乔二爷 Lv2
2019年04月20日 阅读 994

关注

面试必备的分布式事务方案

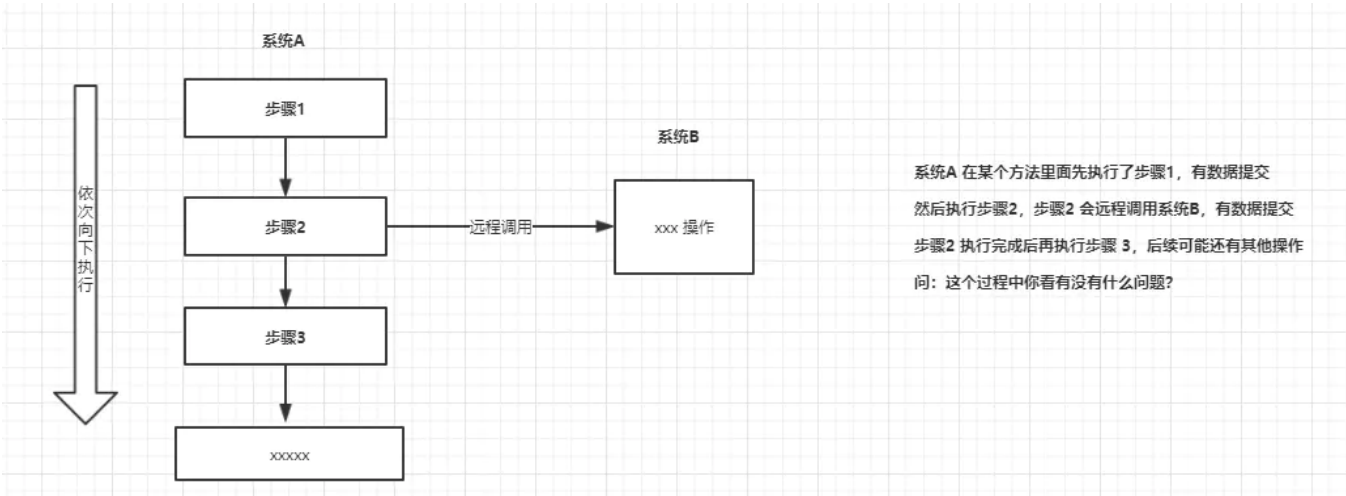
背景

四月初，去面试了本市的一家之前在做办公室无人货架的公司，虽然他们现在在面临着转型，但是对于我这种想从传统企业往互联网行业走的孩子来说，还是比较有吸引力的。

在面试过程中就提到了分布式事务问题。我又一次在没有好好整理的问题上吃了亏，记录一下，**还是长记性**！！！

先看面试过程

面试官先是在纸上先画了这样一张图：



让我看这张图按照上面的流程走，有没有什么问题？面试官并没有直接说出来这里面会有分布式事务的问题，而是让我来告诉他，**这就是面试套路呀**。

我回答了这中间可能存在分布式事务的问题，当步骤 2 在调用 B 系统时，可能存在B 系统处理完成后，在响应的时候超时，导致 A 系统误认为 B 处理失败了，从而导致A 系统回滚，跟 B 系统存在数据不一致的情况。

接着，他继续问：“那你有什么好的解决方式吗？”

此时我脑子里面只有两阶段提交的大概流程图的印象，然后巴卡巴拉的跟他说了一番，什么中间来个协调者呀，先预提交什么的，如果有失败，就 **rollback**，如果 ok，再真正的提交事务，就是网上这些大神们说的这些理论。

然后面试官就继续问：**那A 在调用 B 的这条线断了，你们代码具体是怎么处理的呢？ 怎么来做到 rollback 的呢？ 说说你代码怎么写的。**

此时，我懵了。

最后结果，大家肯定也能猜到，凉凉。

什么是事务

这里我们说的事务一般是指 **数据库事务**，简称 **事务**，是数据库管理系统执行过程中的一个逻辑单位，由一个有限的数据库操作序列构成。维基百科中这么说的。

用转账的例子来说，A 账户要给 B 账户转 100块，这中间至少包含了两个操作：

1. A 账户 减 100 块
2. B 账户 加 100 块

在支持事务的数据库管理系统来说，就是得确保上面两个操作（整个“事务”）都能完成，不能存在，A 的100块扣了，然后B 的账户又没加上去的情况。

数据库事务包含了四个特性，分别是：

- 原子性（Atomicity）：事务作为一个整体被执行，包含在其中的对数据库的操作要么全部被执行，要么都不执行。对于转账来说，A账户扣钱，B 账户加钱，要么同时成功，要么同时失败。
- 一致性（Consistency）：事务应确保数据库的状态从一个一致状态转变为另一个一致状态。一致状态的含义是数据库中的数据应满足完整性约束
- 隔离性（Isolation）：多个事务并发执行时，一个事务的执行不应影响其他事务的执行。其他账户在转账的时候，不能影响到上面的 A 跟 B 之前的交易。
- 持久性（Durability）：已被提交的事务对数据库的修改应该永久保存在数据库中。

