事务，是操作数据库中的数据的逻辑单元。在处理一个业务过程中，事务保证多个数据修改的操作，要么都修改成功，要么都失败。同时，几个事务之间又相互独立，不会相互影响。

在这篇文章中，我们会先带大家理解事务，以及Spring中的事务，通过Spring的事务抽象引出JTA事务，以及JTA的分布式事务。理解了事务以后，再介绍分布式系统、以及分布式系统的原则，和分布式系统中实现事务的原则。

## **事务的ACID原则**

在介绍分布式事务之前，先来来回顾一下事务的ACID原则：

* 原子性（A）：原子性是指一个事务的所有操作，要么都做完，要么都不做。
* 一致性（C）：一致性是指一个事务的执行，不管外部环境如何，不管怎么执行，结果应该都是一致的。
* 隔离性（I）：隔离性是指几个事务在同时执行的时候，相互之间不会受影响。
* 持久性（D）：持久性就是事务完成以后，数据就被保存。

那么，在分布式系统中，这个原则是否能够保证呢？答案是不能，Not even close! 以原子性为例，在有多个系统的分布式系统中，一个分布式事务是在不同的系统内部执行的，我们没有办法保证它们能够同时完成，或者都不做。至于分布式事务的原则，我们过一会再说，我们先把事务搞清楚。

## **Spring中使用事务**

Spring是一个伟大的框架，从一开始只是一个容器框架，到现在已经发展成为了一个包含企业开发中的方方面面的很多框架的总称。它不但从复杂度上，发展出了用于各个方面的子框架。它还从易用性出发，推出了像Spring-Boot这样的框架，使得搭建环境变得异常的简单。

很早之前Spring就已经有了一套自己的事务规范。（在org.springframework.transaction包中），而且用起来也非常的简单：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | @Service  public Class OrderService {  @Transactional  public TicketOrder buyTicket(OrderDTO orderDTO) {  TicketOrder tkOrder = new TicketOrder();  jdbcTemplate.execute(createOrderSQL);  return tkOrder;  }  } |

我们只需要在方法上加一个Transactional标签，那个这个方法就会在一个事务里面执行。这是用代理模式实现的。Spring容器在初始化这个service实例的时候，实际上是创建一个代理类，然后在调用这个方法的时候，包装一个事务的处理。上面的方式使用代理模式展开，大致如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public Class OrderServiceProxy {  // 通过代理实现的伪代码，在原先的代码外再包一层事务的创建、commit、rollback  public Order buyTicket(OrderDTO orderDTO) {  // get transaction from entityManager  theTransaction.begin();  try {  orderServiceImpl.buyTicket(orderDTO)  theTransaction.commit();  } catch(Exception e) {  theTransaction.rollback();  throw e;  }  }  } |

从这个流程可以看出，在更改数据的时候jdbcTemplate.save(order)，事务并没有提交，用户查看最新的数据的时候，也看不到这条数据（隔离性），只有commit以后，所有的数据修改才会同时起效（原子性）。如果期间发生任何错误，事务就会回退rollback，所有的数据修改又回到未修改状态。

可能很多Java开发人员对事务的了解，就到这一步，没有再往下了解过。我们现在就来了解一下Spring的事务管理，了解了Spring的事务，才能知道如何在分布式系统中使用Spring的事务管理实现分布式事务。

### **Spring的事务抽象**

由于历史原因，从很早之前，Spring就已经有了一套自己的事务规范（在org.springframework.transaction包中）。但是针对JTA规范，Spring也做了很多工作，使得我们在实现事务的时候不需要关心具体用的是哪一个。但是，这也有一些问题：  
第一个问题就是，很多做Java开发的人，都不知道JTA和Spring Transaction的区别。  
第二，这两种规范在某些地方还是有些区别，使用不当也会出现问题。

我们在上面介绍本地事务的时候说，使用Spring框架的标签@Transactional，来方便的实现事务，但是需要说明的是，这个标签类，在Spring的事务以及JavaEE事务规范中都有定义，分别是：

* org.springframework.transaction.annotation.Transactional
* javax.transaction.Transactional

我们在使用的时候就需要注意，如果你用Spring boot，那么大部分情况下这两种标签都能使用。spring boot提供了很多自动配置，能够根据你是否包含了JTA的依赖，来判断是否要使用JTA的事务。如果没有，即使你用的javax.transaction.Transactional标签，也会使用spring的事务机制来处理。

