


高并发高性能 数据库设计-进阶篇



• 讲师：KimmKing •

目录 | Contents

1. 性能分析与性能优化
2. SQL 与查询优化
3. MySQL 高可用架构
4. 数据库分库分表
5. BASE 分布式事务
6. 数据库框架与中间件

三、MySQL 高可用架构

为什么要做数据库高可用？

- 1、读写分离，提升读的处理能力
- 2、故障转移，提供 failover 能力

加上业务侧连接池的心跳重试，实现断线重连，业务不间断，降低 RTO 和 RPO。

三、MySQL 高可用架构

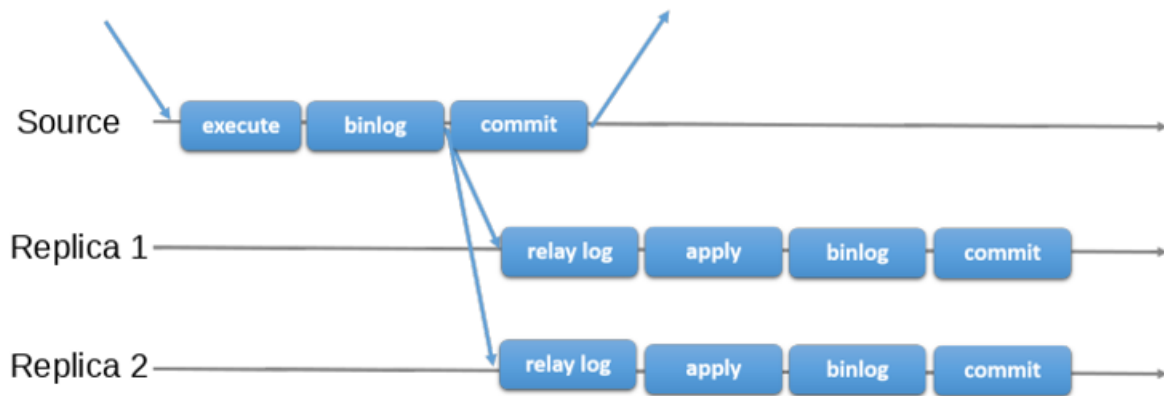
① MySQL高可用介绍

1. Primary-Secondary Replication (传统主从复制)
2. MySQL Group Replication (MGR)
3. MySQL InnoDB Cluster (完整的数据库层高可用解决方案)
4. orchestrator (MySQL高可用管理工具)
5. mega-HA (内部闭源)

三、MySQL 高可用架构

Primary-Secondary Replication (传统主从复制)

异步复制：网络或机器故障，会造成数据不一致

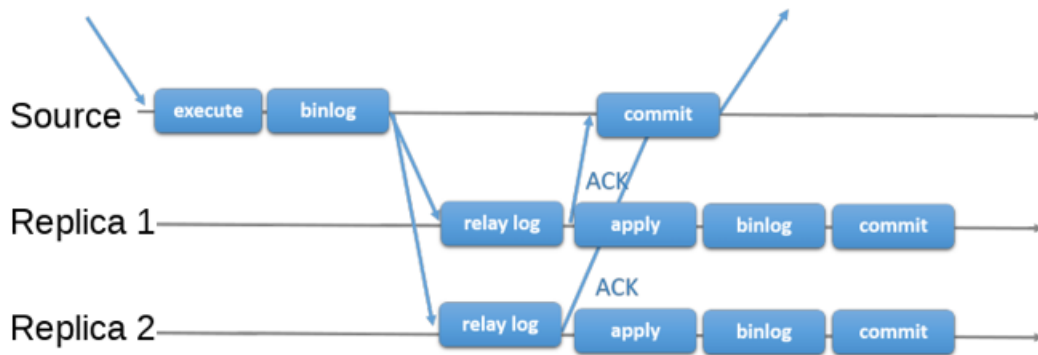


MySQL Asynchronous Replication

三、MySQL 高可用架构

Primary-Secondary Replication (传统主从复制)

半同步复制：保证Source和Replica最终一致性

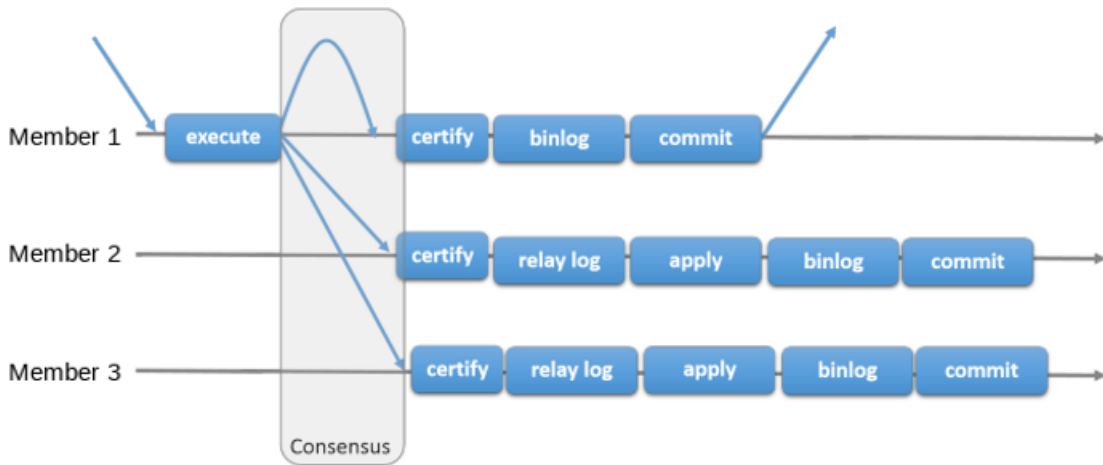


MySQL Semisynchronous Replication

三、MySQL 高可用架构

MySQL Group Replication (MGR)

组复制：基于分布式Paxos协议实现组复制，保证数据一致性



三、MySQL 高可用架构

MySQL Group Replication (MGR)

MGR的特点

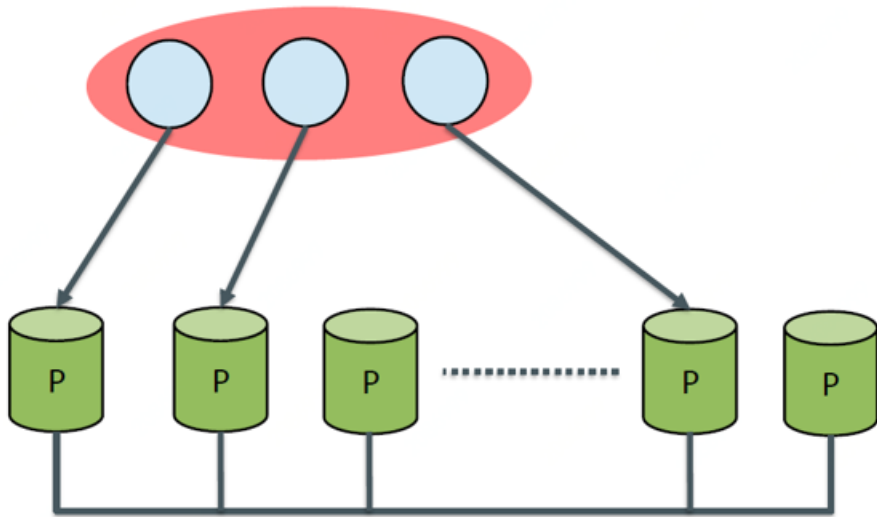
1. 高一致性：基于分布式Paxos协议实现组复制，保证数据一致性；
2. 高容错性：自动检测机制，只要不是大多数节点宕机就可以继续工作，内置防脑裂保护机制；
3. 高扩展性：节点的增加与移除会自动更新组成员信息，新节点加入后，自动从其他节点同步增量数据，直到与其他节点数据一致；
4. 高灵活性：提供单主模式和多主模式，单主模式在主库宕机后能够自动选主，所有写入都在主节点进行，多主模式支持多节点写入。

