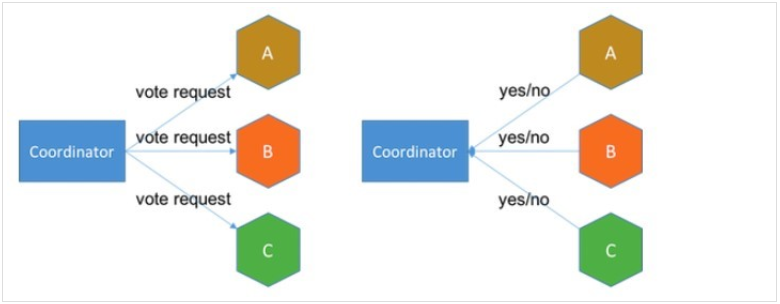
1. **CAP定理：一个Web应用至多只能同时支持下面的两个属性**
2. 一致性（Consistency）：客户端知道一系列的操作都会同时发生（生效）。
3. 可用性（Availability）：每个操作都必须以可预期的响应结束。
4. 分区容错性（Partition tolerance）：即使出现单个组件无法可用，操作依然可以完成。
5. **BASE理论：**
6. Basically Available（基本可用）
7. Soft state（软状态）
8. Eventually consistent（最终一致性）

在分布式系统中，往往追求的是可用性，它的重要程度比一致性高。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，采用适当的方式使系统达到最终一致性**。

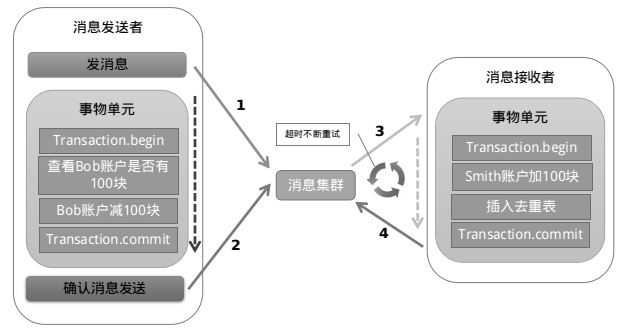
1. **分布式事务的几种解决方案**
2. 两阶段提交（2PC，Two-phase Commit）：通过引入协调者（Coordinator）来协调参与者的行为，并最终决定这些参与者是否要真正执行事务。
3. 准备阶段：协调者询问参与者事务是否执行成功，参与者发回事务执行结果



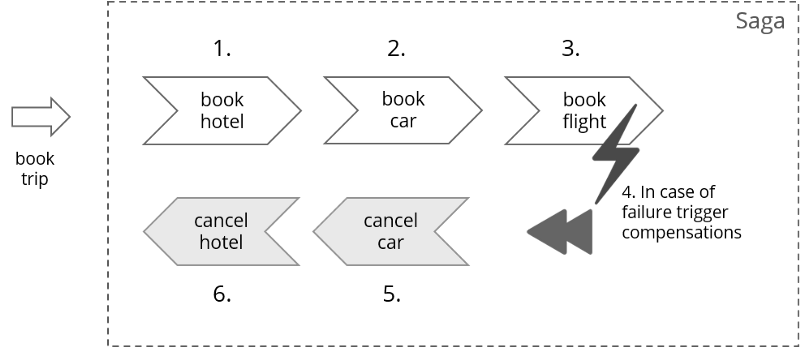
1. 提交阶段：如果事务在每个参与者上都执行成功了，事务协调者发送通知让参与者提交事务；否者，协调者发送通知让参与者回滚事务。在准备阶段，参与者执行了事务，但是还未提交。只有在提交阶段接收到协调者发过来的通知后，才进行提交或者回滚。
2. 存在问题：
3. 同步阻塞：所有事务参与者在等待其他参与者响应的时候都处于同步阻塞状态，无法进行其他操作。
4. 单点问题：协调者在2PC中起到非常大的作用，发生故障将会造成很大影响。特别是在阶段二发生故障，所有参与者会一直等待状态，无法完成其他操作。
5. 数据不一致：在阶段二，如果协调者只发送了部分commit消息，此时网络发生异常，那么只有部分参与者接收到commit消息，也就是说只有部分参与者提交了事务，使得系统数据不一致。
6. 太过保守：任意节点失败就会导致整个事务失败，没有完善的容错机制。
7. 宿舍
8. 补偿事务（TCC,Try Confirm Cancel）：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。
9. 三个阶段：
10. Try阶段主要是对业务系统做检测及资源预留。
11. 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。
12. 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
13. 示例：假如Bob要向smitch转账
14. 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。
15. 在Confirm阶段，执行远程调用的转账的操作，转账成功进行解冻。
16. 如果第2步执行成功，那么转账成功，如果第二步执行失败，则远程调用冻结接口对应的解冻方法（Cancel）。
17. 优缺点：
18. 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差一些。
19. 缺点：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不太好定义以及处理
20. 宿舍
21. 本地消息表（异步确保）：本地消息表与业务数据表处于同一个数据库中，这样就能利用本地事务来保证在对这两个表的操作满足事务特性，并且使用了消息队列来保证最终一致性。



1. 基本思路：
2. 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。
3. 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。
4. 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在发送一遍。如果有靠谱的自动对账和补账逻辑，这种方案还是非常实用的。
5. 这种方案遵循BASE理论，采用的是最终一致性，比较适合实际业务场景。
6. 优缺点：
7. 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。
8. 缺点：消息表会耦合到业务系统中，如果没有封装好，会有很多杂活需要处理。
9. 是
10. MQ事务消息



1. 阿里的RocketMQ是支持事务消息的，他们支持事务消息的方式也类似于采用的二阶段提交，但是主流的MQ是不支持事务消息的，如RabbitMQ和Kafka
2. 大致思路：
3. 第一阶段Prepared消息，会拿到消息的地址。
4. 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。
5. 业务方法内想要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。这样久包装了消息发送与本地事务同时成功或同时失败。
6. 优缺点：
7. 实现了最终一致性，不需要依赖本地数据库事务。
8. 实现难度打，主流MQ不支持，RocketMQ事务消息部分代码也未开源。
9. Sagas事务模型
10. Saga事务模型又叫做长时间运行的事务（Long-runing-transaction），它描述的是另外一种在没有两阶段提交的情况下解决分布式系统中复杂的业务事务问题。
11. 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。
12. 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，他们分别是预定车俩、预定宾馆、预定机票，他们分别属于三个不同的远程接口。从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属于同一事务的。



1. 话花
2. 宿舍
3. **生生世**