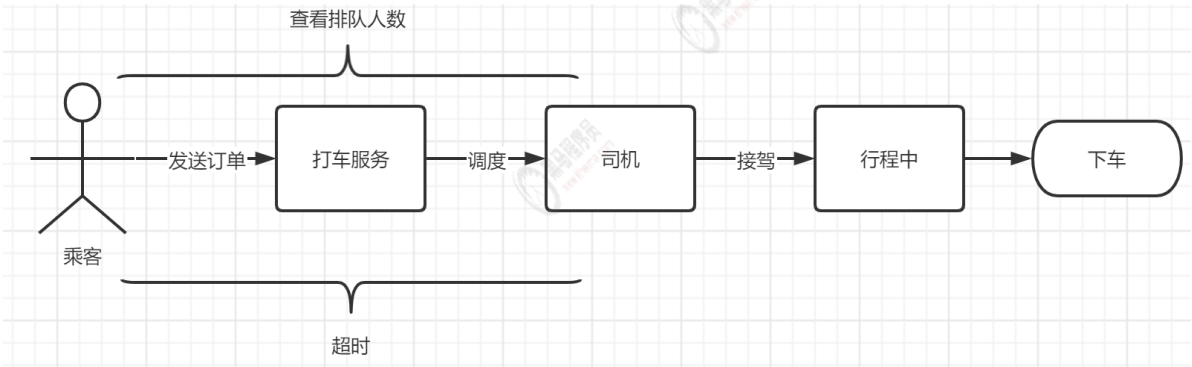
RabbitMQ高阶使用

1 能力目标

- 能够理解和使用RabbitMQ延时任务
- 能够理解和使用RabbitMQ的死信队列
- 能够理解RabbitMQ的生产和消费的保证
- 能够使用zset实现排队(排行榜)功能
- 能够理解websocket消息推送服务的原理和方案

2 1. 从打车开始说起

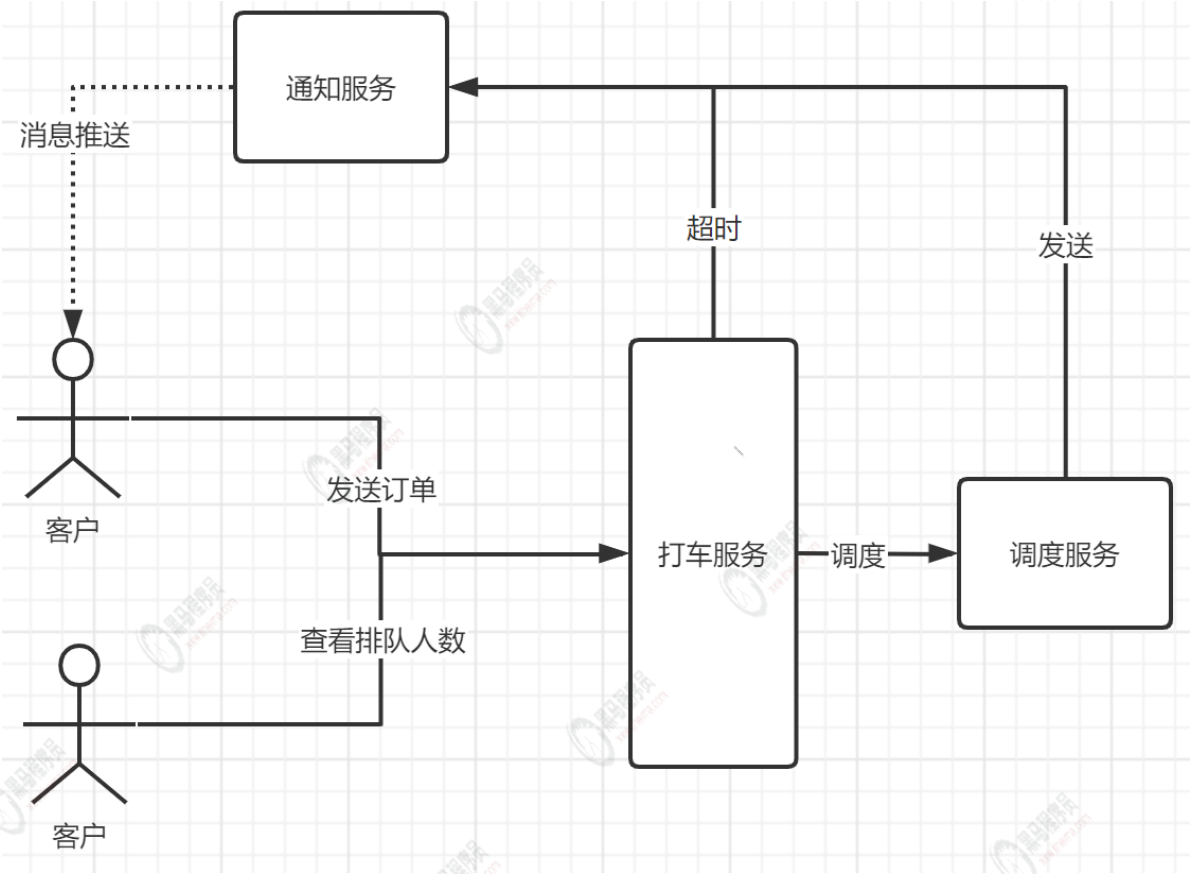
> 我们把滴滴打车的流程简化下



1. 登录app后点击打车开始进行打车
2. 打车服务开始为司机派单
3. 司机接单后开始给来接驾
4. 上车乘客后处于行程中
5. 行程结束后完成本次打车服务

3 1.1 需要解决的问题

> 我们需要实现派单服务，用户发送打车订单后需要进行进行派单，如果在指定时间内没有找到司机就会收到派单超时的通知，并且能够实时查看当前排队的抢单人数



> 下面我们来介绍下涉打车涉及到的一些问题

3.1 1.1.2 打车超时

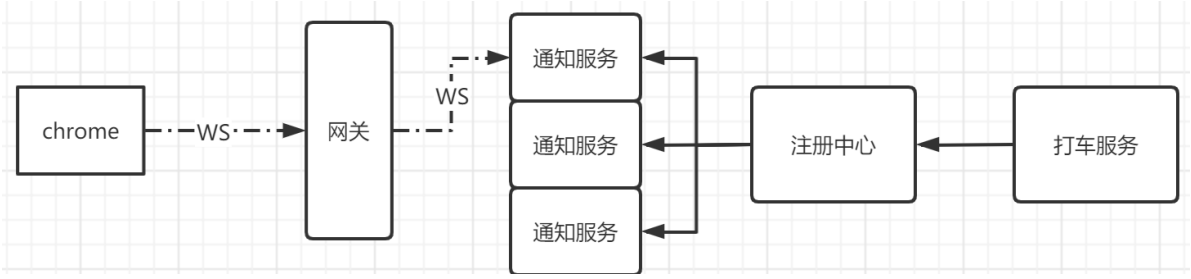
> 主要讲解打车服务在超时后的处理，比如打车后等待多长时间没有打到车后会通知等待超时。

3.2 1.1.2 打车排队

> 因为排队我们需要获取当前的排队人数，所以需要通过redis的zset结构来实现排队功能，后面会详解讲解

3.3 1.1.3 消息推送

> 我们需要将我们的异步处理结构返回到客户端，我们的客户端是使用的websocket连接的，因为websocket是点对点连接的，连接到一台固定的通知服务后，只能从这一台通知服务来获取数据，因为我们的通知服务允许分布式部署，这个问题改如何解决?



> 后面我们会一个个的来进行解决，我们先来学习下打车超时的问题

4 2. 延时队列

> 我们先看下如何来解决打车超时的问题

4.1 2.1 什么是延时队列

> 在开发中，往往会遇到一些关于延时任务的需求。例如
>
> - 生成订单30分钟未支付，则自动取消
> - 生成订单60秒后,给用户发短信
> - 滴滴打车订单完成后，如果用户一直不评价，48小时后会将自动评价为5星。

4.1.1 2.1.1 和定时任务区别

> 对上述的任务，我们给一个专业的名字来形容，那就是**延时任务**。那么这里就会产生一个问题，这个**延时任务**和**定时任务**的区别究竟在哪里呢？一共有如下几点区别

1. 定时任务有明确的触发时间，延时任务没有
2. 定时任务有执行周期，而延时任务在某事件触发后一段时间内执行，没有执行周期
3. 定时任务一般执行的是批处理操作是多个任务，而延时任务一般是单个任务

4.2 2.2 延时队列使用场景

> 那么什么时候需要用延时队列呢？考虑一下以下场景：

1. 订单在十分钟之内未支付则自动取消。
2. 新创建的店铺，如果在十天内都没有上传过商品，则自动发送消息提醒。
3. 账单在一周内未支付，则自动结算。
4. 用户注册成功后，如果三天内没有登陆则进行短信提醒。
5. 用户发起退款，如果三天内没有得到处理则通知相关运营人员。
6. 预定会议后，需要在预定的时间点前十分钟通知各个与会人员参加会议

> 可以想一下美团点餐，超时时间

4.3 2.3 常见方案

> 下面我们来介绍下常见的延时任务的解决方案

4.3.1 2.3.1 数据库轮询

该方案通常是在小型项目中使用，即通过一个线程定时的去扫描数据库，通过订单时间来判断是否有超时的订单，然后进行update或delete等操作

4.3.1.1 优点

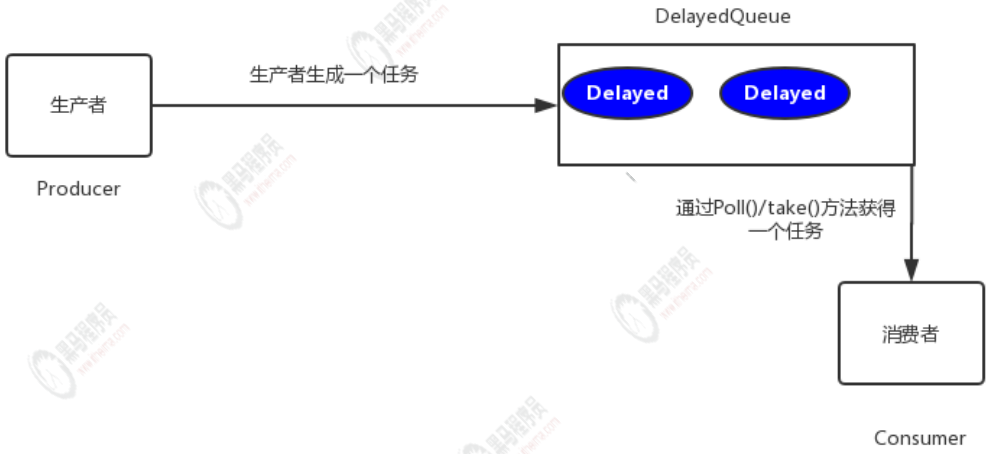
简单易行，支持集群操作

4.3.1.2 缺点

1. 对服务器内存消耗大
2. 存在延迟，比如你每隔3分钟扫描一次，那最坏的延迟时间就是3分钟
3. 假设你的订单有几千万条，每隔几分钟这样扫描一次，数据库损耗极大

