分布式事务的实现原理

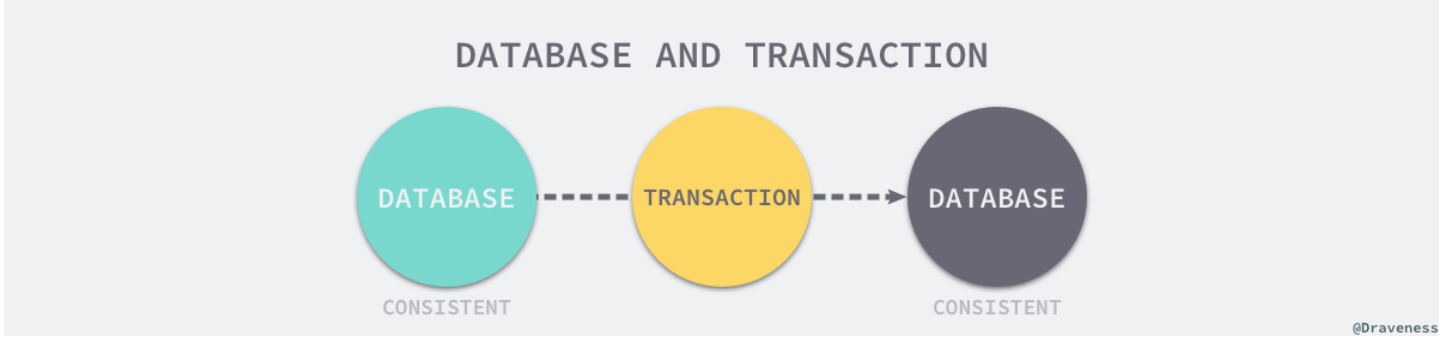
14 AUG 2018 分布式系统 分布式事务 事务

事务是数据库系统中非常有趣也非常重要的概念，它是数据库管理系统执行过程中的一个逻辑单元，它能够保证一个事务中的所有操作要么全部执行，要么全不执行；在 SOA 与微服务架构大行其道的今天，在分布式的多个服务中保证业务的一致性就需要我们实现分布式事务。

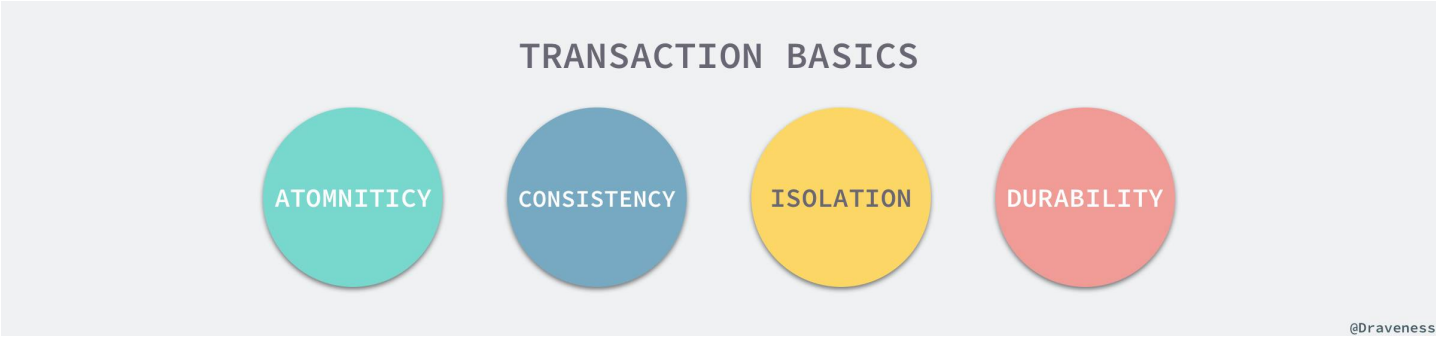
在这篇文章中，我们将介绍 事务的实现原理、分布式事务的理论基础以及实现原理。

事务

在文章的开头，我们已经说过事务是数据库管理系统执行过程中的一个逻辑单位，它能保证一组数据库操作要么全部执行，要么全不执行，我们能够通过事务将数据库从一个状态迁移到另一个状态，在每一个状态中，数据库中的数据都保持一致性。



数据库事务拥有四个特性，原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）和持久性（Durability）：



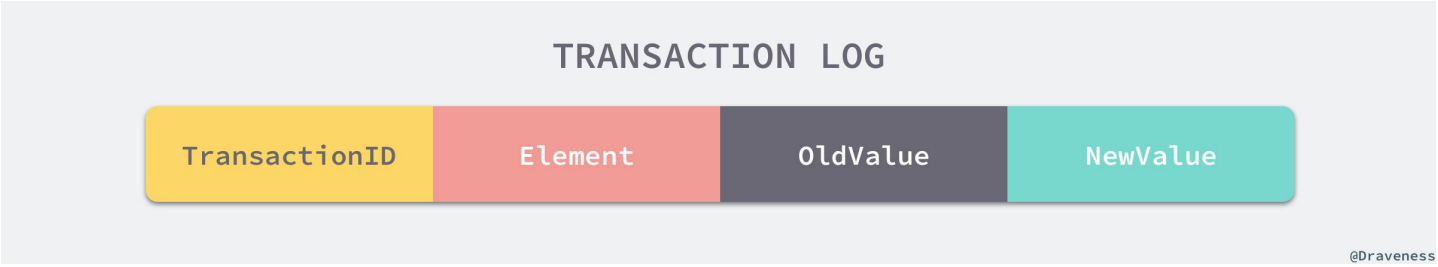
我们经常将这上述的四大特性简写为 ACID，而数据库事务的实现原理其实也就是实现这四大特性的原理。

实现原理

在之前的文章 『浅入深出』MySQL 中事务的实现 中其实已经对如何实现事务的 ACID 这几个基本属性给出了比较详细的介绍和分析，在这里就简单介绍几个比较重要的实现细节，关于展开的内容，可以阅读上述文章。

事务日志

为了实现确保事务能在执行的任意过程中回滚（原子性）并且提交的事务会永久保存在数据库中，我们会使用事务日志来存储事务执行过程中的数据库的变动，每一条事务日志中都包含事务的 ID、当前被修改的元素、变动前以及变动后的值。

