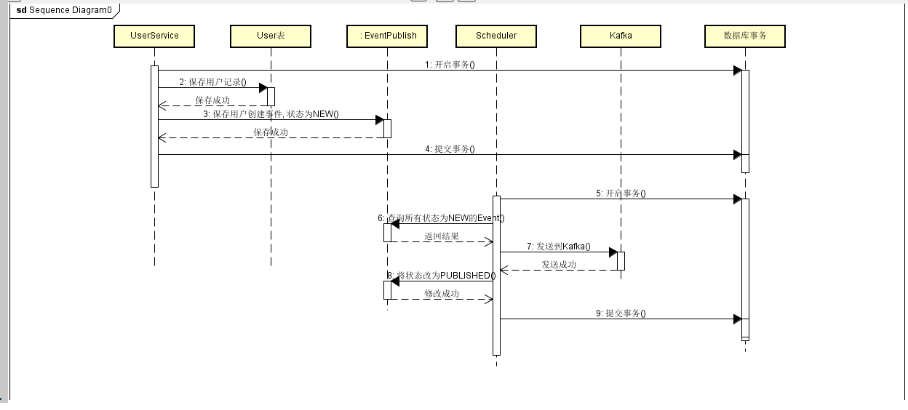
事务补偿：

不同质的数据源，没有办法使用多数据源的方式来实现事务一致性。

市面上分布式框架两段，三段提交，但是性能有点问题。而且需要一些容器来处理。



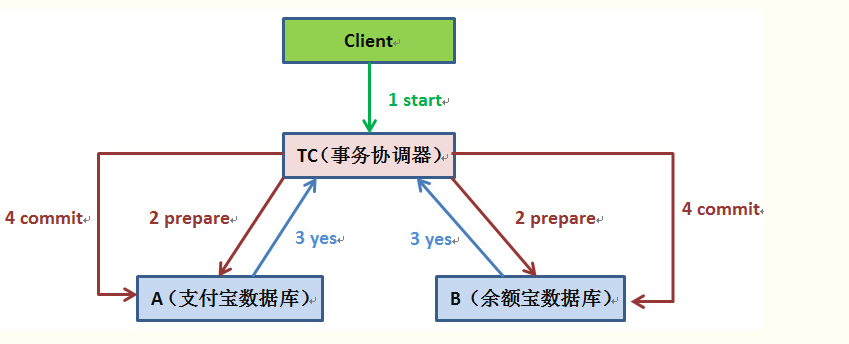
和2PC/3PC这种同步事务处理的方式相比, 这种异步事务处理方式具有异步系统通常都有的优点:

1. 事务吞吐量大. 因为不需要等待其他数据源响应.  
2. 容错性好. A服务在发布事件的时候, B服务甚至可以不在线.

缺点:  
1. 编程与调试较复杂.  
2. 容易出现较多的中间状态. 比如上面的例子, 在用户服务已经保存了用户并发布了事件, 但是代金券服务还没来得及处理之前, 用户如果登录系统, 会发现自己是没有代金券的. 这种情况可能在有些业务中是能够容忍的, 但是有些业务却不行. 所以开发之前要考虑好.

二段提交事务：

两阶段提交协议（Two-phase Commit，2PC）经常被用来实现分布式事务。一般分为协调器C和若干事务执行者Si两种角色，这里的事务执行者就是具体的数据库，协调器可以和事务执行器在一台机器上。



1） 我们的应用程序（client）发起一个开始请求到TC***；（通过日志来保证）***

2） TC先将<prepare>消息写到本地日志，之后向所有的Si发起<prepare>消息。以支付宝转账到余额宝为例，TC给A的prepare消息是通知支付宝数据库相应账目扣款1万，TC给B的prepare消息是通知余额宝数据库相应账目增加1w。为什么在执行任务前需要先写本地日志，主要是为了故障后恢复用，***本地日志起到现实生活中凭证 的效果，如果没有本地日志（凭证），出问题容易死无对证***；

3） Si收到<prepare>消息后，执行具体本机事务，但不会进行commit，如果成功返回<yes>，不成功返回<no>。同理，返回前都应把要返回的消息写到日志里，当作凭证。

4） TC收集所有执行器返回的消息，如果所有执行器都返回yes，那么给所有执行器发生送commit消息，执行器收到commit后执行本地事务的commit操作；如果有任一个执行器返回no，那么给所有执行器发送abort消息，执行器收到abort消息后执行事务abort操作。

注：TC或Si把发送或接收到的消息先写到日志里，主要是为了故障后恢复用。如某一Si从故障中恢复后，先检查本机的日志，如果已收到<commit >，则提交，如果<abort >则回滚。如果是<yes>，则再向TC询问一下，确定下一步。如果什么都没有，则很可能在<prepare>阶段Si就崩溃了，因此需要回滚。

现如今实现基于两阶段提交的分布式事务也没那么困难了，如果使用[Java](http://lib.csdn.net/base/javaee" \o "Java EE知识库" \t "_blank)，那么可以使用开源软件atomikos(<http://www.atomikos.com/>)来快速实现。

