

## 12306 深度优化

# 1 课程介绍

## 1.1 课程背景介绍

每年的春节是中国人的传统节日，大多数中国人都会在这一天选择回家团聚。为了方便用户进行购票，2012 年春节，铁道部推出 12306 网站，进行网络实名购票。然而，历年春节假期，巨大的访问请求都让中国铁路客户服务中心网站（www.12306.cn）陷入“万劫不复”。根据新浪的调查，在 2013 年春节，有近 90%的网友表示 12306 网站缓慢、页面崩溃，严重影响正常购票。

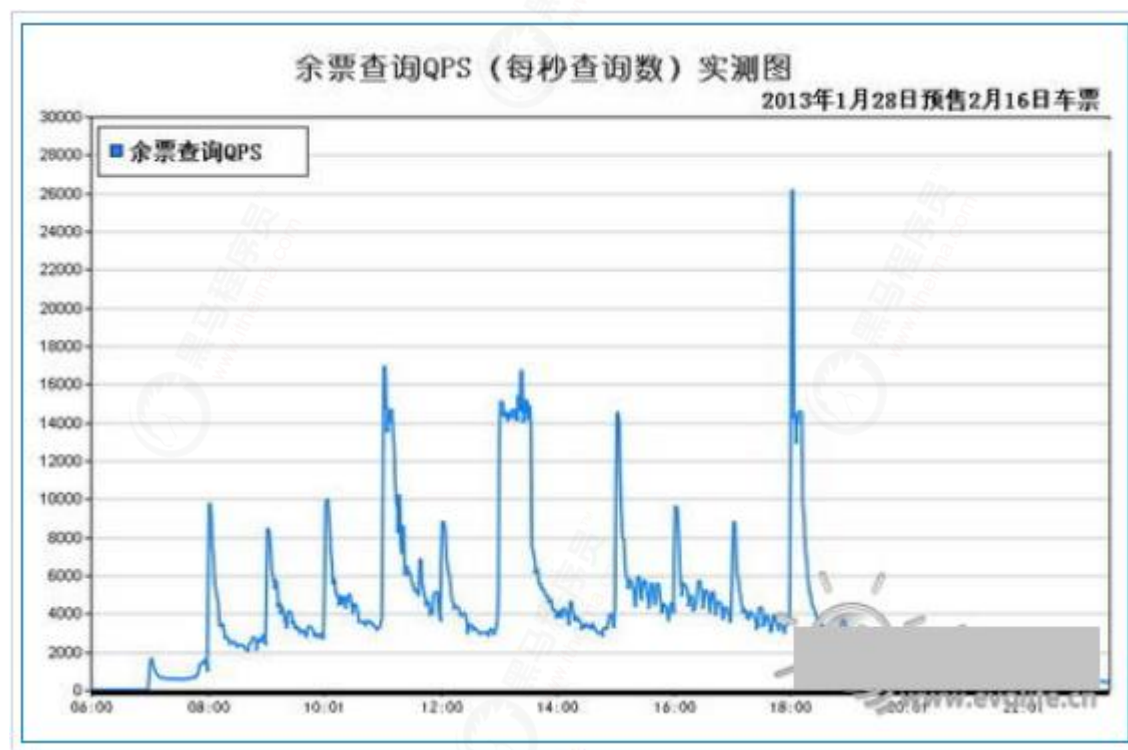
世界级的人口迁徙带来了一个世界级的难题：**要通过网络，把火车票及时卖给有需要的人？**

12306 网站所面临的问题分析：

铁道部在线车票发售网站 12306 基本不存在大量图片、视频这些占带宽资源的东西，所面临的主要问题就是数据库的高并发量——用中国的人口基数来算，这是一个极为恐怖的并发量，在车票发售的高峰时间点，向 12306 发起的并发请求数量大得就像一场国家规模的 DDOS 攻击。

中国铁路客户服务中心网站（www.12306.cn）是世界规模最大的实时交易系统之一，媲美 Amazon.com，节假日尤其是春节的访问高峰，网站压力巨大。据统计，在 2012 年初的春运高峰期间，每天有 2000 万人访问该网站，日点击量最高达到 14 亿。

而到了 2013 年春节期间，12306 的网站订票系统系统峰值负载达 2.6 万 QPS（每秒钟 2.6 万次访问请求）或 11 万 TPS（TPS 指每秒服务器处理、传输的事物处理个数，一条消息输入、一条消息输出、一次数据库访问，均可以折算成 TPS），与 2012 淘宝双 11 活动峰值负载基本相当。



所以 12306 所面临的难题本质上也是属于**高并发**访问问题，类似与一些电商网站所搞的“秒杀”活动一样。通过对 12306 的深度优化，2015 年 12306 网站顺利过关，没有“瘫痪”，是值得庆祝的。而我们本次课题主要就是来探究一下如何对 12306 网站做深度优化来抵御高并发访问。

## 1.2 高并发访问

那么 12306 做了什么样的优化，才解决了高并发访问呢？

12306 技术部主任单杏花在接受一次记者采访的时候有说到：我们研发了**分布式的内存计算的余票计算技术**，让余票计算变得非常高效。与此同时单杏花及其团队还研发了**异步交易排队系统**，这种系统采用**售取分离、读写分离**的核心系统架构等多种技术，为 12306 售票系统提供技术支撑。

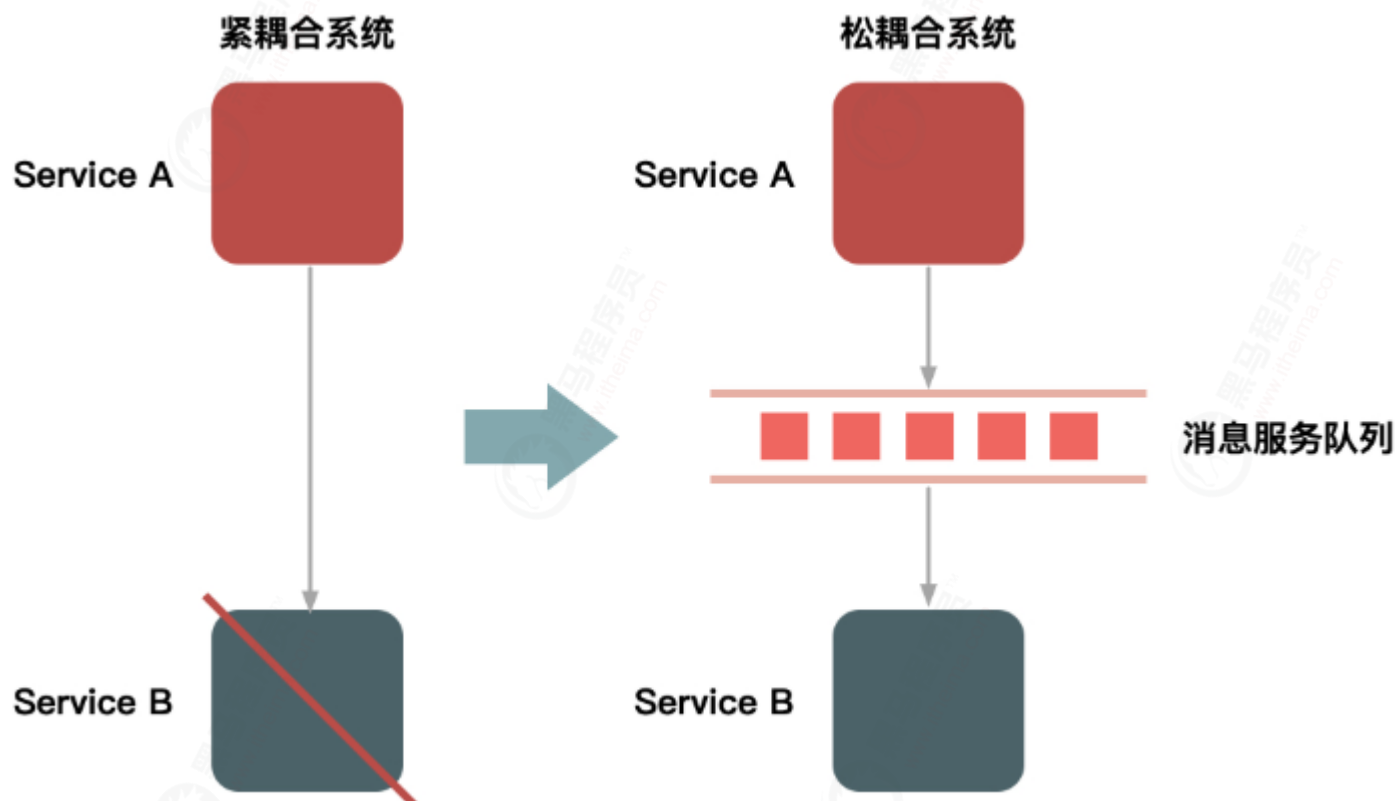
其实通过她的描述，我们可以得出一些处理高并发访问方式：

- 1、采用内存计算（使用缓存系统）
- 2、异步处理请求（进行流量错峰）
- 3、数据库进行读写分离操作

常见的分布式缓存系统：MongoDB ， **Redis**， MemCache

### 流量错峰的方案

1、要对流量进行削峰，最容易想到的解决方案就是**用消息队列来缓冲瞬时流量**，把同步的直接调用转换成异步的间接推送，中间通过一个队列在一端承接瞬时的流量洪峰，在另一端平滑地将消息推送出去。在这里，消息队列就像“水库”一样，拦蓄上游的洪水，削减进入下游河道的洪峰流量，从而达到减免洪水灾害的目的。



## 2、进行答题

你是否还记得，最早期的秒杀只是纯粹地刷新页面和点击购买按钮，它是后来才增加了答题功能的。那么，为什么要增加答题功能呢？这主要是为了增加购买的复杂度，从而达到两个目的。

第一个目的是防止部分买家使用秒杀器在参加秒杀时作弊。2011 年秒杀非常火的时候，秒杀器也比较猖獗，因而没有达到全民参与和营销的目的，所以系统增加了答题来限制秒杀器。增加答题后，下单的时间基本控制在 2s 后，秒杀器的下单比例也大大下降。答题页面如下图所示。



第二个目的其实就是延缓请求，起到对请求流量进行削峰的作用，从而让系统能够更好地支持瞬时的流量高峰。这个重要的功能就是把峰值的下单请求拉长，从以前的 1s 之内延长到 2s~10s。

这样一来，请求峰值基于时间分片了。这个时间的分片对服务端处理并发非常重要，会大大减轻压力。而且，由于请求具有先后顺序，靠后的请求到来时自然也就没有库存了，因此根本到不了最后的下单步骤，所以真正的并发写就非常有限了。

## 3、分时间段进行产品上架处理

其实处理高并发访问还有两种常见手段：**静态化**、**集群**

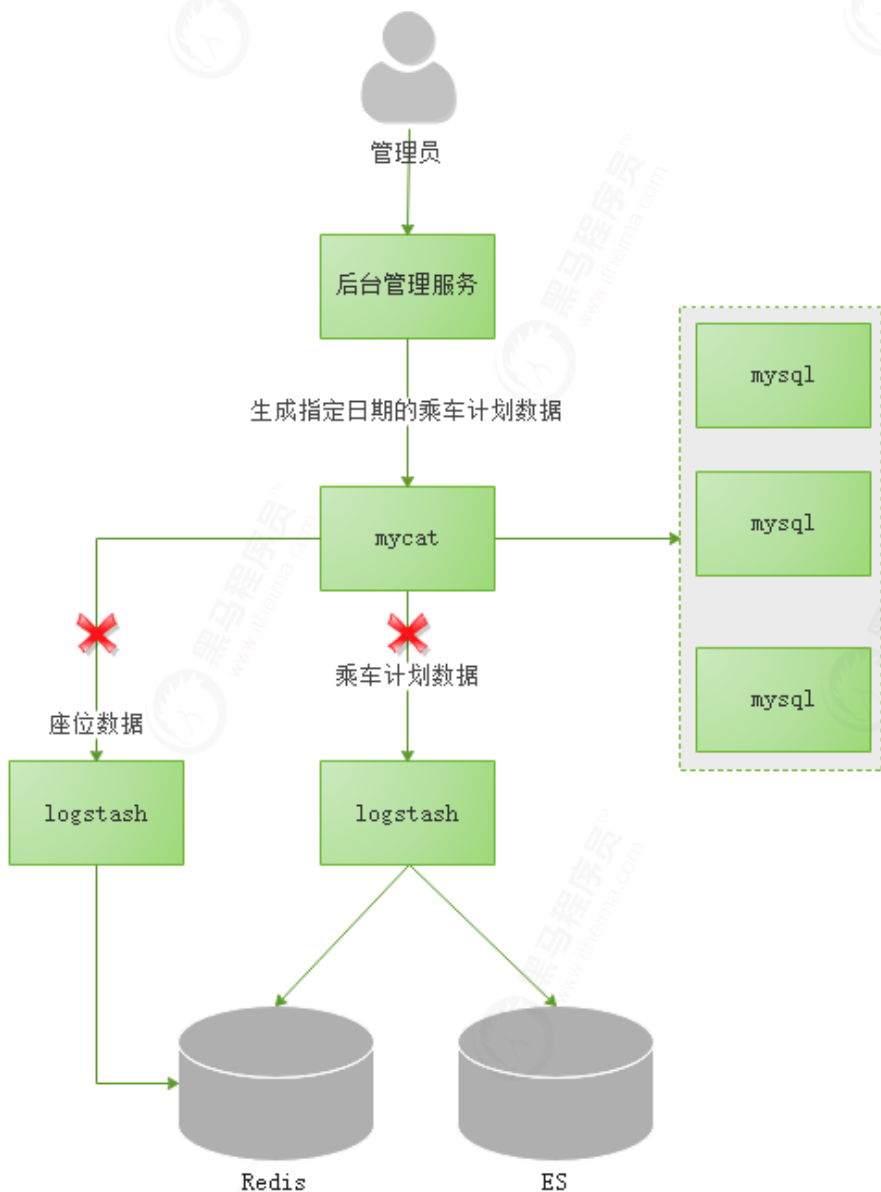
**静态化**：分布式缓存是为了解决数据库服务器和 Web 服务器之间的瓶颈，如果一个网站流量很大这个瓶颈将会非常明显，每次数据库查询耗费的时间将不容乐观。对于更新速度不是很快的站点，可以采用静态化来避免过多的数据查询，可使用 Freemake 或 Velocity 来实现页面静态化。

**集群**：使用多台服务器去处理并发请求

## 1.3 系统架构介绍

基于以上几点高并发的处理方案，我们本次所设计的 12306 后端系统架构如下所示。

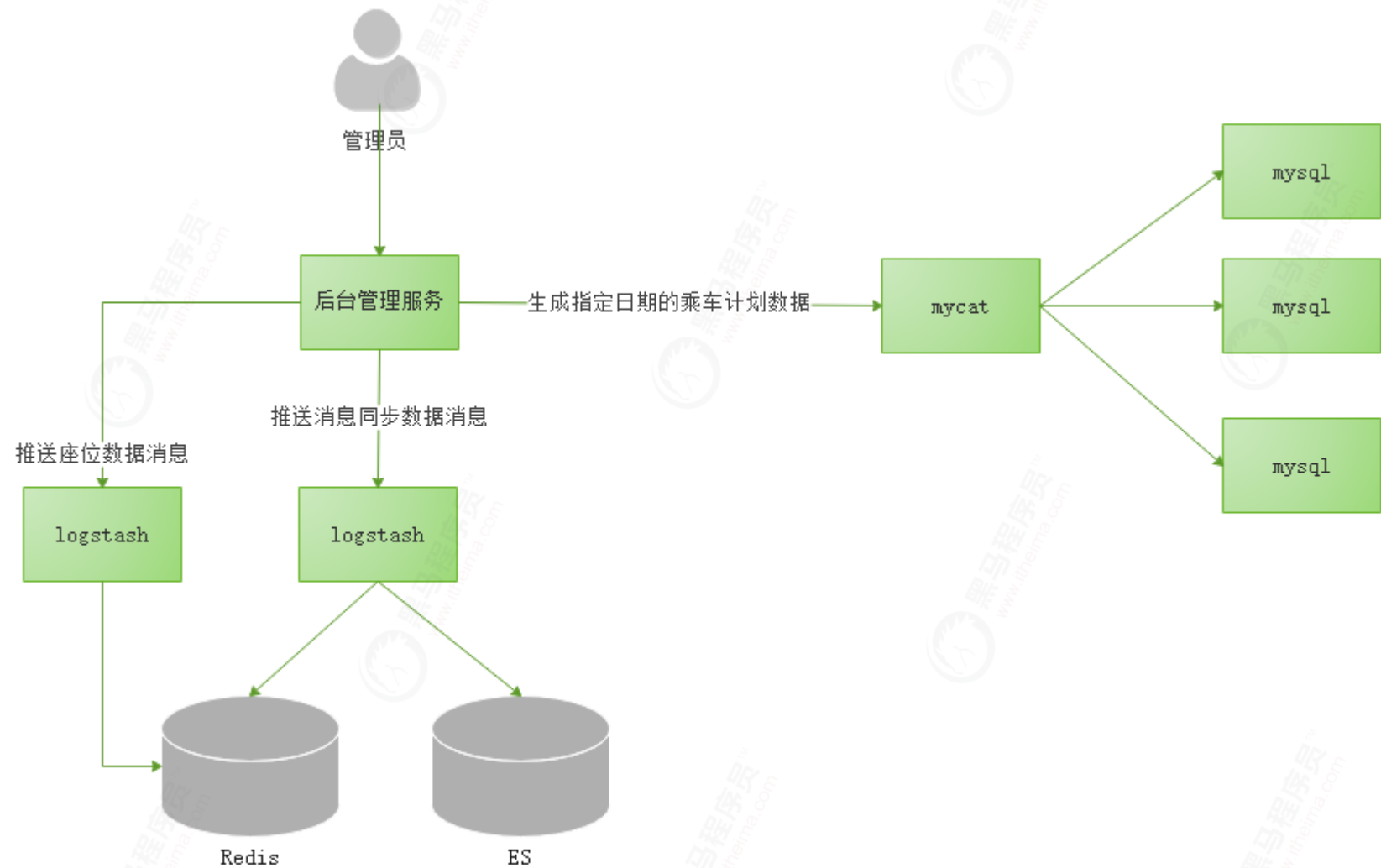
1.3.1 数据同步架构



系统管理员通过后台管理系统基于一些基础数据（座位数据，列车车次数据，乘车计划数据）生成指定日期的乘车计划数据。然后我们通过 logstash 将生成的数据同步到 ES 和 Redis 中。