什么是分布式事务

但是如果我们系统中出现垮库操作，比如一个操作中，我需要操作多个库，或者说这个操作会垮应用之前的调用，那么Spring 的事务管理机制就对这种场景没有办法了。

就像上面面试题中出现的问题一样，在系统 A 的步骤 2 在远程调用 B 的时候，由于网络超时，导致B没有正常响应，但是A 这边调用失败，进行了回滚，而 B 又提交了事务。此时就可能会导致数据不一致的情况，参生分布式事务的问题。

分布式事务的存在，就是解决数据不一致的情况。

为什么我们要保证一致性

CAP 理论

分布式系统中有这么一个广为流传的理论：**CAP 定理**

这个定理呀，起源于加州大学柏克莱分校（University of California, Berkeley）的计算机科学家埃里克·布鲁尔在 2000年的分布式计算原理研讨会（PODC）上提出的一个猜想。后来在2002年，麻省理工学院（MIT）的赛斯·吉尔伯特和南希·林奇发表了布鲁尔猜想的证明，使之成为一个定理。【摘自维基百科】

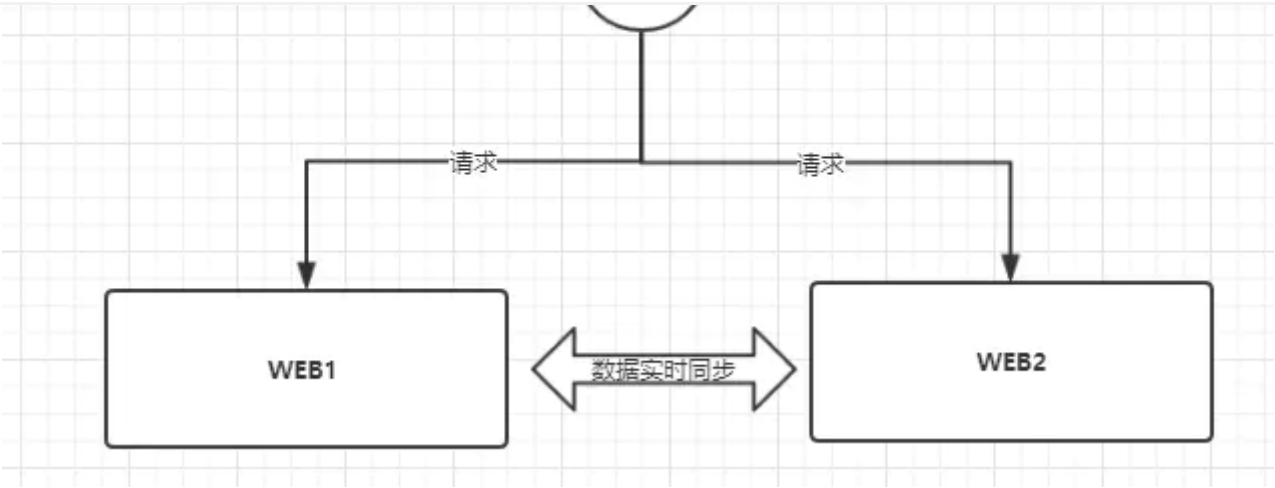
他说呀，对于一个分布式计算系统来说，不可能同时满足以下三点：

- 一致性（Consistency）
- 可用性（Availability）
- 分区容错性（Partition tolerance）

而一个分布式系统最多只能满足其中的两项。

那么，上面的三点分别是什么玩意儿？为什么又只能同时满足两项呢？

我们先看这样一个场景，现在我们系统部署了两份（两个节点，web1 和 web2 ）,同样的业务代码，但是维护的是自己这个节点生成的数据。但是用户访问进来，可能会访问到不同的节点。但是不管是访问web1 还是web2 ,在用户参数数据 过后，这个数据都必须得同步到另外的节点，让用户不管访问哪个节点，都是响应他需要的数据。如下图：



分区容错性

我们先说 **分区容错性**：也就是说呀，就算上面这两个节点之间发生了网络故障，无法发生同步的问题，但是用户访问进来，不管到哪个节点，这个节点都得单独提供服务，这一点对于互联网公司来说，是必须要满足的。

当 web1 和 web2 之间的网络发生故障，导致数据无法进行同步。用户在web1 上写了数据，马上又访问进来读取数据，请求到了 web2，但是此时 web2 是没有数据的。那么我们是 给用户返回 null？ 还是说给一些提示，说系统不可用，稍后重试呢?

都不妥吧，兄弟。

一致性

如果要保证可用性，那么有数据的节点返回数据，没数据的节点返回 null ,就会出现用户那里看到的是一会儿有数据，一会儿没有数据，此时就存在 **一致性** 的问题。

可用性

如果保证一致性，那么在用户访问的时候，不管 web1 还是web2 ，我们可能会返回一些提示信息，说系统不可用，稍后再试等等，保证每次都是一致的。明明我们有数据在，但是我们系统却响应的是提示信息，此时就是 **可用性** 的问题。