三、MySQL 高可用架构

适用场景

弹性复制

Environments that require a very fluid replication infrastructure, where the number of servers has to grow or shrink dynamically and with as little pain as possible.

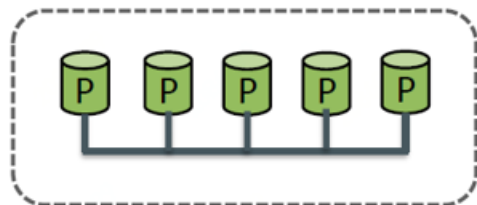


三、MySQL 高可用架构

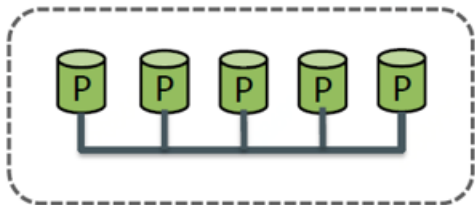
适用场景

高可用分片

Sharding is a popular approach to achieve write scale-out. Users can use MySQL Group Replication to implement highly available shards. Each shard can map into a Replication Group.

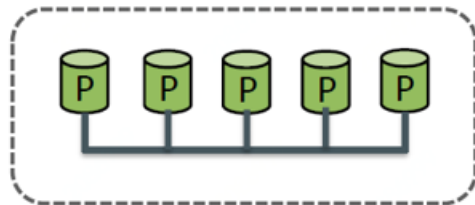


Group Replication Group



Group Replication Group

.....



Group Replication Group

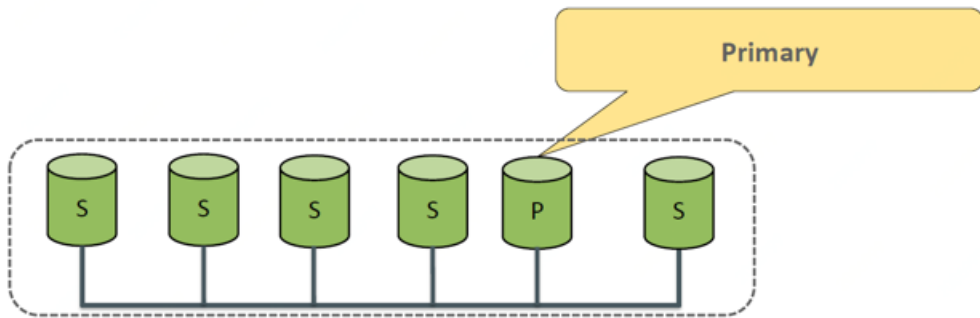
三、MySQL 高可用架构

适用场景

传统主从替代方案

Single-primary mode provides further automation on such setups

1. Automatic PRIMARY/SECONDARY roles assignment
2. Automatic new PRIMARY election on PRIMARY failures
3. Automatic setup of read/write modes on PRIMARY and SECONDARIES
4. Global consistent view of which server is the PRIMARY

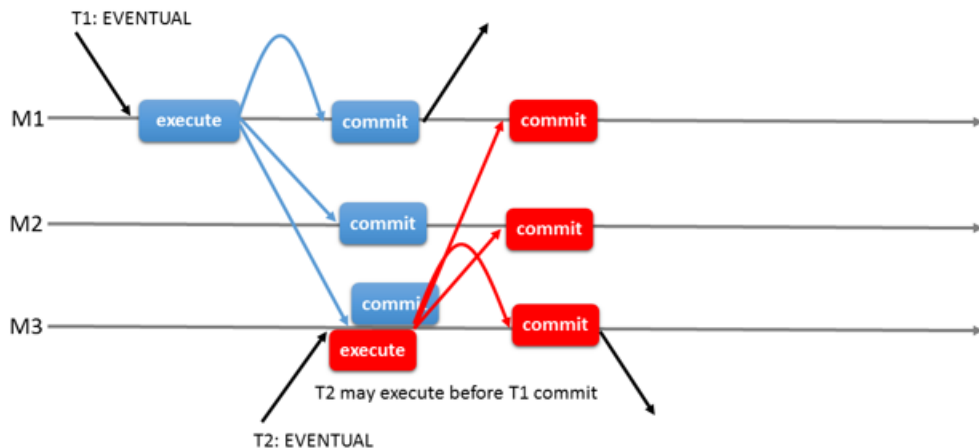


三、MySQL 高可用架构

一致性级别

EVENTUAL (the default)

The transaction does not wait for preceding transactions to be applied before executing, nor does it wait for other members to apply its changes. This was the behaviour of Group Replication before 8.0.14.

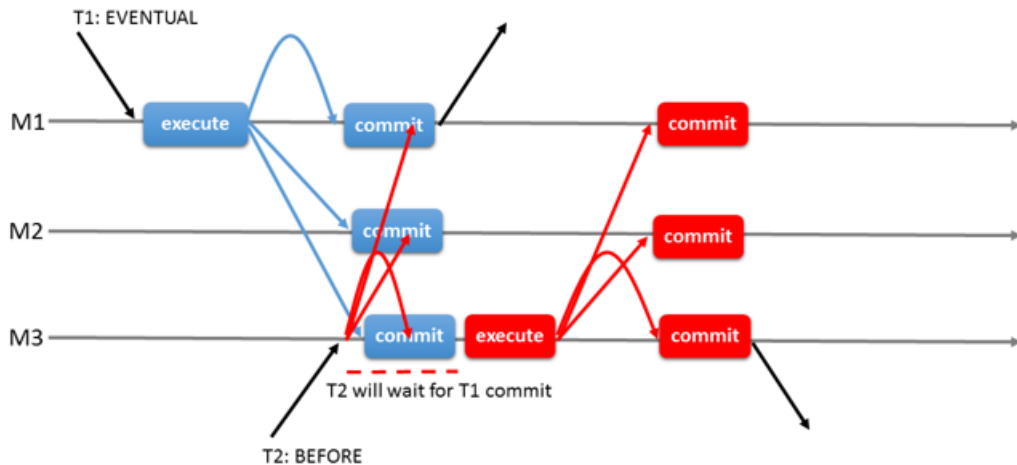


三、MySQL 高可用架构

一致性级别

BEFORE

The transaction will wait until all preceding transactions are complete before starting its execution. This ensures that this transaction will execute on the most up-to-date snapshot of the data, regardless of which member it is executed on.

