交易系统策略架构

对于虚拟电商而言，它的很多业务在架构上跟传统电商是一样的，也就是说主体的电商架构都差不多，譬如：比如：商品，订单，收银/结算，优惠促销等；但是不同虚拟业务场景下外围的一些架构各有不同，由此也引申出针对不同虚拟场景的交易策略。

撮合交易策略：撮合交易是指证券经营机构受理投资者的买卖委托后，应即刻将信息按时间先后顺序传送到交易所主机，公开申报竞价。股票申报竞价时，可依有关规定采用集合竞价或连续竞价方式进行，交易所的撮合系统将按“价格优先，时间优先”的原则自动撮合成交。目前，沪、深两家交易所均存在集合竞价和连续竞价方式。

撮合系统在撮合时实际上是依据前一笔成交价而定出最新成交价的。如果前一笔成交价低于或等于卖出价，则最新成交价就是卖出价；如果前一笔成交价高于或等于买入价，则最新成交价就是买入价；如果前一笔成交价在卖出价与买入价之间，则最新成交价就是前一笔的成交价。

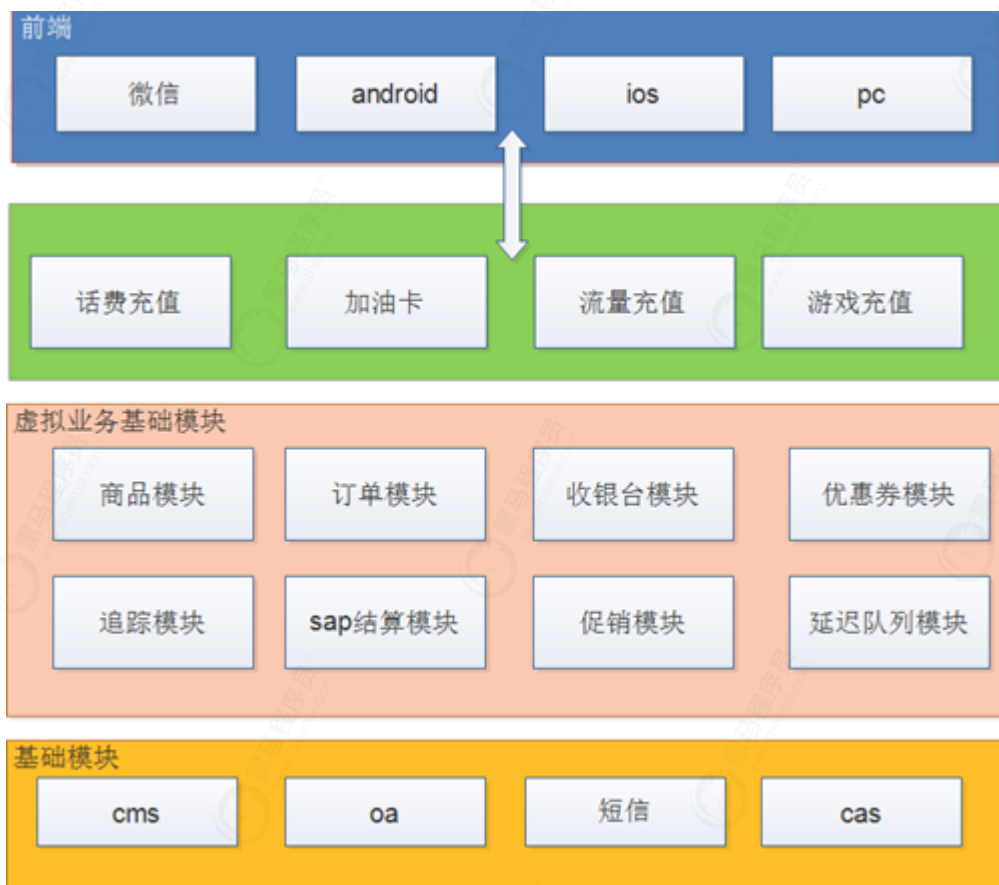
举例说明：

比如，买方出价1399点，卖方出价是1397点。如果前一笔成交价为1397或1397点以下，最新成交价就是1397点；如果前一笔成交价为1399或1399点以上，则最新成交价就是1399点；如果前一笔成交价是1398点，则最新成交价就是1398点。

推荐交易策略：推荐策略本身也不仅仅针对虚拟商品交易，几乎可以面向所有电商产品，而推荐策略系统会涉及到一些实时流数据的特征处理，对数据进行过滤，排序，解释，总之这中间会用到相应的一些算法引擎，数据经过算法引擎之后就会在特定的算法场景中有所体现，比如：商品推荐，活动推荐，店铺推荐，虚拟品类推荐，相似推荐等。

本课程中着重是以话费充值等这种类似的业务场景来说明一下在这个过程中的一些交易策略

类话费充值业务功能架构如下：



通过图可以看出在功能架构上跟传统的实体电商业务功能上有些是重合的，但也有自己特殊的功能架构，主要体现在虚拟业务技术实现上的一些特点，比如以话费充值为例，在技术实现上我们就需要考虑以下的一些特殊的业务场景：

1. 余额/押金导致的供应商轮转
2. 接口调用失败之后的重试
3. 重试次数的阈值限定，超过后的退款等业务
4. 对接成功，接口回调的后续业务处理
5. 主动的状态检查，接口回调的补偿

在这些业务场景的实现过程中衍生出了虚拟交易特殊的策略系统：**延迟任务系统**

2.2.延迟任务系统架构

2.2.1.什么是延迟任务

延迟任务：顾名思义，我们把需要延迟执行的任务叫做**延迟任务**。它是由事件触发的，任务可添加，延时执行，也可取消；有别于定时任务，定时任务往往是按照固定的时间规则周期性的执行。延迟任务的使用场景很丰富，譬如：

1. 红包 24 小时未被查收，需要延迟执行退还业务；
2. 订单下单之后 30 分钟后，用户如果没有付钱，系统需要自动取消订单；
3. 接口对接过程中因为网络等原因导致失败，1分钟后重试，直到成功，当然失败次数达阈值后取消重试

2.2.2.为何要自研延迟任务系统

延迟任务的实现方案有很多，方案本身没有好坏之分，和系统架构一样，关键是适合自身业务系统，不过随着市场业务的日新月异以及IT技术的不断翻新，有些市面上开源中间件已经不能满足大厂自身业务，很多大厂都是重新定制开发，下面我们从几种常见的延迟任务实现方案来看一下自研延迟任务系统的必要性：

- **Java API 实现延迟任务**

Java API 提供了两种实现延迟任务的方法：DelayQueue 和 ScheduledExecutorService

DelayQueue 是一个支持延时获取元素的阻塞队列，队列中的元素必须实现 Delayed 接口，并重写 `getDelay(TimeUnit)` 和 `compareTo(Delayed)` 方法；在创建元素时可以指定多久才可以从队列中获取当前元素，只有在延迟期满时才能从队列中提取元素。

使用ScheduledExecutorService中的 `scheduleWithFixedDelay(Runnable)` 方法以固定的频率一直执行任务，虽然可以延迟执行任务，但由于会以某个频率一直循环执行使得它更像定时任务。

优势：单机版，实现简单，

弊端：没办法做成一个通用组件，满足不了大型复杂的业务场景，无法高可用，另外此种方式能添加及消费任务但是无法友好的取消任务。

- **Redis 实现延迟任务**

借助zset 数据类型，把延迟任务存储在此数据集中，然后在开启一个无限循环查询当前时间的所有任务进行消费。

redis 有序集合sorted Set和集合Set一样也是string类型元素的集合,且不允许重复的成员。

不同的是每个元素都会关联一个double类型的分数。redis正是通过分数来为集合中的成员进行从小到大的排序。

存储任务数据时可将任务的执行时间当作元素的分数，这样所有任务在zset中按照执行时间会进行排序，并且redis支持按照分数来获取元素

优势：redis本身是一个通用的组件，现在开发一个系统基本也离不开它，用它来实现非常的方便，在一定的数据量下性能比较高

弊端：redis的zset既要完成去重，排序，又要承担着任务元素的添加，消费，移除等工作，在大型复杂的业务场景下，一旦缓存的数据量太大，性能和业务完整性无法保证。另外使用redis也无法做到保留任务的历史数据，如果基于它自身的持久化机制对性能上又有损耗。

- **MQ 实现延迟任务**

几乎所有的 MQ 中间件都可以实现延迟任务，或者更准确的叫法应该叫延时队列。

以rabbitmq为例，实现延迟队列的方式有两种：

- 1：通过消息过期后进入死信交换器，再由交换器转发到延迟消费队列，实现延迟功能。
- 2：使用 rabbitmq-delayed-message-exchange 插件实现延迟功能

优势：方案成熟，性能和可靠性上有保证

弊端：如果专门开启一个 MQ中间件来执行延迟任务，就有点杀鸡用宰牛刀般的奢侈了，除非系统已经有MQ环境了；另外使用MQ同样无法友好的取消任务，也无法保留任务的历史数据

- **定时任务框架实现延迟任务**

使用一些定时任务框架譬如：springTask,Quartz，这些都是功能比较强大的任务调度工具，可实现复杂的调度功能，

优势：不用去考虑实现的细节，套用框架即可，各方面细节都有所保证

弊端：对应用有一定侵入性，需要承担框架带来的维护成本，框架隐藏了实现的细节虽然方便但也为性能调优工作带来了困难。其次这些框架更适合去做定时任务。

总之：为了满足大型且复杂的业务场景，为了保证延迟任务系统的高性能，高可用，为了让延迟任务组件更通用化，我们需要自研一套延迟任务系统，我们自研的系统弥补了现有方案的短板，即支持任务的添加，消费，取消，持久化，大数据量下的性能和可靠性保证，普适性；同时也兼备现有方案的一些优点，使用简单，方便接入且无侵入。

3.系统数据存储设计

3.1.环境启动

1: docker相关环境启动

对于课程中所需要使用到的一些基础组件，已经通过docker部署完成，直接使用即可，比如：mysql，redis，zookeeper，consul，rabbitmq等

挂载课程资料中提供的 `docker` 虚拟机，并且登陆到这台虚拟主机，主机ip地址：192.168.200.129，登陆账号：root，登陆密码：itcast

登陆成功后在 `/root/virtual-trade` 目录下有提供好的使用 `docker-compose` 编排的各个容器，我们使用如下命令启动即可

```
1 docker-compose up -d
```

2: 项目工程环境启动

直接从课程资料中导入提供好的工程即可！

3.2.数据库设计

对于基础组件-延迟任务系统我们要保留任务的历史数据就必须要对任务数据进行持久化操作，因此我们选择了mysql

登陆远程129主机上的mysql，root账号的密码是：root123，查看表结构

```
1 CREATE TABLE `taskinfo` (  
2     `task_id` bigint(20) NOT NULL comment '任务id',  
3     `execute_time` datetime(3) NOT NULL comment '执行时间',  
4     `parameters` longblob comment '参数',  
5     `priority` int(11) NOT NULL comment '优先级',  
6     `task_type` int(11) NOT NULL comment '任务类型',  
7     PRIMARY KEY (`task_id`),  
8     KEY `index_taskinfo_time` (`task_type`,`priority`,`execute_time`)  
9 ) ENGINE=InnoDB DEFAULT CHARSET=utf8;  
10  
11 CREATE TABLE `taskinfo_logs` (  
12     `task_id` bigint(20) NOT NULL COMMENT '任务id',  
13     `execute_time` datetime(3) NOT NULL COMMENT '执行时间',
```

```

14 `parameters` longblob COMMENT '参数',
15 `priority` int(11) NOT NULL COMMENT '优先级',
16 `task_type` int(11) NOT NULL COMMENT '任务类型',
17 `version` int(11) NOT NULL COMMENT '版本号,用乐观锁',
18 `status` int(11) DEFAULT '0' COMMENT '状态 0=初始化状态 1=EXECUTED 2=CANCELLED',
19 PRIMARY KEY (`task_id`)
20 ) ENGINE=InnoDB DEFAULT CHARSET=utf8;

```

1 知识小贴士: mysql BLOB类型介绍

2 MySQL中, BLOB是一个二进制大型对象, 是一个可以存储大量数据的容器, 它能容纳不同大小的数据。BLOB类型实际是个类型系列 (TinyBlob、Blob、MediumBlob、LongBlob), 除了在存储的最大信息量上不同外, 他们是等同的

3 MySQL的四种BLOB类型:

类型	大小(单位: 字节)
4 TinyBlob	5 最大 255
6 Blob	7 最大 65K
8 MediumBlob	最大 16M
9 LongBlob	最大 4G

3.3.缓存存储结构设计

在延迟任务的设计过程中我们会用到redis的以下两种数据结构:

- 有序集合Sorted Set

Redis 有序集合和集合一样也是string类型元素的集合,且不允许重复的成员。

不同的是每个元素都会关联一个double类型的分数。redis正是通过分数来为集合中的成员进行从小到大的排序, 有序集合的成员是唯一的,但分数(score)却可以重复。

我们依然会用zset来存储一些任务数据, 但并不是全部数据, 是满足一定条件下的任务数据, 同时我们会按照任务的类型和优先级拆分成多个zset集合, 而不是简单的一个zset。

我们会用到的zset相关的命令大概有:

[\[ZADD key score1 member1 \[score2 member2\]\]](#): 向有序集合添加一个或多个成员, 或者更新已存在成员的分

[\[ZRANGEBYSCORE key min max \[WITHSCORES\] \[LIMIT\]\]](#): 通过分数返回有序集合指定区间内的成员

[\[ZREM key member \[member ...\]\]](#): 移除有序集合中的一个或多个成员

- 列表 List

Redis列表是简单的字符串列表, 按照插入顺序排序。你可以添加一个元素到列表的头部 (左边) 或者尾部 (右边)

我们利用List的特点去构造一个高速的任务消费队列, 保证任务消费的快速和高效

我们会用到的List相关的命令大概有:

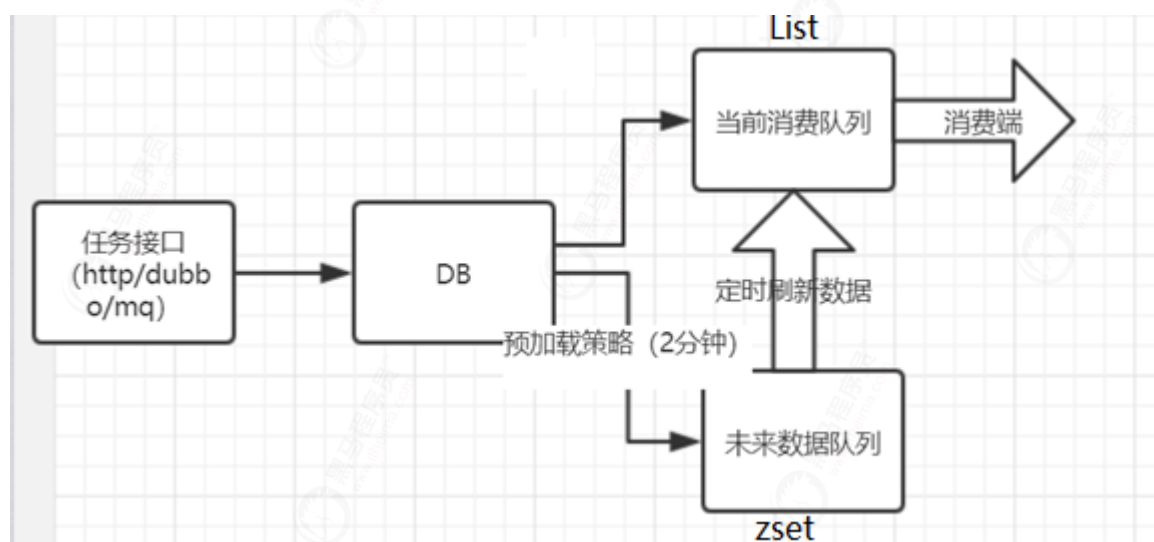
[\[LPUSH key value1 \[value2\]\]](#): 将一个或多个值插入到列表头部

[\[RPOP key\]](#): 移除并获取列表最后一个元素

4.系统基础组件架构设计与开发

4.1.架构设计

首先我们来看基础组件-延迟任务整体的架构设计图：



当其他系统或组件调用延迟任务系统的任务接口时：

- 1：任务数据会入库
- 2：根据任务的执行时间判断，如果任务需要被立即消费，则添加到当前消费队列，由消费端即时消费
- 3：如果任务是在未来2分钟内需要执行，则添加到未来数据集合，集合中的数据会根据任务的执行时间进行排序
- 4：如果任务是在未来2分钟外才执行的，数据只需暂时入库
- 5：使用一个定时任务实时的将未来数据集合中当前需要执行的任务数据刷新到当前消费者队列
- 6：使用一个定时任务每隔2分钟将数据库中未来2分钟内需要执行的数据预加载到未来数据集合中去进行排序等待
- 7：预加载的时间支持可配置
- 8：在redis中不论是未来数据集合zset还是当前消费者队列List我们需要按照任务的类型和优先级进行分组，减轻单一数据结构的压力，如下图：



8: 将延迟任务系统组件改造成一个服务，采用SpringCloud体系技术，注册中心采用consul，同时采用它作为配置中心，支持配置数据的动态刷新，改造成一个服务后对外只需暴露出任务对应的feign接口即可，其他系统接入使用非常的方便。

4.2.业务接口开发

4.2.1.添加任务接口

1: 在delay-task-service模块中创建接口：com.itheima.delaytask.inf.TaskService，并添加如下接口方法：

```
1 package com.itheima.delaytask.inf;
2 import com.itheima.delaytask.dto.Task;
3 import com.itheima.delaytask.exception.ScheduleSystemException;
4 import com.itheima.delaytask.exception.TaskNotExistException;
5
6 /**
7  * Created by 传智播客*黑马程序员.
8  */
9 public interface TaskService {
10     /**
11      * 添加任务
12      * @param task 任务对象
13      * @return 任务id
14      * @throws ScheduleSystemException
15      */
16     public long addTask(Task task) throws ScheduleSystemException;
17
18     /**
19      * 取消任务
20      * @param taskId 任务id
21      * @return 取消结果
22      * @throws TaskNotExistException
23      */
24     public boolean cancelTask(long taskId) throws
```

```

TaskNotExistException, ScheduleSystemException;
25
26  /**
27   * 拉取消费任务，消费的是当前需要执行的任务
28   * @return
29   * @throws TaskNotExistException
30   */
31  public Task poll() throws ScheduleSystemException;
32  }

```

2: 创建接口实现: com.itheima.delaytask.service.TaskServiceImpl

```

1  /**
2   * Created by 传智播客*黑马程序员.
3   */
4  @Service
5  @Slf4j
6  public class TaskServiceImpl implements TaskService {
7      @Autowired
8      private TaskInfoMapper taskInfoMapper;
9      @Autowired
10     private TaskInfoLogsMapper taskInfoLogsMapper;
11     @Autowired
12     private CacheService cacheService;
13
14     @Override
15     public long addTask(Task task) throws ScheduleSystemException {
16         return 0;
17     }
18     @Override
19     public boolean cancelTask(long taskId) throws
TaskNotExistException, ScheduleSystemException {
20         return false;
21     }
22     @Override
23     public Task poll() throws ScheduleSystemException {
24         return null;
25     }
26 }

```

3: 添加任务接口业务逻辑:

- 先将任务添加到数据库: boolean addTaskToDb(Task task)
- 添加成功后, 将任务添加到缓存: void addTaskToCache(Task task)

具体实现如下:

```

1  @Override
2  @Transactional
3  public long addTask(Task task) throws ScheduleSystemException {
4      /**
5       * 先将任务添加到数据库: boolean addTaskToDb(Task task)

```

```

6      * 添加成功后，将任务添加到缓存: void addTaskToCache(Task task)
7      */
8      boolean success = addTaskToDb(task);
9      if(success){
10         addTaskToCache(task);
11     }
12     return task.getTaskId();
13 }
14
15 private void addTaskToCache(Task task) {
16     //先暂时将数据存储到zset，后续会对此优化
17     cacheService.zAdd(DelayTaskRedisKeyConstants.DBCACHE,
JSON.toJSONString(task),task.getExecuteTime());
18 }
19
20 private boolean addTaskToDb(Task task) {
21     boolean success = false;
22     try {
23         //未执行任务入库
24         TaskInfo taskInfo = new TaskInfo();
25         taskInfo.setTaskType(task.getTaskType());
26         taskInfo.setPriority(task.getPriority());
27         taskInfo.setParameters(task.getParameters());
28         taskInfo.setExecuteTime(new Date(task.getExecuteTime()));
29
30         taskInfoMapper.insert(taskInfo);
31
32         //填充返回的主键值-任务的id
33         task.setTaskId(taskInfo.getTaskId());
34
35         //保存任务日志信息---任务日志和任务字段差不多，多了一个版本号和任务状态
36         TaskInfoLogs taskInfoLogs = new TaskInfoLogs();
37         BeanUtils.copyProperties(taskInfo,taskInfoLogs);
38         taskInfoLogs.setVersion(1);
39         taskInfoLogs.setStatus(TaskStatusConstants.SCHEDULED); //状态是刚提交的状态
40         taskInfoLogsMapper.insert(taskInfoLogs);
41         success = true;
42     } catch (Exception e) {
43         log.error("add task exception,task={}",task);
44         throw new ScheduleSystemException(e);
45     }
46     return success;
47 }

```

注意：

1：两个对象进行属性值传递，属性少建议用setter，速度最快，属性多用setter不优雅，可以使用属性拷贝工具，譬如：cglib BeanCopier，Spring beanUtils，Apache beanUtils，中都提供了类似的方法。

2：addTask方法需要添加事务：@Transactional

4.2.2.取消任务接口

1: 取消任务接口业务逻辑:

- 根据任务id更新数据库: Task updateDb(long taskId,int status): 更新任务日志表状态为已取消, 删除任务表数据
- 更新完成后返回任务对象Task, 从redis中删除任务数据: void removeTaskFromCache(Task task)

```
1  @Override
2  @Transactional
3  public boolean cancelTask(long taskId) throws TaskNotExistException, ScheduleSystemException
4  {
5      /**
6       * Task updateDb(taskId);删除任务表数据, 更新任务日志表状态为已取消
7       * void removeTaskFromCache(Task task),从redis中删除任务数据
8       */
9      boolean success = false;
10     Task task = updateDb(taskId, TaskStatusConstants.CANCELLED);
11     if(task!=null){
12         removeTaskFromCache(task);
13         success = true;
14     }
15     return success;
16 }
17 private void removeTaskFromCache(Task task) {
18     //暂时这样写, 后续优化
19     cacheService.zRemove(DelayTaskRedisKeyConstants.DBCACHE, JSON.toJSONString(task));
20 }
21
22 private Task updateDb(long taskId, int status) {
23     Task task=null;
24
25     //修改任务日志表中的状态,
26     TaskInfoLogs taskInfoLogs = taskInfoLogsMapper.selectById(taskId);
27     if(taskInfoLogs==null){
28         throw new TaskNotExistException("task not exist,taskId={}",taskId);
29     }
30
31     try {
32         //更改状态
33         UpdateWrapper<TaskInfoLogs> updateWrapper = new UpdateWrapper<>();
34         updateWrapper.lambda()
35             .set(TaskInfoLogs::getStatus, status)
36             .eq(TaskInfoLogs::getTaskId, taskInfoLogs.getTaskId());
37         taskInfoLogsMapper.update(null, updateWrapper);
38
39         //删除任务表数据,按照主键删除
40         taskInfoMapper.deleteById(taskId);
41         //构造返回
42         task = new Task();
43         //封装数据
44         BeanUtils.copyProperties(taskInfoLogs, task);
45         task.setExecuteTime(taskInfoLogs.getExecuteTime().getTime());
46     } catch (Exception e) {
```

```

47     log.error("cancel task exception,taskid={}",taskId);
48     throw new ScheduleSystemException(e);
49 }
50
51     return task;
52 }

