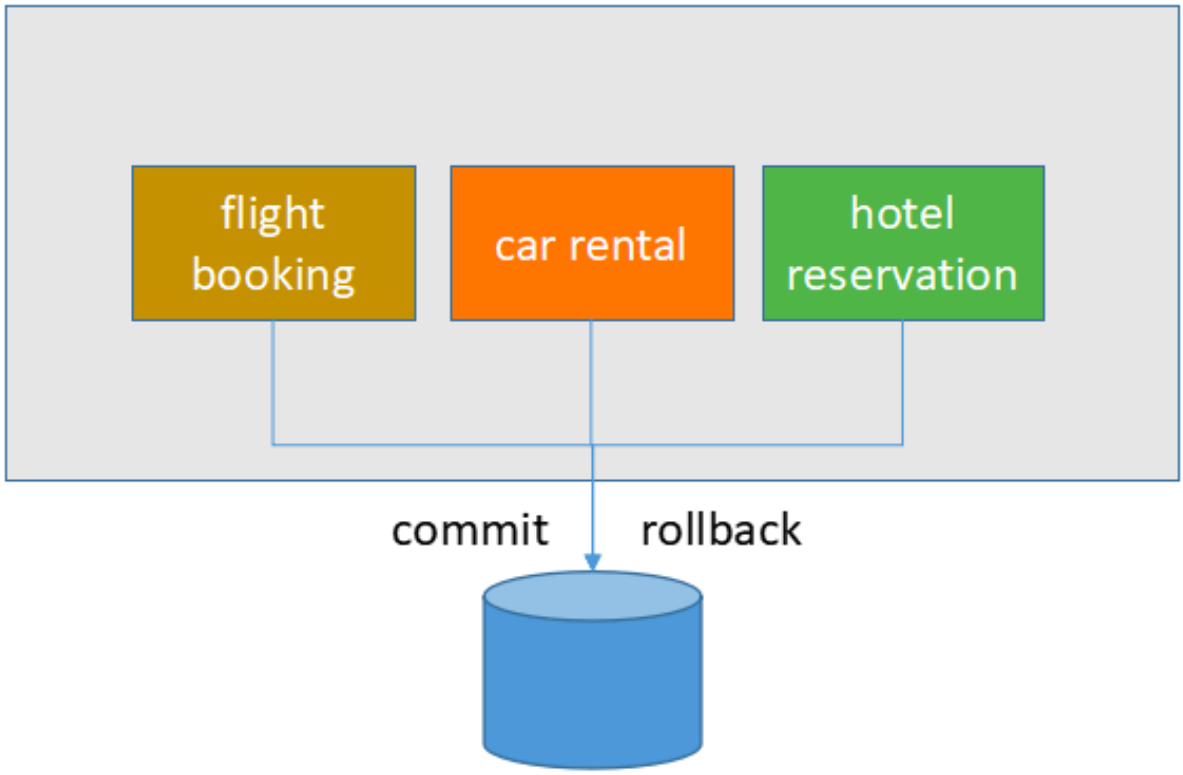
Java面试题-分布式相关

一.怎么考虑数据一致性问题

1.单体应用的数据一致性

想象一下如果我们经营着一家大型企业，下属有航空公司、租车公司、和连锁酒店。我们为客户提供一站式的旅游行程规划服务，这样客户只需要提供出行目的地，我们帮助客户预订机票、租车、以及预订酒店。从业务的角度，我们必须保证上述三个服务的预订都完成才能满足一个成功的旅游行程，否则不能成行。

我们的单体应用要满足这个需求非常简单，只需将这个三个服务请求放到同一个数据库事务中，数据库会帮我们保证全部成功或者全部回滚。



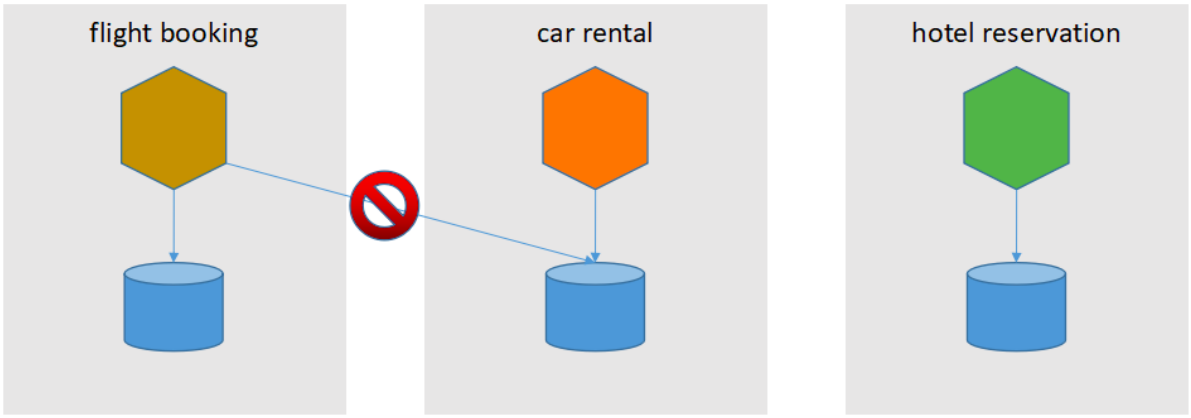
当这个功能上线以后，公司非常满意，客户也非常高兴。

2.微服务场景下的数据一致性

这几年中，我们的行程规划服务非常成功，企业蒸蒸日上，用户量也翻了数十

倍。企业的下属航空公司、租车公司、和连锁酒店也相继推出了更多服务以满足客户需求，我们的应用和开发团队也因此日渐庞大。如今我们的单体应用已变得如此复杂，以至于没人了解整个应用是怎么运作的。更糟的是新功能的上线现在需要所有研发团队合作，日夜奋战数周才能完成。看着市场占有率每况愈下，公司高层对研发部门越来越不满意。

经过数轮讨论，我们最终决定将庞大的单体应用一分为四：机票预订服务、租车服务、酒店预订服务、和支付服务。服务各自使用自己的数据库，并通过 HTTP 协议通信。负责各服务的团队根据市场需求按照自己的开发节奏发版上线。如今我们面临新的挑战：如何保证最初三个服务的预订都完成才能满足一个成功的旅游行程， 否则不能成行的业务规则？现在服务有各自的边界，而且数据库选型也不尽相同，通过数据库保证数据一致性的方案已不可行。

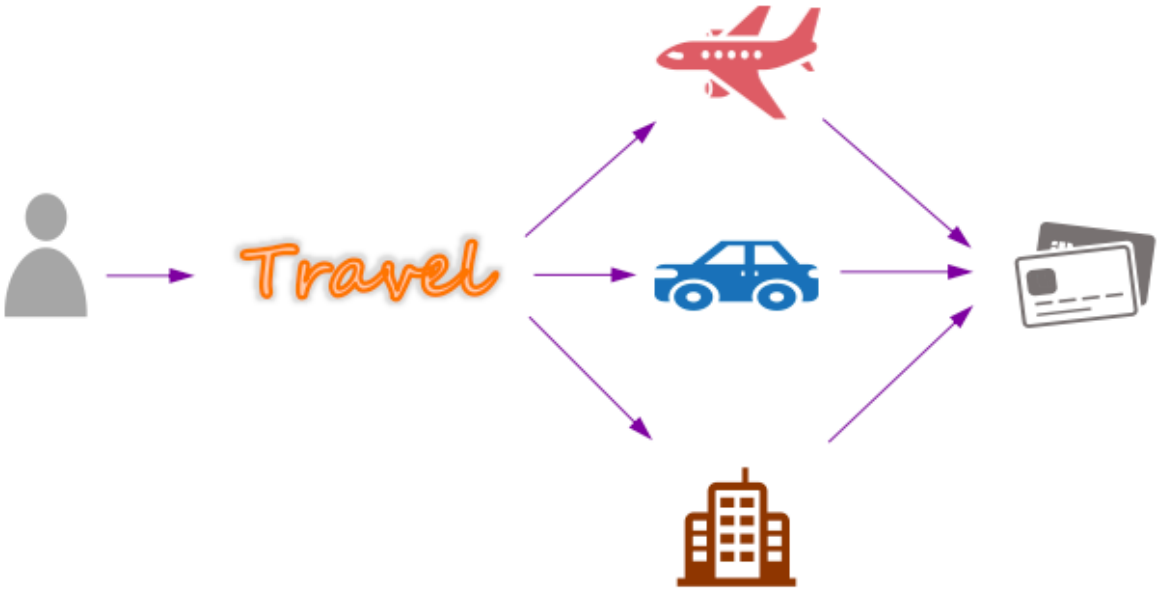


3.Sagas解决方案

幸运的是我们在互联网找到一篇精彩的论文，文中提出的数据一致性解决方案Saga恰好满足我们的业务要求。

> Saga 是一个长活事务，可被分解成可以交错运行的子事务集合。其中每个子事务都是一个保持数据库一致性的真实事务。

在我们的业务场景下，一个行程规划的事务就是一个 Saga，其中包含四个子事务：机票预订、租车、酒店预订、和支付。



Chris Richardson 在他的文章 Pattern: Saga 中对 Saga 有所描述。 Caitie McCaffrey 也在她的演讲中提到如何在微软的 光晕4 游戏中如何应用 saga 解决数据一致性问题。