不过但凡使用过的上述两阶段提交的同学都可以发现性能实在是太差，根本不适合高并发的系统。为什么？

* 1）两阶段提交涉及多次节点间的网络通信，通信时间太长！
* 2）事务时间相对于变长了，锁定的资源的时间也变长了，造成资源等待时间也增加好多！

正是由于分布式事务存在很严重的性能问题，大部分高并发服务都在避免使用，往往通过其他途径来解决数据一致性问题。

**补充:**

之前看多阿里大神程立的一个关于分布式事务的文档，目前使用较多的分布式事务解决方案有几种：  
一、结合MQ消息中间件实现的可靠消息最终一致性  
二、TCC补偿性事务解决方案  
三、最大努力通知型方案  
  
第一种方案：可靠消息最终一致性，需要业务系统结合MQ消息中间件实现，在实现过程中需要保证消息的成功发送及成功消费。即需要通过业务系统控制MQ的消息状态

第二种方案：TCC补偿性，分为三个阶段TRYING-CONFIRMING-CANCELING。每个阶段做不同的处理。  
TRYING阶段主要是对业务系统进行检测及资源预留  
CONFIRMING阶段是做业务提交，通过TRYING阶段执行成功后，再执行该阶段。默认如果TRYING阶段执行成功，CONFIRMING就一定能成功。  
CANCELING阶段是回对业务做回滚，在TRYING阶段中，如果存在分支事务TRYING失败，则需要调用CANCELING将已预留的资源进行释放。

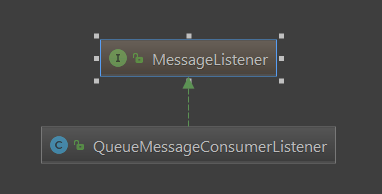
第三种方案：最大努力通知型，这种方案主要用在与第三方系统通讯时，比如：调用微信或支付宝支付后的支付结果通知。这种方案也是结合MQ进行实现，例如：通过MQ发送http请求，设置最大通知次数。达到通知次数后即不再通知。  
具体的案例你也可以参考下这篇博客，它上面的这个案例就是结合电商支付做的系统分布式事务实现案例：http://www.roncoo.com/article/detail/124243

基于事务消息的MQ方案是目前公认的较为理想的分布式事务解决方案，各大电商都在应用这一方案。种方式适合的业务场景广泛，而且比较可靠。不过这种方式技术实现的难度比较大。目前主流的开源MQ（ActiveMQ、RabbitMQ、Kafka）均未实现对事务消息的支持，所以需二次开发或者新造轮子。

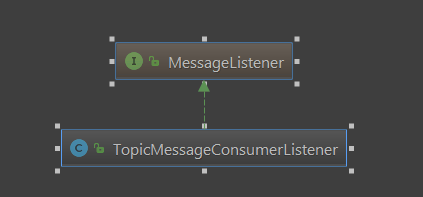
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*ebao 现有代码实现\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

JMS的事务补偿：

IN

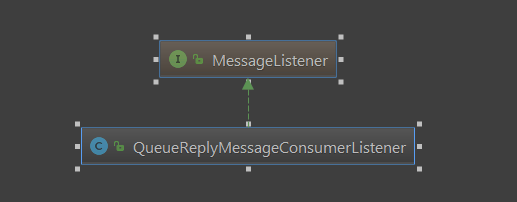


this.eventuallyConsistentService.processInCommand(command);



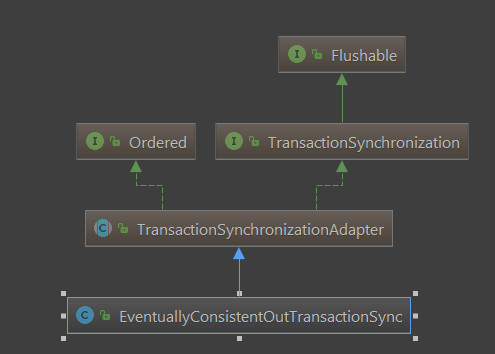
this.callBack.execute(receivedMessage);

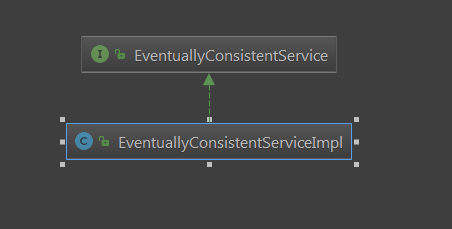
Out：



this.replyCallback.execute(receivedMessage);

Out：





RabbitMQ：

1. AMQP只是对消息发送，和消息确认有事务。对于exchange，queue，route等的创建，删除，修改等都是没有事务的。
2. 消息的确认事务和 消息的消费事务不是同一个，是分开管理。所以消息消费方可以在后面的事务中reject 消息，即使消息确认的消息已经发送。
3. AMQP只能对单个消息队列的消息有事务保护作用，也就是如果route对应的有多个，或者广播的时候，都没有事务。
4. AMQP对消息的发送和确认都是异步的。
5. RabbitMQ 是通过commit-ok的时候，才确认消息能客户端消费，或者持续化到磁盘。

