浅谈分布式事务

文章 发布于 2017年07月21日 阅读 7008

并发编程 分布式架构

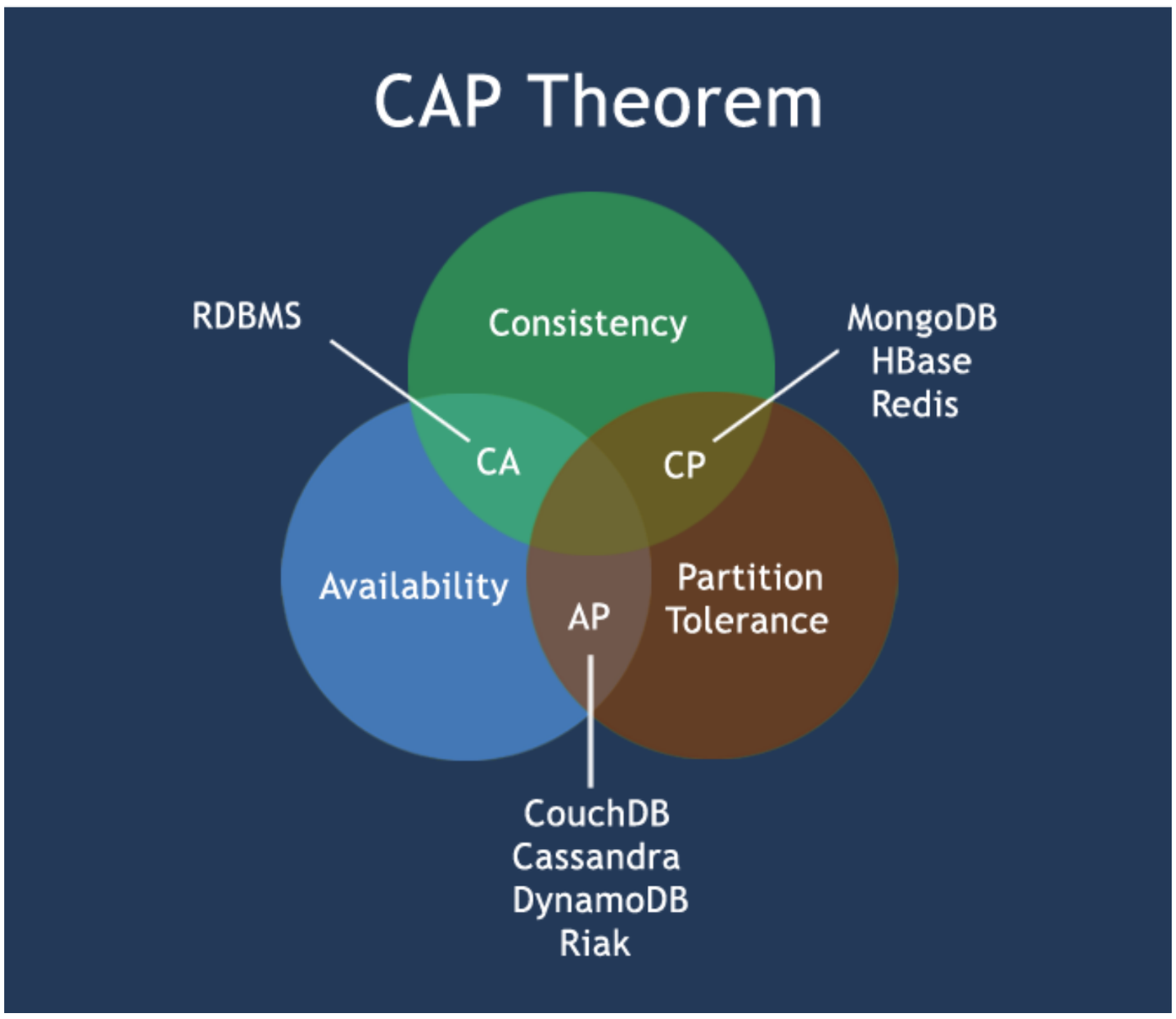
您目前处于: 架构&实践 - 架构

现今互联网界，分布式系统和微服务架构盛行。一个简单操作，在服务端非常可能是由多个服务和数据库实例协同完成的。在一致性要求较高的场景下，多个独立操作之间的一致性问题显得格外棘手。

基于水平扩容能力和成本考虑，传统的强一致的解决方案（e.g.单机事务）纷纷被抛弃。其理论依据就是响当当的CAP原理。往往为了可用性和分区容错性，忍痛放弃强一致支持，转而追求最终一致性。

分布式系统的特性

在分布式系统中，同时满足CAP定律中的一致性 Consistency、可用性 Availability和分区容错性 Partition Tolerance三者是不可能的。在绝大多数的场景，都需要牺牲强一致性来换取系统的高可用性，系统往往只需要保证最终一致性。



CAP理解:

- Consistency：强一致性就是在客户端任何时候看到各节点的数据都是一致的（All nodes see the same data at the same time）。
- Availability：高可用性就是在任何时候都可以读写（Reads and writes always succeed）。
- Partition Tolerance：分区容错性是在网络故障、某些节点不能通信的时候系统仍能继续工作（The system continue to operate despite arbitrary message loss or failure of part of the the system）。以实际效果而言，分区相当于对通信的时限要求。系统如果不能在时限内达成数据一致性，就意味着发生了分区的情况，必须就当前操作在C和A之间做出选择。

ACID理解:

- Atomicity 原子性：一个事务中的所有操作，要么全部完成，要么全部不完成，不会结束在中间某个环节。事务在执行过程中发生错误，会被回滚到事务开始前的状态，就像这个事务从来没有执行过一样。
- Consistency 一致性：在事务开始之前和事务结束以后，数据库的完整性没有被破坏。
- Isolation 隔离性：数据库允许多个并发事务同时对其数据进行读写和修改的能力，隔离性可以防止多个事务并发执行时由于交叉执行而导致数据的不一致。
- Durability 持久性：事务处理结束后，对数据的修改就是永久的，即便系统故障也不会丢失。

分布式事务的基本介绍

分布式事务服务（Distributed Transaction Service，DTS）是一个分布式事务框架，用来保障在大规模分布式环境下事务的最终一致性。

张松然

京东商城，商家研发部架构师。
丰富的构建高性能高可用大规模分布式系统的研发、架构经验。2013年加入京东，目前负责京麦服务网关和京麦服务市场的系统研发工作。

WeChat

热门标签

并发编程 服务性能 Netty
敏捷实践 京麦架构 系统架构
HTTP NIO Protocol Buffers
敏捷之旅 桥接模式 JVM
ElasticSearch Object-C
Zookeeper Redis Tomcat
敏捷工坊 Spring Guava
数据结构与算法 Struts2
Linux MYSQL 网络通信
反应堆模式 jQuery 单元测试
持续集成 影响地图
实例化需求 组合模式
代理模式 日志追踪 Servlet
分布式架构 Git 商学院
阿里云 观察者模式 消息队列
Vue HBase Hadoop

好文推荐

深度解读Tomcat中的NIO模型
Model Operating Systems - Memory Management
京东王栋：618大促网关承载十亿调用量背后的架构实践
阿里商业服务生态解读
构建流式计算卖家日志系统应用实践

好书推荐

CAP理论告诉我们在分布式存储系统中，最多只能实现上面的两点。而由于当前的网络硬件肯定会出现延迟丢包等问题，所以分区容忍性是我们必须需要实现的，所以我们只能在一致性和可用性之间进行权衡。

为了保障系统的可用性，互联网系统大多将强一致性需求转换成最终一致性的需求，并通过系统执行幂等性的保证，保证数据的最终一致性。

数据一致性理解：

- 强一致性：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。这种是对用户最友好的，就是用户上一次写什么，下一次就保证能读到什么。根据 CAP 理论，这种实现需要牺牲可用性。
- 弱一致性：系统并不保证后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。
- 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。在没有故障发生的前提下，不一致窗口的时间主要受通信延迟，系统负载和复制副本的个数影响。DNS 是一个典型的最终一致性系统。

常用的分布式技术说明

1. 本地消息表

这种实现方式的思路是源于ebay，其基本的设计思想是将远程分布式事务拆分成一系列的本地事务。

举个经典的跨行转账的例子来描述。

第一步伪代码如下，扣款1W，通过本地事务保证了凭证消息插入到消息表中。

```
Begin transaction
    update A set amount = amount - 10000 where userId = 1;
    insert into message(userId, price, status) values(1, 10000, 1);
End transaction
commit;
```

第二步，通知对方银行账户上加1W了，通常采用两种方式：

- 采用时效性高的MQ，由对方订阅消息并监听，有消息时自动触发事件。
- 采用定时轮询扫描的方式，去检查消息表的数据。

2. 消息中间件

非事务性的消息中间件

还是以上述提到的跨行转账为例，我们很难保证在扣款完成之后对MQ投递消息的操作就一定能成功。这样一致性似乎很难保证。

```
try {
    bool result = dao.update(model); // 操作数据库失败，会抛出异常
    if (result) {
        mq.send(model); // 如果mq方式执行失败，会抛出异常
    }
} catch(Exception e) {
    rollback(); // 如果发生异常，则回滚
}
```

我们来分析下可能的情况：

- 操作数据库成功，向MQ中投递消息也成功，皆大欢喜。
- 操作数据库失败，不会向MQ中投递消息了。
- 操作数据库成功，但是向MQ中投递消息时失败，向外抛出了异常，刚刚执行的更新数据库的操作将被回滚。