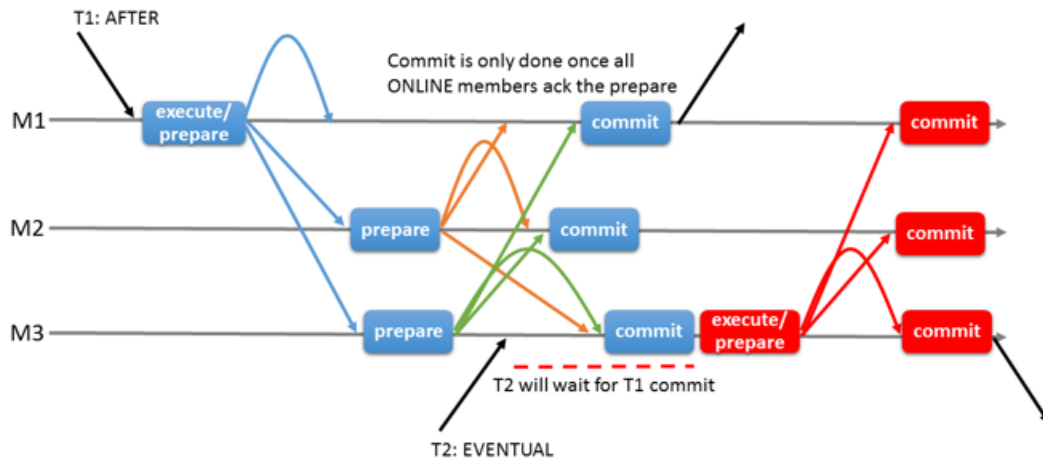


三、MySQL 高可用架构

一致性级别

AFTER

The transaction will wait until its changes have been applied on other members. This ensures that once this transaction completes, all following transactions read a database state that includes its changes, regardless of which member they are executed on.



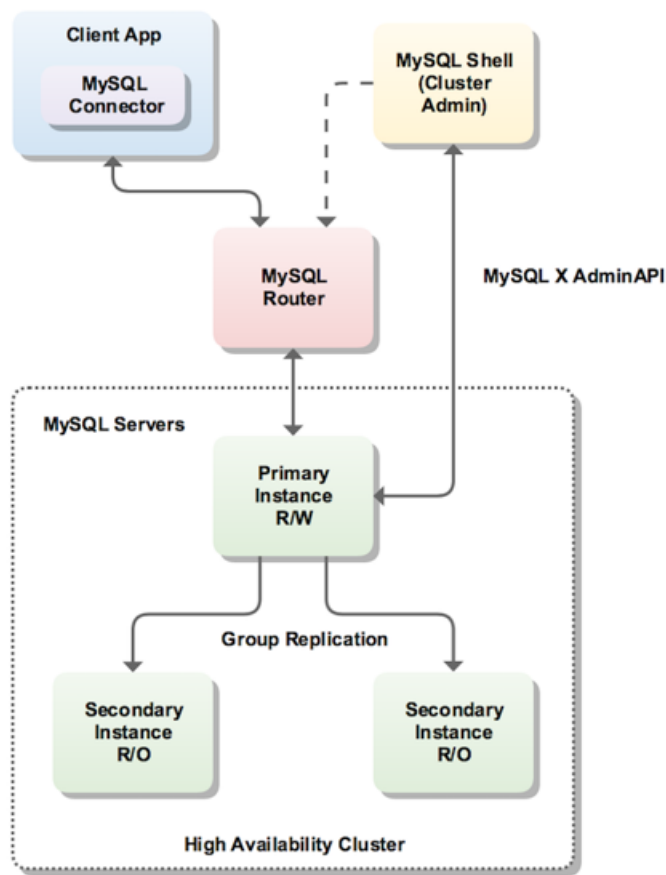
三、MySQL 高可用架构

MySQL InnoDB Cluster

完整的数据库层高可用解决方案

MySQL InnoDB Cluster是一个高可用的框架，它由下面这几个组件构成：

1. **MySQL Group Replication**：提供DB的扩展、自动故障转移
2. **MySQL Router**：轻量级中间件，提供应用程序连接目标的故障转移
3. **MySQL Shell**：新的MySQL客户端，多种接口模式。可以设置群组复制及Router



三、MySQL 高可用架构

MySQL InnoDB Cluster

MySQL Shell

MySQL Shell是MySQL团队打造的一个统一的客户端，它可以对MySQL执行数据操作和管理。它支持通过JavaScript, Python, SQL对关系型数据模式和文档型数据模式进行操作。使用它可以轻松配置管理 InnoDB Cluster。

MySQL Router

MySQL Router是一个轻量级的中间件，可以提供负载均衡和应用连接的故障转移。它是MySQL团队为MGR量身打造的，通过使用Router 和 Shell，用户可以利用MGR实现完整的数据库层的解决方案。如果您在使用MGR，请一定配合使用Router和Shell，您可以理解为它们是为MGR而生的，会配合MySQL的开发路线图发展的工具。

三、MySQL 高可用架构

orchestrator

一款MySQL高可用和复制拓扑管理工具，支持复制拓扑结构的调整，自动故障转移和手动主从切换等。后端数据库用MySQL或SQLite存储元数据，并提供Web界面展示MySQL复制的拓扑关系及状态，通过Web可更改MySQL实例的复制关系和部分配置信息，同时也提供命令行和API接口，方便运维管理。

特点：

1. 自动发现MySQL的复制拓扑，并且在web上展示；
2. 重构复制关系，可以在web进行拖图来进行复制关系变更；
3. 检测主异常，并可以自动或手动恢复，通过Hooks进行自定义脚本；
4. 支持命令行和web界面管理复制。

三、MySQL 高可用架构

orchestrator

两种部署方式

orchestrator/raft:

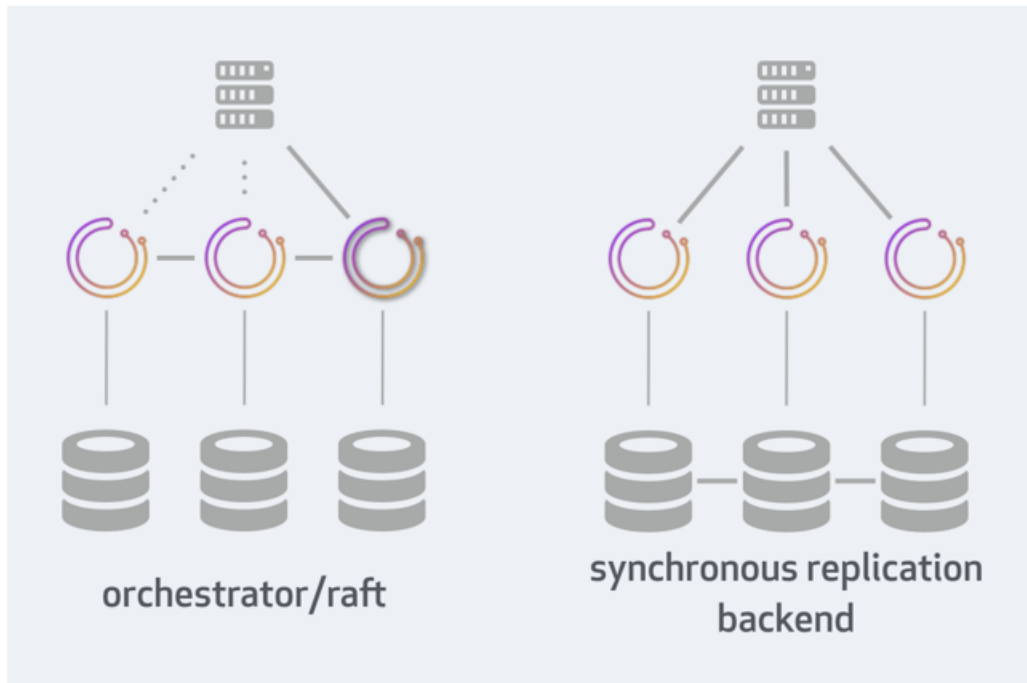
1. 数据一致性由orchestrator的raft协议保证
2. 数据库之间不通信

orchestrator/[galera | xtradb cluster | innodb cluster]:

1. 数据一致性由数据库集群保证
2. 数据库结点之间通信

如果不部署client

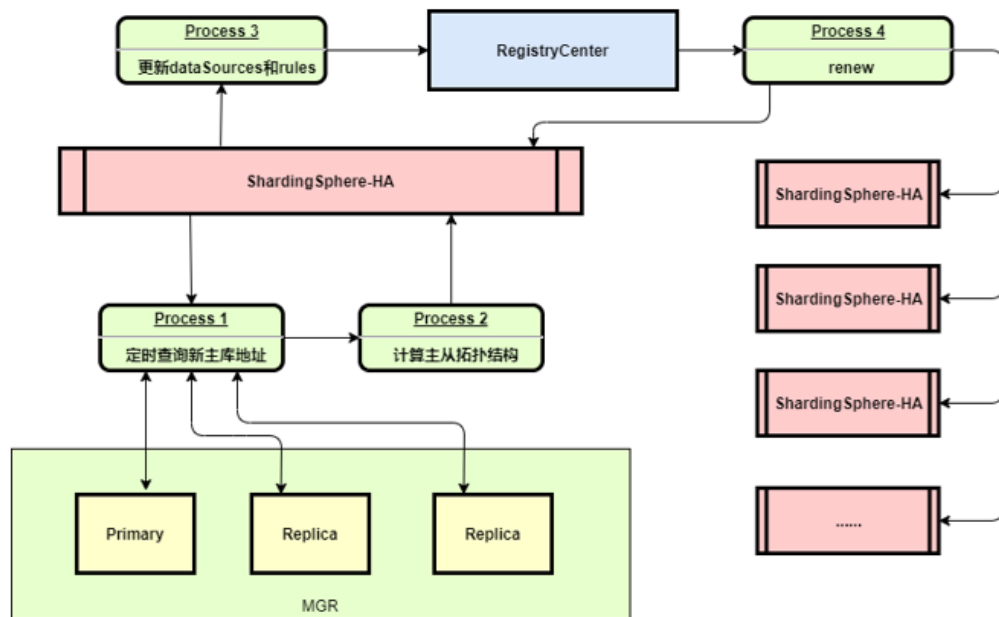
1. 使用HTTP (/api/leader-check) 查询并路由到主节点



三、MySQL 高可用架构

集成MGR

开发ShardingSphere-HA模块，定义新的HA rule，每一个MGR集群定义为一个逻辑dataSource提供给其他rule。这种方式对其他配置方式(Sharding/Encrypt等)无侵入，可以无感知集成HA。



```
rules:
- !HA
  dataSources:
    pr_ds:
      name: pr_ds
      primaryDataSourceName: primary_ds
      replicaDataSourceNames:
        - replica_ds_0
        - replica_ds_1
      readWriteSplit: false
```