4.3.2 2.3.1 JDK的延迟队列

该方案是利用JDK自带的DelayQueue来实现，这是一个无界阻塞队列，该队列只有在延迟期满的时候才能从中获取元素，放入DelayQueue中的对象，是必须实现Delayed接口的。



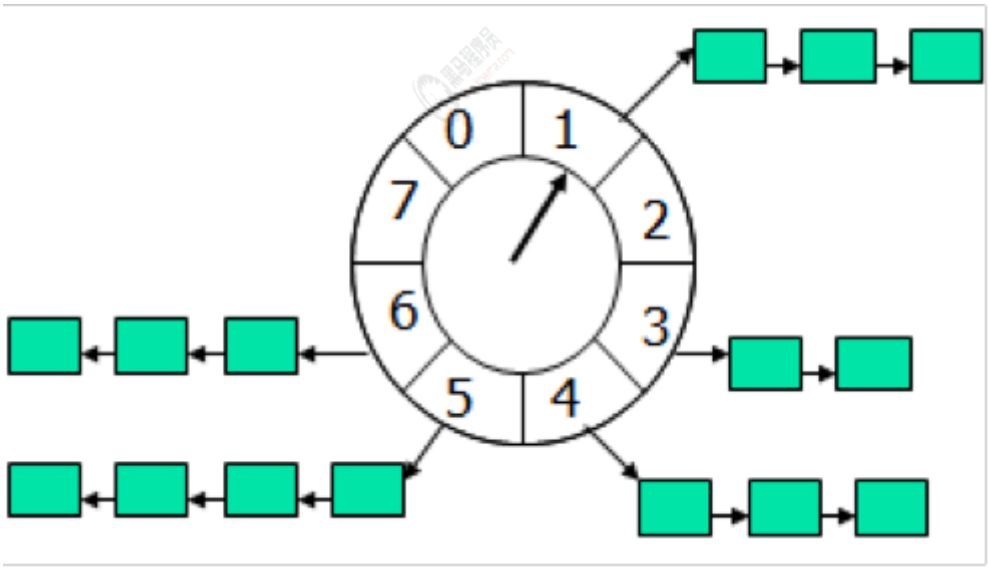
4.3.2.1 优点

效率高,任务触发时间延迟低。

4.3.2.2 缺点

1. 服务器重启后，数据全部消失，怕宕机
2. 集群扩展相当麻烦
3. 因为内存条件限制的原因，比如下单未付款的订单数太多，那么很容易就出现OOM异常
4. 代码复杂度较高

4.3.3 2.3.3 netty时间轮算法



时间轮算法可以类比于时钟，如上图箭头（指针）按某一个方向按固定频率轮动，每一次跳动称为一个 tick。这样可以看出定时轮由个3个重要的属性参数，ticksPerWheel（一轮的tick数），tickDuration（一个tick的持续时间）以及 timeUnit（时间单位），例如当ticksPerWheel=60，tickDuration=1，timeUnit=秒，这就和现实中的始终的秒针走动完全类似了。

如果当前指针指在1上面，我有一个任务需要4秒以后执行，那么这个执行的线程回调或者消息将会被放在5上。那如果需要在20秒之后执行怎么办，由于这个环形结构槽数只到8，如果要20秒，指针需要多转2圈。位置是在2圈之后的5上面（20 % 8 + 1）

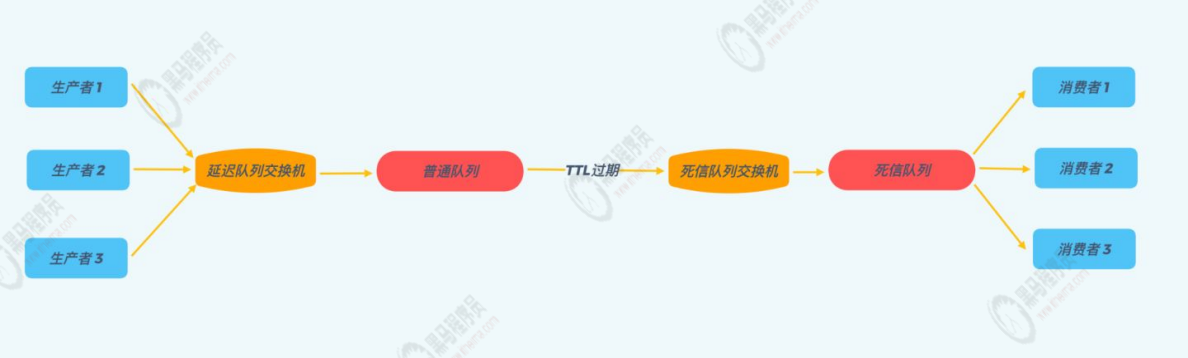
4.3.3.1 优点

效率高,任务触发时间延迟时间比delayQueue低，代码复杂度比delayQueue低。

4.3.3.2 缺点

- 服务器重启后，数据全部消失，怕宕机
- 集群扩展相当麻烦
- 因为内存条件限制的原因，比如下单未付款的订单数太多，那么很容易就出现OOM异常

4.3.4 2.3.4 使用消息队列



> 我们可以采用rabbitMQ的延时队列。RabbitMQ具有以下两个特性，可以实现延迟队列
>
> - RabbitMQ可以针对Queue和Message设置 x-message-tt，来控制消息的生存时间，如果超时，则消息变为dead letter
> - lRabbitMQ的Queue可以配置x-dead-letter-exchange 和x-dead-letter-routing-key（可选）两个参数，用来控制队列内出现了deadletter，则按照这两个参数重新路由。

4.3.4.1 优点

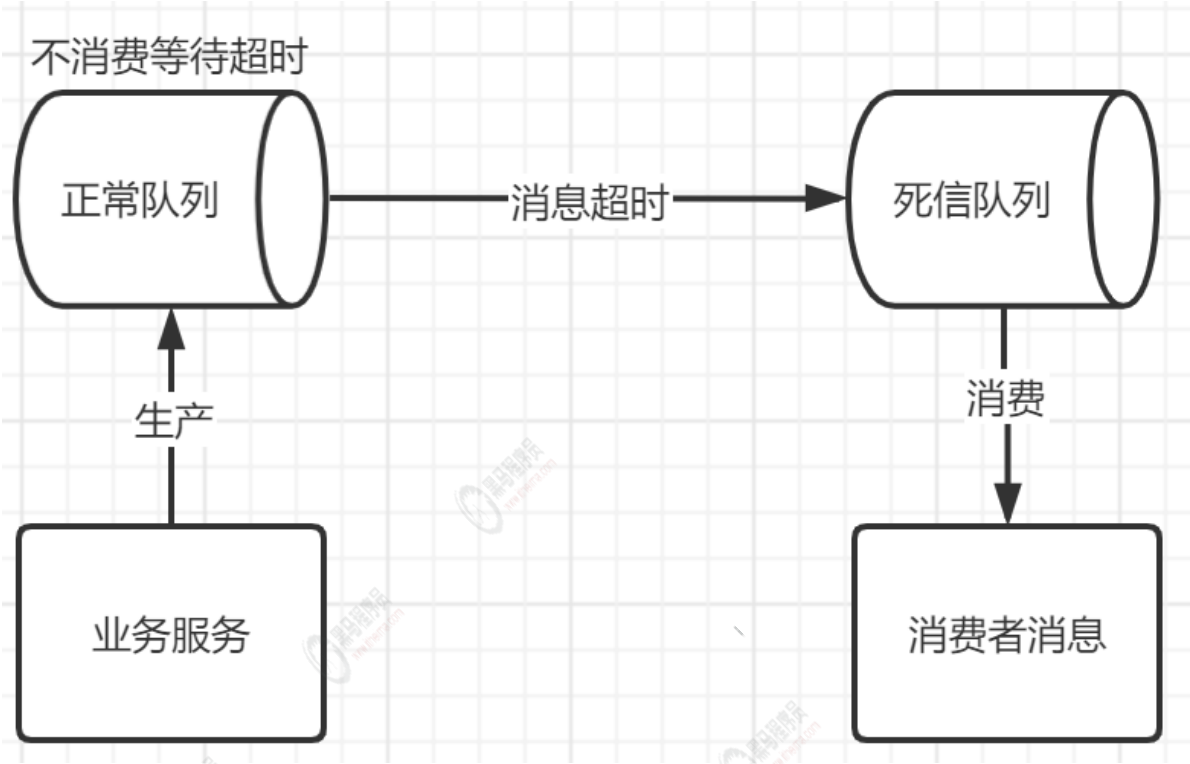
高效,可以利用rabbitmq的分布式特性轻易的进行横向扩展,消息支持持久化增加了可靠性。

4.3.4.2 缺点

本身的易用度要依赖于rabbitMq的运维.因为要引用rabbitMq,所以复杂度和成本变高

4.4 2.4 延时队列

> RabbitMQ中没有对消息延迟进行实现，但是我们可以通过TTL以及死信路由来实现消息延迟。



4.4.1 2.4.1 TTL(消息过期时间)

> 在介绍延时队列之前，还需要先介绍一下RabbitMQ中的一个高级特性——`TTL（Time To Live）`。

`TTL`是RabbitMQ中一个消息或者队列的属性，表明`一条消息或者该队列中的所有消息的最大存活时间`，单位是毫秒。换句话说，如果一条消息设置了TTL属性或者进入了设置TTL属性的队列，那么这条消息如果在TTL设置的时间内没有被消费，则会成为“死信”，如果不设置TTL，表示消息永远不会过期，如果将TTL设置为0，则表示除非此时可以直接投递该消息到消费者，否则该消息将会被丢弃。

4.4.1.1 2.4.1.1 配置队列TTL

> 一种是在创建队列的时候设置队列的“x-message-ttl”属性

```
@Bean
public Queue taxiOverQueue() {
    Map<String, Object> args = new HashMap<>(2);
    args.put("x-message-ttl", 30000);
    return QueueBuilder.durable(TAXI_OVER_QUEUE).withArguments(args).build();
}
```

> 这样所有被投递到该队列的消息都最多不会存活超过30s，但是消息会到哪里呢，如果没有任何处理，消息会被丢弃，如果配置有死信队列，超时的消息会被投递到死信队列

4.5 2.5 死信队列

> 讲到延时消息就不能不讲死信队列

4.5.1 2.5.1 什么是死信队列

先从概念解释上搞清楚这个定义，死信，顾名思义就是无法被消费的消息，字面意思可以这样理解，一般来说，producer将消息投递到broker或者直接到queue里了，consumer从queue取出消息进行消费，但某些时候由于特定的原因导致queue中的某些消息无法被消费，这样的消息如果没有后续的处理，就变成了死信，有死信，自然就有了死信队列；