Saga 的运行原理

> Saga 中的事务相互关联，应作为（非原子）单位执行。任何未完全执行的 Saga 是不满足要求的，如果发生，必须得到补偿。要修正未完全执行的部分，每个 saga 子交易 T1 应提供对应补偿事务 C1

我们根据上述规则定义以下事务及其相应的事务补偿：

服务	事务	补偿
机票预订	预订机票	取消预订
租车	租车	取消预订
酒店预订	预订房间	取消预订
支付	支付	退款

```
当每个 saga 子事务 T1, T2, …, Tn 都有对应的补偿定义 C1, C2, …, Cn-1, 那么 saga 系统可以保证子事务序列 T1, T2, …, Tn 得以完成（最佳情况）
或者序列 T1, T2, …, Tj, Cj, …, C2, C1, 0 < j < n, 得以完成
```

换句话说，通过上述定义的事务/补偿，saga 保证满足以下业务规则：

- 所有的预订都被执行成功，如果任何一个失败，都会被取消
- 如果最后一步付款失败，所有预订也将被取消

Saga 的恢复方式

原论文中描述了两种类型的 Saga 恢复方式：

> 向后恢复 补偿所有已完成的事务，如果任一子事务失败
> 向前恢复 重试失败的事务，假设每个子事务最终都会成功

显然，向前恢复没有必要提供补偿事务，如果你的业务中，子事务（最终）总会成功，或补偿事务难以定义或不可能，向前恢复更符合你的需求。

理论上补偿事务永不失败，然而，在分布式世界中，服务器可能会宕机，网络可能会失败，甚至数据中心也可能会停电。在这种情况下我们能做些什么？最后的手段是提供回退措施，比如人工干预。

使用 Saga 的条件

Saga 看起来很有希望满足我们的需求。所有长活事务都可以这样做吗？这里有一些限制：

- Saga 只允许两个层次的嵌套，顶级的 Saga 和简单子事务
- 在外层，全原子性不能得到满足。也就是说，sagas 可能会看到其他 sagas 的部分结果
- 每个子事务应该是独立的原子行为
- 在我们的业务场景下，航班预订、租车、酒店预订和付款是自然独立的行为，而且每个事务都可以用对应服务的数据库保证原子操作。

我们在行程规划事务层面也不需要原子性。一个用户可以预订最后一张机票，而后由于信用卡余额不足而被取消。同时另一个用户可能开始会看到已无余票，接着由于前者预订被取消，最后一张机票被释放，而抢到最后一个座位并完成行程规划。

补偿也有需考虑的事项：

- 补偿事务从语义角度撤消了事务 Ti 的行为，但未必能将数据库返回到执行 Ti 时的状态。（例如，如果事务触发导弹发射，则可能无法撤消此操作）

但这对我们的业务来说不是问题。其实难以撤消的行为也有可能被补偿。例如，

发送电邮的事务可以通过发送解释问题的另一封电邮来补偿。

现在我们有了通过 Saga 来解决数据一致性问题的方案。它允许我们成功地执行所有事务，或在任何事务失败的情况下，补偿已成功的事务。虽然 Saga 不提供 ACID 保证，但仍适用于许多数据最终一致性的场景。那我们如何设计一个 Saga 系统?

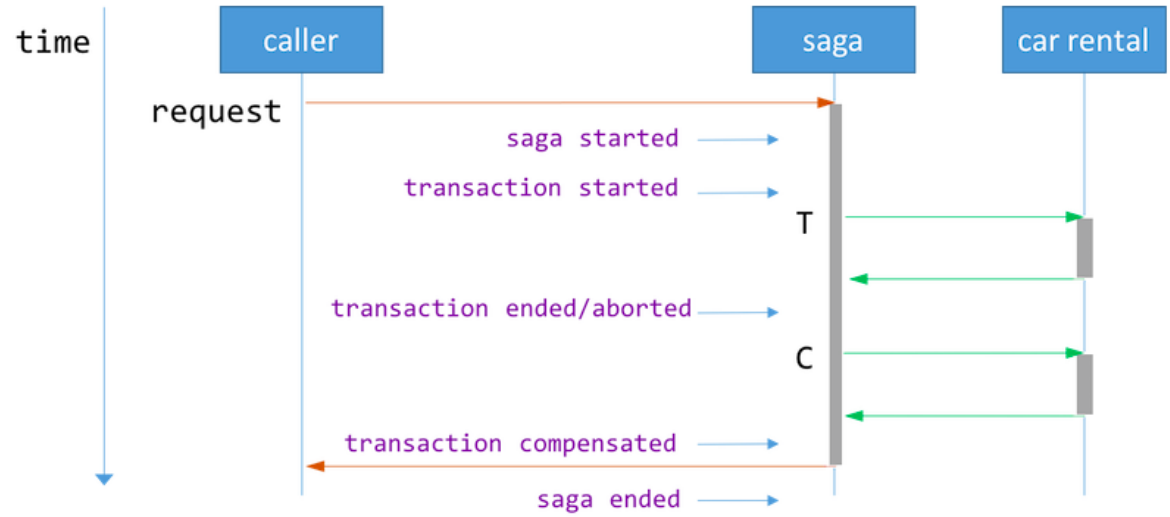
Saga Log

Saga 保证所有的子事务都得以完成或补偿，但 Saga 系统本身也可能会崩溃。Saga 崩溃时可能处于以下几个状态：

- Saga 收到事务请求，但尚未开始。因子事务对应的微服务状态未被 Saga 修改，我们什么也不需要做。
- 一些子事务已经完成。重启后，Saga 必须接着上次完成的事务恢复。
- 子事务已开始，但尚未完成。由于远程服务可能已完成事务，也可能事务失败，甚至服务请求超时，saga 只能重新发起之前未确认完成的子事务。这意味着子事务必须幂等。
- 子事务失败，其补偿事务尚未开始。Saga 必须在重启后执行对应补偿事务。
- 补偿事务已开始但尚未完成。解决方案与上一个相同。这意味着补偿事务也必须是幂等的。
- 所有子事务或补偿事务均已完成，与第一种情况相同。

为了恢复到上述状态，我们必须追踪子事务及补偿事务的每一步。我们决定通过事件的方式达到以上要求，并将以下事件保存在名为 saga log 的持久存储中：

- Saga started event 保存整个 saga 请求，其中包括多个事务/补偿请求
- Transaction started event 保存对应事务请求
- Transaction ended event 保存对应事务请求及其回复
- Transaction aborted event 保存对应事务请求和失败的原因
- Transaction compensated event 保存对应补偿请求及其回复
- Saga ended event 标志着 saga 事务请求的结束，不需要保存任何内容