同时，只是表示消息被接受到了，并不是被消费掉了。

服务器端可能失败复活确认信息，所以客户端可能二次消费到相同的信息。

Retry：

StatefulRetryOperationsInterceptor

消息confirm 和reply：

<https://www.rabbitmq.com/blog/2011/02/10/introducing-publisher-confirms/>

<https://docs.spring.io/spring-amqp/docs/1.4.5.RELEASE/reference/html/amqp.html>

<https://github.com/pzxwhc/MineKnowContainer/issues/49>

# 测试以及输出结果

当发送 curl localhost:8080/test/123 的时候，也就是说，发送的交换机和路由键都是正确的，结果如下：

Received111 <123 from RabbitMQ!>

================

correlationData = null

ack = true

cause = null

================

当发送 curl localhost:8080/test1/123 的时候，也就是说，发送的交换机是正确的，但是路由键是错误的，结果如下：

================

message = (Body:'123 from RabbitMQ!'MessageProperties [headers={}, timestamp=null, messageId=null, userId=null, appId=null, clusterId=null, type=null, correlationId=null, replyTo=null, contentType=text/plain, contentEncoding=UTF-8, contentLength=0, deliveryMode=PERSISTENT, expiration=null, priority=0, redelivered=null, receivedExchange=null, receivedRoutingKey=null, deliveryTag=0, messageCount=null])

replyCode = 312

replyText = NO\_ROUTE

exchange = testPublishModeExchange

routingKey = key11

================

================

correlationData = null

ack = true

cause = null

================

当发送 curl localhost:8080/test2/123 的时候，也就是说，发送的交换机是错误的，但是路由键是正确的，结果如下：

================

correlationData = null

ack = false

cause = channel error; protocol method: #method<channel.close>(reply-code=404, reply-text=NOT\_FOUND - no exchange 'testPublishModeExchange1' in vhost '/', class-id=60, method-id=40)

================

2016-04-21 10:36:07.714 ERROR 43884 --- [ 127.0.0.1:5672] o.s.a.r.c.CachingConnectionFactory : Channel shutdown: channel error; protocol method: #method<channel.close>(reply-code=404, reply-text=NOT\_FOUND - no exchange 'testPublishModeExchange1' in vhost '/', class-id=60, method-id=40)

当发送 curl localhost:8080/test2/123 的时候，也就是说，发送的交换机是错误的，但是路由键是错误的，结果如下：

================

correlationData = null

ack = false

cause = channel error; protocol method: #method<channel.close>(reply-code=404, reply-text=NOT\_FOUND - no exchange 'testPublishModeExchange1' in vhost '/', class-id=60, method-id=40)

================

2016-04-21 10:36:48.269 ERROR 43884 --- [ 127.0.0.1:5672] o.s.a.r.c.CachingConnectionFactory : Channel shutdown: channel error; protocol method: #method<channel.close>(reply-code=404, reply-text=NOT\_FOUND - no exchange 'testPublishModeExchange1' in vhost '/', class-id=60, method-id=40)

结论：观察如上，我们可以发现：

* ***如果我们发送的消息到达不了交换机，也就是说，发送的交换机写错误了，那么会马上回调 confirm 方法，并且 ack = false，同时也会报给你 cause***，如 no exchange 'testPublishModeExchange1' in vhost '/'。
* 如果我们发送的消息到达交换机，但是路由键写错了，交换机转发不到队列中，confirm 会被回调，并且显示 ack = true，意思是 交换机正确收到了消息，但是，同时 returnedMessage 方法也会被调用，会把我们发送的消息返回来。

# ConfirmCallback vs ReturnCallback

个人一开始对这两者是比较疑惑的，参考 [stackoverflow](http://stackoverflow.com/questions/30034092/spring-amqp-return-callback-vs-retry-callback) 的解释：

Returns are when the broker returns a message because it's undeliverable

(no matching bindings on the exchange to which the message was published,

and the mandatory bit is set).

Confirms are when the broker sends an ack back to the publisher,

indicating that a message was successfully routed.

再结合自己的测试，自己对 这两者的一个理解是，

* ***confirm 主要是用来判断消息是否有正确到达交换机，如果有，那么就 ack 就返回 true；如果没有，则是 false。***
* ***return 则表示如果你的消息已经正确到达交换机但是后续处理出错了（比如没有达到队列），那么就会回调 return，并且把信息送回给你（前提是需要设置了 Mandatory，不设置那么就丢弃）；如果消息没有到达交换机，那么不会调用 return 的东西。***

3.3.2 Publisher Confirms and Returns

通过接口PublisherCallbackChannelConnectionFactory申明，只有CachingConnectionFactory实现改接口里面消息的确认和return

确认模式：

/\*\*

\* No acks - {@code autoAck=true} in {@code Channel.basicConsume()}.

\*/

NONE,

/\*\*

\* Manual acks - user must ack/nack via a channel aware listener.

\*/

MANUAL,

/\*\*

\* Auto - the container will issue the ack/nack based on whether

\* the listener returns normally, or throws an exception.

\* <p><em>Do not confuse with RabbitMQ {@code autoAck} which is

\* represented by {@link #NONE} here</em>.