```

注意：

1: cancelTask方法需要添加事务：@Transactional

4.2.3.消费任务接口

1: 消费任务接口业务逻辑：

```

1  @Override
2  @Transactional
3  public Task poll() throws ScheduleSystemException{
4      Task task = null;
5
6      //从zset集合中去拉取当前需要可执行的任务,按照分数获取即可,分数就是任务的执行时间
7      Set<String> byScore = cacheService.zRangeByScore(DelayTaskRedisKeyConstants.DBCACHE, 0,
8      System.currentTimeMillis());
9      if(byScore==null || byScore.size() ==0){
10         return null;
11     }
12     try {
13         //获取当前第一个任务
14         String task_json = byScore.iterator().next();
15         task = JSON.parseObject(task_json,Task.class);
16         //更新数据库中任务信息--删除任务表数据,日志状态改为已执行--复用之前的方法
17         updateDb(task.getTaskId(),TaskStatusConstants.EXECUTED);
18         //删除缓存中的该任务数据
19         cacheService.zRemove(DelayTaskRedisKeyConstants.DBCACHE,task_json);
20     } catch (Exception e) {
21         log.error("poll task exception,msg={}",e.getMessage());
22         throw new ScheduleSystemException(e);
23     }
24     return task;
25 }

```

注意：

1: poll方法添加事务：@Transactional

2: 我们一次消费只消费一个

4.2.4.业务接口测试

在delay-task-service模块下创建测试类：com.itheima.delaytask.TaskServiceTest

```

1  /**
2   * Created by 传智播客*黑马程序员.
3   */
4  @SpringBootTest
5  @RunWith(SpringRunner.class)
6  public class TaskServiceTest {
7
8      @Autowired
9      private TaskService taskService;
10
11     /**
12      * 测试添加任务:
13      * 添加三个任务, 执行时间分别为:当前, 10秒后, 20秒后
14      */
15     @Test
16     public void addTask(){
17         long currTime = System.currentTimeMillis();
18         for (int i=0;i<3;i++){
19             Task task = new Task();
20             task.setTaskType(1);
21             task.setPriority(1);
22             task.setExecuteTime(currTime+(i*10000));
23             task.setParameters("{\"orderNo\":\"12321321\", 'paytype': 'wx'}".getBytes());
24             long taskId = taskService.addTask(task);
25             System.out.println("任务id:"+taskId);
26         }
27     }
28     /**
29      * 测试取消任务
30      */
31     @Test
32     public void cancelTask(){
33         taskService.cancelTask(1263728301835403265L);
34     }
35     /**
36      * 测试消费任务
37      */
38     @Test
39     public void pollTask(){
40         addTask();
41         //每隔1秒钟消费一次
42         while (true){
43             Task task = taskService.poll();
44             if(task!=null){
45                 System.out.println("消费了一个任务:"+task+", 当前时间是:"+
LocalDateTime.now().format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));
46             }
47             try {
48                 TimeUnit.SECONDS.sleep(1);
49             } catch (InterruptedException e) {
50                 e.printStackTrace();
51             }
52         }

```

```
53     }
54 }
```

注意：

- 1：添加任务测试方法执行完成后可去数据库，缓存中查看是否有任务数据
- 2：取消任务时可选择取消最后一个任务，取消后去查看任务表数据是否删除，日志表状态是否改变，以及缓存中数据是否删除

4.3.业务接口缓存优化

4.3.1.消费者队列和未来队列设计

为什么要优化？

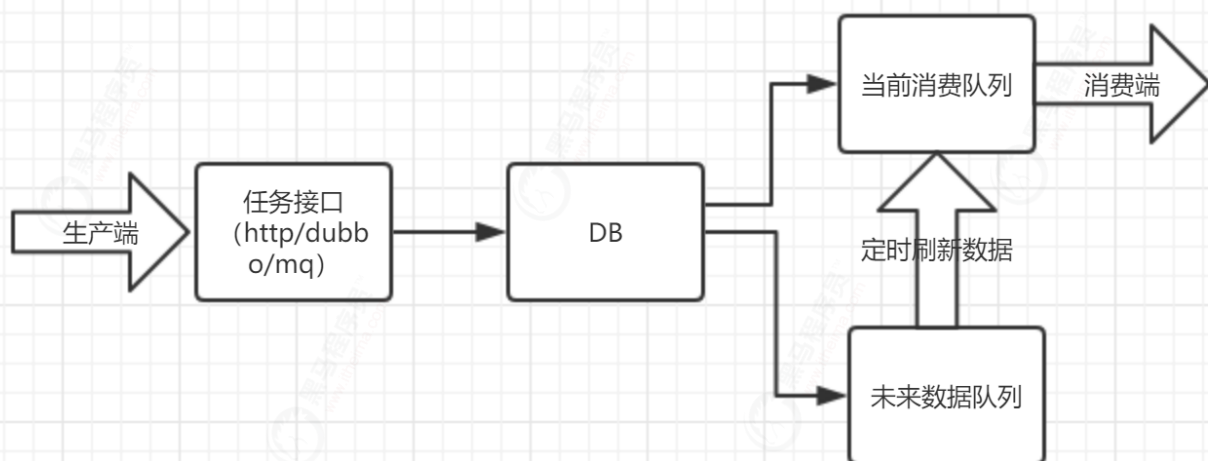
zset既充当了排序，去重，又要承担着任务元素的添加，消费，移除等工作，一旦缓存数据太大，性能和业务完整性无法保证；况且我们在redis中用一个zset集合也无法满足我们多元化场景的需求。

哪些地方可以优化？

- 1：Zset自身有性能缺陷，http://redisdoc.com/sorted_set/index.html，添加数据到zset和根据分数获取数据zRangebyScore操作的时间复杂度高；在操作的时间复杂度上我们希望能使用到一些O(1)复杂度的命令。
- 2：设计缺陷，所有任务都存储在一个zSet集合中，我们可以学习mq消息中间，根据主题细分，不同的主题（任务类型）划分到不同的集合中各自处理。

如何优化？

- 1：仍然需要用到zSet来排序，但是我们可以按照任务类型和优先级拆分成多个zSet集合，将不同的任务存储到不同的zSet集合中
- 2：设计未来数据集合zset和消费者队列List，根据任务执行时间来区分，立马可以执行的就直接放入消费者队列去消费，不需要在放入zset中去排序了，List数据存取的时间复杂度都是O(1)；未来某一时间才执行的任务放入zSet集合中排序等待消费。



- 3：消费者队列也可以按照任务类型和优先级拆分成多个，减轻一个队列的压力

4.3.2.业务接口改造

1: 添加任务接口改造

我们主要找到添加任务时操作缓存的部分：addTaskToCache 方法，

- 根据任务执行时间进行判断任务是在未来数据集合zset中还是在当前消费者队列List中
- 不论是zset还是List都需要根据任务类型和优先级进行分组

```
1 private void addTaskToCache(Task task) {
2     //要根据任务类型和优先级进行分组，每组的key不一样
3     String key = task.getTaskType()+"_"+task.getPriority();
4     //根据时间判断
5     if(task.getExecuteTime() <= System.currentTimeMillis()){
6         //任务进入消费者队列List---数据从左侧入队列，消费时从右侧出队列--保证任务消费的一个顺序性
7         //同时我们让所有分组的key都有一个相同的前缀，方便进行管理
8
9         cacheService.lLeftPush(DelayTaskRedisKeyConstants.TOPIC+key, JSON.toJSONString(task));
10    }else {
11        //任务进入zset排序等待
12
13        cacheService.zAdd(DelayTaskRedisKeyConstants.FUTURE+key, JSON.toJSONString(task), task.getExecuteTime());
14    }
15
16    //先暂时将数据存储到zset，后续会对此优化
17    //cacheService.zAdd(DelayTaskRedisKeyConstants.DBCACHE,
18    JSON.toJSONString(task), task.getExecuteTime());
19 }
```

2: 取消任务接口改造

找到取消任务操作缓存的部分：removeTaskFromCache 方法

- 根据任务执行时间判断，看需要从哪个数据结构中去删除任务数据

```
1 private void removeTaskFromCache(Task task) {
2     String key = task.getTaskType()+"_"+task.getPriority();
3     //判断
4     if(task.getExecuteTime() <= System.currentTimeMillis()){
5         //从消费者队列中移除---
6
7         cacheService.lRemove(DelayTaskRedisKeyConstants.TOPIC+key, 0, JSON.toJSONString(task));
8     }else {
9         cacheService.zRemove(DelayTaskRedisKeyConstants.FUTURE+key, JSON.toJSONString(task));
10    }
11
12    //暂时这样写，后续优化
13    //cacheService.zRemove(DelayTaskRedisKeyConstants.DBCACHE, JSON.toJSONString(task));
14 }
```

3: 消费任务接口改造

现在任务不论是在set中还是在list中都是按照任务类型和优先级进行了分组，所以我们消费任务的时候也需要分组进行消费。

1: 首先需要修改TaskService中的消费接口方法的声明，注释掉原有的poll方法，添加新的poll方法

```
1  /**
2   * 拉取任务
3   * @return
4   * @throws TaskNotExistException
5   */
6  public Task poll() throws ScheduleSystemException;
7
8  /**
9   * 按照类型和优先级来拉取任务
10  * @param type
11  * @param priority
12  * @return
13  * @throws ScheduleSystemException
14  */
15  public Task poll(int type,int priority) throws ScheduleSystemException;
```

2: 在接口的实现类中先注释掉原有的方法实现，然后来实现新的方法

- 从消费者队列List中右侧pop最外层元素
- 更新数据库日志表中该任务的状态为已消费同时删除任务表中数据

```
1  @Override
2  @Transactional
3  public Task poll(int type, int priority) throws ScheduleSystemException {
4      Task task = null;
5
6      //消费时只需从消费者队列List中右侧pop最外层元素即可
7      String key = type+"_"+priority;
8      String task_json = cacheService.lRightPop(DelayTaskRedisKeyConstants.TOPIC + key);
9      if(StringUtils.isEmpty(task_json)){
10         return null;
11     }
12     try {
13         task = JSON.parseObject(task_json,Task.class);
14         //更新数据库日志表中该任务的状态为已消费同时删除任务表中数据
15         updateDb(task.getTaskId(),TaskStatusConstants.EXECUTED);
16     } catch (Exception e) {
17         log.error("poll task exception,type={},priority={}",task,priority);
18         throw new ScheduleSystemException(e);
19     }
20     return task;
21 }
```

注意：方法需要添加事务

4: 测试

回到测试类TaskServiceTest，修改测试方法pollTask，然后测试，测试之前清空数据库和缓存信息。

注意：

- 1: 测试添加后查看任务数据是否是一个在当前消费者队列，两个在未来数据集合
- 2: 测试取消，比如：取消第三个，
- 3: 清空数据后，再次添加3个任务，然后测试消费

问题思考：还有两个任务为什么一直没有被消费？

4.3.3.未来数据定时刷新

现在出现的问题是：未来数据集合中的任务到执行时间后并不会进入到消费者队列中去被消费，

解决方案：可以采用一个定时任务去实现，定时的将未来数据集合中当前需要执行的任务刷新到消费者队列。直接使用Spring 中提供的 `@Scheduled` 注解实现定时执行某个方法。

- 1: 添加定时任务

在TaskService接口实现中添加一个定时刷新的方法：`void refresh()`，方法执行的时间规则是每隔1秒钟执行一次，也就是说每隔1秒钟将未来数据集合中当前需要执行的所有任务刷新到消费者队列中，该时间规则由cron表达式指定

```
1 @Scheduled(cron = "*/1 * * * * ?")
2 public void refresh(){
3
4 }
```

注意：

- 1: 方法需要添加`@Scheduled`
- 2: 需要在启动类上添加开启定时的注解：`@EnableScheduling`

2: 定时刷新的业务逻辑是什么样的呢？

192.168.200.129

db0 (5/0)

- future_2000_1
- future_2000_2
- topic_1000_2
- topic_2000_2
- topic_3000_2

- 因为redis是以key-value结构存储数据，首先要从redis中找到属于未来数据集合的所有key，这些key都有一个特点，就是以 `future_` 开头。
- 然后再获取这些key对应的value，其实就是存储任务的zset集合，从这些集合中找到当前需要执行的任务数据
- 将这些任务数据存入到对应分组的消费者队列List中，key都是以 `topic_` 开头，然后从zset中删除这些任务数据

这个过程的技术难点是什么？

技术难点在于：如何从redis中找到属于未来数据集合的所有key；

这里有两种实现方案：

方案1：keys 模糊匹配

官方文档：<http://redisdoc.com/database/keys.html>

```
1 keys future_*
```

keys的模糊匹配功能很方便也很强大，但是在生产环境需要慎用！开发中使用keys的模糊匹配却发现redis的CPU使用率极高，所以一般很多企业在生产环境将keys命令禁用了

方案2：scan

官方文档：<http://redisdoc.com/database/scan.html>

scan及相关的命令不仅可以去迭代扫描redis中的key，也可以去扫描某个key下的value

SCAN 命令是一个基于游标的迭代器，SCAN 命令每次被调用之后，都会向用户返回一个新的游标，用户在下次迭代时需要使用这个新游标作为 SCAN 命令的游标参数，以此来延续之前的迭代过程。

```
1 scan 0 MATCH future_* COUNT 1000
```

我们提供的CacheService缓存工具类中也封装了对keys和scan的相关操作，可在对应工具类中查看该方法

3：定时刷新逻辑实现

- 使用scan找到未来数据集合的所有key
- 根据每一个key从该分组下 获取当前需要执行的任务数据Set集合
- 将这组数据添加到消费者队列的对应分组上，并从zset中移除

```
1 @Scheduled(cron = "*/1 * * * * ?")
2 public void refresh(){
3     //找到未来数据集合的所有key
4     Set<String> future_keys =
5     cacheService.scan(DelayTaskRedisKeyConstants.FUTURE+"*");//scan future_*
6     if(future_keys==null || future_keys.size() ==0){
7         return;
8     }
9     for (String future_key : future_keys) {
10        //根据每一个key从该分组下 获取当前需要执行的任务数据Set集合
11        Set<String> values = cacheService.zRangeByScore(future_key, 0,
12        System.currentTimeMillis());
13        if(values == null || values.size()==0){
14            continue;
15        }
16        //将这组数据添加到消费者队列的对应分组上，并从zset中移除
17        String topicKey =
```

```

DelayTaskRedisKeyConstants.TOPIC+future_key.split(DelayTaskRedisKeyConstants.FUTURE)
[1]; //future_2001_1
17     for (String value : values) {
18         cacheService.lLeftPush(topicKey,value);
19         cacheService.zRemove(future_key,value);
20     }
21 }
22 }

```

4: 测试

直接执行TaskServiceTest测试类中的消费任务方法：pollTask即可，

4.3.4.redis管道优化

现在对队列按照类型+优先级做了拆分，来降低单一队列数据量，来提高性能，不过如果在某一秒内的需要处理的任务还是过多，刷新会不及时，这样会造成任务系统延迟，虽然从业务角度，可以忍受1,2秒延迟，不过从技术角度还是值得去深究，力争最大优化。

哪个地方还可以优化？

在定时刷新方法refresh()中，根据每一个key获得到该分组下的所有任务数据集合后，循环的向消费者队列去lLeftPush任务数据，并将任务数据从未来数据集合中移除，每次循环的过程中都需要和redis交互一次，在这个过程中增加了性能消耗

```

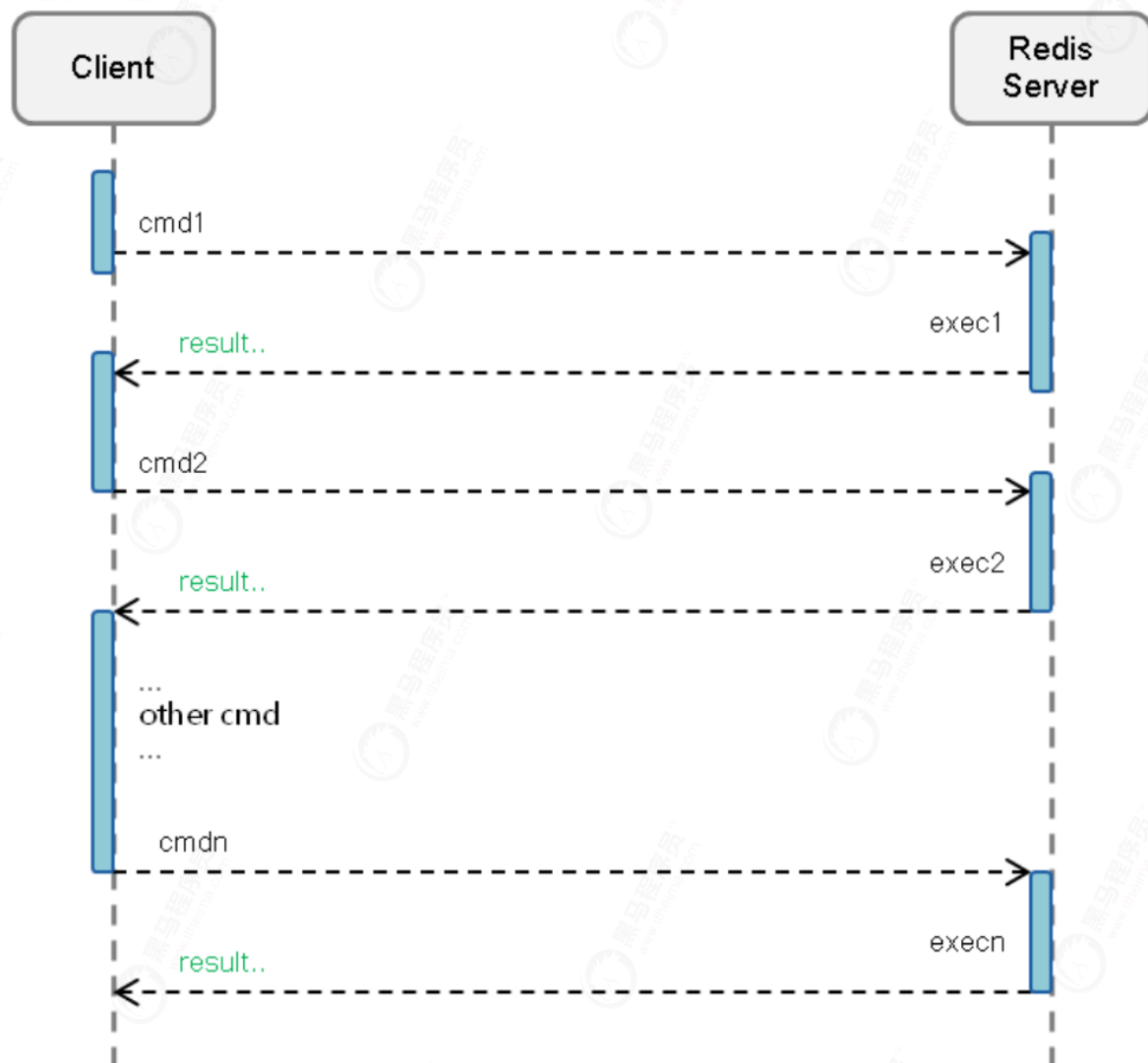
1  for (String value : values) {
2      cacheService.lLeftPush(topicKey,value);
3      cacheService.zRemove(future_key,value);
4  }

```

什么是reids管道技术？

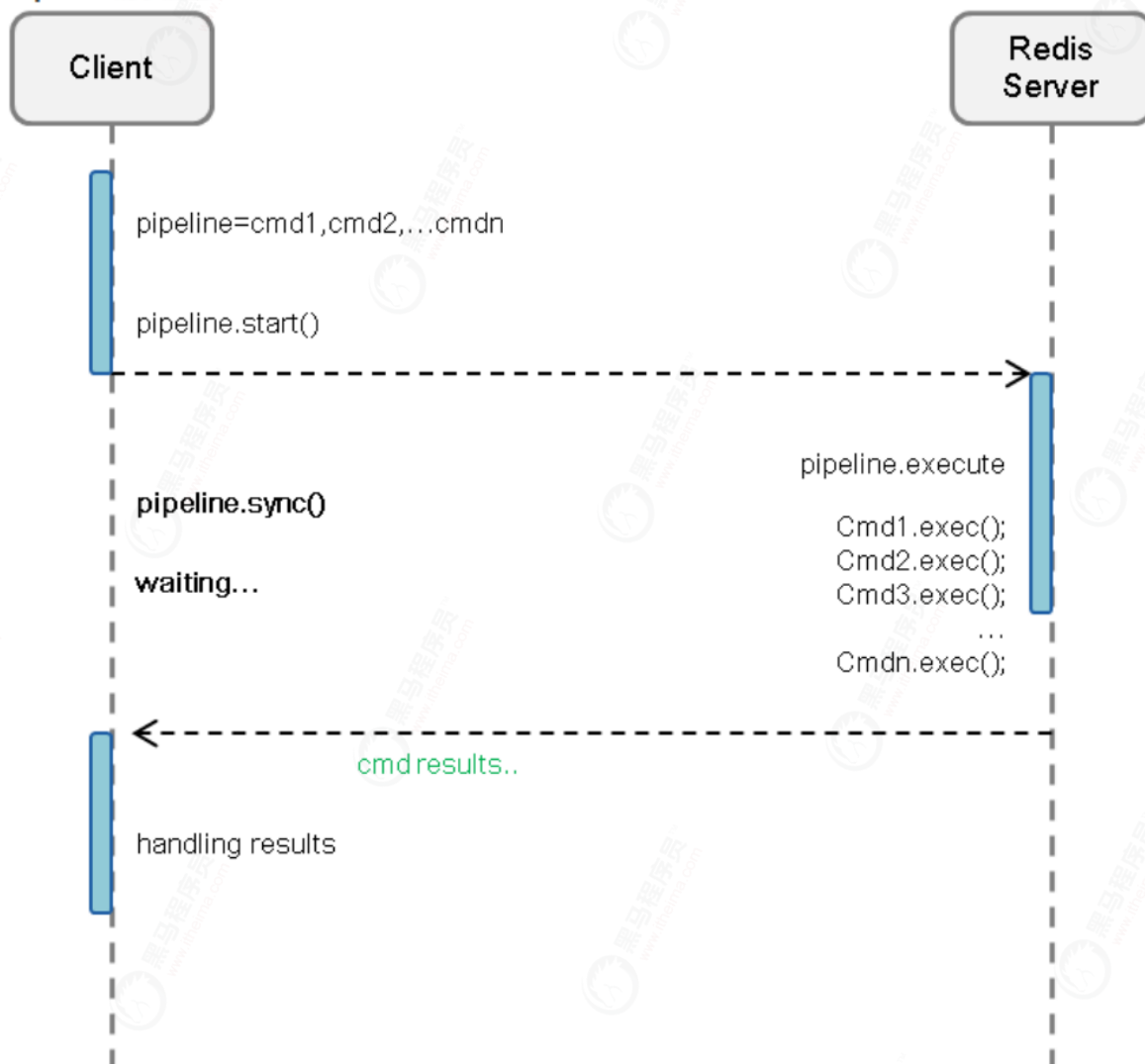
普通redis客户端和服务端交互模式

普通请求模型



普通的请求模型是同步的，每次请求对应一次IO操作等待；redis为了提高性能，提供了管道（Pipeline）技术后，可以将命令合并为一次IO，除了时延可以降低之外，还能大幅度提升系统吞吐量。

Pipeline请求模型



spring 封装的 `redisTemplate`，在常规调用时候，一般直接调用 `opsfor..` 来操作 Redis 数据库，每执行一条命令是要重新拿一个连接，因此很耗资源，`executePipelined` 方法 专门用来支持管道开发，可以将批量执行的命令 封装到 `RedisCallback`