从上面分析的几种情况来看，基本上能保证发送者发送消息的可靠性。我们再来分析下消费者端面临的问题：

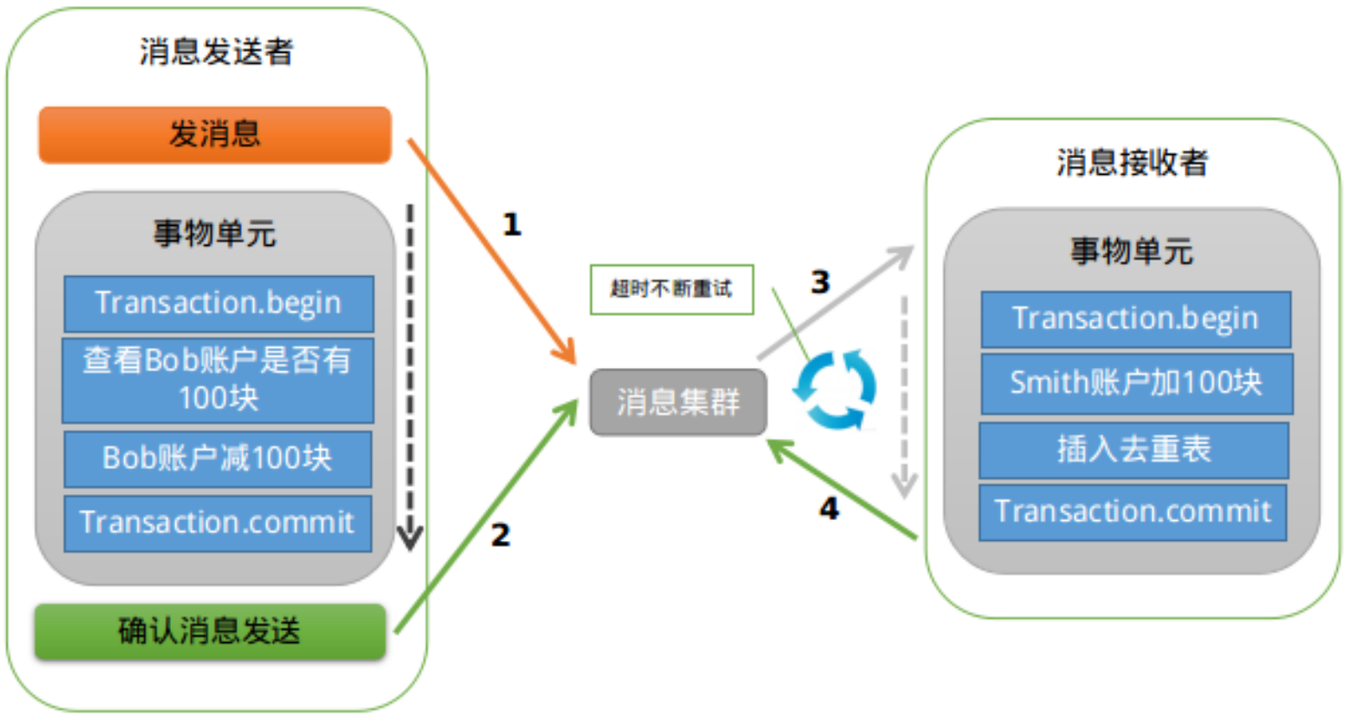
- 消息出列后，消费者对应的业务操作要执行成功。如果业务执行失败，消息不能失效或者丢失。需要保证消息与业务操作一致。
- 尽量避免消息重复消费。如果重复消费，也不能因此影响业务结果。

支持事务的消息中间件

除了上面介绍的通过异常捕获和回滚的方式外，还有没有其他的思路呢？

阿里巴巴的RocketMQ中间件就支持一种事务消息机制，能够确保本地操作和发送消息达到本地事务一样的效果。

- 第一阶段，RocketMQ在执行本地事务之前，会先发送一个Prepared消息，并且会持有这个消息的地址。
- 第二阶段，执行本地事物操作。
- 第三阶段，确认消息发送，通过第一阶段拿到的地址去访问消息，并修改状态，如果本地事务成功，则修改状态为已提交，否则修改状态为已回滚。



但是如果第三阶段的确认消息发送失败了怎么办？RocketMQ会定期扫描消息集群中的事物消息，如果发现了prepare状态的消息，它会向消息发送者确认本地事务是否已执行成功，如果成功是回滚还是继续发送确认消息呢。RocketMQ会根据发送端设置的策略来决定是回滚还是继续发送确认消息。这样就保证了消息发送与本地事务同时成功或同时失败。

目前主流的开源MQ（ActiveMQ、RabbitMQ、Kafka）均未实现对事务消息的支持，比较遗憾的是，RocketMQ事务消息部分的代码也并未开源，需要自己去实现。

理解2PC和3PC协议

为了解决分布式一致性问题，前人在性能和数据一致性的反反复复权衡过程中总结了许多典型的协议和算法。其中比较著名的有二阶提交协议（2 Phase Commitment Protocol），三阶提交协议（3 Phase Commitment Protocol）。