由于分区容错性（P）是必须保证的，那么我们分布式系统就更多是在一致性（CP） 和可用性（AP）上做平衡了，只能同时满足两个条件。

其实，大家想想，ZK 是不是就是严格实现了 CP ，而 Eureka 则是保证了 AP。

几种分布式事务解决方案

2PC

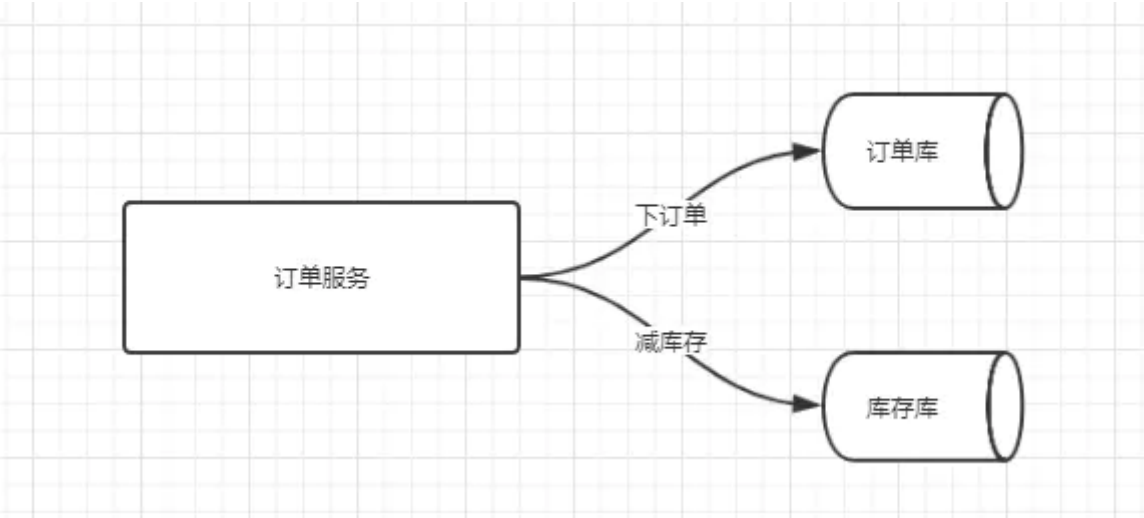
在说 2PC 之前，我们先了解一下 **XA规范** 是个什么东西?

XA规范 描述了全局的事务管理器与局部的资源管理器之间的接口。XA规范的目的是允许多个资源（如数据库，应用服务器，消息队列，等等）在同一事务中访问，这样可以使ACID属性跨越应用程序而保持有效。

XA 使用 **两阶段提交（2PC）** 来保证所有资源同时提交或回滚任何特定的事务。

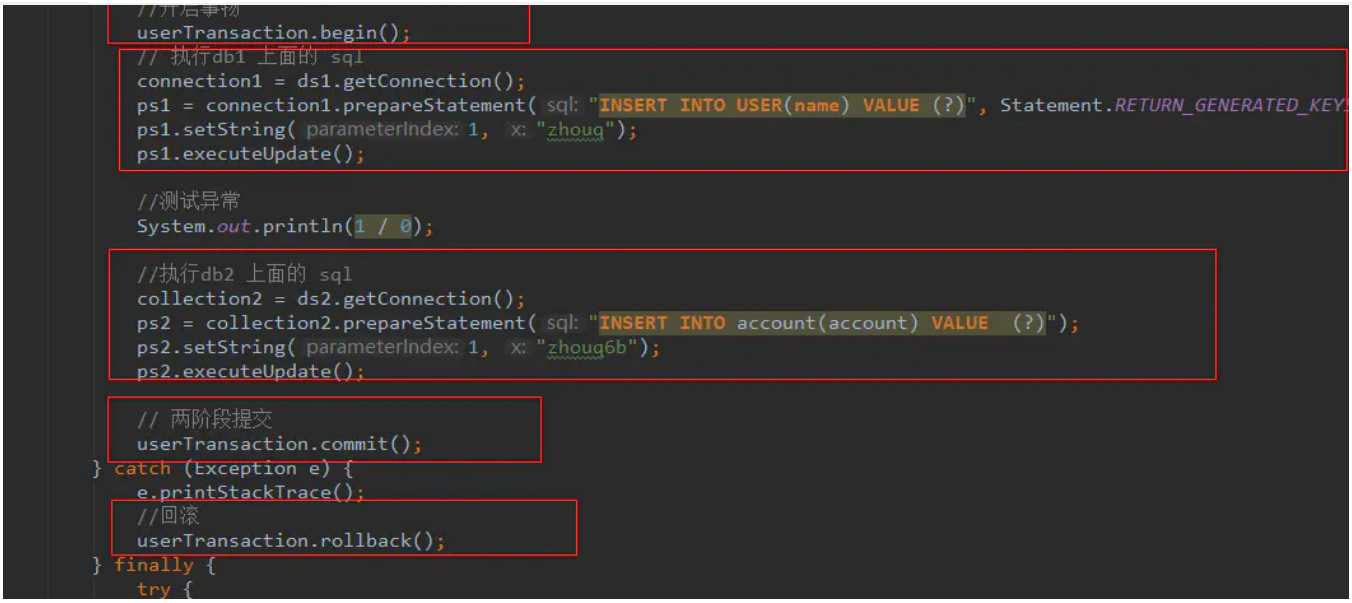
大家想一个场景，在做单应用的时候，有的同学连过两个库吧？在一个事务中会同时向两个系统插入数据。但是对于普通事务来讲，是管不了的。