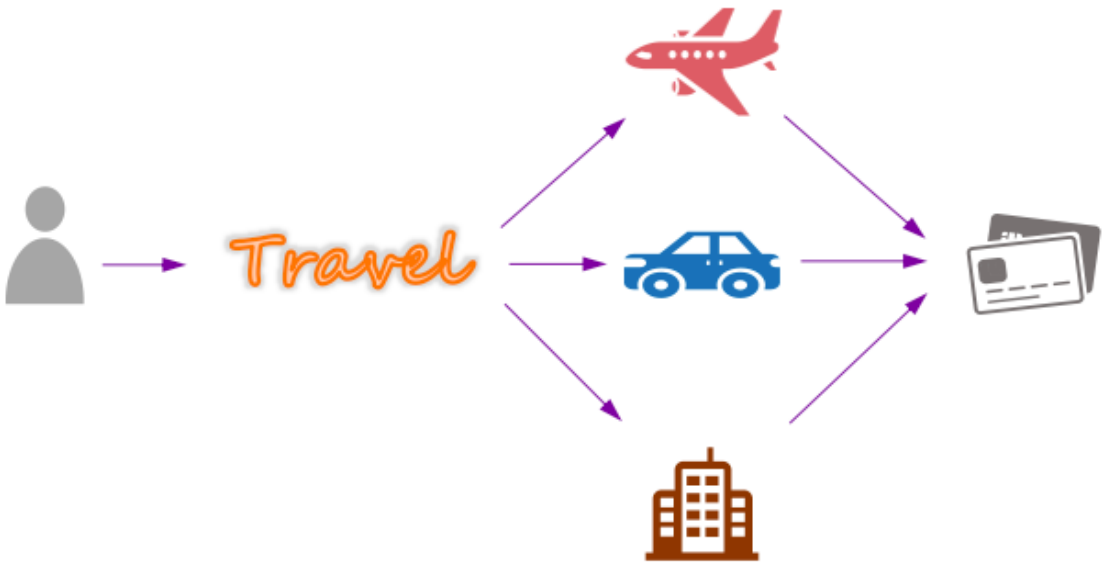


通过将这些事件持久化在 saga log 中，我们可以将 saga 恢复到上述任何状态。

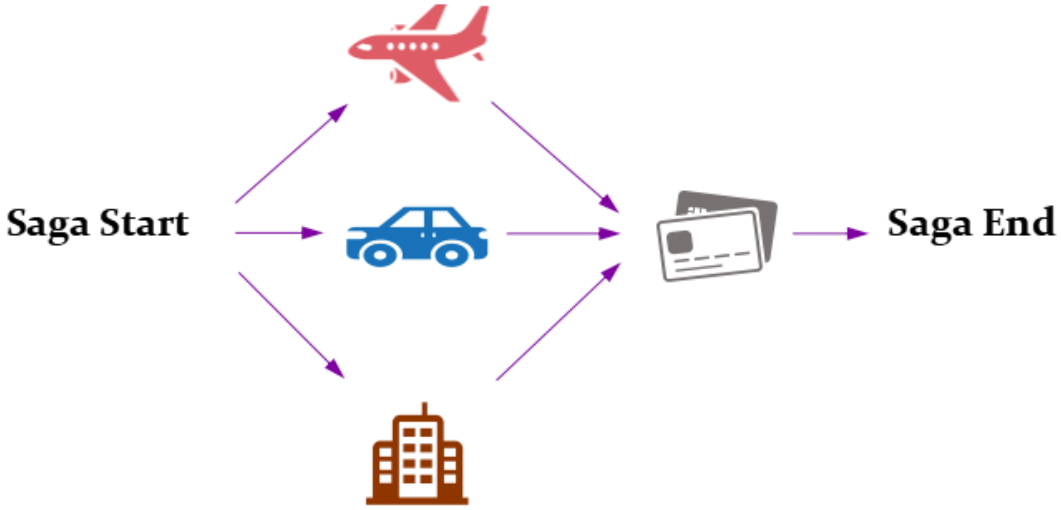
由于 Saga 只需要做事件的持久化，而事件内容以 JSON 的形式存储，Saga log 的实现非常灵活，数据库（SQL 或 NoSQL），持久消息队列，甚至普通文件可以用作事件存储，当然有些能更快得帮 saga 恢复状态。

Saga 请求的数据结构

在我们的业务场景下，航班预订、租车、和酒店预订没有依赖关系，可以并行处理，但对于我们的客户来说，只在所有预订成功后一次付费更加友好。那么这四个服务的事务关系可以用下图表示：



将行程规划请求的数据结构实现为有向非循环图恰好合适。图的根是 saga 启动任务，叶是 saga 结束任务。

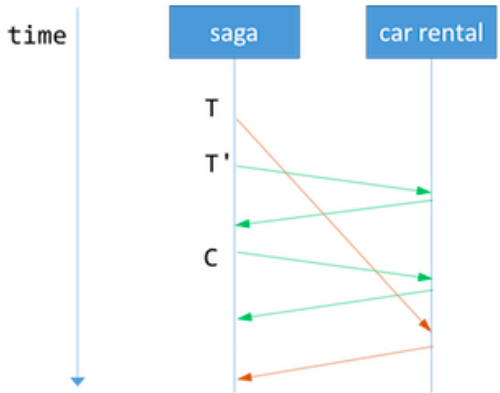


Parallel Saga

如上所述，航班预订，租车和酒店预订可以并行处理。但是这样做会造成另一个问题：如果航班预订失败，而租车正在处理怎么办？我们不能一直等待租车服务回应，因为不知道需要等多久。

最好的办法是再次发送租车请求，获得回应，以便我们能够继续补偿操作。但如果租车服务永不回应，我们可能需要采取回退措施，比如手动干预。

超时的预订请求可能最后仍被租车服务收到，这时服务已经处理了相同的预订和取消请求。



因此，服务的实现必须保证补偿请求执行以后，再次收到的对应事务请求无效。Caitie McCaffrey 在她的演讲 Distributed Sagas: A Protocol for Coordinating MicroServices 中把这个称为可交换的补偿请求 (commutative compensating request)。

ACID and Saga

ACID 是具有以下属性的一致性模型：

- 原子性（Atomicity）
- 一致性（Consistency）
- 隔离性（Isolation）
- 持久性（Durability）

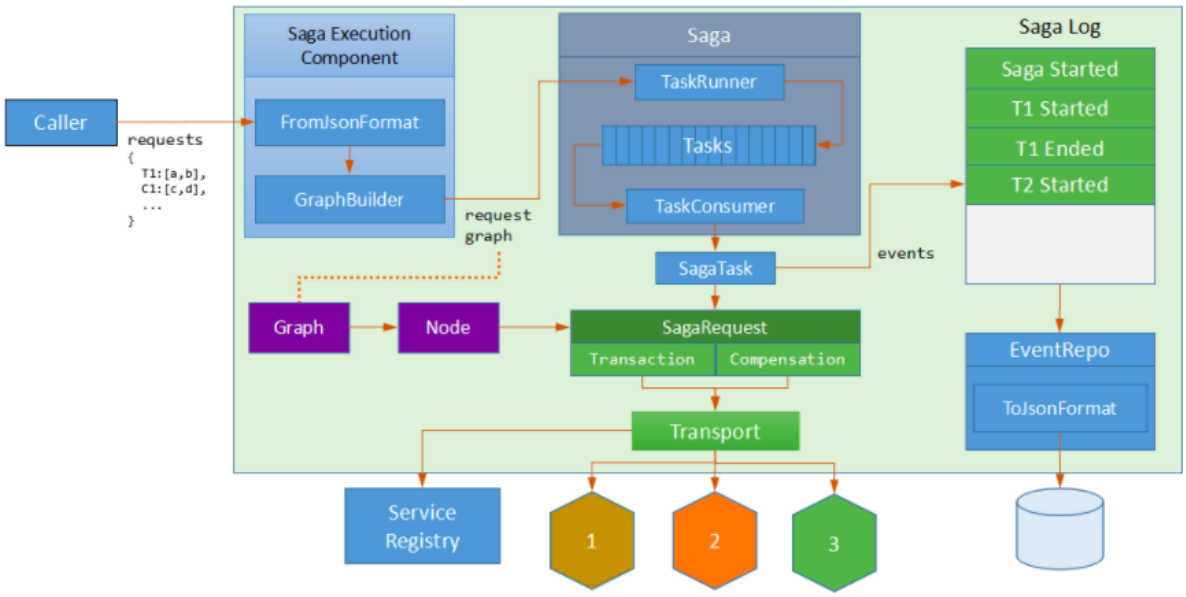
Saga 不提供 ACID 保证，因为原子性和隔离性不能得到满足。原论文描述如下：

> full atomicity is not provided. That is, sagas may view the partial results of other sagas

通过 saga log，saga 可以保证一致性和持久性。

Saga 架构

最后，我们的 Saga 架构如下：



- Saga Execution Component 解析请求 JSON 并构建请求图
- TaskRunner 用任务队列确保请求的执行顺序
- TaskConsumer 处理 Saga 任务，将事件写入 saga log，并将请求发送到远程服务

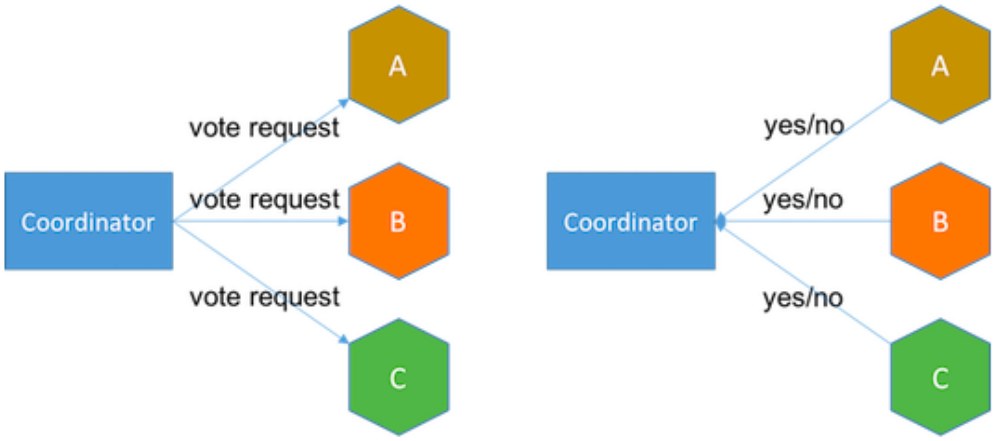
在上文中，谈到了 ServiceComb 下的 Saga 是怎么设计的。 然而，业界还有其他数据一致性解决方案，如 **两阶段提交（2PC）** 和 **Try-Confirm / Cancel（TCC）** 。那 saga 相比之下有什么特别?