三、MySQL 高可用架构

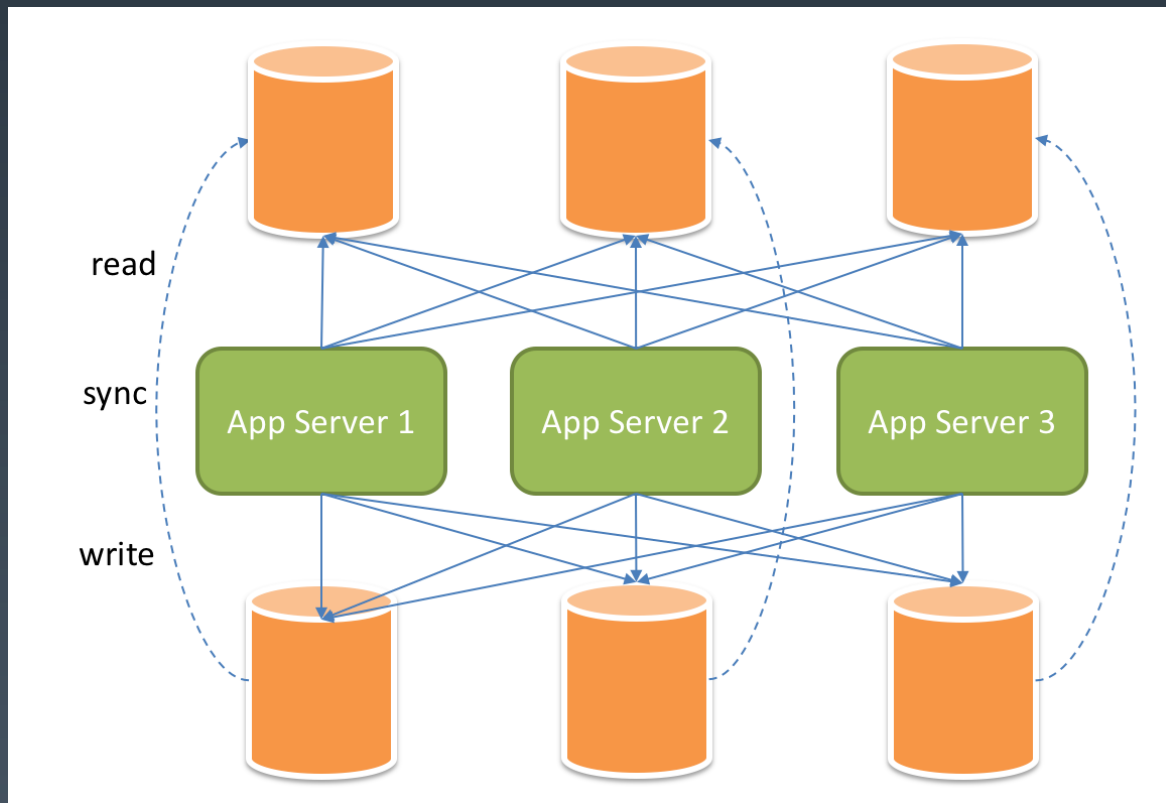
高可用的总结与选择。

应用侧的使用方式：

- 1、动态数据源
- 2、框架与中间件

主从延迟的处理。

主从不同步的处理。



三、MySQL 高可用架构

基于数据复制的读写分离

案例演示。

四、数据库分库分表

主从结构解决了高可用，读扩展，但是单机容量不变，单机写性能无法解决。

提升容量-->分库分表，分布式，多个数据库，作为数据分片的集群提供服务。

降低单个节点的写压力。

提升整个系统的数据容量上限。

四、数据库分库分表

单库单表数据量过大导致的问题

传统的将数据集中存储至单一数据节点的解决方案，在容量、性能、可用性和运维成本这三方面已经难于满足互联网的海量数据场景。我们在单库单表数据量超过一定容量水位的情况下，索引树层级增加，磁盘 IO 也很可能出现压力，会导致很多问题。

从性能方面来说，由于关系型数据库大多采用 B+ 树类型的索引，在数据量超过阈值的情况下，索引深度的增加也将使得磁盘访问的 IO 次数增加，进而导致查询性能的下降；同时，高并发访问请求也使得集中式数据库成为系统的最大瓶颈。

从可用性的方面来讲，服务化的无状态型，能够达到较小成本的随意扩容，这必然导致系统的最终压力都落在数据库之上。而单一的数据节点，或者简单的主从架构，已经越来越难以承担。数据库的可用性，已成为整个系统的关键。从运维成本方面考虑，当一个数据库实例中的数据达到阈值以上，对于 DBA 的运维压力就会增大。数据备份和恢复的时间成本都将随着数据量的大小而愈发不可控。一般来讲，单一数据库实例的数据的阈值在 1TB 之内，是比较合理的范围。

四、数据库分库分表

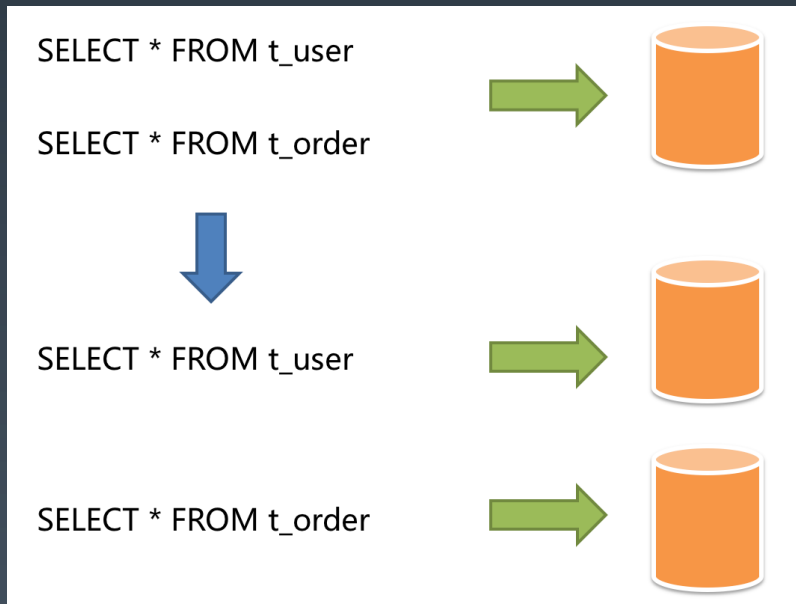
举几个例子：

- 1、无法执行 DDL，比如添加一列，或者增加索引，都会直接影响线上业务，导致长时间的数据库无响应。
- 2、无法备份，与上面类似，备份会自动锁数据库的所有表，然后导出数据，量大了就没法执行了。
- 3、影响性能与稳定性，系统越来越慢，随时可能会出现主库延迟高，主从延迟很高，且不可控，对业务系统有极大的破坏性影响。

四、数据库分库分表

垂直拆分（拆库）：例如拆分所有订单的数据和产品的数据，变成两个独立的库，这种方式对业务系统有极大的影响，因为数据结构本身发生了变化，SQL 和关联关系也必随之发生了改变。原来一个复杂 SQL 直接把一批订单和相关的产品都查了出来，现在这个 SQL 不能用了，得改写 SQL 和程序。先查询订单库数据，拿到这批订单对应的所有产品 id，再根据产品 id 集合去产品库查询所有的产品信息，最后在业务代码里进行组装。

垂直拆分（拆表）：如果单表数据量过大，还可能需要对单表进行拆分。比如一个 200 列的订单主表，拆分成十几个子表：订单表、订单详情表、订单收件信息表、订单支付表、订单产品快照表等等。这个对业务系统的影响有时候可能会大到跟新作一个系统差不多。对于一个高并发的线上生产系统进行改造，就像是给心脑血管做手术，动的愈多，越核心，出现大故障的风险越高。所以，我们一般情况下，尽量少用这种办法。