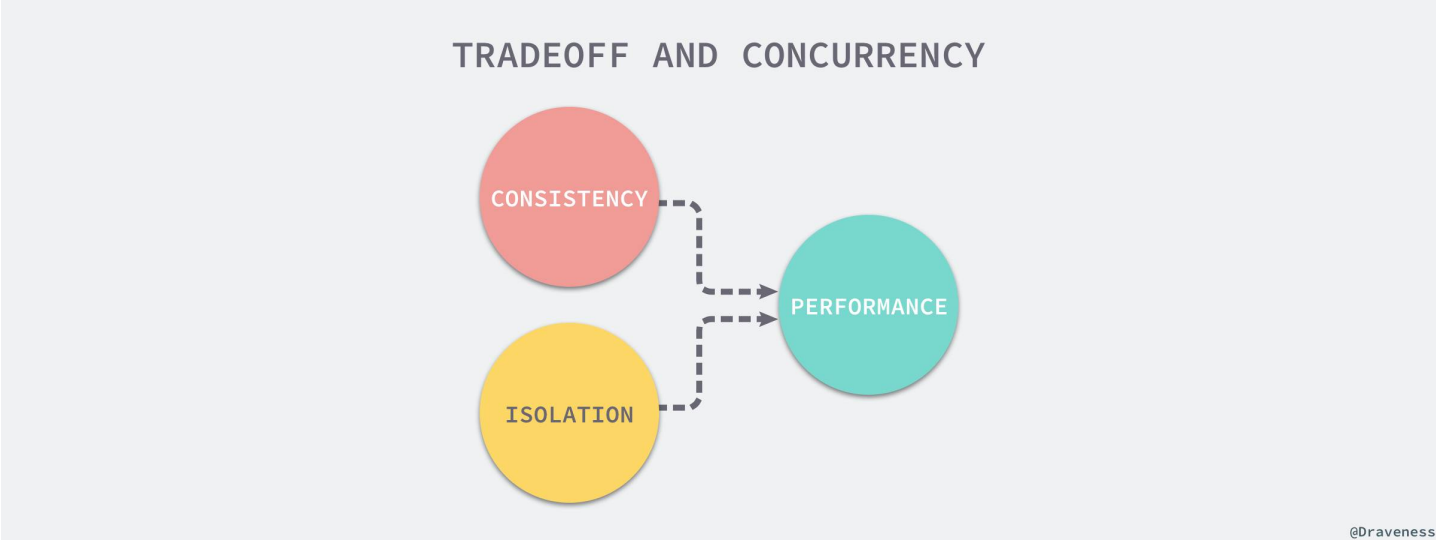


当我们有以上的事务日志之后，一旦需要对事务进行回滚就非常容易了，数据库会根据上述日志生成一个相反的操作恢复事务发生之前的状态；事务日志除了能够对事务进行回滚保证原子性之外，还能够实现持久性，当一个事务常食对数据库进行修改时，它其实会先生成一条日志并刷新到磁盘上，写日志的操作由于是追加的所以非常快，在这之后才会向数据库中写入或者更新对应的记录。

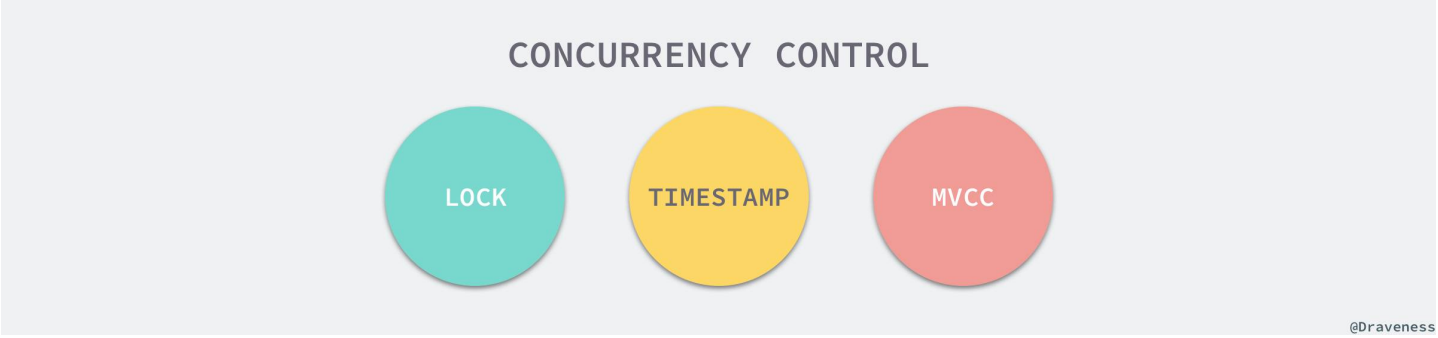
在 MySQL 最常见的存储引擎 InnoDB 中，事务日志其实有两种，一种是回滚日志（undo log），另一种是重做日志（redo log），其中前者保证事务的原子性，后者保证事务的持久性，两者可以统称为事务日志。

并发控制

数据库作为最关键的后端服务，很难想象只能串行执行每一个数据库操作带来的性能影响，然而在并发执行 SQL 的过程中就可能无法保证数据库对于隔离性的要求，归根结底这就是一致性、隔离性与性能之间的权衡。



为了避免并发带来的一致性问题、满足数据库对于隔离性要求，数据库系统往往都会使用并发控制机制尽可能地充分利用机器的效率，最常见的几种并发控制机制就是锁、时间戳和 MVCC：

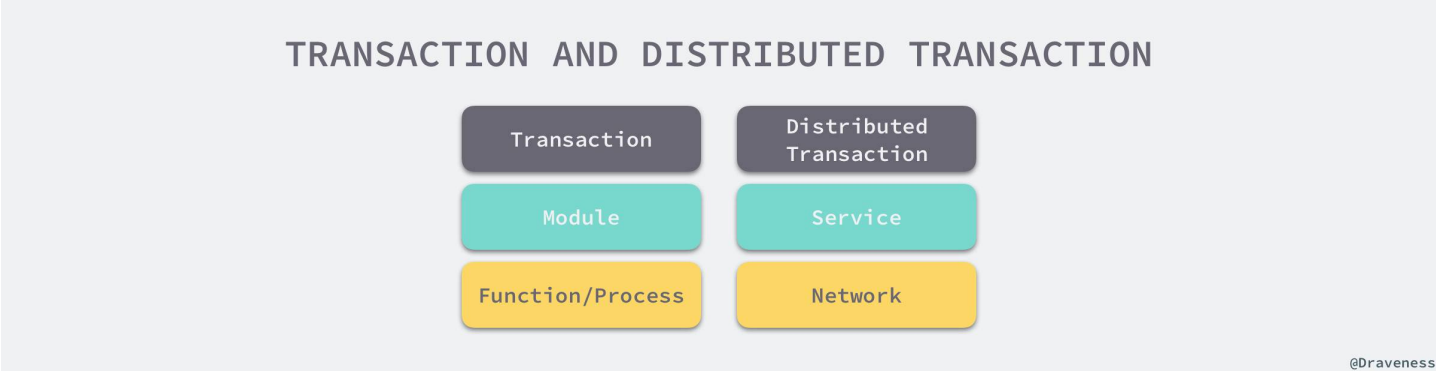


作为悲观并发控制机制，锁使用在更新资源之前对资源进行锁定的方式保证多个数据库的会话同时修改某一行记录时不会出现脱离预期的行为，而时间戳这种方式在每次提交时对资源是否被改变进行检查。