管道的一个核心思想是：将一些批量操作放到管道中一同执行，而无需循环的调用 redis 来执行。

定时刷新管道优化

将每个分组 key 下当前需要执行的任务数据刷新到消费者队列 List 并且从未来数据集 zset 中移除的这部分循环批量操作改造成使用 redis 的管道技术来实现，因此需要去改造我们的缓存操作工具类，步骤如下

1：在 common 工程中的 `CacheService` 工具类中，添加一个方法将未来数据集添加到消费者队列并删除未来数据集中的数据，使用 reids 的管道技术

```
1 public List<Object> refreshWithPipeline(String future_key,String
2   topick_key,Collection<String> values){
3     List<Object> results = stringRedisTemplate.executePipelined(new RedisCallback<Object>()
4     {
5         @Nullable
```

```

5      @Override
6      public Object doInRedis(RedisConnection redisConnection) throws DataAccessException
7      {
8          //拿到连接StringRedisConnection,stringRedisTemplate和RedisTemplate所使用的
            Connection不一样,各自互不影响
9          StringRedisConnection stringRedisConnection =
            (StringRedisConnection)redisConnection;
10         String[] strings = values.stream().toArray(String[]::new);
11         //将任务数据添加到消费者队列
            stringRedisConnection.lPush(topick_key,strings);
12         //将任务数据从未来数据集合中移除
            stringRedisConnection.zRem(future_key,strings);
13
14         //因为此次操作没有数据需要返回所以return null
15         return null;
16     }
17 }
18 });
19 return results;
20 }

```

2: 改造定时刷新方法: refresh, 如下

```

1  @Scheduled(cron = "*/1 * * * * ?")
2  public void refresh(){
3      //找到未来数据集合的所有key
4      Set<String> future_keys =
            cacheService.scan(DelayTaskRedisKeyConstants.FUTURE+"*");//scan future_*
5      if(future_keys==null || future_keys.size() ==0){
6          return;
7      }
8      for (String future_key : future_keys) {
9          //根据每一个key从该分组下 获取当前需要执行的任务数据Set集合
10         Set<String> values = cacheService.zRangeByScore(future_key, 0,
            System.currentTimeMillis());
11         if(values == null || values.size()==0){
12             continue;
13         }
14         //将这组数据添加到消费者队列的对应分组上, 并从zset中移除
15         String topicKey =
            DelayTaskRedisKeyConstants.TOPIC+future_key.split(DelayTaskRedisKeyConstants.FUTURE)
            [1];//future_2001_1
16         /*for (String value : values) {
17             cacheService.lLeftPush(topicKey,value);
18             cacheService.zRemove(future_key,value);
19         }*/
20         cacheService.refreshWithPipeline(future_key,topicKey,values);//只需此一句,改造完成
21     }
22 }

```

3: 测试: 业务的完整性

清空数据库，缓存所有数据，然后找到测试类：TaskServiceTest，执行添加任务后再执行消费任务，看业务是否完成

4: 测试：性能是否提升

性能上是否有提升？，直接在TaskServiceTest中使用如下测试代码进行测试

```
1  @Autowired
2  private CacheService cacheService;
3
4  public List<String> initData(){
5      List<String> list = new ArrayList<String>();
6      //向未来数据集添加1万任务数据
7      for (int i = 0; i < 10000 ; i++) {
8          Task task = new Task();
9          task.setExecuteTime(new Date().getTime());
10         task.setTaskType(1001);
11         task.setPriority(1);
12         list.add(JSON.toJSONString(task));
13         cacheService.zAdd("future_1001_1",JSON.toJSONString(task),task.getExecuteTime());
14     }
15     return list;
16 }
17 @Test
18 public void refreshPipelineTest(){
19     List<String> list = initData();
20
21     long start=System.currentTimeMillis();
22     //使用原始方式:将未来数据集中的任务数据添加到消费者队列中并从未来数据集中移除该任务数据
23     for (String value : list) {
24         cacheService.lLeftPush("topic_1001_1",value);
25         cacheService.zRemove("future_1001_1",value);
26     }
27     System.out.println("未使用管道耗时" + (System.currentTimeMillis()-start));
28     list = initData();
29     start=System.currentTimeMillis();
30     cacheService.refreshWithPipeline("future_1001_1", "topic_1001_1", list);
31     System.out.println("使用管道耗时" + (System.currentTimeMillis()-start));
32 }
```

查看输出结果，对比性能差异！

4.4.业务接口线程池优化

4.4.1.启动时数据恢复

什么叫数据恢复？

场景：因为一些特殊的问题，特别是缓存问题可能会导致数据库中的任务信息和缓存中的任务信息数据不同步，也就造成了一致性问题，譬如缓存引发的宕机等，

思路：为了解决类似的问题，我们可以在系统启动后立即开启数据库和缓存的同步，将数据库中的待执行任务加载到缓存，有一个注解：`@PostConstruct` 可以帮助我们在Bean初始化后完成某方法的执行

执行的顺序：构造方法----->@Autowired----->@PostConstruct

实现：

1：在TaskServiceImpl中编写一个方法：syncData 并添加上 @PostConstruct 注解

2：整体的实现逻辑是：

- 首先清除缓存中原有的一些数据，因为这部分数据可能是脏数据：clearCache()
- 从数据库任务表中查询所有待执行任务，按分组进行查询，因为任务在缓存中是进行分组的

查询数据库如何对任务分组？

分组sql语句：

```
1 select task_type, priority from taskinfo group by task_type,priority
```

然后按照每组的任务类型type和优先级priority去查询该分组下的所有任务数据即可

- 将每组的任务数据添加到缓存，调用之前写好的addTaskToCache(Task)方法，

```
1 @PostConstruct
2 public void syncData(){
3
4     //先清除缓存中的原有数据
5     clearCache();
6
7     //查询任务表中的所有分组
8     QueryWrapper<TaskInfo> queryWrapper = new QueryWrapper<>();
9     queryWrapper.lambda()
10         .select(TaskInfo::getTaskType,TaskInfo::getPriority)
11         .groupBy(TaskInfo::getTaskType,TaskInfo::getPriority);
12     List<TaskInfo> group_tasks = taskInfoMapper.selectList(queryWrapper);
13     //获取每个分组下的任务数据将其添加到缓存
14     if(Objects.isNull(group_tasks)){
15         return;
16     }
17     for (TaskInfo group_task : group_tasks) {
18         //查询该分组下的任务
19         queryWrapper = new QueryWrapper<>();
20         queryWrapper.lambda()
21             .eq(TaskInfo::getTaskType,group_task.getTaskType())
22             .eq(TaskInfo::getPriority,group_task.getPriority());
23         List<TaskInfo> taskInfos = taskInfoMapper.selectList(queryWrapper);
24         //调用addTaskToCache(Task)方法添加到缓存
25         for (TaskInfo taskInfo : taskInfos) {
26             Task task = new Task();
27             BeanUtils.copyProperties(taskInfo,task);
28             task.setExecuteTime(taskInfo.getExecuteTime().getTime());
29             addTaskToCache(task);
30         }
31     }
32 }
33
```

```

34 private void clearCache() {
35     //缓存中要清除的有消费者队列和未来数据集合---我们只需找到所有跟延迟任务相关的key，然后删除这些key即可
36     Set<String> futureKeys = cacheService.scan(DelayTaskRedisKeyConstants.FUTURE + "*");
37     Set<String> topicKeys = cacheService.scan(DelayTaskRedisKeyConstants.TOPIC + "*");
38     //删除这些key即可
39     cacheService.delete(futureKeys);
40     cacheService.delete(topicKeys);
41 }

```

测试：

清空所有数据，执行测试类中添加3个任务的测试方法，执行完成后，清空缓存，然后启动延迟任务系统启动类，查看缓存是否恢复

4.4.2.线程池知识回顾

4.4.2.1.为什么引入线程池框架

1: new Thread()的缺点是耗费性能，调用new Thread()创建的线程缺乏管理，被称为野线程，而且可以无限创建，之间相互竞争，会导致过多占用系统资源导致系统瘫痪，不利于扩展。

2: 采用线程池的优点是：重用存在的线程，减少对象创建、消亡的开销，性能佳，可有效控制最大并发线程数，提高系统资源的使用率，同时避免过多资源竞争，避免堵塞，提供定时执行、定期执行、单线程、并发数控制等功能

4.4.2.2.Executor和ExecutorService

1: Executor接口定义为：

```

1 public interface Executor {
2     void execute(Runnable command);
3 }

```

执行已提交的Runnable 任务对象。此接口提供一种将任务提交与每个任务将如何运行的机制（包括线程使用的细节、调度等）分离开来的方法

2: ExecutorService接口定义为：

ExecutorService其实是一个更通用的线程池接口，它可以接收任务，然后根据配置来分配线程，并控制其调度，可以对线程统一管理。

```

1 public interface ExecutorService extends Executor{
2
3     //关闭线程池，不再接收新的任务，但是会将当前线程池中的所有任务执行完后再关闭
4     void shutdown();
5     //立即关闭，不再接收新的任务，抛弃线程池中还未执行的任务并中断所有正在执行的任务，返回等待执行的任务列表
6     List<Runnable> shutdownNow();
7     //判断线程池是否关闭
8     boolean isShutdown();
9     //线程池是否终止
10    boolean isTerminated();

```

```

11     boolean awaitTermination(long timeout, TimeUnit unit) throws InterruptedException;
12
13     // 提交任务, 任务是一个实现了Callable的类, 该类中返回任务结果, 该方法返回Future, 可以通过Future
    获得任务结果
14     <T> Future<T> submit(Callable<T> task);
15
16     <T> Future<T> submit(Runnable task, T result);
17     //提交任务, 任务是实现了Runnable的类, 该类中的run方法是void因此无法返回结果, 该方法仍然会返回
    Future, 但是通过Future得不到结果
18     Future<?> submit(Runnable task);
19
20     // 提交任务集合, 返回Future集合
21     <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks) ;
22     // 提交任务集合, 返回其中一个任务的结果
23     <T> T invokeAny(Collection<? extends Callable<T>> tasks)
24 }

```

核心方法说明:

- execute(Runnable)

execute(Runnable) 方法接受一个java.lang.Runnable对象的实例, 并异步执行

```

1 executorService.execute(new Runnable() {
2     public void run() {
3         System.out.println("Asynchronous task");
4     }
5 });

```

这种方式不能获得Runnable执行的结果, 如果有这种需要, 需要要使用Callable

我们都知道创建线程的2种方式, 一种是直接继承Thread, 另外一种就是实现Runnable接口; 但是这2种方式都有一个缺陷就是: 在执行完任务之后无法获取执行结果。如果需要获取执行结果, 就必须通过共享变量或者使用线程通信的方式来达到效果, 这样使用起来就比较麻烦。而自从Java 1.5开始, 就提供了Callable和Future, 通过它们可以在任务执行完毕之后得到任务执行结果。

- submit(Callable)

submit(Callable)方法与submit(Runnable)方法相似, 除了接收的参数有所不同。Callable实例非常类似于Runnable, 不同的是Callable.call()方法可以返回一个结果, Runnable.run()方法不能返回一个结果

```

1 //Callable的声明
2 public interface Callable<V> {
3     V call() throws Exception;
4 }
5
6 Future future = executorService.submit(new Callable<String>(){
7     public String call() throws Exception {
8         System.out.println("Asynchronous Callable");
9         return "Callable Result";
10    }
11 });
12 System.out.println("future.get() = " + future.get());

```

- submit(Runnable)

submit(Runnable) 方法也可以接收一个Runnable接口的具体实现，并返回一个Future对象。Future对象可以用来检测Runnable是否执行完成。

```

1 Future future = executorService.submit(new Runnable() {
2     public void run() {
3         System.out.println("Asynchronous task");
4     }
5 });
6 future.get(); //如果任务正常完成则该方法返回null

```

此方法仍然不能获得Runnable的执行结果！

- shutdown()/shutdownNow()

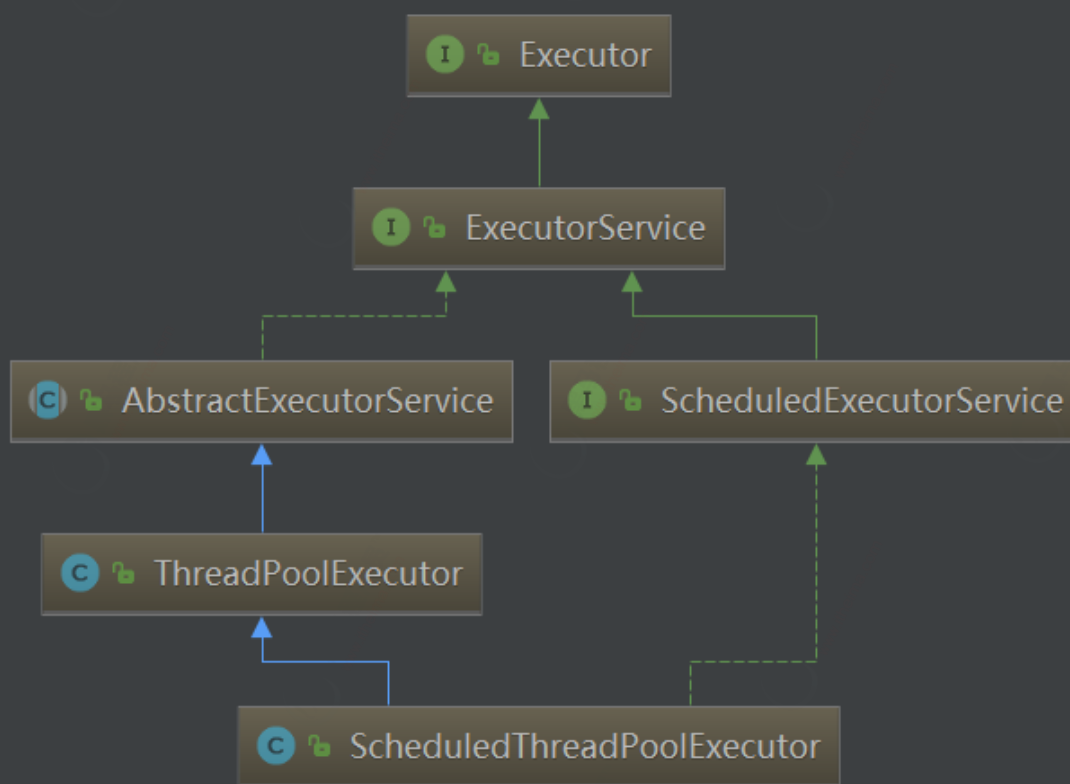
使用完ExecutorService后，你应该关闭它，使得线程不能持续运行。例如，你的应用程序从main()方法开始并且你的主线程退出应用程序，这时如果存在激活状态的ExecutorService，你的应用程序将仍然会保持运行。ExecutorService中激活的线程会阻止JVM关闭。

为了终止ExecutorService中的线程，你需要调用shutdown()方法。ExecutorService不会立即关闭，但是它也不会接受新的任务，直到它里面的所有线程都执行完毕，ExecutorService才会关闭。所有提交到ExecutorService中的任务会在调用shutdown()方法之前被执行。

如果你想立即关闭ExecutorService,你可以调用shutdownNow()方法。这将会尝试立即停止所有正在执行的任务，并且忽略所有提交的但未被处理的任务。对于正在执行的任务是不能确定的，也许它们停止了，也行它们执行直到结束

4.4.2.3.ThreadPoolExecutor及参数

ThreadPoolExecutor是ExecutorService接口的实现，jdk中提供的executor框架结构图如图所示：



其中AbstractExecutorService是一个抽象类，它的作用是对ExecutorService接口中执行方法有默认实现，ThreadPoolExecutor继承AbstractExecutorService。

首先来看如何创建一个ThreadPoolExecutor，下面是ThreadPoolExecutor常用的一个构造方法：

```
1 public class ThreadPoolExecutor extends AbstractExecutorService {
2     //常用的构造函数
3     public ThreadPoolExecutor(int corePoolSize,
4                             int maximumPoolSize,
5                             long keepAliveTime,
6                             TimeUnit unit,
7                             BlockingQueue<Runnable> workQueue) {
8         this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
9             Executors.defaultThreadFactory(), defaultHandler);
10    }
11    //核心的构造函数，其他构造函数都是调用该构造函数
12    public ThreadPoolExecutor(int corePoolSize,
13                            int maximumPoolSize,
14                            long keepAliveTime,
15                            TimeUnit unit,
16                            BlockingQueue<Runnable> workQueue,
17                            ThreadFactory threadFactory,
18                            RejectedExecutionHandler handler) {
19        if (corePoolSize < 0 ||
20            maximumPoolSize <= 0 ||
```

```

21         maximumPoolSize < corePoolSize ||
22         keepAliveTime < 0)
23         throw new IllegalArgumentException();
24
25         if (workQueue == null || threadFactory == null || handler == null)
26             throw new NullPointerException();
27         this.corePoolSize = corePoolSize;
28         this.maximumPoolSize = maximumPoolSize;
29         this.workQueue = workQueue;
30         this.keepAliveTime = unit.toNanos(keepAliveTime);
31         this.threadFactory = threadFactory;
32         this.handler = handler;
33     }
34 }

```

线程池核心参数介绍：

参数名	作用
corePoolSize	核心线程池基本大小，核心线程数
maximumPoolSize	线程池最大线程大小
keepAliveTime	线程空闲后的存活时间
TimeUnit unit	线程空闲后的存活时间单位
BlockingQueue workQueue	存放任务的阻塞队列
ThreadFactory threadFactory	创建线程的工厂
RejectedExecutionHandler handler	当阻塞队列和最大线程池都满了之后的饱和策略

- **corePoolSize：核心线程数量**

1：线程池刚创建时，线程数量为0，当每次执行execute添加新的任务时会在线程池创建一个新的线程，直到线程数量达到corePoolSize为止。

2：核心线程会一直存活，即使没有任务需要执行，当线程数小于核心线程数时，即使有线程空闲，线程池也会优先创建新线程处理

3：设置allowCoreThreadTimeout=true（默认false）时，核心线程会超时关闭

- **workQueue：阻塞队列**

1：当线程池正在运行的线程数量已经达到corePoolSize，那么再通过execute添加新的任务则会被加workQueue队列中，在队列中排队等待执行，而不会立即执行。

一般来说，这里的阻塞队列有以下几种选择：

ArrayBlockingQueue;

LinkedBlockingQueue;

SynchronousQueue;

- **maximumPoolSize: 最大线程数**

- 1: 当池中的线程数 $\geq$ corePoolSize, 且任务队列已满时。线程池会创建新线程来处理任务
- 2: 当池中的线程数=maximumPoolSize, 且任务队列已满时, 线程池会拒绝处理任务而抛出异常

- **keepAliveTime: 线程空闲时间**

- 1: 当线程空闲时间达到keepAliveTime时, 线程会退出, 直到线程数量=corePoolSize
- 2: 如果allowCoreThreadTimeout=true, 则会直到线程数量=0

- **threadFactory: 线程工厂, 主要用来创建线程**

- **rejectedExecutionHandler: 任务拒绝处理器, 两种情况会拒绝处理任务**

- 1: 当线程数已经达到maxPoolSize, 且队列已满, 会拒绝新任务
- 2: 当线程池被调用shutdown()后, 会等待线程池里的任务执行完毕, 再shutdown。如果在调用shutdown()和线程池真正shutdown之间提交任务, 会拒绝新任务

3: 当拒绝处理任务时线程池会调用rejectedExecutionHandler来处理这个任务。如果没有设置默认是AbortPolicy, 另外在ThreadPoolExecutor类有几个内部实现类来处理这类情况

ThreadPoolExecutor.AbortPolicy: 丢弃任务并抛出RejectedExecutionException异常。

ThreadPoolExecutor.CallerRunsPolicy: 由调用线程处理该任务

ThreadPoolExecutor.DiscardPolicy: 也是丢弃任务, 但是不抛出异常

ThreadPoolExecutor.DiscardOldestPolicy: 丢弃队列最前面的任务, 然后重新尝试执行任务 (重复此过程)

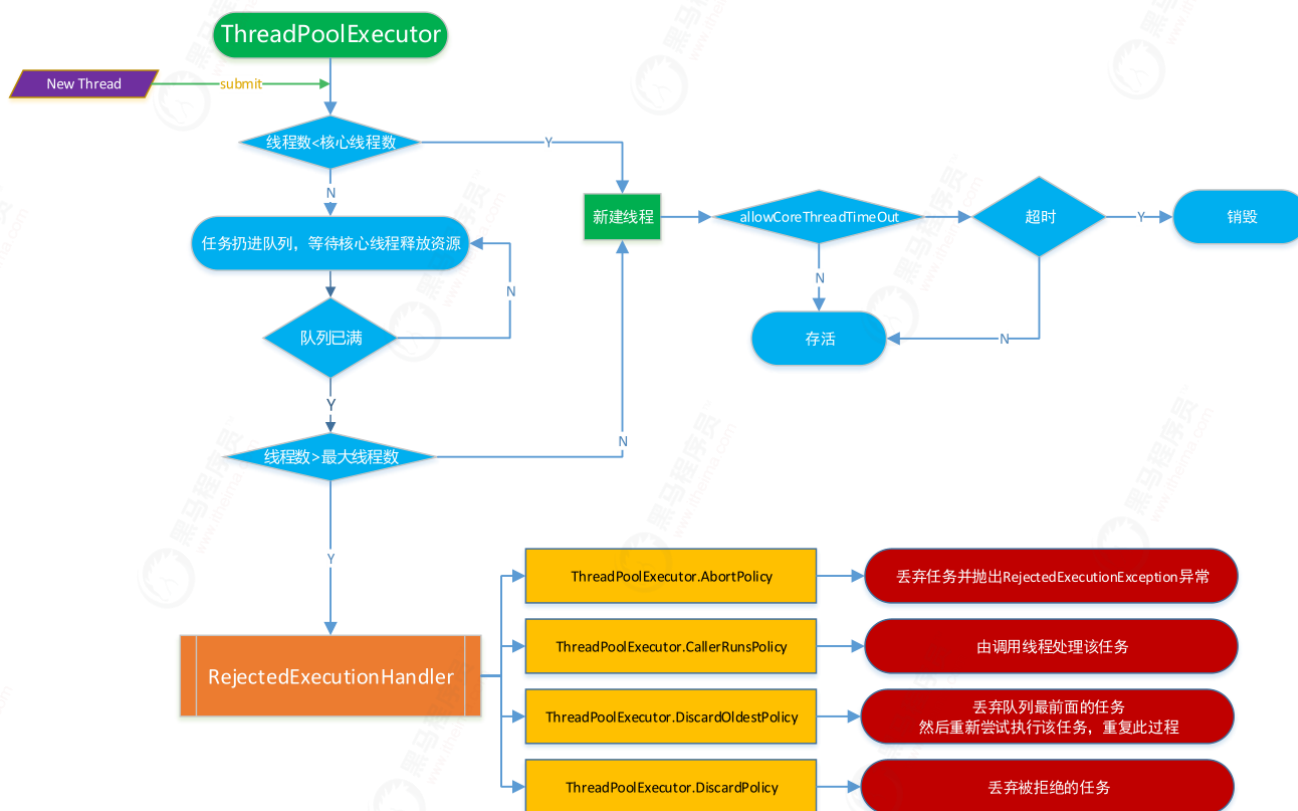
4.4.2.4.execute/submit 方法的执行流程

提交一个任务到线程池, 线程池具体是如何处理的呢?