4.5.2 2.5.2 死信队列使用场景

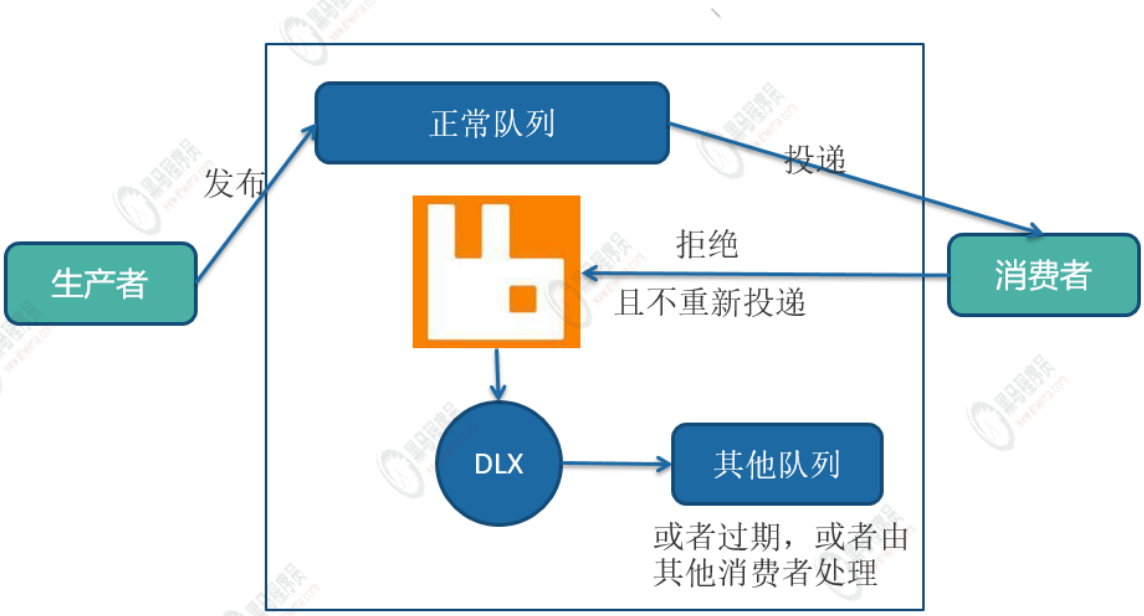
> RabbitMQ中的死信交换器(dead letter exchange)可以接收下面三种场景中的消息:

- 消费者对消息使用了`basicReject`或者`basicNack`回复,并且`requeue`参数设置为`false`,即不再将该消息重新在消费者间进行投递.
- 消息在队列中超时. RabbitMQ可以在单个消息或者队列中设置`TTL`属性.
- 队列中的消息已经超过其设置的最大消息个数.

4.5.3 2.5.3 死信队列如何使用

死信交换器不是默认的设置，这里是被投递消息被拒绝后的一个可选行为，是在创建队列的时进行声明的，往往用在对问题消息的诊断上。

死信交换器仍然只是一个普通的交换器，创建时并没有特别要求和操作。在创建队列的时候，声明该交换器将用作保存被拒绝的消息即可，相关的参数是`x-dead-letter-exchange`。

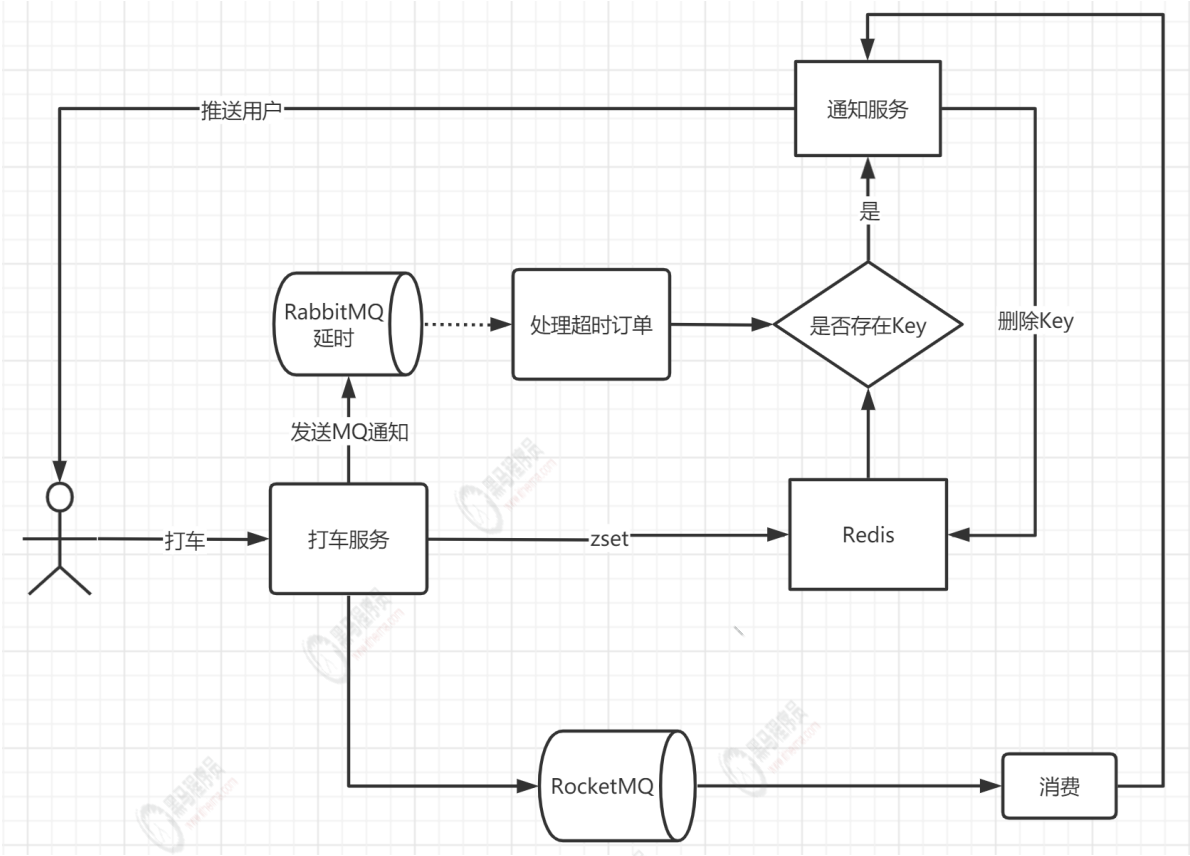


4.5.4 2.5.4 相关代码

```
@Bean
public Queue taxiOverQueue() {
    Map<String, Object> args = new HashMap<>(2);
    // x-dead-letter-exchange    这里声明当前队列绑定的死信交换机
    args.put("x-dead-letter-exchange", TAXI_DEAD_QUEUE_EXCHANGE);
    // x-dead-letter-routing-key  这里声明当前队列的死信路由key
    args.put("x-dead-letter-routing-key", TAXI_DEAD_KEY);
    return QueueBuilder.durable(TAXI_OVER_QUEUE).withArguments(args).build();
}
```

4.6 2.6 打车超时处理

> 用户通过调用打车服务将数据放进RabbitMQ的死信队列进行延时操作，等待一段时间后，正常的业务处理还没有处理到我们发起的数据，将会进行超时处理，通过通知服务将我们的处理结构通过websocket方式推送到我们的客户端。



4.6.1 2.6.1 打车超时实现

> 在创建队列的时候配置死信交换器并设置队列的“x-message-ttl”属性

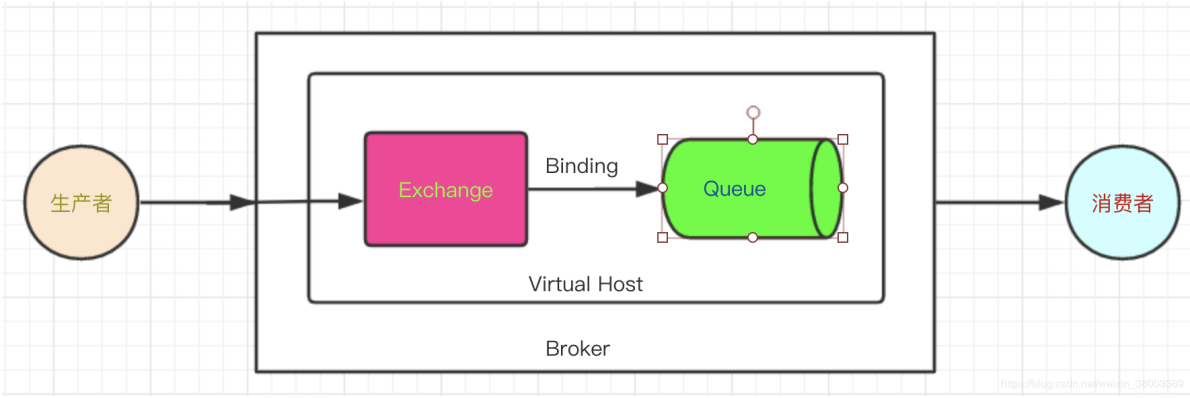
```
@Bean
public Queue taxiDeadQueue() {
    return new Queue(TAXI_DEAD_QUEUE,true);
}

@Bean
public Queue taxiOverQueue() {
    Map<String, Object> args = new HashMap<>(2);
    // x-dead-letter-exchange    这里声明当前队列绑定的死信交换机
    args.put("x-dead-letter-exchange", TAXI_DEAD_QUEUE_EXCHANGE);
    // x-dead-letter-routing-key  这里声明当前队列的死信路由key
    args.put("x-dead-letter-routing-key", TAXI_DEAD_KEY);
    // x-message-ttl  声明队列的TTL
    args.put("x-message-ttl", 30000);
    return QueueBuilder.durable(TAXI_OVER_QUEUE).withArguments(args).build();
}
```

这样所有被投递到该队列的消息都最多不会存活超过30s，超时后的消息会被投递到死信交换器

5 3. RabbitMQ消息可靠性保障

> 消息的可靠性投递是使用消息中间件不可避免的问题，管是使用kafka、rocketMQ或者rabbitMQ，那么在RabbitMQ中如何保证消息的可靠性投递呢?