但是，如果你的spring应用比较复杂，有很多自定义的配置，就需要注意这两种标签。最好是根据需要明确的使用一种。

Spring之所以能够实现对两种事务的支持，是因为在spring的Transaction规范中，定义了一个统一的PlatformTransactionManager事务管理器。即使你没有使用某个JPA框架，而是直接用JDBTemplate，Spring也能够使用默认的DataSourceTransactionManager来使用JDBC的事务来实现事务。而且也可以直接通过标签org.springframework.transaction.annotation.Transactional来实现事务。也就是说，你不需要任何JPA的实现框架，只是使用Spring-Transaction就能实现数据库的事务。

Spring的PlatformTransactionManager，也有JTA的实现JtaTransactionManager。也就是说，你可以使用Spring的事务规范，却使用JTA的实现，而且也几乎不需要任何配置，只要在具体的运行环境中包含包含JTA的实现可以。比如你用JBoss的应用服务器，系统就会使用Jboss的JTA实现；如果你的class path里面有Atomikos的库，系统就会使用Atomikos的JTA实现。如果你使用spring-boot，你只需要在你的依赖里面、或运行环境里面，提供你所需要的JTA实现，它就会自动使用。

除了数据库，spring的事务还支持JMS的事务，也就是在通过JMS使用某个消息中间件时，也能用spring的事务来实现读写消息的事务。

再总结一下Spring的事务抽象，它定义了抽象的事务管理，可以管理任何支持事务操作(也就是commit和rollback)的资源，如：

1. JDBC Connection: JDBC的连接支持commit()和rollback()操作。。
2. JPA的Entity: 从JPA的EntityManager中获得EntityTransaction，通过它实现。
3. JMS的session：jms的session提供commit()和rollback()操作。
4. JTA的事务：JTA的事务当然提供了事务的操作。

### **本地事务和全局事务**

如果查看Spring事务相关的文档，经常会看到’local transactions’和’external transactions’，本地事务和全局事务(或叫外部事务)。上面我们说了，对任何资源，只要它提供了事务的操作，我们就能使用spring的事务管理来提供事务。由于spring提供了一个事务管理的抽象接口，而事务的控制，可以是在spring容器来控制，也可以由外部的事务管理模块来控制，这就是本地事务和全局事务的区别。

本地事务就是指的是由Spring容器创建和维护的事务。例如在使用JDBC事务操作数据库的时候，spring容器会在需要的时候创建事务的上下文，开启一个JDBC的事务，然后调用业务方法，执行完成后，调用commit方法；然后在出错的时候调用资源的rollback方法。还有事务的传播、隔离等也都是由Spring容器来提供。本地事务只能针对一个资源实现完全的事务控制。如果要在一个本地事务中操作两个资源（例如两个数据库），实际上先后在两个数据库的Connection上调用commit()方法去提交。

而外部事务，就是spring只负责通过事务的接口来开始事务、提交事务、回滚事务，而具体的操作还是得有外部提供的事务管理的模块或组件来执行和维护。例如我们使用JBoss来运行我们的web应用，然后在JBoss上配置了JTA的事务。那么事务的具体管理和维护就是由JBoss提供的事务管理模块来进行。

本地事务和外部事务的一个主要区别就是，是否能对多个资源实现事务控制。我们来通过一个例子来说明它们的区别：使用JDBCTemplate和JMSTemplate对一个数据库和一个MQ进行操作。使用Spring的代码大致如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | @Service  public Class OrderService {  @Transactional  public TicketOrder buyTicket(OrderDTO orderDTO) {  orderRepository.save(order);  jmsTemplate.convertAndSend("order:need\_to\_pay", order);  return tkOrder;  }  } |

也就是在@Transactional标记的方法里，通过jdbcTemplate操作数据库，使用JMSTemplate操作MQ。这个方法虽然是在一个事务里，但是，如果我们使用本地事务，那么这两个资源（数据库和MQ）实际上是在各自的事务里面分别操作。把这段代码展开成它实际的样子，大致如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | jmsTransaction.begin(); // get transactions from jms session  dbTransaction.begin(); // get transactions from JDBC connection  try {  orderRepository.save(order);  jmsTemplate.convertAndSend("order:need\_to\_pay", dto);  dbTransaction.commit();  jmsTransaction.commit();  } catch(Exception e) {  dbTransaction.rollback();  jmsTransaction.rollback();  } |

