聊聊分布式事务，再说说解决方案

前言

最近很久没有写博客了，一方面是因为公司事情最近比较忙，另外一方面是因为在进行 CAP 的下一阶段的开发工作，不过目前已经告一段落了。

接下来还是开始我们今天的话题，说说分布式事务，或者说是我眼中的分布式事务，因为每个人可能对其的理解都不一样。

分布式事务是企业集成中的一个技术难点，也是每一个分布式系统架构中都会涉及到的一个东西，特别是在微服务架构中，几乎可以说是无法避免，本文就分布式事务来简单聊一下。

数据库事务

在说分布式事务之前，我们先从数据库事务说起。 数据库事务可能大家都很熟悉，在开发过程中也会经常使用到。但是即使如此，可能对于一些细节问题，很多人仍然不清楚。比如很多人都知道数据库事务的几个特性：原子性(Atomicity)、一致性(Consistency)、隔离性或独立性(Isolation)和持久性(Durabilily)，简称就是ACID。但是再往下比如问到隔离性指的是什么的时候可能就不知道了，或者是知道隔离性是什么但是再问到数据库实现隔离的都有哪些级别，或者是每个级别他们有什么区别的时候可能就不知道了。

本文并不打算介绍这些数据库事务的这些东西，有兴趣可以搜索一下相关资料。不过有一个知识点我们需要了解，就是假如数据库在提交事务的时候突然断电，那么它是怎么样恢复的呢？ 为什么要提到这个知识点呢？ 因为分布式系统的核心就是处理各种异常情况，这也是分布式系统复杂的地方，因为分布式的网络环境很复杂，这种“断电”故障要比单机多很多，所以我们在做分布式系统的时候，最先考虑的就是这种情况。这些异常可能有机器宕机、网络异常、消息丢失、消息乱序、数据错误、不可靠的TCP、存储数据丢失、其他异常等等...

我们接着说本地事务数据库断电的这种情况，它是怎么保证数据一致性的呢？我们使用SQL Server来举例，我们知道我们在使用 SQL Server 数据库是由两个文件组成的，一个数据库文件和一个日志文件，通常情况下，日志文件都要比数据库文件大很多。数据库进行任何写入操作的时候都是要先写日志的，同样的道理，我们在执行事务的时候数据库首先会记录下这个事务的redo操作日志，然后才开始真正操作数据库，在操作之前首先会把日志文件写入磁盘，那么当突然断电的时候，即使操作没有完成，在重新启动数据库时候，数据库会根据当前数据的情况进行undo回滚或者是redo前滚，这样就保证了数据的强一致性。

接着，我们就说一下分布式事务。

分布式理论

当我们的单个数据库的性能产生瓶颈的时候，我们可能会对数据库进行分区，这里所说的分区指的是物理分区，分区之后可能不同的库就处于不同的服务器上了，这个时候单个数据库的ACID已经不能适应这种情况了，而在这种ACID的集群环境下，再想保证集群的ACID几乎是很难达到，或者即使能达到那么效率和性能会大幅下降，最为关键的是再很难扩展新的分区了，这个时候如果再追求集群的ACID会导致我们的系统变得很差，这时我们就需要引入一个新的理论原则来适应这种集群的情况，就是 CAP 原则或者叫CAP定理，那么CAP定理指的是什么呢?

CAP定理

CAP定理是由加州大学伯克利分校Eric Brewer教授提出来的，他指出WEB服务无法同时满足一下3个属性：

- 一致性(Consistency)： 客户端知道一系列的操作都会同时发生(生效)
- 可用性(Availability)： 每个操作都必须以可预期的响应结束
- 分区容错性(Partition tolerance)： 即使出现单个组件无法可用,操作依然可以完成

具体地讲在分布式系统中，在任何数据库设计中，一个Web应用至多只能同时支持上面的两个属性。显然，任何横向扩展策略都要依赖于数据分区。因此，设计人员必须在一致性与可用性之间做出选择。

这个定理在迄今为止的分布式系统中都是适用的！ 为什么这么说呢?