看下图（只是举例这种操作的套路，不局限于下面的业务）：



一个服务里面要去操作两个库，如何保证事务成功呢。

这里我们介绍一个框架 **Atomikos** ，他就是实现了这种 XA 的套路。看代码：



```
    //开启事物
    userTransaction.begin();
    // 执行db1 上面的 sql
    connection1 = ds1.getConnection();
    ps1 = connection1.prepareStatement( sql: "INSERT INTO USER(name) VALUE (?)", Statement.RETURN_GENERATED_KEY
    ps1.setString( parameterIndex: 1,  x: "zhouq");
    ps1.executeUpdate();

    //测试异常
    System.out.println(1 / 0);

    //执行db2 上面的 sql
    collection2 = ds2.getConnection();
    ps2 = collection2.prepareStatement( sql: "INSERT INTO account(account) VALUE  (?)");
    ps2.setString( parameterIndex: 1,  x: "zhouq6b");
    ps2.executeUpdate();

    // 两阶段提交
    userTransaction.commit();
} catch (Exception e) {
    e.printStackTrace();
    //回滚
    userTransaction.rollback();
} finally {
    try {
```

具体代码移步 Github AtomikosJTATest: github.com/heyxyw/lear...

看到上面的图了哇，Atomikos 自己实现了一个事务管理器。我们获取的连接都从它哪里拿。

- 第一步先开启事务，然后进行预提交，db1 和 db2 都先进行预先执行，**注意：这里没有提交事务**。
- 第二步才是真正的提交事务，由 Atomikos 来发起提交的，如果出现异常则发起回滚操作。

这个过程是不是就有两个角色了，一个 事务管理器，一个资源管理器（我们这里是 数据库，也可以是其他的组件，消息队列什么的）。

整个执行过程是这样：

上图是正常情况，下图是一方出现故障的情况。

图片来自：XA 事务处理：www.infoq.cn/article/xa-... ，具体关于XA 的详细讲解，可以好好看看。整个2PC 的流程：

1. 协调者节点向所有参与者节点询问是否可以执行提交操作，并开始等待各参与者节点的响应。
2. 参与者节点执行询问发起为止的所有事务操作，并将Undo信息和Redo信息写入日志。
3. 各参与者节点响应协调者节点发起的询问。如果参与者节点的事务操作实际执行成功，则它返回一个"同意"消息；如果参与者节点的事务操作实际执行失败，则它返回一个"中止"消息。

第二阶段 (提交执行阶段)：

成功，当协调者节点从所有参与者节点获得的相应消息都为"同意"时：

1. 协调者节点向所有参与者节点发出"正式提交"的请求。
2. 参与者节点正式完成操作，并释放在整个事务期间内占用的资源。
3. 参与者节点向协调者节点发送"完成"消息。
4. 协调者节点收到所有参与者节点反馈的"完成"消息后，完成事务。

失败，如果任一参与者节点在第一阶段返回的响应消息为"终止"，或者 协调者节点在第一阶段的询问超时之前无法获取所有参与者节点的响应消息时：

1. 协调者节点向所有参与者节点发出"回滚操作"的请求。
2. 参与者节点利用之前写入的Undo信息执行回滚，并释放在整个事务期间内占用的资源。
3. 参与者节点向协调者节点发送"回滚完成"消息。
4. 协调者节点收到所有参与者节点反馈的"回滚完成"消息后，取消事务。

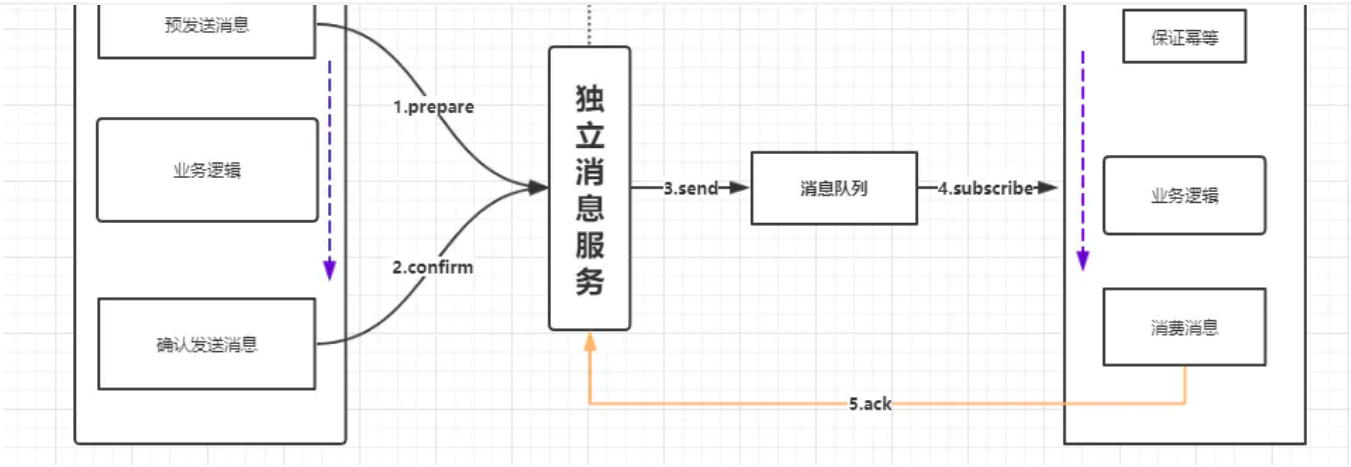
有时候，第二阶段也被称作完成阶段，因为无论结果怎样，协调者都必须在此阶段结束当前事务。