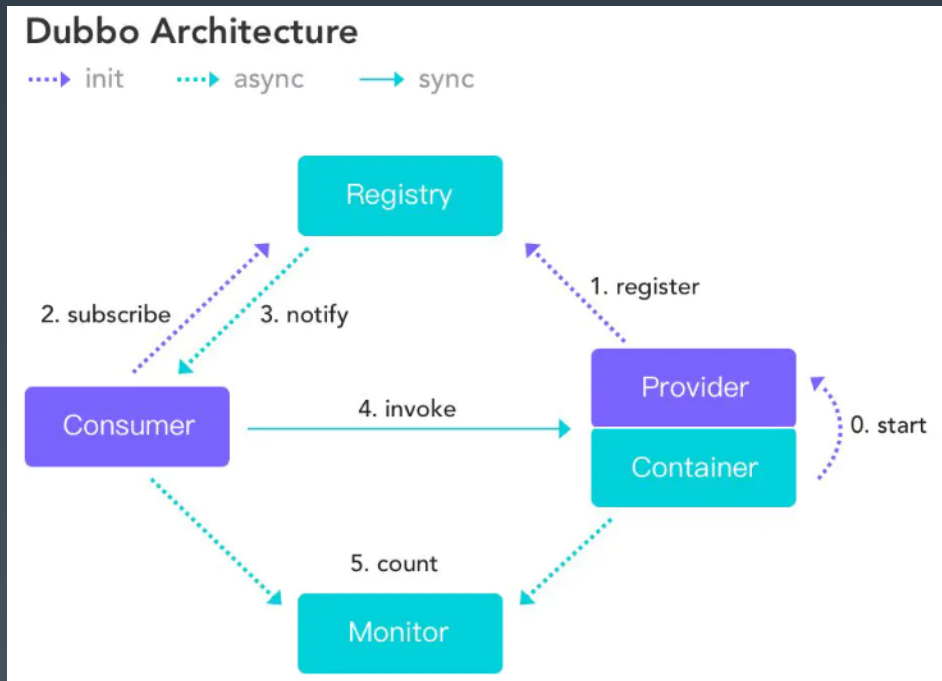


四、数据库分库分表

垂直分库分表 => 分布式服务化 => 微服务架构

以淘宝系统架构为例 说明。

- 1、服务不能复用
- 2、连接数不够

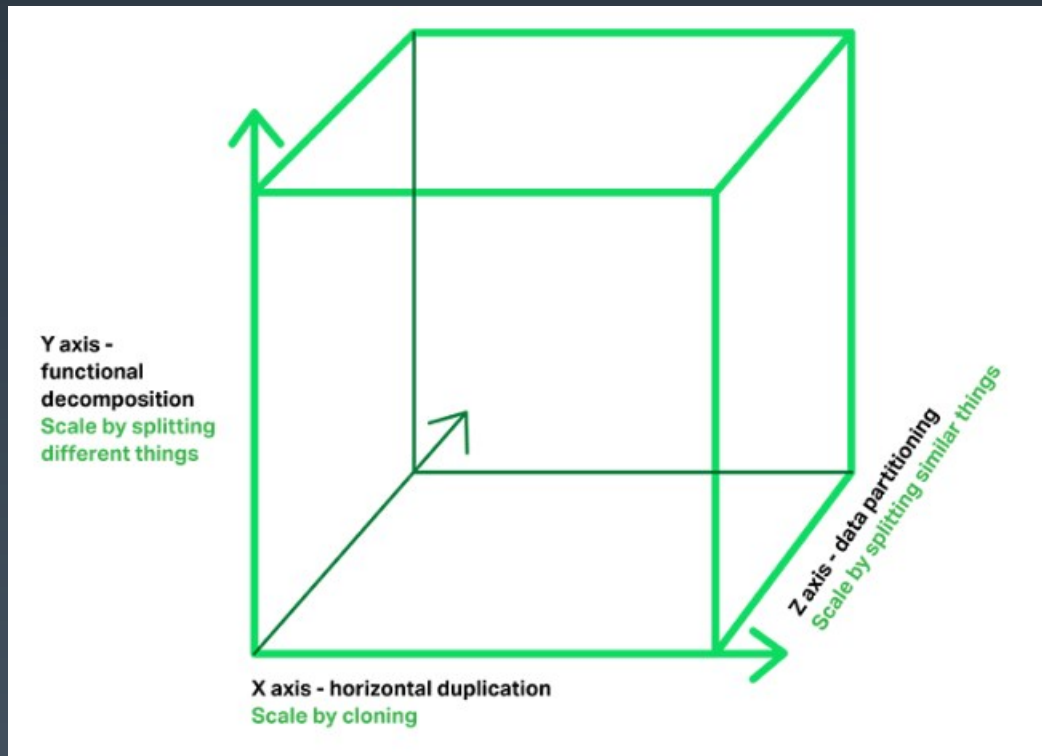


四、数据库分库分表

垂直分库分表 => 分布式服务化 => 微服务架构

微服务架构，拆分业务，拆分数据

容量问题，~系统扩展 ==> 扩展立方体



四、数据库分库分表

水平分库分表

分为，分库、分表、分库分表三类

有什么区别？

四、数据库分库分表

水平拆分（按主键分库分表）：水平拆分就是直接对数据进行分片，有分库和分表两个具体方式，但是都只是降低单个节点数据量，但不改变数据本身的结构。这样对业务系统本身的代码逻辑来说，就不需要做特别大的改动，甚至可以基于一些中间件做到透明。

比如把一个 10 亿条记录的订单单库单表（orderDB 库 t_order 表）。我们按照用户 id 除以 32 取模，把单库拆分成 32 个库 orderDB_00..31；再按订单 id 除以 32 取模，每个库里再拆分成 32 个表 t_order_00..31。这样一共是 1024 个子表，单个表的数据量就只是 10 万条了。

一个查询如果能够直接路由到某个具体的子表，比如 orderDB05.t_order_10，那么查询效率就会高很多。

一般情况下，如果数据本身的读写压力较大，磁盘 IO 已经成为瓶颈，那么分库比分表要好。分库将数据分散到不同的数据库实例，使用不同的磁盘，从而可以并行提升整个集群的并行数据处理能力。相反的情况下，可以尽量多考虑分表，降低单表的数据量，从而减少单表操作的时间，同时也能在单个数据库上使用并行操作多个表来增加处理能力。

四、数据库分库分表

水平分库分表

SELECT * FROM t_user WHERE id=1

SELECT * FROM t_user WHERE id=2



SELECT * FROM t_user WHERE id=1

$id \% 2 = 1$



SELECT * FROM t_user WHERE id=2

$id \% 2 = 0$



四、数据库分库分表

水平分库分表

水平拆分（按时间分库分表）：很多时候，我们的数据是有时间属性的，所以自然可以按照时间维度来拆分。比如当前数据表和历史数据表，甚至按季度，按月，按天来划分不同的表。这样我们按照时间维度来查询数据时，就可以直接定位到当前的这个子表。更详细的分析参考下一个小节。

自定义方式分库分表：指定某些条件的数据进入到某些库或表。

四、数据库分库分表

垂直与水平对比，优缺点分析：

1、优点：

2、问题：

四、数据库分库分表

水平分库分表

为什么有些 DBA 不建议分表，只建议分库？

为什么互联网公司，都只用简单 SQL 和 MyBatis？

四、数据库分库分表

水平分库分表

使用限制：

- 1、复杂 SQL 的处理问题
- 2、性能问题
- 3、业务侵入性问题
- 4、数据迁移问题

四、数据库分库分表

1.水平分库分表

案例演示。

四、数据库分库分表

2.数据动态迁移

案例演示。

四、数据库分库分表

通过分类处理提升数据管理能力

随着我们对业务系统、对数据本身的进一步了解，我们就会发现，很多数据对质量的要求是不同的。

比如，订单数据，肯定一致性要求最高，不能丢数据。而日志数据和一些计算的中间数据，我们则是可以不要那么高的一致性，丢了不要了，或者从别的地方找回来。

同样地，我们对于同样一张表里的订单数据，也可以采用不同策略，无效订单如果比较多，我们可以定期的清除或者转移（一些交易系统里有 80% 以上是的机器下单然后取消的无意义订单，没有人会去查询它，所以可以清理）。

四、数据库分库分表

如果没有无效订单，那么我们也可以考虑：

1. 最近一周下单但是未支付的订单，被查询和支付的可能性较大，再长时间的订单，我们可以直接取消掉。
2. 最近 3 个月下单的数据，被在线重复查询和系统统计的可能性最大。
3. 超过 3 个月、3 年以内的数据，查询的可能性非常小，我们可以不提供在线查询。
4. 3 年以上的数据，我们可以直接不提供任何方式的查询。