\*/

AUTO;

Reply：

broker 将在下面的情况中对消息进行 confirm ：

* **broker 发现当前消息无法被路由到指定的 queues 中**（如果设置了 mandatory 属性，则 broker 会先发送 basic.return）

**》》》》：**这个是对的。这样可以利用没有Return 有confirm的情况下判断是正常到达

* **非持久属性的消息到达了其所应该到达的所有 queue 中（和镜像 queue 中）**
* **持久消息到达了其所应该到达的所有 queue 中（和镜像 queue 中），并被持久化到了磁盘（被 fsync）**

**》》》》： 到达exchange不是queue？**

* 持久消息从其所在的所有 queue 中被 consume 了（如果必要则会被 acknowledge）？

**》》》》：这个好像一直没有反应，是解耦的，通过dead letter相应的来处理。**

**确认模式的必要性：（这里主要向publish 方确认，来确保如下过程的正常）**

broker 会丢失持久化消息，如果 broker 在将上述消息写入磁盘前异常。在一定条件下，这种情况会导致 broker 以一种奇怪的方式运行。例如，考虑下述情景：

   1.  一个 client 将持久消息 publish 到持久 queue 中  
   2.  另一个 client 从 queue 中 consume 消息（注意：该消息具有持久属性，并且 queue 是持久化的），当尚未对其进行 ack  
   3.  broker 异常重启  
   4.  client 重连并开始 consume 消息

   在上述情景下，client 有理由认为消息需要被（broker）重新 deliver 。但这并非事实：重启（有可能）会令 broker 丢失消息。为了确保持久性，client 应该使用 confirm 机制。如果 publisher 使用的 channel 被设置为 confirm 模式，publisher 将不会收到已丢失消息的 ack（这是因为 consumer 没有对消息进行 ack ，同时该消息也未被写入磁盘）。

**3. 编程实现**

首先要区别AMQP协议mandatory和immediate标志位的作用。

mandatory和immediate是AMQP协议中basic.pulish方法中的两个标志位，它们都有当消息传递过程中不可达目的地时将消息返回给生产者的功能。具体区别在于：

1. mandatory标志位

当mandatory标志位设置为true时，如果exchange根据自身类型和消息routeKey无法找到一个符合条件的queue，那么会调用basic.return方法将消息返还给生产者；当mandatory设为false时，出现上述情形broker会直接将消息扔掉。

2. immediate标志位

当immediate标志位设置为true时，如果exchange在将消息route到queue(s)时发现对应的queue上没有消费者，那么这条消息不会放入队列中。当与消息routeKey关联的所有queue(一个或多个)都没有消费者时，该消息会通过basic.return方法返还给生产者。

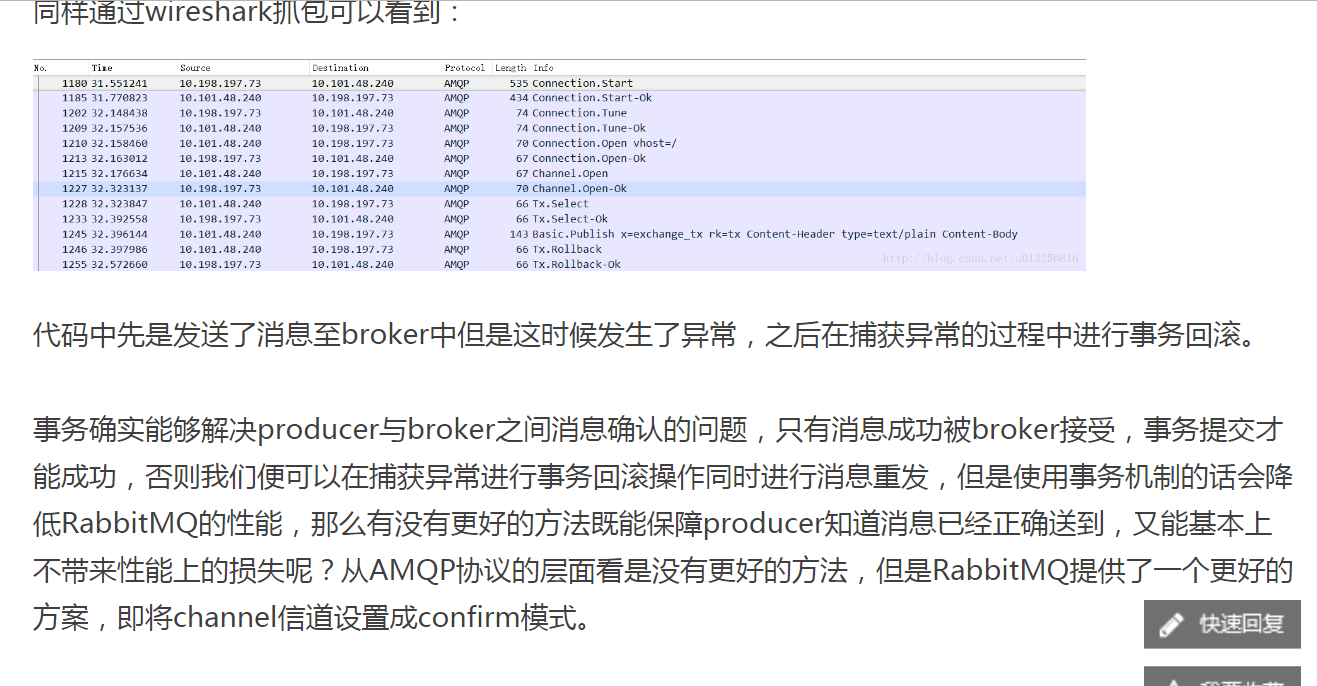
具体的代码参考请参考参考资料1.