2PC

分布式事务最常用的解决方案就是二阶段提交。在分布式系统中，每个节点虽然可以知晓自己的操作时成功或者失败，却无法知道其他节点的操作的成功或失败。当一个事务跨越多个节点时，为了保持事务的ACID特性，需要引入一个作为协调者的组件来统一掌控所有参与者节点的操作结果并最终指示这些节点是否要把操作结果进行真正的提交。

因此，二阶段提交的算法思路可以概括为：参与者将操作成败通知协调者，再由协调者根据所有参与者的反馈情报决定各参与者是否要提交操作还是中止操作。

所谓的两个阶段是指：第一阶段：准备阶段（投票阶段）和第二阶段：提交阶段（执行阶段）。

第一阶段：投票阶段

该阶段的主要目的在于打探数据库集群中的各个参与者是否能够正常的执行事务，具体步骤如下：

1. 协调者向所有的参与者发送事务执行请求，并等待参与者反馈事务执行结果。

2. 事务参与者收到请求之后，执行事务，但不提交，并记录事务日志。

3. 参与者将自己事务执行情况反馈给协调者，同时阻塞等待协调者的后续指令。

第二阶段：事务提交阶段

在第一阶段协调者的询盘之后，各个参与者会回复自己事务的执行情况，这时候存在三种可能：

1. 所有的参与者回复能够正常执行事务。

2. 一个或多个参与者回复事务执行失败。

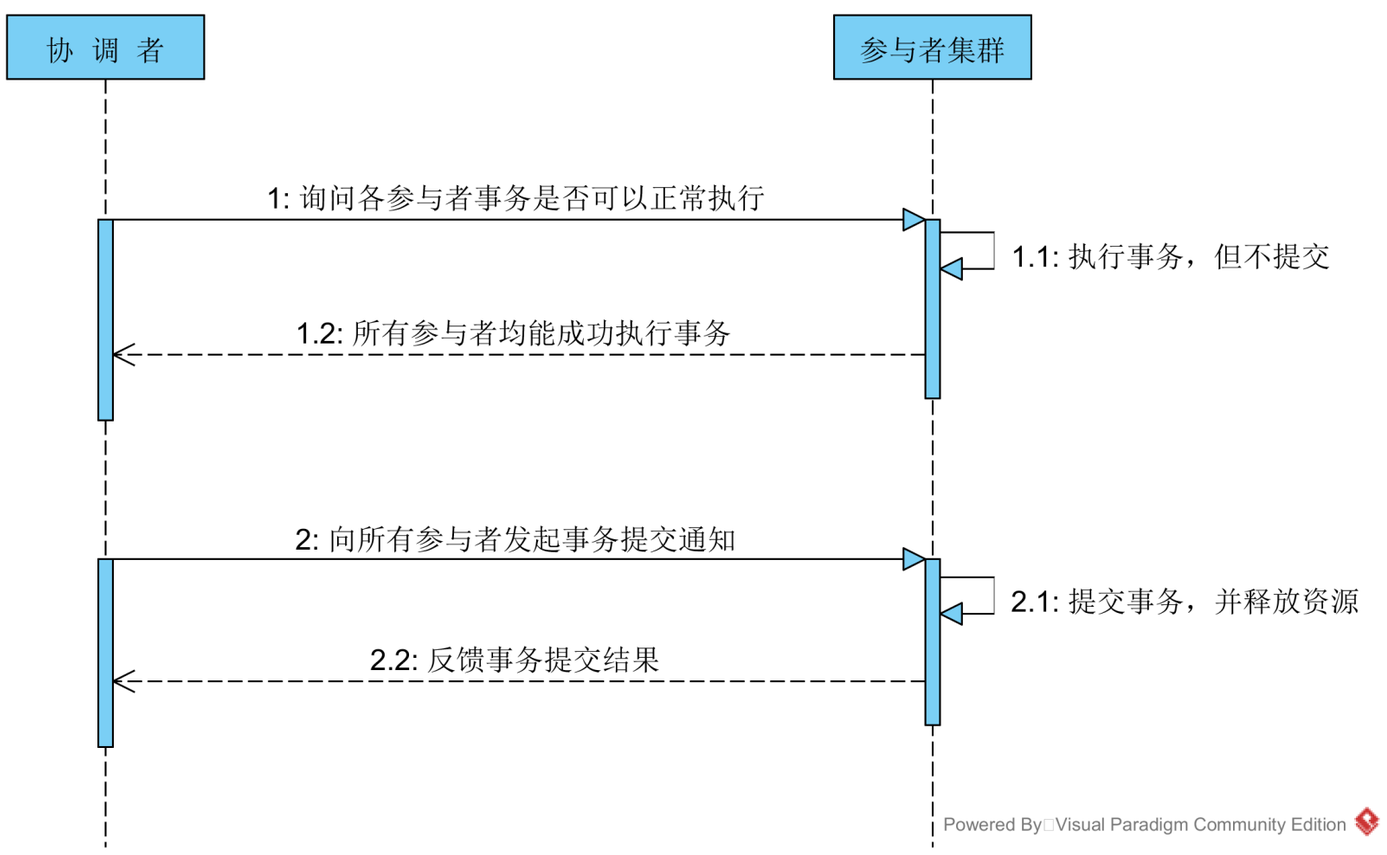
3. 协调者等待超时。

对于第一种情况，协调者将向所有的参与者发出提交事务的通知，具体步骤如下：

1. 协调者向各个参与者发送commit通知，请求提交事务。