这样，如果上述代码在jmsTransaction.commit();的时候出错，这时候数据库的事务已经提交，就无法回滚。如果这时候这个方法被重新执行，数据库的操作就会被重复执行。

如果我们使用外部事务，那么这里就不会针对两个资源出现两个事务，而是只有一个事务，来统一管理多个资源。如果在多个资源上的事务出错了，外部的事务也能够保证回滚，这是通过事务的两阶段提交(2PC)来实现。使用JTA实现的事务正是这种外部事务。

由于JTA使用两阶段提交来实现多个资源之间的事务，这就会带来很大的性能问题。因为它要同步多个资源的事务，对每个资源使用两阶段提交，这就使得这个事务所花的时间比本地事务多很多。而且在这个时间段内，由于事务的隔离性，可能会造成长时间的资源占用，使得其它的事务无法同步访问该资源上的一些数据。

## **JTA事务**

在上面已经多次提到JTA事务，那么JTA到底是什么呢？介绍JTA之前，先看看XA。

### **XA**

XA是由X/Open组织提出的分布式事务的架构（或者叫协议）。XA架构主要定义了（全局）事务管理器（Transaction Manager）和（局部）资源管理器（Resource Manager）之间的接口。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。也就是说，在基于XA的一个事务中，我们可以针对多个资源进行事务管理，例如一个系统访问多个数据库，或即访问数据库、又访问像消息中间件这样的资源。这样我们就能够实现在多个数据库和消息中间件直接实现全部提交、或全部取消的事务。XA规范不是java的规范，而是一种通用的规范，

目前各种数据库、以及很多消息中间件都支持XA规范。  
JTA是满足XA规范的、用于Java开发的规范。所以，当我们说，使用JTA实现分布式事务的时候，其实就是说，使用JTA规范，实现系统内多个数据库、消息中间件等资源的事务。

### **什么是JTA**

JTA(Java Transaction API)，是J2EE的编程接口规范，它是XA协议的JAVA实现。它主要定义了：

* 一个事务管理器的接口javax.transaction.TransactionManager，定义了有关事务的开始、提交、撤回等操作。
* 一个满足XA规范的资源定义接口javax.transaction.xa.XAResource，一种资源如果要支持JTA事务，就需要让它的资源实现该XAResource接口，并实现该接口定义的两阶段提交相关的接口。

如果我们有一个应用，它使用JTA接口实现事务，应用在运行的时候，就需要一个实现JTA的容器，一般情况下，这是一个J2EE容器，像JBoss，Websphere等应用服务器。但是，也有一些独立的框架实现了JTA，例如[Atomikos](https://www.atomikos.com/" \t "http://codin.im/2017/05/14/rest-micro-services-distributed-trasaction-1-jta/_blank), [bitronix](https://github.com/bitronix/btm" \t "http://codin.im/2017/05/14/rest-micro-services-distributed-trasaction-1-jta/_blank)都提供了jar包方式的JTA实现框架。这样我们就能够在Tomcat或者Jetty之类的服务器上运行使用JTA实现事务的应用系统。

在上面的本地事务和外部事务的区别中说到，JTA事务是外部事务，可以用来实现对多个资源的事务性。它正是通过每个资源实现的XAResource来进行两阶段提交的控制。感兴趣的同学可以看看这个接口的方法，除了commit, rollback等方法以外，还有end(), forget(), isSameRM(), prepare()等等。光从这些接口就能够想象JTA在实现两阶段事务的复杂性。

### **JPA**

在上面介绍Spring的事务抽象的时候，说过Spring事务支持JPA的事务，这里再针对这个做一下说明。  
可能很多做Java开发的，都没弄清楚JTA和JPA的区别，即使我一开始也以为JTA和JPA就是一回事，其实不然。JPA是Java Persistence API，也就是Java持久化编程接口。它定义了Java对象和它的持久化之间的联系,也就是Java的Object和Relation之间的Mapping，也就是通常说的ORM。而这个对象的持久化，不仅限于数据库，也可能是NoSQL，可能是文件，也可能是其它能够序列化后保存的地方。JPA使用@Entity标记一个Java对象，并将这个Java对象和数据库的某一个表关联。通过ID将一个实例映射到表中的一条记录。  
像Hibernate就是一个实现JPA的框架。  
使用JPA实现事务，用的是EntityManager来获取一个事务EntityTransaction，而在JTA中，用的是TransactionManager。