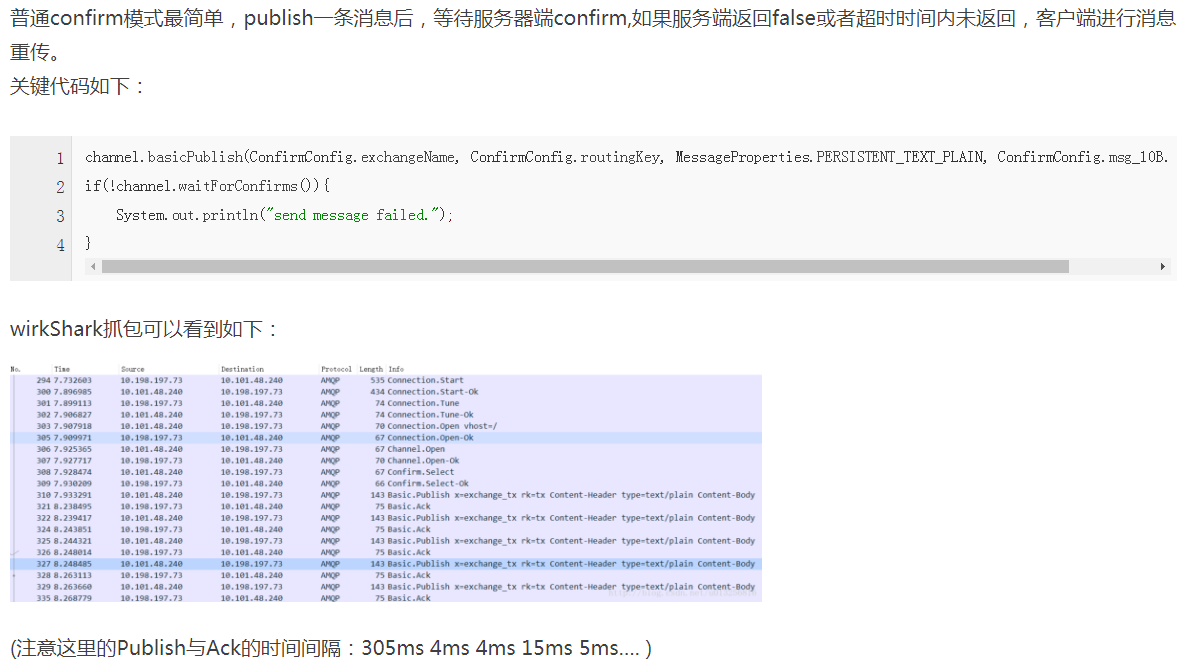
[**RabbitMQ之消息确认机制（事务+Confirm）**](http://blog.csdn.net/u013256816/article/details/55515234)

http://blog.csdn.net/u013256816/article/details/55515234

RabbitMQ中与事务机制有关的方法有三个：txSelect(), txCommit()以及txRollback(), txSelect用于将当前channel设置成transaction模式，txCommit用于提交事务，txRollback用于回滚事务，在通过txSelect开启事务之后，我们便可以发布消息给broker代理服务器了，如果txCommit提交成功了，则消息一定到达了broker了，如果在txCommit执行之前broker异常崩溃或者由于其他原因抛出异常，这个时候我们便可以捕获异常通过txRollback回滚事务了







Dead Letter Exchanges：（死信道的方案）

对于投递失败的信息，包括一下三种情况：

Messages from a queue can be 'dead-lettered'; that is, republished to another exchange when any of the following events occur:

*1）The message is rejected (basic.reject or basic.nack) with requeue=false,*

*2） The TTL for the message expires; or*

*3） The queue length limit is exceeded.*

Configuration using arguments

To set the dead letter exchange for a queue, set the x-dead-letter-exchange argument to the name of the exchange:

*channel.exchangeDeclare("some.exchange.name", "direct");*

*Map<String, Object> args = new HashMap<String, Object>();*

*args.put("x-dead-letter-exchange", "some.exchange.name");*

*channel.queueDeclare("myqueue", false, false, false, args);*

消息消费方确认方式：

<http://www.rabbitmq.com/confirms.html>

Consumer Acknowledgements and Publisher Confirms

人工模式：

@RabbitListener(queues = "${eventqueue}")

public void receiveMessage(Order order, Channel channel,

@Header(AmqpHeaders.DELIVERY\_TAG) long tag) throws Exception {

...

}

spring.rabbitmq.listener.acknowledge-mode=manual

@RabbitListener(queues = "so38728668")

public void receive(String payload, Channel channel, @Header(AmqpHeaders.DELIVERY\_TAG) long tag)

throws IOException {

System.out.println(payload);

channel.basicAck(tag, false);

latch.countDown();

}

**如果这个地方**channel.basicAck(tag, false);不确认的话，就一直保留在队列中。

默认的方式是auto，也就是发送到客户端，如果报错的话，就进行retry，retry之后就不再保留了。

[**rabbitmq消息一致性问题**](http://rd-030.iteye.com/blog/2339875)

**http://rd-030.iteye.com/blog/2339875**

发送者发送消息出来，在数据一致性的要求下，我们通常认为必须达到以下条件

1. broker持久化消息

2. publisher知道消息已经成功持久化

首先，我们可以采用事务来解决此问题。每个消息都必须经历以上两个步骤，就算一次事务成功。

事务是同步的。因此，如果采用事务，发送性能必然很差。官方给出来的性能是：

引用

It takes a bit more than 4 minutes to publish 10000 messages.