> 从上面的图可以看到，消息的投递有三个对象参与：

- 生产者
- broker
- 消费者

5.1 3.1 生产者保证

> 生产者发送消息到broker时，要保证消息的可靠性，主要的方案有以下2种

- 发送发确认
- 失败通知

5.1.1 3.1.1 RabbitMQ回顾

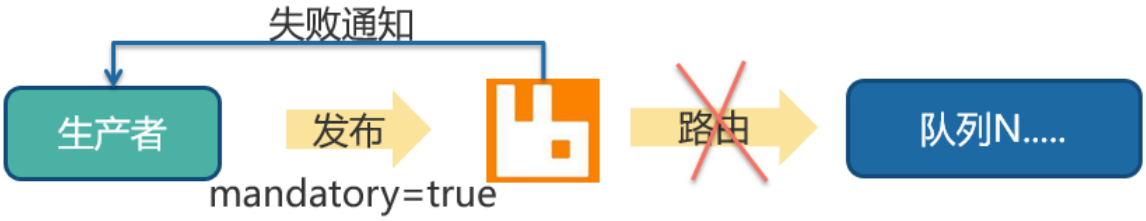
生产者通过指定一个 exchange 和 routingkey 把消息送达到某个队列中去，然后消费者监听队列，进行消费处理。但是在某些情况下，如果我们在发送消息时，当前的 exchange 不存在或者指定的 routingkey 路由不到，这个时候如果要监听这种不可达的消息，这个时候就需要失败通知。

不做任何配置的情况下，生产者是不知道消息是否真正到达RabbitMQ，也就是说消息发布操作不返回任何消息给生产者。

5.1.2 3.1.2 失败通知

> 如果出现消息无法投递到队列会出现失败通知

那么怎么保证我们消息发布的可靠性？这里我们就可以启动**失败通知**，在原生编程中在发送消息时设置`mandatory`标志，即可开启故障检测模式。



注意：它只会让 RabbitMQ 向你通知失败，而不会通知成功。如果消息正确路由到队列，则发布者不会受到任何通知。带来的问题是无法确保发布消息一定是成功的，因为通知失败的消息可能会丢失。

5.1.2.1 3.1.2.1 实现方式

> spring配置

```
spring:
  rabbitmq:
    # 消息在未被队列收到的情况下返回
    publisher-returns: true
```

> 关键代码，注意需要发送者实现 `ReturnCallback` 接口方可实现失败通知

```
/**
  * 失败通知
  * 队列投递错误应答
  * 只有投递队列错误才会应答
  */
@Override
public void returnedMessage(Message message, int replyCode, String replyText,
String exchange, String routingKey) {
    //消息体为空直接返回
    if (null == message) {
        return;
    }
    TaxiBO taxiBO = JSON.parseObject(message.getBody(), TaxiBO.class);
    if (null != taxiBO) {
        //删除rediskey
        redisHelper.handelAccountTaxi(taxiBO.getAccountId());
        //记录错误日志
        recordErrorMessage(taxiBO, replyText, exchange, routingKey, message,
replyCode);
    }
}
```

5.1.2.2 3.1.2.2 遇到的问题问题

如果消息正确路由到队列，则发布者不会受到任何通知。带来的问题是无法确保发布消息一定是成功的，因为通知失败的消息可能会丢失。

我们可以使用RabbitMQ的发送方确认来实现，它不仅仅在路由失败的时候给我们发送消息，并且能够在消息路由成功的时候也给我们发送消息。

5.1.3 3.1.3 发送发确认

> 发送方确认是指生产者投递消息后，如果 Broker 接收到消息，则会给生产者一个应答。生产者进行接收应答，用来确认这条消息是否正常的发送到 Broker，这种方式也是消息可靠性投递的核心保障
>
> rabbitmq消息发送分为两个阶段：

- 将消息发送到broker，即发送到exchage交换机
- 消息通过交换机exchange被路由到队列queue

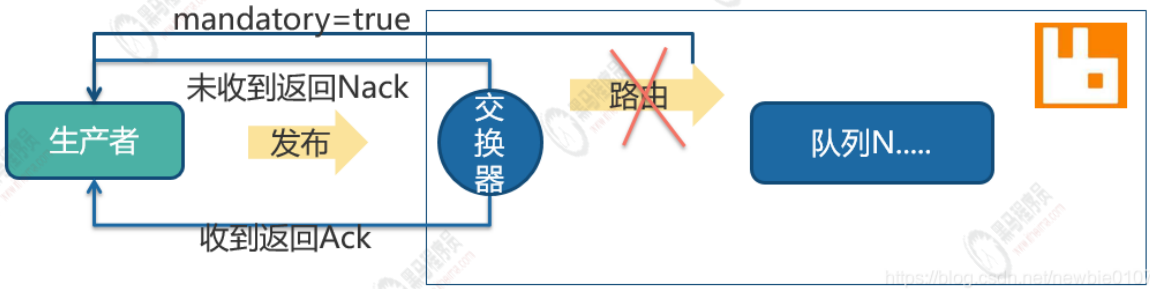
一旦消息投递到队列，队列则会向生产者发送一个通知，如果设置了消息持久化到磁盘，则会等待消息持久化到磁盘之后再发送通知

注意：发送发确认只有出现RabbitMQ内部错误无法投递才会出现发送发确认失败。

> **发送方确认模式**需要分两种情况下列来看，首先我们先来看一看消息不可路由的情况

5.1.3.1 3.1.3.1 不可路由

> 当前消息到达交换器后对于发送者确认是成功的



首先当RabbitMQ交换器不可路由时，消息也根本不会投递到队列中，所以这里只管到交换器的路径，**当消息成功送到交换器后，就会进行确认操作。**

另外在这过程中，生产者收到了确认消息后，那么因为消息无法路由，所以该消息也是无效的，无法投递到队列，所以一般情况下这里会结合**失败通知**来一同使用，这里一般会进行设置`mandatory`模式，失败则会调用addReturnListener监听器来进行处理。

> **发送方确认模式**的另一种情况肯定就是消息可以进行路由

5.1.3.2 3.1.3.2 可以路由

> 只要消息能够到达队列即可进行确认，一般是RabbitMQ发生内部错误才会出现失败

可以路由的消息，要等到消息被投递到所有匹配的队列之后，broker会发送一个确认给生产者(包含消息的唯一ID)，这就使得生产者知道消息已经正确到达目的队列了。

如果消息和队列是可持久化的，那么确认消息会在将消息写入磁盘之后发出，broker回传给生产者的确认消息中delivery-tag域包含了确认消息的序列号。

5.1.3.3 3.1.3.3 使用方式

> spring配置

```
spring:
  rabbitmq:
    # 开启消息确认机制
    publisher-confirm-type: correlated
```

> 关键代码，注意需要发送者实现`ConfirmCallback`接口方可实现失败通知

```
/**
  * 发送发确认
  * 交换器投递后的应答
  * 正常异常都会进行调用
  *
  * @param correlationData
  * @param ack
  * @param cause
  */
@Override
public void confirm(CorrelationData correlationData, boolean ack, String cause)
{
    //只有异常的数据才需要处理
    if (!ack) {
        //关联数据为空直接返回
        if (correlationData == null) {
            return;
        }
        //检查返回消息是否为null
        if (null != correlationData.getReturnedMessage()) {
            TaxiBO taxiBO =
JSON.parseObject(correlationData.getReturnedMessage().getBody(), TaxiBO.class);
            //处理消息还原用户未打车状态
            redisHelper.handelAccountTaxi(taxiBO.getAccountId());
            //获取交换器
            String exchange =
correlationData.getReturnedMessage().getMessageProperties().getHeader("SEND_EXCH
ANGE");
            //获取队列信息
            String routingKey =
correlationData.getReturnedMessage().getMessageProperties().getHeader("SEND_ROUT
ING_KEY");
            //获取当前的消息体
            Message message = correlationData.getReturnedMessage();
            //记录错误日志
            recordErrorMessage(taxiBO, cause, exchange, routingKey, message,
-1);
        }
    }
}
```

5.1.4 3.1.4 Broker丢失消息

前面我们从生产者的角度分析了消息可靠性传输的原理和实现，这一部分我们从broker的角度来看一下如何能保证消息的可靠性传输？

假设有现在一种情况，生产者已经成功将消息发送到了交换机，并且交换机也成功的将消息路由到了队列中，但是在消费者还未进行消费时，mq挂掉了，那么重启mq之后消息还会存在吗？如果消息不存在，那就造成了消息的丢失，也就不能保证消息的可靠性传输了。

也就是现在的问题变成了如何在mq挂掉重启之后还能保证消息是存在的？

> 开启RabbitMQ的持久化，也即消息写入后会持久化到磁盘，此时即使mq挂掉了，重启之后也会自动读取之前存储的额数据

5.1.4.1 3.1.4.1 持久化队列

```
@Bean
public Queue queue(){
    return new Queue(queueName,true);
}
```

5.1.4.2 3.1.4.2 持久化交换器

```
@Bean
DirectExchange directExchange() {
    return new DirectExchange(exchangeName,true,false);
}
```

5.1.4.3 3.1.4.3 发送持久化消息

> 发送消息时，设置消息的deliveryMode=2
>
> 注意：如果使用SpringBoot的话，发送消息时自动设置deliveryMode=2，不需要人工再去设置

5.1.4.4 3.1.4.4 Broker总结

> 通过以上方式，可以保证大部分消息在broker不会丢失，但是还是有很小的概率会丢失消息，什么情况下会丢失呢？

假如消息到达队列之后，还未保存到磁盘mq就挂掉了，此时还是有很小的几率会导致消息丢失的。

这就要mq的持久化和前面的confirm进行配合使用，只有当消息写入磁盘后才返回ack，那么就是在持久化之前mq挂掉了，但是由于生产者没有接收到ack信号，此时可以进行消息重发。

5.2 3.2 消费方消息可靠性

5.2.1 3.2.1 消费者手动确认

> 消费者接收到消息，但是还未处理或者还未处理完，此时消费者进程挂掉了，比如重启或者异常断电等，此时mq认为消费者已经完成消息消费，就会从队列中删除消息，从而导致消息丢失。

那该如何避免这种情况呢？这就要用到RabbitMQ提供的**ack**机制，RabbitMQ默认是自动ack的，此时需要将其修改为手动ack，也即**自己的程序确定消息已经处理完成后，手动提交ack**，此时如果再遇到消息未处理进程就挂掉的情况，由于没有提交ack，RabbitMQ就不会删除这条消息，而是会把这条消息发送给其他消费者处理，但是消息是不会丢的。

5.2.1.1 3.2.1.1 配置文件

```
spring:
  rabbitmq:
    listener:
      simple:
        acknowledge-mode: manual  # 手动ack
```

5.2.1.2 3.2.1.2 参数介绍

> **acknowledge-mode: manual**就表示开启手动ack，该配置项的其他两个值分别是none和auto

- auto：消费者根据程序执行正常或者抛出异常来决定是提交ack或者nack，不要把none和auto搞混了
- manual: 手动ack，用户必须手动提交ack或者nack
- none: 没有ack机制