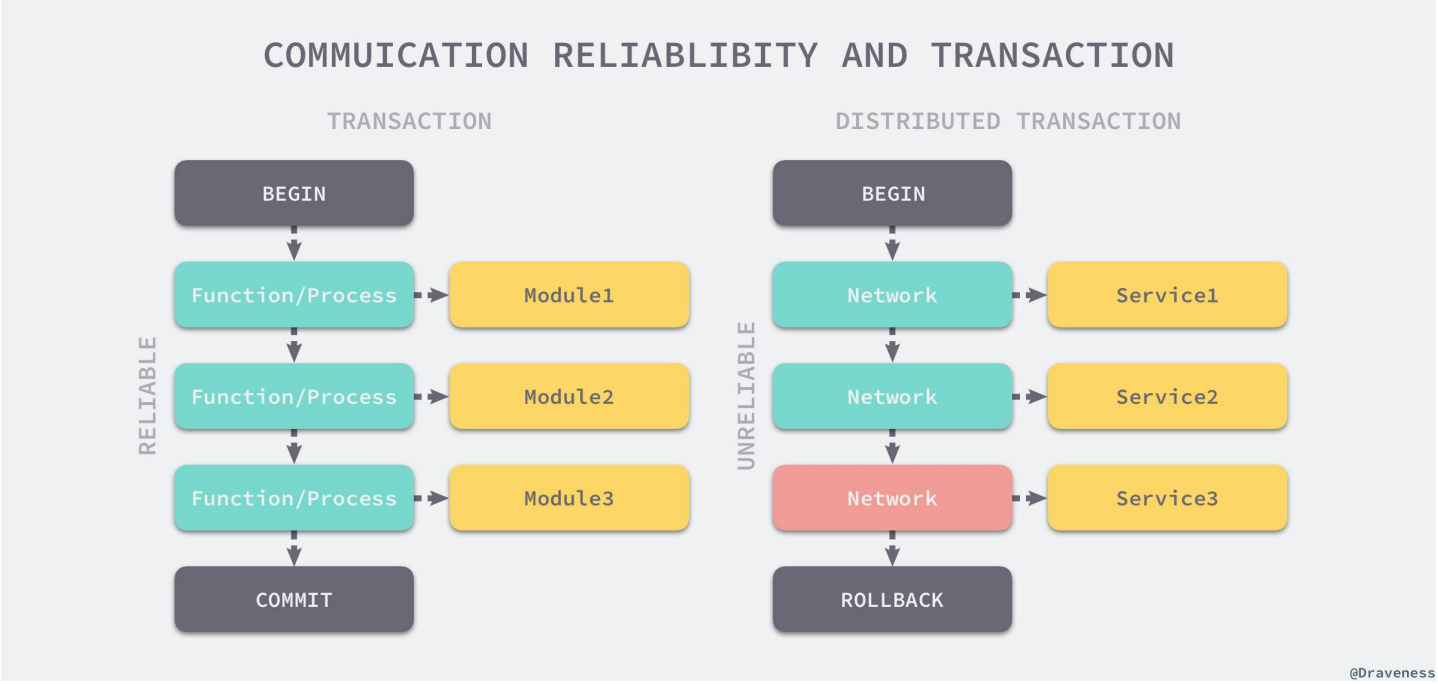
> *在 浅谈数据库并发控制 - 锁和 MVCC 中，作者介绍过几种不同的并发控制机制的实现原理，想要了解更多相关的内容的可以阅读这篇文章。*

分布式事务

从广义上来看，分布式事务其实也是事务，只是由于业务上的定义以及微服务架构设计的问题，所以需要在多个服务之间保证业务的事务性，也就是 ACID 四个特性；从单机的数据库事务变成分布式事务时，原有单机中相对可靠的方法调用以及进程间通信方式已经没有办法使用，同时由于网络通信经常是不稳定的，所以服务之间信息的传递会出现障碍。



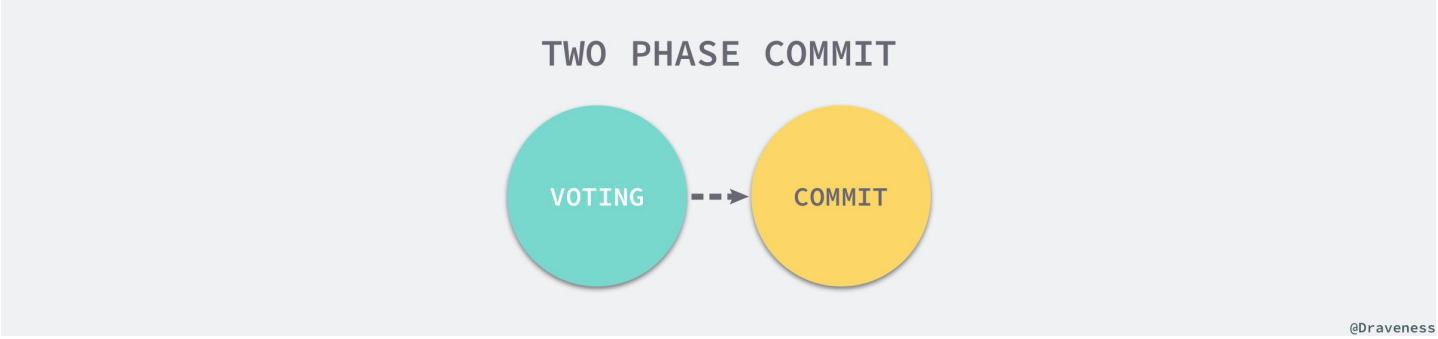
模块（或服务）之间通信方式的改变是造成分布式事务复杂的最主要原因，在同一个事务之间的执行多段代码会因为网络的不稳定造成各种奇怪的问题，当我们通过网络请求其他服务的接口时，往往会得到三种结果：正确、失败和超时，无论是成功还是失败，我们都能得到唯一确定的结果，**超时代表请求的发起者不能确定接受者是否成功处理了请求**，这也是造成诸多问题的诱因。



系统之间的通信可靠性从单一系统中的可靠变成了微服务架构之间的不可靠，分布式事务其实就是在不可靠的通信下实现事务的特性。无论是事务还是分布式事务实现原子性都无法避免对**持久存储**的依赖，事务使用磁盘上的日志记录执行的过程以及上下文，这样无论是需要**回滚**还是**补偿**都可以通过日志追溯，而分布式事务也会依赖数据库、Zookeeper 或者 ETCD 等服务追踪事务的执行过程，总而言之，**各种形式的日志是保证事务几大特性的重要手段**。

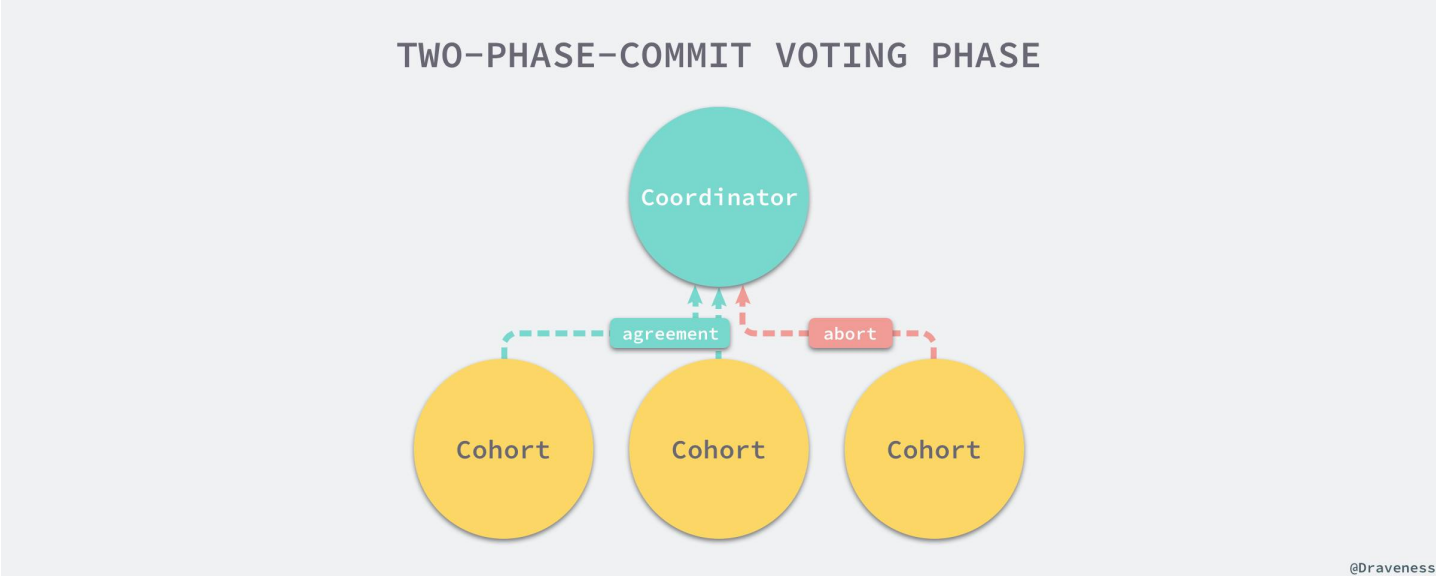
2PC 与 3PC

两阶段提交是一种使分布式系统中所有节点在进行事务提交时保持一致性而设计的一种协议；在一个分布式系统中，所有的节点虽然都可以知道自己执行操作后的状态，但是无法知道其他节点执行操作的状态，在一个事务跨越多个系统时，就需要引入一个作为协调者的组件来统一掌控全部的节点并指示这些节点是否把操作结果进行真正的提交，想要在分布式系统中实现一致性的其他协议都是在两阶段提交的基础上做的改进。