这个时候有同学可能会把数据库的2PC（两阶段提交）搬出来说话了。OK，我们就来看一下数据库的两阶段提交。

对数据库分布式事务有了解的同学一定知道数据库支持的2PC，又叫做 XA Transactions。

> MySQL从5.5版本开始支持，SQL Server 2005 开始支持，Oracle 7 开始支持。

其中，XA 是一个两阶段提交协议，该协议分为以下两个阶段：

- 第一阶段：事务协调器要求每个涉及到事务的数据库预提交(precommit)此操作，并反映是否可以提交.
- 第二阶段：事务协调器要求每个数据库提交数据。

其中，如果有任何一个数据库否决此次提交，那么所有数据库都会被要求回滚它们在此事务中的那部分信息。这样做的缺陷是什么呢? 咋看之下我们可以在数据库分区之间获得一致性。

如果CAP 定理是对的，那么它一定会影响到可用性。

如果说系统的可用性代表的是执行某项操作相关所有组件的可用性的和。那么在两阶段提交的过程中，可用性就代表了涉及到的每一个数据库中可用性的和。我们假设两阶段提交的过程中每一个数据库都具有99.9%的可用性，那么如果两阶段提交涉及到两个数据库，这个结果就是99.8%。根据系统可用性计算公式，假设每个月43200分钟，99.9%的可用性就是43157分钟, 99.8%的可用性就是43114分钟，相当于每个月的宕机时间增加了43分钟。

以上，可以验证出来，CAP定理从理论上来讲是正确的，CAP我们先看到这里，等会再接着说。

BASE理论

在分布式系统中，我们往往追求的是可用性，它的重要程序比一致性要高，那么如何实现高可用性呢? 前人已经给我们提出来了另外一个理论，就是BASE理论，它是用来对CAP定理进行进一步扩充的。BASE理论指的是:

- Basically Available（基本可用）
- Soft state（软状态）
- Eventually consistent（最终一致性）

BASE理论是对CAP中的一致性和可用性进行一个权衡的结果，理论的核心思想就是：**我们无法做到强一致，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）。

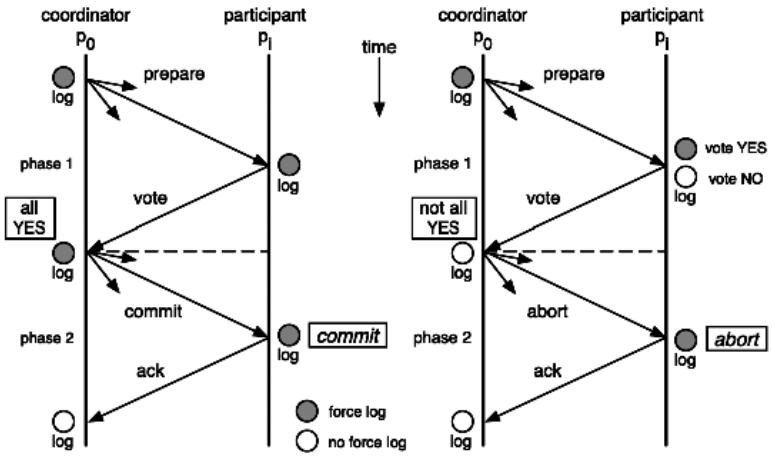
有了以上理论之后，我们来看一下分布式事务的问题。

分布式事务

在分布式系统中，要实现分布式事务，无外乎那几种解决方案。

一、两阶段提交（2PC）

和上一节中提到的数据库XA事务一样，两阶段提交就是使用XA协议的原理，我们可以从下面这个图的流程来很容易的看出中间的一些比如commit和abort的细节。



两阶段提交这种解决方案属于牺牲了一部分可用性来换取的一致性。在实现方面，在 .NET 中，可以借助 TransactionScop 提供的 API 来编程实现分布式系统中的两阶段提交，比如WCF中就有实现这部分功能。不过在多服务器之间，需要依赖于DTC来完成事务一致性，Windows下微软搞的有MSDTC服务，Linux下就比较悲剧了。

另外说一句，TransactionScop 默认不能用于异步方法之间事务一致，因为事务上下文是存储于当前线程中的，所以如果是在异步方法，需要显式的传递事务上下文。