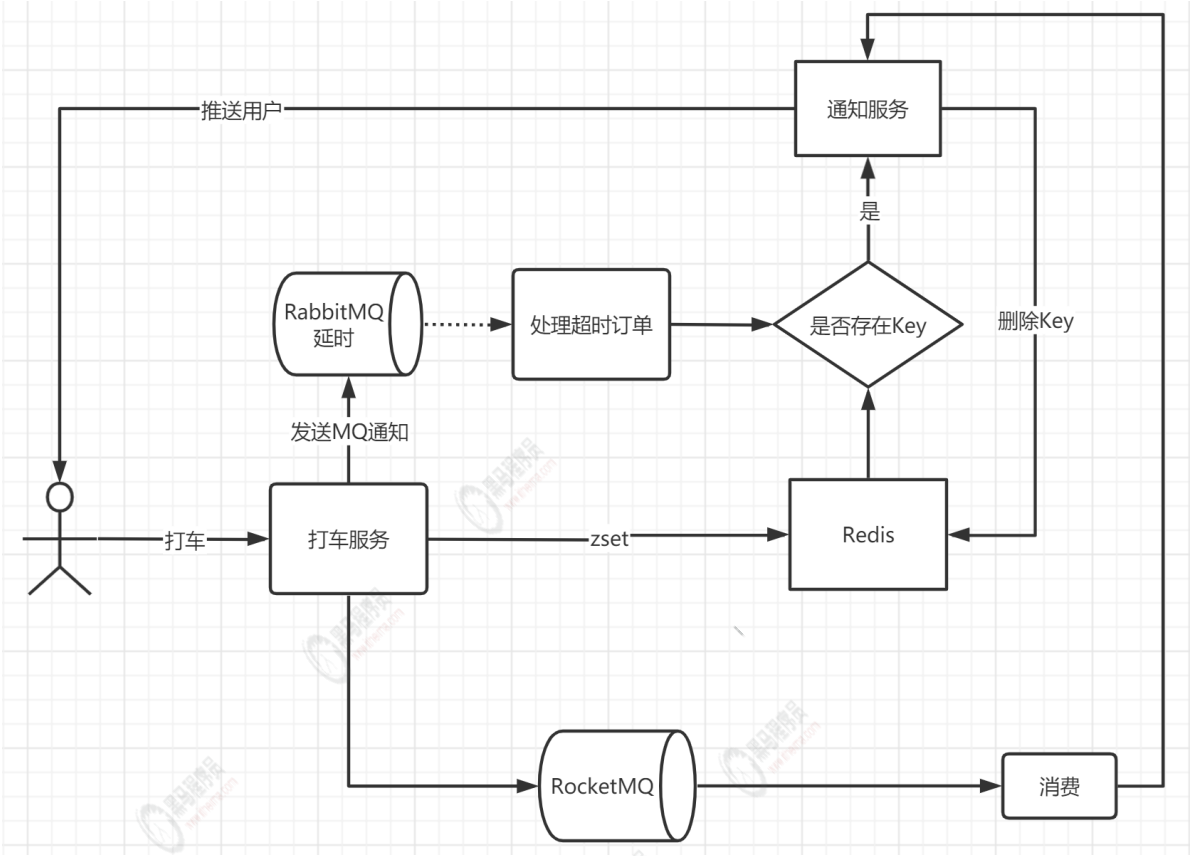
默认值是auto，如果将ack的模式设置为auto，此时如果消费者执行异常的话，就相当于执行了nack方法，消息会被放置到队列头部，消息会被无限期的执行，从而导致后续的消息无法消费。

5.2.1.3 3.3.1.3 消费者实现

```
@RabbitListener(
            bindings =
                    {
                            @QueueBinding(value = @Queue(value =
RabbitConfig.TAXI_DEAD_QUEUE, durable = "true"),
                                    exchange = @Exchange(value =
RabbitConfig.TAXI_DEAD_QUEUE_EXCHANGE), key = RabbitConfig.TAXI_DEAD_KEY)
                    })
    @RabbitHandler
    public void processOrder(Message massage, Channel channel,
@Header(AmqpHeaders.DELIVERY_TAG) long tag) {
        TaxiBO taxiBO = JSON.parseObject(massage.getBody(), TaxiBO.class);
        try {
            //开始处理订单
            logger.info("处理超时订单，订单详细信息：" + taxiBO.toString());
            taxiService.taxiTimeout(taxiBO);
            //手动确认机制
            channel.basicAck(tag, false);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

5.3 3.3 业务可靠性分析

> 在超时订单业务中，无需过多的考虑消息丢失以及幂等性问题



5.3.1 3.3.1 消息丢失

在这个业务场景中，用户发起打车请求，如果用户消息丢失，对整体业务是没有任何影响的，用户可以再次发起打车操作，这个消息丢失问题概率很低，可以进行简单化设计，如果出现发送失败只需要回退redis中的操作即可。

5.3.2 3.3.2 幂等性校验

因为使用了延时队列，对于这个业务来说是不需要进行幂等性校验的，因为第一次超时时如果存在redis用户排名的key就会被删除，下一次redis没有的值在删除一次，这种操作是幂等的，所以不需要考虑幂等性

5.3.3 3.3.3 数据回滚

虽然无需做到消息完全不丢失以及消息的幂等性，但是需要考虑如果出现问题，需要将插入Redis的的key值回滚掉，防止影响业务正常判断

6 4. 排队人数

6.1 4.1 需求

> 在打车的过程中如果人数较多的情况下会在派单中等待，如果想知道我的前面还有多少人呢，我们就需要一个排队人数的功能

接受用户的派单数据，但因为派单处理需要一定的时间，所以只能在MQ中有序消费数据，对用户进行排队操作。当然这个排队操作，用户是不透明的，某些用户的请求可能被优先处理，但是通过MQ可以实现整体的有序。

用户很关心自己派单目前的处理进度，即和我一样打车的前面还有多少人，打车APP上显示“你前面还有多少人在排队”。所以后台要能告知用户目前他的派单进度。

6.1.1 4.1.1 需求分析

- 入队：可以理解为写操作，需要后端存储数据。
- 获取进度：可以理解为读操作，而且可以预见这个读操作应该比写操作频繁。如果用户很关注她的订单进展，说不定会一直刷新查看他的订单排队情况。

6.2 4.2 实现方案

6.2.1 4.2.1 MySQL

用户的订单数据肯定得持久化存储，MySQL是一个不错的选择。既然需求这么简单，无非一个订单数据嘛。暂且用一张表“订单表（T_Order）”来保存正在排队的订单，已经处理完毕的订单则从T_Order表迁移至“（历史订单表T_History_Order）”。这样的好处避免订单表数据量太大，提高读写性能。

6.2.1.1 4.2.1.1 入队

> 完成订单的入库，显然就是一个insert语句了

```
insert into T_Order(...) values(...);
```

6.2.1.2 4.2.1.2 获取进度

> 需查询自己订单的排队情况，那肯定看比自己订单时间还早的用户有多少人了。这些比自己下单时间还早的人，就是排在自己前面的人了。假设一个用户同时只能有一个订单在排队。

```
#先查出自己订单时间，假设是1429389316
select orderTime from T_Order where uid=8888;

