Rabbitmq消费端实战

消费者如何确认消费?

为什么要确认消费？ 默认情况下 消费者在拿到rabbitmq的消息时 已经自动确认这条消息已经消费了， 讲白话就是rabbitmq的队列里就会删除这条消息了， 但是 我们实际开发中 难免会遇到这种情况， 比如说 拿到这条消息 发现我处理不了 比如说 参数不对， 又比如说 我当前这个系统出问题了， 暂时不能处理这个消息， 但是 这个消息已经被你消费掉了 rabbitmq的队列里也删除掉了， 你自己这边又处理不了， 那么 ，这个消息就被遗弃了。 这种情况在实际开发中是不合理的， rabbitmq提供了解决这个问题的方案， 也就是我们上面所说的confirm模式 只是我们刚刚讲的是发送方的 这次我们来讲消费方的。

首先 我们在消费者这边（再强调一遍 我这里建议大家消费者和生产者分两个项目来做，包括我自己就是这样的，虽然一个项目也可以，我觉得分开的话容易理解一点）

设置一下消息确认为手动确认:

当然 我们要对我们的消费者监听器进行一定的配置的话， 我们需要先实例一个监听器的Container 也就是容器，那么我们的监听器（一个消费者里面可以实例多个监听器） 可以指定这个容器 那么我们只需要对这个Container（容器） 进行配置就可以了

首先得声明一个容器并且在容器里面指定消息确认为手动确认:

```
@Bean
public SimpleRabbitListenerContainerFactory
simpleRabbitListenerContainerFactory(ConnectionFactory connectionFactory){
    SimpleRabbitListenerContainerFactory simpleRabbitListenerContainerFactory =
            new SimpleRabbitListenerContainerFactory();
    //这个connectionFactory就是我们自己配置的连接工厂直接注入进来
    simpleRabbitListenerContainerFactory.setConnectionFactory(connectionFactory);
    //这边设置消息确认方式由自动确认变为手动确认
    simpleRabbitListenerContainerFactory.setAcknowledgeMode(AcknowledgeMode.MANUAL);
    return simpleRabbitListenerContainerFactory;
}
```

AcknowledgeMode关于这个类 就是一个简单的枚举类 我们来看看:

```
public enum AcknowledgeMode {
    NONE,
    MANUAL,
    AUTO;

    private AcknowledgeMode() {
    }

    public boolean isTransactionAllowed() { return this == AUTO || this == MANUAL; }

    public boolean isAutoAck() { return this == NONE; }

    public boolean isManual() { return this == MANUAL; }
```

3个状态 不确认 手动确认 自动确认

我们刚刚配置的就是中间那个 手动确认

既然现在是手动确认了 那么我们在处理完这条消息之后 得使这条消息确认：

```
@Component
public class TestListener  {


    //containerFactory:指定我们刚刚配置的容器
 @RabbitListener(queues = "testQueue",containerFactory =
"simpleRabbitListenerContainerFactory")
    public void getMessage(Message message, Channel channel) throws Exception{
        System.out.println(new String(message.getBody(),"UTF-8"));
        System.out.println(message.getBody());
        //这里我们调用了一个下单方法  如果下单成功了 那么这条消息就可以确认被消费了
        boolean f =placeAnOrder();
        if (f){
            //传入这条消息的标识， 这个标识由rabbitmq来维护 我们只需要从message中拿出来就可以
            //第二个boolean参数指定是不是批量处理的 什么是批量处理我们待会儿会讲到
            channel.basicAck(message.getMessageProperties().getDeliveryTag(),false);
        }else {
            //当然 如果这个订单处理失败了  我们也需要告诉rabbitmq 告诉他这条消息处理失败了 可以退回
也可以遗弃 要注意的是 无论这条消息成功与否  一定要通知 就算失败了 如果不通知的话 rabbitmq端会显示这条
消息一直处于未确认状态，那么这条消息就会一直堆积在rabbitmq端 除非与rabbitmq断开连接 那么他就会把这条
消息重新发给别人  所以 一定要记得通知!
            //前两个参数 和上面的意义一样， 最后一个参数 就是这条消息是返回到原队列 还是这条消息作废
就是不退回了。

 channel.basicNack(message.getMessageProperties().getDeliveryTag(),false,true);
            //其实 这个API也可以去告诉rabbitmq这条消息失败了  与basicNack不同之处 就是 他不能批量
处理消息结果 只能处理单条消息   其实basicNack作为basicReject的扩展开发出来的

 //channel.basicReject(message.getMessageProperties().getDeliveryTag(),true);
        }
    }
```

```
}
```