可靠消息最终一致性方案

基于普通的消息队列中间件

上面我们说了两阶段提交的方案，接下来我们讲讲怎么基于可靠消息最终一致性方案来解决分布式事务的问题。

这个方案，就有消息服务中间件角色参与进来了。我们先看一个大提的流程图：



我们以创建订单下单过程和 后面出库 的流程为例来讲述上面的图。

在下单逻辑里面（Producer 端），我们先生成一个订单的数据，比如订单号，数量等关键的信息，先包装成一条消息，并把消息的状态置为 **init** ,然后发送到 独立消息服务中，并且入库。

接下来继续处理 下单的其他本地的逻辑。

处理完成后，走到确认发送消息这一步，说明我的订单是能够下成功的。那么我们再向消息服务里面发送一条confirm 的消息，消息服务里面就可以把这个订单的消息状态修改为 **send** 并且，发送到消息队列里面。

接下来，消费者端去消费这条消息。处理自己这边的逻辑，处理完成以后，然后反馈消息处理结果到独立消息服务，独立消息服务把消息状态置为 **end** 状态 ,表示结束。但是得注意保证接口的幂等性，避免重复消费带来的问题。

这里面可能出现的问题，以及各个步骤怎么解决的：

1. 比如在 **prepare** 阶段就发生异常，那么这里订单这块都不会下成功。但是我们说，我们这里是基于可靠消息，得保证我们的消息服务是正常的。
2. 在 **comfirm** 出现异常，此时发送确认失败，但是我们的单已经下成功了。这种情况，我们就可以在独立消息服务中起一个定时任务，定时去查询 消息状态为 **init** 的数据，去反向查询 订单系统中的单号是否存在，如果存在，那么我们就把消息置为 **send** 状态，然后发送到 消息队列里面，如果查询到不存在的订单，那么就直接抛弃掉这条消息。所以这里我们的订单系统得提供批量查询订单的接口，还有下游的消费系统得保证幂等。保证重复消费的一致性。
3. 消息队列丢消息或者下游系统一直处理失败，导致没有消息反馈过来，出现一直是 **send** 状态的消息。此时独立消息我们还需要一个定时任务，就是处理这种 **send** 状态的消息，我们可以进行重发，直到后面系统消费成功为止。

情况，消息系统会重新发送消息，我们再处理就是。如果是一直失败，你就要考虑是不是你的代码真的有问题，有bug 了吧。

5. 最后的保底方案，**记录日志**，出现问题人肉处理数据。现在我们系统出现错误，以目前的技术手段是没办法做到都靠机器去解决的，都得靠我们人。据我了解，现在很多大厂都会有这样的人，专门处理这种类型的问题，手动去修改数据库的方式。我们之前待的小厂，基本上都是靠我们自己去写 sql 去修改数据的，想想，是不是?

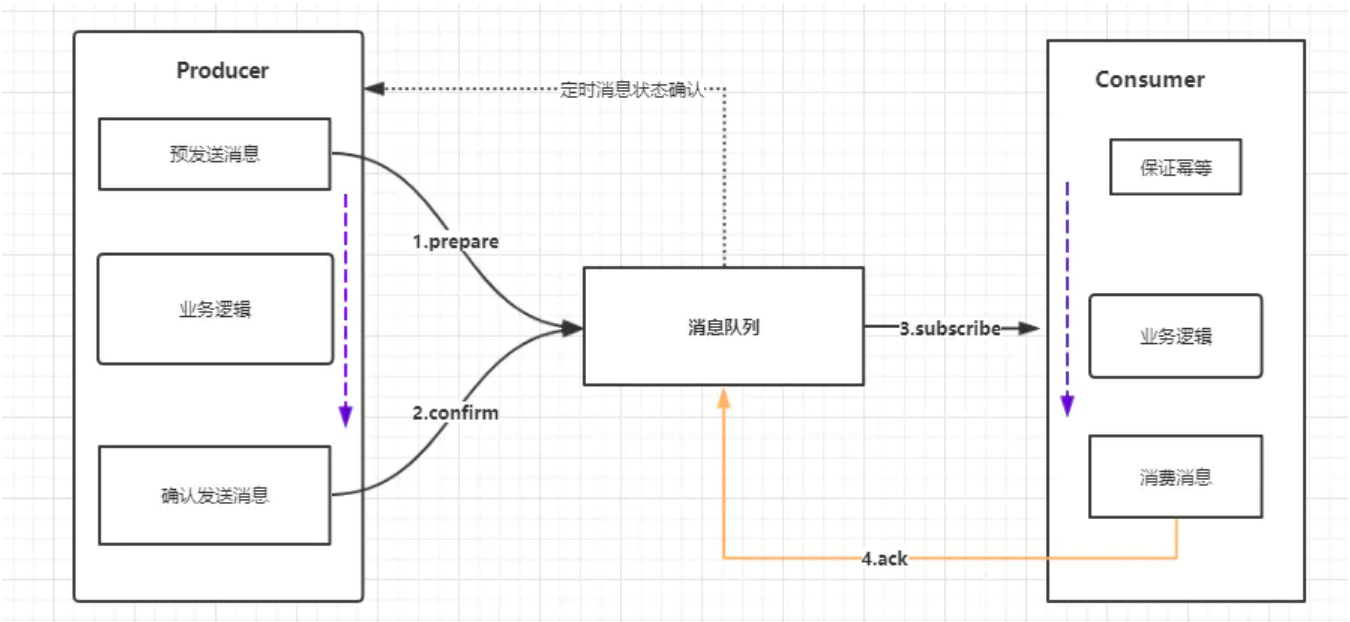
贴一下关键的**独立消息服务核心逻辑代码框架**:

定时任务:

基于 RocketMQ实现

这种方案，跟上面的独立消息服务一致，这里直接去掉独立服务，只利用消息队列来实现，也就是阿里的 **RocketMQ** 。

流程图如下：



这里的整个流程跟上面基于消息服务是一致的。这里就不过多阐述，具体代码实现请参考：www.jianshu.com/p/453c6e7ff... ，写得非常好。

针对这里的 **可靠消息最终一致性方案** 来说，我们说的 **可靠** 是指保证消息一定能发送到消息中间件里面去，保证这里可靠。

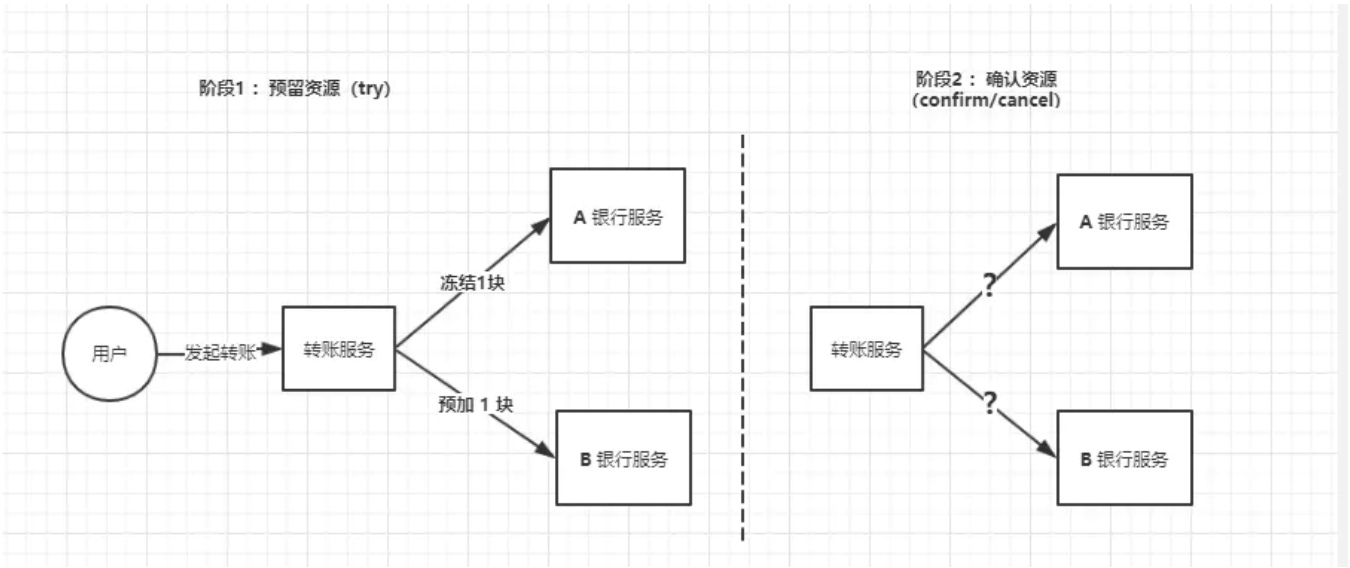
对于下游的系统来说，消费不成功，一般来说就是采取失败重试，重试多次不成功，那么就记录日志，后续人工介入来进行处理。所以这里得强调一下，后面的系统，一定要处理 **幂等**，**重试**，**日志** 这几个东西。

TCC 方案

TCC 的全程分为三个阶段，分别是 Try、Confirm、Cancel：

1. **Try阶段**：这个阶段说的是对各个服务的资源做检测以及对资源进行锁定或者预留
2. **Confirm阶段**：这个阶段说的是在各个服务中执行实际的操作
3. **Cancel阶段**：如果任何一个服务的业务方法执行出错，那么这里就需要进行补偿，就是执行已经执行成功的业务逻辑的回滚操作

还是以转账的例子为例，在跨银行进行转账的时候，需要涉及到两个银行的分布式事务，从A 银行向 B 银行转 1 块，如果用TCC 方案来实现：



大概思路就是这样的：

1. **Try 阶段**：先把A 银行账户先冻结 1 块，B银行账户中的资金给预加 1 块。
2. **Confirm 阶段**：执行实际的转账操作，A银行账户的资金扣减 1块，B 银行账户的资金增加 1 块。
3. **Cancel 阶段**：如果任何一个银行的操作执行失败，那么就需要回滚进行补偿，就是比如A银行账户如果已经扣减了，但是B银行账户资金增加失败了，那么就得把A银行账户资金给加回去。

这种方案就比较复杂了，一步操作要做多个接口来配合完成。

以 ByteTCC 框架的实现例子来大概描述一下上面的流程，示例地址 gitee.com/bytesoft/By...