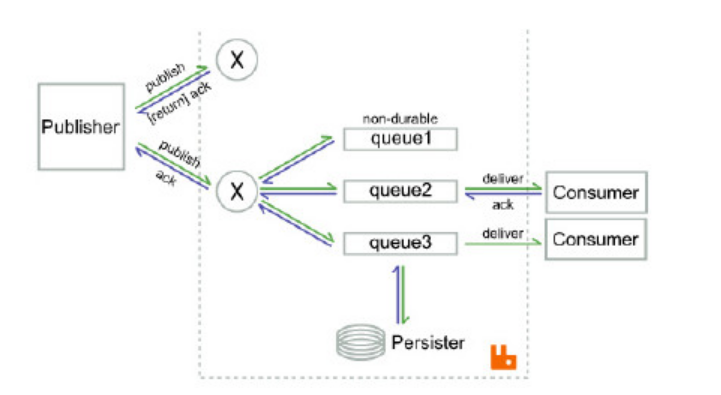
我们可以采用异步的方式来解决此问题。publisher发送消息后，不进行等待，而是异步监听是否成功。这种方式又分为两种模式，

一种是return，

另一种是confirm.

前一种是publisher发送到exchange后，异步收到消息。

第二种是publisher发送消息到exchange,queue,consumer收到消息后才会收到异步收到消息。可见，第二种方式更加安全可靠。



异步的方法的效率是事务方法效率的100倍

引用

It takes a bit more than 2 seconds to publish 10000 messages.

但是，异步也存在些局限性。如果一旦出现broker挂机或者网络不稳定，broker已经成功接收消息，但publisher并没有收到confirm或return.

这时，对于publisher来说，只能重发消息解决问题。而在这里面，我们会发生

引用

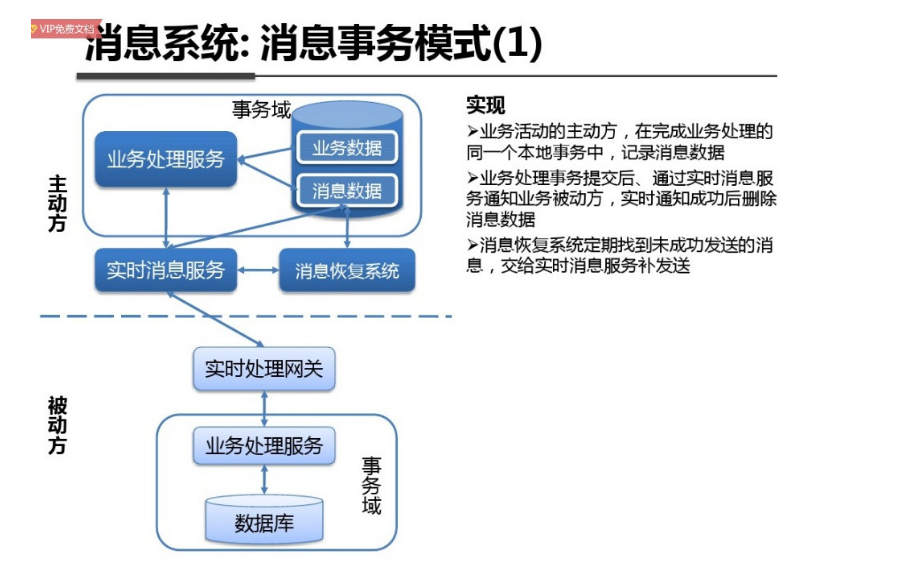
重复消息的问题。

当然，如果业务类型要求数据一致性非常高，可以采用低效率的事务型解决方案

分布式事务的几种方式：

<http://kaimingwan.com/post/fen-bu-shi/fen-bu-shi-shi-wu-de-dian-xing-chu-li-fang-shi-2pc-tcc-yi-bu-que-bao-he-zui-da-nu-li-xing>