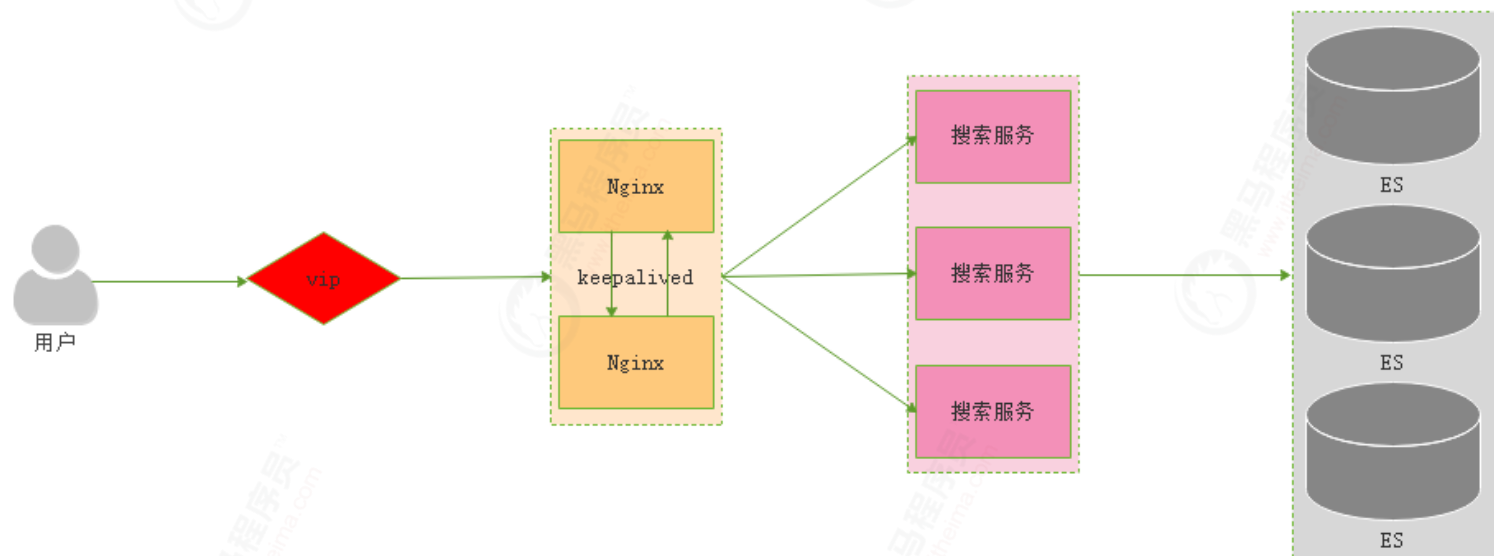
Logstash 常见的数据获取方式拉，推。上述架构给大家展示的是拉的模式，但是这种方式我们当前这个系统环境中不太适合，原因是因为我们使用了 MyCat 进行分库分表的处理，而 Logstash 在进行拉取数据的时候如果数据量较大我们就需要进行分页拉取，那么此时 Logstash 就会生成类似这样的一条 sql 语句：select count(\*) as `count` from ....来查询满足条件总条数，但是这个 count 别名使用了反引号，而这个反引号在 MyCat 中无法使用，因此就会产生异常。

因此本次我们在进行数据同步的时候使用的是 Logstash 的推模式进行数据同步，如下所示：



### 1.3.2 数据搜索架构

数据同步完毕以后，用户就可以搜索相关的乘车计划数据了。具体的搜索架构如下所示：

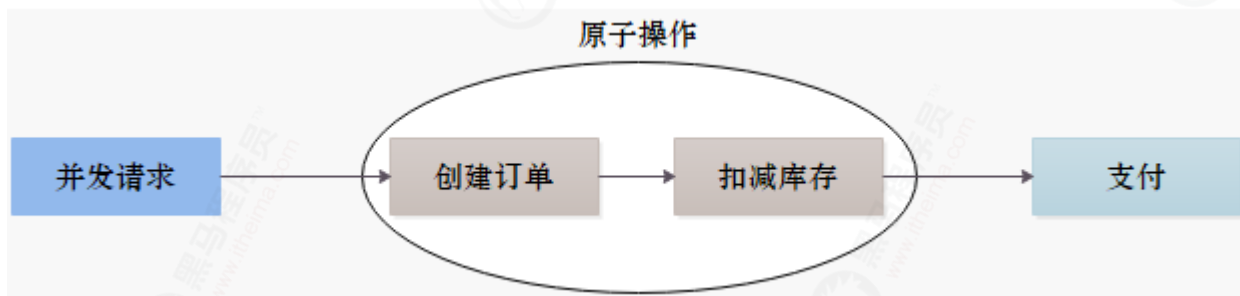


### 1.3.3 用户下单架构

通常订票系统要处理**生成订单**、**减扣库存**、**用户支付**这三个基本的阶段，我们系统要做的事情是要保证火车票订单不超卖、不少卖，每张售卖的车票都必须支付才有效，还要保证系统承受极高的并发。

这三个阶段的先后顺序该怎么分配才更加合理呢？

#### 方案一

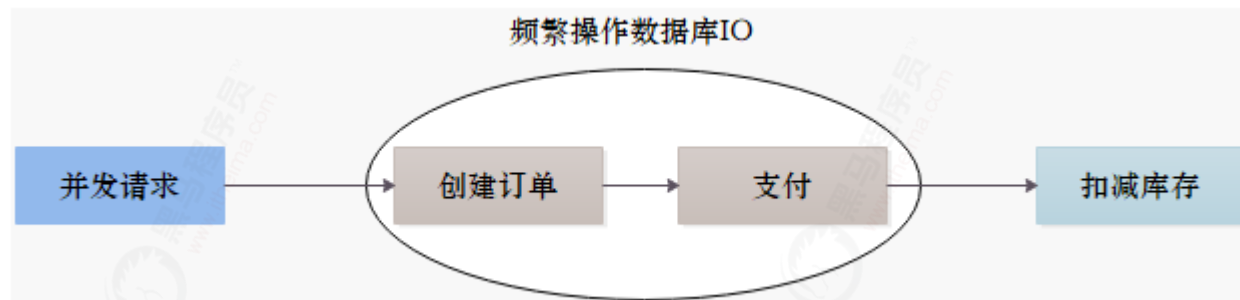


当用户并发请求到达服务端时，首先创建订单，然后扣除库存，等待用户支付。这种顺序是我们一般人首先会想到的解决方案，这种情况下也能保证订单不会超卖，因为创建订单之后就会减库存，这是一个原子操作。

#### 存在的问题：

- 1、就是在极限并发情况下，任何一个内存操作的细节都至关影响性能，尤其像创建订单这种逻辑，一般都需要存储到磁盘数据库的，对数据库的压力是可想而知的；
- 2、存在如果用户存在恶意下单的情况，只下单不支付这样库存就会变少，会少卖很多订单。

#### 方案二

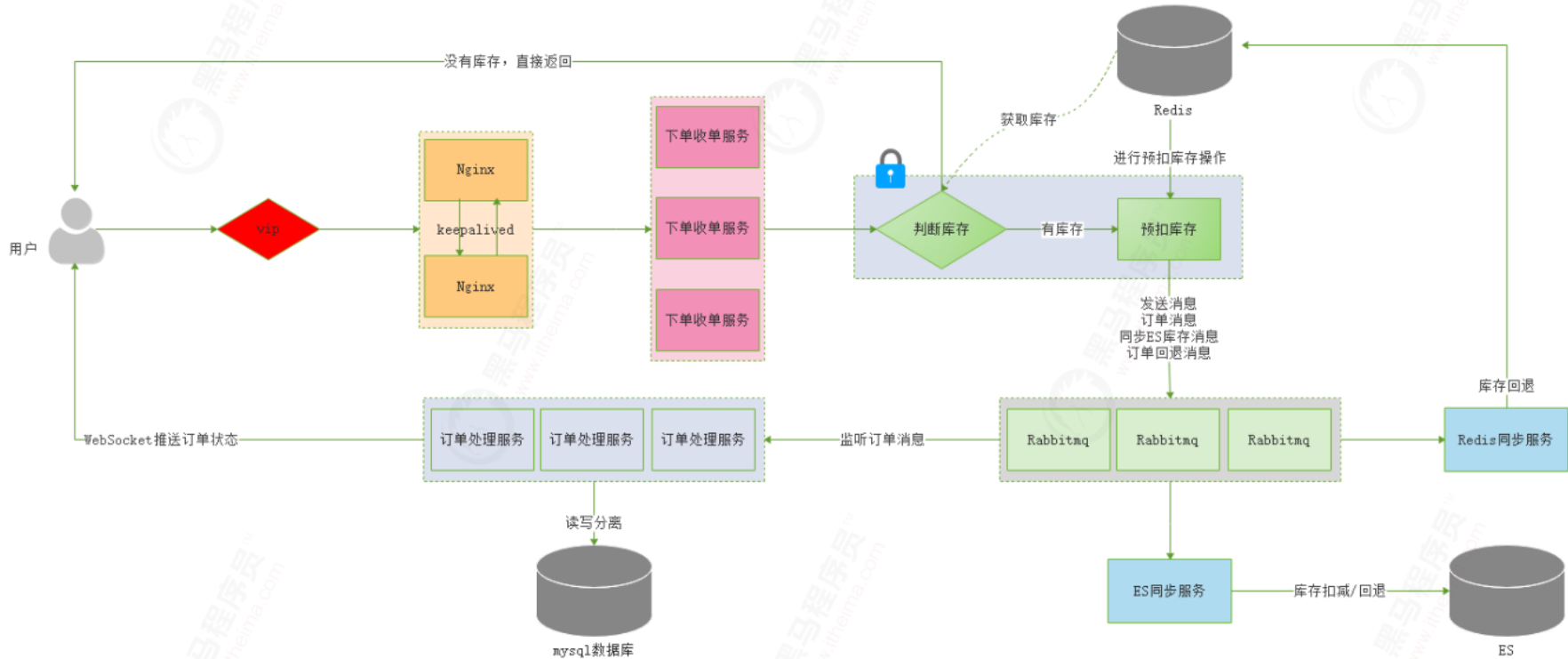


如果等待用户支付了订单在减库存，第一感觉就是不会少卖。但是这是并发架构的大忌，因为在极限并发情况下，用户可能会创建很多订单，当库存减为零的时候很多用户发现抢到的订单支付不了了，这也就是所谓的**超卖**，并且这种方案也不能避免并发操作数据库磁盘 IO。

#### 方案三

从上边两种方案的考虑，我们可以得出结论：**只要创建订单，就要频繁操作数据库 IO**。那么有没有一种不需要直接操作数据库 IO 的方案呢，这就是**预扣库存**。先扣除了库存，保证不超卖，然后**异步生成用户订单**，这样响应给用户的速度就会快很多；

那么怎么保证不少卖呢？用户拿到了订单，不支付怎么办？我们都知道现在订单都有有效期，比如说用户五分钟内不支付，订单就失效了，订单一旦失效，就会加入新的库存，这也是现在很多网上零售企业保证商品不少卖采用的方案。订单的生成是异步的,一般都会放到 MQ（消费队列）中处理,订单量比较少的情况下，生成订单非常快，用户几乎不用排队。如下图所示：



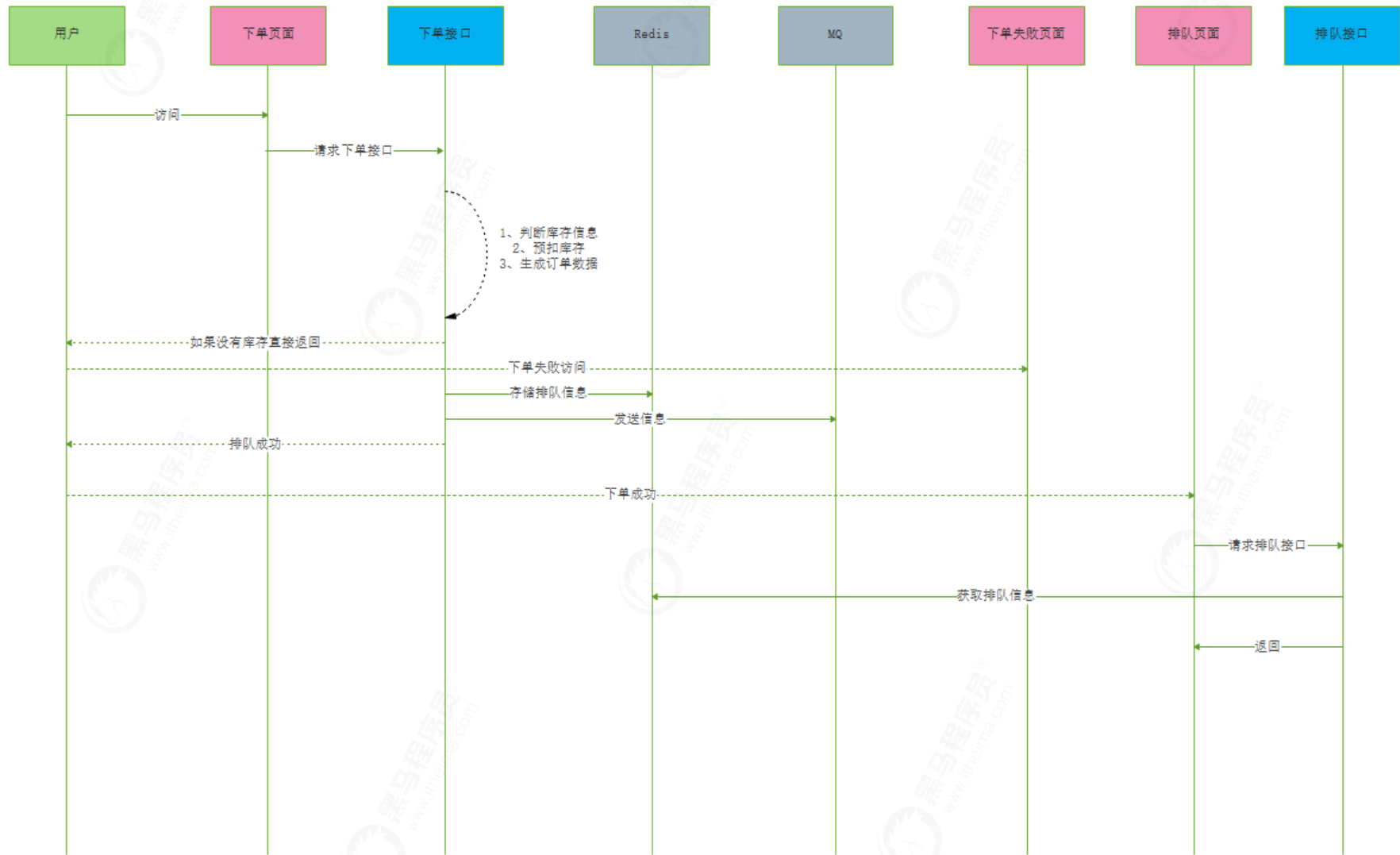
这种方案也就是单杏花主任所提出的异步交易排队系统。当然 12306 网站的还有一个改造的关键技术**建立可伸缩扩展的云应用平台**。根据互联网上的新闻，中国铁道科学研究院电子计算技术研究所副所长，12306 网站技术负责人朱建生说，为了应对 2015 年春运售票高峰，该网站采取 5 项措施：

- 一、利用外部云计算资源分担系统查询业务，可根据高峰期业务量的增长按需及时扩充。
- 二、对系统的互联网接入带宽进行扩容，并可根据流量情况快速调整，保证高峰时段旅客顺畅访问网站。
- 三、防范恶意抢票，通过技术手段屏蔽抢票软件产生的恶意流量，保证网站健康运行，维护互联网售票秩序。
- 四、制定了多套应急预案，以应对突发情况。

## 2 用户下单

### 2.1 下单流程

用户下单排队的具体流程如下所示：



### 2.2 下单页面

注意：下单之前需要初始化一些用户数据到 Redis 中(train\_manager:userInfo)，默认用户已经登录  
下单页面：12306\otm\confirmPassenger\initDc.html



## 2.3 订单服务实现

### 2.3.1 下单接口定义

#### 实体类创建

OrderParams（接收下单请求参数的实体类）

```
@Data // 通过 lombok 生成 getter 和 setter 方法
@NoArgsConstructor // 通过 lombok 生成无参构造方法
@ToString // 通过 lombok 生成 toString 方法
@EqualsAndHashCode // 通过 lombok 生成 hashCode 和 equals 方法
public class OrderParams {

    private String ticketType ; // 座位类型
    private Long ridingPlanId ;
    private String trainNum ;
    private String trainRidingDate ;
    private String userId ;

}
```

Order（订单实体类）

```
@Data // 通过 lombok 生成 getter 和 setter 方法
@NoArgsConstructor // 通过 lombok 生成无参构造方法
@ToString // 通过 lombok 生成 toString 方法
@EqualsAndHashCode // 通过 lombok 生成 hashCode 和 equals 方法
public class Order {

    private Long id ;
    private Long ridingPlanDateId ;
    private String startStationName ;
    private String receiveStationName ;
    private String trainNum ;
    private String trainRidingDate ;
    private String startTime ;
    private String endTime ;
    private String seatNum ;
    private String seatNature ; // 座位性质
    private String coachNum ; // 车厢号
    private Double seatPrice ;
    private String payStatus ; // 支付状态，0 表示未支付，1 表示支付
    private String userId ;
    private String isEffect ; // 是否有效：1 表示有效，0 表示无效

}
```

ResponseResult（用于封装下单结果）

```
@Data // 通过 lombok 生成 getter 和 setter 方法
@NoArgsConstructor // 通过 lombok 生成无参构造方法
@ToString // 通过 lombok 生成 toString 方法
@EqualsAndHashCode // 通过 lombok 生成 hashCode 和 equals 方法
public class ResponseResult<T> {

    private boolean result ;
    private String message ;
    private T data ;

}
```

#### 订单工程环境搭建

- 1、导入下单工程（itheima-train-manager-order）
- 2、在nacos 配置中心中添加配置项（itheima-train-manager-order-dev.yml）

```
spring:
  application:
    name: itheima-train-manager-order
  cloud:
    nacos:
      config:
        server-addr: 127.0.0.1:8848,127.0.0.1:8849,127.0.0.1:8850
        file-extension: yaml
  profiles:
    active: dev
```



## 下单接口定义

```
@RestController
@RequestMapping(value = "/order")
public class OrderController {

    @Autowired
    private OrderService orderService ;

    @RequestMapping(value = "/submitOrder")
    public ResponseEntity<String> submitOrder(@RequestBody OrderParams orderParams) {
        return orderService.submitOrder(orderParams) ;
    }
}
```

## 2.3.2 Nginx 配置

### 反向代理配置

```
# 下单服务的代理转发地址

upstream train-manager-order {

    server 127.0.0.1:9056 ;

}

# 配置下单工程的反向代理

server {

    listen      80;

    server_name www.trainmanager.order.com;

    access_log  logs/host.access.order.log main ;           # nginx 访问日志

    location / {

        proxy_pass http://train-manager-order ;

    }

    error_page  500 502 503 504  /50x.html;

    location = /50x.html {

        root    html;

    }

}
```