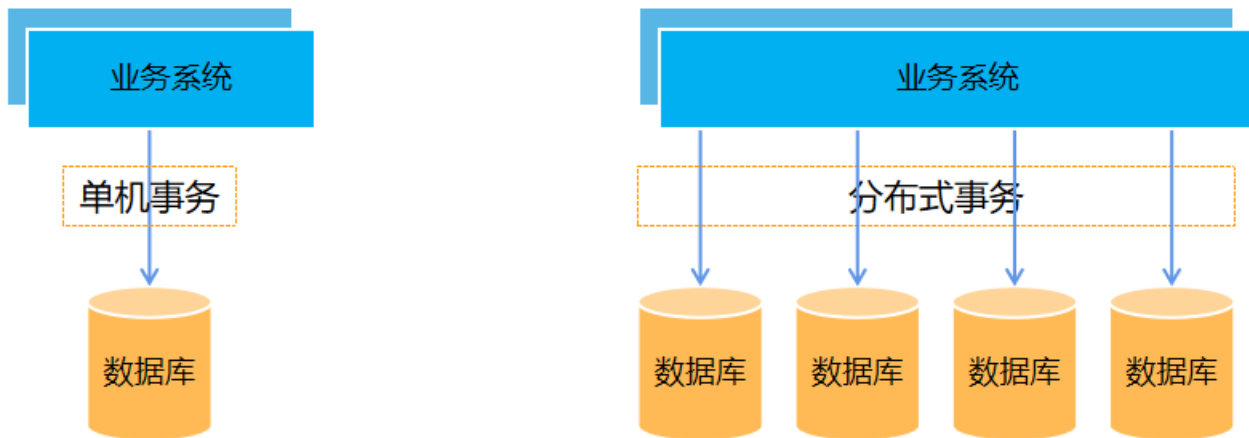
这样的话，我们就可以采取一定的手段去优化系统：

1. 定义一周内下单但未支付的数据为热数据，同时放到数据库和内存；
2. 定义三个月内的数据为温数据，放到数据库，提供正常的查询操作；
3. 定义 3 个月到 3 年的数据，为冷数据，从数据库删除，归档到一些便宜的磁盘，用压缩的方式（比如 MySQL 的 tokuDB 引擎，可以压缩到几十分之一）存储，用户需要邮件或者提交工单来查询，我们导出后发给用户；
4. 定义 3 年以上的数据为冰数据，备份到磁带之类的介质上，不提供任何查询操作。

我们可以看到，上面都是针对一些具体场景的 case，来分析和给出解决办法。我们知道没有能解决一切问题的银弹，那么通过在各种不同的场景下，都对现有的技术和手段进行一些补充，我们就会逐渐得到一个复杂的技术体系。

五、BASE 分布式事务

随着互联网、金融等行业的快速发展，业务越来越复杂，一个完整的业务往往需要调用多个子业务或服务，随着业务的不断增多，涉及的服务及数据也越来越多，越来越复杂。传统的系统难以支撑，出现了应用和数据库等的分布式系统。分布式系统又带来了数据一致性的问题，从而产生了分布式事务。



五、BASE 分布式事务

本地事务 -> XA(2PC) -> BASE

如果将实现了 ACID 的事务要素的事务称为刚性事务的话，那么基于 BASE 事务要素的事务则称为柔性事务。BASE 是基本可用、柔性状态和最终一致性这三个要素的缩写。

基本可用 (Basically Available) 保证分布式事务参与方不一定同时在线。

柔性状态 (Soft state) 则允许系统状态更新有一定的延时，这个延时对客户来说不一定能够察觉。

而最终一致性 (Eventually consistent) 通常是通过消息传递的方式保证系统的最终一致性。

在 ACID 事务中对隔离性的要求很高，在事务执行过程中，必须将所有的资源锁定。柔性事务的理念则是通过业务逻辑将互斥锁操作从资源层面上移至业务层面。通过放宽对强一致性要求，来换取系统吞吐量的提升。

基于 ACID 的强一致性事务和基于 BASE 的最终一致性事务都不是银弹，只有在最适合的场景中才能发挥它们的最大长处。可通过下表详细对比它们之间的区别，以帮助开发者进行技术选型。

五、BASE 分布式事务

	本地事务	两 (三) 阶段事务	柔性事务
业务改造	无	无	实现相关接口
一致性	不支持	支持	最终一致
隔离性	不支持	支持	业务方保证
并发性能	无影响	严重衰退	略微衰退
适合场景	业务方处理不一致	短事务 & 低并发	长事务 & 高并发

五、BASE 分布式事务

柔性事务常见模式

1、TCC

通过手动补偿处理

2、AT

通过自动补偿处理

五、BASE 分布式事务

Seata-TCC 柔性事务

Seata 是阿里集团和蚂蚁金服联合打造的分布式事务框架。其 AT 事务的目标是在微服务架构下，提供增量的事务 ACID 语义，让开发者像使用本地事务一样，使用分布式事务，核心理念同 Apache ShardingSphere 一脉相承。

Seata AT 事务模型包含 TM (事务管理器)，RM (资源管理器) 和 TC (事务协调器)。TC 是一个独立部署的服务，TM 和 RM 以 jar 包的方式同业务应用一同部署，它们同 TC 建立长连接，在整个事务生命周期内，保持远程通信。TM 是全局事务的发起方，负责全局事务的开启，提交和回滚。RM 是全局事务的参与者，负责分支事务的执行结果上报，并且通过 TC 的协调进行分支事务的提交和回滚。