4.两阶段提交 Two-Phase Commit (2PC)方案

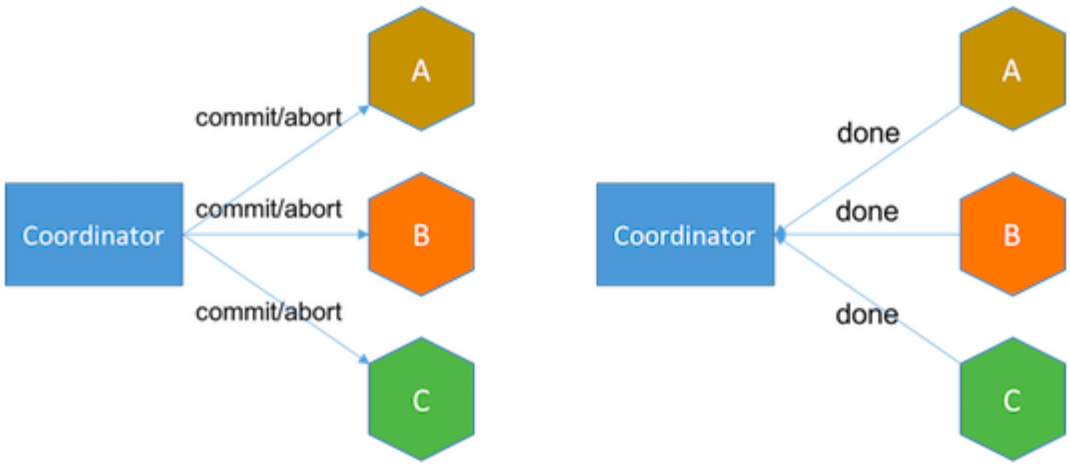
> 两阶段提交协议是一种分布式算法，用于协调参与分布式原子事务的所有进程，以保证他们均完成提交或中止（回滚）事务。

2PC 包含两个阶段：

- 投票阶段 协调器向所有服务发起投票请求，服务回答 yes 或 no。如果有任何服务回复 no 以拒绝或超时，协调器则在下一阶段发送中止消息。



- 决定阶段 如果所有服务都回复 yes，协调器则向服务发送 commit 消息，接着服务告知事务完成或失败。如果任何服务提交失败， 协调器将启动额外的步骤以中止该事务。



在投票阶段结束之后与决策阶段结束之前，服务处于不确定状态，因为他们不确定交易是否继续进行。当服务处于不确定状态并与协调器失去连接时，它只能选择等待协调器的恢复，或者咨询其他在确定状态下的服务来得知协调器的决定。在最坏的情况下，n 个处于不确定状态的服务向其他 n-1 个服务咨询将产生 O(n2) 个消息。

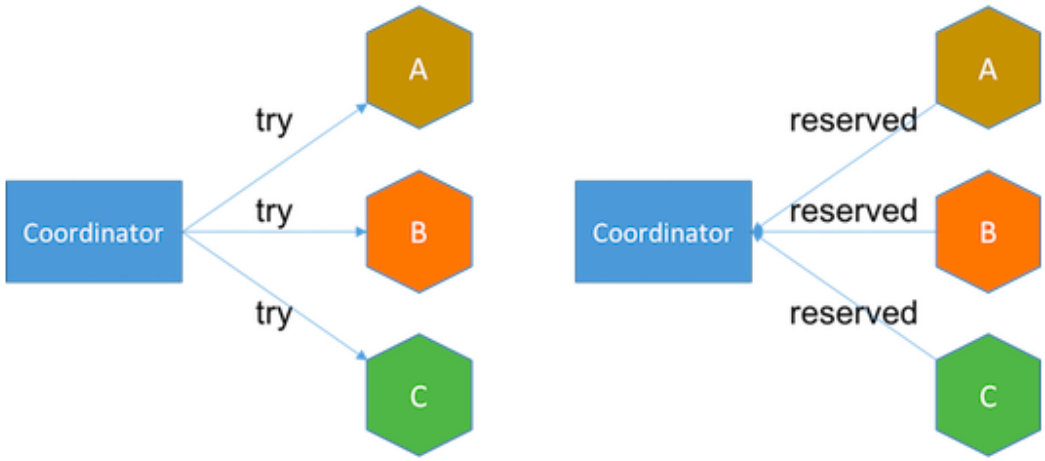
另外，2PC 是一个阻塞协议。服务在投票后需要等待协调器的决定，此时服务会

阻塞并锁定资源。由于其阻塞机制和最差时间复杂度高，2PC 不能适应随着事务涉及的服务数量增加而扩展的需要。

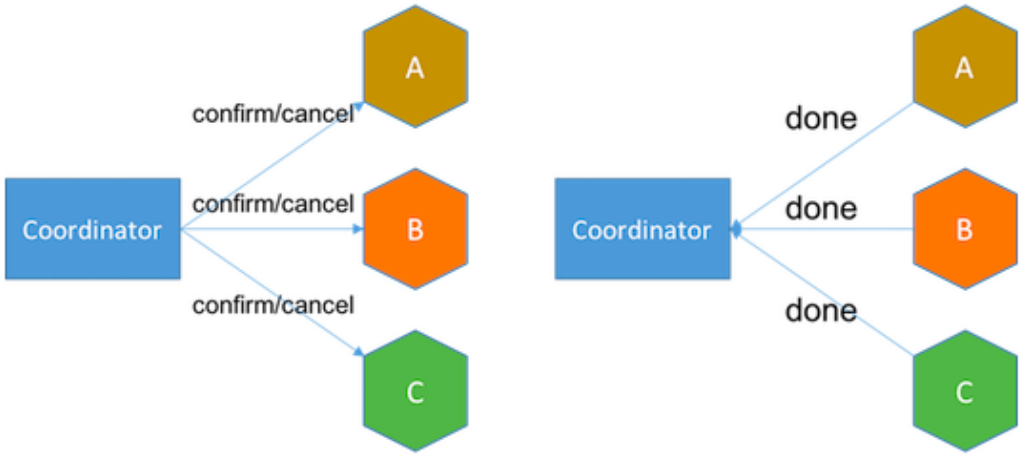
5.Try-Confirm/Cancel (TCC)解决方案

TCC 也是补偿型事务模式，支持两阶段的商业模型。

- 尝试阶段 将服务置于待处理状态。例如，收到尝试请求时，航班预订服务将为客户预留一个座位，并在数据库插入客户预订记录，将记录设为预留状态。如果任何服务失败或超时，协调器将在下一阶段发送取消请求。



- 确认阶段 将服务设为确认状态。确认请求将确认客户预订的座位，这时服务已可向客户收取机票费用。数据库中的客户预订记录也会被更新为确认状态。如果任何服务无法确认或超时，协调器将重试确认请求直到成功，或在重试了一定次数后采取回退措施，比如人工干预。



与 saga 相比，TCC 的优势在于，尝试阶段将服务转为待处理状态而不是最终状态，这使得设计相应的取消操作轻而易举。

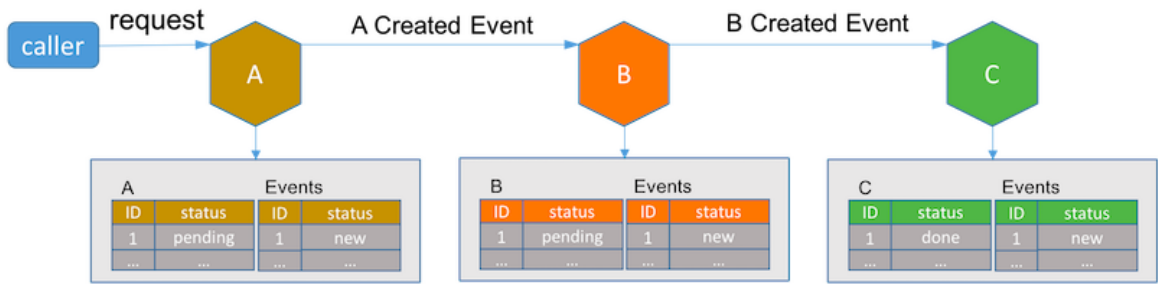
例如，电邮服务的尝试请求可将邮件标记为准备发送，并且仅在确认后发送邮件，其相应的取消请求只需将邮件标记为已废弃。但如果使用 saga，事务将发送电子邮件，及其相应的补偿事务可能需要发送另一封电子邮件作出解释。