正常情况下的效果， 我就不演示给大家看了， 这里给大家看一个如果忘记退回消息的效果:

这里 我把消息确认的代码注释掉:

```
//            channel.basicAck(message.getMessageProperties().getDeliveryTag(),false);
```

然后调用生产者发送一条消息 我们来看管理页面:

Overview				Messages			Message rates			+/-
Virtual host	**Name**	Features	**State**	**Ready**	**Unacked**	**Total**	**incoming**	**deliver / get**	**ack**	
testhost	**testQueue**	D	idle	1	0	1	0.00/s	0.00/s	0.00/s	

这里能看到 有一条消息在rabbitmq当中 而且状态是ready

然后我们使用消费者来消费掉他 注意 这里我们故意没有告诉rabbitmq我们消费成功了 来看看效果

这里 消费的结果打印就不截图了 还是来看管理页面:

Overview				Messages			Message rates			+/-
Virtual host	**Name**	Features	**State**	**Ready**	**Unacked**	**Total**	**incoming**	**deliver / get**	**ack**	
testhost	**testQueue**	D	idle	0	1	1	0.00/s	0.00/s	0.00/s	

就算我们消费端消费了下次 但是能看到 这条消息还是会在rabbitmq当中 只是他的状态为 unacked 就是未确认

这就是我们刚刚说的那种情况 无论消费成功与否 一定要通知rabbitmq 不然就会这样 一直囤积在rabbitmq当中 直到连接断开为止.

消息预取

扯完消息确认 我们来讲一下刚刚所说的批量处理的问题

什么情况下回遇到批量处理的问题呢?

在这里 就要先扯一下rabbitmq的消息发放机制了

rabbitmq 默认 他会最快 以轮询的机制吧队列所有的消息发送给所有客户端 （如果消息没确认的话 他会添加一个Unacked的标识上图已经看过了）

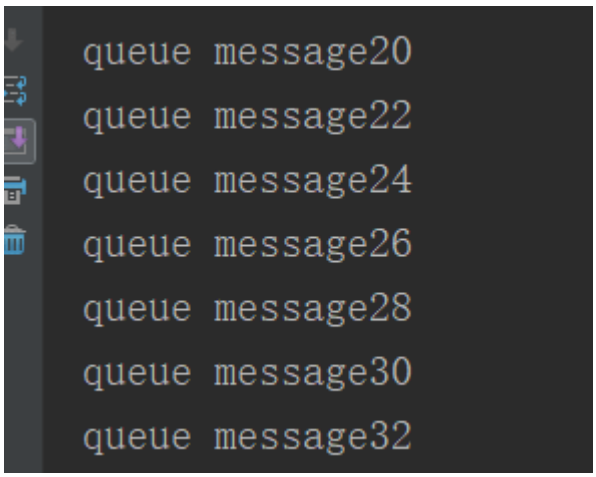
那么 这种机制会有什么问题呢， 对于Rabbitmq来讲 这样子能最快速的使自己不会囤积消息而对性能造成影响，但是 对于我们整个系统来讲， 这种机制会带来很多问题， 比如说 我一个队列有2个人同时在消费，而且他们处理能力不同， 我打个最简单的比方 有100个订单消息需要处理（消费） 现在有消费者A 和消费者B ， 消费者A消费一条消息的速度是 10ms 消费者B 消费一条消息的速度是15ms （ 当然 这里只是打比方） 那么 rabbitmq 会默认给消费者A B 一人50条消息让他们消费 但是 消费者A 他500ms 就可以消费完所有的消息 并且处于空闲状态 而 消费

者B需要750ms 才能消费完 如果从性能上来考虑的话 这100条消息消费完的时间一共是750ms（因为2个人同时在消费） 但是如果 在消费者A消费完的时候 能把这个空闲的性能用来和B一起消费剩下的信息的话， 那么这处理速度就会快非常多。

这个例子可能有点抽象， 我们通过代码来演示一下

我往Rabbitmq生产100条消息 由2个消费者来消费 其中我们让一个消费者在消费的时候休眠0.5秒（模拟处理业务的延迟） 另外一个消费者正常消费 我们来看看效果：

正常的那个消费者会一瞬间吧所有消息（50条）全部消费完（因为我们计算机处理速度非常快） 下图是加了延迟的消费者：



可能我笔记里面你看不出效果，这个你自己测试就会发现 其中一个消费者很快就处理完自己的消息了 另外一个消费者还在慢慢的处理 其实 这样严重影响了我们的性能了。

其实讲了这么多 那如何来解决这个问题呢?