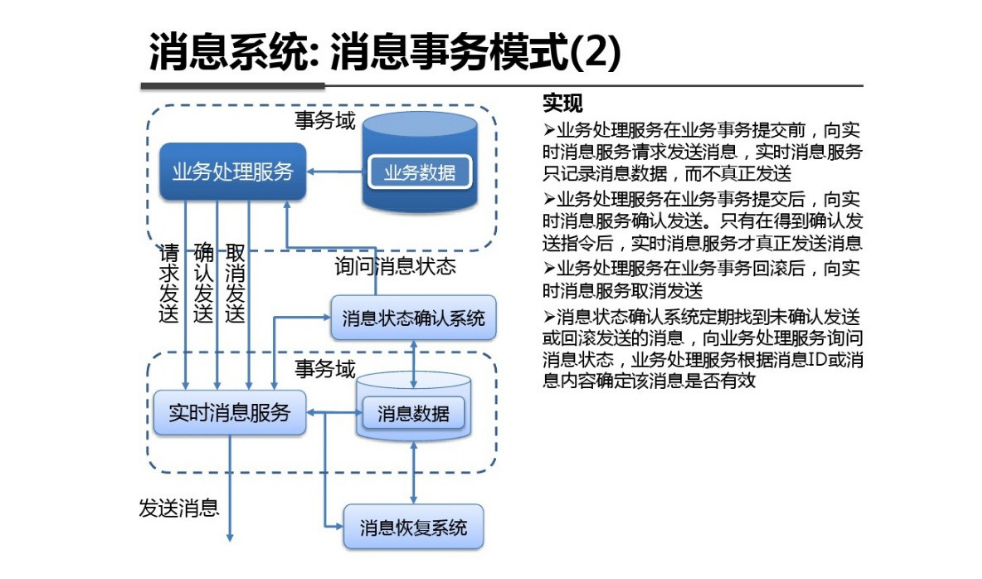
最大事务补偿型：



我们的实现方式：

1. 通过本地事务来保存信息，log消息。
2. 通过confirm来更新状态
3. 通过页面的方式来实现消息重发和重试

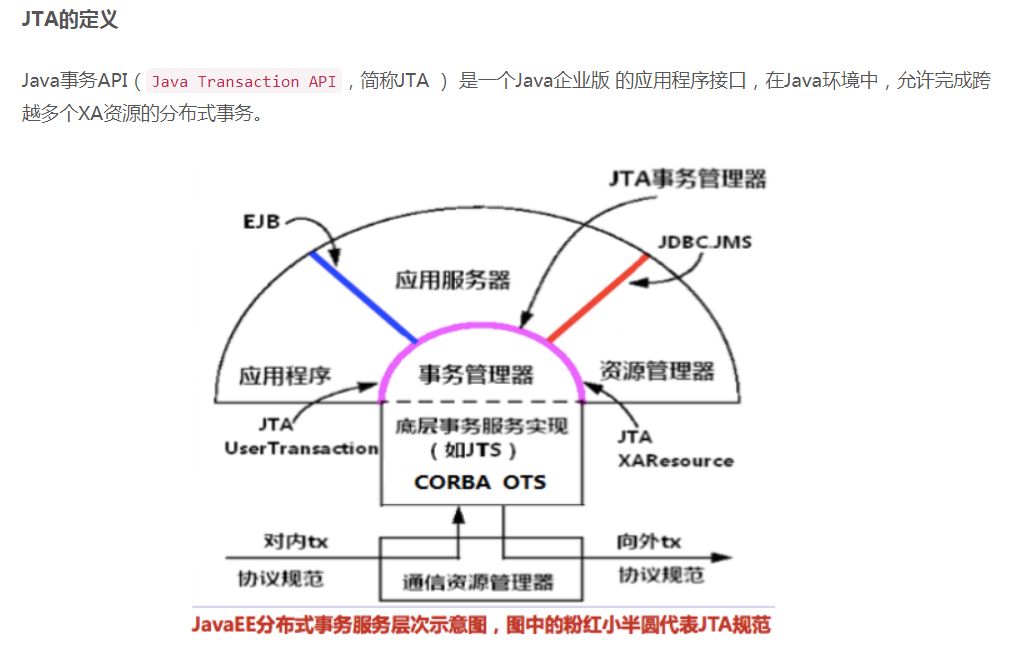
结合消息中间件的cache 机制：



最后一步，消息确认系统，要通过消息数据事务域，和事务域两者的数据比较来实现消息ID的对比找到没有正确处理的消息

JTA 事务：

<http://www.hollischuang.com/archives/1658>



JTA和它的同胞Java事务服务(JTS；Java TransactionService)，为J2EE平台提供了分布式事务服务。不过JTA只是提供了一个接口，并没有提供具体的实现，而是由j2ee服务器提供商 根据JTS规范提供的，常见的JTA实现有以下几种：

* 1.J2EE容器所提供的JTA实现(JBoss)
* 2.独立的JTA实现:如JOTM，Atomikos.这些实现可以应用在那些不使用J2EE应用服务器的环境里用以提供分布事事务保证。如Tomcat,Jetty以及普通的java应用。

不是使用了UserTransaction就能把普通的JDBC操作直接转成JTA操作，JTA对DataSource、Connection和Resource 都是有要求的，只有符合[XA规范](http://www.hollischuang.com/archives/681)，并且实现了XA规范的相关接口的类才能参与到JTA事务中来，关于XA规范，请看我的另外一篇文章中有相关介绍。这里，提一句，目前主流的数据库都支持XA规范。

要想使用用 JTA 事务，那么就需要有一个实现 javax.sql.XADataSource 、 javax.sql.XAConnection 和 javax.sql.XAResource 接口的 JDBC 驱动程序。一个实现了这些接口的驱动程序将可以参与 JTA 事务。一个 XADataSource 对象就是一个 XAConnection 对象的工厂。XAConnection 是参与 JTA 事务的 JDBC 连接。

要使用JTA事务，必须使用XADataSource来产生数据库连接，产生的连接为一个XA连接。

XA连接（javax.sql.XAConnection）和非XA（java.sql.Connection）连接的区别在于：XA可以参与JTA的事务，而且不支持自动提交。