TCC 的缺点是其两阶段协议需要设计额外的服务待处理状态，以及额外的接口来处理尝试请求。另外，TCC 处理事务请求所花费的时间可能是 saga 的两倍，因为 TCC 需要与每个服务进行两次通信，并且其确认阶段只能在收到所有服务对尝试请求的响应后开始。

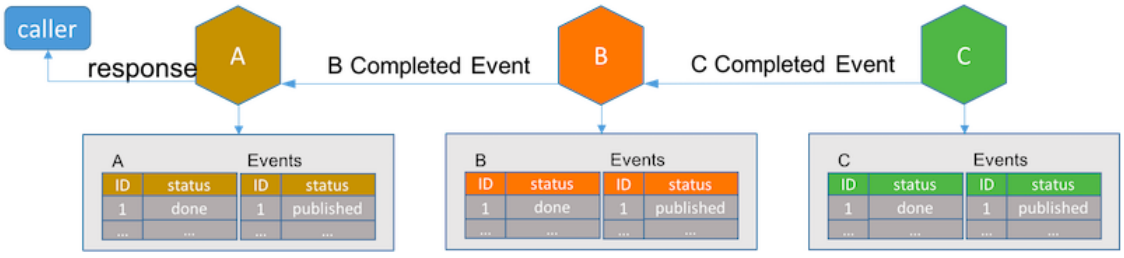
6.事件驱动的架构

和 TCC 一样，在事件驱动的架构中，长活事务涉及的每个服务都需要支持额外的待处理状态。接收到事务请求的服务会在其数据库中插入一条新的记录，将该记录状态设为待处理并发送一个新的事件给事务序列中的下一个服务。

因为在插入记录后服务可能崩溃，我们无法确定是否新事件已发送，所以每个服务还需要额外的事件表来跟踪当前长活事务处于哪一步。

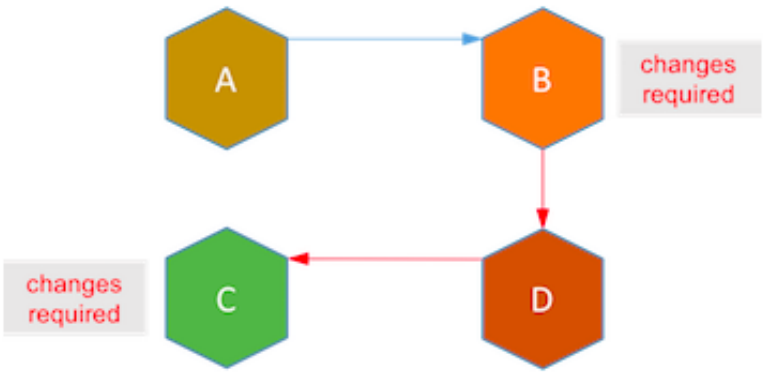


一旦长活事务中的最后一个服务完成其子事务，它将通知它在事务中的前一个服务。接收到完成事件的服务将其在数据库中的记录状态设为完成。



如果仔细比较，事件驱动的架构就像非集中式的基于事件的 TCC 实现。如果去掉待处理状态而直接把服务记录设为最终状态，这个架构就像非集中式的基于事件的 saga 实现。去中心化能达到服务自治，但也造成了服务之间更紧密的的耦合。假设新的业务需求在服务 B 和 C 之间的增加了新的流程 D。在事件驱动架

构下，服务 B 和 C 必须改动代码以适应新的流程 D。



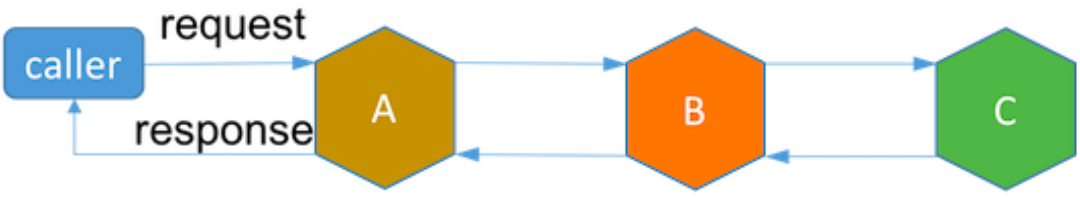
Saga 则正好相反，所有这些耦合都在 saga 系统中，当在长活事务中添加新流程时，现有服务不需要任何改动。

7.集中式与非集中式实现

这个 Saga 系列的文章讨论的都是集中式的 saga 设计。但 saga 也可用非集中式的方案来实现。那么非集中式的版本有什么不同?

非集中式 saga 没有专职的协调器。启动下一个服务调用的服务就是当前的协调器。例如：

- 服务 A 收到要求服务 A，B 和 C 之间的数据一致性的事务请求。
- A 完成其子事务，并将请求传递给事务中的下一个服务，服务 B.
- B 完成其子事务，并将请求传递给 C，依此类推。
- 如果 C 处理请求失败，B 有责任启动补偿事务，并要求 A 回滚。



与集中式相比，非集中式的实现具有服务自治的优势。但每个服务都需要包含数据一致性协议，并提供其所需的额外持久化设施。

我们更倾向于自治的业务服务，但服务还关联很多应用的复杂性，如数据一致性，服务监控和消息传递，将这些棘手问题集中处理，能将业务服务从应用的复杂性中释放，专注于处理复杂的业务，因此我们采用了集中式的 saga 设计。

另外，随着长活事务中涉及的服务数量增长，服务之间的关系变得越来越难理解。

8.总结

本文将 saga 与其他数据一致性解决方案进行了比较。Saga 比两阶段提交更易扩展。在事务可补偿的情况下，相比 TCC，saga 对业务逻辑几乎没有改动的需要，而且性能更高。集中式的 saga 设计解耦了服务与数据一致性逻辑及其持久化设施，并使排查事务中的问题更容易。

二.分布式锁的场景与实现

1.使用场景

首先，我们看这样一个场景：客户下单的时候，我们调用库存中心进行减库存，那我们一般的操作都是：

```
update store set num = $num where id = $id
```

这种通过设置库存的修改方式，我们知道在并发量高的时候会存在数据库的丢失更新，比如 a, b 当前两个事务，查询出来的库存都是 5，a 买了 3 个单子要把库存设置为 2，而 b 买了 1 个单子要把库存设置为 4，那这个时候就会出现 a 会覆盖 b 的更新，所以我们更多的都是会加个条件：

```
update store set num = $num where id = $id and num = $query_num
```

即乐观锁的方式来处理，当然也可以通过版本号来处理乐观锁，都是一样的，但是这是更新一个表，如果我们牵扯到多个表呢，我们希望和这个单子关联的所有的表同一时间只能被一个线程来处理更新，多个线程按照不同的顺序去更新同一个单子关联的不同数据，出现死锁的概率比较大。对于非敏感的数据，我们也没有必要去都加乐观锁处理，我们的服务都是多机器部署的，要保证多进程多线程同时只能有一个进程的一个线程去处理，这个时候我们就需要用到分布式锁。分布式锁的实现方式有很多，我们今天分别通过数据库，Zookeeper, Redis 以及 Tair 的实现逻辑

2.数据库实现

加 xx 锁

更新一个单子关联的所有的数据，先查询出这个单子，并加上排他锁，在进行一系列的更新操作

```
begin transaction;
select ...for update;
doSomething();
commit();
```

这种处理主要依靠排他锁来阻塞其他线程，不过这个需要注意几点：

1. 查询的数据一定要在数据库里存在，如果不存在的话，数据库会加 gap 锁，而 gap 锁之间是兼容的，这种如果两个线程都加了gap 锁，另一个再更新的话会出现死锁。不过一般能更新的数据都是存在的
2. 后续的处理流程需要尽可能的时间短，即在更新的时候提前准备好数据，保证事务处理的时间足够的短，流程足够的短，因为开启事务是一直占着连接的，如果流程比较长会消耗过多的数据库连接的

唯一键

通过在一张表里创建唯一键来获取锁，比如执行 saveStore 这个方法

```
insert table lock_store ('method_name') values($method_name)
```

其中 `method_name` 是个唯一键，通过这种方式也可以做到，解锁的时候直接删除改行记录就行。不过这种方式，锁就不会是阻塞式的，因为插入数据是立马可以得到返回结果的。

那针对以上数据库实现的两种分布式锁，存在什么样的优缺点呢?

优点

简单，方便，快速实现

缺点

- 基于数据库，开销比较大，性能可能会存在影响
- 基于数据库的当前读来实现，数据库会在底层做优化，可能用到索引，可能不用到索引，这个依赖于查询计划的分析

3.Zookeeper 实现

获取锁

1. 先有一个锁跟节点，lockRootNode，这可以是一个永久的节点
2. 客户端获取锁，先在 lockRootNode 下创建一个顺序的瞬时节点，保证客户端断开连接，节点也自动删除
3. 调用 lockRootNode 父节点的 getChildren() 方法，获取所有的节点，并从小到大排序，如果创建的最小的节点是当前节点，则返回 true,获取锁成功，否则，关注比自己序号小的节点的释放动作(exist watch)，这样可以保证每一个客户端只需要关注一个节点，不需要关注所有的节点，避免羊群效应。
4. 如果有节点释放操作，重复步骤 3

释放锁

只需要删除步骤 2 中创建的节点即可

使用 Zookeeper 的分布式锁存在什么样的优缺点呢?