我刚刚解释过了 造成这个原因的根本就是rabbitmq消息的发放机制导致的， 那么我们现在来讲一下解决方案: **消息预取**

什么是消息预取？ 讲白了以前是rabbitmq一股脑吧所有消息都均发给所有的消费者（不管你受不受得了） 而现在是在我消费者消费之前 先告诉rabbitmq **我一次能消费多少数据 等我消费完了之后告诉rabbitmq** rabbitmq再给我发送数据

在代码中如何体现?

在使用消息预取前 要注意一定要设置为手动确认消息， 原因参考上面划重点的那句话。

因为我们刚刚设置过了 这里就不贴代码了， 完了之后设置一下我们预取消息的数量 一样 是在容器（Container）里面设置：

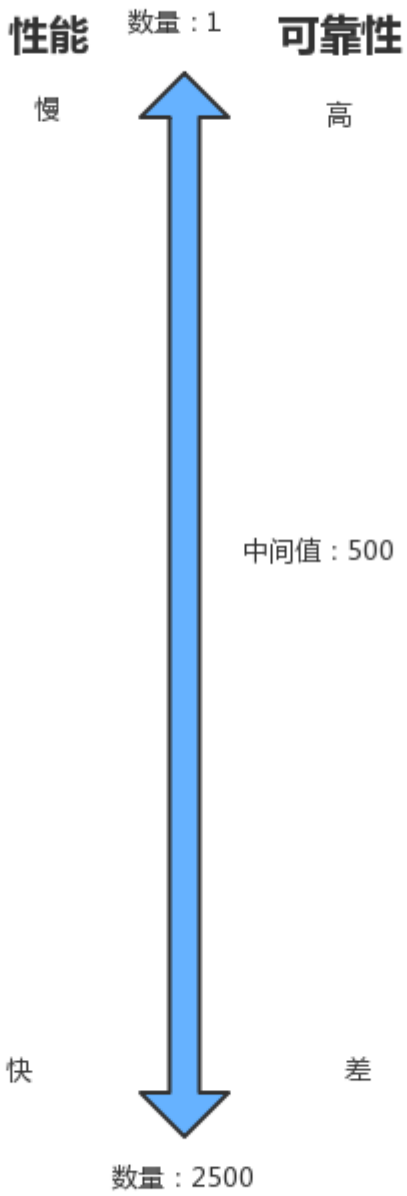
```
@Bean
public SimpleRabbitListenerContainerFactory
simpleRabbitListenerContainerFactory(ConnectionFactory connectionFactory){
    SimpleRabbitListenerContainerFactory simpleRabbitListenerContainerFactory =
            new SimpleRabbitListenerContainerFactory();
    simpleRabbitListenerContainerFactory.setConnectionFactory(connectionFactory);
    //手动确认消息
    simpleRabbitListenerContainerFactory.setAcknowledgeMode(AcknowledgeMode.MANUAL);
    //设置消息预取的数量
    simpleRabbitListenerContainerFactory.setPrefetchCount(1);
    return simpleRabbitListenerContainerFactory;
}
```

那么设置完之后是什么效果呢？ 还是刚刚那个例子 还是2个消费者 因为会在消费者返回消息的确认之后 rabbitmq 才会继续发送消息给客户端 而且客户端的消息累计量不会超过我们刚刚设置预取的数量， 所以我们再运行同样的例子的话 会发现 A消费者消费完99条消息了 B消费者才消费1条 （因为B消费者休眠了0.5秒才消费完{返回消息确认} 但是0.5秒之内A消费者就已经把所有消息消费完毕了 当然 如果计算机处理速度较慢这个结果可能会有差异,效果大概就是A消费者会处理大量消息）

我这里的效果就是B消费者只消费一条消息 A消费者就消费完了， 效果图就不发了 这里同学们尽量自己测试一下或者改变一下参数看看效果。

关于这个预取的数量如何设置呢？ 我们发现 如果设置为1 能极大的利用客户端的性能（我消费完了就可以赶紧消费下一条 不会导致忙的很忙 闲的很闲） 但是， 我们每消费一条消息 就要通知一次rabbitmq 然后再取出新的消息， 这样对于rabbitmq的性能来讲 是非常不合理的 所以这个参数要根据业务情况设置

我根据我查阅到的资料然后加以测试， 这个数值的大小与性能成正比 但是有上限，与数据可靠性，以及我们刚刚所说的客户端的利用率成反比 大概如下图：



那么批量确认， 就是对于我们预取的消息，进行统一的确认。

死信交换机

我们来看一段代码:

```
channel.basicNack(message.getMessageProperties().getDeliveryTag(),false,true);
```

我们上面解释过 这个代码是消息处理失败的确认 然后第三个参数我有解释过是消息是否返回到原队列， 那么问题来了，如果 没有返回给原队列 那么这条消息就被作废了?