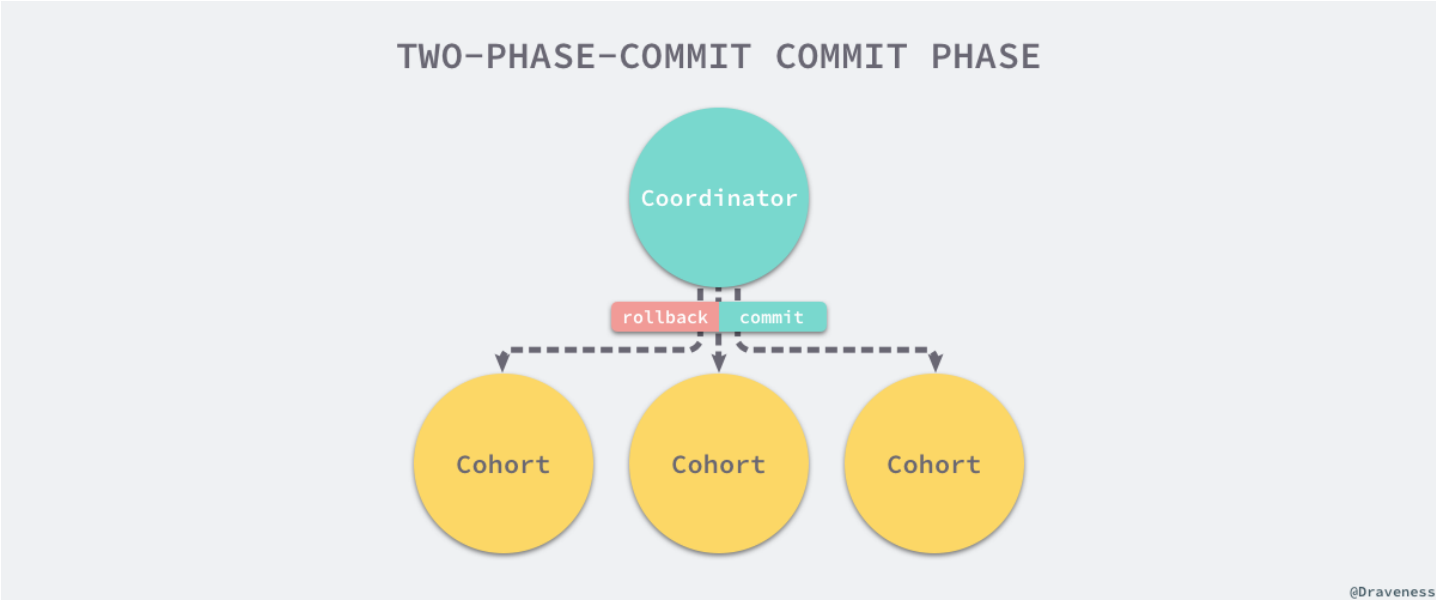


两阶段提交的执行过程就跟它的名字一样分为两个阶段，*投票阶段*和*提交阶段*，在投票阶段中，协调者（Coordinator）会向事务的参与者（Cohort）询问是否可以执行操作的请求，并

等待其他参与者的响应，参与者会执行相对应的事务操作并**记录重做和回滚日志**，所有执行成功的参与者会向协调者发送 `AGREEMENT` 或者 `ABORT` 表示执行操作的结果。



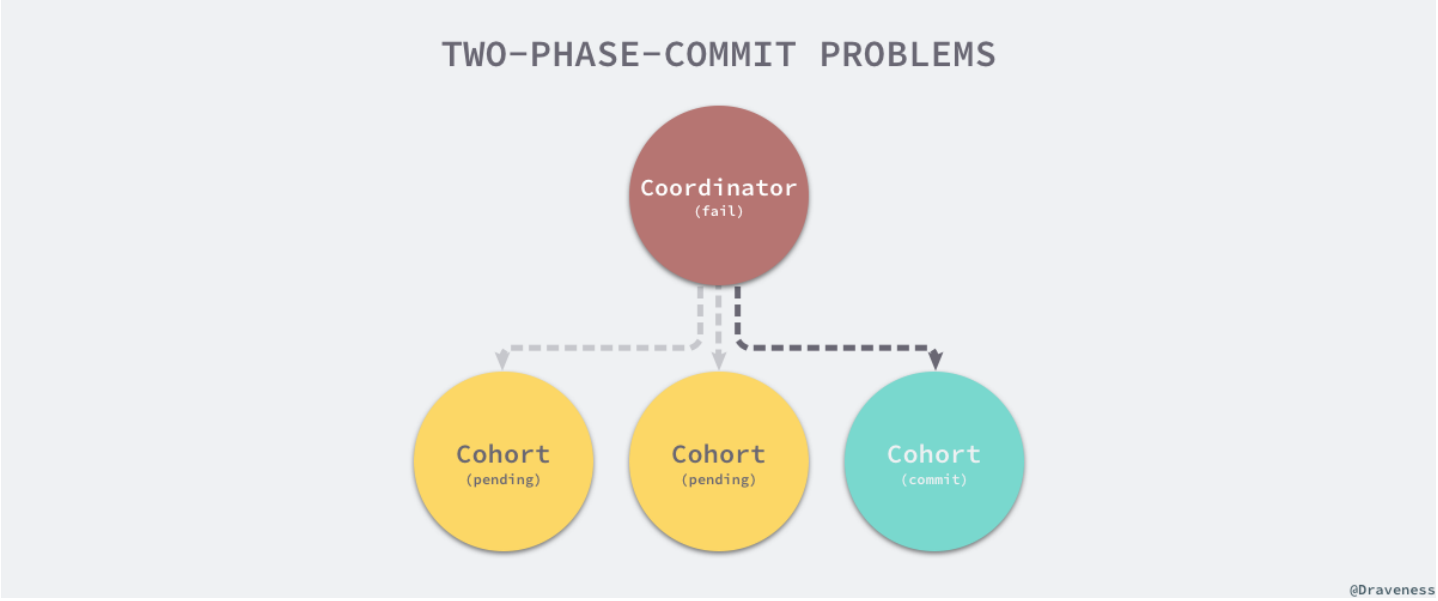
当所有的参与者都返回了确定的结果（同意或者终止）时，两阶段提交就进入了提交阶段，协调者会根据投票阶段的返回情况向所有的参与者发送提交或者回滚的指令。



当事务的所有参与者都决定提交事务时，协调者会向参与者发送 `COMMIT` 请求，参与者在完成操作并释放资源之后向协调者返回完成消息，协调者在收到所有参与者的完成消息时会结束整个事务；与之相反，当有参与者决定 `ABORT` 当前事务时，协调者会向事务的参与者发送回滚请求，参与者会根据之前执行操作时的**回滚日志**对操作进行回滚并向协调者发送完成的消息，在提交阶段，无论当前事务被提交还是回滚，所有的资源都会被释放并且事务也一定会结束。