在 hosts 文件中配置 www.trainmanager.order.com 域名映射

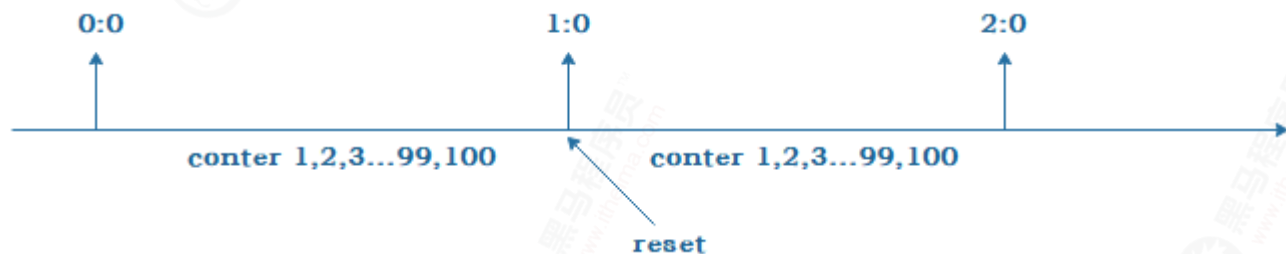
### Nginx 限流配置

为了防止一些抢票助手所发起的一些无用请求，我们可以使用 nginx 中的限流策略进行限流操作。

常见的限流算法：计数器、漏桶算法、令牌桶算法

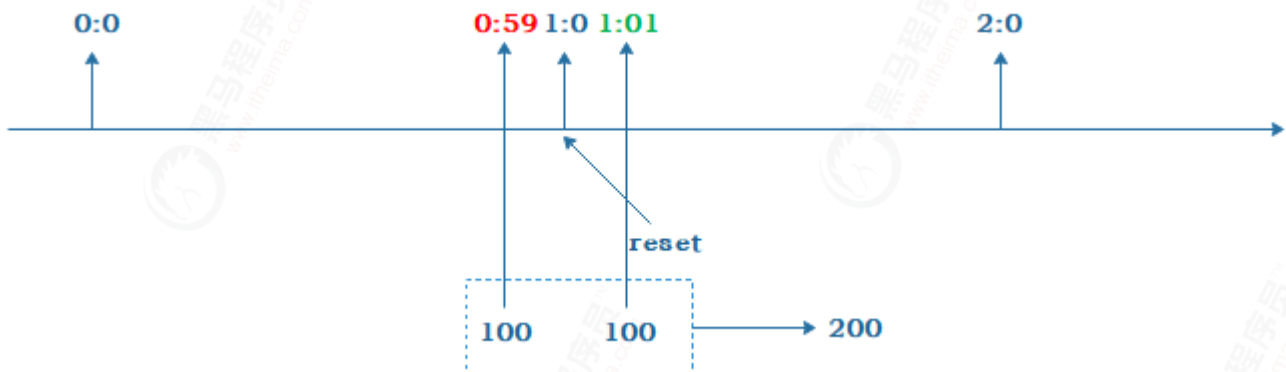
### 计数器限流算法

计数器法是限流算法里最简单也是最容易实现的一种算法。比如我们规定，对于 A 接口来说，我们 1 分钟的访问次数不能超过 100 个。那么我们可以设置一个计数器 counter，其有效时间为 1 分钟（即每分钟计数器会被重置为 0），每当一个请求过来的时候，counter 就加 1，如果 counter 的值大于 100，就说明请求数过多；如下图所示：



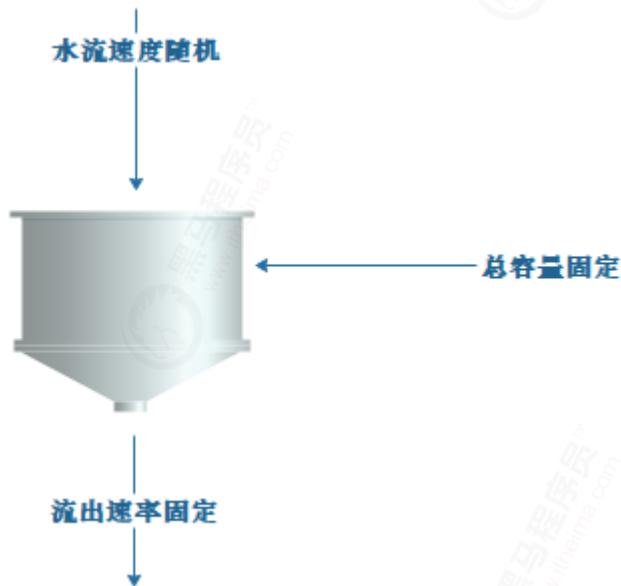
这个算法虽然简单，但是有一个十分致命的问题，那就是临界问题。

如下图所示，在 1:00 前一刻到达 100 个请求，1:00 计数器被重置，1:00 后一刻又到达 100 个请求，显然计数器不会超过 100，所有请求都不会被拦截；然而这一时间段内请求数已经达到 200，远超 100。



### 漏桶算法

漏桶算法其实很简单，可以粗略的认为就是注水漏水过程，往桶中以一定速率流出水，以任意速率流入水，当水超过桶流量则丢弃，因为桶容量是不变的，保证了整体的速率。如下图所示：



### 令牌桶算法

令牌桶是一个存放固定容量令牌的桶，按照固定速率  $r$  往桶里添加令牌；桶中最多存放  $b$  个令牌，当桶满时，新添加的令牌被丢弃；当一个请求达到时，会尝试从桶中获取令牌；如果有，则继续处理请求；如果没有则排队等待或者直接丢弃；可以发现，漏桶算法的流出速率恒定，而令牌桶算法的流出速率却有可能大于  $r$ ；





从作用上来说，漏桶和令牌桶算法最明显的区别就是是否允许突发流量(burst)的处理，漏桶算法能够强行限制数据的实时传输（处理）速率，对突发流量不做额外处理；而令牌桶算法能够在限制数据的平均传输速率的同时允许某种程度的突发传输。

## Nginx 的限流策略

Nginx 的限流主要是两种方式：**限制访问频率**和**限制并发连接数**。

Nginx 按请求速率限速模块使用的是漏桶算法，即能够强行保证请求的实时处理速度不会超过设置的阈值。

Nginx 官方版本限制 IP 的连接和并发分别有两个模块：

- 1、**limit\_req\_zone**：用来限制单位时间内的请求数，即速率限制，采用的漏桶算法 "leaky bucket"。
- 2、**limit\_conn\_zone**：用来限制同一时间连接数，即并发限制。

### limit\_req\_zone

使用语法：**limit\_req\_zone key zone rate**

**key**：定义限流对象，binary\_remote\_addr 是一种 key，表示基于 remote\_addr(客户端 IP) 来做限流，binary\_ 的目的是压缩内存占用量。

**zone**：定义共享内存区来存储访问信息，myRateLimit:10m 表示一个大小为 10M，名字为 myRateLimit 的内存区域。1M 能存储 16000 IP 地址的访问信息，10M 可以存储 16W IP 地址访问信息。

**rate**：用于设置最大访问速率，rate=10r/s 表示每秒最多处理 10 个请求。Nginx 实际上以毫秒为粒度来跟踪请求信息，因此 10r/s 实际上是限制：每 100 毫秒处理一个请求。这意味着，自上一个请求处理完后，若后续 100 毫秒内又有请求到达，将拒绝处理该请求。

举例：

```
http {  
    # 定义限流策略  
  
    limit_req_zone $binary_remote_addr zone=rateLimit:10m rate=1r/s ;  
  
    # 搜索服务的虚拟主机  
  
    server {  
        location / {  
            # 使用限流策略，burst=5，重点说明一下这个配置，burst 爆发的意思，这个配置的意思是设置一个大小为 5 的缓冲区（队列）  
            # 当有大量请求（爆发）过来时，  
  
            # 超过了访问频次限制的请求可以先放到这个缓冲区内。nodelay，如果设置，超过访问频次而且缓冲区也满了的时候就会直接返回 503，如果没有设置，则所  
  
            # 有请求会等待排队。  
  
            limit_req zone=rateLimit burst=5 nodelay;  
  
            proxy_pass http://train-manager-search ;  
        }  
    }  
}
```

### limit\_conn\_zone

使用语法：**limit\_conn\_zone key zone**

**key**：定义限流对象，binary\_remote\_addr 是一种 key，表示基于 remote\_addr(客户端 IP) 来做限流，binary\_ 的目的是压缩内存占用量。

**zone**：定义共享内存区来存储访问信息，myRateLimit:10m 表示一个大小为 10M，名字为 myRateLimit 的内存区域。1M 能存储 16000 IP 地址的访问信息，10M 可以存储 16W IP 地址访问信息。

举例：

```
http {  
    # 定义限流策略  
  
    limit_conn_zone $binary_remote_addr zone=perip:10m;  
  
    limit_conn_zone $server_name zone=perserver:10m;  
  
    # 搜索服务的虚拟主机  
  
    server {  
        location / {  
            # 对应的 key 是 $binary_remote_addr，表示限制单个 IP 同时最多能持有 1 个连接。  
  
            limit_conn perip 1;  
        }  
    }  
}
```



# 对应的 key 是 \$server name，表示虚拟主机(server) 同时能处理并发连接的总数。注意，只有当 request header 被后端 server 处理后，这个连接才进行计数。

```
limit_conn perserver 10 ;

proxy_pass http://train-manager-search ;

}

}
```

## JMeter 测试工具

为了模拟并发访问，我们可以使用一下 JMeter 这个压测工具。

### JMeter 工具下载

我们可以登录 JMeter 的官网来下载 Jmeter: [http://jmeter.apache.org/download\\_jmeter.cgi](http://jmeter.apache.org/download_jmeter.cgi)

## Binaries

[apache-jmeter-5.2.1.tgz sha512 pgp](#)

[apache-jmeter-5.2.1.zip sha512 pgp](#)

### JMeter 使用

- 1、启动 JMeter：执行 bin 目录下的 jmeter.bat
- 2、在测试计划中添加一个线程组
- 3、在指定的线程组中创建一个 Http 请求，其中线程的数量表示并发的数据量
- 4、如果需要发送 post 请求，并且传递 json 数据。那么我们还需要给 http 请求添加一个配置单元(http 信息头管理器)，指定请求数据的格式是 json(Content-Type: application/json)

测试数据:

```
{

  "ridingPlanId":264546196624769024,

  "ticketType":"secondSeat",

  "trainNum":"G26",

  "trainRidingDate":"2020-06-01",

  "userId":"fb71889cbfbf4857a32f701f175fe256"

}
```

- 5、添加请求监听器：查看结果树
- 6、执行测试，查看 Nginx 的日志记录

### 测试方案

方案一：测试 limit\_req\_zone,注释掉其他的两个限流方式

方案二：测试 limit\_conn\_zone，注释请求次数限流策略

## 2.3.3 生成订单

### 2.3.3.1 预扣库存

```
// 依赖注入
@Autowired
private StringRedisTemplate redisTemplate ;

// 下单操作
public ResponseResult<String> submitOrder(OrderParams orderParams) {

    // 定义 ResponseResult 返回对象
    ResponseResult<String> responseResult = new ResponseResult<>() ;

    // 获取相关参数
    String trainNum = orderParams.getTrainNum() ;
    String trainRidingDate = orderParams.getTrainRidingDate();
```



```
Long ridingPlanId = orderParams.getRidingPlanId();
String ticketType = orderParams.getTicketType(); // 座位的性质
String redisticketTypeKey = ticketType + "Stock"; // busSeat busSeatStock

// 构建乘车计划的 key
// riding plan:G26:2020-06-01:270101019289976832
LOGGER.info("【预扣库存开始了】 ---> " + JSON.toJSONString(orderParams));
String ridingPlanDateKey = "riding_plan:" + trainNum + ":" + trainRidingDate + ":" + ridingPlanId ;
int count = Integer.parseInt(redisTemplate.boundHashOps(ridingPlanDateKey).get(redisticketTypeKey).toString());
if(count <= 0 ) {
    responseResult.setResult(false);
    return responseResult ;
}

// 动态库存扣减
String stockCountKeyDynamic = "riding_plan:" + trainNum + ":" + trainRidingDate + ":*" ;
Set<String> keys = redisTemplate.keys(stockCountKeyDynamic);
for(String key : keys) {
    redisTemplate.boundHashOps(key).put(redisticketTypeKey , String.valueOf(count - 1));
}
LOGGER.info("【预扣库存结束了】 ---> " + JSON.toJSONString(orderParams));

return null ;
}
```

### 2.3.3.2 分配座位

具体的实现步骤如下所示：

- 1、获取所有的指定座位类型的所有车厢的 key（集合）
- 2、遍历集合获取每一个车厢数据（集合）
- 3、遍历集合获取每一个座位所对应的状态值
- 4、状态值如果是 0，记录座位标号
- 5、更改座位状态值

具体的代码实现如下所示：

```
// 定义前端参数和座位性质之间的 Map 集合
private static Map<String , String> seatNatureHashMap = new HashMap<String , String>() ;
static {
    seatNatureHashMap.put("busSeat" , "2") ;
    seatNatureHashMap.put("firstSeat" , "3") ;
    seatNatureHashMap.put("secondSeat" , "4") ;
    seatNatureHashMap.put("hardSleeper" , "7") ;
    seatNatureHashMap.put("softSleeper" , "6") ;
    seatNatureHashMap.put("hardSeat" , "5") ;
    seatNatureHashMap.put("softSeat" , "9") ;
}

// 依赖注入
@Autowired
private StringRedisTemplate redisTemplate ;

// 下单操作
public ResponseResult<String> submitOrder(OrderParams orderParams) {

    ...

    // 获取前端所传递的参数或者后端座位类型
    LOGGER.info("【分配座位开始了】 ---> " + JSON.toJSONString(orderParams));
    String seatNature = seatNatureHashMap.get(ticketType);

    // 分配座位: train seat:C1002:2020-06-01:4:6
    // 1. 获取所有的指定座位类型的所有车厢的 key
    String seatNatureKey = "train_seat:" + trainNum + ":" + trainRidingDate + ":" + seatNature + ":*" ;
    Set<String> allSeatNatureCoach = redisTemplate.keys(seatNatureKey);

    // 2. 遍历集合
    String coach = null ;
    String seatNo = null;
    out: for(String key : allSeatNatureCoach) {

        // 获取每一个车厢数据
        Map<Object, Object> entries = redisTemplate.boundHashOps(key).entries();
        coach = entries.get("coach_num").toString();

        // 遍历集合
        for(Object seatNoKey : entries.keySet()) {
```

```
// 获取每一个座位所对应的状态值
String seatStatus = entries.get(seatNoKey).toString();

// 状态值如果是 0，记录座位标号
if("0".equals(seatStatus)) {
    seatNo = seatNoKey.toString();
    redisTemplate.boundHashOps(key).put(seatNoKey.toString(), "1");
    break out;
}

}

}

LOGGER.info("【分配座位结束了】 ---> coach ---> {} , seatNo ---> {} " , coach , seatNo);

return null ;
}
```

### 2.3.3.3 生成订单

为了提高数据响应速度，我们在构建订单数据的时候可以使用一个线程来完成。并且在该线程执行完毕以后，要获取执行结果，因此这个任务类我们需要使用 Callable，执行这个任务的线程我们可以使用线程池来完成。具体的实现步骤如下所示：

- 1、创建雪花算法对应的类对象
- 2、创建线程池对象
- 3、获取指定日期的乘车计划数据，并将其封装到一个 TbRidingPlanDate 对象中
- 4、生成订单

具体的代码实现如下所示：

```
// 创建雪花算法对应类对象
private SnowflakeGenerator snowflakeGenerator = new SnowflakeGenerator( 0, 2) ;

// 创建一个线程池
private ExecutorService executorService = Executors.newFixedThreadPool(10) ;

// 依赖注入
@Autowired
private StringRedisTemplate redisTemplate ;

// 下单操作
public ResponseResult<String> submitOrder(OrderParams orderParams) {

    ...
    // 获取 ridingPlanDateKey 所对应的日期乘车计划数据
    Map<Object, Object> ridingPlanDateHashMap = redisTemplate.boundHashOps(ridingPlanDateKey).entries();
    TbRidingPlanDate tbRidingPlanDate = JSON.parseObject(JSON.toJSONString(ridingPlanDateHashMap),
    TbRidingPlanDate.class );

    ...
    // 生成订单数据
    String finalCoachNum = coach ;
    String finalSeatNum = seatNo ;
    Callable<ResponseResult<String>> callable = () -> {

        // 创建订单对象
        LOGGER.info("【构建订单开始了】 ---> " + JSON.toJSONString(orderParams));
        Long orderId = snowflakeGenerator.nextId();

        // 构建订单数据
        Order order = new Order();
        order.setId(orderId);
        order.setCoachNum(finalCoachNum);
        order.setIsEffect("1");
        order.setPayStatus("0");
        order.setRidingPlanDateId(orderParams.getRidingPlanId());
        order.setSeatNature(seatNature);
        order.setSeatNum(finalSeatNum);
        order.setTrainNum(orderParams.getTrainNum());
        order.setUserId(orderParams.getUserId());
        order.setTrainRidingDate(orderParams.getTrainRidingDate());
        order.setStartStationName(tbRidingPlanDate.getStartStationName());
        order.setReceiveStationName(tbRidingPlanDate.getReceiveStationName());

        if ("终点站".equals(tbRidingPlanDate.getStartTime())) {
            order.setStartTime(tbRidingPlanDate.getReceiveStartStationTime());
        } else {
            order.setStartTime(tbRidingPlanDate.getStartTime());
        }

        order.setEndTime(tbRidingPlanDate.getReceiveEndStationTime());
    };
}
```

```
// 根据座位情况设置具体的票价
if ("7".equals(seatNature)) { // 硬卧
    if (finalSeatNum.contains("上铺")) {
        order.setSeatPrice(tbRidingPlanDate.getHardSleeperUpPrice());
    } else if (finalSeatNum.contains("中铺")) {
        order.setSeatPrice(tbRidingPlanDate.getHardSleeperMiddlePrice());
    } else if (finalSeatNum.contains("下铺")) {
        order.setSeatPrice(tbRidingPlanDate.getHardSleeperDownPrice());
    }
} else if ("6".equals(seatNature)) { // 软卧
    if (finalSeatNum.contains("上铺")) {
        order.setSeatPrice(tbRidingPlanDate.getSoftSleeperUpPrice());
    } else if (finalSeatNum.contains("下铺")) {
        order.setSeatPrice(tbRidingPlanDate.getSoftSleeperDownPrice());
    }
} else if ("3".equals(seatNature)) {
    order.setSeatPrice(tbRidingPlanDate.getFirstSeatPrice());
} else if ("4".equals(seatNature)) {
    order.setSeatPrice(tbRidingPlanDate.getSecondSeatPrice());
} else if ("2".equals(seatNature)) {
    order.setSeatPrice(tbRidingPlanDate.getBusSeatPrice());
} else if ("5".equals(seatNature)) {
    order.setSeatPrice(tbRidingPlanDate.getHardSeatPrice());
} else if ("9".equals(seatNature)) {
    order.setSeatPrice(tbRidingPlanDate.getSoftSeatPrice());
}
LOGGER.info("【构建订单结束了】 ---> " + JSON.toJSONString(order));

responseResult.setResult(true);
return responseResult ;

} ;

try {

    Future<ResponseResult<String>> future = executorService.submit(callable);
    ResponseResult<String> stringResponseResult = future.get();
    return stringResponseResult ;

} catch (Exception e) {
    e.printStackTrace();
}

return null ;
}
```

### 2.3.3.4 Redis 排队

采用 Redis 中的 ZSet 集合存储排队信息，使用：列车编号 + 乘车日期 + 用户 id 作为 key，使用当前系统时间的纳秒值作为 value

```
LOGGER.info("【Redis 排队开始了】 ---> " + JSON.toJSONString(order));
String sortedKey = trainNum + ":" + trainRidingDate + ":" + orderParams.getUserId() ;
redisTemplate.boundZSetOps("train_manager:sorted_queue").add(sortedKey , System.nanoTime());
LOGGER.info("【Redis 排队结束了】 ---> " + JSON.toJSONString(order));
```

## 2.3.4 同步 ES 库存

### 2.3.4.1 发送同步数据

那么我们首先来完成一下生产者的代码，具体的步骤如下所示：

1、定义要同步的数据的 JavaBean

```
@Data // 通过 lombok 生成 getter 和 setter 方法
@NoArgsConstructor // 通过 lombok 生成无参构造方法
@ToString // 通过 lombok 生成 toString 方法
@EqualsAndHashCode // 通过 lombok 生成 hashCode 和 equals 方法
public class OrderSynEs {

    private String ridingPlanDate ; // 乘车日期
    private String seatNature ; // 座位性质
    private String trainNum ; // 列车车次

}
```

2、添加消息队列的相关配置

nacos 配置中心添加 rabbitmq 的配置信息

```
rabbitmq:
```

```
host: 172.17.0.224
port: 5672
username: admin
password: admin
virtual-host: /
publisher-confirm-type: correlated # 设置生产者通道支持confirm模式
```