优点

- 客户端如果出现宕机故障的话，锁可以马上释放
- 可以实现阻塞式锁，通过 watcher 监听，实现起来也比较简单
- 集群模式，稳定性比较高

缺点

- 一旦网络有任何的抖动，Zookeeper 就会认为客户端已经宕机，就会断掉连接，其他客户端就可以获取到锁。当然 Zookeeper 有重试机制，这个就比较依赖于其重试机制的策略了
- 性能上不如缓存

4.Redis 实现

我们先举个例子，比如现在我要更新产品的信息，产品的唯一键就是 productId

简单实现 1

```
public boolean lock(String key, V v, int expireTime){
        int retry = 0;
```

```
        //获取锁失败最多尝试10次
        while (retry < failRetryTimes){
            //获取锁
            Boolean result = redis.setNx(key, v, expireTime);
            if (result){
                return true;
            }

            try {
                //获取锁失败间隔一段时间重试
                TimeUnit.MILLISECONDS.sleep(sleepInterval);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                return false;
            }

        }

        return false;
    }
    public boolean unlock(String key){
        return redis.delete(key);
    }
    public static void main(String[] args) {
        Integer productId = 324324;
        RedisLock<Integer> redisLock = new RedisLock<Integer>();
        redisLock.lock(productId+"", productId, 1000);
    }
}
```

这是一个简单的实现，存在的问题：

1. 可能会导致当前线程的锁误被其他线程释放，比如 a 线程获取到了锁正在执行，但是由于内部流程处理超时或者 gc 导致锁过期，这个时候b线程获取到了锁，a 和 b 线程处理的是同一个 productId，b还在处理的过程中，这个时候 a 处理完了，a 去释放锁，可能就会导致 a 把 b 获取的锁释放了。
2. 不能实现可重入
3. 客户端如果第一次已经设置成功，但是由于超时返回失败，此后客户端尝试会一直失败

针对以上问题我们改进下：

1. v 传 requestId，然后我们在释放锁的时候判断一下，如果是当前 requestId，那就可以释放，否则不允许释放

2. 加入 count 的锁计数，在获取锁的时候查询一次，如果是当前线程已经持有的锁，那锁技术加 1，直接返回 true

简单实现 2

```
private static volatile int count = 0;
public boolean lock(String key, V v, int expireTime){
    int retry = 0;
    //获取锁失败最多尝试10次
    while (retry < failRetryTimes){
        //1.先获取锁,如果是当前线程已经持有，则直接返回
        //2.防止后面设置锁超时，其实是设置成功，而网络超时导致客户端返回失败，所以获取锁之前需要查询一下
        V value = redis.get(key);
        //如果当前锁存在，并且属于当前线程持有，则锁计数+1，直接返回
        if (null != value && value.equals(v)){
            count ++;
            return true;
        }

        //如果锁已经被持有了，那需要等待锁的释放
        if (value == null || count <= 0){
            //获取锁
            Boolean result = redis.setNx(key, v, expireTime);
            if (result){
                count = 1;
                return true;
            }
        }

        try {
            //获取锁失败间隔一段时间重试
            TimeUnit.MILLISECONDS.sleep(sleepInterval);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            return false;
        }

    }

    return false;
}
public boolean unlock(String key, String requestId){
    String value = redis.get(key);
    if (Strings.isNullOrEmpty(value)){
        count = 0;
```

```
            return true;
        }
        //判断当前锁的持有者是否是当前线程，如果是的话释放锁，不是的话返回false
        if (value.equals(requestId)){
            if (count > 1){
                count -- ;
                return true;
            }

            boolean delete = redis.delete(key);
            if (delete){
                count = 0;
            }
            return delete;
        }

        return false;
    }
    public static void main(String[] args) {
        Integer productId = 324324;
        RedisLock<String> redisLock = new RedisLock<String>();
        String requestId = UUID.randomUUID().toString();
        redisLock.lock(productId+"", requestId, 1000);
    }
```

这种实现基本解决了误释放和可重入的问题，这里说明几点：

1. 引入 count 实现重入的话，看业务需要，并且在释放锁的时候，其实也可以直接就把锁删除了，一次释放搞定，不需要在通过 count 数量释放多次，看业务需要吧
2. 关于要考虑设置锁超时，所以需要在设置锁的时候查询一次，可能会有性能的考量，看具体业务吧
3. 目前获取锁失败的等待时间是在代码里面设置的，可以提出来，修改下等待的逻辑即可

错误实现

获取到锁之后要检查下锁的过期时间，如果锁过期了要重新设置下时间,大致代码如下：

```
public boolean tryLock2(String key, int expireTime){
    long expires = System.currentTimeMillis() + expireTime;
```

```
        // 获取锁
        Boolean result = redis.setNx(key, expires, expireTime);
        if (result){
            return true;
        }

        V value = redis.get(key);
        if (value != null && (Long)value < System.currentTimeMillis()){
            // 锁已经过期
            String oldValue = redis.getSet(key, expireTime);
            if (oldValue != null && oldValue.equals(value)){
                return true;
            }
        }

        return false;
    }
```

这种实现存在的问题，过度依赖当前服务器的时间了，如果在大量的并发请求下，都判断出了锁过期，而这个时候再去设置锁的时候，最终是会只有一个线程，但是可能会导致不同服务器根据自身不同的时间覆盖掉最终获取锁的那个线程设置的时间。

5.Tair 实现

通过 Tair 来实现分布式锁和 Redis 的实现核心差不多，不过 Tair 有个很方便的 api，感觉是实现分布式锁的最佳配置，就是 Put api 调用的时候需要传入一个 version，就和数据库的乐观锁一样，修改数据之后，版本会自动累加，如果传入的版本和当前数据版本不一致，就不允许修改。

三.分布式事务

1.分布式一致性

在分布式系统中，为了保证数据的高可用，通常，我们会将数据保留多个副本(replica)，这些副本会放置在不同的物理的机器上。为了对用户提供正确的 CRUD 等语义，我们需要保证这些放置在不同物理机器上的副本是一致的。

为了解决这种分布式一致性问题，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（Two Phase**

Commitment Protocol）、三阶提交协议（Three Phase Commitment Protocol）和 Paxos 算法

2.分布式事务

> 分布式事务是指会涉及到操作多个数据库的事务.其实就是将对同一库事务的概念扩大到了对多个库的事务。目的是为了保证分布式系统中的数据一致性。分布式事务处理的关键是必须有一种方法可以知道事务在任何地方所做的所有动作，提交或回滚事务的决定必须产生统一的结果（全部提交或全部回滚）

在分布式系统中，各个节点之间在物理上相互独立，通过网络进行沟通和协调。由于存在事务机制，可以保证每个独立节点上的数据操作可以满足 ACID。但是，相互独立的节点之间无法准确的知道其他节点中的事务执行情况。所以从理论上讲，两台机器理论上无法达到一致的状态。如果想让分布式部署的多台机器中的数据保持一致性，那么就要保证在所有节点的数据写操作，要不全部都执行，要么全部的都不执行。但是，一台机器在执行本地事务的时候无法知道其他机器中的本地事务的执行结果。所以他也就不知道本次事务到底应该 commit 还是 rollback。所以，常规的解决办法就是引入一个“协调者”的组件来统一调度所有分布式节点的执行。

3.XA 规范

X/Open 组织（即现在的 Open Group ）定义了分布式事务处理模型。 X/Open DTP 模型（ 1994 ）包括应用程序（ AP ）、事务管理器（ TM ）、资源管理器（ RM ）、通信资源管理器（ CRM ）四部分。一般，常见的事务管理器（ TM ）是交易中间件，常见的资源管理器（ RM ）是数据库，常见的通信资源管理器（ CRM ）是消息中间件。 通常把一个数据库内部的事务处理，如对多个表的操作，作为本地事务看待。数据库的事务处理对象是本地事务，而分布式事务处理的对象是全局事务。 所谓全局事务，是指分布式事务处理环境中，多个数据库可能需要共同完成一个工作，这个工作即是一个全局事务，例如，一个事务中可能更新几个不同的数据库。对数据库的操作发生在系统的各处但必须全部被提交或回滚。此时一个数据库对自己内部所做操作的提交不仅依赖本身操作是否成功，还要依赖与全局事务相关的其它数据库的操作是否成功，如果任一数据库的任一操作失败，则参与此事务的所有数据库所做的所有操作都必须回滚。 一般情况下，某一数据库无法知道其它数据库在做什么，因此，在一个 DTP 环境中，交易中间件是必需的，由它通知和协调相关数据库的提交或回滚。而一个数据库只将其自己所做的操作（可恢复）影射到全局事务中。