看上面关于分布式事务的介绍是不是和2PC中的事务管理比较像？的却，2PC其实就是符合XA规范的事务管理器协调多个资源管理器的一种实现方式

**JTA的优缺点**

JTA的优点很明显，就是提供了分布式事务的解决方案，严格的ACID。但是，标准的JTA方式的事务管理在日常开发中并不常用，因为他有很多缺点:

* 实现复杂
  + 通常情况下，JTA UserTransaction需要从JNDI获取。这意味着，如果我们使用JTA，就需要同时使用JTA和JNDI。
* JTA本身就是个笨重的API
* 通常JTA只能在应用服务器环境下使用，因此使用JTA会限制代码的复用性。

2. 分布式事务处理 : Java 事务编程接口（JTA：Java Transaction API）和 Java 事务服务 (JTS；Java Transaction Service) 为 J2EE 平台提供了分布式事务服务。分布式事务（Distributed Transaction）包括事务管理器（Transaction Manager）和一个或多个支持 XA 协议的资源管理器 ( Resource Manager )。我们可以将资源管理器看做任意类型的持久化数据存储；事务管理器承担着所有事务参与单元的协调与控制。JTA 事务有效的屏蔽了底层事务资源，使应用可以以透明的方式参入到事务处理中；但是与本地事务相比，XA 协议的系统开销大，在系统开发过程中应慎重考虑是否确实需要分布式事务。若确实需要分布式事务以协调多个事务资源，则应实现和配置所支持 XA 协议的事务资源，如 JMS、JDBC 数据库连接池等。使用 JTA 处理事务的示例如下（注意：connA 和 connB 是来自不同数据库的连接）

XAResource 与 Xid： XAResource 是 Distributed Transaction Processing: The XA Specification 标准的 Java 实现，它是对底层事务资源的抽象，定义了分布式事务处理过程中事务管理器和资源管理器之间的协议，各事务资源提供商（如 JDBC 驱动，JMS）将提供此接口的实现。使用此接口，开发人员可以通过自己的编程实现分布式事务处理，但这些通常都是由应用服务器实现的（服务器自带实现更加高效，稳定） 为了说明，我们将举例说明他的使用方式。  
在使用分布式事务之前，为了区分事务使之不发生混淆，必须实现一个 Xid 类用来标识事务，可以把 Xid 想象成事务的一个标志符，每次在新事务创建是都会为事务分配一个 Xid，Xid 包含三个元素：formatID、gtrid（全局事务标识符）和 bqual（分支修饰词标识符）。 formatID 通常是零，这意味着你将使用 OSI CCR（Open Systems Interconnection Commitment, Concurrency 和 Recovery 标准）来命名；如果你要使用另外一种格式，那么 formatID 应该大于零，-1 值意味着 Xid 为无效。

RabbitMQ支持本地事务，但是不支持JTA事务或XA事务。

RabbitTransactionManager：

\* This local strategy is an alternative to executing Rabbit operations within, and synchronized with, external

\* transactions. This strategy is <i>not</i> able to provide XA transactions, for example in order to share transactions

\* between messaging and database access.

X/Open XA接口是双向的系统接口，在事务管理器（Transaction Manager）以及一个或多个资源管理器（Resource Manager）之间形成通信桥梁。



对于spring rabbit transaction的文档中，显示融合了transaction的说明

<https://docs.spring.io/spring-amqp/reference/htmlsingle/>

**3.1.14 Transactions**

StackOverflow：

<https://stackoverflow.com/questions/35148506/spring-amqp-transaction-semantics>

这个人的问题很好的讲解了，rabbit 提供的事务的实现方式，和局限。

1. 就是spring ampq的是通过监听Spring的transaction，通过事务钩子是把自己的事务来处理了。

但是如果在两者事务提交的间隙中，broken出现问题或者网络问题了，这个时候没有办法实现一致性了。也就是说这种方式，不能保证小概率的事情。

|  |  |
| --- | --- |
| A: | Assuming the transactionManager is your database tm, since your listener is @Transactional, there's not a lot of difference for these scenarios.  In the first case, the transaction is started by the container before the listener is invoked (actually before the message is retrieved from an internal queue so a transaction will start even if there's no message).  In the second case, the transaction is started by the transaction interceptor when we invoke the listener.  Consider the case where the listener is not transactional, but some downstream component is; let's say the listener invokes that component successfully, then does some more work before throwing an exception. In that case, the DB commit would be successful and the message rejected. This might not be the desired behavior, especially if messages are requeued. In cases like this, it's generally better ***to synchronize the rabbit transaction with the database transaction by*** injecting the database tm.  In your case, ***there is little chance of a failure*** between the db commit and the rabbit ack so this really doesn't apply in your case and you don't need a tm in the container. |

Q:Regarding your last sentence in the answer. What can actually go wrong between db commit and the rabbit ack?

A: Since there's only framework code there will be no code failures in that time. However, if, say, the connection to rabbitmq is lost, before the ack is sent, your rabbit template operation will roll back and the message will be requeued, but your DB transaction is committed. This applies in the other scenario too, so you should always have code to handle duplicate deliveries