ThreadPoolExecutor中对execute(Runnable)方法就是具体的实现, 方法实现如下:

```
1 public void execute(Runnable command) {
2     if (command == null) throw new NullPointerException();
3     int c = ctl.get();
4     if (workerCountOf(c) < corePoolSize) {
5         if (addWorker(command, true))
6             return;
7         c = ctl.get();
8     }
9     if (isRunning(c) && workQueue.offer(command)) {
10        int recheck = ctl.get();
11        if (! isRunning(recheck) && remove(command))
12            reject(command);
13        else if (workerCountOf(recheck) == 0)
14            addWorker(null, false);
15    }
16    else if (!addWorker(command, false))
17        reject(command);
18 }
```

图解如下:



对于可以返回任务结果的submit(Callable)方法又是如何实现的呢？

submit(Callable)方法是在ThreadPoolExecutor的父类AbstractExecutorService中实现的，

```

1 public <T> Future<T> submit(Callable<T> task) {
2     if (task == null) throw new NullPointerException();
3     RunnableFuture<T> ftask = newTaskFor(task);
4     execute(ftask);
5     return ftask;
6 }
7 protected <T> RunnableFuture<T> newTaskFor(Callable<T> callable) {
8     return new FutureTask<T>(callable);
9 }
  
```

可以发现submit底层还是调用的execute，submit方法需要传入的是一个实现了Callable的任务类，而execute方法需要传入一个实现了Runnable的类，其中newTaskFor方法就是通过callable构造RunnableFuture的子类，

其中他们的关系是：

```

1 public interface Runnable {
2     public abstract void run();
3 }
4
5 public interface Callable<V> {
6     V call() throws Exception;
7 }
8 //
9 public interface RunnableFuture<V> extends Runnable, Future<V> {
  
```

```

10     void run();
11 }
12
13 public class FutureTask<V> implements RunnableFuture<V> {
14     ...
15 }

```

4.4.2.5.Executors线程池工厂

Executors提供了一些创建线程池的工具方法

1: Executors.newSingleThreadExecutor()

```

1 public static ExecutorService newSingleThreadExecutor() {
2     return new FinalizableDelegatedExecutorService
3         (new ThreadPoolExecutor(1, 1,
4                                 0L, TimeUnit.MILLISECONDS,
5                                 new LinkedBlockingQueue<Runnable>()));
6 }

```

corePoolSize和maximumPoolSize都为1，也就是创建了一个固定大小是1的线程池，workQueue是new LinkedBlockingQueue < Runnable >()是一种无界阻塞队列，队列的大小是Integer.MAX_VALUE，可以认为是队列的大小不限制。

由此可以得出通过该方法创建的线程池，每次只能同时运行一个线程，当有多个任务同时提交时，那也要一个一个排队执行

2: Executors.newFixedThreadPool(int nThreads)

```

1 public static ExecutorService newFixedThreadPool(int nThreads) {
2     return new ThreadPoolExecutor(nThreads, nThreads,
3                                   0L, TimeUnit.MILLISECONDS,
4                                   new LinkedBlockingQueue<Runnable>());
5 }

```

创建了一个固定大小的线程池，可以指定同时运行的线程数量为nThreads。

3: Executors.newCachedThreadPool()

```

1 public static ExecutorService newCachedThreadPool() {
2     return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
3                                   60L, TimeUnit.SECONDS,
4                                   new SynchronousQueue<Runnable>());
5 }

```

构造一个缓冲功能的线程池，配置corePoolSize=0, maximumPoolSize=Integer.MAX_VALUE, keepAliveTime=60s,以及一个无容量的阻塞队列 SynchronousQueue，因此任务提交之后，将会创建新的线程执行；线程空闲超过60s将会销毁

4: Executors.newScheduledThreadPool(int corePoolSize)

```

1 public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize) {
2     return new ScheduledThreadPoolExecutor(corePoolSize);
3 }
4
5 public static ScheduledExecutorService newScheduledThreadPool(
6     int corePoolSize, ThreadFactory threadFactory) {
7     return new ScheduledThreadPoolExecutor(corePoolSize, threadFactory);
8 }
9
10 public ScheduledThreadPoolExecutor(int corePoolSize,
11     ThreadFactory threadFactory) {
12     super(corePoolSize, Integer.MAX_VALUE, 0, TimeUnit.NANOSECONDS,
13         new DelayedWorkQueue(), threadFactory);
14 }

```

构造有定时功能的线程池，配置corePoolSize，无界延迟阻塞队列DelayedWorkQueue；maximumPoolSize=Integer.MAX_VALUE，由于DelayedWorkQueue是无界队列，所以这个值是没有意义的

```

1 知识小贴士：注意：阿里巴巴编程规约中对于并发处理有几条强制要求如下：
2
3 线程池不允许使用Executors去创建，而是通过ThreadPoolExecutor的方式，这样的处理方法让写的同学更加明确
  线程池的运行规则，规避资源耗尽的风险；对于Executors返回线程池对象的弊端有：
4
5 FixedThreadPool和SingleThreadPool允许请求的任务等待队列长度为Integer.MAX_VALUE，可能会堆积大量的请
  求任务，从而导致OOM
6
7 CachedThreadPool和ScheduledThreadPool允许创建的最大线程数为Integer.MAX_VALUE，可能会创建大量的线
  程，从而导致OOM

```

从阿里的编码规约来看推荐我们自主的创建ThreadPoolExecutor，那对于线程池最核心的几个参数应该如何选取呢？线程池参数不能胡乱制定否则对服务的性能影响很大，需要根据任务的性质来决定，我们主要参考以下两个方面

1：I/O密集型： CPU使用率较低，程序中会存在大量I/O操作占据时间，导致线程空余时间出来，线程个数为CPU核数的两倍。当其中的线程在IO操作的时候，其他线程可以继续用CPU，提高了CPU的利用率

2：CPU密集型：

CPU使用率较高（也就是一些复杂运算，逻辑处理），所以线程数一般只需要CPU核数的线程就可以了。这一类型的在开发中多出现的一些业务复杂计算和逻辑处理过程中。线程个数为CPU核数。这几个线程可以并行执行，不存在线程切换到开销，提高了CPU的利用率的同时也减少了切换线程导致的性能损耗

4.4.2.6.Spring对线程池的封装

多线程并发处理起来通常比较麻烦，如果使用spring容器事情就好办了多了。spring封装了Java的多线程的实现，我们只需要关注于并发事物的流程以及一些并发负载量等特性。

TaskExecutor接口

```
1 public interface TaskExecutor extends Executor {
2     void execute(Runnable task);
3 }
```

继承自java.util.concurrent.Executor接口，设计之初是为了其他Spring组件提供一个线程池的抽象，根据线程池的配置接受一个任务并执行。

TaskExecutor接口的实现类简要说明

1: SimpleAsyncTaskExecutor类

这个实现不重用任何线程，或者说它每次调用都启动一个新线程，性能消耗比较严重。

2: ConcurrentTaskExecutor 类

Java SE 5.0引入了ThreadPoolExecutor、ScheduledThreadPoolExecutor。Spring 2.x借助ConcurrentTaskExecutor和ThreadPoolTaskExecutor能够通过IoC配置形式自定义它们暴露的各个属性。很少需要使用ConcurrentTaskExecutor,有另一个备选, ThreadPoolTaskExecutor类

3: ThreadPoolTaskExecutor 类

ThreadPoolTaskExecutor内部对ThreadPoolExecutor进行了包装，同时提供能够通过IOC的形式来配置线程池的各个参数，比较常用

4: ThreadPoolTaskScheduler类

ThreadPoolTaskScheduler内部对ScheduledThreadPoolExecutor进行了包装，除了能执行异步任务外支持定时/延迟任务的执行，属于一种高级特性。

在SpringBoot中，对于第三和第四个实现默认情况下帮我们进行自动配置到容器中，我们需要的时候直接注入使用即可，

测试SpringBoot中自动配置的线程池，在TaskServiceTest测试类中添加如下代码

```
1 @Autowired
2 private ThreadPoolTaskExecutor threadPoolTaskExecutor;
3
4 @Test
5 public void test(){
6     //向线程池中提交100个任务
7     for(int i=0;i<100;i++){
8         threadPoolTaskExecutor.execute(new Runnable() {
9             @Override
10             public void run() {
11                 System.out.println("ThreadPoolTaskExecutor test
12                 "+Thread.currentThread().getName());
13             }
14         });
15     }
16     //ThreadPoolTaskExecutor支持对线程池参数的ioc配置
17     System.out.println("核心线程数:"+threadPoolTaskExecutor.getCorePoolSize());
18     System.out.println("最大线程数:"+threadPoolTaskExecutor.getMaxPoolSize());
19     System.out.println("线程等待超时时
```

```

19     间:"+threadPoolTaskExecutor.getKeepAliveSeconds());
20     System.out.println("当前活跃的线程数:"+threadPoolTaskExecutor.getActiveCount());
21     System.out.println("线程池内线程的名称前缀:"+threadPoolTaskExecutor.getThreadNamePrefix());
22 }

```

输出:

```

1  核心线程数:8
2  最大线程数:2147483647
3  线程等待超时时间:60
4  当前活跃的线程数:8
5  线程池内线程的名称前缀:task-

```

如果我们不想要SpringBoot帮我们默认配置的线程池参数,我们可以自行配置,ThreadPoolTaskExecutor支持对线程池核心参数的重新配置,借助注解:@Configuration和@Bean

```

1  @Bean(name = "delayTaskPool")
2  public ThreadPoolTaskExecutor mythreadpool(){
3      ThreadPoolTaskExecutor threadPool = new ThreadPoolTaskExecutor();
4      //设置核心线程数
5      threadPool.setCorePoolSize(8);
6      //设置最大线程数
7      threadPool.setMaxPoolSize(10000);
8      //设置线程超时等待时间
9      threadPool.setKeepAliveSeconds(60);
10     //设置任务等待队列的大小
11     threadPool.setQueueCapacity(10000);
12     //设置线程池内线程的名称前缀---阿里编码规约推荐----出错了方便调试
13     threadPool.setThreadNamePrefix("delay-task-");
14     //设置任务拒绝策略
15     threadPool.setRejectedExecutionHandler(new ThreadPoolExecutor.AbortPolicy());
16     //直接初始化
17     threadPool.initialize();
18     return threadPool;
19 }

```

4.4.3.数据恢复多线程改造

数据恢复的相关操作代码截图如下:

```

    clearCache();

    //查询任务表中的所有分组
    QueryWrapper<TaskInfo> queryWrapper = new QueryWrapper<>();
    queryWrapper.lambda()
        .select(TaskInfo::getTaskType, TaskInfo::getPriority)
        .groupBy(TaskInfo::getTaskType, TaskInfo::getPriority);
    List<TaskInfo> group_tasks = taskInfoMapper.selectList(queryWrapper);
    //获取每个分组下的任务数据将其添加到缓存
    if(Objects.isNull(group_tasks)){
        return;
    }
    for (TaskInfo group_task : group_tasks) {
        //查询该分组下的任务
        queryWrapper = new QueryWrapper<>();
        queryWrapper.lambda()
            .eq(TaskInfo::getTaskType, group_task.getTaskType())
            .eq(TaskInfo::getPriority, group_task.getPriority());
        List<TaskInfo> taskInfos = taskInfoMapper.selectList(queryWrapper);
        //调用addTaskToCache(Task)方法添加到缓存
        for (TaskInfo taskInfo : taskInfos) {
            Task task = new Task();
            BeanUtils.copyProperties(taskInfo, task);
            task.setExecuteTime(taskInfo.getExecuteTime().getTime());
            addTaskToCache(task);
        }
    }
}

```

哪个地方能够优化?

分组恢复的时候如果每个分组下数据量特别大，会造成整个数据恢复过程变的很长，由此会导致任务消费延后

测试：

将如下代码拷贝到测试类中执行，查看当前这种数据恢复模式下，数据恢复的耗时

```

1  @Autowired
2  private TaskServiceImpl taskServiceImpl;
3  @Test
4  public void testRecovery() throws InterruptedException {
5      TimeUnit.SECONDS.sleep(5);
6      //造10000数据
7      addData();
8      //拷贝数据恢复代码
9      long start = System.currentTimeMillis();
10     taskServiceImpl.syncData();
11     System.out.println("数据恢复耗时:"+(System.currentTimeMillis()-start));
12 }
13 @Autowired
14 private TaskInfoMapper taskInfoMapper;
15 private void addData(){
16     for (int i=0;i<10000;i++){
17         TaskInfo taskInfo = new TaskInfo();

```

```

18     taskInfo.setTaskType(i%10);//10个分组
19     taskInfo.setPriority(1);
20     taskInfo.setParameters("{\"orderNo':'fadfa'\"}.getBytes());
21     taskInfo.setExecuteTime(new Date(System.currentTimeMillis()+(i*100)));
22     taskInfoMapper.insert(taskInfo);
23 }
24 }

```

查看耗时: 13400

如何优化?

分组恢复数据时可以并行恢复, 使用多线程技术, 一个组一个线程去恢复

代码实现

注入线程池, 将分组恢复部分提交到线程池中执行

```

1  @Autowired
2  private ThreadPoolTaskExecutor taskExecutor;
3  @PostConstruct
4  public void syncData(){
5      //先清除缓存中的原有数据
6      clearCache();
7      //查询任务表中的所有分组
8      QueryWrapper<TaskInfo> queryWrapper = new QueryWrapper<>();
9      queryWrapper.lambda()
10         .select(TaskInfo::getTaskType,TaskInfo::getPriority)
11         .groupBy(TaskInfo::getTaskType,TaskInfo::getPriority);
12      List<TaskInfo> group_tasks = taskInfoMapper.selectList(queryWrapper);
13      //获取每个分组下的任务数据将其添加到缓存
14      if(Objects.isNull(group_tasks)){
15          return;
16      }
17      for (TaskInfo group_task : group_tasks) {
18          //将每个分组都提交到线程池中执行
19          taskExecutor.execute(new Runnable() {
20              @Override
21              public void run() {
22                  //查询该分组下的任务
23                  QueryWrapper<TaskInfo> queryWrapper = new QueryWrapper<>();
24                  queryWrapper.lambda()
25                     .eq(TaskInfo::getTaskType,group_task.getTaskType())
26                     .eq(TaskInfo::getPriority,group_task.getPriority());
27                  List<TaskInfo> taskInfos = taskInfoMapper.selectList(queryWrapper);
28                  //调用addTaskToCache(Task)方法添加到缓存
29                  for (TaskInfo taskInfo : taskInfos) {
30                      Task task = new Task();
31                      BeanUtils.copyProperties(taskInfo,task);
32                      task.setExecuteTime(taskInfo.getExecuteTime().getTime());
33                      addTaskToCache(task);
34                  }

```

```

35     }
36     });
37 }
38 }

```

知识小贴士：如何去统计多线程执行时间呢？

需求：目前已经将数据恢复分组并且并行恢复，每个组开辟独立的线程去做，那如何统计数据恢复消耗的时间？

解决方案：使用jdk提供的线程计数器CountDownLatch

操作代码如下：

```

1  @PostConstruct
2  public void syncData(){
3      long start = System.currentTimeMillis();//计时代码
4      //先清除缓存中的原有数据
5      clearCache();
6
7      //查询任务表中的所有分组
8      QueryWrapper<TaskInfo> queryWrapper = new QueryWrapper<>();
9      queryWrapper.lambda()
10         .select(TaskInfo::getTaskType,TaskInfo::getPriority)
11         .groupBy(TaskInfo::getTaskType,TaskInfo::getPriority);
12      List<TaskInfo> group_tasks = taskInfoMapper.selectList(queryWrapper);
13      //获取每个分组下的任务数据将其添加到缓存
14      if(Objects.isNull(group_tasks)){
15          return;
16      }
17      CountDownLatch countDownLatch = new CountDownLatch(group_tasks.size());//计时代码
18      for (TaskInfo group_task : group_tasks) {
19          //将每个分组都提交到线程池中执行
20          taskExecutor.execute(new Runnable() {
21              @Override
22              public void run() {
23                  //查询该分组下的任务
24                  QueryWrapper<TaskInfo> queryWrapper = new QueryWrapper<>();
25                  queryWrapper.lambda()
26                     .eq(TaskInfo::getTaskType,group_task.getTaskType())
27                     .eq(TaskInfo::getPriority,group_task.getPriority());
28                  List<TaskInfo> taskInfos = taskInfoMapper.selectList(queryWrapper);
29                  //调用addTaskToCache(Task)方法添加到缓存
30                  for (TaskInfo taskInfo : taskInfos) {
31                      Task task = new Task();
32                      BeanUtils.copyProperties(taskInfo,task);
33                      task.setExecuteTime(taskInfo.getExecuteTime().getTime());
34                      addTaskToCache(task);
35                  }
36                  countDownLatch.countDown();//计时代码
37              }
38          });
39      }

```

```

40 //计时代码
41 try {
42     countdownLatch.await(5, TimeUnit.MINUTES);
43     log.info("多线程分组数据恢复耗时:{}", System.currentTimeMillis() - start);
44 } catch (InterruptedException e) {
45     e.printStackTrace();
46 }
47 }

```

查看耗时: 5031

4.4.4.定时刷新多线程改造

为什么要改造?

定时刷新的方法: refresh, 目前是每隔一秒钟定时刷新一次, 如果中间业务处理超过1秒, 刷新有延迟现象, 也会造成任务消费延后

制造刷新延迟场景: 在refresh方法中让当前线程停2秒钟, 运行DelayTaskApplication启动类测试, 查看定时刷新情况, 可将如下代码添加至refresh方法顶部进行测试

```

1 log.info("{}进行了定时刷新",
  LocalDateTime.now().format(DateTimeFormatter.ISO_DATE_TIME));
2 try {
3     TimeUnit.SECONDS.sleep(2);
4 } catch (InterruptedException e) {
5     e.printStackTrace();
6 }

```

启动测试结果发现: 每隔3秒钟运行一次, 极大的降低了刷新的频率, 导致很多任务消费的时延

如何改造?

将定时刷新的所有操作提交到线程池中执行, 这样每次刷新都是在独立的线程中执行, 单次刷新时间长并不会影响整体的刷新频率

```

1 @Scheduled(cron = "*/1 * * * * ?")
2 public void refresh(){
3     taskExecutor.execute(new Runnable() {
4         @Override
5         public void run() {
6             log.info("{}进行了定时刷新",
7                 LocalDateTime.now().format(DateTimeFormatter.ISO_DATE_TIME));
8             try {
9                 TimeUnit.SECONDS.sleep(2);
10            } catch (InterruptedException e) {
11                e.printStackTrace();
12            }
13            //找到未来数据集的所有key
14            Set<String> future_keys =

```

```

cacheService.scan(DelayTaskRedisKeyConstants.FUTURE+"*");//scan future_*
15     if(future_keys==null || future_keys.size() ==0){
16         return;
17     }
18     for (String future_key : future_keys) {
19         //根据每一个key从该分组下 获取当前需要执行的任务数据Set集合
20         Set<String> values = cacheService.zRangeByScore(future_key, 0,
System.currentTimeMillis());
21         if(values == null || values.size()==0){
22             continue;
23         }
24
25         //将这组数据添加到消费者队列的对应分组上, 并从zset中移除
26         String topicKey =
DelayTaskRedisKeyConstants.TOPIC+future_key.split(DelayTaskRedisKeyConstants.FUTURE)
[1]; //future_2001_1
27         /*for (String value : values) {
28             cacheService.lLeftPush(topicKey,value);
29             cacheService.zRemove(future_key,value);
30         }*/
31         cacheService.refreshWithPipeline(future_key,topicKey,values); //只需此一句,改
造成完成
32     }
33 }
34 });
35 }

```

再次运行启动类, 查看定时刷新的频率, 然后删除无关的暂停的代码

4.5.业务接口数据预加载

4.5.1.预加载需求及方案设计

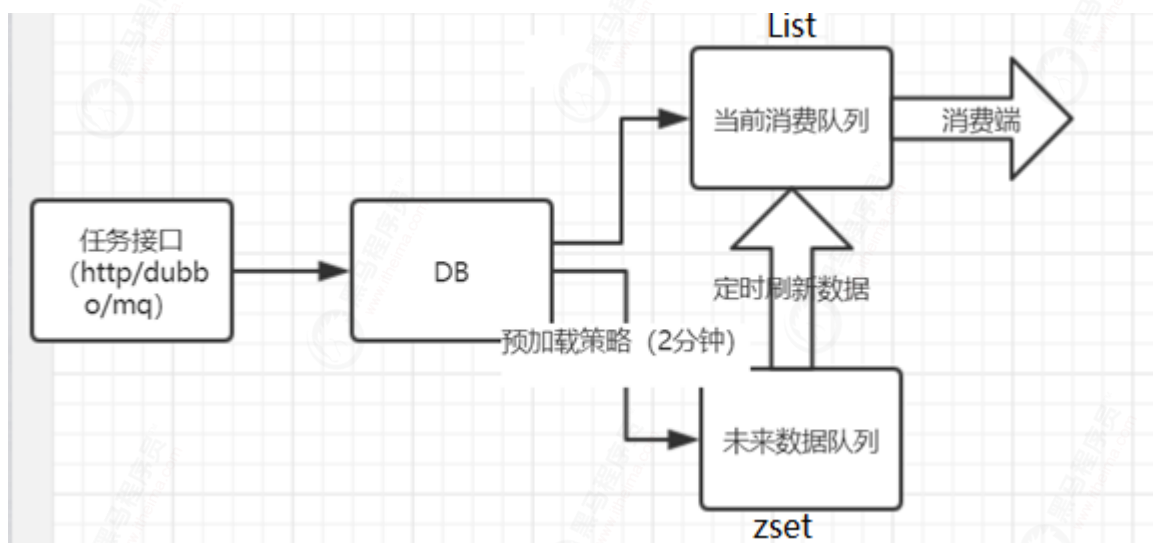
虽然现在已经对系统做了很多的优化工作, 系统性能和吞吐量都提高不少, 本着力争最优的原则思考一下系统还能否进一步优化, 哪些地方还能继续优化呢?