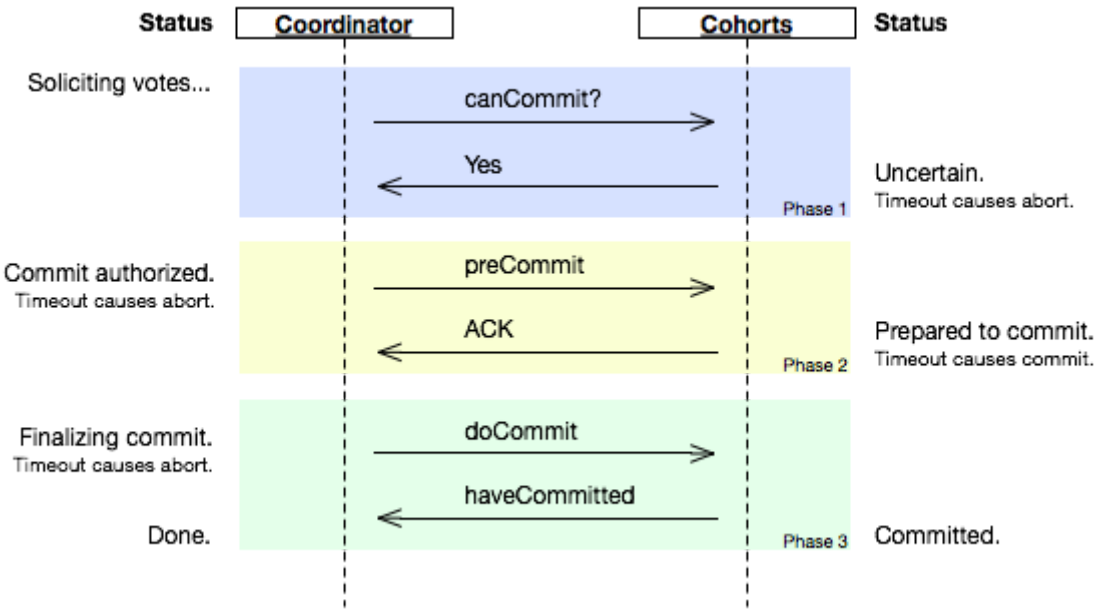
两阶段提交协议是一个阻塞协议，也就是说在两阶段提交的执行过程中，除此之外，如果事务的执行过程中协调者永久宕机，事务的一部分参与者将永远无法完成事务，它们会等待协调者

发送 `COMMIT` 或者 `ROLLBACK` 消息，甚至会出现多个参与者状态不一致的问题。



3PC

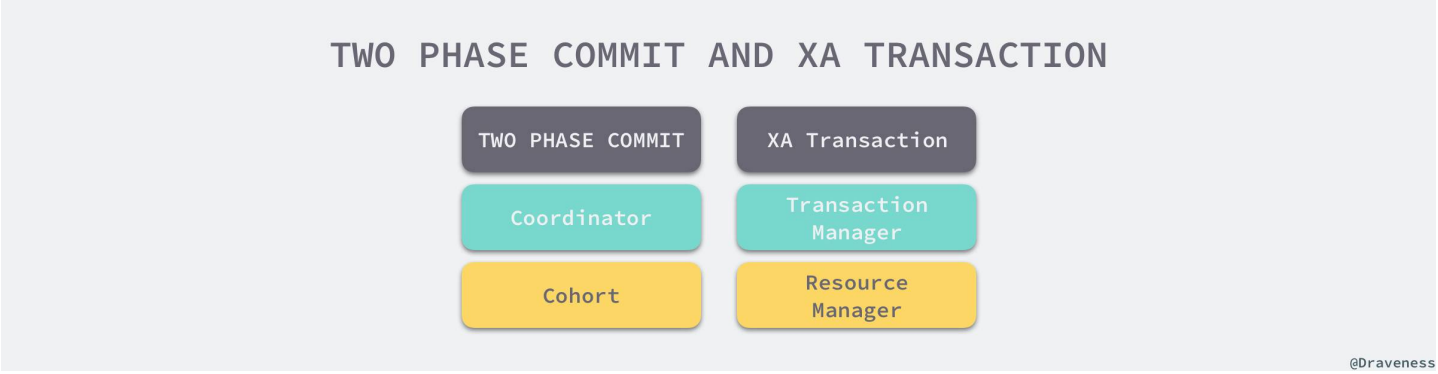
为了解决两阶段提交在协议的一些问题，三阶段提交引入了*超时机制*和*准备阶段*，如果协调者或者参与者在规定的之间内没有接受到来自其他节点的响应，就会根据当前的状态选择提交或者终止整个事务，准备阶段的引入其实让事务的参与者有了除回滚之外的其他选择。



当参与者向协调者发送 ACK 后，如果长时间没有得到协调者的响应，在默认情况下，参与者会自动将超时的事务进行提交，不会像两阶段提交中被阻塞住；上述的图片非常清楚地说明了在不同阶段，协调者或者参与者的超时会造成什么样的行为。

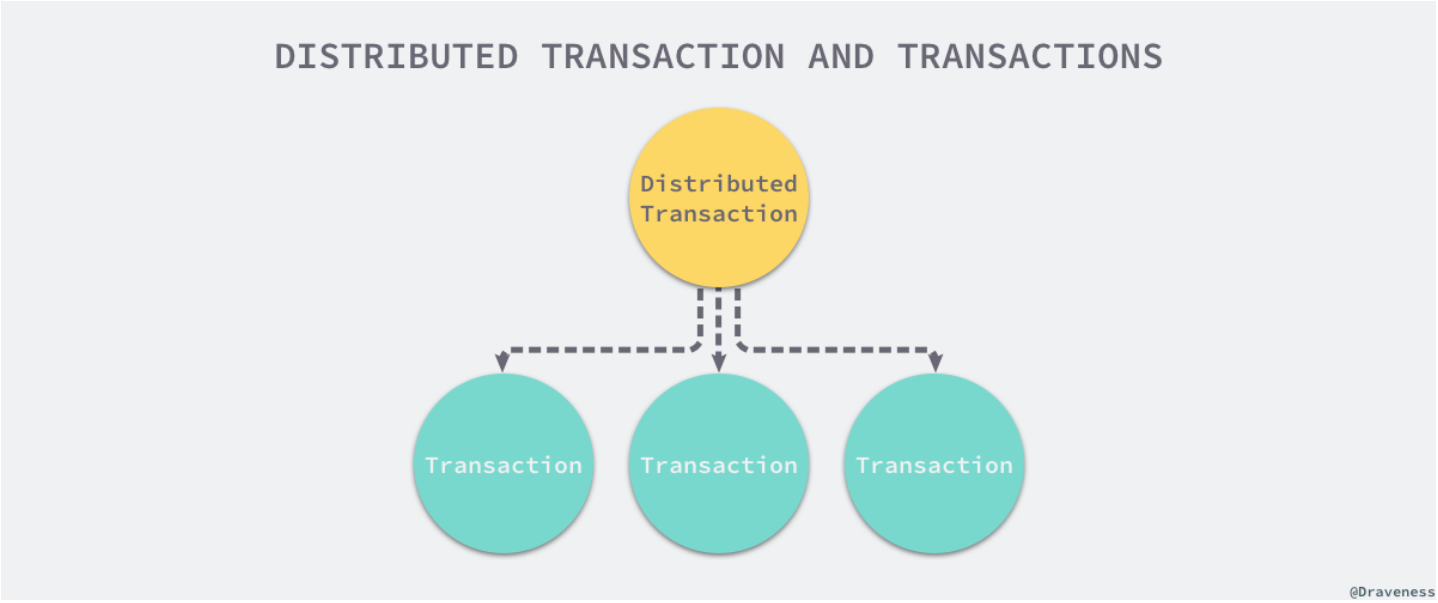
XA 事务

MySQL 的 InnoDB 引擎其实能够支持分布式事务，也就是我们经常说的 XA 事务；XA 事务就是用了我们在上一节中提到的两阶段提交协议实现分布式事务，其中事务管理器为协调者，而资源管理器就是分布式事务的参与者。