优点： 尽量保证了数据的强一致，适合对数据强一致要求很高的关键领域。（其实也不能100%保证强一致）

缺点： 实现复杂，牺牲了可用性，对性能影响较大，不适合高并发高性能场景，如果分布式系统跨接口调用，目前 .NET 界还没有实现方案。

二、补偿事务（TCC）

TCC 其实就是采用的补偿机制，其核心思想是：针对每个操作，都要注册一个与其对应的确认和补偿（撤销）操作。它分为三个阶段：

- Try 阶段主要是对业务系统做检测及资源预留
- Confirm 阶段主要是对业务系统做确认提交，Try阶段执行成功并开始执行 Confirm阶段时，默认 Confirm阶段是不会出错的。即：只要Try成功，Confirm一定成功。
- Cancel 阶段主要是在业务执行错误，需要回滚的状态下执行的业务取消，预留资源释放。

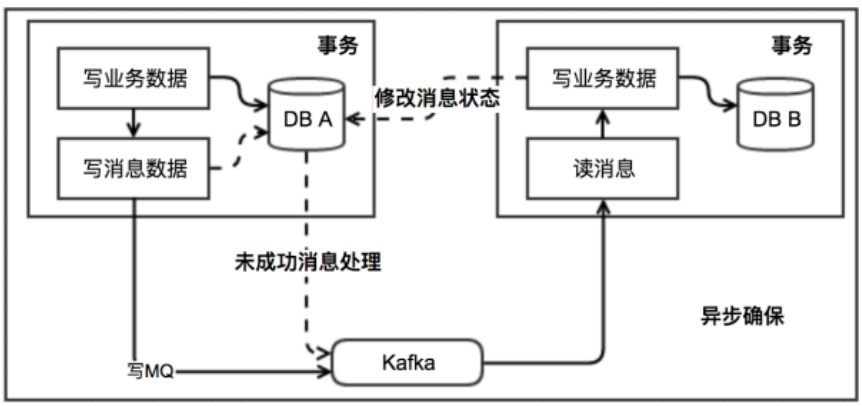
举个例子，假入 Bob 要向 Smith 转账，思路大概是：
我们有一个本地方法，里面依次调用
1、首先在 Try 阶段，要先调用远程接口把 Smith 和 Bob 的钱给冻结起来。
2、在 Confirm 阶段，执行远程调用的转账的操作，转账成功进行解冻。
3、如果第2步执行成功，那么转账成功，如果第二步执行失败，则调用远程冻结接口对应的解冻方法(Cancel)。

优点： 跟2PC比起来，实现以及流程相对简单了一些，但数据的一致性比2PC也要差一些

缺点： 缺点还是比较明显的，在2,3步中都有可能失败。TCC属于应用层的一种补偿方式，所以需要程序员在实现的时候多写很多补偿的代码，在一些场景中，一些业务流程可能用TCC不太好定义及处理。

三、本地消息表（异步确保）

本地消息表这种实现方式应该是业界使用最多的，其核心思想是将分布式事务拆分成本地事务进行处理，这种思路是来源于ebay。我们可以从下面的流程图中看出其中的一些细节：



基本思路就是：

消息生产方，需要额外建一个消息表，并记录消息发送状态。消息表和业务数据要在一个事务里提交，也就是说他们要在一个数据库里面。然后消息会经过MQ发送到消息的消费方。如果消息发送失败，会进行重试发送。

消息消费方，需要处理这个消息，并完成自己的业务逻辑。此时如果本地事务处理成功，表明已经处理成功了，如果处理失败，那么就会重试执行。如果是业务上面的失败，可以给生产方发送一个业务补偿消息，通知生产方进行回滚等操作。

生产方和消费方定时扫描本地消息表，把还没处理完成的消息或者失败的消息再发送一遍。如果有靠谱的自动对账补账逻辑，这种方案还是非常实用的。

这种方案遵循BASE理论，采用的是最终一致性，笔者认为是这几种方案里面比较适合实际业务场景的，即不会出现像2PC那样复杂的实现(当调用链很长的时候，2PC的可用性是非常低的)，也不会像TCC那样可能出现确认或者回滚不了的情况。