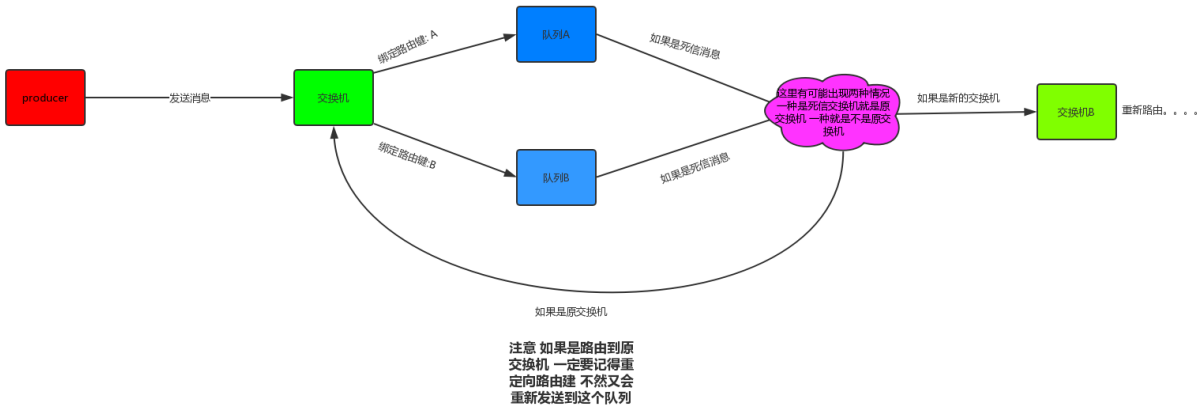
rabbitmq考虑到了这个问题提供了解决方案： **死信交换机**(有些人可能叫作垃圾回收器,垃圾交换机等)

死信交换机有什么用呢? 在创建队列的时候 可以给这个队列附带一个交换机， 那么这个队列作废的消息就会被重新发到附带的交换机，然后让这个交换机重新路由这条消息

理论是这样， 代码如下:

```
@Bean
    public Queue queue() {
        Map<String,Object> map = new HashMap<>();
        //设置消息的过期时间 单位毫秒
        map.put("x-message-ttl",10000);
        //设置附带的死信交换机
        map.put("x-dead-letter-exchange","exchange.dlx");
        //指定重定向的路由建 消息作废之后可以决定需不需要更改他的路由建 如果需要 就在这里指定
        map.put("x-dead-letter-routing-key","dead.order");
        return new Queue("testQueue", true,false,false,map);
    }
```

大概是这样的一个效果:



其实我们刚刚发现 所谓死信交换机， 只是对应的队列设置了对应的交换机是死信交换机， 对于交换机来讲， 他还是一个普通的交换机 。

下面会列出rabbitmq的常用配置:

队列配置:

参数名	配置作用
x-dead-letter-exchange	死信交换机
x-dead-letter-routing-key	死信消息重定向路由键
x-expires	队列在指定毫秒数后被删除
x-ha-policy	创建HA队列
x-ha-nodes	HA队列的分布节点
x-max-length	队列的最大消息数
x-message-ttl	毫秒为单位的消息过期时间，队列级别
x-max-priority	最大优先值为255的队列优先排序功能

消息配置：

参数名	配置作用
content-type	消息体的MIME类型，如application/json
content-encoding	消息的编码类型
message-id	消息的唯一性标识，由应用进行设置
correlation-id	一般用做关联消息的message-id，常用于消息的响应
timestamp	消息的创建时刻，整形，精确到秒
expiration	消息的过期时刻， 字符串，但是呈现格式为整型，精确到秒
delivery-mode	消息的持久化类型，1为非持久化，2为持久化，性能影响巨大
app-id	应用程序的类型和版本号
user-id	标识已登录用户，极少使用
type	消息类型名称，完全由应用决定如何使用该字段
reply-to	构建回复消息的私有响应队列
headers	键/值对表，用户自定义任意的键和值
priority	指定队列中消息的优先级