RabbitmqConfiguration 配置类添加如下配置

```
// 声明队列
@Bean(name = "syn_stock_es_queue")
public Queue synOrderStockEsQueue() {
    return QueueBuilder.durable("syn_stock_es_queue").build();
}

// 声明交换机和队列的绑定关系
@Bean
public Binding trainExBindOrderStockEsQueue(@Qualifier(value = "train_manager_ex") Exchange exchange,
@Qualifier(value = "syn_stock_es_queue") Queue queue) {
    return BindingBuilder.bind(queue).to(exchange).with("syn_stock").noargs();
}
```

3、定义发送数据的 service（直接复制 itheima-train-manager 工程中的 RabbitmqProducer）

4、更改 OrderService 构建 OrderSynEs 对象，发送同步数据

```
LOGGER.info("【发送 ES 同步库存数据开始了】 ---> " + JSON.toJSONString(order));
OrderSynEs orderSynEs = new OrderSynEs();
orderSynEs.setRidingPlanDate(trainRidingDate);
orderSynEs.setSeatNature(seatNature);
orderSynEs.setTrainNum(trainNum);
rabbitmqProducer.sendMessage(JSON.toJSONString(orderSynEs), "syn_stock");
LOGGER.info("【发送 ES 同步库存数据结束了】 ---> " + JSON.toJSONString(orderSynEs));
```

### 2.3.4.2 接收同步数据

接下来我们就需要在 itheima-train-manager-es 同步数据工程中来接收同步数据，更新 ES 的库存信息。具体的实现步骤如下所示：

1、在 RabbitmqConfiguration 配置类添加如下配置：

```
// 声明队列
@Bean(name = "syn_stock_es_queue")
public Queue synOrderStockEsQueue() {
    return QueueBuilder.durable("syn_stock_es_queue").build();
}

// 声明交换机和队列的绑定关系
@Bean
public Binding trainExBindOrderStockEsQueue(@Qualifier(value = "train_manager_ex") Exchange exchange,
@Qualifier(value = "syn_stock_es_queue") Queue queue) {
    return BindingBuilder.bind(queue).to(exchange).with("syn_stock").noargs();
}
```

2、在 EsConsumer 类中添加监听同步库存队列的方法，方法代码逻辑如下所示：

```
@RabbitListener(queues = "syn_stock_es_queue")
public void synStock(String orderSynEsJson, Message message, Channel channel) {

    try {

        // 解析 json 数据
        OrderSynEs orderSynEs = JSON.parseObject(orderSynEsJson, OrderSynEs.class);

        // 调用 esIndexService 中的方法实现库存的同步操作
        esIndexService.synStockInfo(orderSynEs);
        logger.info("【同步下单扣减库存数据成功】 ---->" + orderSynEsJson);

        // 手动应答
        channel.basicAck(message.getMessageProperties().getDeliveryTag(), false);

    } catch (IOException e) {
        e.printStackTrace();
        logger.error("【同步下单扣减库存数据失败】 ---->" + orderSynEsJson);
    }
}
```

ESIndexService 类添加如下方法：

```
// 同步库存的扣减操作
public void synStockInfo(OrderSynEs orderSynEs) throws IOException {

    // 创建搜索请求对象
    SearchRequest searchRequest = new SearchRequest("riding_plan_date");
}
```

```
SearchSourceBuilder searchSourceBuilder = new SearchSourceBuilder().trackTotalHits(true) ; // 查询所有满足条件的数据
BoolQueryBuilder boolQueryBuilder = QueryBuilders.boolQuery() ;
boolQueryBuilder.must(QueryBuilders.termQuery("trainNum" , orderSynEs.getTrainNum())) ;
boolQueryBuilder.must(QueryBuilders.termQuery("trainRidingDate" , orderSynEs.getRidingPlanDate())) ;
searchSourceBuilder.query(boolQueryBuilder) ;
searchSourceBuilder.size(1000) ;
searchRequest.source(searchSourceBuilder) ;

// 进行搜索
SearchResponse searchResponse = restHighLevelClient.search(searchRequest, RequestOptions.DEFAULT) ;
SearchHits searchHits = searchResponse.getHits() ;
SearchHit[] hits = searchHits.getHits() ;
for(SearchHit searchHit : hits) {
    String sourceAsString = searchHit.getSourceAsString() ;
    TbRidingPlanDate ridingPlanDate = JSON.parseObject(sourceAsString, TbRidingPlanDate.class) ;
    switch (orderSynEs.getSeatNature()) {
        case "3" :
            ridingPlanDate.setFirstSeatStock(ridingPlanDate.getFirstSeatStock() - 1) ;
            break ;
        case "4" :
            ridingPlanDate.setSecondSeatStock(ridingPlanDate.getSecondSeatStock() - 1) ;
            break ;
        case "7" :
            ridingPlanDate.setHardSleeperStock(ridingPlanDate.getHardSleeperStock() - 1) ;
            break ;
        case "6" :
            ridingPlanDate.setSoftSleeperStock(ridingPlanDate.getSoftSleeperStock() - 1) ;
            break ;
        case "2" :
            ridingPlanDate.setBusSeatStock(ridingPlanDate.getBusSeatStock() - 1) ;
            break ;
        case "5" :
            ridingPlanDate.setHardSeatStock(ridingPlanDate.getHardSeatStock() - 1) ;
            break ;
        case "9" :
            ridingPlanDate.setSoftSeatStock(ridingPlanDate.getSoftSeatStock() - 1) ;
            break ;
    }

    // 创建一个更新的请求对象
    UpdateRequest updateRequest = new UpdateRequest("riding_plan_date" ,
searchHit.getId()).doc(JSON.toJSONString(ridingPlanDate) , XContentType.JSON) ;
    restHighLevelClient.update(updateRequest , RequestOptions.DEFAULT) ;
}
}
```

### 2.3.5 发送订单数据

#### 实现思路

按照我们最初的想法，我们只需要将订单 Order 对象发送到 MQ 中，然后订单处理服务从队列中监听到 Order 数据生成订单即可。但是我们后续还有另外一个操作，就是在下单成功以后，我们需要将下单状态通过 websocket 推送给用户，因此我们需要在订单处理服务中不仅仅需要获取到订单数据，还需要获取到用户的数据。因此在发送下单数据的时候单单发送订单的数据是不够的。我们需要创建一个实体类，用来封装订单以及该订单所对应的用户数据（OrderHandler）。

#### 代码实现

具体的实现步骤如下所示：

- 1、创建实体类(OrderHandler)（用于封装发送给 MQ 的订单数据）

```
@Data // 通过lombok生成getter和setter方法
@NoArgsConstructor // 通过lombok生成无参构造方法
@ToString // 通过lombok生成toString方法
@EqualsAndHashCode // 通过lombok生成hashCode和equals方法
public class OrderHandler {

    private Order order ; // 订单信息
    private User user ; // 用户信息
}
```

- 2、RabbitmqConfiguration 配置类添加如下配置

```
// 声明队列
@Bean(name = "syn_gen_order_queue")
public Queue genOrderQueue() {
    return QueueBuilder.durable("syn_gen_order_queue").build() ;
}

// 声明交换机和队列的绑定关系
@Bean
public Binding trainExBindGenOrderQueue(@Qualifier(value = "train_manager_ex") Exchange exchange ,
@Qualifier(value = "syn_gen_order_queue") Queue queue) {
```

```
return BindingBuilder.bind(queue).to(exchange).with("gen_order").noargs() ;
}
```

### 3、更改 OrderService 发送订单数据

```
// 创建一个 OrderHandler 对象
LOGGER.info("【发送订单数据开始了】 ---> " + JSON.toJSONString(order));
User user = User
JSON.parseObject(redisTemplate.boundHashOps("train_manager:userInfo").get(orderParams.getUserId()).toString(), User.class);
OrderHandler orderHandler = new OrderHandler();
orderHandler.setOrder(order);
orderHandler.setUser(user);
rabbitmqProducer.sendMessage(JSON.toJSONString(orderHandler), "gen_order");
LOGGER.info("【发送订单数据结束了】 ---> " + JSON.toJSONString(orderHandler));
```

## 2.3.6 查询排队接口

当用户下单完毕以后，就会跳转到下单成功页面。那么在下单成功页面就需要调用查询排队接口，去查询排队信息。  
接口定义

```
@RequestMapping(value = "/queryQueue")
public ResponseResult<String> queryQueue(@RequestBody OrderParams orderParams) {
    return orderService.querySortedInfo(orderParams);
}
```

业务逻辑实现

```
// 查询排队信息
public ResponseResult<String> querySortedInfo(OrderParams orderParams) {

    // 创建 ResponseResult 对象
    ResponseResult<String> responseResult = new ResponseResult<>();

    // 从 Redis 中去获取排队信息
    String sortedKey = orderParams.getTrainNum() + ":" + orderParams.getTrainRidingDate() + ":" + orderParams.getUserId();
    System.out.println(sortedKey);
    Long rank = redisTemplate.boundZSetOps("train_manager:sorted_queue").rank(sortedKey);
    System.out.println(rank);

    // 判断结果
    if(rank == null) {
        responseResult.setResult(false);
    } else {
        responseResult.setResult(true);
        responseResult.setData(String.valueOf(rank));
    }

    // 返回
    return responseResult;
}
```

## 2.4 订单处理服务

### 2.4.1 获取订单队列数据

我们首先来完成以下订单处理服务获取订单数据的代码。具体的实现步骤如下所示：

- 1、导入订单处理服务工程(itheima-train-manager-order-handler)
- 2、在 nacos 配置中心中创建订单处理服务的配置 (itheima-train-manager-order-handler-dev.yml)

```
spring:
  application:
    name: itheima-train-manager-order-handler
  rabbitmq:
    host: 172.17.0.224
    port: 5672
    username: admin
    password: admin
    virtual-host: /
    listener:
      simple:
        acknowledge-mode: manual # 设置消息手动应答
# 配置数据源
datasource:
  driver-class-name: com.mysql.jdbc.Driver
  url: jdbc:mysql://172.17.0.224:3366/db_train
  username: root
```



```
password: 1234
# 配置mybatis的信息
mybatis:
  type-aliases-package: com.itheima.train.manager.domain
  mapper-locations: classpath:com/itheima/train/manager/order/handler/mapper/*Mapper.xml
server:
  port: 10001
```

### 3、添加消息监听代码

```
@Component
public class OrderHandlerConsumer {

    // 定义日志记录器
    private Logger LOGGER = LoggerFactory.getLogger(OrderHandlerConsumer.class);

    @RabbitListener(queues = "syn_gen_order_queue")
    public void orderHandlerConsumer(String orderHandlerJsonInfo, Message message, Channel channel) {

        try {

            // 解析数据
            OrderHandler orderHandler = JSON.parseObject(orderHandlerJsonInfo, OrderHandler.class);

            // 调用service进行数据保存
            LOGGER.info("【保存订单数据完成】 ---> " + orderHandlerJsonInfo);

            // 给出应答
            channel.basicAck(message.getMessageProperties().getDeliveryTag(), false);

        } catch (IOException e) {
            e.printStackTrace();
            LOGGER.info("【保存订单数据失败】 ---> " + orderHandlerJsonInfo);
        }

    }

}
```

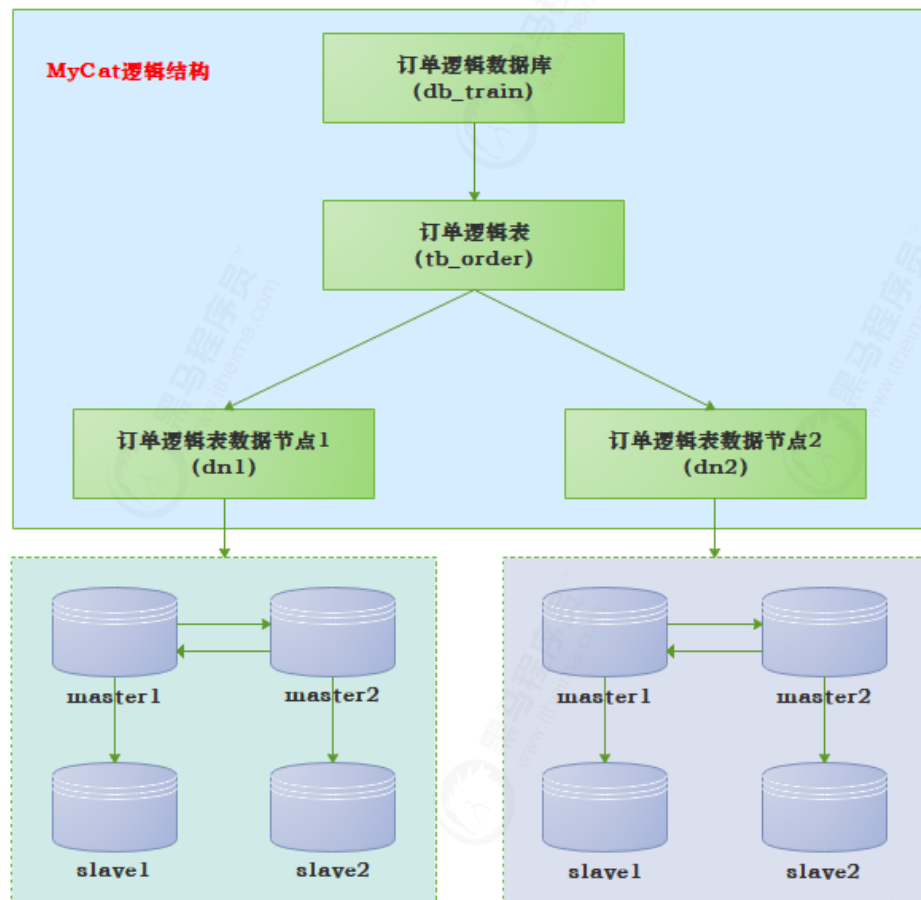
## 2.4.2 订单数据库架构

订单数据特点：写并发量大于读并发量

如何提高我们写数据的能力，给用户良好的用户体验，就是我们需要研究的目标！

设计方向：

- 1、多个节点进行数据写入
  - 2、进行读写分离操作，提高单节点写数据的并发能力
  - 3、要保证每一个写入节点的高可用，当主节点出现问题以后，从节点立马升级为主节点
- 基于以上几点的设计思路，我们所设计出来的订单数据库的架构如下所示：



### 2.4.3 MySql 主从复制

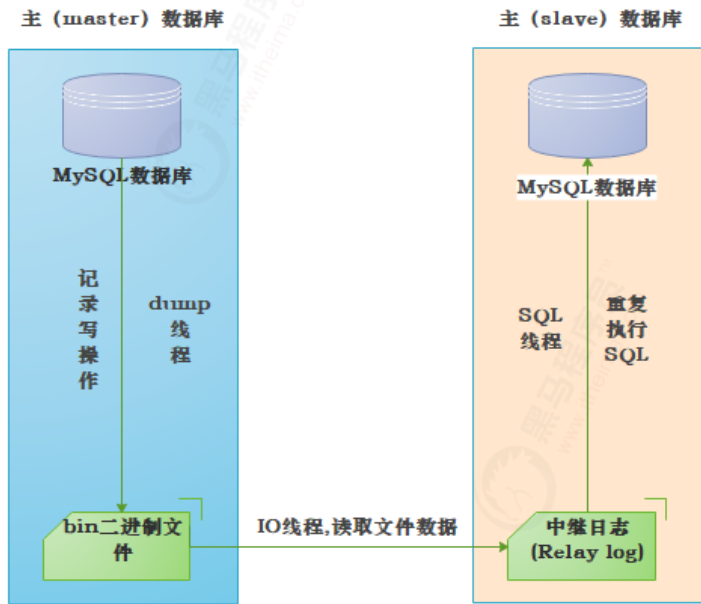
#### 主从复制简介

就是有两个数据库服务器，一个是主（master）数据库服务器，另一个是从（slave）数据库服务器。当主（master）数据库有数据写入，包括插入、删除、修改，都会在主（slave）数据库上操作一次。这样的操作下，主从（slave）数据库的数据都是一样的，就相当于时刻在做数据备份，就算主（master）数据库的数据全部丢失了，还有从（slave）数据库的数据，我们就可以把从（slave）数据库的数据导出来进行数据恢复。

#### 主从复制原理

主从复制原理主要有三个线程不断在工作：

- 1、主（master）数据库启动 bin 二进制日志，这样会有一个 Dump 线程，这个线程是把主（master）数据库的写入操作都会记录到这个 bin 的二进制文件中。
- 2、然后从（slave）数据库会启动一个 I/O 线程，这个线程主要是把主（master）数据库的 bin 二进制文件读取到本地，并写入到中继日志（Relay log）文件中。
- 3、最后从（slave）数据库其他 SQL 线程，把中继日志（Relay log）文件中的事件再执行一遍，更新从（slave）数据库的数据，保持主从数据一致。



### 2.4.4 一主一从介绍

#### 一主一从部署

在 Mycat 中，读写分离可以说有两种：一种是一主一从，另一种是多主多从。接下来我们就来首先给大家介绍一下一主一从的部署。分为两步进行实现：

- 1、一主一从的数据库配置实现
- 2、MyCat 一主一从实现读写分离配置

一主一从的 MySql 部署步骤如下所示：

- 1、在指定的目录下创建对应的文件夹（做 mysql 容器的目录映射）

```
mysql-master-slave
```

```
mysql10
```

```
config
```

```
data
```

```
mysql11
```

```
config
```

```
data
```

- 2、将 docker 中的 mysql 配置文件分别复制到 mysql10/config 和 mysql11/config 目录中

- 3、对两个节点的配置文件（/usr/local/mysql-master-slave/mysql10/config/mysql.conf.d）做如下配置：

```
lower_case_table_names=1      # 不区分字母大小写

log-bin=mysql-bin             # 开启二进制日志

server-id=10                  # 设置 server-id , 从节点指定一个其他的 id 即可

# 不同步哪些数据库

binlog-ignore-db = mysql
```

```
binlog-ignore-db = test
```

```
binlog-ignore-db = information_schema
```

#### 4、创建两个 mysql 的容器

```
docker run -di --network=docker-network --ip=172.19.0.110 --name=train mysql 01 -p 3410:3306 --privileged=true -e MYSQL_ROOT_PASSWORD=1234 -v /usr/local/mysql-master-slave/mysql10/data:/var/lib/mysql -v /usr/local/mysql-master-slave/mysql10/config:/etc/mysql/ mysql:5.6
```

```
docker run -di --network=docker-network --ip=172.19.0.111 --name=train mysql 02 -p 3411:3306 --privileged=true -e MYSQL_ROOT_PASSWORD=1234 -v /usr/local/mysql-master-slave/mysql11/data:/var/lib/mysql -v /usr/local/mysql-master-slave/mysql11/config:/etc/mysql/ mysql:5.6
```

#### 5、进入到 mysql10 容器中，主节点状态

```
mysql> show master status ;
```

```
+-----+-----+-----+-----+-----+
| File           | Position | Binlog_Do_DB | Binlog_Ignore_DB | Executed_Gtid_Set |
+-----+-----+-----+-----+-----+
| mysql-bin.000004 |      120 |              | mysql,test,information_schema |                    |
+-----+-----+-----+-----+-----+

1 row in set (0.00 sec)
```

```
mysql>
```