最开始 A 银行账户 与 B 银行账户都分别为：amount（数量）=1000，frozen（冻结金额）= 0

从A银行账户发起转账到 B 银行账户 1 块：

```
        SpringApplication application = new SpringApplication(GenericConsumerMain.class);
        application.setBannerMode(Banner.Mode.OFF);
        application.run(args);

        try {
            ITransferService transferService = (ITransferService) context.getBean("genericTransferService");
            transferService.transfer("1001", "2001", 1.00d);
        } catch (Exception ex) {
            ex.printStackTrace();
        } finally {
            shutdown();
        }
    }
```

try 阶段：A 银行账户金额减 1，冻结金额 加 1，B 银行 账户 冻结金额加 1 。

```
@Compensable(interfaceClass = ITransferService.class, confirmableKey = "transferServiceConfirm", cancellableKey = "transferServiceCancel")
public class GenericTransferServiceImpl implements ITransferService {

    @javax.annotation.Resource(name = "jdbcTemplate2")
    private JdbcTemplate jdbcTemplate;
    @com.alibaba.dubbo.config.annotation.Reference(interfaceClass = IAccountService.class, group = "x-bytetcc", filter = "bytetcc", loa
    private IAccountService remoteAccountService;

    @Transactional(rollbackFor = ServiceException.class)
    public void transfer(String sourceAcctId, String targetAcctId, double amount) throws ServiceException {
        this.remoteAccountService.decreaseAmount(sourceAcctId, amount);
        this.increaseAmount(targetAcctId, amount);
    }

    private void increaseAmount(String acctId, double amount) throws ServiceException {
        this.jdbcTemplate.update("update tb_account_two set frozen = frozen + ? where acct_id = ?", amount, acctId);
        System.out.printf("exec increase: acct= %s, amount= %7.2f%n", acctId, amount);
    }
```

源账户 金额减 1 ，冻结金额 加 1

目标账户冻结金额 加 1

```
@Service("accountService")
@Compensable(interfaceClass = IAccountService.class, confirmableKey = "accountServiceConfirm", cancellableKey = "accountServiceCancel")
public class AccountServiceImpl implements IAccountService {

    @javax.annotation.Resource(name = "jdbcTemplate1")
    private JdbcTemplate jdbcTemplate;

    @Transactional(propagation = Propagation.REQUIRES_NEW, rollbackFor = ServiceException.class)
    public void increaseAmount(String acctId, double amount) throws ServiceException {
        this.jdbcTemplate.update("update tb_account_one set frozen = frozen + ? where acct_id = ?", amount, acctId);
        System.out.printf("exec increase: acct= %s, amount= %7.2f%n", acctId, amount);
    }

    @Transactional(propagation = Propagation.REQUIRES_NEW, rollbackFor = ServiceException.class)
    public void decreaseAmount(String acctId, double amount) throws ServiceException {
        this.jdbcTemplate.update("update tb_account_one set amount = amount - ?, frozen = frozen + ? where acct_id = ?", amount,
                amount, acctId);
        System.out.printf("exec decrease: acct= %s, amount= %7.2f%n", acctId, amount);
    }

}
```

此时：

- A 银行账户：amount（数量）= 1000 - 1 = 999，frozen（冻结金额）= 0 + 1 = 1
- B 银行账户：amount（数量）= 1000，frozen（冻结金额）= 0 + 1 = 1

confirm 阶段： A银行账户冻结金额 减 1，B 银行账户金额 加 1，冻结金额 减 1

```
    private JdbcTemplate jdbcTemplate;

    @Transactional(rollbackFor = ServiceException.class)
    public void transfer(String sourceAcctId, String targetAcctId, double amount) throws ServiceException {
        this.jdbcTemplate.update("update tb_account_two set amount = amount + ?, frozen = frozen - ? where acct_id = ?", amount,
                amount, targetAcctId);
        System.out.printf("done increase: acct= %s, amount= %7.2f%n", targetAcctId, amount);
    }

}
```

目标账户金额 + 1，冻结金额 -1

```
@Service("accountServiceConfirm")
public class AccountServiceConfirm implements IAccountService {

    @javax.annotation.Resource(name = "jdbcTemplate")
    private JdbcTemplate jdbcTemplate;

    @Transactional(rollbackFor = ServiceException.class)
    public void increaseAmount(String acctId, double amount) throws ServiceException {
        this.jdbcTemplate.update("update tb_account_one set amount = amount + ?, frozen = frozen - ? where acct_id = ?", amount,
                amount, acctId);
        System.out.printf("done increase: acct= %s, amount= %7.2f%n", acctId, amount);
    }

    @Transactional(rollbackFor = ServiceException.class)
    public void decreaseAmount(String acctId, double amount) throws ServiceException {
        this.jdbcTemplate.update("update tb_account_one set frozen = frozen - ? where acct_id = ?", amount, acctId);
        System.out.printf("done decrease: acct= %s, amount= %7.2f%n", acctId, amount);
    }

}
```