优点： 一种非常经典的实现，避免了分布式事务，实现了最终一致性。在 .NET中 有现成的解决方案。

缺点： 消息表会耦合到业务系统中，如果没有封装好的解决方案，会有很多杂活需要处理。

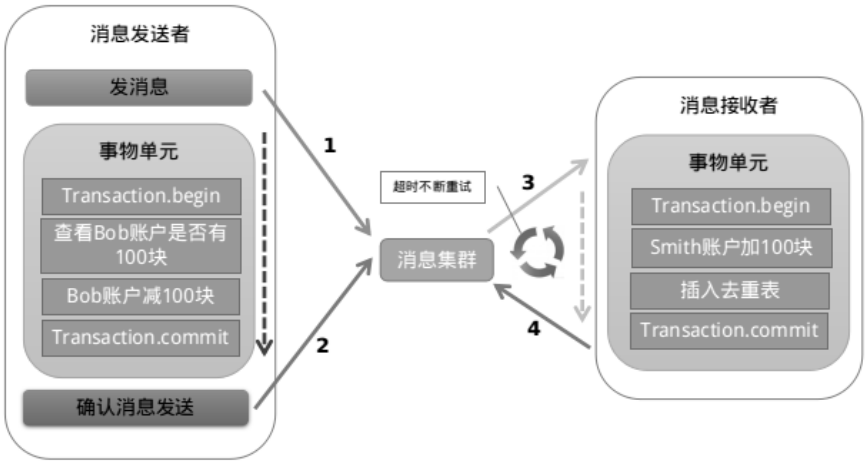
四、MQ 事务消息

有一些第三方的MQ是支持事务消息的，比如RocketMQ，他们支持事务消息的方式也是类似于采用的二阶段提交，但是市面上一些主流的MQ都是不支持事务消息的，比如 RabbitMQ 和 Kafka 都不支持。

以阿里的 RocketMQ 中间件为例，其思路大致为：

第一阶段Prepared消息，会拿到消息的地址。
第二阶段执行本地事务，第三阶段通过第一阶段拿到的地址去访问消息，并修改状态。

也就是说在业务方法内要想消息队列提交两次请求，一次发送消息和一次确认消息。如果确认消息发送失败了RocketMQ会定期扫描消息集群中的事务消息，这时候发现了Prepared消息，它会向消息发送者确认，所以生产方需要实现一个check接口，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。



遗憾的是，RocketMQ并没有 .NET 客户端。有关 RocketMQ的更多消息，大家可以查看这篇博客

优点： 实现了最终一致性，不需要依赖本地数据库事务。

缺点： 实现难度大，主流MQ不支持，没有.NET客户端，RocketMQ事务消息部分代码也未开源。

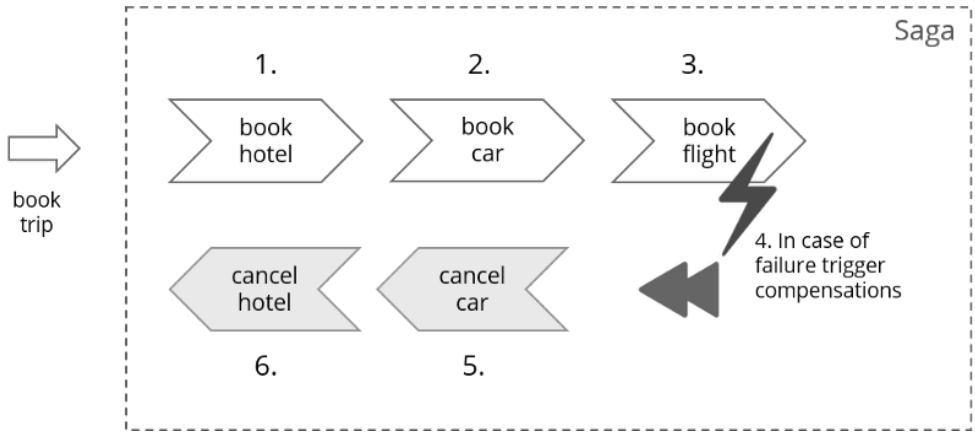
五、Sagas 事务模型

Saga事务模型又叫做长时间运行的事务（Long-running-transaction），它是由普林斯顿大学的H.Garcia-Molina等人提出，它描述的是另外一种在没有两阶段提交的的情况下解决分布式系统中复杂的业务事务问题。你可以在这里看到 Sagas 相关论文。