2. 参与者收到事务提交通知之后，执行commit操作，然后释放占有的资源。

3. 参与者向协调者返回事务commit结果信息。

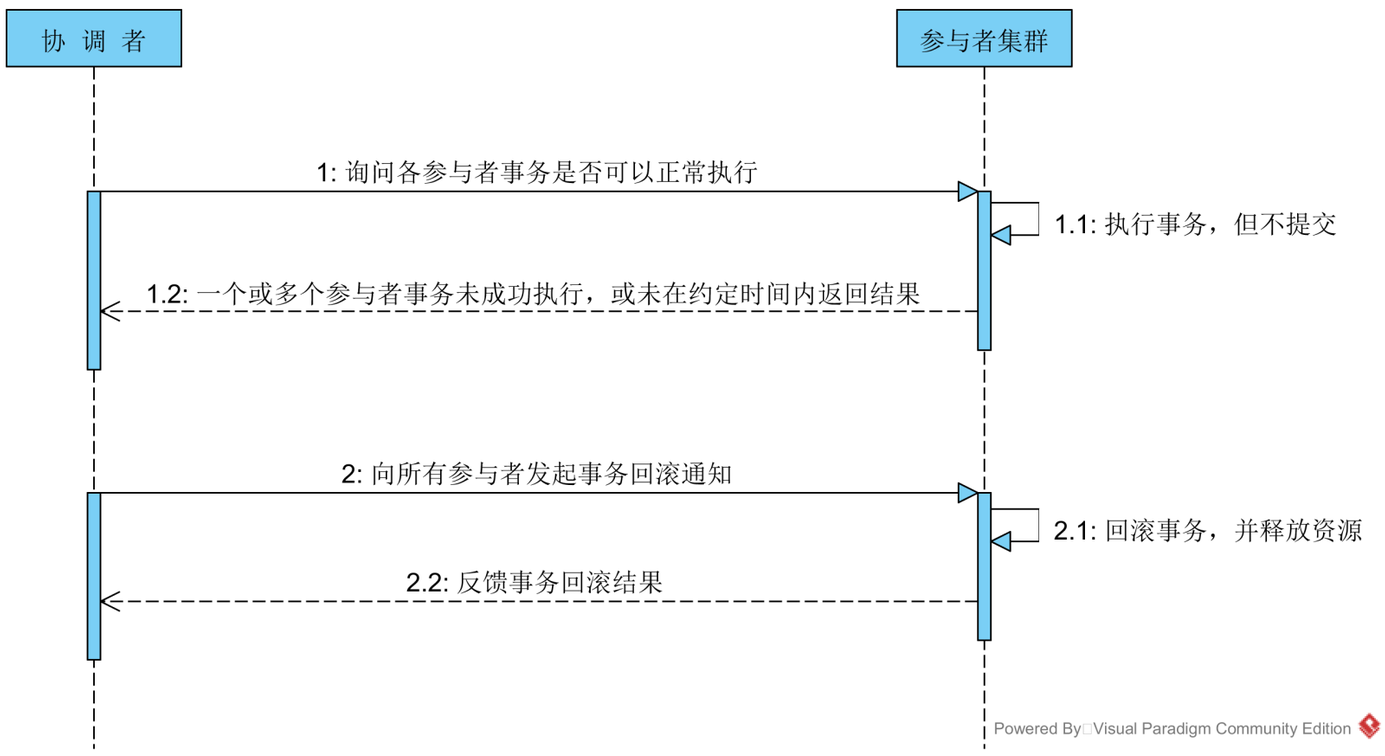


对于第二、三种情况，协调者均认为参与者无法正常成功执行事务，为了整个集群数据的一致性，所以要向各个参与者发送事务回滚通知，具体步骤如下：

1. 协调者向各个参与者发送事务rollback通知，请求回滚事务。

2. 参与者收到事务回滚通知之后，执行rollback操作，然后释放占有的资源。

3. 参与者向协调者返回事务rollback结果信息。



两阶段提交协议解决的是分布式数据库数据强一致性问题，其原理简单，易于实现，但是缺点也是显而易见的，主要缺点如下：

- 单点问题：协调者在整个两阶段提交过程中扮演着举足轻重的作用，一旦协调者所在服务器宕机，那么就会影响整个数据库集群的正常运行，比如在第二阶段中，如果协调者因为故障不能正常发送事务提交或回滚通知，那么参与者们将一直处于阻塞状态，整个数据库集群将无法提供服务。
- 同步阻塞：两阶段提交执行过程中，所有的参与者都需要听从协调者的统一调度，期间处于阻塞状态而不能从事其他操作，这样效率及其低下。
- 数据不一致性：两阶段提交协议虽然为分布式数据强一致性所设计，但仍然存在数据不一致性的可能，比如在第二阶段中，假设协调者发出了事务commit的通知，但是因为网络问题该通知仅被一部分参与者所收到并执行了commit操作，其余的参与者则因为没有收到通知一直处于阻塞状态，这时候就产生了数据的不一致性。