#再查有多少人的订单时间比自己的早
select count(orderTime) from T_Order where orderTime <= 1429389316;
```

6.2.1.3 4.2.1.3 遇到问题

互联网的精髓就是“小步快跑，快速迭代”。用MySQL快速完成需求，面向用户服务后。初期阶段，一切ok。但是当这个业务运营得好，用户量大的时候，就会发现用户经常投诉“我查询自己的订单排队进度，经常报错”。甚至处理订单的同事，也经常抱怨从订单系统里面查看订单，非常缓慢。select count操作基本都是全表扫描操作，看来MySQL面对这么大规模的全表查询操作，还是有点吃力。

6.2.2 4.2.3 Redis Zset

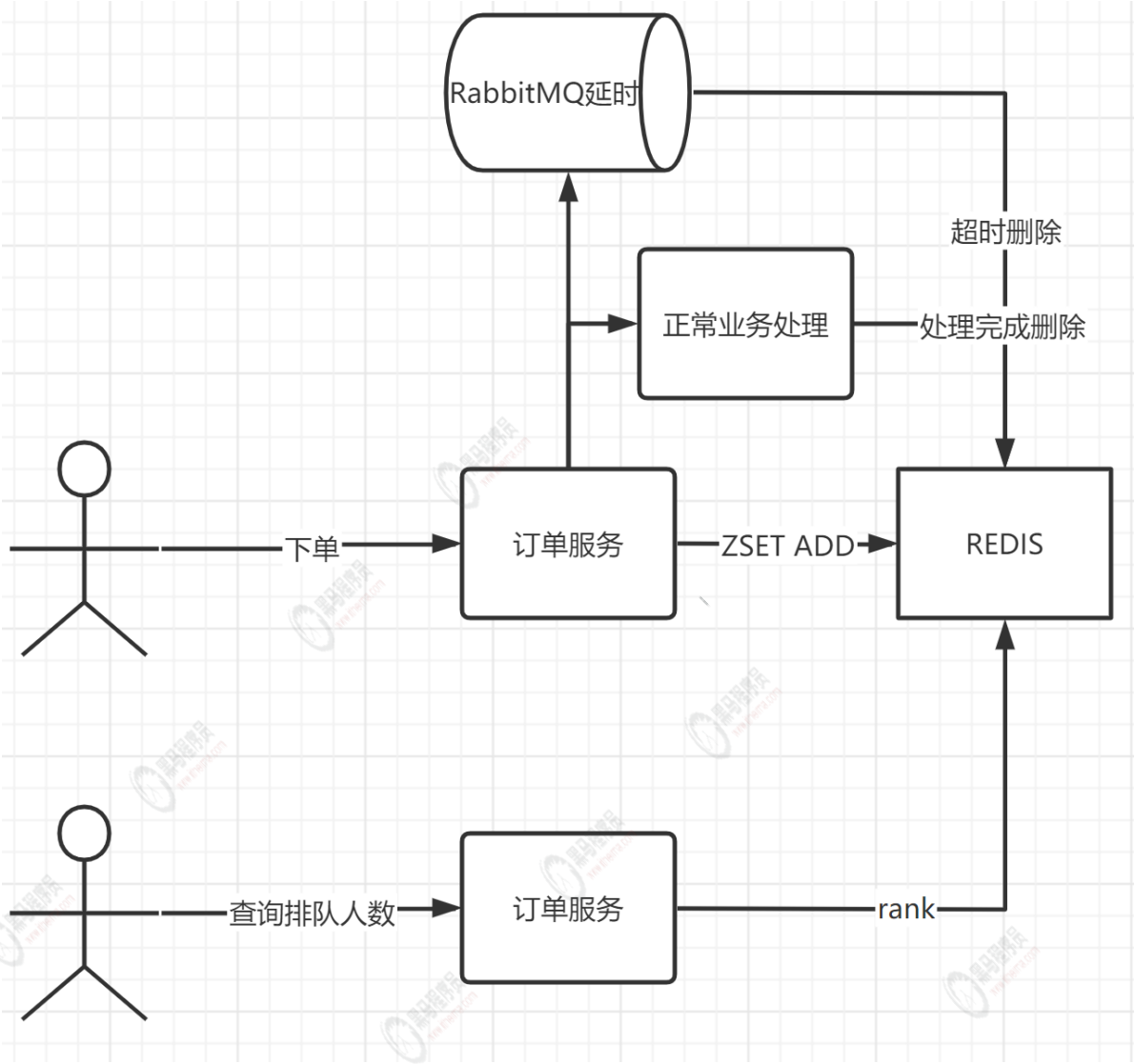
NoSQL在互联网领域的江湖地位已经很牢靠了，看来得请他老人家出来救场了。没错，使用Redis的有序集合(sorted sets)数据结构，就可以完美的解决这个问题。因为有序集合底层的实现是跳表这种数据结构，时间复杂度是logN，即使有序集合里面的订单有100万之多，耗时也基本都是纳秒级别（基本不到1毫秒）。

1. 用户提交一个订单，我们写入redis的zset中。
2. 用户要查询自己的订单排队情况，这时候我们只要查询redis的有序集合就可以了。命令为rank
3. 当这个订单被处理完成后，直接一个zrem命令将订单从有序集合中删除即可

因为Redis基本都是内存操作，而且有序集合的底层实现是跳表这种效率媲美平衡树，但是实现又简单的数据结构，从而完美的释放了MySQL的读压力。

6.3 4.3 排队人数架构介绍

打车如果出现排队我们需要能够对当前排队的人数进行预估，能够知道当前我们前面有多少人在排队，我们采用redis的zset来实现排队，整体架构如下。



1. 用户打车通过zset加入到redis的有序集合
2. 异步将数据推送到RabbitMQ延迟以及进行正常业务处理
3. 处理完成后在zset中删除元素
4. 用户查询人数通过Rank命令进行查询

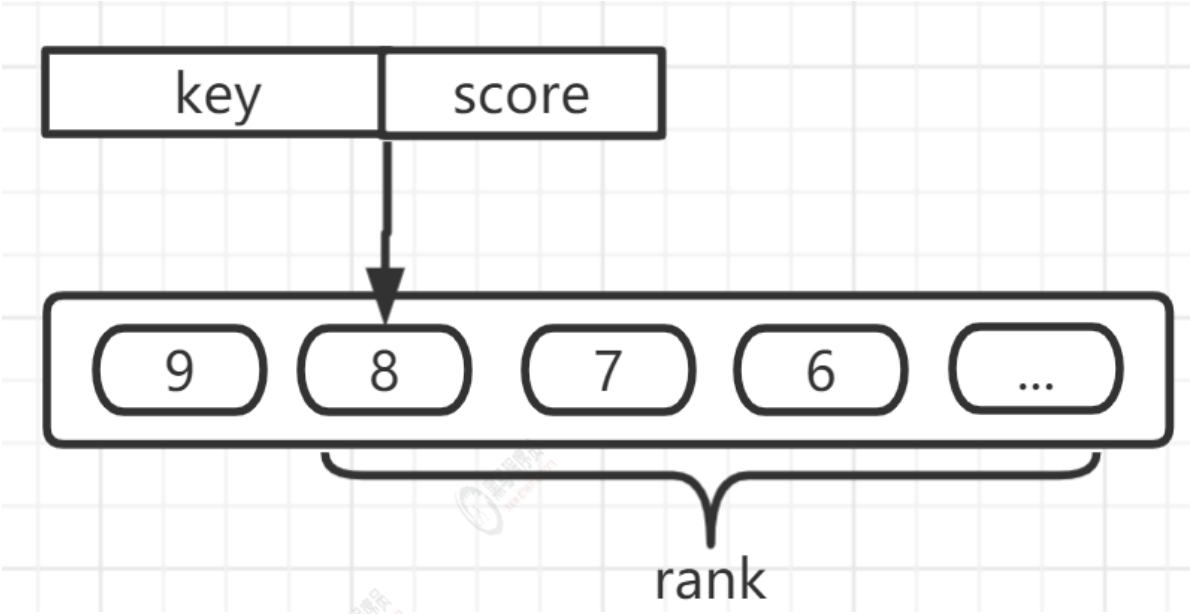
6.4 4.4 数据结构

6.4.1 4.4.2 zset结构

> 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。

我们只需要使用rank命令统计从0-当前key对应的分数的key的数量就可以得到当前的排名了

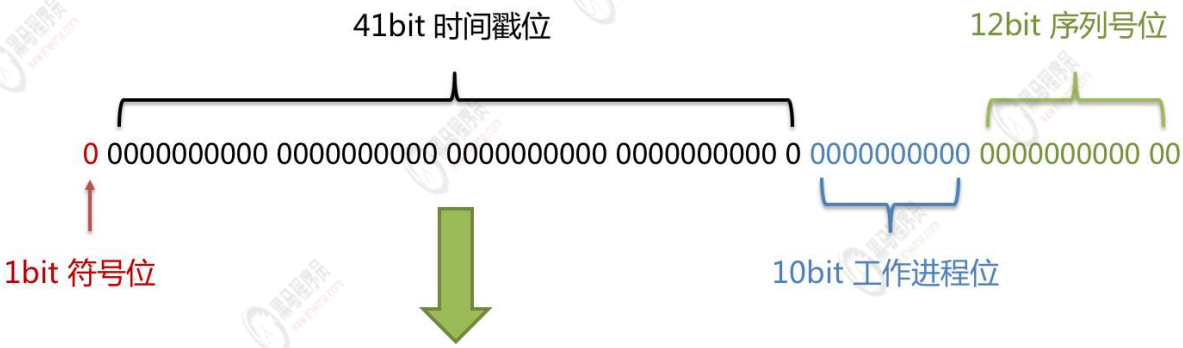


因为Redis基本都是内存操作，而且有序集合的底层实现是跳表这种效率媲美平衡树，但是实现又简单的数据结构，从而完美的释放了MySQL的读压力。

> 我们如何来保证分数不重复，并且是有序递增的呢，这里就要祭出来我们的雪花算法

6.4.2 4.4.1 雪花算法

> SnowFlake算法生成id的结果是一个64bit大小的整数，它的结构如下图：



时间范围： 2^{41} / (365 * 24 * 60 * 60 * 1000L) = 69.73年
工作进程数量： 2^{10} = 1024
生成不碰撞序列的TPS： 2^{12} * 1000 = 409.6万

1. **1bit**，不用，因为二进制中最高位是符号位，1表示负数，0表示正数。生成的id一般都是用整数，所以最高位固定为0。
2. **41bit-时间戳**，用来记录时间戳，毫秒级。
   - 41位可以表示$2^{41}-1$个数字，
   - 如果只用来表示正整数（计算机中正数包含0），可以表示的数值范围是：0 至 $2^{41}-1$，减1是因为可表示的数值范围是从0开始算的，而不是1。
   - 也就是说41位可以表示$2^{41}-1$个毫秒的值，转化成单位年则是
   $(2^{41}-1)/(1000*60*60*24*365)=69$年
3. **10bit-工作机器id**，用来记录工作机器id。
   - 可以部署在$2^{10}=1024$个节点，包括5位datacenterId和5位workerId
   - 5位（bit）可以表示的最大正整数是$2^5-1=31$，即可以用0、1、2、3、....31这32个数字，来表示不同的datecenterId或workerId
4. **12bit-序列号**，序列号，用来记录同毫秒内产生的不同id。
   - 12位（bit）可以表示的最大正整数是$2^{12}-1=4095$，即可以用0、1、2、3、....4094这

4095个数字，来表示同一机器同一时间截（毫秒)内产生的4095个ID序号。

由于在Java中64bit的整数是long类型，所以在Java中SnowFlake算法生成的id就是long来存储的。

6.4.2.1 SnowFlake可以保证

1. 所有生成的id按时间趋势递增
2. 整个分布式系统内不会产生重复id（因为有datacenterId和workerId来做区分）

6.5 4.5 功能实现

6.5.1 4.5.1 派单

> 使用redisTemplate操作zset将username以及workid压入zset中