### **JTA分布式事务**

使用JTA事务，可以实现对多个资源实现事务，这也是一说到分布式事务，就会说JTA的原因。

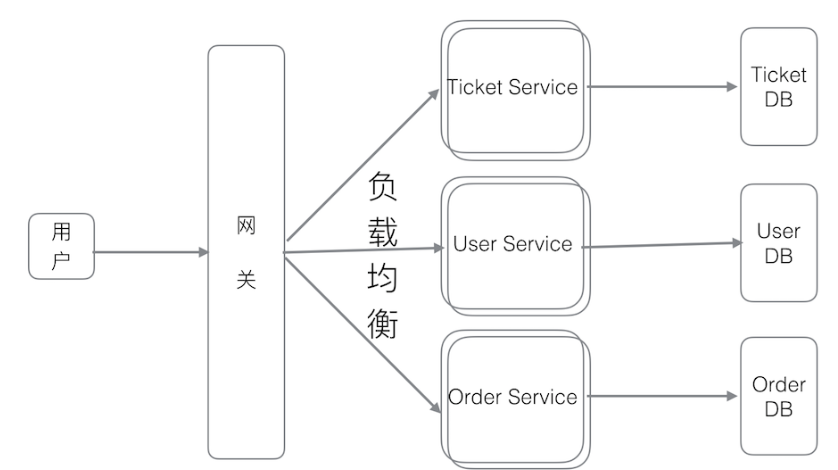
如果你的分布式系统只是把数据库按照功能地区等进行分区分片的划分，再使用MQ等资源，那你就完全可以通过使用JTA来实现不同资源的分布式事务。

但是，现在流行的微服务框架，往往是部署多个服务，一个事务可能需要调用多个服务，调用多个数据库、MQ，对于这种微服务架构的分布式事务，又需要使用其它的方式来实现。

## **分布式系统**

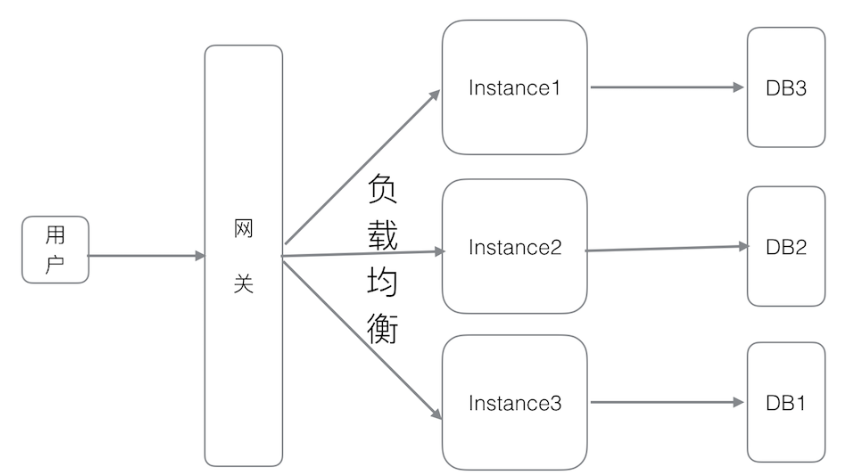
分布式系统从一开始到现在，有多种形式，从应用的个数和使用的数据库角度来说，简单列举了如下几种：

1. 一个服务同时访问多个数据库，有多个数据源。
2. 对一个应用部署多个实例，每个实例都无差别的为用户提供服务，同一个应用的多个实例可能访问同一个数据库。
3. 同一个应用部署多个实例，各个实例通过对数据库的分片分区，来访问不同的数据库。
4. 按照功能划分，不同的功能由不同的服务提供，相互之前通过网关、服务中心等方式通信。