3PC

针对两阶段提交存在的问题，三阶段提交协议通过引入一个“预询盘”阶段，以及超时策略来减少整个集群的阻塞时间，提升系统性能。三阶段提交的三个阶段分别为：can_commit，pre_commit，do_commit。

第一阶段：can_commit

该阶段协调者会去询问各个参与者是否能够正常执行事务，参与者根据自身情况回复一个预估值，相对于真正的执行事务，这个过程是轻量的，具体步骤如下：

1. 协调者向各个参与者发送事务询问通知，询问是否可以执行事务操作，并等待回复。

2. 各个参与者依据自身状况回复一个预估值，如果预估自己能够正常执行事务就返回确定信息，并进入预备状态，否则返回否定信息。

第二阶段：pre_commit

本阶段协调者会根据第一阶段的询盘结果采取相应操作，询盘结果主要有三种：

1. 所有的参与者都返回确定信息。

2. 一个或多个参与者返回否定信息。

3. 协调者等待超时。

针对第一种情况，协调者会向所有参与者发送事务执行请求，具体步骤如下：

1. 协调者向所有的事务参与者发送事务执行通知。

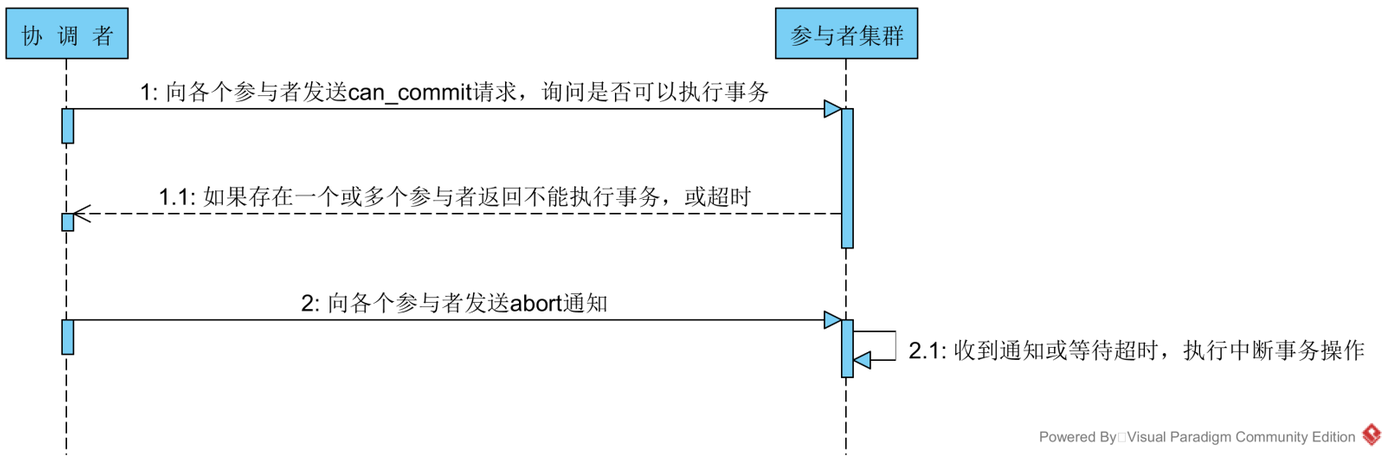
2. 参与者收到通知后，执行事务，但不提交。

3. 参与者将事务执行情况返回给客户端。

在上面的步骤中，如果参与者等待超时，则会中断事务。 针对第二、三种情况，协调者认为事务无法正常执行，于是向各个参与者发出abort通知，请求退出预备状态，具体步骤如下：