其中 File 和 Position 都是我们在设置从（slave）数据库的时候用到的；

#### 6、登录到从数据库，进行以下配置。

配置主（master）数据库的 IP 地址，用户名，登录密码，刚才在主（master）数据库中查到的 bin 二进制文件的名称和所在的位置。

```
mysql> stop slave; # 先关闭从节点
```

```
Query OK, 0 rows affected, 1 warning (0.00 sec)
```

```
mysql> change master to master host='172.19.0.110', master user='root', master password='1234',
master_log_file='mysql-bin.000004', master_log_pos=120;
```

```
Query OK, 0 rows affected, 2 warnings (0.34 sec)
```

```
mysql> start slave;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> show slave status\G # 查看从节点状态
```

### 一主一从测试

接下来我们就来对刚才所部署的一主一从架构的数据库进行测试。

具体的测试步骤如下所示：

- 1、使用 Navicat 连接 mysql10 和 mysql11
- 2、在 mysql10 节点上创建一个数据库 db\_train，然后查看从数据库的变化
- 3、在主节点上 db\_train 数据库中创建表以及插入数据，然后查看从数据库的变化

### MyCat 读写分离

接下来我们就来研究一下基于一主一从架构的 mysql，要实现读写分离我们的 mycat 应该如何进行配置。

#### 1、配置 server.xml 文件（给指定的用户添加逻辑库）

```
<user name="root" >
  <property name="password">1234</property>
  <property name="schemas">db train</property>
  <property name="defaultSchema">db train</property>
</user>
```

#### 2、mycat 的读写分离主要是在 schema.xml 文件中进行配置的，通过 dataHost 的 3 个属性结合其子标签<writeHost>和<readHost>配合完成

**balance 属性负载均衡类型，目前的取值有 4 种：**

balance="0"，不开启读写分离机制，所有读操作都发送到当前可用的 writeHost 上。

balance="1"，全部的 readHost 与 stand by writeHost 参与 select 语句的负载均衡，简单的说，当双主双从模式(M1->S1，M2->S2，并且 M1 与 M2 互为主备)，正常情况下，M2,S1,S2 都参与 select 语句的负载均衡。

balance="2"，所有读操作都随机的在 writeHost、readhost 上分发。

balance="3"，所有读请求随机的分发到 writerHost 对应的 readhost 执行，writerHost 不负担读压力，注意 balance=3 只在 1.4 及其以后版本有，

1.3 没有。

**writeType 属性，负载均衡类型，目前的取值有 3 种：**

writeType="0"，所有写操作发送到配置的第一个 writeHost，第一个挂了切到还生存的第二个 writeHost，重新启动后已切换后的为准，切换记录在配置文件中:dnindex.properties .

writeType="1"，所有写操作都随机的发送到配置的 writeHost，1.5 以后废弃不推荐。

writeType="2"，官方文档没有介绍。

**switchType 属性：**

-1 表示不自动切换

1 默认值，自动切换

2 基于 MySQL 主从同步的状态决定是否切换

```
<!--配置逻辑库-->
<schema name="db train" checkSQLschema="true" sqlMaxLimit="100" dataNode="dataNode1"></schema>

<!--配置分片节点-->
<dataNode name="dataNode1" dataHost="train mysql 01" database="db train" />

<!--配置节点主机-->
<dataHost name="train mysql 01" maxCon="1000" minCon="10" balance="1" writeType="0" dbType="mysql"
dbDriver="native" switchType="1" slaveThreshold="100">
  <heartbeat>select user()</heartbeat>
  <writeHost host="M1" url="172.19.0.110:3306" user="root" password="1234">
    <readHost host="S1" url="172.19.0.111:3306" user="root" password="1234"/>
  </writeHost>
</dataHost>
```

3、重启 mycat 进行测试，测试思路如下所示：

删除 mysql11 上的 id 为 1 的数据，然后在进行查询，如果此时查询不到数据，那么说明查询是从 mysql11（slave）节点进行查询的。如果可以查询到数据那么就说明查询还是使用的 mysql10（master）节点。

## 2.4.5 双主双从介绍

接下来我们就来给大家介绍一个 mysql 多主多从架构的主从复制，我们本次以双主双从为例来给大家演示（为了不影响本次演示，我们最好删除之前一主一从的测试数据）。

主要分为两步进行实现：

1、双主双从的数据库配置实现

2、MyCat 双主双从实现读写分离配置

### MySQL 环境部署

1、在指定的位置按照如下的目录结构创建目录（后期做 MySql 的目录映射）

mysql-master-slave

mysql20

config

data

mysql21

config

data

2、将 docker 中的 mysql 配置文件分别复制到 mysql20/config 和 mysql21/config 目录中

3、对两个节点的配置文件做如下配置：

/usr/local/mysql-master-slave/mysql20/config/mysql.conf.d

```
lower_case_table_names=1      # 不区分字母大小写

log-bin=mysql-bin             # 开启二进制日志

server-id=20                  # 设置 server-id

log-slave-updates              # 在作为从数据库的时候，有写入操作也要更新二进制日志文件

# 不同步哪些数据库
```

```
binlog-ignore-db = mysql

binlog-ignore-db = test

binlog-ignore-db = information_schema

/usr/local/mysql-master-slave/mysql21/config/mysql.conf.d
```

```
lower_case_table_names=1                # 不区分字母大小写

log-bin=mysql-bin                        # 开启二进制日志

server-id=21                             # 设置 server-id

# 不同步哪些数据库

binlog-ignore-db = mysql

binlog-ignore-db = test

binlog-ignore-db = information_schema
```

4、更改 mysql10 节点的配置文件，添加如下配置

```
log-slave-updates                # 在作为从数据库的时候，有写入操作也要更新二进制日志文件（重启容器）
```

5、创建 mysql 容器

```
docker run -di --network=docker-network --ip=172.19.0.120 --name=train mysql 03 -p 3420:3306 --privileged=true -e MYSQL_ROOT_PASSWORD=1234 -v /usr/local/mysql-master-slave/mysql20/data:/var/lib/mysql -v /usr/local/mysql-master-slave/mysql20/config:/etc/mysql/ mysql:5.6

docker run -di --network=docker-network --ip=172.19.0.121 --name=train mysql 04 -p 3421:3306 --privileged=true -e MYSQL_ROOT_PASSWORD=1234 -v /usr/local/mysql-master-slave/mysql21/data:/var/lib/mysql -v /usr/local/mysql-master-slave/mysql21/config:/etc/mysql/ mysql:5.6
```

## 双主双从配置

1、进入到 train\_mysql\_01 容器中，查看该节点的 mysql 的主节点状态

```
mysql> show master status ;

+-----+-----+-----+-----+-----+
| File           | Position | Binlog_Do_DB | Binlog_Ignore_DB | Executed_Gtid_Set |
+-----+-----+-----+-----+-----+
| mysql-bin.000004 | 1799 | | mysql,test,information_schema | |
+-----+-----+-----+-----+-----+

1 row in set (0.00 sec)

mysql>
```

其中 File 和 Position 都是我们在设置从（slave）数据库的时候用到的；

2、配置 train\_mysql\_01 的从数据库，需要配置的是 train\_mysql\_02 和 train\_mysql\_03 的数据库，配置如下：train\_mysql\_02 配置步骤如下所示：

```
mysql> stop slave ;

Query OK, 0 rows affected (0.00 sec)

mysql> change master to master_host='172.19.0.110', master_user='root', master_password='1234',
master_log_file='mysql-bin.000004', master_log_pos=1799;

Query OK, 0 rows affected, 2 warnings (0.05 sec)

mysql> start slave;

Query OK, 0 rows affected (0.00 sec)

mysql> show slave status\G
```

3、配置 train\_mysql\_03 的从数据库，需要配置的是 train\_mysql\_01 和 train\_mysql\_04 的数据库，配置如下：进入到 train\_mysql\_03 的 docker 容器，查看 train\_mysql\_03 的主节点信息：

```
mysql> show master status ;
```

```
+-----+-----+-----+-----+-----+
| File           | Position | Binlog_Do_DB | Binlog_Ignore_DB | Executed_Gtid_Set |
+-----+-----+-----+-----+-----+
| mysql-bin.000004 |      120 |              | mysql,test,information_schema |                    |
+-----+-----+-----+-----+-----+

1 row in set (0.00 sec)
```

需要配置的是 train\_mysql\_01 和 train\_mysql\_04 的数据库，配置如下：

```
mysql> stop slave ;

Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql> change master to master_host='172.19.0.120', master_user='root', master_password='1234',
master_log_file='mysql-bin.000004', master_log_pos=120;

Query OK, 0 rows affected, 2 warnings (0.06 sec)

mysql> start slave ;

Query OK, 0 rows affected (0.01 sec)
```

### 双主双从测试

接下来我们就来对刚才所部署的多主多从架构的数据库进行测试。具体的测试步骤如下所示：

- 1、使用 Navicat 分别连接 train\_mysql\_01, train\_mysql\_02, train\_mysql\_03, train\_mysql\_04 节点。然后在 train\_mysql\_01 节点上创建一个数据库，并且创建表并且插入数据。看看其他数据库是否存在变化。
- 2、在 train\_mysql\_03 节点上创建一个数据库，然后在创建表并且插入数据。看看其他数据库是否存在变化。

### MyCat 读写分离

接下来我们就来研究一下基于双主双从架构的 mysql, mycat 如何进行配置以实现双主的主备写入操作。具体的实现步骤如下所示：

- 1、修改 schema.xml 文件 train\_mysql\_01 节点主机的配置

```
<!--配置节点主机-->
<dataHost name="train mysql 01" maxCon="1000" minCon="10" balance="1" writeType="0" dbType="mysql"
dbDriver="native" switchType="1" slaveThreshold="100">
  <heartbeat>select user()</heartbeat>
  <writeHost host="M1" url="172.19.0.110:3306" user="root" password="1234">
    <readHost host="S1" url="172.19.0.111:3306" user="root" password="1234"/>
  </writeHost>
  <writeHost host="M2" url="172.19.0.120:3306" user="root" password="1234">
    <readHost host="S2" url="172.19.0.121:3306" user="root" password="1234"/>
  </writeHost>
</dataHost>
```

- 2、重启 mycat
- 3、进行测试

由于我们之前已经测试过读写分离了，因此本次我们就不在测试读写分离了。本次我们主要测试两部分内容：

- 读数据的负载均衡
- 写数据的高可用

**读数据的负载均衡**具体的测试步骤如下所示：

- 1、创建数据库表 tb\_user

```
create table tb user(
  id int not null primary key ,
  name varchar(20) ,
  city varchar(20)
) ;
```

- 2、插入测试数据

```
insert into tb user (id, name , city) values (1 , "test1" , 'beijing') ;
insert into tb user (id, name , city) values (2 , "test2", 'xian') ;
insert into tb user (id, name , city) values (3 , "test3", 'shanghai') ;
insert into tb user (id, name , city) values (4 , "test4", 'shandong') ;
insert into tb_user (id, name , city) values (5 , "test5", 'guangxi') ;
```

此时我们发现 4 个节点上的数据都是相同的。

- 3、删除节点 3 上的 4 和 5 的数据，保留 id 为 1,2,3 的数据
- 4、删除节点 4 上的 id 为 1,2,3 的数据，重新插入 id 为 4 和 5 的数据

```
insert into tb_user (id , name , city) values (4 , "test4", 'shandong') ;
insert into tb_user (id , name , city) values (5 , "test5", 'guangxi') ;
```

5、给节点 2 插入数据

```
insert into tb_user (id , name , city) values (6 , "test6", 'guangxi') ;
```

6、通过 mycat 进行插入，观察数据的变化

最后有一点要说的是，在真实的项目中，不应该对从数据库（slave）做写入操作，这样会破坏数据的一致性的。

**写数据的高可用**具体的测试步骤如下所示：

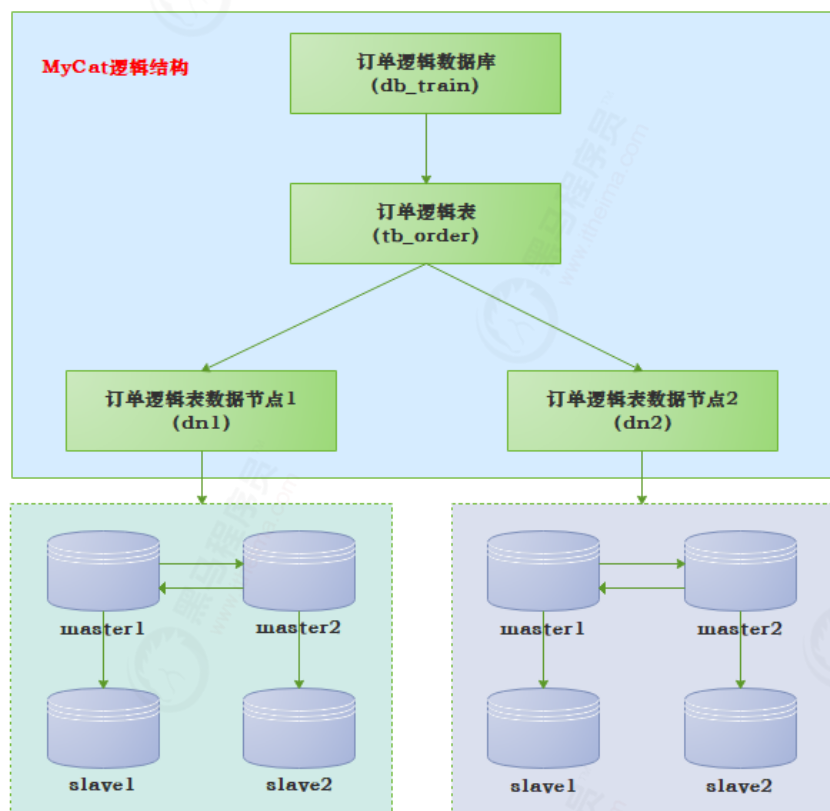
- 1、关闭 train\_mysql\_01 节点
- 2、通过 mycat 插入数据，观察数据是否可以被正常插入

```
insert into tb_user (id , name , city) values (7 , "test7", 'beijing') ;
```

3、关闭 train\_mysql\_03 节点，然后在尝试插入数据，观察数据是否可以被正常插入

## 2.4.6 订单数据库部署

接下来我们就需要按照我们之前所讲解的订单数据库的架构，来部署我们的订单数据库。



按照上述架构我们需要部署两套双主双从的 mysql，双主双从的 mysql 我们已经讲解过了。因此第二套双主双从的步骤我们不在课程中讲解了。我们主要讲解的是 mycat 的配置，具体配置如下所示：

1、修改 schema.xml 文件

```
<!--配置逻辑库-->
<schema name="db train" checkSQLSchema="true" sqlMaxLimit="100" dataNode="dataNode1">

    <!-- 订单数据表分片 -->
    <table name="tb order" dataNode="dataNode1,dataNode2" rule="sharding-by-murmur-order" primaryKey="id" />
</schema>

<!--配置分片节点-->
<dataNode name="dataNode1" dataHost="train mysql 01" database="db train" />
<dataNode name="dataNode2" dataHost="train mysql 02" database="db train" />

<!--配置节点主机-->
<dataHost name="train mysql 01" maxCon="1000" minCon="10" balance="1" writeType="0" dbType="mysql"
dbDriver="native" switchType="1" slaveThreshold="100">
    <heartbeat>select user()</heartbeat>
    <writeHost host="M1" url="172.19.0.110:3306" user="root" password="1234">
        <readHost host="S1" url="172.19.0.111:3306" user="root" password="1234"/>
    </writeHost>
    <writeHost host="M2" url="172.19.0.120:3306" user="root" password="1234">
        <readHost host="S2" url="172.19.0.121:3306" user="root" password="1234"/>
    </writeHost>
</dataHost>

<!--配置节点主机-->
<dataHost name="train mysql 02" maxCon="1000" minCon="10" balance="1" writeType="0" dbType="mysql"
dbDriver="native" switchType="1" slaveThreshold="100">
    <heartbeat>select user()</heartbeat>
    <writeHost host="M1" url="172.19.0.130:3306" user="root" password="1234">
```

```
<readHost host="S1" url="172.19.0.131:3306" user="root" password="1234"/>
</writeHost>
<writeHost host="M2" url="172.19.0.140:3306" user="root" password="1234">
  <readHost host="S2" url="172.19.0.141:3306" user="root" password="1234"/>
</writeHost>
</dataHost>
```

2、在 rule.xml 加入订单数据分片规则

```
<!-- 订单数据的分片规则 -->
<tableRule name="sharding-by-murmur-order">
  <rule>
    <columns>id</columns>
    <algorithm>murmur-order</algorithm>
  </rule>
</tableRule>

<!-- 分片算法配置 -->
<function name="murmur-order" class="io.mycat.route.function.PartitionByMurmurHash">
  <property name="seed">0</property>
  <property name="count">2</property>      <!-- 定义分片数量 -->
  <property name="virtualBucketTimes">160</property>
</function>
```

3、连接 mycat 创建订单数据库表

```
CREATE TABLE `tb_order` (
  `id` bigint(25) NOT NULL,
  `riding plan date id` bigint(25) DEFAULT NULL,
  `start station name` varchar(255) DEFAULT NULL,
  `receive station name` varchar(11) DEFAULT NULL,
  `train num` varchar(11) DEFAULT NULL,
  `train riding date` varchar(20) DEFAULT NULL,
  `start time` varchar(11) DEFAULT NULL,
  `end time` varchar(11) DEFAULT NULL,
  `seat num` varchar(10) DEFAULT NULL,
  `seat nature` varchar(10) DEFAULT NULL,
  `coach num` varchar(10) DEFAULT NULL,
  `seat price` double(10,2) DEFAULT NULL,
  `pay status` varchar(3) DEFAULT NULL,
  `user id` varchar(100) DEFAULT NULL,
  `is effect` varchar(3) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

## 2.4.7 保存订单数据

接下来我们就来完成以下保存订单数据的代码，具体的实现步骤如下所示：

- 1、创建 Mapper 接口（TbOrderMapper）
- 2、创建 Mapper 映射文件（TbOrderMapper.xml）

```
<!-- 保存订单数据 -->
<insert id="saveOrder" parameterType="com.itheima.train.manager.domain.Order">

  insert into tb_order (id , riding plan date id , start station name , receive station name , train num ,
train riding date , start time , end time , seat num , seat nature , coach num , seat price , pay status ,
user id , is effect)
  values (
    #{id } ,
    #{ridingPlanDateId } ,
    #{startStationName } ,
    #{receiveStationName } ,
    #{trainNum } ,
    #{trainRidingDate } ,
    #{startTime } ,
    #{endTime } ,
    #{seatNum } ,
    #{seatNature } ,
    #{coachNum } ,
    #{seatPrice } ,
    #{payStatus } ,
    #{userId } ,
    #{isEffect }
  );

</insert>

<!-- 映射结果集 -->
<resultMap id="BASE_RESULT_MAP" type="com.itheima.train.manager.domain.Order">
  <id property="id" column="id" />
  <result property="ridingPlanDateId" column="riding plan date id" />
  <result property="startStationName" column="start_station_name" />
</resultMap>
```

```
<result property="receiveStationName" column="receive_station_name" />
<result property="trainNum" column="train_num"/>
<result property="trainRidingDate" column="train_riding_date" />
<result property="startTime" column="start_time"/>
<result property="endTime" column="end_time"/>
<result property="seatNum" column="seat_num"/>
<result property="seatNature" column="seat_nature"/>
<result property="coachNum" column="coach_num"/>
<result property="seatPrice" column="seat_price"/>
<result property="payStatus" column="pay_status" />
<result property="isEffect" column="is_effect" />
<result property="userId" column="user_id" />
</resultMap>

<!-- 根据用户的id，列车的编号，以及乘车日期查询订单信息 -->
<select id="queryOrderInfo" parameterType="map" resultMap="BASE_RESULT_MAP">
    select * from tb_order where user id = #{userId} and train num = #{trainNum} and train riding date =
    #{trainRidingDate} and is effect = "1"
</select>
```

3、编写 service，修改 OrderHandlerConsumer 类