> XA 就是 X/Open DTP 定义的交易中间件与数据库之间的接口规范（即接口函数），交易中间件用它来通知数据库事务的开始、结束以及提交、回滚等。 XA 接口函数由数据库厂商提供。

二阶提交协议和三阶提交协议就是根据这一思想衍生出来的。可以说二阶段提交其实就是实现 XA 分布式事务的关键(确切地说：两阶段提交主要保证了分布式事务的原子性：即所有结点要么全做要么全不做)

4.2PC

> 二阶段提交(Two-phaseCommit)是指，在计算机网络以及数据库领域内，为了使基于分布式系统架构下的所有节点在进行事务提交时保持一致性而设计的一种算法(Algorithm)。通常，二阶段提交也被称为是一种协议(Protocol))。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节点(称作参与者)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(比如将更新后的数据写入磁盘等等)。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）**。

准备阶段

事务协调者(事务管理器)给每个参与者(资源管理器)发送 Prepare 消息，每个参与者要么直接返回失败(如权限验证失败)，要么在本地执行事务，写本地的 redo 和 undo 日志，但不提交，到达一种“万事俱备，只欠东风”的状态。

可以进一步将准备阶段分为以下三个步骤：

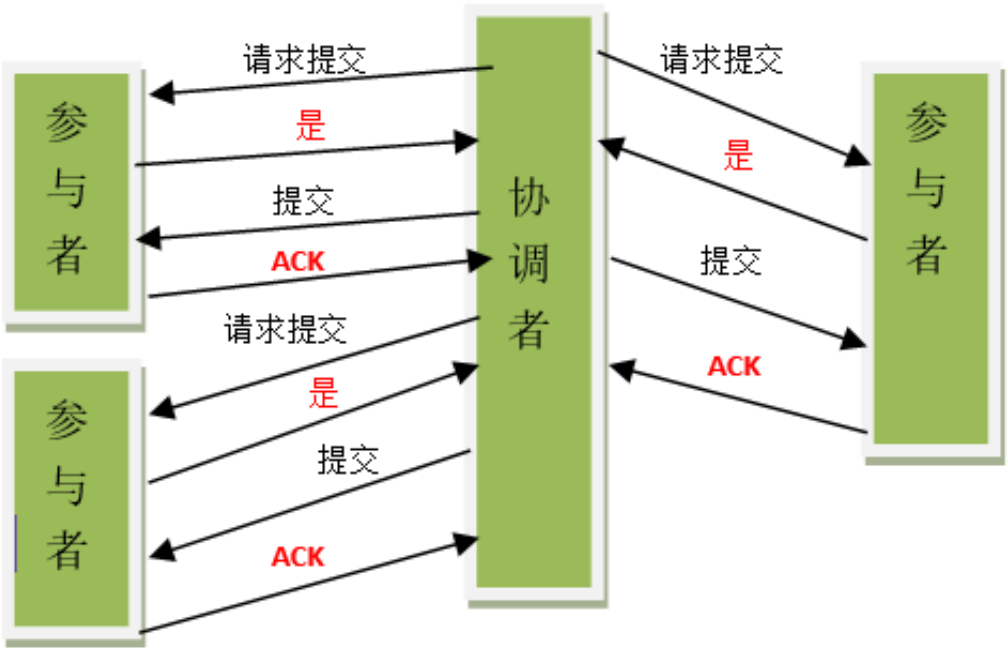
1. 协调者节点向所有参与者节点询问是否可以执行提交操作(vote)，并开始等待各参与者节点的响应。
2. 参与者节点执行询问发起为止的所有事务操作，并将 Undo 信息和 Redo 信息写入日志。（注意：若成功这里其实每个参与者已经执行了事务操作）
3. 各参与者节点响应协调者节点发起的询问。如果参与者节点的事务操作实际执行成功，则它返回一个”同意”消息；如果参与者节点的事务操作实际执行失败，则它返回一个”中止”消息。

提交阶段

如果协调者收到了参与者的失败消息或者超时，直接给每个参与者发送回滚(Rollback)消息；否则，发送提交(Commit)消息；参与者根据协调者的指令执行提交或者回滚操作，释放所有事务处理过程中使用的锁资源。(注意:必须在最后阶段释放锁资源)

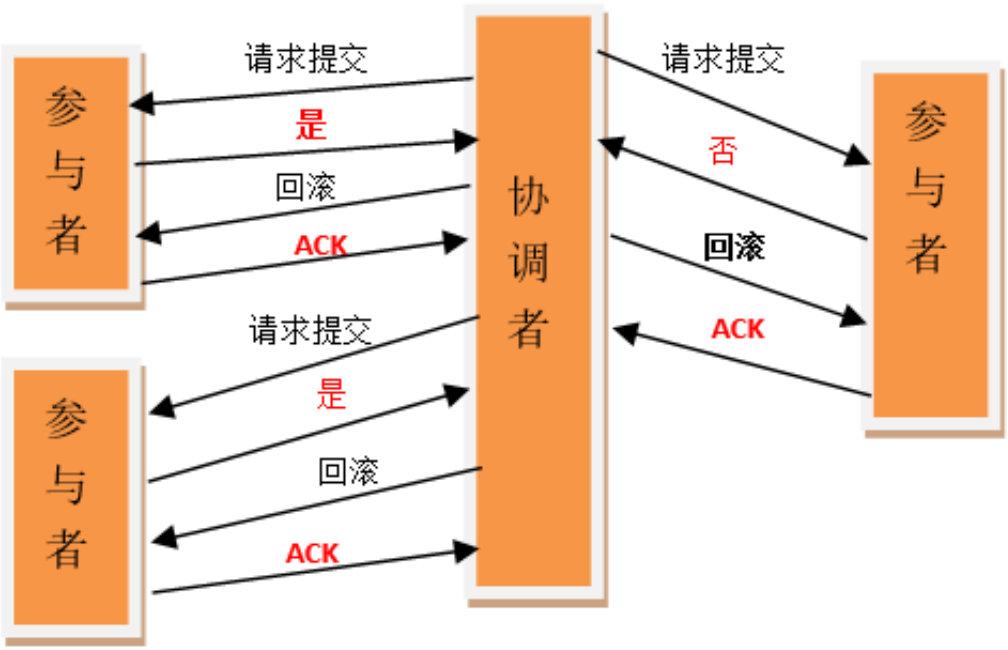
接下来分两种情况分别讨论提交阶段的过程。

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(commit)”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. 协调者节点受到所有参与者节点反馈的”完成”消息后，完成事务。

如果任一参与者节点在第一阶段返回的响应消息为”中止”，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(rollback)”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. 协调者节点受到所有参与者节点反馈的”回滚完成”消息后，取消事务。

不管最后结果如何，第二阶段都会结束当前事务。

二阶段提交看起来确实能够提供原子性的操作，但是不幸的事，二阶段提交还是有几个**缺点**的：

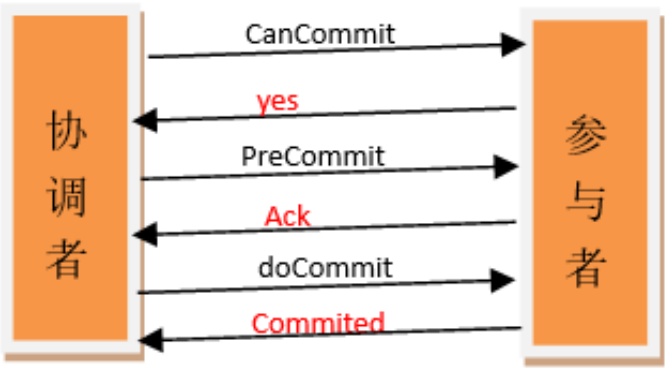
1. **同步阻塞问题**：执行过程中，所有参与节点都是事务阻塞型的。当参与者占有公共资源时，其他第三方节点访问公共资源不得不处于阻塞状态。
2. **单点故障**：由于协调者的重要性，一旦协调者发生故障。参与者会一直阻塞下去。尤其在第二阶段，协调者发生故障，那么所有的参与者还都处于锁定事务资源的状态中，而无法继续完成事务操作。（如果是协调者挂掉，可以重新选举一个协调者，但是无法解决因为协调者宕机导致的参与者处于阻塞状态的问题）
3. **数据不一致**：在二阶段提交的阶段二中，当协调者向参与者发送 commit 请求之后，发生了局部网络异常或者在发送 commit 请求过程中协调者发生了故障，这回导致只有一部分参与者接受到了 commit 请求。而在这部分参与者接到 commit 请求之后就会执行 commit 操作。但是其他部分未接到 commit 请求的机器则无法执行事务提交。于是整个分布式系统便出现了数据部一致性的现象。
4. 二阶段无法解决的问题：协调者再发出 commit 消息之后宕机，而唯一接收