1: 数据库层面的相关操作, 可以使用索引来提升查询性能, 对大概率使用的task_type,priority字段已经创建了索引

2: 添加任务的设计缺陷: 目前添加任务时除了将任务存入数据库外, 我们还需要根据任务的执行时间来判断, 如果是当前要执行的则添加到消费者队列, 如果是未来要执行的则放入zset集合中排序等待, 而对于延迟任务来说, 基本都是未来某个时间点才来执行的, 也就是说大量的任务数据还是在未来数据集合zset中, 虽然我们对zset做了分组, 但是如果某个时间段内产生了超大量的任务数据, 对我们的缓存系统来说, 即使是每个分组内仍然可能会有大量的数据, 此时会造成不小的压力

预加载方案

只加载未来某一段时间内的任务数据进入缓存, 比如未来2分钟, 这样相比整个未来数据而言, 2分钟内的任务数据就少很多, 况且2分钟内的数据还会有分组; 当然我们还需要定时(比如每隔2分钟)的去将数据库中未来2分钟内的数据恢复到缓存中



4.5.2.恢复方法/添加方法改造

1: 配置预加载的时间参数: 时间参数可动态配置, 根据自身系统的任务量而定

找到application.yml配置文件, 添加自定义的配置

```

1 delaytask:
2   preLoad: 1 #预加载时间配置

```

在delay-task-service中创建配置类加载配置属性:

```
com.itheima.delaytask.properties.SystemParamProperties
```

```

1 /**
2  * Created by 传智播客*黑马程序员.
3  */
4 @Data
5 @Configuration
6 @ConfigurationProperties(prefix = "delaytask")
7 public class SystemParamProperties {
8     private int preLoad;
9 }

```

2: 改造数据恢复方法:

定义未来2分钟的时间

分组恢复查询数据时添加时间条件

添加定时恢复的支持, 可使用spring封装好的: ThreadPoolTaskScheduler

```

1 @Autowired
2 private SystemParamProperties paramProperties;
3 private long nextScheduleTime; //下一次未来2分钟的时间节点
4 @Autowired
5 private ThreadPoolTaskScheduler threadPoolTaskScheduler;
6

```

```

7  @PostConstruct
8  public void syncData(){
9      //首次运行即执行一次，然后按照设定的时间周期按照固定频率执行
10     threadPoolTaskScheduler.scheduleAtFixedRate(new Runnable() {
11         @Override
12         public void run() {
13             log.info("{}开始进行定时数据恢
14             复", LocalDateTime.now().format(DateTimeFormatter.ISO_DATE_TIME));
15             reloadData();
16         }
17     }, TimeUnit.MINUTES.toMillis(paramProperties.getPreLoad())); //需要传递一个毫秒值
18 private void reloadData(){
19     long start = System.currentTimeMillis();
20     //先清除缓存中的原有数据
21     clearCache();
22
23     //查询任务表中的所有分组
24     QueryWrapper<TaskInfo> queryWrapper = new QueryWrapper<>();
25     queryWrapper.lambda()
26         .select(TaskInfo::getTaskType, TaskInfo::getPriority)
27         .groupBy(TaskInfo::getTaskType, TaskInfo::getPriority);
28     List<TaskInfo> group_tasks = taskInfoMapper.selectList(queryWrapper);
29     //获取每个分组下的任务数据将其添加到缓存
30     if(Objects.isNull(group_tasks)){
31         return;
32     }
33     CountDownLatch countDownLatch = new CountDownLatch(group_tasks.size());
34     //定义未来2分钟的时间节点变量
35     Calendar calendar = Calendar.getInstance();
36     calendar.add(Calendar.MINUTE, paramProperties.getPreLoad());
37     nextScheduleTime = calendar.getTimeInMillis();
38
39     for (TaskInfo group_task : group_tasks) {
40         //将每个分组都提交到线程池中执行
41         taskExecutor.execute(new Runnable() {
42             @Override
43             public void run() {
44                 //查询该分组下的任务
45                 QueryWrapper<TaskInfo> queryWrapper = new QueryWrapper<>();
46                 queryWrapper.lambda()
47                     .eq(TaskInfo::getTaskType, group_task.getTaskType())
48                     .eq(TaskInfo::getPriority, group_task.getPriority())
49                     .le(TaskInfo::getExecuteTime, calendar.getTime()); //添加时间的查询条件
50                 List<TaskInfo> taskInfos = taskInfoMapper.selectList(queryWrapper);
51                 //调用addTaskToCache(Task)方法添加到缓存
52                 for (TaskInfo taskInfo : taskInfos) {
53                     Task task = new Task();
54                     BeanUtils.copyProperties(taskInfo, task);
55                     task.setExecuteTime(taskInfo.getExecuteTime().getTime());
56                     addTaskToCache(task);
57                 }
58
59                 countDownLatch.countDown();

```

```

59     }
60     });
61 }
62 try {
63     countDownLatch.await(5,TimeUnit.MINUTES);
64     log.info("多线程分组数据恢复耗时:{}",System.currentTimeMillis()-start);
65 } catch (InterruptedException e) {
66     e.printStackTrace();
67 }
68 }

```

3: 改造添加任务接口方法:

添加任务改造, 只需要改造addTaskToCache方法, 如下

```

1 private void addTaskToCache(Task task) {
2     //要根据任务类型和优先级进行分组, 每组的key不一样
3     String key = task.getTaskType()+"_"+task.getPriority();
4     //根据时间判断
5     if(task.getExecuteTime() <= System.currentTimeMillis()){
6         //任务进入消费者队列List---数据从左侧入队列, 消费时从右侧出队列--保证任务消费的一个顺序性
7         //同时我们让所有分组的key都有一个相同的前缀, 方便进行管理
8
9         cacheService.lLeftPush(DelayTaskRedisKeyConstants.TOPIC+key,JSON.toJSONString(task));
10    }else if(task.getExecuteTime() <= nextScheduleTime){
11        //任务进入zset排序等待
12
13        cacheService.zAdd(DelayTaskRedisKeyConstants.FUTURE+key,JSON.toJSONString(task),task.getExecuteTime());
14    }
15
16    //先暂时将数据存储在zset, 后续会对此优化
17    //cacheService.zAdd(DelayTaskRedisKeyConstants.DBCACHE,
18    JSON.toJSONString(task),task.getExecuteTime());
19 }

```

4: 测试

直接使用以下测试方法测试即可

```

1 @Test
2 public void testPreLoad(){
3     // 添加任务数据
4     Date now = new Date();
5     for(int i=0;i<3;i++){
6         Task task = new Task();
7         task.setTaskType(250);
8         task.setPriority(250);
9         task.setExecuteTime(now.getTime() + 50000 * i);
10        task.setParameters("testpooltask".getBytes());
11        taskService.addTask(task);
12    }

```

```

13 // 构造了三个任务：分别在 0 50 100 秒执行
14 // 第一个任务加到消费者队列 立即被消费了
15 // 第二个任务被加到了ZSet集合中,50秒后被消费
16 // 第三个任务还在数据库中
17 // 第60秒时 第三个任务从数据库被加载到ZSet中,40秒后被消费
18 // 消费拉取任务
19 while (true){
20     Task task = taskService.poll(250,250);
21     if(task !=null){
22
23         System.out.println(LocalDate.now().format(DateTimeFormatter.ISO_DATE_TIME)+"成功消费了任
24         务:"+task.getTaskId());
25     }
26     //每隔一秒消费一次
27     try {
28         Thread.sleep(1000);
29     } catch (InterruptedException e) {
30         e.printStackTrace();
31     }
32 }

```

4.6.延迟任务服务化和高可用化

虽然程序目前做了各种优化，不过还是单机版，需要改造成微服务（分布式）并要保证高可用；当下两种主流的方案分别是基于dubbo和SpringCloud，对于延迟任务系统的服务化改造，我们采用了基于http协议的SpringCloud体系，这种形式我们开发起来比较顺手，因为说简单了就是传统的SSM开发，目前已经完成了service和mapper的开发，现在只需完成controller的编写即可

4.6.1.延迟任务web层接口开发

1：创建Controller：com.itheima.delaytask.controller.TaskController；依次编写：添加任务，消费任务，取消任务的方法

添加任务：pushTask

消费任务：pollTask

取消任务：cancelTask

```

1 /**
2  * Created by 传智播客*黑马程序员.
3  */
4 @Slf4j
5 @RestController
6 @RequestMapping("/task")
7 public class TaskController {
8
9     @Autowired
10     private TaskService taskService;
11
12     @PostMapping("/push")
13     public ResponseMessage pushTask(@RequestBody Task task){

```

```

14     log.info("add task {}",task);
15     //参数校验
16     Assert.notNull(task.getTaskType(),"任务类型不能为空");
17     Assert.notNull(task.getPriority(),"任务优先级不能为空");
18     Assert.notNull(task.getExecuteTime(),"任务执行时间不能为空");
19     Long taskId = taskService.addTask(task);
20     return ResponseMessage.ok(taskId);
21 }
22
23 @GetMapping("/poll/{type}/{priority}")
24 public ResponseMessage pollTask(@PathVariable("type") Integer type,
25 @PathVariable("priority") Integer priority){
26     log.info("poll task {},{}",type,priority);
27     Task task = taskService.poll(type, priority);
28     return ResponseMessage.ok(task);
29 }
30 @PostMapping("/cancel")
31 public ResponseMessage cancelTask(@RequestParam("taskId") Long taskId){
32     log.info("cancel task,taskId={}",taskId);
33     boolean cancelTask = taskService.cancelTask(taskId);
34     return ResponseMessage.ok(cancelTask);
35 }
36 }

```

2: 测试: 使用postman工具进行测试

测试用例:

```

1 {"executeTime":1590339440327,"priority":1,"taskType":1001}

```

4.6.2.SpringCloud集成Consul

要把延迟任务做成一个微服务, 一个很重要的环节就是服务治理, 延迟任务系统需要在注册中心去注册, 我们选择使用consul, consul不仅可以作为微服务的注册中心还可以作为配置中心使用, 因为它支持K/V结构数据的存储, 并且支持配置数据的动态刷新

1: 添加相关坐标依赖, 父工程中已做好了依赖管理, 在delay-task-service中添加如下

```

1 <!--健康检查依赖包-->
2 <dependency>
3     <groupId>org.springframework.boot</groupId>
4     <artifactId>spring-boot-starter-actuator</artifactId>
5 </dependency>
6 <!--SpringCloudConsul支持包-->
7 <dependency>
8     <groupId>org.springframework.cloud</groupId>
9     <artifactId>spring-cloud-starter-consul-discovery</artifactId>
10 </dependency>
11 <dependency>
12     <groupId>org.springframework.cloud</groupId>
13     <artifactId>spring-cloud-starter-consul-config</artifactId>

```

```
14 </dependency>
```

2: 创建: bootstrap.yml配置文件, 添加如下配置

```
1 spring:
2   application:
3     name: delay-task-service #服务名
4   cloud:
5     consul:
6       host: 192.168.200.129 #默认localhost
7       port: 8500 #默认8500
8       discovery:
9         enabled: true #开启服务发现, 默认为true
10        service-name: ${spring.application.name} #注册服务名称
11        prefer-ip-address: true
12        register: true #进行服务注册, 默认为true
13        deregister: true
14        register-health-check: true #开启健康检查, 默认true
15        health-check-interval: 10s
16        health-check-critical-timeout: 2m #2分钟之后健康检查未通过取消注册
17        heartbeat:
18          enabled: true
19      config:
20        enabled: true #开启配置中心, 默认true
21        format: yaml #配置格式yaml
22        prefix: config #配置数据的基本目录, 默认config
23        default-context: delay-task #应用读取配置的目录, 可据此区分不同应用的配置
24        data-key: data #consul中K/V的key, 默认data
```

3: 在consul中创建对应的K/V, `config/delay-task/data`, 将application.yml中一些配置保存到consul, 譬如, 数据库配置, redis配置, 自己的业务配置; 对于有一些配置比如端口的配置可留在application.yml中, 操作完如下图所示:

data

Value

```
1 spring:
2   datasource:
3     url: jdbc:mysql://192.168.200.129:3306/delay_task?serverTimezone=Asia/Shanghai
4     driver-class-name: com.mysql.cj.jdbc.Driver
5     username: root
6     password: root123
7   redis:
8     host: 192.168.200.129
9     password: root123
10    port: 6379
11
12 delaytask:
13   preLoad: 1 #预加载时间配置
```

Save

Cancel changes

4: 运行启动类: DelayTaskApplication, 查看129主机上consul控制台界面

知识小贴士:

一个服务还可以运行在多个节点上, 启动完服务后修改端口再次启动

4.6.3.延迟任务feign集成

现在延迟任务系统已经注册成为了一个微服务, 并且也已经启动了多个节点, 成功改造成了分布式, 对于服务接口的调用仍然是基于web层接口的http调用, 现在我们需要用feign集成, 将服务的http调用转换为feign接口方法的调用, 使得对服务的调用更加的简单。此外Spring Cloud Feign还整合了Spring Cloud Ribbon (负载均衡) 和Spring Cloud Hystrix (熔断器)。

只需要在delay-task-feign模块中编写好延迟任务系统对应的feign接口即可, 至于为什么要单独抽取一个delay-task-feign模块的原因是为了让其他模块使用起来更加的方便, 只需要依赖delay-task-feign模块即可使用。

0: 添加相关坐标

1: 创建feign接口: com.itheima.delaytask.feign.TaskService, 并添加三个接口方法

```
1 /**
2  * Created by 传智播客*黑马程序员.
```

```

3  */
4  @FeignClient(name = "delay-task-service")
5  public interface TaskService {
6
7      //添加任务
8      @PostMapping("/task/push")
9      public ResponseMessage push(@RequestBody Task task);
10
11     //消费任务
12     @GetMapping("/task/poll/{type}/{priority}")
13     public ResponseMessage poll(@PathVariable("type") Integer type,
14     @PathVariable("priority") Integer priority);
15
16     //取消任务
17     @PostMapping("/task/cancel")
18     public ResponseMessage cancel(@RequestParam("taskId") Long taskId);
19 }

```

2: 创建FallbackFactory: com.itheima.delaytask.feign.fallbackfactory.TaskServiceFallbackFactory

```

1  /**
2   * Created by 传智播客*黑马程序员.
3   */
4  @Component
5  @Slf4j
6  public class TaskServiceFallbackFactory implements FallbackFactory<TaskService> {
7      @Override
8      public TaskService create(Throwable throwable) {
9          return new TaskService() {
10              @Override
11              public ResponseMessage push(Task task) {
12                  log.error("push task error by feign,task={},errorMsg=
13                  {}",task,throwable.getMessage());
14                  return ResponseMessage.error(null);
15              }
16
17              @Override
18              public ResponseMessage poll(Integer type, Integer priority) {
19                  log.error("poll task error by feign,type={},priority={},errorMsg=
20                  {}",type,priority,throwable.getMessage());
21                  return ResponseMessage.error(null);
22              }
23
24              @Override
25              public ResponseMessage cancel(Long taskId) {
26                  log.error("cancel task error by feign,taskId={},errorMsg=
27                  {}",taskId,throwable.getMessage());
28                  return ResponseMessage.error(false);
29              }
30          };
31      }
32  }

```

3: 在接口上指定fallbackFactory

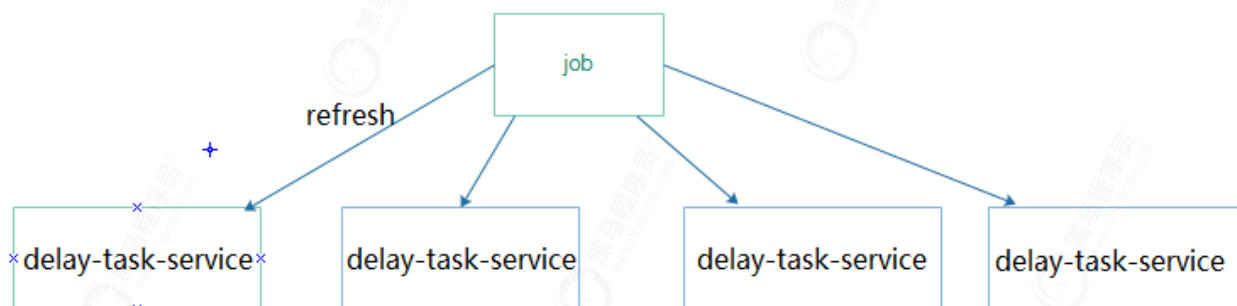
```
1 @FeignClient(name = "delay-task-service", fallbackFactory = TaskServiceFallbackFactory.class)
```

4.6.4.延迟任务job节点开发

现在延迟任务系统已经改造成分布式系统了，并且一个服务可以启多个节点，在延迟任务系统中有个定时的业务逻辑是将未来数据集中的数据刷新到消费者队列，目前的状态是每个节点都会执行，势必会产生冲突，需要有统一的分布式任务调度job系统。

```
1 @Scheduled(cron = "0/1 * * * * ?") //每个延迟任务节点都在独立的刷新，会造成冲突
2 public void refresh() {}
```

架构设计



- 延迟任务系统暴露定时刷新的接口refresh，使得job可以通过feign调用
- 在job系统中来定时的刷新，刷新逻辑就是通过feign调用delay-task-service延迟任务服务的刷新方法refresh，同时还能利用feign的负载均衡减轻每个节点的压力

代码实现

1: 延迟任务系统业务层抽取refresh接口，找到delay-task-service模块中的业务层接口，添加如下接口方法

```
1 /**
2  * 定时刷新
3  */
4 public void refresh();
```

接口方法的实现之前已经实现完成

2: 找到web层controller，添加如下定时刷新的接口方法

```

1 @GetMapping("/refresh")
2 public ResponseMessage refresh(){
3     log.info("refresh----");
4     taskService.refresh();
5     return ResponseMessage.ok(null);
6 }

```

3: 找到delay-task-feign模块中的feign接口, 添加定时刷新的feign接口方法

```

1 //刷新
2 @GetMapping("/task/refresh")
3 public ResponseMessage refresh();

```

同时需要在fallbackfactory中添加方法实现

```

1 @Override
2 public ResponseMessage refresh() {
3     log.error("refresh by feign ,errorMsg={}",throwable.getMessage());
4     return ResponseMessage.error(null);
5 }

```

4: 在delay-task-job模块中需要通过feign的refresh接口调用延迟任务节点完成刷新操作, 简单点说就是需要将之前运行在延迟任务节点上的定时任务拿掉, 放到job系统中去每隔1秒钟定时执行, 执行的时候通过调用延迟任务的feign接口来完成刷新; 同时为了保证job节点定时刷新的一个频率我们可以采用线程池进行优化。

在delay-task-job模块中创建: com.itheima.job.DelayTaskJob, 并编写一个方法refresh方法

```

1 /**
2  * Created by 传智播客*黑马程序员.
3  */
4 @Component
5 @Slf4j
6 public class DelayTaskJob {
7
8
9     @Autowired
10     private ThreadPoolTaskExecutor threadPoolTaskExecutor;
11
12     @Autowired
13     private TaskService taskService;
14
15     @Scheduled(cron = "*/1 * * * * ?")
16     public void refresh(){
17         threadPoolTaskExecutor.execute(new Runnable() {
18             @Override
19             public void run() {
20                 log.info("job节点{}开始进行定时刷新",
21                     LocalDateTime.now().format(DateTimeFormatter.ISO_DATE_TIME));
22                 taskService.refresh();
23             }
24         });
25     }
26 }

```

```
23     });
24 }
25 }
```

同时需要在启动类上添加相关注解：

```
1 @SpringBootApplication
2 @EnableScheduling
3 @EnableFeignClients({"com.itheima.delaytask.feign"})
4 public class JobApplication {
5     public static void main(String[] args) {
6         SpringApplication.run(JobApplication.class,args);
7     }
8 }
```

5：在delay-task-job模块中相关配置已提供

6：回到delay-task-service模块中，注释掉 `TaskServiceImpl` 类中定时刷新方法上的定时任务注解

7：测试

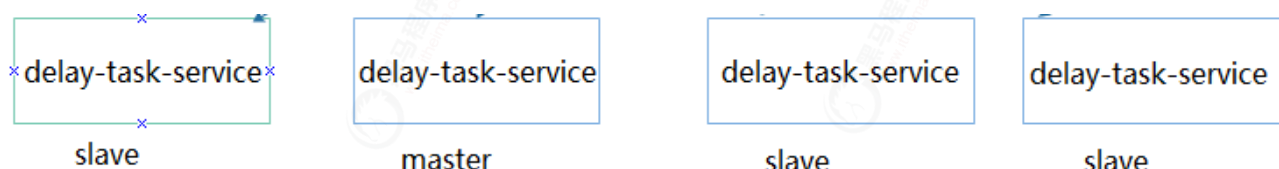
延迟任务分别在81/82两个节点上启动，然后启动job节点，查看定时刷新的情况

预期：job中每秒刷新一次，81/82两个节点平分刷新压力。

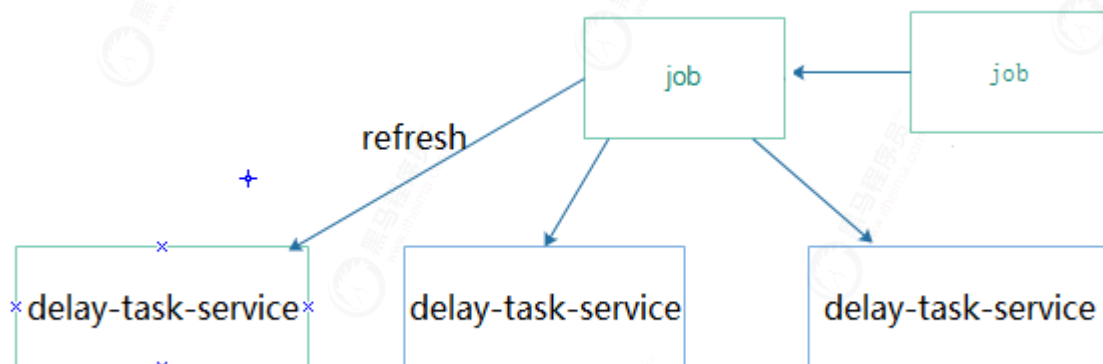
4.6.5.延迟任务高可用改造

4.6.5.1.高可用设计

1：项目启动后就开始进行的数据恢复和定时预加载，仍然是每个节点都在进行数据预加载恢复，此时会造成数据错误，因此只需要让其中一个节点进行这个操作即可，即要做集群选主，让主节点做数据恢复，从节点作为备份，如果主节点宕机，从诸多从节点中选取新的主节点继续进行数据恢复操作。



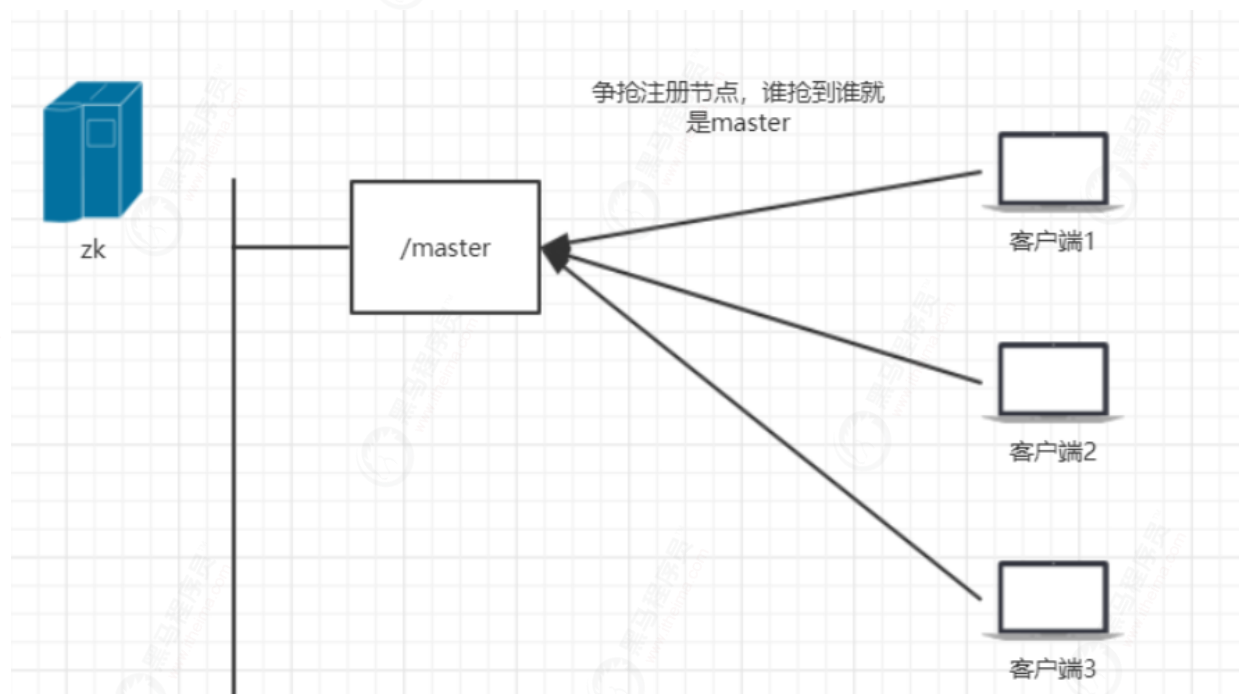
2：job节点开发完成后，延迟任务系统的定时刷新已经从单节点调度变成了统一的分布式调度，而对于job系统来说，为了保证其高可用，需要部署多个job节点，但是只需要让其中一个主节点进行定时刷新调度，其他节点备份，一旦主机宕机，从节点立马升级为主节点继续定时刷新调度，不中断业务的进行