我们这里说的是一种基于 Sagas 机制的工作流事务模型，这个模型的相关理论目前来说还是比较新的，以至于百度上几乎没有什么相关资料。

该模型其核心思想就是拆分分布式系统中的长事务为多个短事务，或者叫多个本地事务，然后由 Sagas 工作流引擎负责协调，如果整个流程正常结束，那么就算是业务成功完成，如果在这过程中实现失败，那么Sagas工作流引擎就会以相反的顺序调用补偿操作，重新进行业务回滚。

比如我们一次关于购买旅游套餐业务操作涉及到三个操作，他们分别是预定车辆，预定宾馆，预定机票，他们分别属于三个不同的远程接口。可能从我们程序的角度来说他们不属于一个事务，但是从业务角度来说是属于同一个事务的。



他们的执行顺序如上图所示，所以当发生失败时，会依次进行取消的补偿操作。

因为长事务被拆分了很多个业务流，所以 Sagas 事务模型最重要的一个部件就是工作流或者你也可以叫流程管理器（Process Manager），工作流引擎和Process Manager虽然不是同一个东西，但是在这里，他们的职责是相同的。在选择工作流引擎之后，最终的代码也许看起来是这样的

```
SagaBuilder saga = SagaBuilder.newSaga("trip")
        .activity("Reserve car", ReserveCarAdapter.class)
        .compensationActivity("Cancel car", CancelCarAdapter.class)
        .activity("Book hotel", BookHotelAdapter.class)
        .compensationActivity("Cancel hotel", CancelHotelAdapter.class)
        .activity("Book flight", BookFlightAdapter.class)
        .compensationActivity("Cancel flight", CancelFlightAdapter.class)
        .end()
        .triggerCompensationOnAnyError();

camunda.getRepositoryService().createDeployment()
        .addModelInstance(saga.getModel())
        .deploy();
```

这里有一个 C# 相关示例，有兴趣的同学可以看一下。

优缺点这里我们就不说了，因为这个理论比较新，目前市面上还没有什么解决方案，即使是 Java 领域，我也没有搜索的太多有用的信息。

分布式事务解决方案：CAP

上面介绍的那些分布式事务的处理方案你在其他地方或许也可以看到，但是并没有相关的实际代码或者是开源代码，所以算不上什么干货，下面就放干货了。

在 .NET 领域，似乎没有什么现成的关于分布式事务的解决方案，或者说是有但未开源。具笔者了解，有一些公司内部其实是有这种解决方案的，但是也是作为公司的一个核心产品之一，并未开源...

鉴于以上原因，所以博主就打算自己写一个并且开源出来，所以从17年初就开始做这个事情，然后花了大半年的时间在一直不断完善，就是下面这个 CAP。

Github CAP：这里的 CAP 就不是 CAP 理论了，而是一个 .NET 分布式事务解决方案的名字。

详细介绍：
http://www.cnblogs.com/savorboard/p/cap.html
相关文档：
http://www.cnblogs.com/savorboard/p/cap-document.html

夸张的是，这个解决方案是具有可视化界面（Dashboard）的，你可以很方面的看到哪些消息执行成功，哪些消息执行失败，到底是发送失败还是处理失败，一眼便知。

最夸张的是，这个解决方案的可视化界面还提供了**实时动态图表**，这样不但可以看到实时的消息发送及处理情况，连当前的系统处理消息的速度都可以看到，还可以看到过去24小时内的历史消息吞吐量。

最最夸张的是，这个解决方案的还帮你集成了 Consul 做分布式节点发现和注册还有心跳检查，你随时可以看到其他的节点的状况。

最最最夸张的是，你以为你看其他节点的数据要登录到其他节点的Dashboard控制台看？错了，你随便打开其中任意一个节点的Dashboard，点一下就可以切换到你想看的节点的控制台界面了，就像你看本地的数据一样，他们是完全去中心化的。