这几年，微服务的概念越来越火，一般来说，用微服务架构实现的分布式系统，有两种方式（这里简化了很多东西，像缓存、监控、消息中间件、日志等等支持系统，只是仅仅考虑一般的业务系统）：  
按照功能划分，每个应用提供某种功能，而每个应用又部署多个实例来实现高可用。  


**[多服务多实例](http://codin.im/2017/05/14/rest-micro-services-distributed-trasaction-1-jta/services-1.png" \o "多服务多实例)**

这种实现的好处是可以每个功能一个应用，那个每个应用模块都比较简单；但是，这就会有很多服务间调用，有时候为了完成一个业务请求，要在好几个服务之间调用好几次。这就需要在拆分模块、设计服务间调用的接口、设计业务的流程的时候都需要综合考虑，尽量减少服务间调用。

还有一种是用同样的应用部署多个实例，各个服务通过分区分片等方式，使用各自的数据库或其它数据源。  


**[单一服务多实例](http://codin.im/2017/05/14/rest-micro-services-distributed-trasaction-1-jta/services-2.png" \o "单一服务多实例)**

这种方式的好处就是，所有的功能都在一个应用里，一个应用部署任意多个，只需要通过合理的数据库的分区分片，让不同的节点访问不同的数据库。但是，一旦你的数据越来越复杂，数据库的分区分片会非常复杂。有时候，也可以把数据库按功能分开，一个节点访问多个数据库。

## **分布式系统的原则**

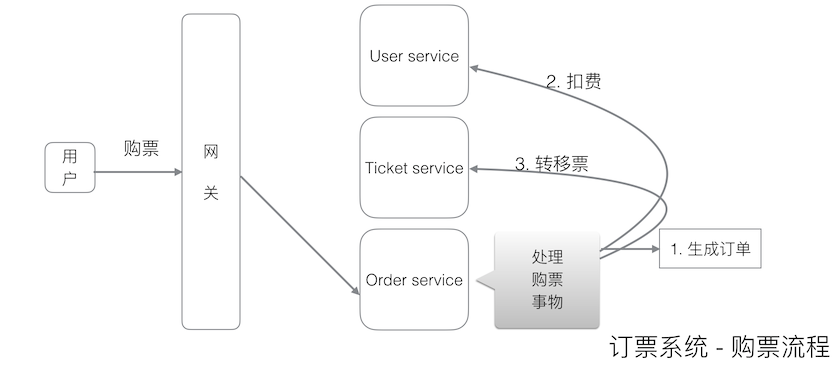
在介绍如何实现分布式系统的事务之前，我们先看看分布式系统的原则。

对于分布式系统来说，很难有一个类似ACID这样的标准，和满足这个标准的开发规范，及其实现的框架，来为我们方便的实现分布式系统的事务。要实现分布式系统的事务，我们往往需要根据实际需要，在可用性（包括性能、系统吞吐量等）、事务性（类似本地事务的ACID）、可操作性（开发和维护的难易程度）之间做出权衡。  
那么，分布式事务的原则是什么呢？我们怎么能确定一个分布式事务的实现，满足了它的事务性的要求呢？首先，我们来看看分布式系统的一个原则，或者叫定理，CAP定理。

### **CAP定理**

CAP定理，包括以下几个方面：

* 一致性（C）：在分布式系统中的所有数据备份，在同一时刻是否同样的值。比如说，在购票流程处理的过程中，如果用户看到自己的余额以及被扣了，那么它应该也能看到票夹里的票、以及支付完成的订单。
* 可用性（A）：可用性是指系统提供的服务必须一直处于可用的状态，包括每个请求都应该在一定的时间内返回结果。它包括时间和结果两个条件，也就是说，即使出现错误、超时等问题，也应该是一定的时间内给用户反馈。
* 分区容错性（P）：如果集群系统中有一部分服务发生故障，仍然能够保证对外提供满足一致性和可用性的服务。也就是说，集群中一部分节点故障后，集群整体还是能响应客户端的读写请求。

为了便于我们理解，在介绍这个CAP定理的时候，我们结合一个业务实例来看看这个CAP定理是什么意思。这个实例是一个简单的订票系统的购票流程，大致如下：  


这是一个微服务架构的分布式系统，有一个网关统一接受用户请求，然后将请求转发到相应的服务上。  
总共有3个服务：Order，User，Ticket分别用于处理交易、用户、票相关的业务。  
每个服务都使用自己的数据库。