```
// 保存订单数据
public Order saveOrder(Order order) {

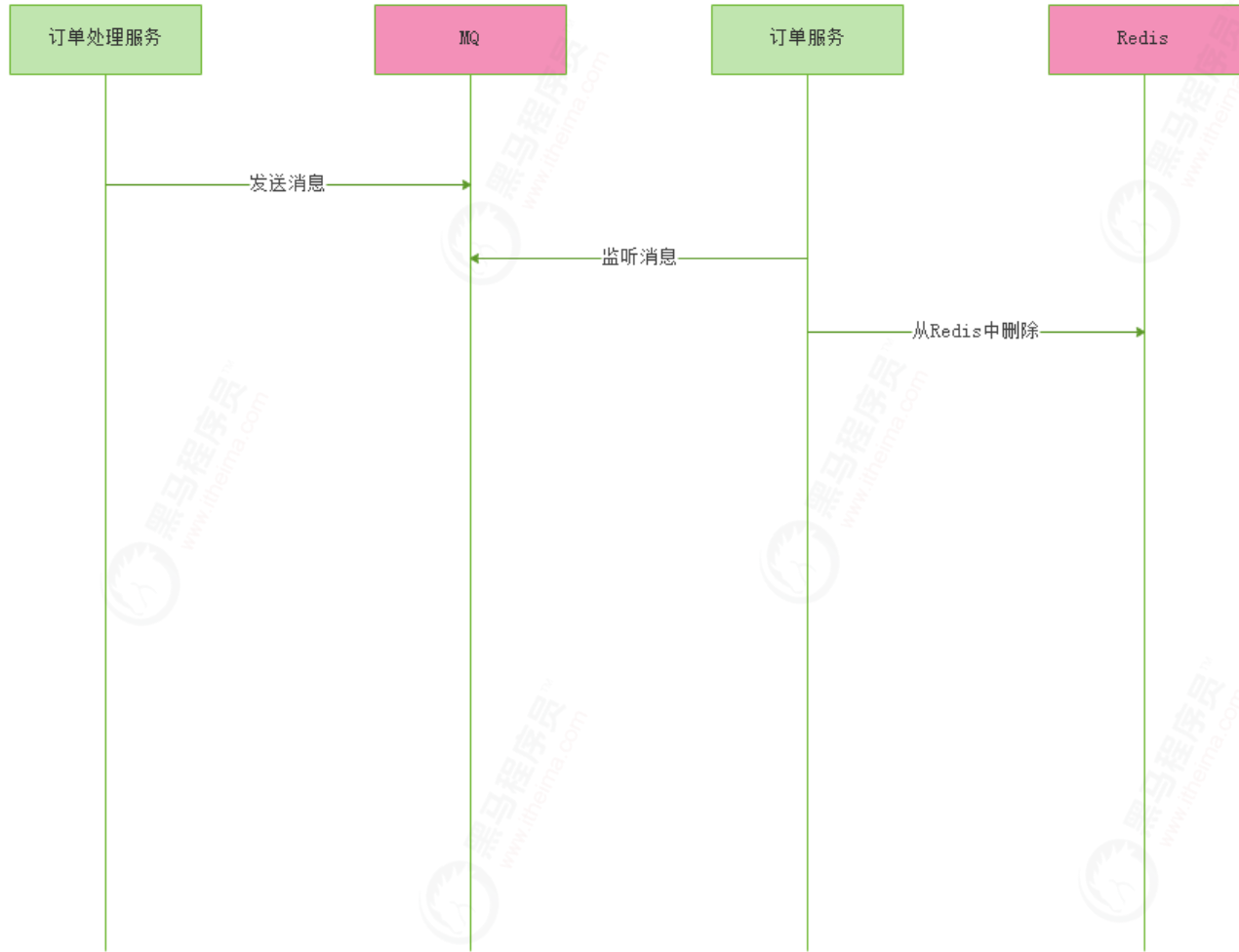
    // 查询数据
    HashMap hashMap = new HashMap() ;
    hashMap.put("userId" , order.getUserId()) ;
    hashMap.put("trainNum" , order.getTrainNum()) ;
    hashMap.put("trainRidingDate" , order.getTrainRidingDate()) ;
    Order dbOrder = tbOrderMapper.queryOrderInfo(hashMap);

    // 判断
    if(dbOrder == null) {
        tbOrderMapper.saveOrder(order);
        return order ;
    }else {
        return dbOrder ;
    }
}
```

### 2.4.8 删除排队信息

#### 订单处理服务

当订单保存成功以后，我们就需要将用户的排队信息从 Redis 中进行删除。删除的具体流程如下所示：



订单处理服务发送消息，具体的实现步骤如下所示：

1、添加队列配置信息

```
// 从 Redis 中删除排队信息的的队列
@Bean(name = "delete_sorted_from_redis")
public Queue deleteSortedFromRedisQueue() {
```

```
return QueueBuilder.durable("delete_sorted_from_redis").build() ;
}

// 声明交换机和队列的绑定关系
@Bean
public Binding trainExBinddeleteSortedFromRedisQueue(@Qualifier(value = "train_manager_ex") Exchange exchange , @Qualifier(value = "delete_sorted_from_redis") Queue queue) {
    return BindingBuilder.bind(queue).to(exchange).with("delete_sorted").noargs() ;
}
```

## 2、配置文件中添加支持生产者确认模式

```
publisher-confirm-type: correlated # 设置生产者通道支持 confirm 模式
```

## 3、发送消息 OrderHandlerConsumer

订单保存成功以后，发送删除消息

```
order = orderService.saveOrder(orderHandler.getOrder());
LOGGER.info("【保存订单数据完成】 ----> " + orderHandlerJsonInfo);

// 发送删除 redis 中的排队信息消息
String messageBody = order.getTrainNum() + ":" + order.getTrainRidingDate() + ":" + order.getUserId() ;
rabbitmqProducer.sendMessage(messageBody , "delete_sorted");
```

## 订单服务

订单服务需要接收消息，将用户的排队信息从 Redis 中删除掉。具体的实现步骤如下所示：

### 1、配置文件中加入如下配置

```
listener:
  simple:
    acknowledge-mode: manual # 设置消息手动应答
```

### 2、添加队列配置

```
// 从 Redis 中删除排队信息的的队列
@Bean(name = "delete_sorted_from_redis")
public Queue deleteSortedFromRedisQueue() {
    return QueueBuilder.durable("delete_sorted_from_redis").build() ;
}

// 声明交换机和队列的绑定关系
@Bean
public Binding trainExBinddeleteSortedFromRedisQueue(@Qualifier(value = "train_manager_ex") Exchange exchange , @Qualifier(value = "delete_sorted_from_redis") Queue queue) {
    return BindingBuilder.bind(queue).to(exchange).with("delete_sorted").noargs() ;
}
```

### 3、接收消息删除排队信息

```
@Component
public class OrderConsumer {

    // 创建日志对象
    private Logger LOGGER = LoggerFactory.getLogger(OrderConsumer.class) ;

    @Autowired
    private StringRedisTemplate redisTemplate ;

    @RabbitListener(queues = "delete_sorted_from_redis")
    public void deleteSortedFromRedis(String msg , Channel channel , Message message) {

        try {

            // 就是从 Redis 中删除排队信息
            LOGGER.info("【删除排队信息开始】 ----> " + msg);
            redisTemplate.boundZSetOps("train_manager:sorted_queue").remove(msg) ;
            LOGGER.info("【删除排队信息结束】 ----> " + msg);

            // 手动应答
            channel.basicAck(message.getMessageProperties().getDeliveryTag() , false);

        } catch (IOException e) {
            e.printStackTrace();
            LOGGER.info("【删除排队信息出错】 ----> " + msg);
        }

    }
}
```

### 2.4.9 websocket 应用

#### webSocket 回顾

百度百科中针对 websocket 的介绍如下：

## WebSocket

编辑

讨论<sup>2</sup>

上传视频

本词条由“科普中国”科学百科词条编写与应用工作项目 审核。

**WebSocket**是一种在单个TCP连接上进行全双工通信的协议。WebSocket通信协议于2011年被IETF定为标准RFC 6455，并由RFC7936补充规范。WebSocket API也被W3C定为标准。

WebSocket使得客户端和服务端之间的数据交换变得更加简单，允许服务端主动向客户端推送数据。在WebSocket API中，浏览器和服务器只需要完成一次握手，两者之间就直接可以创建持久性的连接，并进行双向数据传输。

|     |           |     |                 |
|-----|-----------|-----|-----------------|
| 中文名 | WebSocket | 解 释 | 基于TCP的全双工通信协议   |
| 外文名 | WebSocket | 优 点 | 服务器可以主动传送数据给客户端 |
|     |           | 功 能 | 实现了浏览器与服务器全双工通信 |

WebSocket 协议在 2008 年诞生，2011 年成为国际标准，现在几乎所有浏览器都已经支持了。它的最大特点就是，服务器可以主动向客户端推送信息，客户端也可以主动向服务器发送信息，浏览器和服务器只需要完成一次握手，两者之间就可以创建持久性的连接，并进行双向数据传输，也就是所谓的\*\*全双工通讯\*\*的协议属于服务器推送技术的一种。

初次接触 WebSocket 的人，都会问同样的问题：我们已经有 HTTP 协议，为什么还需要另一个协议？它能带来什么好处？  
答案很简单：因为 HTTP 协议有一个缺陷：通信只能由客户端发起，HTTP 协议做不到服务器主动向客户端推送信息。

比如：没有 websocket 出现前我们获取数据是这样的

1、首先是 ajax 轮询，ajax 轮询 的原理非常简单，让浏览器隔个几秒就发送一次请求，询问服务器是否有新信息

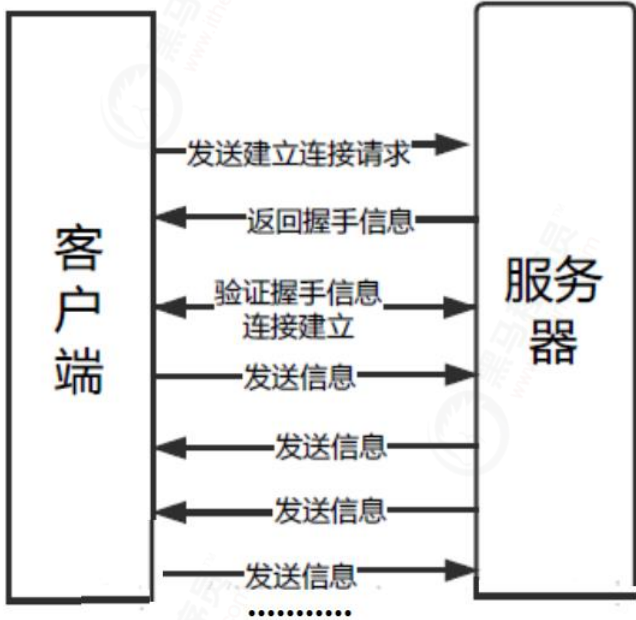
客户端：有没有新信息(Request)  
服务端：没有 (Response)  
客户端：有没有新信息(Request)  
服务端：没有。。 (Response)  
客户端：有没有新信息(Request)  
服务端：没有  
.....

2、long poll（长轮询） 其实原理跟 ajax 轮询 差不多，都是采用轮询的方式，不过采取的是阻塞模型，也就是说，客户端发起连接后，如果没消息，就一直不返回 Response 给客户端。直到有消息才返回，返回完之后，客户端再次建立连接，周而复始。

从上面可以看出其实这两种方式，都是在不断地建立 HTTP 连接，然后等待服务端处理，可以体现 HTTP 协议的另外一个特点，被动性。何为被动性呢，其实就是，服务端不能主动联系客户端，只能有客户端发起。

从上面很容易看出来，不管怎么样，上面这两种都是非常消耗资源的。

#### websocket 通信



1、发送连接请求

客户端通过一个格式为：ws://host:port/的请求地址发起 WebSocket 连接请求，并由 JavaScript 实现 WebSocket API 与服务器建立 WebSocket 连接，其中 host 为服务器主机 IP 地址或域名，port 为端口。

## 2、握手

当服务器收到请求后，会解析请求头信息，根据升级后的协议判断该请求为 WebSocket 请求，并取出请求信息中的 Sec-WebSocket-Key 字段的数据按照某种算法重新生成一个新的字符串序列放入响应头 Sec-WebSocket-Accept 中。

Sec-WebSocket-Accept: 服务器接受客户端 HTTP 协议升级的证明

## 4、WebSocket 链接建立

WebSocket 建立连接客户端接收服务器的响应后，同样会解析请求头信息，取出请求信息中的 Sec-WebSocket-Accept 字段，并用服务器内部处理 Sec-WebSocket-Key 字段的算法处理之前发送的 Sec-WebSocket-Key，把处理得到的结果与 Sec-WebSocket-Accept 进行对比，数据相同则表示客户端与服务器成功建立 WebSocket 连接，反之失败

### 服务端代码实现

具体的实现步骤，如下所示:

#### 1、引入起步依赖

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-websocket</artifactId>
</dependency>
```

#### 2、开启 WebSocket 支持端点

```
@Configuration
public class WebSocketConfig {

    @Bean
    public ServerEndpointExporter serverEndpointExporter() {
        return new ServerEndpointExporter();
    }

}
```

#### 3、创建 server 核心类

因为 WebSocket 是类似客户端服务端的形式（采用 ws 协议），那么这里的 WebSocketServer 其实就相当于一个 ws 协议的 Server 直接通过 @ServerEndpoint("/im/{userId}")、@Component 启用即可，然后在里面实现 @OnOpen 开启连接，@onClose 关闭连接，@onMessage 接收消息等方法。

```
@Component
@ServerEndpoint(value = "/im/{userId}")
public class WebSocketServer {

    // 创建一个 HashMap 集合，通过该集合来存储 session 信息
    private static HashMap<String, Session> sessionHashMap = new HashMap<>();

    // 创建一个变量，用来记录 userId
    private String userId = null;

    /**
     * 当前客户端和服务端建立了以后，在这里会执行该方法
     * @param session
     * @param userId
     */
    @OnOpen
    public void onOpen(Session session, @PathParam(value = "userId") String userId) {

        // 把userId赋值给成员的userId
        this.userId = userId;

        // 就是把我们的 session 信息存储起来，后期我们在进行消息推送的时候，其实就是通过 session 对象中的方法完成消息的推送
        if(sessionHashMap.containsKey(userId)) {
            sessionHashMap.remove(userId);
        }

        // 进行保存
        sessionHashMap.put(userId, session);
    }

    @OnClose
    public void onClose() {
        if(sessionHashMap.containsKey(userId)) {
            sessionHashMap.remove(userId);
        }
    }

    // 消息的推送
```

```
public void sendMessageToUser(String userId , String message) {
    try {
        Session session = sessionHashMap.get(userId);
        session.getBasicRemote().sendText(message);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

#### 4、修改 OrderHandlerConsumer 类

保存完订单数据以后，调用 WebSocketServer 中的 sendMessageToUser 方法完成消息推送。

```
// 进行消息的推送
public void sendOrderInfo(String userId , boolean result , String orderId) {
    HashMap<String , Object> resultHashMap = new HashMap<>() ;
    resultHashMap.put("result" , result) ;
    resultHashMap.put("orderId" , orderId) ;
    websocketServer.sendMessageToUser(userId , JSON.toJSONString(resultHashMap));
}
```

#### 5、Nginx 配置虚拟主机

# 订单处理服务虚拟主机

```
server {

    listen      80;

    server_name www.trainmanager.order.handler.com;

    access_log logs/host.access.order.handler.log main ;           # nginx 访问日志

    location / {

        # limit_req zone=rateLimit burst=5 nodelay;

        # limit_conn perip 1 ;

        # limit_conn perserver 100 ;

        proxy_pass http://train-manager-order-handler ;

        proxy_http_version 1.1;

        proxy_read_timeout 6000 ;                                # 设置超过时间，客户端和服务器建立连接了以后，默认超过 60s 没有数据交互，nginx 会自动将连接断开

        proxy_set_header Upgrade $http_upgrade;

        proxy_set_header Connection "upgrade";

    }

    error_page 500 502 503 504 /50x.html;

    location = /50x.html {

        root html;

    }

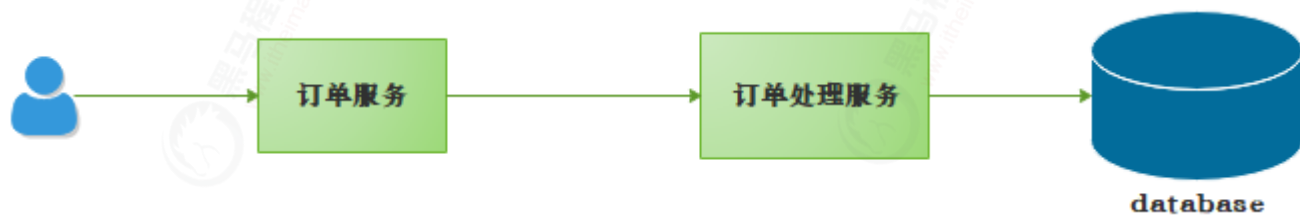
}
```

### 2.4.10 查询订单接口

#### 查询订单架构介绍

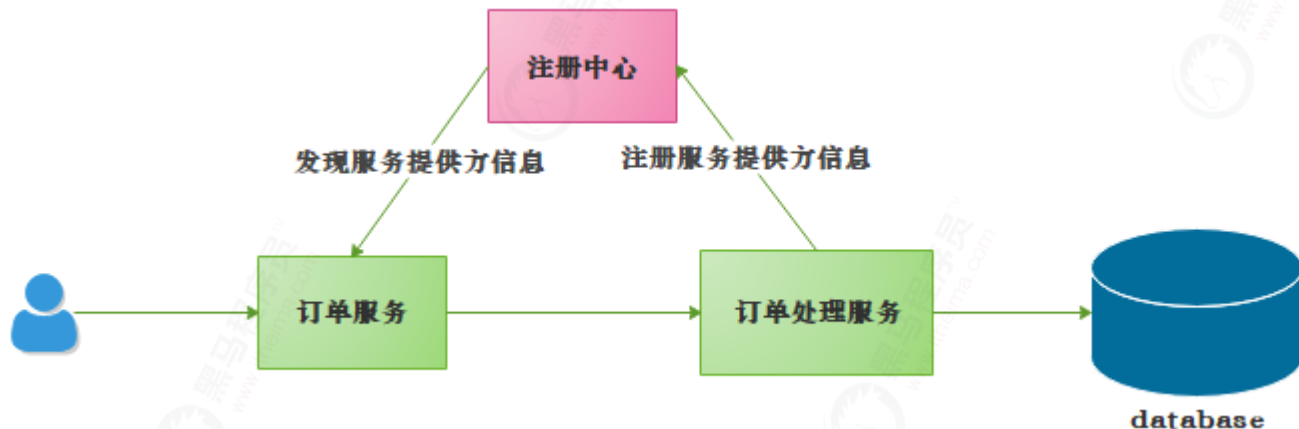
通过分析前端页面，我们可以看到在进行订单查询的时候调用的其实是 itheima-train-manager-order 项目中的接口。而我们真正的订单信息是存储在数据库中的，而和数据库打交道的是 itheima-train-manager-order-handler 工程，因此我们需要在 itheima-train-manager-order-handler 工程中提供一个查询接口，供我们的 itheima-train-manager-order 来进行调用。

最初的调用架构：



弊端：订单处理服务进行扩容或者缩容的时候，我们都需要对订单服务做相应的更改

优化以后的调用架构：



本次我们还是选用 nacos 作为我们的服务注册中心。

## 服务提供方

### 集成 Nacos

服务提供方集成 nacos 的具体步骤如下所示：

1、在 pom.xml 文件中添加如下依赖