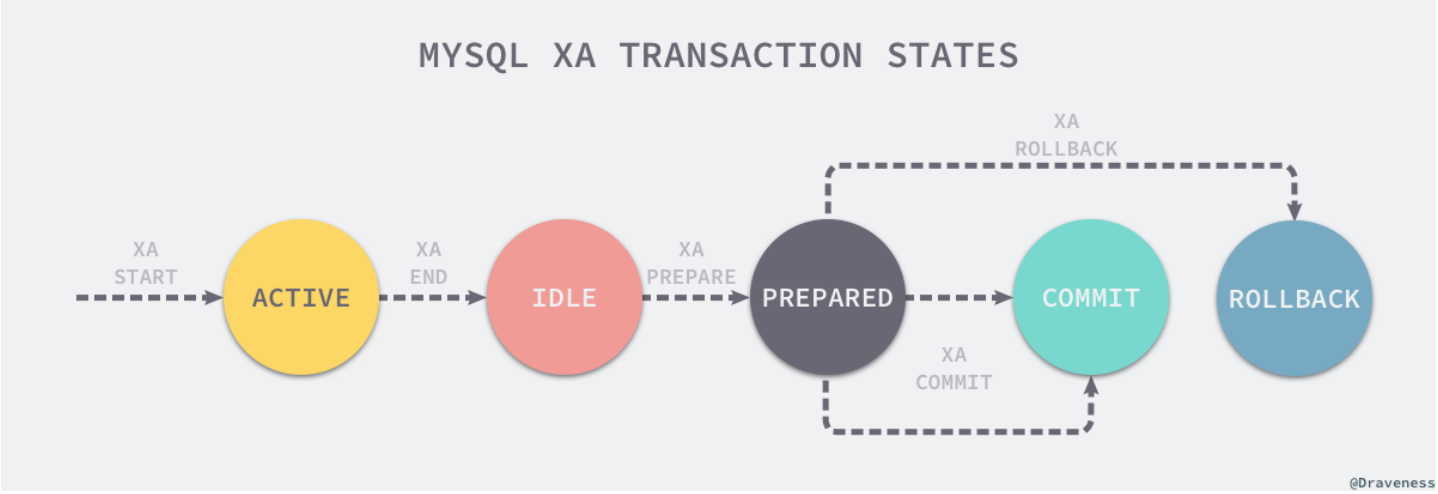


到这里，其实我们已经能够清晰地知道 MySQL 中的 XA 事务是如何实现的：

- 资源管理器提供了访问事务资源的能力，数据库就是一种常见的资源管理器，它能够提交或者回滚其管理的事务；
- 事务管理器协调整个分布式事务的各个部分，它与多个资源管理器通信，分别处理他们管理的事务，这些事务都是整体事务的一个分支。



正如两阶段提交协议中定义的，MySQL 提供的 XA 接口可以非常方便地实现协议中的投票和提交阶段，我们可以通过一下的流程图简单理解一下 MySQL XA 的接口是如何使用的：



XA 确实能够保证较强的一致性，但是在 MySQL XA 的执行过程中会对相应的资源加锁，阻塞其他事务对该资源的访问，如果事务长时间没有 `COMMIT` 或者 `ROLLBACK`，其实会对数据库造成比较严重的影响。

Saga

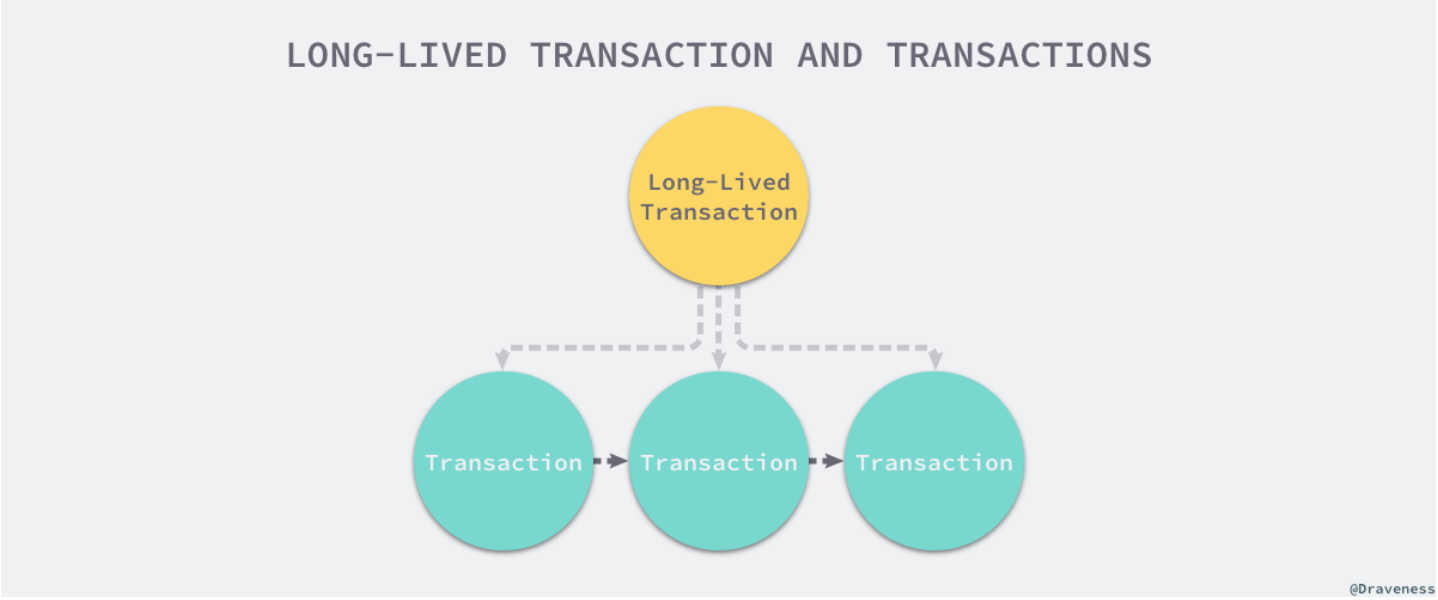
两阶段提交其实可以保证事务的强一致性，但是在很多业务场景下，我们其实只需要保证业务的最终一致性，在一定的时间窗口内，多个系统中的数据不一致是可以接受的，在过了时间窗口之后，所有系统都会返回一致的结果。

Saga 其实就一种简化的分布式事务解决方案，它将一系列的分布式操作转化成了一系列的本地事务，在每一个本地事务中我们都会更新数据库并且向集群中的其他服务发送一条的新的消息来触发下一个本地的事务；一旦本地的事务因为违反了业务逻辑而失败，那么就会立刻触发一系列的回滚操作来撤回之前本地事务造成的副作用。

LLT

相比于本地的数据库事务来说，长事务（Long Lived Transaction）会对一些数据库资源持有相对较长的一段时间，这会严重地影响其他正常数据库事务的执行，为了解决这一问题，Hector Garcia-Molina 和 Kenneth Salem 在 1987 发布了论文 Sagas 用于解决这一问题。