一个购票流程大致如下：

1. 用户触发一个购票流程
2. Order服务处理这个请求，先在当前服务创建订单等信息，将数据保存在order数据库
3. 然后调用User服务，进行扣费等操作
4. 再调用Ticket服务，进行票的转移等操作
5. 整个流程完成后，用户应该能看到自己的订单，自己余额的减少，以及票夹里的票。

再来看看CAP定理：

* 一致性（C）：一致性就是说，在购票流程处理的过程中，如果用户看到自己的余额以及被扣了，那么它应该也能看到票夹里的票、以及支付完成的订单。
* 可用性（A）：可用性就是在购票过程中，要及时给用户反馈，即使有些步骤时间比较长，可以通过异步的方式处理，给用户一个正在处理的结果。
* 分区容错性（P）：假设某一个Ticket服务的节点出了故障，也应该有其它的节点能够提供Ticket服务；如果所有的Ticket服务都不能用了，也不应该影响用户访问其它的部分，而只是不能下单买票。

由于分布式系统形式的多样性和复杂性，如果想完全满足上述的原则设计一个分布式系统，几乎是不可能的。首先，分布式服系统就是要把系统的各个部分部署到不同的服务器上，那我们就必须要通过分区容错来避免由于网络、机器故障等原因造成的问题。所以分区容错性是必不可少的，否则可用性都无法保证。  
对于可用性来说，如果我们要严格保证可用性，即使是在分区容错性得到保障的前提下，所有的服务都是可用的，有时候，我们也需要通过异步的方式来处理一些业务，这就会造成数据的不一致。如已经从用户账户上扣费，但是票还没有转移完成等。  
再来看一致性，是否有办法能够实现呢？那我们就需要先来看看几种一致性：

* 强一致：当更新操作完成之后，任何多个后续进程或者线程的访问都会返回最新的更新过的值。这就像本地事务的原子性（A）和隔离性（I）的统一。在分布式系统中，如果一个业务处理需要多个系统都更新数据，那就要求多个系统的更新同时完成。但是，因为它们的不同的系统，‘同时完成‘需要服务间的协作、同步才能完成，在完成之前，用户不能看到更新后的数据，也不能看到更新前的（因为要强一致），所以用户只能等待。这就违背了可用性；同时，为了保证强一致性，需要做很多额外的工作，又大大增加了出错的可能性。所以在分布式系统中，强一致性一般都无法实现。
* 弱一致性：系统并不保证续进程或者线程的访问都会返回最新的更新过的值。系统在数据写入成功之后，不承诺立即可以读到最新写入的值，也不会具体的承诺多久之后可以读到。例如上面的例子，我们有几个子系统，当订单系统生成订单，然后交由用户系统处理的时候，这时候用户就能够看到自己的新的订单。当票务系统处理票的转移的时候，用户能看到已经扣费，但是又看不到票夹里的票。虽然这个时间可能很短，但是也是存在的。
* 最终一致性：弱一致性的特定形式。系统保证在没有后续更新的前提下，系统最终返回上一次更新操作的值。例如上面的例子中， 我们用异步方式处理用户的购票，先生成订单、扣费，异步处理票，返回给用户结果，这时用户看到的订单状态是正在处理，只有整个流程处理完了，用户才能看到订单结束，并且能看到买到的票。除了异步操作造成的一致性问题以外，还有在某一个节点发生故障的情况下，通过重试、取消等机制，或者人工参与，使得系统的数据也能最终达到一个一致的状态。

在一般的分布式系统的设计中，我们大都以最终一致性为目标，来设计我们的分布式事务。这既能保证系统的可用性和容错性，也能在绝大多数情况下保证数据的弱一致性，并且在少数出错或网络高延迟的情况下，也能保证数据的最终一致性。

### **幂等性原则**

上面说了分布式事务的原则，以最终一致性为目标，那为了实现这个目标，我就需要很多异常情况的处理，包括数据库失败、业务代码失败、网络错误等。举例来说，在一个接口调用，如果发生超时，我就需要重试。但是也有可能对方的服务已经处理完这个请求，只是在完成返回结果的时候，网络传输的问题导致超时。那么服务调用端再重试，实际上就是发了两次请求。所以，我就需要对于分布式服务的事务处理，对于同样的消息，只会处理一次。