到这条消息的参与者同时也宕机了。那么即使协调者通过选举协议产生了新的协调者，这条事务的状态也是不确定的，没人知道事务是否被已经提交。

由于二阶段提交存在着诸如同步阻塞、单点问题、脑裂等缺陷，所以，研究者们在二阶段提交的基础上做了改进，提出了三阶段提交。

5.3PC

> 三阶段提交（Three-phase commit），也叫三阶段提交协议（Three-phase commit protocol），是二阶段提交（2PC）的改进版本。



与两阶段提交不同的是，三阶段提交有两个改动点。

1. 引入超时机制。同时在协调者和参与者中都引入超时机制。
2. 在第一阶段和第二阶段中插入一个准备阶段。保证了在最后提交阶段之前各参与节点的状态是一致的。

也就是说，除了引入超时机制之外，3PC 把 2PC 的准备阶段再次一分为二，这样三阶段提交就有 `CanCommit`、`PreCommit`、`DoCommit` 三个阶段。

CanCommit 阶段

3PC 的 CanCommit 阶段其实和 2PC 的准备阶段很像。协调者向参与者发送 commit 请求，参与者如果可以提交就返回 Yes 响应，否则返回 No 响应。

1. **事务询问**：协调者向参与者发送CanCommit请求。询问是否可以执行事务提交操作。然后开始等待参与者的响应。
2. **响应反馈**：参与者接到CanCommit请求之后，正常情况下，如果其自身认为可以顺利执行事务，则返回Yes响应，并进入预备状态。否则反馈No

PreCommit 阶段

协调者根据参与者的反应情况来决定是否可以记性事务的 PreCommit 操作。根据响应情况，有以下两种可能。

1. 假如协调者从所有的参与者获得的反馈都是 Yes 响应，那么就会执行事务的预执行。

- 发送预提交请求：协调者向参与者发送 PreCommit 请求，并进入 Prepared 阶段。
- 事务预提交：参与者接收到 PreCommit 请求后，会执行事务操作，并将 undo 和 redo 信息记录到事务日志中。
- 响应反馈：如果参与者成功的执行了事务操作，则返回 ACK 响应，同时开始等待最终指令。

2. 假如有任何一个参与者向协调者发送了 No 响应，或者等待超时之后，协调者都没有接到参与者的响应，那么就执行事务的中断。

- 发送中断请求：协调者向所有参与者发送 abort 请求。
- 中断事务：参与者收到来自协调者的 abort 请求之后（或超时之后，仍未收到协调者的请求），执行事务的中断。

doCommit 阶段

该阶段进行真正的事务提交，也可以分为以下两种情况。

1. 执行提交

- 发送提交请求：协调接收到参与者发送的 ACK 响应，那么他将从预提交状态进入到提交状态。并向所有参与者发送 doCommit 请求。
- 事务提交：参与者接收到 doCommit 请求之后，执行正式的事务提交。并在完成事务提交之后释放所有事务资源。
- 响应反馈：事务提交完之后，向协调者发送 ACK 响应。
- 完成事务：协调者接收到所有参与者的 ACK 响应之后，完成事务。

2. 中断事务

协调者没有接收到参与者发送的 ACK 响应（可能是接受者发送的不是 ACK 响应，也可能响应超时），那么就会执行中断事务。

- 发送中断请求：协调者向所有参与者发送 abort 请求
- 事务回滚：参与者接收到 abort 请求之后，利用其在阶段二记录的 undo 信息来执行事务的回滚操作，并在完成回滚之后释放所有的事务资源。

- 反馈结果：参与者完成事务回滚之后，向协调者发送 ACK 消息
- 中断事务：协调者接收到参与者反馈的ACK消息之后，执行事务的中断。

> 在 `doCommit` 阶段，如果参与者无法及时接收到来自协调者的 `doCommit` 或者 `abort` 请求时，会在等待超时之后，会继续进行事务的提交。（其实这个应该是基于概率来决定的，当进入第三阶段时，说明参与者在第二阶段已经收到了 `PreCommit` 请求，那么协调者产生 `PreCommit` 请求的前提条件是他在第二阶段开始之前，收到所有参与者的 `CanCommit` 响应都是 `Yes`。（一旦参与者收到了 `PreCommit`，意味他知道大家其实都同意修改了）所以，一句话概括就是，当进入第三阶段时，由于网络超时等原因，虽然参与者没有收到 `commit` 或者 `abort` 响应，但是他有理由相信：成功提交的几率很大。）

6.2PC 与 3PC 的区别

相对于 2PC，3PC 主要解决的单点故障问题，并减少阻塞，因为一旦参与者无法及时收到来自协调者的信息之后，他会默认执行 commit。而不会一直持有事务资源并处于阻塞状态。但是这种机制也会导致数据一致性问题，因为，由于网络原因，协调者发送的 abort 响应没有及时被参与者接收到，那么参与者在等待超时之后执行了 commit 操作。这样就和其他接到 abort 命令并执行回滚的参与者之间存在数据不一致的情况。

四.session 分布式处理

1.Session 复制

在支持 Session 复制的 Web 服务器上，通过修改 Web 服务器的配置，可以实现将 Session 同步到其它 Web 服务器上，达到每个 Web 服务器上都保存一致的 Session。

- 优点：代码上不需要做支持和修改。
- 缺点：需要依赖支持的 Web 服务器，一旦更换成不支持的 Web 服务器就不能使用了，在数据量很大的情况下不仅占用网络资源，而且会导致延迟。
- 适用场景：只适用于Web服务器比较少且 Session 数据量少的情况。
- 可用方案：开源方案 tomcat-redis-session-manager，暂不支持 Tomcat8。

2.Session 粘滞

将用户的每次请求都通过某种方法强制分发到某一个 Web 服务器上，只要这个 Web 服务器上存储了对应 Session 数据，就可以实现会话跟踪。

- 优点：使用简单，没有额外开销。
- 缺点：一旦某个 Web 服务器重启或宕机，相对应的 Session 数据将会丢失，而且需要依赖负载均衡机制。
- 适用场景：对稳定性要求不是很高的业务情景。

3.Session 集中管理

在单独的服务器或服务器集群上使用缓存技术，如 Redis 存储 Session 数据，集中管理所有的 Session，所有的Web服务器都从这个存储介质中存取对应的 Session，实现 Session 共享。

- 优点：可靠性高，减少 Web 服务器的资源开销。
- 缺点：实现上有些复杂，配置较多。
- 适用场景：Web服务器较多、要求高可用性的情况。
- 可用方案：开源方案 Spring Session，也可以自己实现，主要是重写 HttpServletRequestWrapper 中的 getSession 方法。

4.基于 Cookie 管理

这种方式每次发起请求的时候都需要将 Session 数据放到 Cookie 中传递给服务端。

- 优点：不需要依赖额外外部存储，不需要额外配置。
- 缺点：不安全，易被盗取或篡改；Cookie 数量和长度有限制，需要消耗更多网络带宽。
- 适用场景：数据不重要、不敏感且数据量小的情况。

5.总结

这四种方式，相对来说，**Session 集中管理** 更加可靠，使用也是最多的。

五.Session 分布式方案

1.基于 nfs(net filesystem) 的 Session 共享

将共享服务器目录 mount 各服务器的本地 session 目录，session 读写受共享服务器 io 限制，**不能满足高并发**。

2.基于关系数据库的 Session 共享

这种方案普遍使用。使用关系数据库存储 session 数据，对于 mysql 数据库，建议使用 heap 引擎。这种方案性能取决于数据库的性能，在高并发下容易造成表锁（虽然可以采用行锁的存储引擎，性能会下降），并且需要自己实现 session 过期淘汰机制。

3.基于 Cookie 的 Session 共享

这种方案也在大型互联网中普遍使用，将用户的 session 加密序列化后以 cookie 的方式保存在网站根域名下（比如 taobao.com），当用户访问所有二级域名站点式，浏览器会传递所有匹配的根域名的 cookie 信息，这样实现了用户 cookie 化 session 的多服务共享。此方案能够节省大量服务器资源，缺点是存储的信息长度受到 http 协议限制；cookie 的信息还需要做加密解密；请求任何资源时都会将 cookie 附加到 http 头上传到服务器，占用了一定带宽。

4.基于 Web 容器的 Session 机制

利用容器机制，通过配置即可实现。

5.基于 Zookeeper 的分布式 Session 存储

6.基于 Redis/Memcached 的 Session 共享存储

这些 key/value 非关系存储有较高的性能，轻松达到 2000 左右的 qps，内置的过期机制正好满足 session 的自动实效特性。