```
<!-- nacos 注册中心依赖 -->
<dependency>
  <groupId>com.alibaba.cloud</groupId>
  <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
</dependency>
```

2、在 nacos 配置中心中添加如下配置信息

```
spring:
  cloud:
    nacos:
      discovery:
        server-addr: 172.17.0.224:8848,172.17.0.224:8849,172.17.0.224:8850
```

3、在启动类上添加如下注解开启服务注册于发现功能

```
@EnableDiscoveryClient
```

4、启动服务，就可以在 nacos 管理后台查看到对应的服务信息

NACOS 1.2.1

配置管理

服务管理

服务列表

订阅者列表

集群管理

权限控制

命名空间

public

服务列表 | public

服务名称:

分组名称:

隐藏空服务: ☒

查询

创建服务

| 服务名                                 | 分组名称          | 集群数目 | 实例数 | 健康实例数 | 触发保护阈值 | 操作                                                             |
|-------------------------------------|---------------|------|-----|-------|--------|----------------------------------------------------------------|
| itheima-train-manager-order-handler | DEFAULT_GROUP | 1    | 1   | 1     | false  | <a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">删除</a> |

### 定义查询接口

具体步骤如下所示：

1、定义 OrderHandlerController

```
@RestController
@RequestMapping(value = "/orderHandler")
public class OrderHandlerController {

    @Autowired
    private TbOrderService orderService ;

    @RequestMapping(value = "/queryById/{orderId}")
    public Order queryById(@PathVariable(value = "orderId") Long orderId) {
        return orderService.queryOrderById(orderId) ;
    }
}
```

2、OrderService 定义查询方法

```
// 就是根据 id 查询订单信息
public Order queryOrderById(Long orderId) {
    return tbOrderMapper.queryOrderById(orderId);
}
```

3、TbOrderMapper 定义根据 id 查询方法

```
public abstract Order queryOrderById(Long orderId);
```

```
<!-- 根据订单的 id 查询订单信息 -->
<select id="queryOrderById" parameterType="java.lang.Long" resultMap="BASE_RESULT_MAP">
    select * from tb order where id = #{value}
</select>
```

4、使用 postman 进行测试

## 服务调用方

### 集成 Nacos

服务调用方集成 Nacos 和服务提供方集成 nacos 方式一样，此处不再累述。

### 远程调用

1、定义根据 id 查询的接口

```
@RequestMapping(value = "/queryById/{orderId}")
public OrderHandler queryById(@PathVariable(value = "orderId") Long orderId) {
    return orderService.queryById(orderId);
}
```

2、OrderService 查询方法定义

```
// 根据订单的 id 去获取订单的详情信息
public OrderHandler queryById(Long orderId) {

    // 远程调用获取订单数据
    Order order = null ;

    // 从 Redis 中获取用户信息
    User user = JSON.parseObject(redisTemplate.boundHashOps("train_manager:userInfo").get(order.getUserId()).toString(), User.class);

    // 创建 OrderHandler 对象
    OrderHandler orderHandler = new OrderHandler();
    orderHandler.setOrder(order);
    orderHandler.setUser(user);

    // 返回
    return orderHandler ;
}
```

3、进行远程调用

## Feign 基本使用

由于 itheima-train-manager-order-handler 提供的是一个 http 接口。因此要调用该接口，我们就需要使用一些 Http 的客户端工具。比如 HttpClient，RestTemplate 等。要想进行负载均衡的调用，我们可以使用 Spring Cloud 中的 Ribbon 组件实现。本次我们在进行远程调用的时候，我们将会使用 Spring Cloud 中的另外一个组件：Feign

Feign 是一个声明式的伪 RPC 的 REST 客户端，它用了基于接口的注解方式，很方便实现客户端配置。将 Java Http 客户端绑定到它的内部，并且在内部集成了 Ribbon 组件，Feign 的首要目标是将 Java Http 客户端调用过程变得简单。

具体的使用步骤如下所示：

1、在 pom.xml 文件中添加如下依赖

```
<!--Feign 依赖加入-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

2、定义远程调用接口

```
@FeignClient(name = "itheima-train-manager-order-handler")
public interface OrderFeignClient {

    @RequestMapping(value = "/orderHandler/queryById/{orderId}")
    public abstract Order queryById(@PathVariable(value = "orderId") Long orderId) ;
}
```



```
}
```

3、启动类上添加如下注解，开启 Feign 的支持

```
@EnableFeignClients
```

4、进行远程调用

```
// 根据订单的 id 去获取订单的详情信息
public OrderHandler queryById(Long orderId) {

    // 远程调用获取订单数据
    Order order = orderFeignClient.queryById(orderId) ;
    ...

    // 返回
    return orderHandler ;
}
```

#### 排队方案测试

为了看到明显的测试效果，我们可以在订单处理服务中加入线程休眠时间，模拟业务的处理时长。更改页面的用户 id 开启多个页面进行测试。

## 2.5 下单优化

### 2.5.1 问题描述

现在的下单看似已经完成了，但是还是存在一些小问题，比如：

- 1、如果该用户已经下单了，我们还需要进行预扣库存操作吗？
- 2、用户虽然已经下单了，但是一直没有对订单进行支付。
- 3、进行预扣减库存的时候存在线程安全问题。

### 2.5.2 预扣库存优化

当该用户已经购买了指定列车的火车票，那么我们就不能再进行预扣库存操作了。

实现思路如下所示：

- 1、在订单处理服务中提供一个查询接口（根据用户的 id ， 已经乘车计划 id），查询订单信息
- 2、在订单服务中，在进行预扣库存之前远程调用接口，查询订单信息
- 3、修改前端页面针对结果是 false 的处理情况

#### 订单处理服务

查询接口如下所示：

OrderHandlerController 添加查询接口

```
@RequestMapping(value = "/queryOrderInfo/{userId}/{trainNum}/{trainRidingDate}")
public Order queryOrderInfo(@PathVariable(value = "userId") String userId, @PathVariable(value = "trainNum")
String trainNum , @PathVariable(value = "trainRidingDate")String trainRidingDate) {
    return orderService.queryOrderInfo(userId , trainNum , trainRidingDate) ;
}
```

OrderService 添加查询方法

```
// 查询订单的详情信息
public Order queryOrderInfo(String userId, String trainNum, String trainRidingDate) {

    // 查询数据
    HashMap hashMap = new HashMap() ;
    hashMap.put("userId" , userId) ;
    hashMap.put("trainNum" , trainNum) ;
    hashMap.put("trainRidingDate" , trainRidingDate) ;
    Order dbOrder = tbOrderMapper.queryOrderInfo(hashMap) ;

    // 返回
    return dbOrder ;
}
```

#### 订单服务远程调用

OrderFeignClient 添加查询接口

```
@RequestMapping(value = "/orderHandler/queryOrderInfo/{userId}/{trainNum}/{trainRidingDate}")  
public abstract Order queryOrderInfo(@PathVariable(value = "userId") String userId , @PathVariable(value = "trainNum") String trainNum , @PathVariable(value = "trainRidingDate") String trainRidingDate) ;
```

OrderService 添加远程调用逻辑

```
// 进行订单详情信息的查询  
Order dbOrder = orderFeignClient.queryOrderInfo(orderParams.getUserId(), trainNum, trainRidingDate);  
if(dbOrder != null) {  
    responseResult.setResult(false);  
    responseResult.setMessage("1");  
    responseResult.setData(String.valueOf(dbOrder.getId()));  
    return responseResult ;  
}  
  
// 获取指定日期的乘车计划数据，并将其封装到一个 TbRidingPlanDate 对象中  
Map<Object, Object> ridingPlanDateHashMap = redisTemplate.boundHashOps(ridingPlanDateKey).entries();  
TbRidingPlanDate tbRidingPlanDate = JSON.parseObject(JSON.toJSONString(ridingPlanDateHashMap),  
TbRidingPlanDate.class);
```

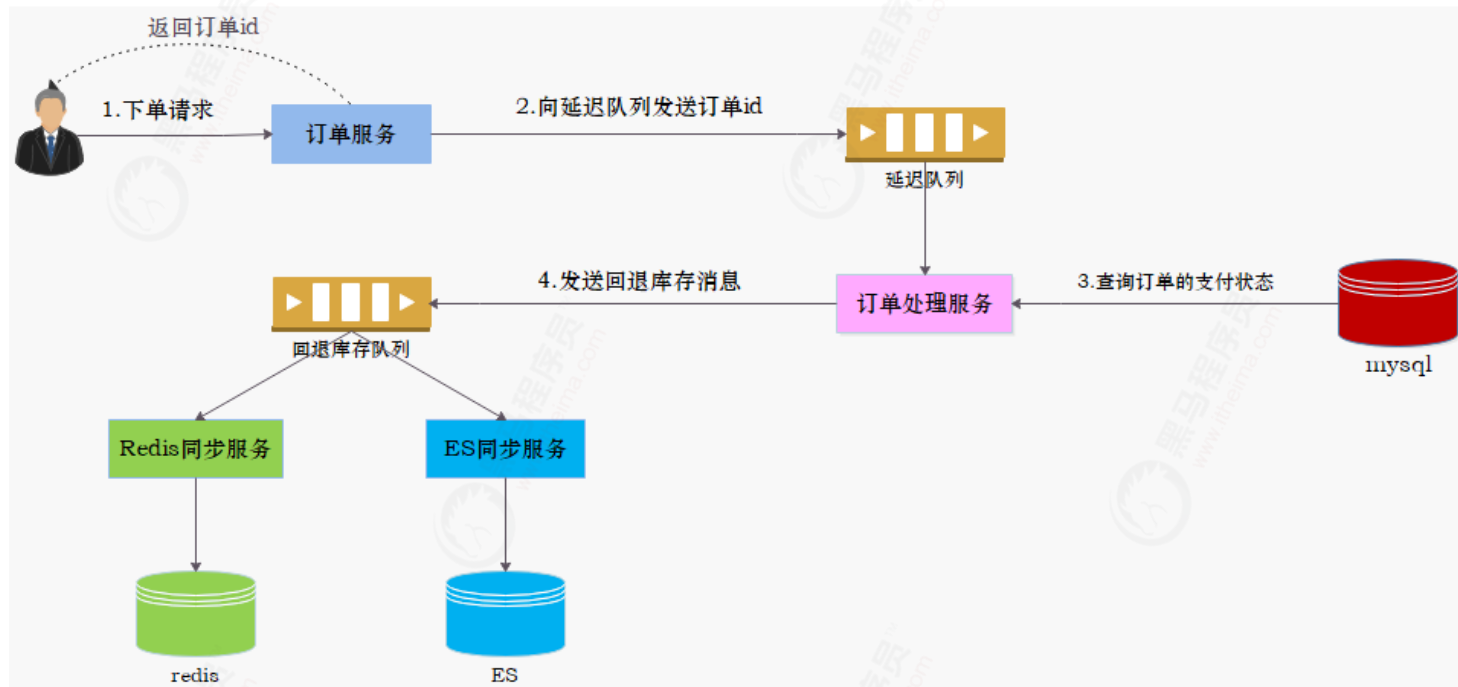
前端页面修改

```
axios.post("http://www.trainmanager.order.com/order/submitOrder"  
this.orderParams).then(function(response) {  
  
    if(response.data.result) { // 下单排队成功，跳转到订单查询页面  
  
        window.location.href="http://www.trainmanager.com/otn/payOrder/preOrderQueue.html?ridingPlanId=" +  
this.orderParams.ridingPlanId + "&trainNum=" + this.orderParams.trainNum + "&trainRidingDate=" +  
_this.orderParams.trainRidingDate  
  
        + "&userId=" + _this.orderParams.userId + "&ticketType=" + _this.orderParams.ticketType;  
    }else {  
  
        if(response.data.message == "1") {  
  
            window.location.href = "http://www.trainmanager.com/otn/payOrder/init.html?orderId=" +  
response.data.data ;  
  
        }else if(response.data.message == "0"){  
  
            window.location.href="/otn/payOrder/genOrderFail.html" ;  
  
        }  
  
    }  
  
});
```

### 2.5.3 回退库存

#### 延迟队列

延迟队列存储的对象是对应的延迟消息，所谓"延迟消息"是指当消息被发送以后，并不想让消费者立刻拿到消息，而是等待特定时间后，消费者才能拿到这个消息进行消费。而我们的在完成订单未支付超时这个需求的时候就可以去使用延迟队列。那么我们首先来给大家介绍一下具体的实现流程，如下图所示：



#### 死信队列

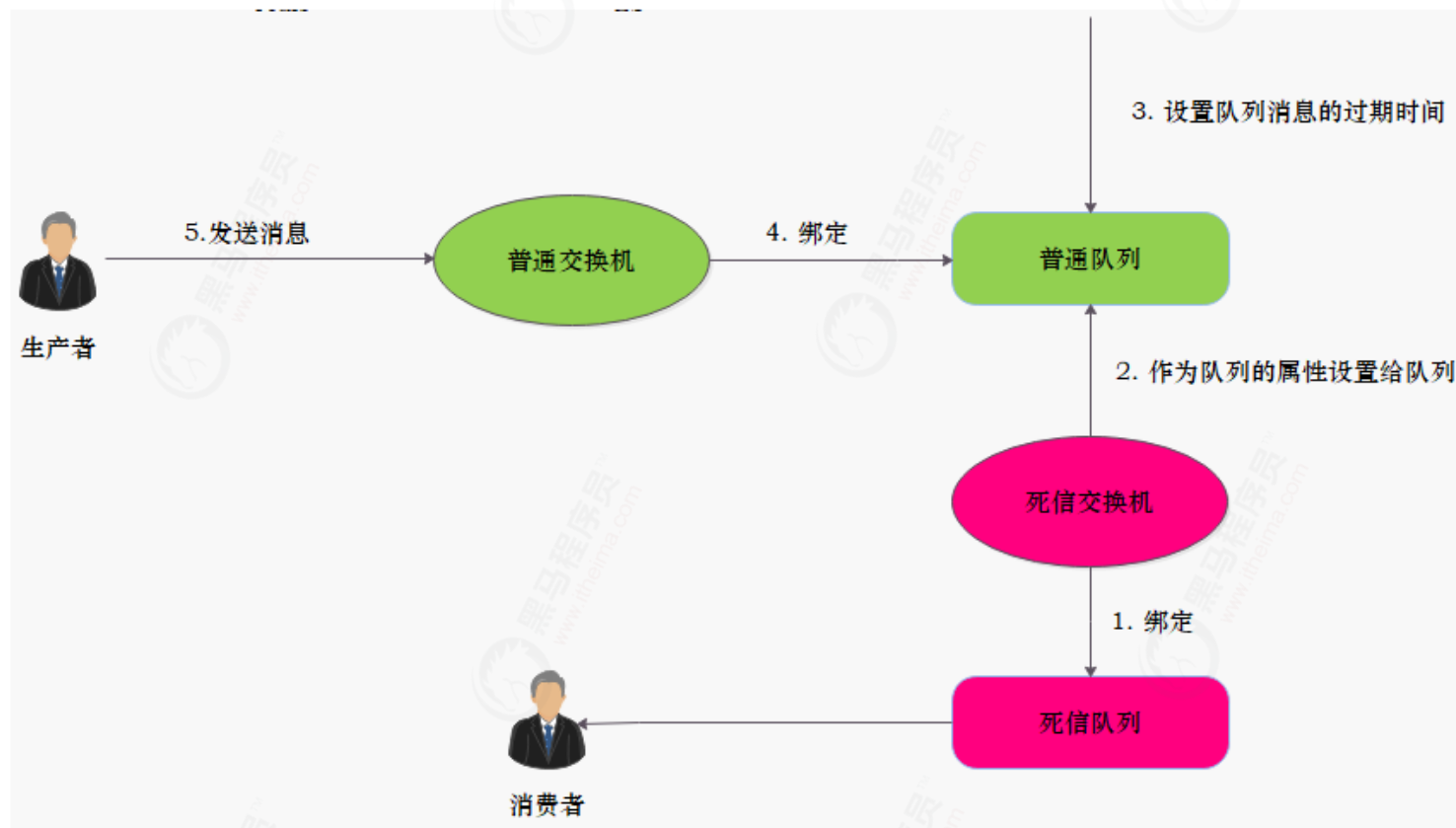
在 Rabbitmq 中没有延迟队列的功能，但是我们可以使用 DLX 结合 TTL 来实现延迟队列的功能。

DLX：可以被称之为死信交换机（Dead-Letter-Exchange），也有人将其称之为死信邮箱。当消息在一个队列中变成死信之后，这个消息能被重新发送到另外一个交换机，这个交换机就是 DLX，与之绑定的队列就是死信队列。

一个消息成为死信的情况：

- 1、消息被拒绝，并且设置 requeue 参数为 false
- 2、消息过期
- 3、队列达到最大长度

如下下图所示：



#### 发送延迟消息

接下来我们就来演示一下使用死信队列模拟延迟队列。

#### 订单服务处理服务

- 1、在（itheima-train-manager-order-handler）RabbitmqConfiguration 中添加死信队列的与死信交换机的配置

```
// 声明回退库存的死信交换机
@Bean(name = "order_timeout_dlx_ex")
public Exchange backStockDlxExchange() {
    return ExchangeBuilder.directExchange("order_timeout_dlx_ex").durable(true).build();
}

// 声明回退库存的死信队列
@Bean(name = "order_timeout_dlx_queue")
public Queue backStockDlxQueue() {
    return QueueBuilder.durable("order_timeout_dlx_queue").build();
}

// 声明交换机和队列的绑定关系
@Bean
public Binding backStockDlxExBindBackStockDlxQueue(@Qualifier(value = "order_timeout_dlx_ex") Exchange exchange, @Qualifier(value = "order_timeout_dlx_queue") Queue queue) {
    return BindingBuilder.bind(queue).to(exchange).with("order_timeout").noargs();
}
```

- 2、在 RabbitmqConfiguration 中添加延迟队列信息

```
// 声明回退库存的队列
@Bean(name = "order_timeout_queue")
public Queue orderTimeOutQueue() {
    QueueBuilder queueBuilder = QueueBuilder.durable("order_timeout_queue"); // 订单队列建造者对象
    queueBuilder.ttl(1000 * 60); // 单位为毫秒
    queueBuilder.deadLetterExchange("order_timeout_dlx_ex"); // 设置死信交换机
    queueBuilder.deadLetterRoutingKey("order_timeout"); // 死信交换机路由键
    return queueBuilder.build();
}

// 声明交换机和队列的绑定关系
@Bean
public Binding trainExBindOrderTimeOutQueue(@Qualifier(value = "train_manager_ex") Exchange exchange, @Qualifier(value = "order_timeout_queue") Queue queue) {
    return BindingBuilder.bind(queue).to(exchange).with("order_timeout").noargs();
}
```

}

## 3、发送订单延迟消息

```
// 发送订单数据
rabbitmqProducer.sendMessage(JSON.toJSONString(orderHandler) , "gen_order");

// 发送延迟消息到MQ中
LOGGER.info("【发送延迟消息开始了】 ---> " + JSON.toJSONString(order));
HashMap<String , Long> msg = new HashMap<>() ;
msg.put("orderId" , order.getId()) ;
rabbitmqProducer.sendMessage(JSON.toJSONString(msg) , "order_timeout");
LOGGER.info("【发送延迟消息结束了】 ---> " + JSON.toJSONString(order));
```