分布式系统的幂等性，就是对于一个处理接口，如果它会对系统造成副作用，也就是修改数据，那就需要保证对于同样的请求，不管请求了多少次，结果都是一致的。

那么，如何保证这个幂等性呢？一种比较通用的方法就是，对每一个请求，生成一个token，而且需要唯一，然后将这个token放在请求的参数里面。服务在处理这个请求之前，先拿到token，检查这个token是否已经处理过，只有没有处理过的才去处理。这个token可以保存在数据库、redis、甚至内存等地方。由于它只是用来记录已经处理的请求的token，所以大可不必保存在内存中。由于在分布式系统中，一个服务会部署多个，一个请求失败后重新发送，有可能是被发送到另一台机器上。所以这个token应该是服务范围共享的，我们需要在同一个服务的多个部署都能共享访问的地方，来保存已经处理过的token。

所以，使用redis来保存token是一个不错的选择。

### **分布式事务的实现**

分布式事务的实现，也有很多种的实现方式，一般称作模式。我们在一个分布式系统的一个业务方法里，往往需要调用外部的数据库、MQ，也有可能调用其它服务。如果你把其它的服务也看作一种资源，那么一个业务方法实际上就是操作了好几个资源。而这，正是XA所做的事情。但是，不是所有的资源都支持XA，像我们的服务间调用，一般就只有一个处理方法，不会提供什么commit()、rollback()之类的方法，更别说两阶段提交需要的其它方式。

但是，我们却能用事务的思想为我们实现分布式事务提供一些启发。从本地事务的处理过程，我们可以看出，它是通过：1.尝试修改 2.提交(完成) 3.取消(出错)的方式来实现的 。根据这个思想，我们可以想到分布式事务的几种实现方式：

* 用先尝试修改、再确认或取消的方式，也就是TCC模式(Try尝试修改 - Commit提交 - Cancel取消）
* 还有一种稍微简单的就是Fallback模式，就是先修改，出错了以后再进行相应的处理，也就是调用Fallback。
* 除了这两种以外，还有一种常用的就是使用消息中间件，各个分布式系统在处理同一个事务的时候，通过消息驱动来执行各自的任务。

有关TCC模式的详细内容，请参考作者的[原文](https://www.atomikos.com/Blog/TCCForTransactionManagementAcrossMicroservices" \t "http://codin.im/2017/05/14/rest-micro-services-distributed-trasaction-1-jta/_blank)。TCC模式的事务实现，我们会在另一篇文章中再介绍。

有关在spring中实现分布式事务，有一篇[文章](http://www.javaworld.com/article/2077963/open-source-tools/distributed-transactions-in-spring--with-and-without-xa.html" \t "http://codin.im/2017/05/14/rest-micro-services-distributed-trasaction-1-jta/_blank)。这篇文章介绍了使用XA和不使用XA实现分布式事务的几种方式。但是都是说的同时使用2种或以上的资源（如数据库和MQ等支持XA的数据源）的情况。虽然不能适用于微服务架构的服务间调用的情况，但是也能有一些借鉴意义。  
在上面的文章中介绍的其中一种方式是：最大努力一阶段提交。还是用之前的例子说明，也就是在一个事务方法中操作DB和MQ：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | @Service  public Class OrderService {  @Transactional  public TicketOrder buyTicket(OrderDTO orderDTO) {  jdbcTemplate.doQuery(sth);  jmsTemplate.send(sth);  return tkOrder;  }  } |

如果这里不用JTA事务，而是使用Spring的本地事务，那么这个方法内的操作执行完以后，spring事务管理器会先后提交DB Connection的事务和JMS Session的事务。在这种方式下，绝大部分情况下都不会有问题。但是有可能出现的一种错误就是，在提交完一个事务后，提交另一个事务的时候出错了。在以最终一致性为目标的分布式事务中，往往就允许这种情况的出现，但是需要采用另一些措施来补救。

至此，我们介绍了Spring事物、JTA事物，还有跨多个资源的事物，也介绍了一下分布式系统和分布式事务，特别是分布式事务的强一致性原则。在之后的几篇文章中，将继续介绍实现分布式事务的几种具体的方法。