4.6.5.2.ZK实现集群选主

zookeeper是一个成熟的分布式协调服务，通过使用zookeeper我们可以较为方便的实现集群的选举。

zookeeper基于目录监听机制来选主，多个客户端节点都可以来对zookeeper上某个目录节点进行监听和注册，但只有其中一个能够注册上，谁能注册上谁就是主节点，主节点会保持着和zookeeper目录节点的长连接，只要该连接不断，那么该客户端节点就一直是主节点，其他客户端节点都会监听者zookeeper的该目录节点，一旦主节点宕机，会立即从其他客户端节点中选出一个新的主节点，也就是说：如果当前master宕机，会立即选出一个新的master。



要想了解如何使用zookeeper实现选举，首先需要了解zookeeper节点的类型

Zookeeper节点类型

- PERSISTENT：该值会永久存在，哪怕创建该节点的机器挂了，节点数据依然会存在。注意，如果有两台机器创建了重复的key，比如/data，第二次创建会失败。
- PERSISTENT_SEQUENTIAL：比如我们创建一个/test节点，zk会在后面加一串数字比如/test/test0000000001。如果重复创建，会创建一个/test/test0000000002节点（一直往后加1，可以多次创建）
- EPHEMERAL：临时节点，当创建该节点的机器失连了，创建的这个节点会被删除

- EPHEMERAL_SEQUENTIAL: 和 PERSISTENT_SEQUENTIAL差不多, 只是节点是临时的

而使用zookeeper进行集群选举又可分为: **公平模式**和**非公平模式**

非公平模式: 假设有100台机器进行选举, 最后会选到哪一个机器, 是完全随机的(看谁抢的快)。比如选到了A机器。某一时刻, A机器挂掉了, 这时候会再次进行选举, 这一次的选举依然是随机的。与某个节点是不是先来的, 是不是等了很久无关

- 1) 首先通过zk创建一个 /zkserver 的**PERSISTENT**节点
- 2) 多台机器同时创建 /zkserver/leader **EPHEMERAL**子节点
- 3) 子节点只能创建一个, 后创建的会失败。创建成功的节点被选为leader节点
- 4) 所有机器监听 /zkserver/leader 的变化, 一旦节点被删除, 就重新进行选举, 抢占式地创建 /zkserver/leader节点, 谁创建成功谁就是leader。

公平模式: 与非公平模式的区别就是增加了先来的优先被选为leader的保证。

- 1) 首先通过zk创建一个 /zkserver 的**PERSISTENT**节点
- 2) 多台机器同时创建 /zkserver/leader **EPHEMERAL_SEQUENTIAL**子节点
- 3) /zkserver/leader000000xxx 后面数字最小的那个节点被选为leader节点
- 4) 所有机器监听 前一个 /zkserver/leader 的变化, 比如 (leader00001监听 leader00002) 一旦节点被删除, 就获取/zkserver下所有leader, 如果自己的数字最小那么自己就被选为leader

ZK选主实现

使用zookeeper自己来实现选举还是有点麻烦的, 这个时候可以使用 `curator`

`curator` 官网: <http://curator.apache.org/>

正如官网所说, curator对于zookeeper就像guava对于java, 它让我们能更加便利、可靠地使用zookeeper。

它提供了一个 `LeaderSelector` 可以帮助我们实现选主功能, 并且在模块 `virtual-common` 中提供了一个封装好的工具类可以直接使用

1: 在virtual-common模块中找到工具类: `ZKSelectMasterUtil`

2: 测试

2.1: 启动三次main方法模拟三个客户端节点, 查看每个节点的输出情况, 预计只有一个能够注册为主节点, 其他两个是从节点。

2.2: 停掉主节点, 查看控制台看剩余两个节点是否有新的主节点产生。

4.6.5.3.延迟任务选主实现

这一小节主要完成两个操作:

- 延迟任务系统通过zk完成选主, 由主节点进行数据恢复和预加载, 如果主节点宕机或者故障, 从剩余从节点中选出新的主节点继续进行数据恢复和预加载, 不中断业务的进行
- job节点同样如此, 由主节点进行全局定时调度, 从节点相当于备份, 一旦主节点故障通过重新选主后继续进行定时调度, 不中断业务的进行

1: 在consul中添加zk服务地址的配置,

```

11
12 delaytask:
13     preLoad: 1 #预加载时间配置
14     zkAddr: 192.168.200.129:2181

```

同时在 `SystemParamProperties` 中加载该配置属性

```

1  /**
2   * Created by 传智播客*黑马程序员.
3   */
4  @Data
5  @Configuration
6  @ConfigurationProperties(prefix = "delaytask")
7  public class SystemParamProperties {
8      private int preLoad;
9      private String zkAddr;
10 }

```

2: 编写 `ZkConfig`，对选主工具 `ZKSelectMasterUtil` 进行配置后将其添加到容器

创建: `com.itheima.delaytask.config.ZkConfig`

```

1  /**
2   * Created by 传智播客*黑马程序员.
3   */
4  @Configuration
5  public class ZkConfig {
6
7      @Autowired
8      private SystemParamProperties systemParamProperties;
9
10     @Bean
11     public ZKSelectMasterUtil zkSelectMasterUtil(){
12         ZKSelectMasterUtil zkSelectMasterUtil = new ZKSelectMasterUtil();
13         zkSelectMasterUtil.setZkAddress(systemParamProperties.getZkAddr());
14         return zkSelectMasterUtil;
15     }
16 }

```

3: 找到 `TaskServiceImpl` 中的数据恢复预加载方法，添加选主实现

```

1  @Autowired
2  private ZKSelectMasterUtil zkSelectMasterUtil;
3
4  @PostConstruct
5  public void syncData(){
6      //启动后去争抢注册成为主节点
7      zkSelectMasterUtil.selectMaster(ZKdirConstants.schedule_leaderPath);
8      //注册也需要有一定时间,为了后面的操作不会因为这个时间差造成误判,在这个地方宁愿等上个1两秒
9      try {

```

```

10     TimeUnit.SECONDS.sleep(1);
11 } catch (InterruptedException e) {
12 }
13
14 //首次运行即执行一次，然后按照设定的时间周期按照固定频率执行
15 threadPoolTaskScheduler.scheduleAtFixedRate(new Runnable() {
16     @Override
17     public void run() {
18         //只有主节点需要进行数据恢复预加载，从节点不用
19         if(zkSelectMasterUtil.checkMaster(ZKdirConstants.schedule_leaderPath)){
20             log.info("主节点{}开始进行定时数据恢
复", LocalDateTime.now().format(DateTimeFormatter.ISO_DATE_TIME));
21             reloadData();
22         }else {
23             log.info("从节点备份");
24         }
25     }
26 },TimeUnit.MINUTES.toMillis(paramProperties.getPreLoad()));
27 }

```

4: 测试

分别在81/82端口启动延迟任务系统，查看控制台的输出即可。

4.6.5.4.job系统选主实现

- 1: 将delay-task-service中的 `SystemParamProperties` , `ZkConfig` 拷贝到delay-task-job模块中
- 2: 在DelayTaskJob中对统一的定时刷新调度方法做出如下修改

```

1  /**
2   * Created by 传智播客*黑马程序员.
3   */
4  @Component
5  @Slf4j
6  public class DelayTaskJob {
7
8
9      @Autowired
10     private ThreadPoolTaskExecutor threadPoolTaskExecutor;
11
12     @Autowired
13     private TaskService taskService;
14
15     @Autowired
16     private ZKSelectMasterUtil zkSelectMasterUtil;
17
18     @PostConstruct
19     public void init(){
20         //开始进行选主
21         zkSelectMasterUtil.selectMaster(ZKdirConstants.job_leaderPath);
22     }
23

```

```

24     @Scheduled(cron = "*/1 * * * * ?")
25     public void refresh(){
26         //主节点才需要进行定时调度
27         if(zkSelectMasterUtil.checkMaster(ZKdirConstants.job_leaderPath)){
28             threadPoolTaskExecutor.execute(new Runnable() {
29                 @Override
30                 public void run() {
31                     log.info("job主节点{}开始进行定时刷新",
32                         LocalDateTime.now().format(DateTimeFormatter.ISO_DATE_TIME));
33                     taskService.refresh();
34                 }
35             });
36         }else {
37             log.info("job从节点备份");
38         }
39     }

```

3: 测试

测试job前先启动delay-task-service，在81/82两个端口启动，然后分别在8082/8083两个端口上启动job系统，查看输出

4.6.5.5.分布式变量共享

现在的延迟任务系统和job系统都已经完全改造成分布式了，在分布式环境下，一些关键性的系统变量涉及其他系统共用问题。

在延迟任务系统中有一个数据恢复预加载的方法，该方法会定时的执行，且在该方法中有一个非常重要的变量：`nextScheduleTime`，该变量的值是每次在进行数据恢复预加载的时候定义的未来2分钟的时间节点

```

.....//定义未来2分钟的时间节点变量
.....Calendar calendar = Calendar.getInstance();
.....calendar.add(Calendar.MINUTE,paramProperties.getPreLoad());
.....nextScheduleTime = calendar.getTimeInMillis();

```

并且该时间节点的值在添加任务的方法中也使用到了，因为在将任务添加到缓存的时候会判断，如果任务执行时间在将来2分钟时间内才会将其存入zset排序等待，否则任务数据只是入库

```

private void addTaskToCache(Task task) {
    //要根据任务类型和优先级进行分组，每组的key不一样
    String key = task.getTaskType()+"_"+task.getPriority();
    //根据时间判断
    if(task.getExecuteTime() <= System.currentTimeMillis()){
        //任务进入消费者队列List--数据从左侧入队列，消费时从右侧出队列--保证任务消费的一个顺序性
        //同时我们让所有分组的key都有一个相同的前缀，方便进行管理
        cacheService.lLeftPush(DelayTaskRedisKeyConstants.TOPIC+key,JSON.toJSONString(task));
    }else if(task.getExecuteTime() <= nextScheduleTime){
        //任务进入zset排序等待
        cacheService.zAdd( key: DelayTaskRedisKeyConstants.FUTURE+key,JSON.toJSONString(task),task.getExecuteTime());
    }
    //先暂时将数据存储到zset，后续会对此优化
    //cacheService.zAdd(DelayTaskRedisKeyConstants.DBCACHE, JSON.toJSONString(task),task.getExecuteTime());
}

```

出现的问题是：系统改造成分布式后，主节点该变量是一直在变的，因为主节点一直在做数据恢复预加载，但是从节点由于不会执行数据恢复预加载方法reload()，因此从节点中该变量是一直不会变的，除非等待该从节点升级为主节点；但是从节点还是支持任务的添加的，由此会造成但凡是在从节点上添加的任务只要不是当前立马要执行的都不会进入到缓存，因为 addTaskToCache 方法的判断永远不会通过

怎么解决呢？要让该变量全局可见，保证全局同步和使用，可以持久化到数据库，或者放到缓存redis

代码改造：

1：改造 reload 方法如下

```
1 private void reloadData(){
2     //-----前面的代码没有变化-----
3     //定义未来2分钟的时间节点变量
4     Calendar calendar = Calendar.getInstance();
5     calendar.add(Calendar.MINUTE,paramProperties.getPreLoad());
6     nextScheduleTime = calendar.getTimeInMillis();
7     //将这个变量存储在redis中
8     cacheService.set(DelayTaskRedisKeyConstants.nextScheduleTime,nextScheduleTime+"");
9     //-----后面的代码没有变化-----
10 }
```

2：改造添加任务到缓存的方法

```
1 private void addTaskToCache(Task task) {
2     //要根据任务类型和优先级进行分组，每组的key不一样
3     String key = task.getTaskType()+"_"+task.getPriority();
4     //从全局获取未来的时间节点
5     long nextScheduleTime = fetchNextScheduleTime();
6     //根据时间判断
7     if(task.getExecuteTime() <= System.currentTimeMillis()){
8         //任务进入消费者队列List---数据从左侧入队列，消费时从右侧出队列--保证任务消费的一个顺序性
9         //同时我们让所有分组的key都有一个相同的前缀，方便进行管理
10
11         cacheService.lLeftPush(DelayTaskRedisKeyConstants.TOPIC+key,JSON.toJSONString(task));
12     }else if(task.getExecuteTime() <= nextScheduleTime){
13         //任务进入zset排序等待
14
15         cacheService.zAdd(DelayTaskRedisKeyConstants.FUTURE+key,JSON.toJSONString(task),task.getExecuteTime());
16     }
17 }
18
19 private long fetchNextScheduleTime() {
20     //看缓存中是否该数据
21     if(cacheService.exists(DelayTaskRedisKeyConstants.nextScheduleTime)){
22         String nextScheduleTime =
23         cacheService.get(DelayTaskRedisKeyConstants.nextScheduleTime);
24         log.info("从缓存中获取了nextScheduleTime:{}",nextScheduleTime);
25         return Long.parseLong(nextScheduleTime);
26     }
27 }
```

```

24 //缓存中没有则进行一个数据补偿,已当前时间为基础,计算未来的时间
25 Calendar calendar = Calendar.getInstance();
26 calendar.add(Calendar.MINUTE,paramProperties.getPreLoad());
27 this.nextScheduleTime = calendar.getTimeInMillis();
28 cacheService.set(DelayTaskRedisKeyConstants.nextScheduleTime,nextScheduleTime+"");
29 log.info("缓存中没有nextScheduleTime,数据补偿{}",nextScheduleTime);
30 return nextScheduleTime;
31 }

```

3: 测试

1: 分别在81/82端口启动delay-task-service, 查看控制台输出一主一从, 查看缓存中是否已有对应的变量值

2: 在job中添加如下测试类: com.itheima.job.TaskServiceTest

```

1 /**
2  * Created by 传智播客*黑马程序员.
3  */
4 @SpringBootTest(classes = {JobApplication.class})
5 @RunWith(SpringJUnit4ClassRunner.class)
6 public class TaskServiceTest {
7
8     @Autowired
9     private TaskService taskService;
10
11     @Test
12     public void test1(){
13         //通过feign调用4次添加任务
14         for(int i=0;i<4;i++){
15             Task task = new Task();
16             task.setTaskType(1001);
17             task.setPriority(1);
18             task.setParameters("task".getBytes());
19             task.setExecuteTime((LocalDateTime.now().getSecond()+i)*1000);
20             taskService.push(task);
21         }
22     }
23
24 }

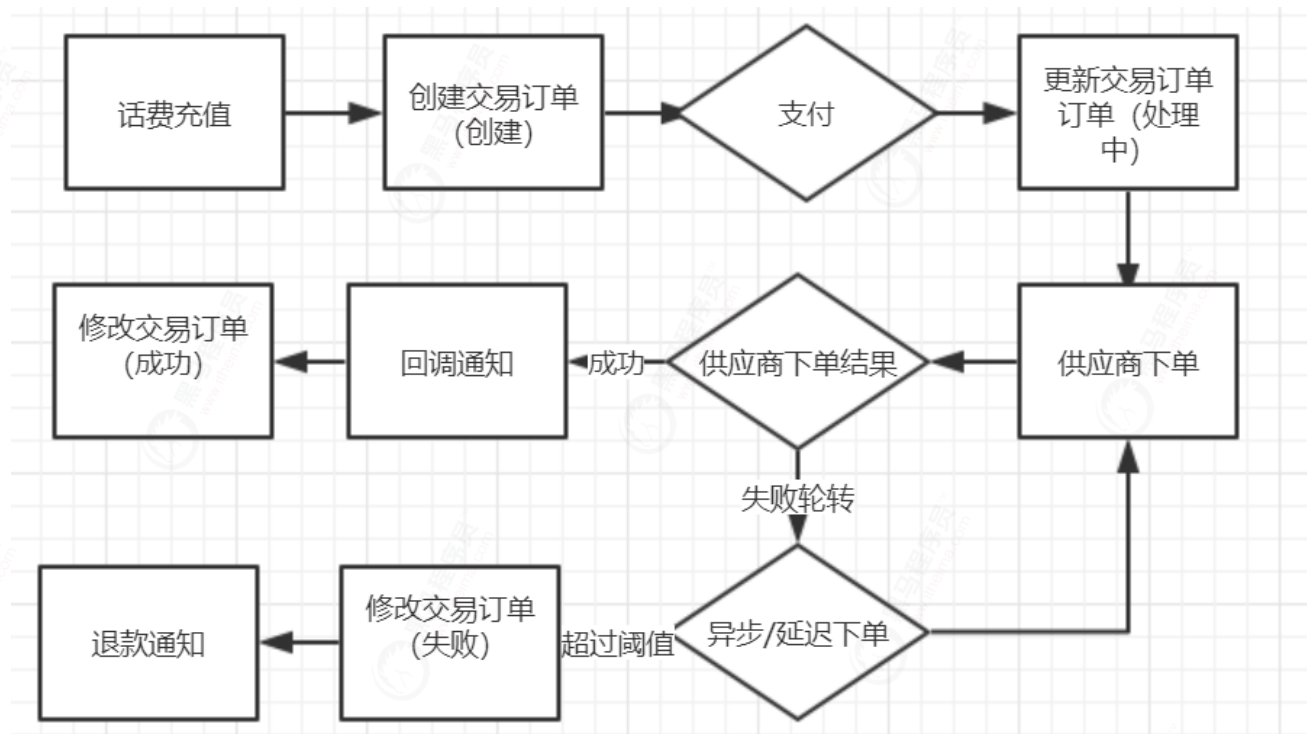
```

因为feign集成了负载均衡, 因此会有添加任务的请求被分配到从节点上, 查看从节点添加任务时是否能从缓存中提取出预加载时间变量

5.系统服务设计与开发

在虚拟商品交易架构体系中的系统服务设计上, 我们选择以话费充值作为具体的业务场景进行落地实践。

由于虚拟业务交易同样也是一种电商交易，在业务上跟传统的实体电商是有重合的，因此也涉及到一些基础业务，譬如：商品（品牌，分类），订单，支付，搜索等；我们在此只深挖虚拟交易自身业务和技术特点；我们会模拟出在话费充值业务场景中所面临的一些问题，我们可以先来看话费充值的业务逻辑



虚拟业务最大技术特点和对接过程中由于网络，双方系统各种问题，会有延迟，重试，及最后订单同步成功失败等一些列问题，所以项目在话费充值业务这块主要实现了

- 1: 接收支付成功的消息然后去和供应商对接
- 2: 对接失败进行重试，并且限制了重试次数
- 3: 平台余额不足进行供应商的轮转
- 4: 网络异常进行重试
- 5: 对接成功后修改订单状态

完整实现后的工程模块图如下：



mock-web：话费服务工程，包含了页面，生成订单mock，支付mock，

mock-order: 订单追踪模块mock, 包含操作订单的Mapper和订单实体

mock-thirdparty: 供应商模块mock, 模拟供应商系统 (由于一些现实问题并未对接实际第三方供应商: 押金, 合同)

supplier-business: 对接供应商下单模块, 负责和各个供应商对接, 处理过程中需要使用我们的基础组件-延迟任务, 订单支付成功后会以异步的形式通知该模块, 因此该模块中的消息监听 `PaySuccessListener` 就是我们业务的开始, 监听方法上有支付成功后异步传递的参数 `RechargeRequest`

5.1.供应商接口对接

1: `supplier-business` 监听到支付成功消息之后, 接收到的消息: `RechargeRequest`, 具体对接的不直接写在监听类中, 单独定义一个对接供应商服务接口

创建接口: `com.itheima.supplier.inf.SupplierService`,

```
1  /**
2   * Created by 传智播客*黑马程序员.
3   */
4  public interface SupplierService {
5      /**
6       * 对接供应商充值下单
7       * @param rechargeRequest
8       */
9      public void recharge(RechargeRequest rechargeRequest);
10 }
```

2: 创建接口的实现: `com.itheima.supplier.service.SupplierServiceImpl`

```
1  /**
2   * Created by 传智播客*黑马程序员.
3   */
4  @Service
5  @Slf4j
6  public class SupplierServiceImpl implements SupplierService {
7      @Override
8      public void recharge(RechargeRequest rechargeRequest) {
9      }
10 }
11 }
```

3: 配置供应商系统的接口地址, 在实际业务中调用第三方系统接口大都基于HTTP协议调用, 我们在 `mock-thirdparty` 模块中模拟了两个供应商 (聚合, 极速), 注意这个模块并没有在充吧的注册中心去注册, 因为我们模拟的是外部系统, 所以我们要调用外部系统必须得知道系统接口的调用地址;

在consul中创建配置: `config/supplier/data`, 添加如下配置, 同时也可将 `application.yml` 中的相关配置托管到consul中

```

1 supplier:
2   apis: {
3     "jisuapi": "http://127.0.0.1:8090/jisuapi/mobilerecharge/recharge",
4     "juheapi": "http://127.0.0.1:8090/juheapi/recharge"
5   }

```

这两个接口分别定义在 `mock-thirdparty` 中，可找到我们对两个第三方供应商的mock代码

4: 编写配置类读取配置，创建: `com.itheima.supplier.properties.SupplierProperties`

```

1 /**
2  * Created by 传智播客*黑马程序员.
3  */
4 @Data
5 @Configuration
6 @ConfigurationProperties(prefix = "supplier")
7 public class SupplierProperties {
8     private Map<String,String> apis; //加载供应商api地址,key是第三方供应商编号,value是其地址
9 }
10

```

并将其注入到 `SupplierServiceImpl` 中

5: 在mq的监听方法中添加接口方法的调用，在支付成功后监听到消息，然后进行第三方充值接口的对接，找到 `PaySuccessListener`，处理如下

```

1 /**
2  * Created by 传智播客*黑马程序员.
3  */
4 @Component
5 @Slf4j
6 public class PaySuccessListener {
7
8     @Autowired
9     private SupplierService supplierService;
10
11     /**
12      * 监听消息:
13      * @param rechargeRequest
14      */
15     @RabbitListener(queues = {"pay"})
16     public void onMessage(RechargeRequest rechargeRequest) {
17         log.info("RabbitListener 监听到了消息,{}",rechargeRequest);
18         //充值对接
19         supplierService.recharge(rechargeRequest);
20     }
21 }

```

6: 接着需要在 `SupplierServiceImpl` 中去实现充值对接的方法 `recharge`，对接第三方的思路和逻辑如下：

- 编写对接逻辑分发方法：Result doDispatchSupplier(RechargeRequest rechargeRequest)，根据供应商编号进行分发
- 编写对接聚合的方法：Result doPostJuhe(RechargeRequest rechargeRequest)
- 编写对接极速的方法：Result doPostJisu(RechargeRequest rechargeRequest)

对接第三方其实就是向第三方系统接口地址发起HTTP请求，我们可以使用HttpClient等工具，为了实现起来更加的方便，我们使用Spring提供的模板：`RestTemplate`

```

1  /**
2   * Created by 传智播客*黑马程序员.
3   */
4  @Service
5  @Slf4j
6  public class SupplierServiceImpl implements SupplierService {
7      @Autowired
8      private SupplierProperties supplierProperties;
9
10     @Override
11     public void recharge(RechargeRequest rechargeRequest) {
12         //对接逻辑分发
13         Result<RechargeResponse> responseResult = doDispatchSupplier(rechargeRequest);
14     }
15
16     //对接逻辑的分发处理
17     private Result<RechargeResponse> doDispatchSupplier(RechargeRequest rechargeRequest){
18         //获取第三方供应商的接口地址,根据供应商编号获取
19         String url = supplierProperties.getApis().get(rechargeRequest.getSupply());
20         rechargeRequest.setRechargeUrl(url);
21         //根据需要对接的供应商的编号确定不同的对接方式---不同的api需要传递的参数类型和参数名称等各不相同
22         if(SupplierConstants.juheapi.equals(rechargeRequest.getSupply())){
23             //对接聚合
24             return doPostJuhe(rechargeRequest);
25         }else if(SupplierConstants.jisuapi.equals(rechargeRequest.getSupply())){
26             //对接极速
27             return doPostJisu(rechargeRequest);
28         }
29
30         return null;
31     }
32
33     private Result<RechargeResponse> doPostJisu(RechargeRequest rechargeRequest) {
34         return null;
35     }
36
37     private Result<RechargeResponse> doPostJuhe(RechargeRequest rechargeRequest) {
38         return null;
39     }
40 }