## 4、在 OrderHandlerConsumer 类中添加监听队列方法

```
@RabbitListener(queues = "order_timeout_dlx_queue")
public void orderTime(String orderIdJson , Message message , Channel channel ) {

    try {

        LOGGER.info("【获取到订单超时消息】 ----> " + orderIdJson);
        HashMap<String , Long> hashMap = JSON.parseObject(orderIdJson, HashMap.class);
        Long orderId = hashMap.get("orderId");

        // 查询数据库信息
        ...
        // 给出应答
        channel.basicAck(message.getMessageProperties().getDeliveryTag() , false);

    } catch (IOException e) {
        e.printStackTrace();
    }

}
```

## 生产者代码

生产者代码实现思路:

- 1、根据 id 查询订单数据
- 2、判断订单的支付状态
- 3、如果是未支付，修改订单的有效状态为无效
- 4、并且向 mq 中发送回退库存消息

具体是实现步骤如下所示:

- 1、定义根据 id 修改订单信息方法

```
// 修改订单信息
public abstract void updateOrderById(Order order) ;
```

## 2、Mapper 映射文件内容

```
<!-- 根据 id 进行修改 -->
<select id="updateOrderById" parameterType="com.itheima.train.manager.domain.Order">
    update tb_order set
        riding plan date id = #{ridingPlanDateId} ,
        start station name = #{startStationName} ,
        receive station name = #{receiveStationName},
        train num = #{trainNum} ,
        train riding date = #{trainRidingDate} ,
        start time = #{startTime} ,
        end time = #{endTime} ,
        seat num = #{seatNum} ,
        seat nature = #{seatNature} ,
        coach num = #{coachNum} ,
        seat price = #{seatPrice} ,
        pay status = #{payStatus} ,
        user id = #{userId} ,
        is effect = #{isEffect}
    where id = #{id}
</select>
```

## 3、OrderService 代码

```
// 就是根据 id 进行修改
public void updateOrderById(Order order) {
    tbOrderMapper.updateOrderById(order);
}
```

## 4、OrderHandlerConsumer 中 orderTime 修改

```
@RabbitListener(queues = "order timeout dlx queue")
public void orderTime(String orderIdJson , Message message , Channel channel ) {

    try {

        LOGGER.info("【获取到订单超时消息】 ----> " + orderIdJson);
        HashMap<String , Long> hashMap = JSON.parseObject(orderIdJson, HashMap.class);
        Long orderId = hashMap.get("orderId");

        // 查询数据库信息
        Order order = orderService.queryOrderById(orderId);
        if("0".equals(order.getPayStatus())) {

            // 修改订单的有效状态为为无效
            order.setIsEffect("0");

            // 数据更新
            orderService.updateOrderById(order);

            // 发送订单回退库存消息到mq 中
            LOGGER.info("【订单超时消息成功获取，回退库存消息发送成功】 ---> " + orderIdJson);

        }

        // 给出应答
        channel.basicAck(message.getMessageProperties().getDeliveryTag() , false);

    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

5、在 RabbitmqConfiguration 添加回退库存队列信息  
回退库存的队列信息如下图所示：



```
// 声明队列
@Bean(name = "back_stock_redis_queue")
public Queue backStockRedisQueue() {
    return QueueBuilder.durable("back_stock_redis_queue").build() ;
}

// 声明队列
@Bean(name = "back_stock_es_queue")
public Queue backStockESQueue() {
    return QueueBuilder.durable("back_stock_es_queue").build() ;
}

// 声明交换机和队列的绑定关系
@Bean
public Binding trainExBindBackStockRedisQueue(@Qualifier(value = "train_manager_ex") Exchange exchange ,
@Qualifier(value = "back_stock_redis_queue") Queue queue) {
    return BindingBuilder.bind(queue).to(exchange).with("back_stock").noargs() ;
}

// 声明交换机和队列的绑定关系
@Bean
public Binding trainExBindBackStockESQueue(@Qualifier(value = "train_manager_ex") Exchange exchange ,
@Qualifier(value = "back_stock_es_queue") Queue queue) {
    return BindingBuilder.bind(queue).to(exchange).with("back_stock").noargs() ;
}
}
```

6、发送回退库存队列消息

```
if("0".equals(order.getPayStatus())) {

    // 修改订单的有效状态为为无效
    order.setIsEffect("0");

    // 数据更新
    orderService.updateOrderById(order);

    // 发送订单回退库存消息到mq 中
    rabbitmqProducer.sendMessage(JSON.toJSONString(order) , "back_stock");
    LOGGER.info("【订单超时消息成功获取，回退库存消息发送成功】 ----> " + orderIdJson);

}
```

## 消费者代码

## Redis 库存回退

1、在 itheima-train-manager-redis 的 RabbitmqConfiguration 添加回退库存队列信息

```
// 声明队列
@Bean(name = "back_stock_redis_queue")
public Queue backStockRedisQueue() {
    return QueueBuilder.durable("back_stock_redis_queue").build() ;
}

// 声明交换机和队列的绑定关系
@Bean
public Binding trainExBindBackStockRedisQueue(@Qualifier(value = "train_manager_ex") Exchange exchange ,
@Qualifier(value = "back_stock_redis_queue") Queue queue) {
    return BindingBuilder.bind(queue).to(exchange).with("back_stock").noargs() ;
}
```

2、给 RedisConsumer 添加回退库存方法

```
@RabbitListener(queues = "back_stock_redis_queue")
public void backStock(String orderJson , Message message , Channel channel) {

    try {

        // 库存回退
        logger.error("【Redis 回退库存开始了】 ---> " + orderJson);
        Order order = JSON.parseObject(orderJson , Order.class) ;
        String stockCountKeyDynamic = "riding_plan:" + order.getTrainNum() + ":" + order.getTrainRidingDate()
+ ":" ;
        Set<String> keys = redisTemplate.keys(stockCountKeyDynamic);
        for(String key : keys) {
            BoundHashOperations<String, Object, Object> boundHashOperations
= redisTemplate.boundHashOps(key);
            switch (order.getSeatNature()) {
                case "3" :
                    int firstSeatStock
= Integer.parseInt(boundHashOperations.get("firstSeatStock").toString());
                    boundHashOperations.put("firstSeatStock" , String.valueOf(firstSeatStock + 1));
                    break;
                case "4" :
                    int secondSeatStock
= Integer.parseInt(boundHashOperations.get("secondSeatStock").toString()) ;
                    boundHashOperations.put("secondSeatStock" , String.valueOf(secondSeatStock + 1));
                    break ;
                case "7" :
                    int hardSleeperStock
= Integer.parseInt(boundHashOperations.get("hardSleeperStock").toString()) ;
                    boundHashOperations.put("hardSleeperStock" , String.valueOf(hardSleeperStock + 1));
                    break ;
                case "6" :
                    int softSleeperStock
= Integer.parseInt(boundHashOperations.get("softSleeperStock").toString()) ;
                    boundHashOperations.put("softSleeperStock" , String.valueOf(softSleeperStock + 1));
                    break ;
                case "2" :
                    int busSeatStock = Integer.parseInt(boundHashOperations.get("busSeatStock").toString()) ;
                    boundHashOperations.put("busSeatStock" , String.valueOf(busSeatStock + 1));
                    break ;
                case "5" :
                    int hardSeatStock
= Integer.parseInt(boundHashOperations.get("hardSeatStock").toString()) ;
                    boundHashOperations.put("hardSeatStock" , String.valueOf(hardSeatStock + 1));
                    break ;
                case "9" :
                    int softSeatStock
= Integer.parseInt(boundHashOperations.get("softSeatStock").toString()) ;
                    boundHashOperations.put("softSeatStock" , String.valueOf(softSeatStock + 1));
                    break ;
            }
        }

        logger.info("【Redis 回退库存成功了】 ---> " + orderJson);

        // 释放座位
        logger.info("【Redis 释放座位开始了】 ---> " + orderJson);
        String seatNatureKey = "train_seat:" + order.getTrainNum() + ":" + order.getTrainRidingDate() + ":"
+ order.getSeatNature() + ":" + order.getCoachNum() ;
        redisTemplate.boundHashOps(seatNatureKey).put(order.getSeatNum() , "0");
        logger.info("【Redis 释放座位成功了】 ---> " + orderJson);

        // 给出应答
        channel.basicAck(message.getMessageProperties().getDeliveryTag() , false);

    } catch (IOException e) {
```

```
e.printStackTrace();
logger.error("【Redis 回退库存失败】 ---> " + orderJson);
}
}
```

### ES 库存回退

1、在 itheima-train-manager-es 的 RabbitmqConfiguration 添加回退库存队列信息

```
// 声明队列
@Bean(name = "back_stock_es_queue")
public Queue backStockESQueue() {
    return QueueBuilder.durable("back_stock_es_queue").build();
}

// 声明交换机和队列的绑定关系
@Bean
public Binding trainExBindBackStockESQueue(@Qualifier(value = "train_manager_ex") Exchange exchange,
@Qualifier(value = "back_stock_es_queue") Queue queue) {
    return BindingBuilder.bind(queue).to(exchange).with("back_stock").noargs();
}
```

2、给 EsConsumer 添加回退库存方法

```
// 库存的回退
public void backStock(Order order) throws IOException {

    // 创建搜索请求对象
    SearchRequest searchRequest = new SearchRequest("riding_plan_date");
    SearchSourceBuilder searchSourceBuilder = new SearchSourceBuilder().trackTotalHits(true); // 查询所有满足条件的数据
    BoolQueryBuilder boolQueryBuilder = QueryBuilders.boolQuery();
    boolQueryBuilder.must(QueryBuilders.termQuery("trainNum", order.getTrainNum()));
    boolQueryBuilder.must(QueryBuilders.termQuery("trainRidingDate", order.getTrainRidingDate()));
    searchSourceBuilder.query(boolQueryBuilder);
    searchSourceBuilder.size(1000);
    searchRequest.source(searchSourceBuilder);

    // 进行搜索
    SearchResponse searchResponse = restHighLevelClient.search(searchRequest, RequestOptions.DEFAULT);
    SearchHits searchHits = searchResponse.getHits();
    SearchHit[] hits = searchHits.getHits();
    for(SearchHit searchHit : hits) {
        String sourceAsString = searchHit.getSourceAsString();
        TbRidingPlanDate ridingPlanDate = JSON.parseObject(sourceAsString, TbRidingPlanDate.class);
        switch (order.getSeatNature()) {
            case "3":
                ridingPlanDate.setFirstSeatStock(ridingPlanDate.getFirstSeatStock() + 1);
                break;
            case "4":
                ridingPlanDate.setSecondSeatStock(ridingPlanDate.getSecondSeatStock() + 1);
                break;
            case "7":
                ridingPlanDate.setHardSleeperStock(ridingPlanDate.getHardSleeperStock() + 1);
                break;
            case "6":
                ridingPlanDate.setSoftSleeperStock(ridingPlanDate.getSoftSleeperStock() + 1);
                break;
            case "2":
                ridingPlanDate.setBusSeatStock(ridingPlanDate.getBusSeatStock() + 1);
                break;
            case "5":
                ridingPlanDate.setHardSeatStock(ridingPlanDate.getHardSeatStock() + 1);
                break;
            case "9":
                ridingPlanDate.setSoftSeatStock(ridingPlanDate.getSoftSeatStock() + 1);
                break;
        }

        // 创建一个更新的请求对象
        UpdateRequest updateRequest = new UpdateRequest("riding_plan_date",
searchHit.getId()).doc(JSON.toJSONString(ridingPlanDate), XContentType.JSON);
        restHighLevelClient.update(updateRequest, RequestOptions.DEFAULT);
    }
}
```

## 2.5.4 分布式锁

### 分布式锁简介

现在的下单预扣减库存的代码还存在一个比较严重的问题-线程安全问题。

```
// 构建乘车计划的 key
LOGGER.info("【预扣库存开始了】 ---> " + JSON.toJSONString(orderParams));
```

```
String ridingPlanDateKey = "riding plan:" + trainNum + ":" + trainRidingDate + ":" + ridingPlanId ;
int count = Integer.parseInt(redisTemplate.boundHashOps(ridingPlanDateKey).get(redisTicketTypeKey).toString());
if(count <= 0 ) {
    responseResult.setResult(false);
    responseResult.setMessage("0"); // 表示票已经售罄了
    return responseResult ;
}
...
// 动态库存扣减
String stockCountKeyDynamic = "riding plan:" + trainNum + ":" + trainRidingDate + ":*" ;
Set<String> keys = redisTemplate.keys(stockCountKeyDynamic);
for(String key : keys) {
    redisTemplate.boundHashOps(key).put(redisTicketTypeKey , String.valueOf(count - 1));
}
LOGGER.info("【预扣库存结束了】 ---> " + JSON.toJSONString(orderParams));
```

在我们进行单机应用开发，涉及并发同步的时候，我们往往采用 `synchronized` 或者 `Lock` 的方式来解决多线程间的代码同步问题，这时多线程的运行都是在同一个 JVM 之下。但当我们的应用是分布式集群工作的情况下，属于多 JVM 下的工作环境，JVM 之间已经无法通过多线程的锁解决同步问题。那么就需要一种更加高级的锁机制，来处理种跨机器的进程之间的数据同步问题——这就是分布式锁。

### 分布式锁实现方式

分布式锁的实现方式比较多：

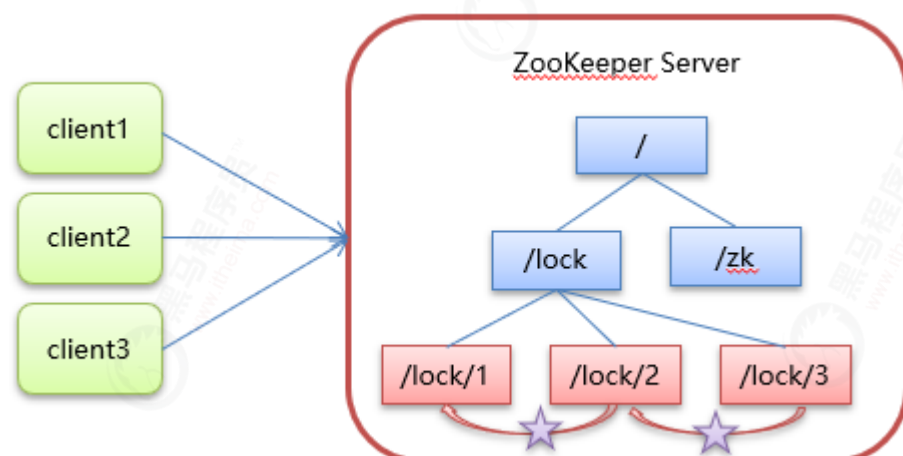


本次我们重点介绍的就是使用 `zookeeper` 结合它的客户端 API（`Curator`）实现分布式锁。

### Zookeeper 实现分布式锁原理

**核心理想：**当客户端要获取锁，则创建节点，使用完锁，则删除该节点。

- 1、客户端获取锁时，在 `lock` 节点下创建临时顺序节点。
- 2、然后获取 `lock` 下面的所有子节点，客户端获取到所有的子节点之后，如果发现自己创建的子节点序号最小，那么就认为该客户端获取到了锁使用完锁后，将该节点删除。
- 3、如果发现自己创建的节点并非 `lock` 所有子节点中最小的，说明自己还没有获取到锁，此时客户端需要找到比自己小的那个节点，同时对其注册事件监听器，监听删除事件。
- 4、如果发现比自己小的那个节点被删除，则客户端的 `Watcher` 会收到相应通知，此时再次判断自己创建的节点是否是 `lock` 子节点中序号最小的，如果是则获取到了锁，如果不是则重复以上步骤继续获取到比自己小的一个节点并注册监听。



### 分布式锁实现

#### Curator 简介

针对 `zookeeper` 实现分布式锁，我们没有必须在自己编写代码进行实现了。`zookeeper` 的 Java 客户端 `Curator` 已经提供了分布式锁的实现。`zookeeper` 的原生 api 相对来说比较繁琐，比如：对节点添加监听事件，当监听触发后，我们需要再次手动添加监听，否则监听只生效一次；再比如，断线重连也需要我们手动代码来判断处理等。

`Curator` 是 `Netflix` 开源的一套 `ZooKeeper` 客户端框架，用它来操作 `zookeeper` 更加简单方便。

`Curator` 框架提供了一套高级的 API，简化了 `ZooKeeper` 的操作。它增加了很多使用 `ZooKeeper` 开发的特性，可以处理 `ZooKeeper` 集群复杂的连接管理和重试机制。目前已经成为 `Apache` 的顶级项目。另外还提供了一套易用性和可读性更强的 `Fluent` 风格的客户端 API 框架。

下面是官网 <http://curator.apache.org/> 对 `Curator` 的描述：

## Welcome to Apache Curator

### What is Curator?

Curator *n* 'kyoor-ä-ter: a keeper or custodian of a museum or other collection - A ZooKeeper Keeper.

Apache Curator is a Java/JVM client library for Apache ZooKeeper, a distributed coordination service. It includes a highlevel API framework and utilities to make using Apache ZooKeeper much easier and more reliable. It also includes recipes for common use cases and extensions such as service discovery and a Java 8 asynchronous DSL.

"Guava is to Java what Curator is to ZooKeeper"  
Patrick Hunt, ZooKeeper committer

在 Curator 中有五种锁方案，主要包含：

InterProcessMutex：分布式可重入排它锁

InterProcessSemaphoreMutex：分布式排它锁（非可重入锁）

InterProcessReadWriteLock：分布式读写锁

InterProcessMultiLock：将多个锁作为单个实体管理的容器

InterProcessSemaphoreV2：共享信号量

一般情况下我们使用的都是 **InterProcessMutex**。

### curator 分布式锁实现

具体的实现步骤如下所示：

1、导入相关的依赖包

```
<dependency>
  <groupId>org.apache.curator</groupId>
  <artifactId>curator-framework</artifactId>
  <version>4.0.0</version>
</dependency>

<dependency>
  <groupId>org.apache.curator</groupId>
  <artifactId>curator-recipes</artifactId>
  <version>4.0.0</version>
</dependency>
```

2、在 nacos 配置中心中添加 zookeeper 集群节点的配置

```
zk:
  cluster:
    nodes: 127.0.0.1:2181,127.0.0.1:2281,127.0.0.1:2381
```

3、创建 CuratorFramework 对象

```
@Configuration
public class CuratorFrameworkConfiguration {

    @Autowired
    private Environment environment ;

    @Bean
    public CuratorFramework curatorFramework() {

        // 第一个参数：两次连接的等待时间，第二个参数：表示的是最大的重试次数
        ExponentialBackoffRetry exponentialBackoffRetry = new ExponentialBackoffRetry(1000 , 3) ;
        CuratorFramework curatorFramework =
        CuratorFrameworkFactory.builder().connectString(environment.getProperty("zk.cluster.nodes"))
            .retryPolicy(exponentialBackoffRetry)
            .namespace("itheima-train-znode")
            .sessionTimeoutMs(5000)
            .build();
        curatorFramework.start();

        return curatorFramework ;
    }

    @Bean
    public InterProcessMutex interProcessMutex(CuratorFramework curatorFramework) {
        return new InterProcessMutex(curatorFramework , "/lock-znode") ;
    }
}
```

4、调用 InterProcessMutex 想法方法获取锁和释放锁

```
// 开始获取锁，指定获得锁最大的等待的时间，抢夺时，如果出现堵塞，会在超过该时间后，返回 false
public boolean acquire(long time, TimeUnit unit)
public void release() // 释放锁
```



黑马程序员  
www.itheima.com

传智播客旗下  
高端IT教育品牌

改变中国IT教育，我们正在行动