你以为这些就够了？不，远远不止：

- CAP 同时支持 RabbitMQ，Kafka 等消息队列
- CAP 同时支持 SQL Server, MySql, PostgreSql 等数据库
- CAP Dashboard 同时支持中文和英文界面双语言，妈妈再也不用担心我看不懂了
- CAP 提供了丰富的接口可以供扩展，什么序列化了，自定义处理了，自定义发送了统统不在话下
- CAP 基于MIT开源，你可以尽管拿去做二次开发。（记得保留MIT的License）

这下你以为我说完了？ 不！

你完全可以把 CAP 当做一个 EventBus 来使用，CAP具有优秀的消息处理能力，不要担心瓶颈会在CAP，那是永远不可能， 因为你随时可以在配置中指定CAP处理的消息使用的进程数， 只要你的数据库配置足够高...

说了这么多，口干舌燥的，你不 **Star** 一下给个精神上的支持说不过去吧？ ^_^

2号传送门： https://github.com/dotnetcore/CAP

> 不 Star 也没关系，我选择原谅你~

总结

通过本文我们了解到两个分布式系统的理论，他们分别是CAP和BASE 理论，同时我们也总结并对比了几种分布式分解方案的优缺点，分布式事务本身是一个技术难题，是没有一种完美的方案应对所有场景的，具体还是要根据业务场景去抉择吧。 然后我们介绍了一种基于本地消息的的分布式事务解决方案CAP。

如果你觉得本篇文章对您有帮助的话，感谢您的【推荐】。

如果你对 .NET Core 有兴趣的话可以关注我，我会定期的在博客分享我的学习心得。

分类: .NET Core , Micro Service

标签: 分布式事务 , CAP

« 上一篇： .NET Core 在程序集中集成Razor视图
» 下一篇： 谈谈微服务中的 API 网关（API Gateway）

posted @ 2017-10-17 09:02 Savorboard 阅读(289648) 评论(71) 编辑 收藏

< Prev 1 2

评论列表

#51楼 2018-12-11 11:05 **架构进化论** 回复 引用

好文要顶

支持(0) 反对(0)

#52楼 2019-01-03 09:49 **行舟水上_温酒听雨** 回复 引用

感谢博主分享,另一篇也不错,讲的挺透彻:https://blog.csdn.net/weixin_40533111/article/details/85069536

支持(0) 反对(1)

#53楼 2019-01-30 16:25 **阳光明媚的你** 回复 引用

支持(0) 反对(0)

#54楼 2019-02-12 14:31 **无视丶** 回复 引用

RocketMQ 4.0 支持事务咯

支持(0) 反对(0)

#55楼 2019-02-13 14:41 **Chendisheng** 回复 引用

github主页说明文档，直接点击【中文】访问不了 链接URL
https://github.com/dotnetcore/CAP/blob/develop/README.zh-cn.md
master分支的url可以用
https://github.com/dotnetcore/CAP/blob/master/README.zh-cn.md

支持(0) 反对(0)

#56楼 2019-02-14 08:34 **withoutaword** 回复 引用

给力

支持(0) 反对(0)

#57楼 2019-02-14 23:23 **kayda** 回复 引用

楼主的文章确实给力 还需要再细细读几遍

支持(0) 反对(0)

#58楼 2019-02-19 21:41 **存..** 回复 引用

关于第三个事务的思路 和 第二个其实是差不多的，只是加了重复确认提交而已

支持(0) 反对(0)

#59楼 2019-02-28 15:07 **小祺x** 回复 引用

为什么没有.net版的TCC开源框架

支持(0) 反对(0)

#60楼 2019-03-05 20:56 **yunque32** 回复 引用

看楼主讲的这么好,好想马上用起来

支持(0) 反对(0)

#61楼 2019-03-15 17:08 **社会青年** 回复 引用

说实话，跟楼主不是一个阶级的，看到我一脸懵逼！

支持(3) 反对(0)

#62楼 2019-05-07 00:48 **冬眠的山谷** 回复 引用

ＭＱ实现事务
第一阶段Prepared消息，会拿到消息的地址。
第二阶段执行本地事务，第三阶段通过第一阶段拿到的地址去访问消息，并修改状态。

这里第一阶段的处理是为了什么呢?