```

7: 完成对接聚合逻辑的编写；聚合给我们的返回的是一种余额不足的场景

```

1  @Autowired
2  private RestTemplate restTemplate;
3
4  private Result<RechargeResponse> doPostJuhu(RechargeRequest rechargeRequest) {
5      //聚合要求传递的是json格式的数据
6      //创建并设置请求头
7      HttpHeaders headers = new HttpHeaders();
8      headers.setContentType(MediaType.APPLICATION_JSON);
9      //创建请求实体
10     HttpEntity httpEntity = new HttpEntity(JSON.toJSONString(rechargeRequest), headers);
11     //发送请求
12     ResponseEntity<String> responseEntity =
13     restTemplate.postForEntity(rechargeRequest.getRechargeUrl(), httpEntity, String.class);
14     //获得结果
15     /*String body = responseEntity.getBody();// Result<RechargeResponse>
16     Result result = JSON.parseObject(body, Result.class);*/因为泛型的问题可能会导致出现问题
17     System.out.println(body);*/
18     Result<RechargeResponse> result = JSON.parseObject(responseEntity.getBody(), new
19     TypeReference<Result<RechargeResponse>>(){});
20     return result;
21 }

```

8: 完成对接极速逻辑的编写; 极速能给我们返回成功和失败的场景

```

1  log.info("doPostJisu,{", rechargeRequest);
2  HttpHeaders headers = new HttpHeaders();
3  headers.setContentType(MediaType.APPLICATION_FORM_URLENCODED);
4  //设置表单参数
5  MultiValueMap<String, String> map= new LinkedMultiValueMap<String, String>();
6  map.add("mobile", rechargeRequest.getMobile());
7  map.add("amount", rechargeRequest.getPamt()+"");
8  map.add("outorderNo", rechargeRequest.getOrderNo());
9  map.add("repeat", ""+rechargeRequest.getRepeat());
10
11  //模拟请求失败
12  map.add("req_status", ""+StatusCode.ERROR);
13  HttpEntity<MultiValueMap<String, String>> requestEntity = new
14  HttpEntity<MultiValueMap<String, String>>(map, headers);
15  ResponseEntity<String> responseEntity =
16  restTemplate.postForEntity(rechargeRequest.getRechargeUrl(), requestEntity, String.class);
17  //转换成统一对象
18  Result<RechargeResponse> result= JSON.parseObject(responseEntity.getBody(), new
19  TypeReference<Result<RechargeResponse>>(){});
20  return result;

```

9: 测试

启动mock-web工程, mock-thirdparty工程, supplier-business工程

访问: <http://localhost:191/> 进行充值, 查看结果

5.2.供应商接口重试

5.2.1.重试逻辑编写

对接第三方供应商我们需要根据结果进行重试，主要是以下两种业务场景：

- 网络故障/充值失败重试，需要添加一个重试任务
- 重试次数到达阈值后停止供应商对接

此时我们将面临需要在第三方供应商对接模块 `supplier-business` 中添加和消费第三方供应商重试任务，

我们在 `supplier-business` 中编写一个跟延迟任务相关的接口，在这个接口中处理相关任务的添加，取消和消费工作，这样做的好处是将供应商对接逻辑和任务相关的逻辑分开

1：创建接口：com.itheima.supplier.inf.SupplierTask

```
1  /**
2   * Created by 传智播客*黑马程序员.
3   */
4  public interface SupplierTask {
5      /**
6       * 添加重试任务
7       * @param rechargeRequest
8       */
9      public void addRetryTask(RechargeRequest rechargeRequest);
10     /**
11      * 消费请求失败重试任务
12      */
13     public void retryRecharge();
14 }
```

代码解释：添加重试任务方法：addRetryTask(RechargeRequest rechargeRequest)参数为什么传RechargeRequest？

任务对象Task中有一个参数parameter，存储实际的任务数据，后期消费时会提取任务中的parameter参数进行实际的业务操作，对于重试任务，任务消费后要重新去对接供应商，需要再次调用SupplierServiceImpl类中的对接方法：recharge(RechargeRequest rechargeRequest)，该方法参数需要RechargeRequest，所以我们添加重试任务的时候方法也传参：RechargeRequest

2：创建接口实现：com.itheima.supplier.service.SupplierTaskImpl

```
1  /**
2   * Created by 传智播客*黑马程序员.
3   */
4  @Service
5  @Slf4j
6  public class SupplierTaskImpl implements SupplierTask {
7      @Autowired
8      private TaskService taskService;
9      @Override
10     public void addRetryTask(RechargeRequest rechargeRequest) {
11     }
12     @Override
13     public void retryRecharge() {
14     }
```

```
15 }
```

添加消费重试任务需要调用延迟任务接口，因此需要注入延迟任务feign接口：TaskService

同时：需要在启动类 SupplierApplication 上添加扫描feign接口的注解：

```
@EnableFeignClients({"com.itheima.delaytask.feign"})
```

3：完成添加重试任务方法逻辑，延迟任务系统添加任务需要传递的是任务对象：Task，而addRetryTask方法接收的是一个RechargeRequest方法，因此需要转换，逻辑实现如下：

```
1 @Override
2 public void addRetryTask(RechargeRequest rechargeRequest) {
3     Task task = new Task();
4     //根据供应商返回的错误码来决定我们任务的类型
5     TaskTypeEnum taskType = TaskTypeEnum.getTaskType(rechargeRequest.getErrorCode());
6     task.setTaskType(taskType.getTaskType());
7     task.setPriority(taskType.getPriority());
8     //任务的执行时间---实际业务中可能是1分钟后重试或者30秒后重试
9     Calendar calendar = Calendar.getInstance();
10    calendar.add(Calendar.SECOND,2);//为了测试方便我们选择2秒后重试
11    task.setExecuteTime(calendar.getTimeInMillis());
12
13    //设置任务参数
14    task.setParameters(ProtostuffUtil.serialize(rechargeRequest));
15    //添加任务
16    taskService.push(task);
17 }
```

代码解释1：任务类型和优先级如何确定？

在chongba_common工程中有一个任务类型和优先级的枚举类：TaskTypeEnum，根据不同的业务状态码绑定不同的任务类型和优先级，这些业务状态码都定义在了：StatusCode类中，在添加重试任务方法的参数RechargeRequest中有一个属性：errorCode代表了业务类型错误码。

```
1 @Getter
2 public enum TaskTypeEnum {
3     ORDER_REQ_FAILED(StatusCode.ORDER_REQ_FAILED, 1001, 1,"话费充值失败，重试"),
4     BALANCE_NOT_ENOUGH(StatusCode.BALANCE_NOT_ENOUGH, 1001, 2,"供应商余额不足，轮转其他供应商"),
5     REMOTEERROR(StatusCode.REMOTEERROR, 1001, 3,"远程调用失败，重试"),
6     STATECHECK(StatusCode.STATECHECK, 1001, 4,"检查订单状态");
7     private int errorCode; //错误码
8     private int taskType; //对应具体业务
9     private int priority; //业务不同级别
10    private String desc; //描述信息
11 }
```

代码解释2：重试任务执行时间如何确定？

重试的时间间隔可以根据业务情况自行定制，可以每隔1分钟重试1次，或者自定义其他时间趋势比如可以采用RechargeRequest对象中的重试次数repeat作为下次重试时间，即呈现：1，2，3，4逐次递增的形式

4: 消费对接重试任务，完成重试下单方法：retryRecharge的逻辑，重试下单其实就是根据重任务的类型和优先级调用延迟任务系统拉取任务，得到任务对象Task，从Task对象中获取其参数parameter反序列化为对接请求对象：RechargeRequest，调用对接供应商接口的recharge(RechargeRequest rechargeRequest)方法

```
1  @Autowired
2  private SupplierService supplierService;
3
4  @Override
5  @Scheduled(fixedRate = 1000)
6  public void retryRecharge() {
7      retry(TaskTypeEnum.ORDER_REQ_FAILED); //消费话费充值失败进行重试的任务，还有别的类型的重试任务
8  }
9
10 /**
11  * 根据任务类型的枚举对象去消费任务
12  * @param taskTypeEnum
13  */
14 private void retry(TaskTypeEnum taskTypeEnum){
15     ResponseMessage responseMessage = taskService.poll(taskTypeEnum.getTaskType(),
16     taskTypeEnum.getPriority());
17     if(!responseMessage.isFlag() || responseMessage.getCode() == StatusCode.ERROR ){
18         return;
19     }
20     if(responseMessage.getData() != null){
21         String taskStr = JSON.toJSONString(responseMessage.getData());
22         Task task = JSON.parseObject(taskStr, Task.class);
23         RechargeRequest rechargeRequest = ProtostuffUtil.deserialize(task.getParameters(),
24         RechargeRequest.class);
25         //每重试一次+1
26         rechargeRequest.setRepeat(rechargeRequest.getRepeat()+1);
27         log.info("{}消费重试任务,任务类型:{},任务参数:{}",
28         LocalDateTime.now().format(DateTimeFormatter.ISO_DATE_TIME), taskTypeEnum.getTaskType(), rechargeRequest);
29         //重试对接
30         supplierService.recharge(rechargeRequest);
31     }
32 }
```

注意：需要去到启动类：SupplierApplication添加开启定时调度的注解：@EnableScheduling

代码解释1：为什么要定时？

对接供应商下单重试任务要得到消费，就必须执行retryRecharge方法，如何发起调用，以前我们消费任务是循环的去拉取任务，因此我们在这可以做一个定时，每隔1秒钟去拉取重试任务。

代码解释2：如何正确的从延迟任务系统返回的ResponseMessage对象中获取任务对象Task？

```

1 // ResponseMessage定义
2 @Data
3 public class ResponseMessage {
4     private boolean flag;
5     private Integer code;
6     private String message;
7     private Object data;
8     private String url;
9 }
10 //获取方式1:
11 Task task = (Task)responseMessage.getData();

```

如果这么获取会报如下错误:

```

1 java.lang.ClassCastException:java.util.LinkedHashMap cannot be cast to
   com.chongba.entity.Task

```

原因是feign帮我们在类型封装的时候把Object类型的封装成了LinkedHashMap, 所以我们可以通过转成json字符串然后再反转成对象的方式来规避这个问题。

5: 模拟对接重试业务并测试, 在SupplierServiceImp类中的对接方法中模拟失败重试,

```

1 @Autowired
2 private SupplierTask supplierTask;
3
4 @Override
5 public void recharge(RechargeRequest rechargeRequest) {
6     //对接逻辑分发
7     Result<RechargeResponse> responseResult = doDispatchSupplier(rechargeRequest);
8     //模拟重试逻辑的编写---添加重试任务
9     rechargeRequest.setErrorCode(StatusCode.ORDER_REQ_FAILED);
10    supplierTask.addRetryTask(rechargeRequest);
11 }

```

测试:

启动各个系统: 延迟任务在81/82端口启动, job节点, mock-thirdparty, mock-web; 然后访问:
<http://localhost:191>, 进行话费充值业务, 看是否有延迟任务的添加和消费

测试结果预期: 能够正常添加失败重试任务, 消费任务, 但是会出现死循环

```
[...] scheduling-1] c.i.s.service.SupplierServiceImpl..... : 开始对接聚合api: doPostJisu, RechargeRequest(orderNo=nul
[...] scheduling-1] c.i.s.service.SupplierServiceImpl..... : 对接聚合的返回结果: {"code":20006,"msg":"余额不足","data"
[...] scheduling-1] c.i.supplier.service.SupplierTaskImpl..... : 2020-05-27T23:56:30.545消费重试任务, 任务类型:1001, 任务参
[...] scheduling-1] c.i.s.service.SupplierServiceImpl..... : 开始对接聚合api: doPostJisu, RechargeRequest(orderNo=nul
[...] scheduling-1] c.i.s.service.SupplierServiceImpl..... : 对接聚合的返回结果: {"code":20006,"msg":"余额不足","data"
[...] scheduling-1] c.i.supplier.service.SupplierTaskImpl..... : 2020-05-27T23:56:31.544消费重试任务, 任务类型:1001, 任务参
[...] scheduling-1] c.i.s.service.SupplierServiceImpl..... : 开始对接聚合api: doPostJisu, RechargeRequest(orderNo=nul
[...] scheduling-1] c.i.s.service.SupplierServiceImpl..... : 对接聚合的返回结果: {"code":20006,"msg":"余额不足","data"
[...] scheduling-1] c.i.supplier.service.SupplierTaskImpl..... : 2020-05-27T23:56:32.545消费重试任务, 任务类型:1001, 任务参
[...] scheduling-1] c.i.s.service.SupplierServiceImpl..... : 开始对接聚合api: doPostJisu, RechargeRequest(orderNo=nul
[...] scheduling-1] c.i.s.service.SupplierServiceImpl..... : 对接聚合的返回结果: {"code":20006,"msg":"余额不足","data"
[...] scheduling-1] c.i.supplier.service.SupplierTaskImpl..... : 2020-05-27T23:56:33.553消费重试任务, 任务类型:1001, 任务参
[...] scheduling-1] c.i.s.service.SupplierServiceImpl..... : 开始对接聚合api: doPostJisu, RechargeRequest(orderNo=nul
[...] scheduling-1] c.i.s.service.SupplierServiceImpl..... : 对接聚合的返回结果: {"code":20006,"msg":"余额不足","data"
```

原因是什么呢? 是因为我们没有对重试次数加以限制, 所以造成了死循环!!!

5.2.2.重试次数限制

1: 在consul中supplier的配置下添加重试次数限制

```
1 supplier:
2   apis: {
3     "jisuapi": "http://127.0.0.1:8090/jisuapi/mobilerecharge/recharge",
4     "juheapi": "http://127.0.0.1:8090/juheapi/recharge"
5   }
6   maxrepeat: 4 #最大重试次数
```

2: 在对应的属性配置类中添加对应的属性以加载该数据

```
1 @Data
2 @Configuration
3 @ConfigurationProperties(prefix = "supplier")
4 public class SupplierProperties {
5     private Map<String,String> apis; //加载供应商api地址,key是第三方供应商编号,value是其地址
6     private int maxrepeat; //最大重试次数
7 }
```

3: 在对接供应商服务接口实现类: SupplierServiceImpl中, 改造对接供应商下单方法: recharge(RechargeRequest rechargeRequest), 添加失败重试次数限制。如果超过了最大次数, 停止供应商对接, 订单失败

```
1 @Override
2 public void recharge(RechargeRequest rechargeRequest) {
3     //限制最大重试次数 如果超过了最大次数, 停止供应商对接,
4     if(rechargeRequest.getRepeat() > supplierProperties.getMaxrepeat()){
5         //订单失败, 订单失败后实际业务中是对接订单服务进行退款等后续业务操作
6         updateTrade(rechargeRequest.getOrderNo(), OrderStatusEnum.FAIL.getCode());
7         return;
8     }
9     //对接逻辑分发
10    Result<RechargeResponse> responseResult = doDispatchSupplier(rechargeRequest);
```

```

11 //模拟重试逻辑的编写---添加重试任务
12 rechargeRequest.setErrorCode(StatusCode.ORDER_REQ_FAILED);
13 supplierTask.addRetryTask(rechargeRequest);
14 }

```

需要定义一个修改订单状态的方法，其中关于订单状态在common工程中定义了一个订单状态的枚举类：OrderStatusEnum

注意：在这个类中所有跟订单相关的操作逻辑在实际的业务中都应该是对接订单服务，调用订单服务完成订单的操作，我们在这只是模拟了一下。实际上是去对接订单系统

```

1 @Autowired
2 private OrderTradeMapper orderTradeMapper;
3 private void updateTrade(String orderNo, int orderStatus) {
4     //修改订单状态
5     QueryWrapper<OrderTrade> queryWrapper = new QueryWrapper<>();
6     queryWrapper.eq("order_no", orderNo);
7     OrderTrade orderTrade = orderTradeMapper.selectOne(queryWrapper);
8     if(orderTrade!=null) {
9         orderTrade.setOrderStatus(orderStatus);
10        orderTradeMapper.updateById(orderTrade);
11    }
12 }

```

4: 重启 supplier-business，重新进行充值，再次测试

5.2.3.重试业务完善

在对接第三方供应商的过程中重试任务的添加，消费，及重试次数限制均已完成，那到底在什么条件下需要开启重试任务呢？一般会有如下两种情况

- 因为网络的原因在对接过程中如果产生了异常，我们需要重试
- 由于对方系统的原因充值失败，我们需要重试

因此我们需要在对接充值下单的方法 recharge 方法中去完成实际的业务场景

1: 如果是因为因为网络的原因在对接过程中如果产生了异常，在对接供应商的方法：SupplierServiceImpl类中的recharge方法中，对调用供应商的代码块加上try{}catch{}，捕获到异常后，添加重试任务，任务类型枚举为：TaskTypeEnum.REMOTEERROR，业务状态码为：StatusCode.REMOTEERROR

```

1 @Override
2 public void recharge(RechargeRequest rechargeRequest) {
3     //限制最大重试次数 如果超过了最大次数，停止供应商对接，
4     if(rechargeRequest.getRepeat() > supplierProperties.getMaxrepeat()){
5         //订单失败,订单失败后实际业务中是对接订单服务进行退款等后续业务操作
6         updateTrade(rechargeRequest.getOrderNo(), OrderStatusEnum.FAIL.getCode());
7         return;
8     }
9     //对接逻辑分发
10    Result<RechargeResponse> responseResult;
11    try {
12        responseResult = doDispatchSupplier(rechargeRequest);

```

```

13     } catch (Exception e) {
14         log.error("充值对接下单失败,{},{},{}",rechargeRequest,e.getMessage());
15         //添加远程调用失败进行重试的任务
16         rechargeRequest.setErrorCode(StatusCode.REMOTEERROR);
17         supplierTask.addRetryTask(rechargeRequest);
18         return;
19     }
20     //模拟重试逻辑的编写---添加重试任务
21     /*rechargeRequest.setErrorCode(StatusCode.ORDER_REQ_FAILED);
22     supplierTask.addRetryTask(rechargeRequest);*/
23 }

```

需要来消费这种重试任务，找到 `SupplierTask` 接口，添加消费因为网络远程调用失败的重试任务

```

1 /**
2  * 远程调用异常重试
3  */
4 public void rechargeException();

```

在 `SupplierTask` 接口的实现中进行实现

```

1 @Override
2 @Scheduled(fixedRate = 1000)
3 public void rechargeException() {
4     retry(TaskTypeEnum.REMOTEERROR); //复用之前的重试方法，传递不同的任务类型即可
5 }

```

2: 如果是由于对方系统的原因充值失败，需要重试，我们通过对接口对接返回的结果进行判断即可

```

1 @Override
2 public void recharge(RechargeRequest rechargeRequest) {
3     //限制最大重试次数 如果超过了最大次数，停止供应商对接，
4     if(rechargeRequest.getRepeat() > supplierProperties.getMaxrepeat()){
5         //订单失败,订单失败后实际业务中是对接订单服务进行退款等后续业务操作
6         updateTrade(rechargeRequest.getOrderNo(), OrderStatusEnum.FAIL.getCode());
7         return;
8     }
9     //对接逻辑分发
10    Result<RechargeResponse> responseResult;
11    try {
12        responseResult = doDispatchSupplier(rechargeRequest);
13    } catch (Exception e) {
14        log.error("充值对接下单失败,{},{},{}",rechargeRequest,e.getMessage());
15        //添加远程调用失败进行重试的任务
16        rechargeRequest.setErrorCode(StatusCode.REMOTEERROR);
17        supplierTask.addRetryTask(rechargeRequest);
18        return;
19    }
20    if(responseResult == null){
21        //对方系统无返回,添加充值请求失败重试的任务

```

```

22     rechargeRequest.setErrorCode(StatusCode.ORDER_REQ_FAILED);
23     supplierTask.addRetryTask(rechargeRequest);
24     return;
25 }
26 if(responseResult.getCode() != StatusCode.OK){
27     //充值对接失败---需要按失败原因进行区分,可能需要添加不同的任务
28     if(responseResult.getCode() == StatusCode.ORDER_REQ_FAILED){
29         rechargeRequest.setErrorCode(StatusCode.ORDER_REQ_FAILED);
30         supplierTask.addRetryTask(rechargeRequest);
31         return;
32     }
33 }
34 //充值对接成功
35
36 }

```

5.3.供应商接口轮转

5.3.1.供应商轮转与排除

我们在做接口对接时有一个默认对接的第三方供应商：聚合平台，聚合返回的是余额不足，也就是说聚合平台因为我们在他们那的余额不足，导致他们的api对我们来说是永远调不通的，此时我们就需要轮转到跟我们平台合作的其他第三方供应商平台。因此我们要从剩下的供应商中选出一个供应商进行对接的调用。此外还需将余额不足的供应商做一个排除，下次调用时直接跳过这些供应商

做供应商排除时：我们可以将供应商编号缓存到redis中，使用Set集合存储，轮转时获取下一个供应商则直接从供应商属性配置类中读取所有的供应商地址列表，找出一个作为新的供应商，但得保证不是被排除的

1：根据对接返回结果错误码进行判断，如果需要进行轮转，

```

1  @Autowired
2  private CacheService cacheService;
3
4  @Override
5  public void recharge(RechargeRequest rechargeRequest) {
6      //限制最大重试次数 如果超过了最大次数，停止供应商对接，
7      if(rechargeRequest.getRepeat() > supplierProperties.getMaxrepeat()){
8          //订单失败,订单失败后实际业务中是对接订单服务进行退款等后续业务操作
9          updateTrade(rechargeRequest.getOrderNo(), OrderStatusEnum.FAIL.getCode());
10         return;
11     }
12     //对接逻辑分发
13     Result<RechargeResponse> responseResult;
14     try {
15         responseResult = doDispatchSupplier(rechargeRequest);
16     } catch (Exception e) {
17         log.error("充值对接下单失败,{},{},{}",rechargeRequest,e.getMessage());
18         //添加远程调用失败进行重试的任务
19         rechargeRequest.setErrorCode(StatusCode.REMOTEERROR);
20         supplierTask.addRetryTask(rechargeRequest);
21         return;
22     }

```

```

23     if(responseResult == null){
24         //对方系统无返回,添加充值请求失败重试的任务
25         rechargeRequest.setErrorCode(StatusCode.ORDER_REQ_FAILED);
26         supplierTask.addRetryTask(rechargeRequest);
27         return;
28     }
29     if(responseResult.getCode() != StatusCode.OK){
30         //充值对接失败---需要按失败原因进行区分,可能需要添加不同的任务
31         if(responseResult.getCode() == StatusCode.ORDER_REQ_FAILED){
32             rechargeRequest.setErrorCode(StatusCode.ORDER_REQ_FAILED);
33             supplierTask.addRetryTask(rechargeRequest);
34             return;
35         }else if(responseResult.getCode() == StatusCode.BALANCE_NOT_ENOUGH){
36             //供应商余额不足,需要轮转到其他供应商
37             //1: 将该供应商的编号存入缓存的排除集合中
38
39             cacheService.sAdd(SupplierConstants.exclude_supplier,rechargeRequest.getSupply());
40             //2:获取下一个可用的供应商
41             String supplier = nextSupplier();
42             //3: 判断是否还有下一个供应商
43             if(supplier == null){
44                 //没有可用的供应商了,对接订单系统及其他系统,进行后续退款等操作
45                 updateTrade(rechargeRequest.getOrderNo(),OrderStatusEnum.FAIL.getCode());
46                 return;
47             }
48             //4: 添加轮转重试任务
49             rechargeRequest.setSupply(supplier);
50             rechargeRequest.setRepeat(0);
51             rechargeRequest.setErrorCode(StatusCode.BALANCE_NOT_ENOUGH);
52             supplierTask.addRetryTask(rechargeRequest);
53             return;
54         }
55     }
56     //充值对接成功
57 }
58
59 private String nextSupplier() {
60     //得到所有的排除供应商集合
61     Set<String> excludes = cacheService.setMembers(SupplierConstants.exclude_supplier);
62     //拿到所有合作的供应商
63     Set<String> allSupplier = supplierProperties.getApis().keySet();
64     //从中找一个,但不能是排除的
65     for (String s : allSupplier) {
66         if(!excludes.contains(s)){
67             return s;
68         }
69     }
70     return null;
71 }