```
redisTemplate.opsForZSet().add(TaxiConstant.TAXT_LINE_UP_KEY,
taxiBO.getUsername(), taxiBO.getId());
```

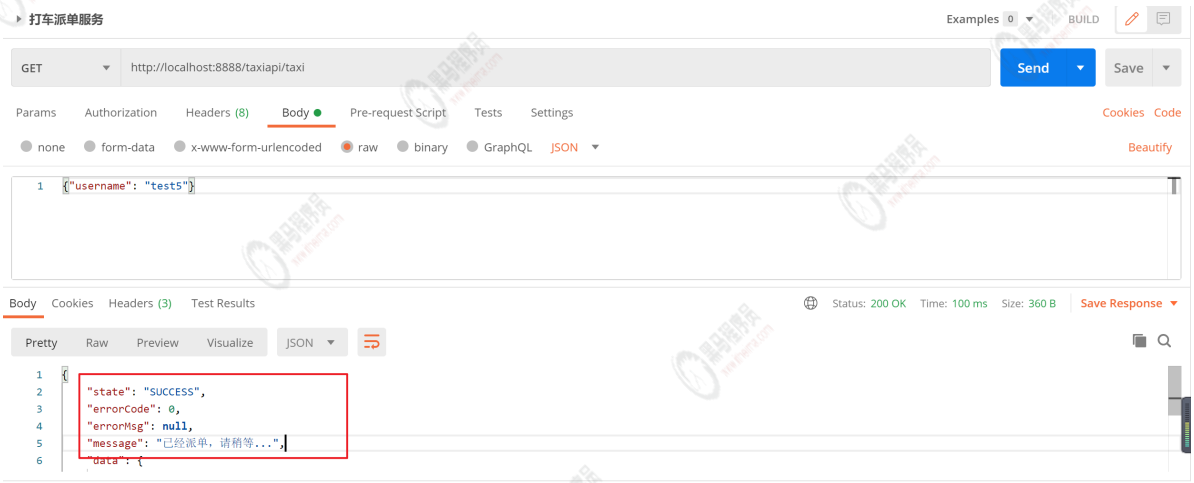
6.5.2 4.5.2 获取排队情况

> 使用redisTemplate操作zset获取username对应的排名

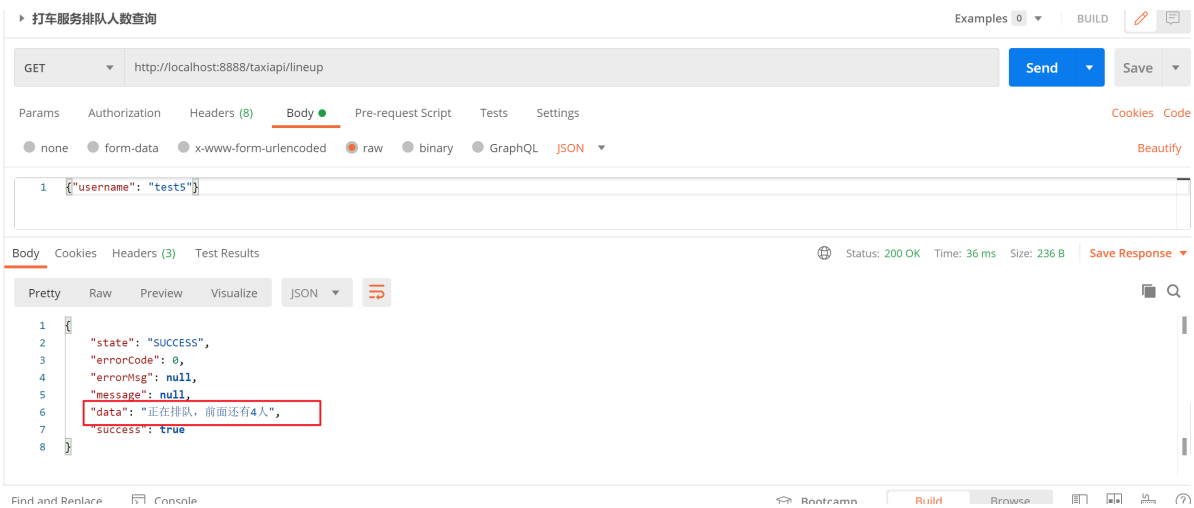
```
redisTemplate.opsForZSet().rank(TaxiConstant.TAXT_LINE_UP_KEY, username);
```

6.6 4.6 演示

6.6.1 4.6.4 派单



6.6.2 4.6.2 排队情况查询



7 5 消息推送

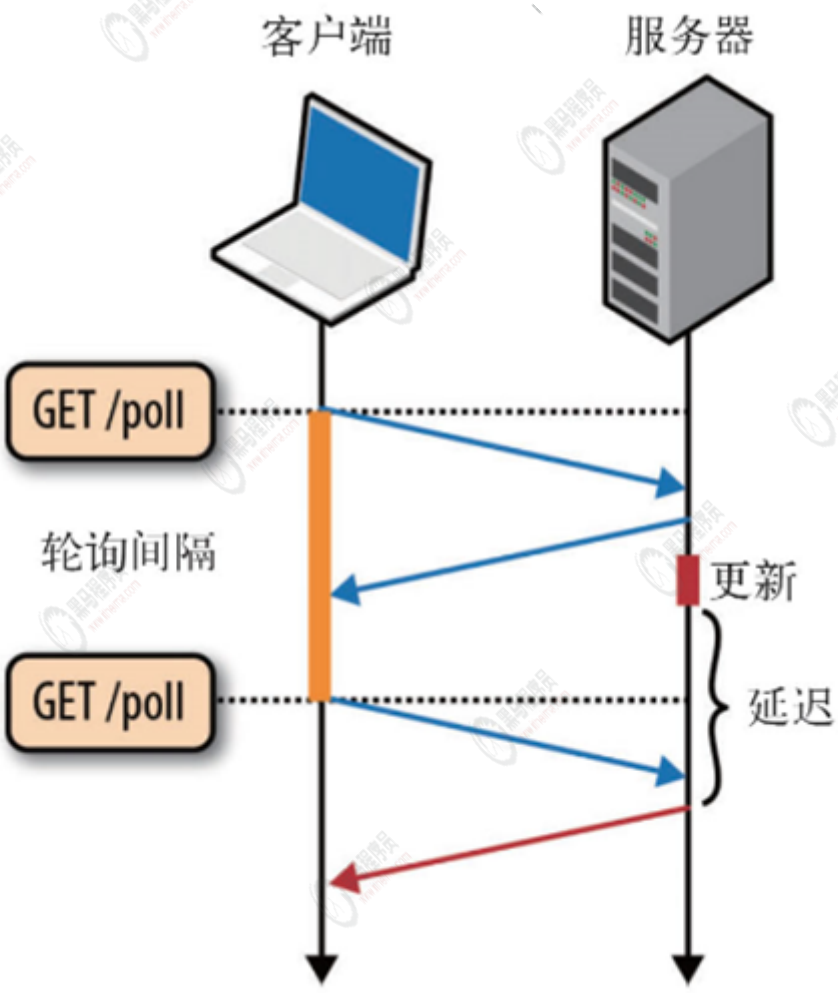
7.1 5.1 什么是消息推送

> 当我们使用http协议探知服务器上是否有内容更新，就必须频繁的从客户端到服务器端进行确认。而http一下的这些标准会成为一个瓶颈：

- 一条连接上只可以发送一个请求
- 请求只能从客户端开始。客户端不可以接收除了响应以外的指令。
- 请求 / 响应首部未经过压缩就直接进行传输。首部的信息越多，那么延迟就越大。
- 发送冗长的首部。每次互相发送相同的首部造成的浪费越多
- 可以任意选择数据压缩格式。非强制压缩发送

7.2 5.2 方案介绍

7.2.1 5.2.1 ajax短轮询

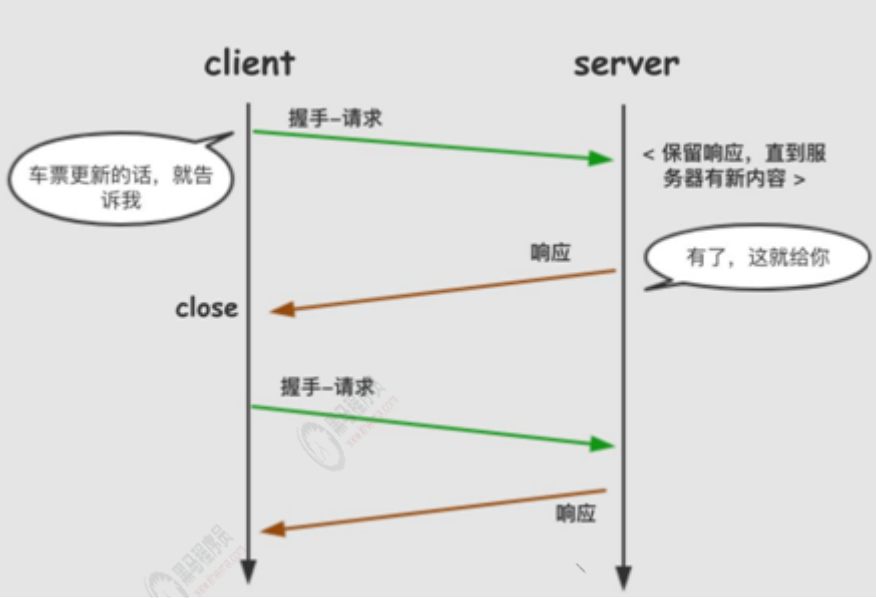


ajax（异步的javascript与xml技术）是一种有效利用javascript和dom的操作，以达到局部web页面的提花和加载的异步通信手段。和以前的同步通信相比，他只更新一部分页面，相应中传输饿数据量会因此的减少。

ajax轮询的原理是，让浏览器每隔一段时间就发送一次请求，询问服务器是否有新消息，而利用ajax实时的从服务器获取内容，有可能导致大量的请求产生。

特点：实现简单、短连接、数据同步不及时、对服务器资源会造成一定压力。此模式广泛应用于：扫描登录、扫码支付、天气更新等（腾讯、京东、阿里一直都在沿用此技术并日渐成熟和稳定）

7.2.2 5.2.2 长轮询



原理和ajax轮询差不多，都是采用轮询的方式，不过采用的是阻塞模型。也就是说，当客户端发起连接后，如果服务器端内容没有更新，将响应至于挂起状态，一直不回复response给客户端，知道有内容更新，再返回响应。

虽然可以做到实时更新，但是为了保留响应，一次连接饿持续时间也变长了。期间，为了维持连接会消费更多的资源。

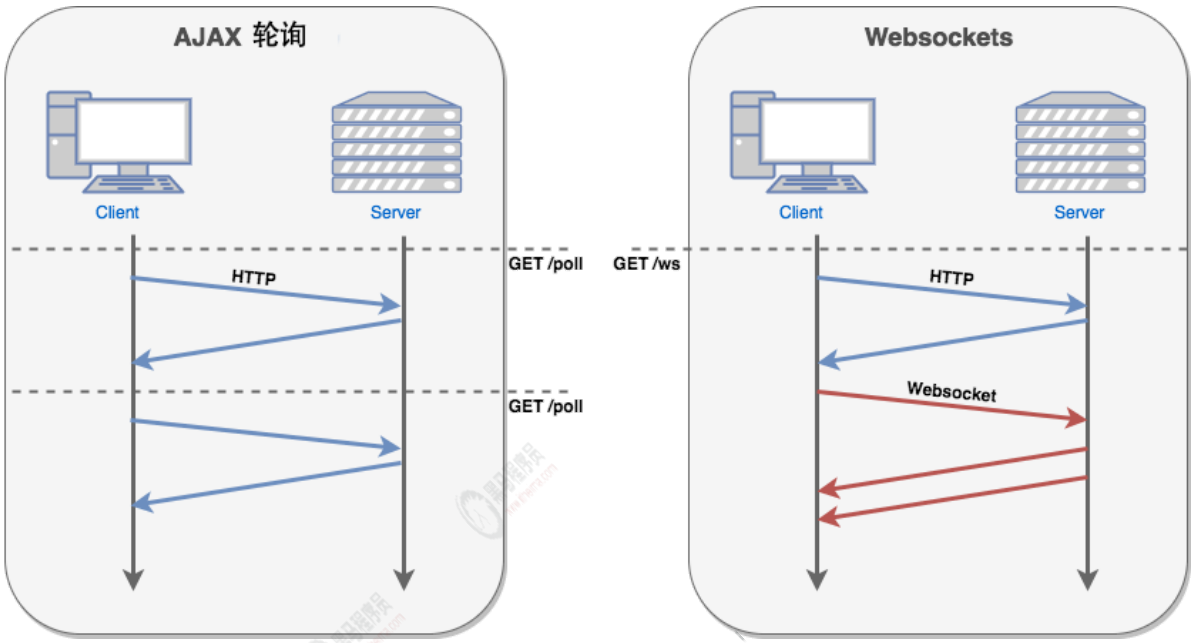
需要有很高的并发，也就是说同时接待客户的能力

从上面两种方式中，其实可以看出是再不断的建立http连接，然后等待服务器处理，可以体现出了http的特点：**被动性**，即：请求只能由客户端发起。服务器端不能主动联系客户端。

特点：无需浏览器或APP端任何单独插件支持、长连接，减少网络(三次)握手和四次挥手、对服务器资源要求较高等。此模式常用于实时消息轮播、金融数据即时刷新、数据图表实时刷新等。JAVA服务器端一般采用Servlet3支持的异步任务、延时结果（DeferedResult）等手段实现。

7.2.3 5.2.3 WebSocket

WebSocket 是 HTML5 开始提供的一种在单个 TCP 连接上进行全双工通讯的协议。WebSocket 使得客户端和服务器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。

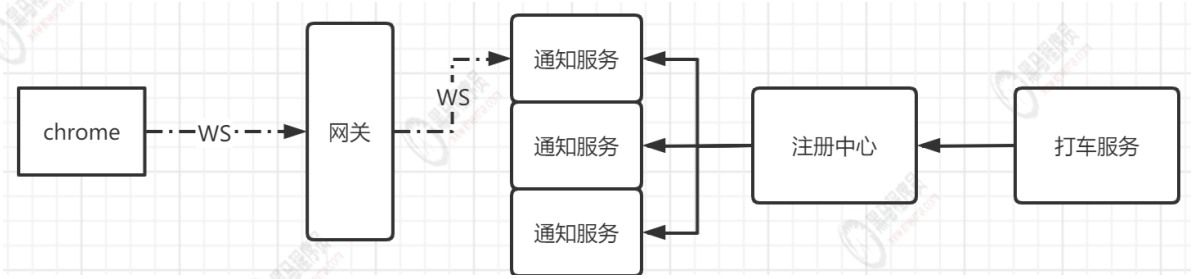


7.3 5.3 WS实现消息推送

> 通过上面我们实现了延时任务处理以及派单排队，但是如果将我们的异步处理结果推送给客户端呢

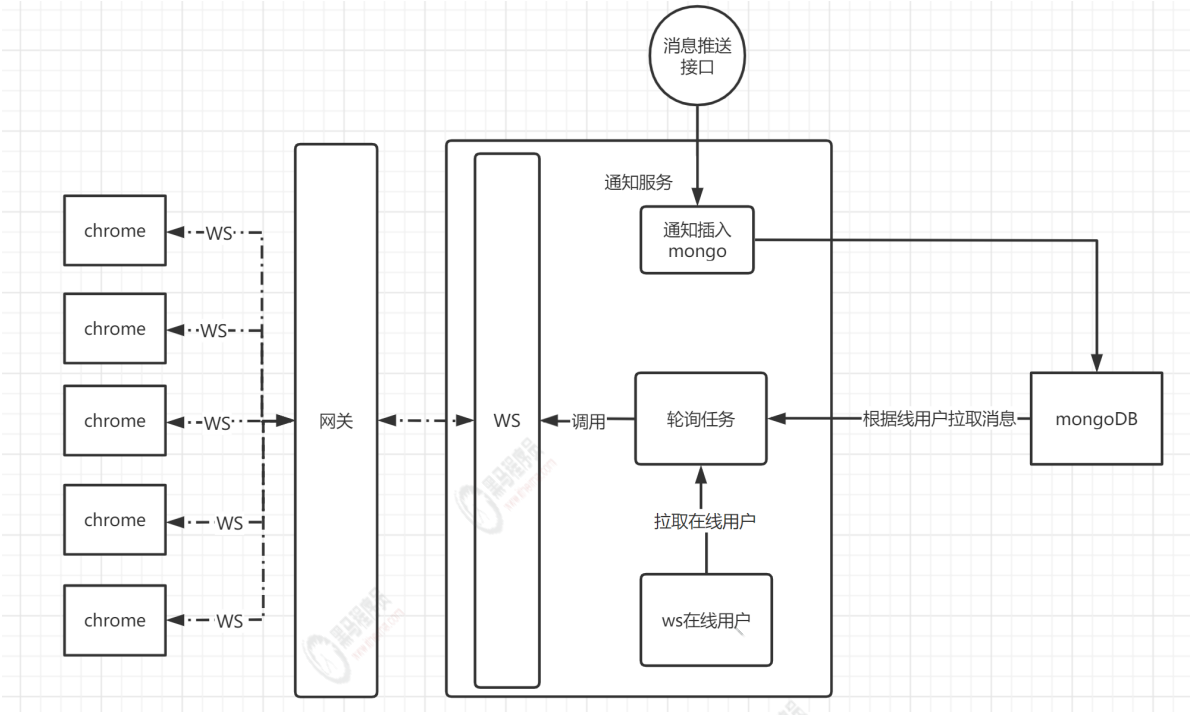
我们就需要使用消息推送技术，需要完成一下功能

- 将消息推送到指定的用户
- 对于未上线用户需要暂存数据，上线后推送



7.3.1 5.3.1 架构介绍

> 因为websocket是点对点的，而服务间调用是轮询的，无法实现微服务之间点对点的消息推送，我们使用定时任务来实现消息推送



1. 调用接口先将消息暂存到MongoDB中
2. 轮询任务首先拉取当前在线人员列表
3. 轮询任务通过在线人员列表到MongoDB中拉取在线用户的通知消息
4. 将消息通过WS推送到指定的用户

7.3.2 5.3.2 暂存数据

> 通过MongoDB将我们的消息数据暂存到数据库中，可以完成对于未上线消息暂存以及对分布式websocket的数据调度

7.3.2.1 5.3.2.1 什么是MongoDB

MongoDB是一个介于关系数据库和非关系数据库之间的产品，是非关系数据库当中功能最丰富，最像关系数据库的。它支持的数据结构非常松散，是类似json的bson格式，因此可以存储比较复杂的数据类型。Mongo最大的特点是它支持的查询语言非常强大，其语法有点类似于面向对象的查询语言，几乎可以实现类似关系数据库单表查询的绝大部分功能，而且还支持对数据建立索引。

7.3.2.2 5.3.2.2 插入数据

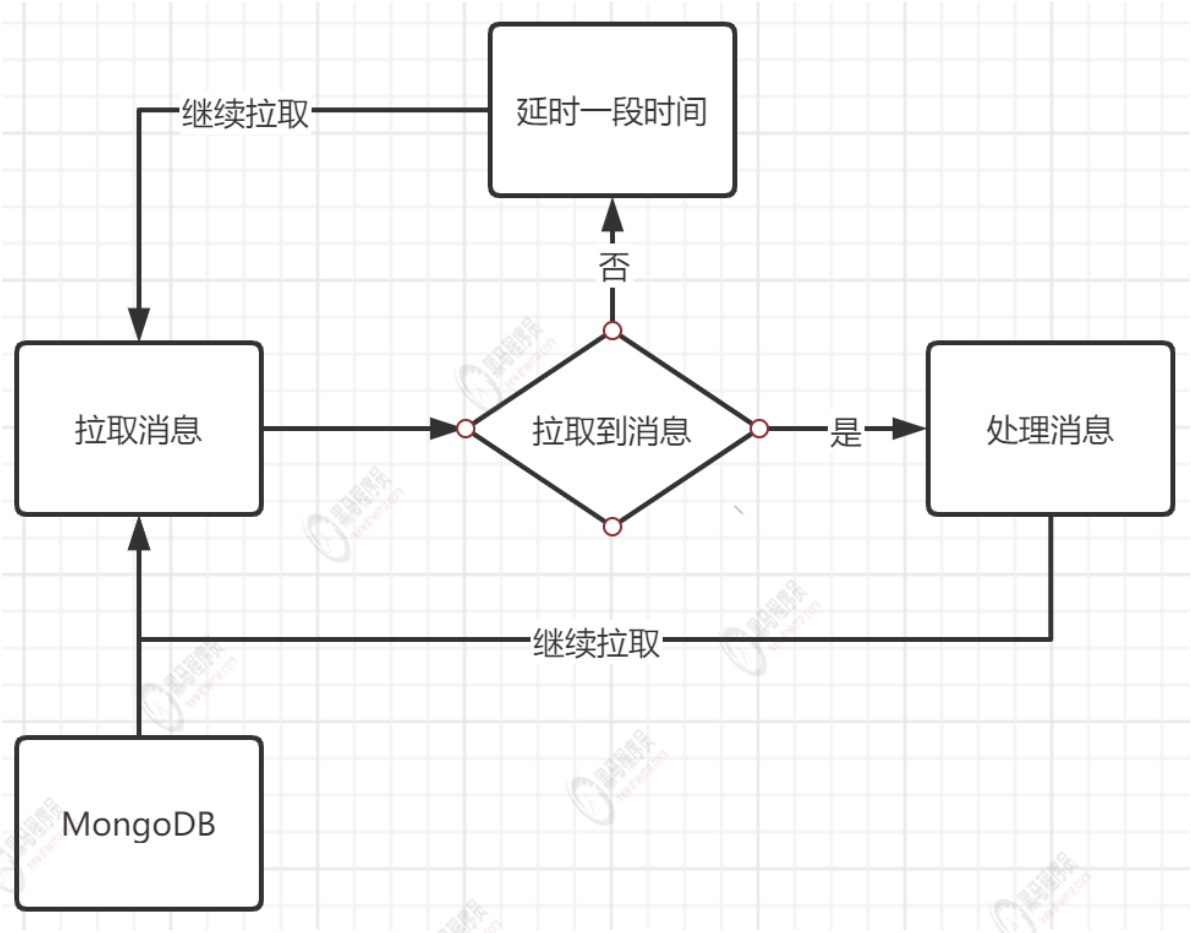
```
@Override
public void addMessage(PushMessage message) {
    mongoTemplate.save(message);
}
```

7.3.2.3 5.3.2.3 查询数据