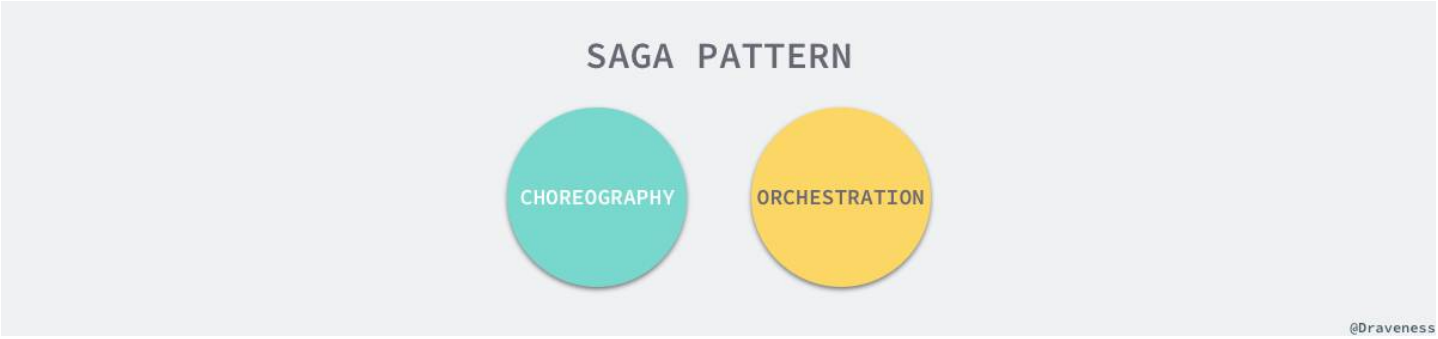
如果一个 LLT 能够被改写成一系列的相互交错重叠的多个数据库事务，那么这个 LLT 就是一个 Saga；数据库系统能够保证 Saga 中一系列的事务要么全部成功执行、要么它们的补偿事务能够回滚全部的副作用，保证整个分布式事务的最终一致性。Saga 的概念和它的实现都是非常简单的，但是它却能够有很大的潜力增加整个系统的处理能力。



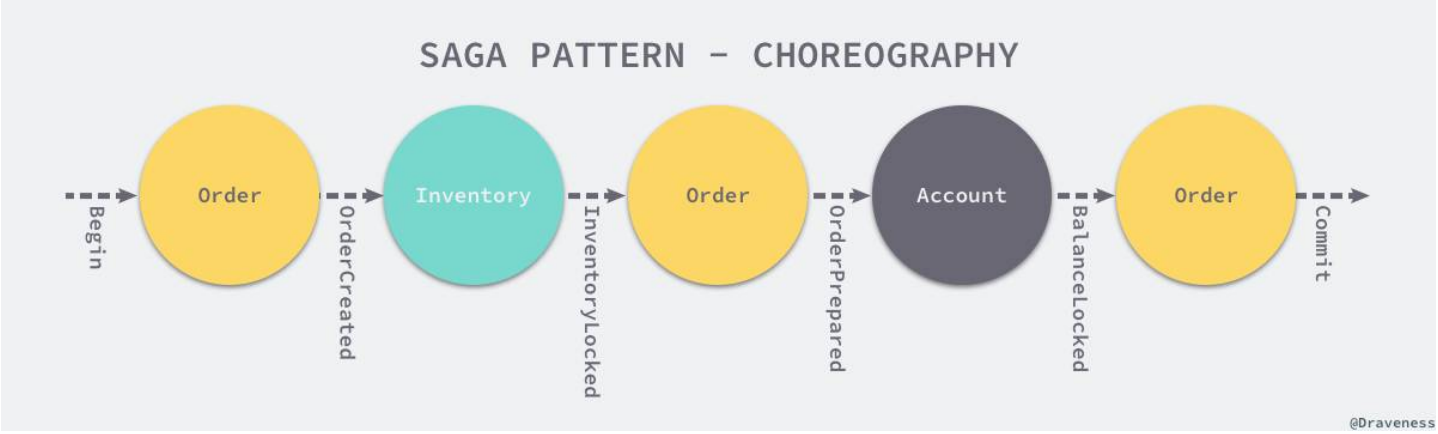
事务越长并且越复杂，那么这个事务由于异常而被回滚以及死锁的可能性就会逐渐增加，Saga 会将一个 LLT 分解成多个短事务，能够非常明显地降低事务被回滚的风险。

协同与编排

当我们使用 Saga 模式开发分布式事务时，有两种协调不同服务的方式，一种是协同（Choreography），另一种是编排（Orchestration）：



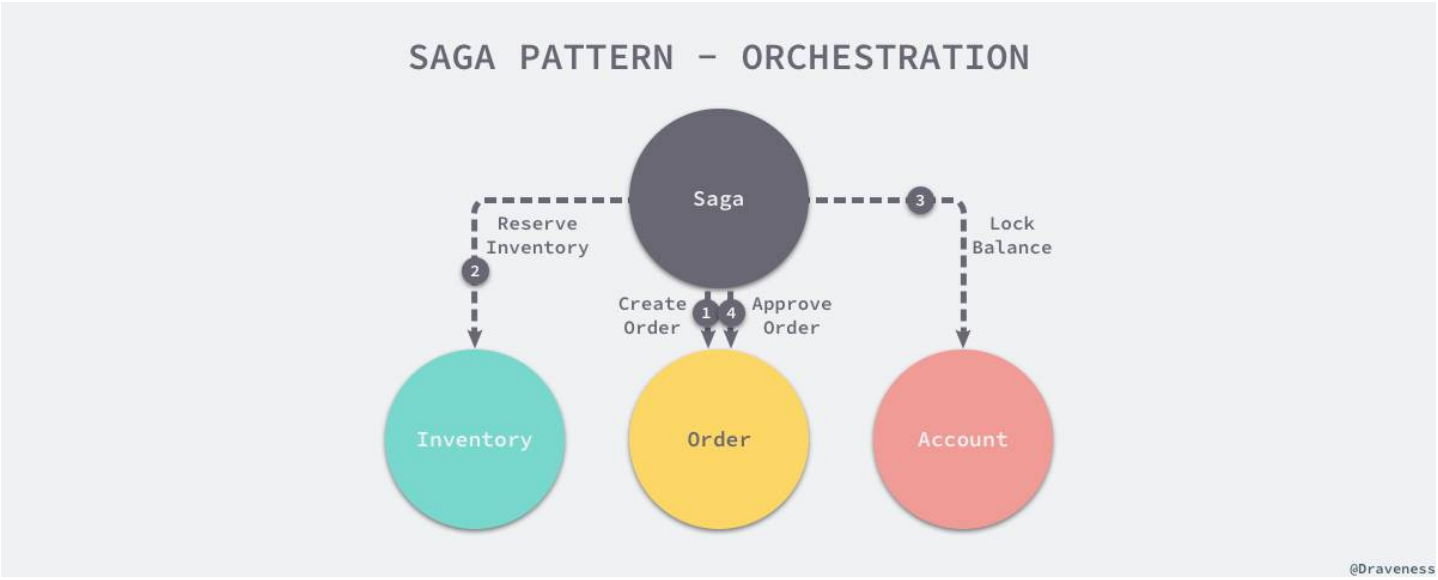
如果对于一个分布式事务，我们采用协同的方式进行开发，每一个本地的事务都会触发一个其他服务中的本地事务的执行，也就是说事务的执行过程是一个流的形式进行的：



当我们选择使用协同的方式处理事务时，服务之间的通信其实就是通过事件进行的，每一个本的事务最终都会向服务的下游发送一个新的事件，既可以是消息队列中的消息，也可以是 RPC 的请求，只是下游提供的接口需要保证幂等和重入。

除此之外，通过协同方式创建的分布式事务其实并没有明显的中心化节点，多个服务参与者之间的交互协议要从全局来定义，每个服务能够处理以及发送的事件和接口都需要进行比较严谨的设计，尽可能提供抽象程度高的事件或者接口，这样各个服务才能实现自治并重用已有的代码和逻辑。

如果我们不想使用协同的方式对分布式事务进行处理，那么也可以选择编排的方式实现分布式事务，编排的方式引入了中心化的协调器节点，我们通过一个 Saga 对象来追踪所有的子任务的调用情况，根据任务的调用情况决定是否需要调用对应的补偿方案，并在网络请求出现超时时进行重试：