```

2: 在接口 `SupplierTask` 中添加消费轮转任务的方法,并在实现类中实现

```

1  /**
2   * 供应商轮转重试任务
3   */
4  public void roundRecharge();

```

```

1  @Override
2  @Scheduled(fixedRate = 1000)
3  public void roundRecharge() {
4      retry(TaskTypeEnum.BALANCE_NOT_ENOUGH);
5  }

```

3: 与供应商对接前先判断RechargeRequest对象中默认传递过来的供应商是否可用, 如果不可用则自动更换供应商。

```

1  @Override
2  public void recharge(RechargeRequest rechargeRequest) {
3      //限制最大重试次数 如果超过了最大次数, 停止供应商对接,
4      if(rechargeRequest.getRepeat() > supplierProperties.getMaxrepeat()){
5          //订单失败, 订单失败后实际业务中是对接订单服务进行退款等后续业务操作
6          updateTrade(rechargeRequest.getOrderNo(), OrderStatusEnum.FAIL.getCode());
7          return;
8      }
9      //判断默认的供应商是否可用, 如果不可用直接排除更换新的供应商
10     String checkSupply = checkSupply(rechargeRequest.getSupply());
11     if(checkSupply == null){
12         //没有可用供应商, 对接其他系统进行后续业务操作
13         updateTrade(rechargeRequest.getOrderNo(), OrderStatusEnum.FAIL.getCode());
14         return;
15     }
16     rechargeRequest.setSupply(checkSupply);
17
18     //对接逻辑分发
19     Result<RechargeResponse> responseResult;
20     //-----后续代码无变更-----
21
22 }
23 /**
24  * 检查并返回一个可用的供应商
25  * @param supply
26  * @return
27  */
28 private String checkSupply(String supply) {
29     Set<String> excludes = cacheService.setMembers(SupplierConstants.exclude_supplier);
30     if(excludes.contains(supply)){
31         return nextSupplier();
32     }else {
33         return supply;
34     }
35 }

```

测试：重启相关服务，再次充值

预期效果：首先会去对接聚合，因为聚合返回的是余额不足，因此轮转到极速平台，极速平台返回的是充值失败，因此会有重试，达到重试次数的阈值后结束

5.3.2. 供应商恢复

1：如果我们平台在某些供应商处余额不足，我们做了供应商排除，每次对接时都会过滤掉该供应商，一般供应商平台会通知我们进行充值，一旦充值成功，该供应商的接口应该恢复调用，因此我们需要提供一个接口用于恢复供应商，即将该供应商编号从缓存redis中的排除名单中删掉即可。

在 `supplier-business` 中创建一个controller类：

```
com.itheima.supplier.controller.RechargeNotifyController
```

```
1  @RestController
2  @Slf4j
3  public class RechargeNotifyController {
4
5      @Autowired
6      private CacheService cacheService;
7
8      @RequestMapping("/recovery")
9      public String recovery(String supply){
10         Set<String> excludes = cacheService.setMembers(SupplierConstants.exclude_supplier);
11         if(excludes.contains(supply)){
12             cacheService.sRemove(SupplierConstants.exclude_supplier,supply);
13             return "恢复成功!";
14         }else {
15             return "供应商编号不存在!";
16         }
17     }
18 }
```

测试：重启服务再次充值，因为之前测试的时候，聚合平台已经被排除了，所以测试时直接对接极速平台

然后访问：<http://localhost:99/recovery?supply=juheapi>进行恢复，再次充值

5.4. 对接成功后的业务处理

在上一节中对接极速平台我们模拟的是下单失败的情况，现在模拟成功的情况，对接调用成功后我们需要将订单状态改为处理中，一段时间后供应商会回调我们系统，我们需要做的就是更改订单状态为充值成功！

1：模拟对接极速成功的情况，在SupplierServiceImpl类中的方法doPostJisu(RechargeRequest rechargeRequest)中，模拟极速返回成功

```
1  //map.add("req_status", ""+StatusCode.ERROR);
2  map.add("req_status", ""+StatusCode.OK);
```

2：对接成功后我们需要去修改订单的相关状态，需要注意的是：此时订单状态还不能修改为充值成功的状态，只能是修改为等待第三方回调确认的状态

在common模块中有一个跟订单状态相关的枚举

```

1  @Getter
2  public enum OrderStatusEnum {
3
4  CREATE(0, "创建"), WARTING(1, "处理中"), SUCCESS(2, "成功"), FAIL(3, "失败"),
    UNAFFIRM(9, "未确认");
5  private Integer code;
6  private String desc;
7  private OrderStatusEnum(Integer code, String desc) {
8      this.code = code;
9      this.desc = desc;
10 }
11 }

```

```

1  @Override
2  public void recharge(RechargeRequest rechargeRequest) {
3      //限制最大重试次数 如果超过了最大次数, 停止供应商对接,
4      if(rechargeRequest.getRepeat() > supplierProperties.getMaxrepeat()){
5          //订单失败, 订单失败后实际业务中是对接订单服务进行退款等后续业务操作
6          updateTrade(rechargeRequest.getOrderNo(), OrderStatusEnum.FAIL.getCode());
7          return;
8      }
9      //判断默认的供应商是否可用, 如果不可用直接排除更换新的供应商
10     String checkSupply = checkSupply(rechargeRequest.getSupply());
11     if(checkSupply == null){
12         //没有可用供应商, 对接其他系统进行后续业务操作
13         updateTrade(rechargeRequest.getOrderNo(), OrderStatusEnum.FAIL.getCode());
14         return;
15     }
16     rechargeRequest.setSupply(checkSupply);
17
18     //对接逻辑分发
19     Result<RechargeResponse> responseResult;
20     try {
21         responseResult = doDispatchSupplier(rechargeRequest);
22     } catch (Exception e) {
23         log.error("充值对接下单失败, {}, {}", rechargeRequest, e.getMessage());
24         //添加远程调用失败进行重试的任务
25         rechargeRequest.setErrorCode(StatusCode.REMOTEERROR);
26         supplierTask.addRetryTask(rechargeRequest);
27         return;
28     }
29     if(responseResult == null){
30         //对方系统无返回, 添加充值请求失败重试的任务
31         rechargeRequest.setErrorCode(StatusCode.ORDER_REQ_FAILED);
32         supplierTask.addRetryTask(rechargeRequest);
33         return;
34     }
35     if(responseResult.getStatusCode() != StatusCode.OK){
36         //充值对接失败---需要按失败原因进行区分, 可能需要添加不同的任务
37         if(responseResult.getStatusCode() == StatusCode.ORDER_REQ_FAILED){

```

```

38 rechargeRequest.setErrorCode(StatusCode.ORDER_REQ_FAILED);
39 supplierTask.addRetryTask(rechargeRequest);
40 return;
41 }else if(responseResult.getCode() == StatusCode.BALANCE_NOT_ENOUGH){
42     //供应商余额不足,需要轮转到其他供应商
43     //1: 将该供应商的编号存入缓存的排除集合中
44
45     cacheService.sAdd(SupplierConstants.exclude_supplier,rechargeRequest.getSupply());
46     //2:获取下一个可用的供应商
47     String supplier = nextSupplier();
48     //3: 判断是否还有下一个供应商
49     if(supplier == null){
50         //没有可用的供应商了,对接订单系统及其他系统, 进行后续退款等操作
51         updateTrade(rechargeRequest.getOrderNo(),OrderStatusEnum.FAIL.getCode());
52         return;
53     }
54     //4: 添加轮转重试任务
55     rechargeRequest.setSupply(supplier);
56     rechargeRequest.setRepeat(0);
57     rechargeRequest.setErrorCode(StatusCode.BALANCE_NOT_ENOUGH);
58     supplierTask.addRetryTask(rechargeRequest);
59     return;
60 }
61
62 //充值对接成功
63 log.info("下单成功,等待充值处理回调! ");
64 //特别注意此时订单状态还不能修改为充值成功-----供应商回调之后才能修改为成功
65 updateTrade(rechargeRequest.getOrderNo(),OrderStatusEnum.UNAFFIRM.getCode()); //充值处理中
66 等待确认
67 return;
68 }

```

3: 成功后极速平台会进行一个回调, 在mock-thirdparty模块中的MockJisuRechargeController中的add方法进行回调, 回调方法在该模块下的BaseController中: rechargeNotify

回调地址配置在了application.yml文件中, 该回调地址是由我们系统提供的, 实际业务中是本平台在供应商平台进行配置, 回调地址为:

```

1 notify-url: http://127.0.0.1:99/order/notify

```

需要在supplier-business模块的RechargeNotifyController类中补全接收回调的方法, 在实际业务中是调用订单的服务处理订单状态

```

1 @Autowired
2 protected OrderTradeMapper orderTradeMapper;
3
4 @RequestMapping(value = "/order/notify")
5 public String notify(@RequestBody String result) {
6     JSONObject jsonObject = (JSONObject) JSON.parse(result);
7
8     String orderNo= (String) jsonObject.get("orderNo");
9
10    // 这里调用订单服务处理订单状态
11 }

```

```

8      int status= Integer.parseInt(jsonObject.get("status").toString());
9      log.info("充值回调成功修改订单{}的状态为{}",orderNo,status);
10     updateTrade(orderNo, status);
11     return "sucess";
12 }
13
14 private void updateTrade(String orderNo, int orderStatus) {
15     //修改订单状态
16     QueryWrapper<OrderTrade> queryWrapper = new QueryWrapper<>();
17     queryWrapper.eq("order_no", orderNo);
18     OrderTrade orderTrade = orderTradeMapper.selectOne(queryWrapper);
19     if(orderTrade!=null) {
20         orderTrade.setOrderStatus(orderStatus);
21         orderTradeMapper.update(orderTrade, queryWrapper);
22     }
23 }

```

4: 测试

启动所有工程，进行话费充值业务，充值成功后进入订单列表，查看订单状态

5.5.对接状态检查

业务需求：供应商对接下单成功后充吧系统将订单状态更改为：等待确认中，此时等待供应商系统进行回调，当供应商系统回调时说明供应商充值成功，供应商回调充吧系统将充吧的订单改为充值成功，如果在这个过程中出现了故障，比如供应商没有回调，或者回调不通，回调失败，我们应该如何处理这系列问题？

解决方案：

- 1: 供应商会提供一个查询订单充值状态的接口，当我们系统对接下单成功后添加一个查询状态的任务，该任务1分钟或者几分钟后去调用供应商查询状态的API，我们根据供应商返回的结果进行充吧订单状态的修改
- 2: 如果供应商系统能够正常回调，则将查询状态的任务取消即可

5.5.1.状态检查接口开发

- 1: 在consul的 `config/supplier/data` 下添加供应商状态检查接口地址和状态检查任务执行时间，如下

```

1  supplier:
2      jisu_url: "http://127.0.0.1:8090/jisuapi/mobilerecharge"
3      juhe_url: "http://127.0.0.1:8090/juheapi"
4      apis: {jisuapi: "${supplier.jisu_url}/recharge",
5             juheapi: "${supplier.juhe_url}/recharge"}
6      maxrepeat: 4
7      checkStateApis: {jisuapi: "${supplier.jisu_url}/orderState",
8                       juheapi: "${supplier.juhe_url}/orderState"}
9      stateCheckTime: 1 #状态检查时间

```

- 2: 修改对应的属性配置类: `SupplierProperties`

```

1  @Data
2  @Configuration
3  @ConfigurationProperties(prefix = "supplier")
4  public class SupplierProperties {
5      private Map<String,String> apis; //加载供应商api地址,key是第三方供应商编号,value是其地址
6      private int maxrepeat; //最大重试次数
7
8      private Map<String,String> checkStateApis;
9      private int stateCheckTime; //订单充值状态检查时间
10 }

```

3: 在 `SupplierService` 接口中添加检查充值状态的接口方法

```

1  public interface SupplierService {
2
3      /**
4       * 对接供应商充值下单
5       * @param rechargeRequest
6       */
7      public void recharge(RechargeRequest rechargeRequest);
8
9      /**
10     * 对接下单成功后检查充值状态
11     * @param checkStatusRequest
12     */
13     public void checkStatus(CheckStatusRequest checkStatusRequest);
14 }

```

接口参数: CheckStatusRequest

```

1  @Data
2  @NoArgsConstructor
3  @AllArgsConstructor
4  public class CheckStatusRequest {
5
6      private String supplier;
7
8      private String orderNo;
9
10     private String tradeNo;
11 }

```

参数说明:

supplier: 不同供应商查询订单充值状态API地址不一样, 需要根据供应商编号去获取

orderNo: 自己系统生成的订单号, 支付成功后通知对接模块下单时已经将订单号传递过来了

tradeNo: 供应商系统生成的唯一的交易号, 当我们调用供应商对接下单API后由供应商生成的唯一交易号, 并返回给了我们系统

orderNo和tradeNo参数是供应商状态检查接口需要传递的参数！实际的企业开发中看第三方的供应商系统接口需要传递什么参数就传递什么参数。

4: 在对应实现类中去实现

```
1  @Override
2  public void checkStatus(CheckStatusRequest checkStatusRequest) {
3      //获取状态检查接口地址
4      String url =
5      supplierProperties.getCheckStateApis().get(checkStatusRequest.getSupplier());
6      //设置请求头
7      HttpHeaders headers = new HttpHeaders();
8      headers.setContentType(MediaType.APPLICATION_FORM_URLENCODED);
9      //封装请求参数---实际业务中看供应商需要传递哪些参数，实际情况中可能要根据不同的供应商传递不同的
10     //参数，那就要在这个逻辑中添加不同的条件分支
11     MultiValueMap<String,String> map = new LinkedMultiValueMap<>();
12     map.add("outorderNo",checkStatusRequest.getOrderNo());
13     map.add("tradeNo",checkStatusRequest.getTradeNo());
14
15     HttpEntity<MultiValueMap<String,String>> httpEntity = new HttpEntity<>(map,headers);
16     ResponseEntity<String> responseEntity = restTemplate.postForEntity(url, httpEntity,
17     String.class);
18     Result<RechargeResponse> result = JSON.parseObject(responseEntity.getBody(), new
19     TypeReference<Result<RechargeResponse>>() {});
20
21     if(result.getCode() == StatusCode.OK){
22         log.info("订单状态检查,订单成功{}", checkStatusRequest);
23         updateTrade(checkStatusRequest.getOrderNo(),result.getData().getStatus());
24     }else{
25         //订单失败
26         log.info("订单状态检查,订单失败{}", checkStatusRequest);
27         updateTrade(checkStatusRequest.getOrderNo(),OrderStatusEnum.FAIL.getCode());
28     }
29 }
```

5.5.2.添加和消费状态查询任务

有了对接供应商状态查询的接口，下面要做的就是对接供应商下单成功之后添加一个查询状态的任务，该任务可在1分钟后执行，即1分钟后去查询供应商充值状态

该任务为什么不需要立即执行？

因为正常情况下第三方系统是会对我们进行回调的，这个状态检查的任务主要是为了避免第三方不回调或者回调不成功的情况

1: 在 supplier-business 模块的 SupplierTask 接口中新增两个方法，分别用来处理添加状态检查的任务和消费状态检查的任务

```

1  /**
2   * 添加状态检查查询任务
3   * @param checkStatusRequest
4   */
5  public void addCheckStatusTask(CheckStatusRequest checkStatusRequest);
6
7  /**
8   * 消费状态检查任务
9   */
10 public void checkStatus();

```

2: 在对应的实现类中去实现该方法

```

1  @Autowired
2  private SupplierProperties supplierProperties;
3
4  @Override
5  public void addCheckStatusTask(CheckStatusRequest checkStatusRequest) {
6      Task task = new Task();
7      TaskTypeEnum taskTypeEnum = TaskTypeEnum.STATECHECK;
8      task.setTaskType(taskTypeEnum.getTaskType());
9      task.setPriority(taskTypeEnum.getPriority());
10     Calendar calendar = Calendar.getInstance();
11     calendar.add(Calendar.MINUTE, supplierProperties.getStateCheckTime());
12     task.setExecuteTime(calendar.getTimeInMillis());
13
14     task.setParameters(ProtostuffUtil.serialize(checkStatusRequest));
15     //添加状态检查任务
16     ResponseMessage result = taskService.push(task);
17 }
18
19 @Override
20 @Scheduled(fixedRate = 1000)
21 public void checkStatus() {
22     //消费状态检查任务-----发起状态检查
23     TaskTypeEnum statecheck = TaskTypeEnum.STATECHECK;
24     ResponseMessage poll = taskService.poll(statecheck.getTaskType(),
statecheck.getPriority());
25     if(!poll.isFlag() || poll.getCode() == StatusCode.ERROR ){
26         return;
27     }
28     if(poll.getData() != null){
29         String taskStr = JSON.toJSONString(poll.getData());
30         Task task = JSON.parseObject(taskStr, new TypeReference<Task>(){});
31         CheckStatusRequest statusRequest = ProtostuffUtil.deserialize(task.getParameters(),
CheckStatusRequest.class);
32         log.info("消费状态检查任务{}", statusRequest);
33         //调用状态检查接口进行状态检查
34         supplierService.checkStatus(statusRequest);
35     }
36 }

```

3: 对接供应商下单成功后添加状态检查任务, 完善SupplierServiceImpl类中的recharge方法

```
1 @Override
2 public void recharge(RechargeRequest rechargeRequest) {
3     //-----方法前面的代码无变更 故省略-----
4
5     //充值对接成功
6     log.info("下单成功,等待充值处理回调!");
7     //特别注意此时订单状态还不能修改为充值成功-----供应商回调之后才能修改为成功
8     updateTrade(rechargeRequest.getOrderNo(), OrderStatusEnum.UNAFFIRM.getCode()); //充值处理中
    等待确认
9
10    log.info("下单成功, 添加状态检查任务, 1分钟后进行状态检查");
11    supplierTask.addCheckStatusTask(new CheckStatusRequest(rechargeRequest.getSupply(),
12    responseResult.getData().getOrderNo(), responseResult.getData().getTradeNo()));
13    return;
14 }
```

4: 测试

现在要测试供应商回调不成功由充吧系统主动发起状态查询的情形, 在chongba_recharge_mock工程中的MockJisuRechargeController中注释掉回调的方法

```
result.setData(response),
// 充值请求成功
if(reqStatus== StatusCode.OK) {
    result.setCode(StatusCode.OK);
    // 实际充值 结果异步通知
    //rechargeNotify(outorderNo, tradeNo, orderStatus);
    // 模拟供应商订单保存
    response.setStatus(orderStatus);
    cacheService.set(outorderNo+tradeNo, JSON.toJSONString(result));
}else if(reqStatus==StatusCode.ERROR){
    // 充值请求失败
    result.setCode(StatusCode.ORDER_REQ_FAILED);
    result.setMsg("请求充值失败, 请重试");
}
return result;
}
```

启动所有系统, 进行话费充值业务! 查看输出

5.5.3.成功回调后取消任务

1: 当供应商成功回调后我们需要取消状态查询任务, 而取消任务需要任务id, 而在供应商的回调方法中, 只有供应商返回的订单号和交易号, 没有任务id, 因此在添加状态查询任务时需要将订单号和任务id进行关联存储,

在SupplierTaskImpl类中完善: addCheckStatusTask; 添加状态检查任务后将订单编号和任务id进行映射存储, 使用redis的hash机构

```
1 @Override
2 public void addCheckStatusTask(CheckStatusRequest checkStatusRequest) {
```

```

3      Task task = new Task();
4      TaskTypeEnum taskTypeEnum = TaskTypeEnum.STATECHECK;
5      task.setTaskType(taskTypeEnum.getTaskType());
6      task.setPriority(taskTypeEnum.getPriority());
7      Calendar calendar = Calendar.getInstance();
8      calendar.add(Calendar.MINUTE, supplierProperties.getStateCheckTime());
9      task.setExecuteTime(calendar.getTimeInMillis());
10
11     task.setParameters(ProtostuffUtil.serialize(checkStatusRequest));
12     //添加状态检查任务
13     ResponseMessage result = taskService.push(task);
14
15     //添加完成后将订单号和任务id做一个关联存储
16     if(result.getCode() == StatusCode.OK){
17
18         cacheService.hPut(SupplierConstants.order_checked, checkStatusRequest.getOrderNo(), String.valueOf(result.getData()));
19     }

```

与此同时，如果该任务被消费后，需要从缓存中删除该任务的id和订单编号的映射，修改消费状态检查任务的方法：checkStatus

```

1  @Override
2  @Scheduled(fixedRate = 1000)
3  public void checkStatus() {
4      //消费状态检查任务-----发起状态检查
5      TaskTypeEnum statecheck = TaskTypeEnum.STATECHECK;
6      ResponseMessage poll = taskService.poll(statecheck.getTaskType(),
7      statecheck.getPriority());
8      if(!poll.isFlag() || poll.getCode() == StatusCode.ERROR ){
9          return;
10     }
11     if(poll.getData() != null){
12         String taskStr = JSON.toJSONString(poll.getData());
13         Task task = JSON.parseObject(taskStr, new TypeReference<Task>(){});
14         CheckStatusRequest statusRequest = ProtostuffUtil.deserialize(task.getParameters(),
15         CheckStatusRequest.class);
16         log.info("消费状态检查任务{}", statusRequest);
17
18         //消费后将订单号和任务id的管理映射删除,避免缓存中多一些无用的数据
19         cacheService.hDelete(SupplierConstants.order_checked, statusRequest.getOrderNo());
20
21         //调用状态检查接口进行状态检查
22         supplierService.checkStatus(statusRequest);
23     }
24 }

```

2: 在SupplierTask中添加一个取消状态检查任务的方法，方法参数是订单号

```

1  /**
2      * 取消状态检查任务
3      * @param orderNo
4      */
5  public void cancelCheckTask(String orderNo);

```

3: 在SupplierTaskImpl类中实现取消状态检查任务的方法

```

1  @Override
2  public void cancelCheckTask(String orderNo) {
3      String taskId = (String) cacheService.hGet(SupplierConstants.order_checked, orderNo);
4      if(taskId!=null){
5          taskService.cancel(Long.valueOf(taskId));
6          cacheService.hDelete(SupplierConstants.order_checked,orderNo);
7      }
8  }

```

4: 在供应商回调成功的方法中调用取消状态检查的任务, 在RechargeNotifyController中的notify(@RequestBody String result)方法, 调用取消状态检查方法

```

1  @Autowired
2  private SupplierTask supplierTask;
3  @RequestMapping(value = "/order/notify")
4  public String notify(@RequestBody String result) {
5      JSONObject jsonObject = (JSONObject) JSON.parse(result);
6      String orderNo= (String) jsonObject.get("orderNo");
7      int status= Integer.parseInt(jsonObject.get("status").toString());
8      log.info("充值回调成功修改订单{}的状态为{}",orderNo,status);
9      updateTrade(orderNo, status);
10
11      log.info("回调成功后取消状态检查任务");
12      supplierTask.cancelCheckTask(orderNo);
13      return "success";
14  }

```

5: 测试,

注意: 在mock-thirdparty模块中将极速的回调通知方法放开, 然后再测试,

启动所有工程, 进行话费充值, 查看控制台输出!