```
@Override
public List<PushMessage> getMessageByUserNames(List<String> userNameList) {
    Query query = new Query(Criteria.where("username").in(userNameList));
    return mongoTemplate.findAllAndRemove(query, PushMessage.class);
}
```

7.3.3 5.4.1 轮询任务

> 轮询任务就是不断的搜索检查是否有新的消息，然后交给WS进行处理



7.3.3.1 5.4.1.1 代码实现

> 使用pull方式将MongoDB中的在线用户的暂存消息取出来，推送给在线用户

```
/**
 * 定时任务 推送暂存消息
 */
@Component
public class ScheduledTask {

    private static final Logger logger =
LoggerFactory.getLogger(ScheduledTask.class);

    @Autowired
    private PushService pushService;

    private static final ExecutorService executorService =
Executors.newFixedThreadPool(10);

    @Autowired
    private WebSocketServer webSocketServer;

    @PostConstruct
    public void init() {
        executorService.execute(() -> {
            autoPushMessage();
        });
```

```
    }


    /**
     * 自动推送消息
     */
    public void autoPushMessage() {
        //轮询并发送消息
        PollingRound.pollingPull(() -> {
            //获取最新需要推送的消息
            List<PushMessagePO> pushMessagesList = getPushMessages();
            //校验消息
            if (null != pushMessagesList && !pushMessagesList.isEmpty()) {
                logger.debug("推送消息线程工作工作中,推送数据条数:{}",
pushMessagesList.size());
                //推送消息
                webSocketServer.pushMessage(pushMessagesList);
                return PollingRound.delayLoop(100);
            }
            logger.debug("推送消息线程工作工作中,推送数据条数:{}", 0);
            return PollingRound.delayLoop(1000);
        });
    }

    public List<PushMessagePO> getPushMessages() {
        List<String> userNameList = webSocketServer.getInLineAccountIds();
        if (null != userNameList && !userNameList.isEmpty()) {
            //在MongoDB中获取当前在线用户的暂存消息
            List<PushMessagePO> pushMessageList =
pushService.getMessageByAccountIds(userNameList);
            //返回消息
            return pushMessageList;
        }
        return null;
    }
}
```