1. 协调者向所有事务参与者发送abort通知

2. 参与者收到通知后，中断事务



第三阶段：do_commit

如果第二阶段事务未中断，那么本阶段协调者将会依据事务执行返回的结果来决定提交或回滚事务，分为三种情况：

1. 所有的参与者都能正常执行事务。

2. 一个或多个参与者执行事务失败。

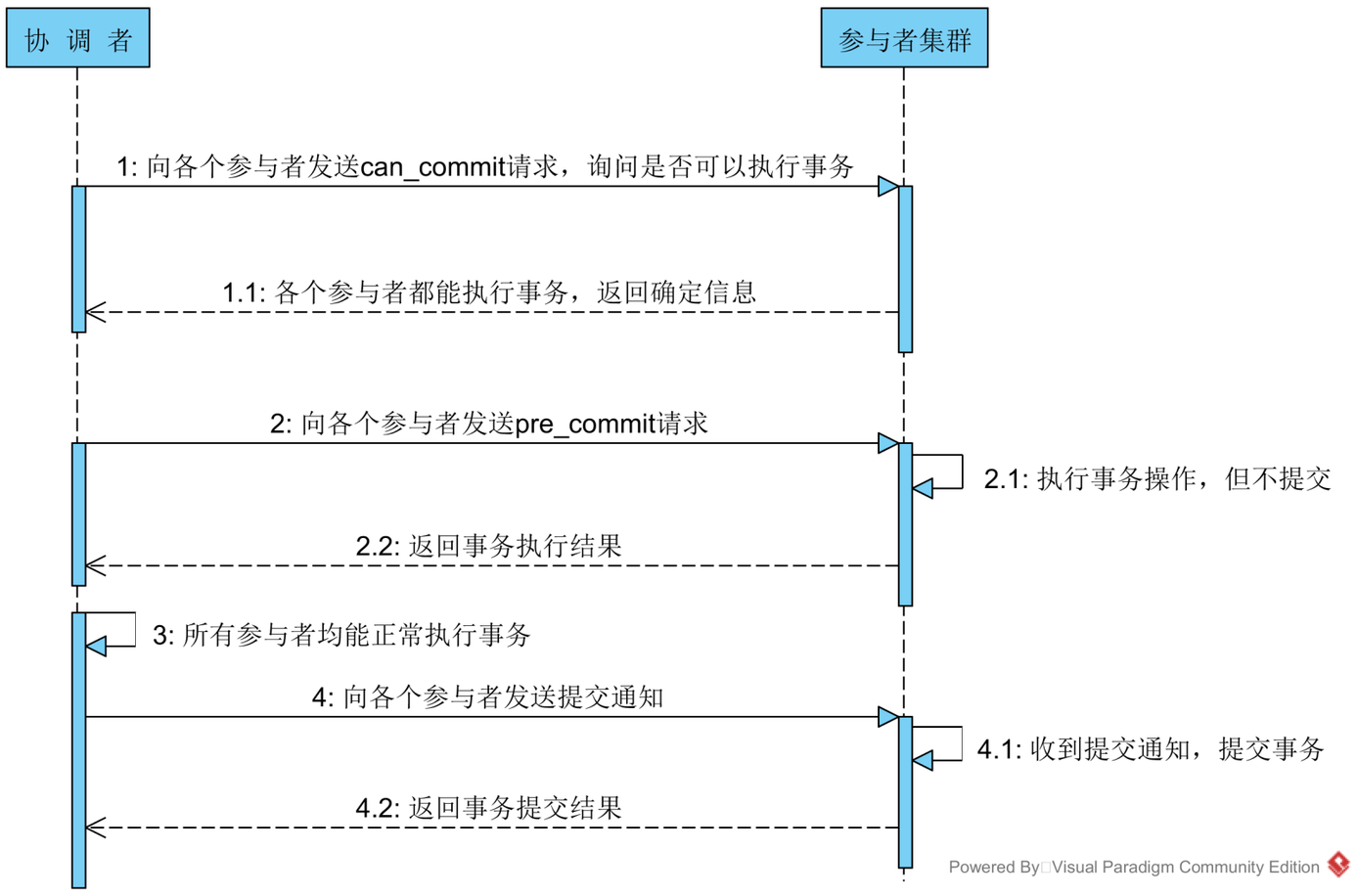
3. 协调者等待超时。

针对第一种情况，协调者向各个参与者发起事务提交请求，具体步骤如下：

1. 协调者向所有参与者发送事务commit通知。

2. 所有参与者在收到通知之后执行commit操作，并释放占有的资源。

3. 参与者向协调者反馈事务提交结果。

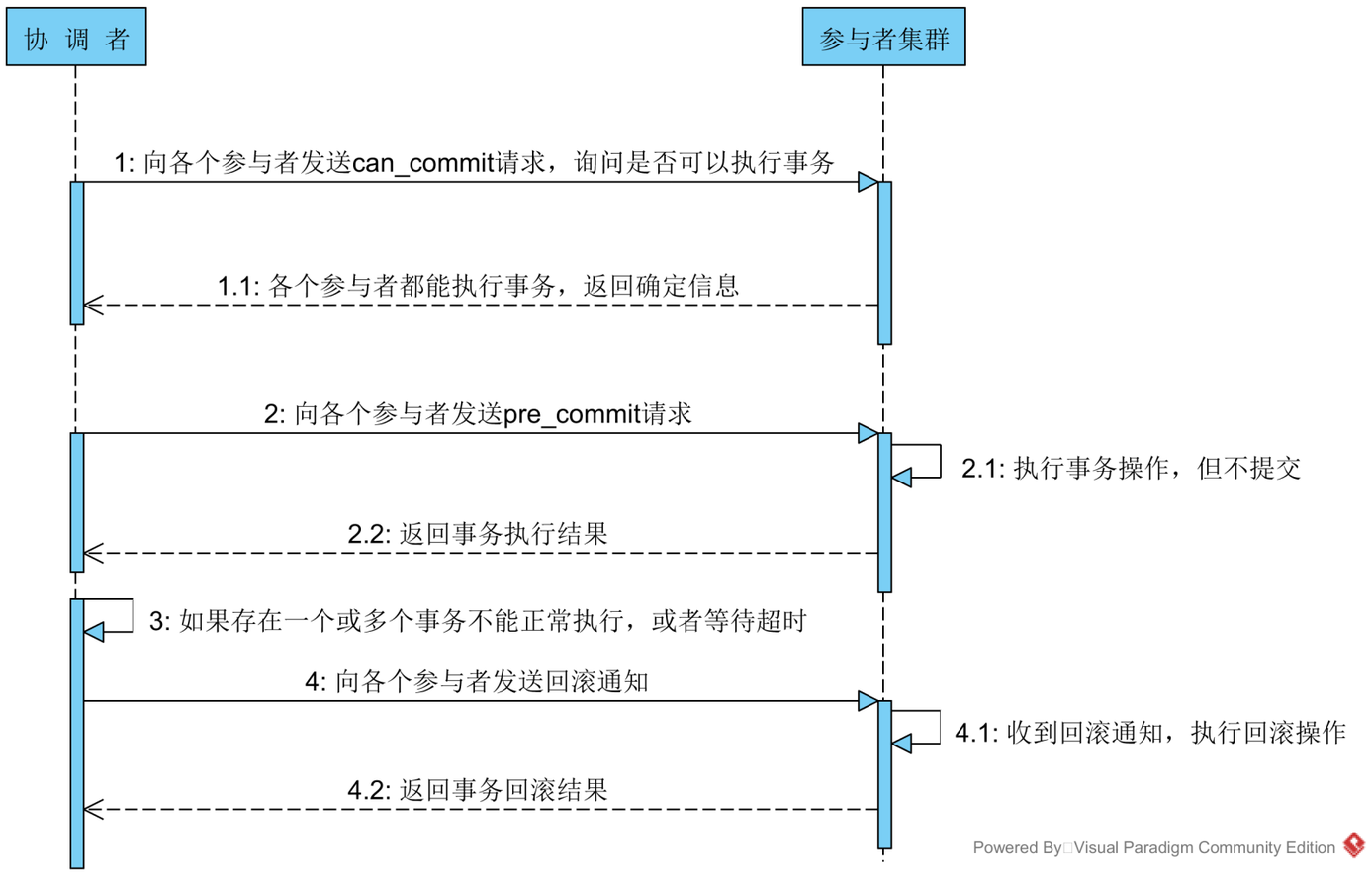


针对第二、三种情况，协调者认为事务无法正常执行，于是向各个参与者发送事务回滚请求，具体步骤如下：

1. 协调者向所有参与者发送事务rollback通知。

2. 所有参与者在收到通知之后执行rollback操作，并释放占有的资源。

3. 参与者向协调者反馈事务提交结果。



在本阶段如果因为协调者或网络问题，导致参与者迟迟不能收到来自协调者的commit或rollback请求，那么参与者将不会如两阶段提交中那样陷入阻塞，而是等待超时后继续commit。相对于两阶段提交虽然降低了同步阻塞，但仍然无法避免数据的不一致性。

Reference

https://zhuanlan.zhihu.com/p/25933039

http://www.infoq.com/cn/articles/solution-of-distributed-system-transaction-consistency

http://blog.csdn.net/jasonsungblog/article/details/49017955

http://blog.csdn.net/suifeng3051/article/details/52691210

https://my.oschina.net/wangzhenchao/blog/736909

评论

1 人参与, 1 条评论

最新评论

qq878400713 [湖南省网友] 2017年9月12日 18:06

您在分布式项目中，用到了那种方式?

回复

关注我们
微信公众号

互链合作　联系我们
QQ群 653389782

LinkedKeeper 是一个致力于打造高品质的技术资源社区。
于**2013年**建站，于**2016年1月**正式更名为 LinkedKeeper，
寓意为，**让知识传播，使你我互联。**社区每个主题都由该领域资深专家撰稿，由浅入深为大家带来一场技术盛宴。

更多内容
新浪微博　Github　简书