源账户 冻结金额 -1

此时：

- A 银行账户：amount（数量）= 999，frozen（冻结金额）= 1 - 1 = 0
- B 银行账户：amount（数量）= 1000 + 1 = 1001，frozen（冻结金额）= 1 - 1 = 0

cancel 阶段： A 银行账户金额 + 1，冻结金额 -1 ，B 银行 冻结金额 -1

```
@Service("accountServiceCancel")
public class AccountServiceCancel implements IAccountService {

    @javax.annotation.Resource(name = "jdbcTemplate")
    private JdbcTemplate jdbcTemplate;

    @Transactional(rollbackFor = ServiceException.class)
    public void increaseAmount(String acctId, double amount) throws ServiceException {
        this.jdbcTemplate.update("update tb_account_one set frozen = frozen - ? where acct_id = ?", amount, acctId);
        System.out.printf("undo increase: acct= %s, amount= %7.2f%n", acctId, amount);
    }

    @Transactional(rollbackFor = ServiceException.class)
    public void decreaseAmount(String acctId, double amount) throws ServiceException {
        this.jdbcTemplate.update("update tb_account_one set amount = amount + ?, frozen = frozen - ? where acct_id = ?", amount,
                amount, acctId);
        System.out.printf("undo decrease: acct= %s, amount= %7.2f%n", acctId, amount);
    }

}
```

源账户 账户金额 加 1 ， 冻结金额 减 1

```
@Service("transferServiceCancel")
public class TransferServiceCancel implements ITransferService {

    @javax.annotation.Resource(name = "jdbcTemplate2")
    private JdbcTemplate jdbcTemplate;

    @Transactional(rollbackFor = ServiceException.class)
    public void transfer(String sourceAcctId, String targetAcctId, double amount) throws ServiceException {
        this.jdbcTemplate.update("update tb_account_two set frozen = frozen - ? where acct_id = ?", amount, targetAcctId);
        System.out.printf("exec decrease: acct= %s, amount= %7.2f%n", targetAcctId, amount);
    }

}
```

目标账户 冻结金额减 1

- A 银行账户：amount（数量）= 999 + 1 = 1000，frozen（冻结金额）= 1 - 1 = 0
- B 银行账户：amount（数量）= 1000，frozen（冻结金额）= 1 - 1 = 0

至此，整个过程就演示完毕，大家记得跑一遍代码。其实还是蛮复杂的，有许多接口一起来配合完成整个业务，试想一下，如果我们项目中大量用到 TCC 来写，你受得了？

再提一下 BASE理论

BASE 理论是 Basically Available(基本可用)，Soft State（软状态）和Eventually Consistent（最终一致性）三个短语的缩写。

1. **基本可用（Basically Available）**： 指分布式系统在出现不可预知故障的时候，允许损失部分可用性。
2. **软状态（ Soft State）**：指允许系统中的数据存在中间状态，并认为该中间状态的存在不会影响系统的整体可用性，即允许系统在不同节点的数据副本之间进行数据同步的过程存在延时。
3. **最终一致（ Eventual Consistency）**：强调的是所有的数据更新操作，在经过一段时间的同步之后，最终都能够达到一个一致的状态。因此，最终一致性的本质是需要系统保证最终数据能够达到一致，而不需要实时保证系统数据的强一致性。

其核心思想是：

> 即使无法做到强一致性（Strong consistency），但每个应用都可以根据自身的业务特点，采用适当的方式来使系统达到最终一致性（Eventual consistency）

到这里大家再想想， 上面 TCC 方案中的账户设计了一个冻结字段 **frozen** ，这里是不是就是 **BASE理论** 中间的 **软状态** 呢？

最后

对存在非常多的微服务的公司来说，服务之间的调用异常的复杂，那么在引入分布式事务的过程中，你需要考虑加入分布式事务后，系统实现起来的复杂性和开发成本，或者说哪些地方根本就不需要搞分布式事务。

其实没必要到处都搞分布式事务，对于大多数的业务来说，其实我们并不需要做分布式事务，直接做日志，做监控就好了。然后出现问题，手工去处理，一个月也不会有那么多的问题的。如果你天天都出现这些问题，你是不是要好好去排查排查你的代码Bug了。

关注下面的标签，发现更多相似文章

面试

乔二爷 Lv2 公众号「乔二爷」
获得点赞 120 · 获得阅读 8,991

关注

评论

输入评论...

相关推荐

专栏 · shanyue · 12小时前 · Node.js / 面试
服务器端如何获得客户端 IP 地址
24

专栏 · 敖丙 · 1天前 · 面试 / Java
「数据库调优」屡试不爽的面试连环combo
140 14

7 2

专栏 · 政采云前端团队 · 2天前 · 面试 / 前端

如何拿下政采云 P6 前端 Offer

77 25

专栏 · zhangferry · 10小时前 · 面试

iOS面试备战-网络篇

3 2

专栏 · cjbniubi · 22小时前 · 面试

手把手带你简单回答真实前端面试题

6 1

专栏 · aoho · 4天前 · 面试 / 后端

抖音、腾讯、阿里、美团春招服务端开发岗位硬核面试（完结）

124 18

专栏 · 山禾 · 2天前 · 面试

【两万字】面试官：听说你很懂集合源码，接我二十道问题！

29 4

专栏 · 前端自学驿站 · 5天前 · 面试

自学前端拿到offer的心路历程

94 46