在这里我们就引入了一个中心化的『协调器』，它会保存当前分布式事务进行到底的状态，并根据情况对事务进行回滚或者提交操作，在服务编排的过程中，我们是从协调者本身触发考虑整个事务的执行过程的，相对于协同的方式，编排实现的过程相对来说更为简单。

协同与编排其实是两种思路截然相反的模式，前者强调各个服务的自治与去中心化，后者需要一个中心化的组件对事务执行的过程进行统一的管理，两者的优缺点其实就是中心化与去中心化的优缺点，中心化的方案往往都会造就一个『上帝服务』，其中包含了非常多组织与集成其他节点的工作，也会有单点故障的问题，而去中心化的方案就会带来管理以及调试上的不便，当我们需要追踪一个业务的执行过程时就需要跨越多个服务进行，增加了维护的成本。

下游约束

当我们选择使用 Saga 对分布式事务进行开发时，会对分布式事务的参与者有一定的约束，每一个事务的参与者都需要保证：

1. 提供接口和补偿副作用的接口；
2. 接口支持重入并通过全局唯一的 ID 保证幂等；

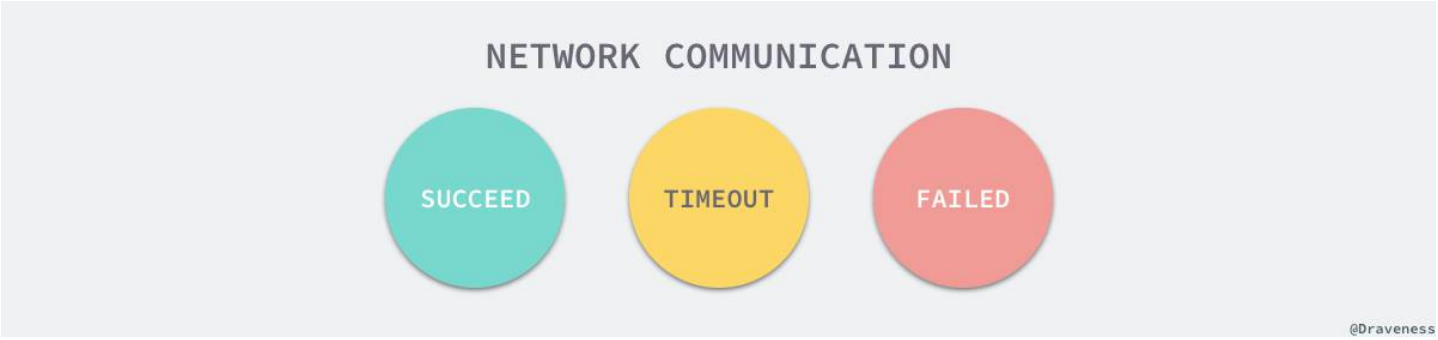
这样我们就能够保证一个长事务能够在网络通信发生超时时进行重试，同时在需要对事务进行回滚时调用回滚接口达到我们的目的。

小结

Saga 这种模式其实完全放弃了同时满足事务四大基本特性 ACID 的想法，而是选择降低实现分布式事务的难度并减少资源同步以及锁定带来的问题，选择实现 BASE(Basic Availability, Soft, Eventual consistency) 事务，达到业务上的基本可用以及最终一致性，在绝大多数的业务场景中，实现最终一致性就能够基本满足业务的全部需求，极端场景下还是应该选择两阶段提交或者干脆放弃分布式事务这种易错的实现方式，转而使用单机中的数据库事务来解决。

消息服务

分布式事务带来复杂度的原因其实就是由于各个模块之间的通信不稳定，当我们发出一个网络请求时，可能的返回结果是成功、失败或者超时。



网络无论是返回成功还是失败其实都是一个确定的结果，当网络请求超时的时候其实非常不好处理，在这时调用方并不能确定这一次请求是否送达而且不会知道请求的结果，但是消息服务可以保证某条信息一定会送达到调用方；大多数消息服务都会提供两种不同的 QoS，也就是服务的等级。

最常见的两种服务等级就是 At-Most-Once 和 At-Least-Once，前者能够保证发送方不对接收方是否能收到消息作保证，消息要么会被投递一次，要么不会被投递，这其实跟一次普通的网络请求没有太多的区别；At-Least-Once 能够解决消息投递失败的问题，它要求发送者检查投递的结果，并在失败或者超时时重新对消息进行投递，发送者会持续对消息进行推送，直到接受者确认消息已经被收到，相比于 At-Most-Once，At-Least-Once 因为能够确保消息的投递会被更多人使用。

除了这两种常见的服务等级之外，还有另一种服务等级，也就是 Exactly-Once，这种服务等级不仅对发送者提出了要求，还对消费者提出了要求，它需要接受者对接收到的所有消息进行去重，发送者和接受者一方对消息进行重试，另一方对消息进行去重，两者分别部署在不同的节点上，这样对于各个节点上的服务来说，它们之间的通信就是 Exactly-Once 的，但是需要注意的是，Exacly-Once 一定需要接收方的参与。

我们可以通过实现 AMQP 协议的消息队列来实现分布式事务，在协议的标准中定义了 `tx_select`、`tx_commit` 和 `tx_rollback` 三个事务相关的接口，其中 `tx_select` 能够开启事务，`tx_commit` 和 `tx_rollback` 分别能够提交或者回滚事务。

使用消息服务实现分布式事务在底层的原理上与其他的方法没有太多的差别，只是消息服务能够帮助我们实现的消息的持久化以及重试等功能，能够为我们提供一个比较合理的 API 接口，方便开发者使用。

总结

分布式事务的实现方式是分布式系统中非常重要的一个问题，在微服务架构和 SOA 大行其道的今天，掌握分布式事务的原理和使用方式已经是作为后端开发者理所应当掌握的技能，从实现 ACID 事务的 2PC 与 3PC 到实现 BASE 补偿式事务的 Saga，再到最后通过事务消息的方式异步地保证消息最终一定会被消费成功，我们为了增加系统的吞吐量以及可用性逐渐降低了系统对一致性的要求。

在业务没有对一致性有那么强的需求时，作者一般会使用 Saga 协议对分布式事务进行设计和开发，而在实际工作中，需要强一致性事务的业务场景几乎没有，我们都可以实现最终一致性，在发生脑裂或者不一致问题时通过补偿的方式进行解决，这就能解决几乎全部的问题。

相关文章

- 基础
  - 分布式事务的实现原理
  - 分布式系统与消息的投递
- 数据库
  - 分布式键值存储 Dynamo 的实现原理
  - 浅析 Bigtable 和 LevelDB 的实现
- 分布式协调 & 服务发现
  - 详解分布式协调服务 ZooKeeper
  - 高可用分布式存储 etcd 的实现原理
  - 详解 DNS 与 CoreDNS 的实现原理
- 容器编排
  - 谈 Kubernetes 的架构设计与实现原理

Reference

- Database transaction · Wikipedia
- 『浅入深出』MySQL 中事务的实现
- MySQL · 特性分析 · 浅谈 MySQL 5.7 XA 事务改进
- XA Transactions
- Two-phase commit protocol

- Pattern: Saga
- Sagas
- RocketMQ 4.3正式发布，支持分布式事务
- Akka Message Delivery - At-Most-Once, At-Least-Once, and Exactly-Once
  - Part 1 At-Most-Once
  - Part 2 At-Least-Once
  - Part 3 Exactly-Once
- Message Delivery Reliability

关于图片和转载

关于评论和留言

如果对本文 分布式事务的实现原理 的内容有疑问，请在下面的评论系统中留言，谢谢。

> *原文链接：分布式事务的实现原理 · 面向信仰编程*
>
> *Follow: Draveness · GitHub*