如果如下处理会有啥问题
第一阶段：本地事务
第二阶段：发送消息直到ack返回－提交本地事务；或者超时时回退本地事务

支持(0) 反对(0)

#63楼 2019-05-20 13:10 **DarryRing** 回复 引用

netcore里面是不是有masstrasint 分布式事物

支持(0) 反对(0)

#64楼 2019-08-23 17:48 **L_老帅** 回复 引用

赞

支持(0) 反对(0)

#65楼 2019-09-07 13:30 **大黄蜂2019** 回复 引用

一致性，不是指同时发生，在同一时刻的各个存储系统的数据是否保持者相同的值。
可用性，也不是“每个操作都必须以可预期的响应结束”，是指发生故障后，集群中是否能继续提供服务。
这些概念可以百度出来的。。。

支持(0) 反对(0)

#66楼 2019-09-25 17:31 **akka_li** 回复 引用

@ 冬眠的山谷
MQ事务消息方案中，之所以分成两个阶段，是为了应对各种异常情况的！例如文中就提到了一种异常情况以及应对方法--“如果确认消息发送失败了，RocketMQ会定期扫描消息集群中的事务消息，这时候发现了Prepared消息，它会向消息发送者确认”!

至于层主说得这种方案，其中【第二阶段：发送消息直到ack返回－提交本地事务；或者超时时回退本地事务】这一步中你提到的“或者超时时回退本地事务”，其中发生超时异常时，存在两种可能，一种可能是网路不同，连接超时，消息没到达目的地(服务)，另一种可能是连接成功，消息到达了目的地，在本地等待对方响应的时候超时了；所以这里不能一遇到超时回退本地事务；你需要自己写一个判断逻辑进行处理！
之前我就就用过层主所说的这种方法处理分布式事务，写这些异常处理代码，写吐了！！！

支持(0) 反对(0)

#67楼 2019-09-25 17:44 **akka_li** 回复 引用

@ StevenWash
“您好，写的非常好，非常感谢。不过有个小问题请教一下，您文中CAP理论的那一部分的这段话『我们假设两阶段提交的过程中每一个数据库都具有99.9%的可用性，那么如果两阶段提交涉及到两个数据库，这个结果就是99.8%』，想请问一下，如果两阶段提交则是99.8%这是怎么的到的啊！ ”
---- 99.9% * 99.9% = 99.8%；某两阶段提交事物过程中涉及到两个数据库，那么这个两阶段提交事务要想成功，就必须保证这两个数据库都可用，假设每个数据库的可用性是99.9%，那么这两个数据库同时可用的概率就是99.9% * 99.9%！

支持(0) 反对(0)

#68楼 2019-11-02 17:25 **给我一个理由** 回复 引用

所以cap 用的是哪一种
2pc
tcc
本地消息表
RocketMQ 可靠消息最终一致性方案
最大努力通知方案

这五种中的哪一种是你的视线思路呢?

支持(0) 反对(0)

#69楼 2019-12-20 15:18 **咕-咚** 回复 引用

“夸张“一词.用的真是妙!

支持(0) 反对(0)

#70楼 2019-12-20 15:35 **咕-咚** 回复 引用

你在成都，那家公司啊

支持(0) 反对(0)

#71楼 2020-03-24 22:23 **bibd** 回复 引用

博主有没有关注过Orleans项目, 完全支持分布式系统ACID, 相关论文 https://www.microsoft.com/en-us/research/wp-content/uploads/2016/10/EldeebBernstein-TransactionalActors-MSR-TR-1.pdf
我看过之后感觉实现方式挺好, 解决的很完美, 但我水平有限, 如果您感兴趣, 能否看看有没有什么问题.

支持(0) 反对(0)

< Prev 1 2

刷新评论 刷新页面 返回顶部

发表评论

编辑 预览

支持 Markdown