五、BASE 分布式事务

Seata 管理的分布式事务的典型生命周期：

TM 要求 TC 开始一个全新的全局事务。TC 生成一个代表该全局事务的 XID。

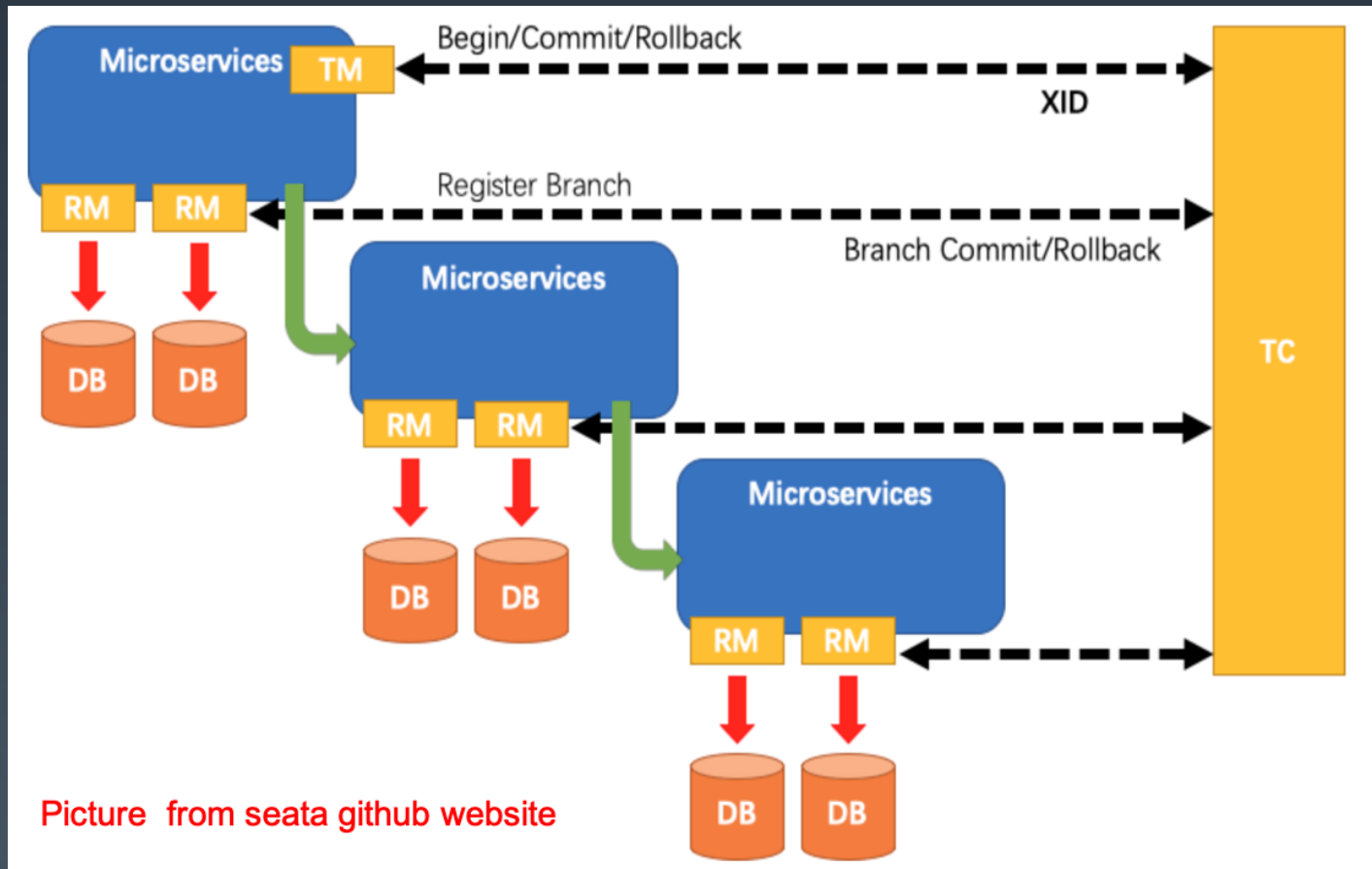
XID 贯穿于微服务的整个调用链。

作为该 XID 对应到的 TC 下的全局事务的一部分，RM 注册本地事务。

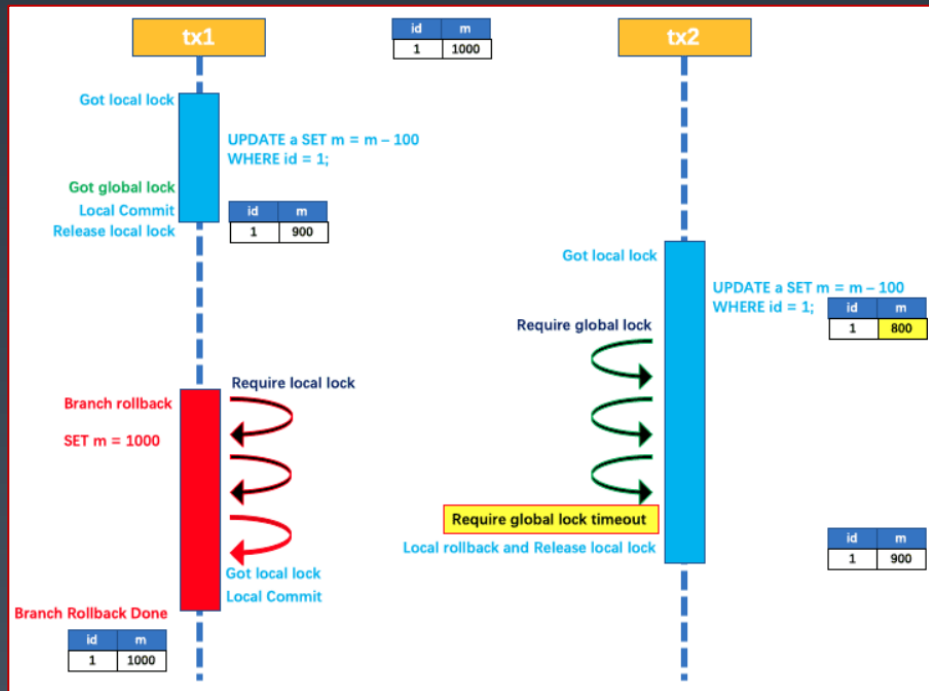
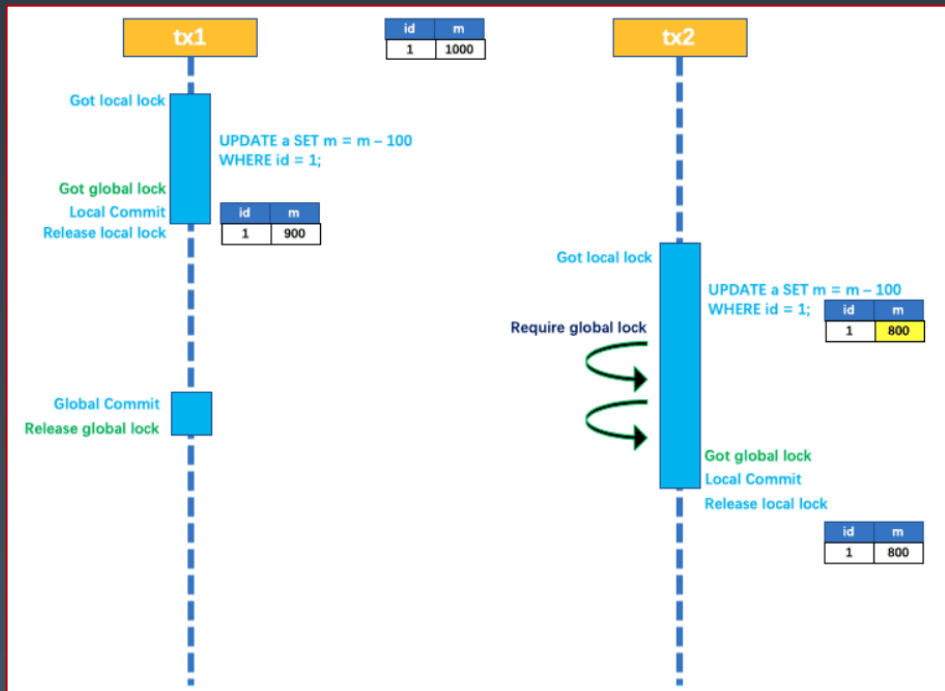
TM 要求 TC 提交或回滚 XID 对应的全局事务。

TC 驱动 XID 对应的全局事务下的所有分支事务完成提交或回滚。

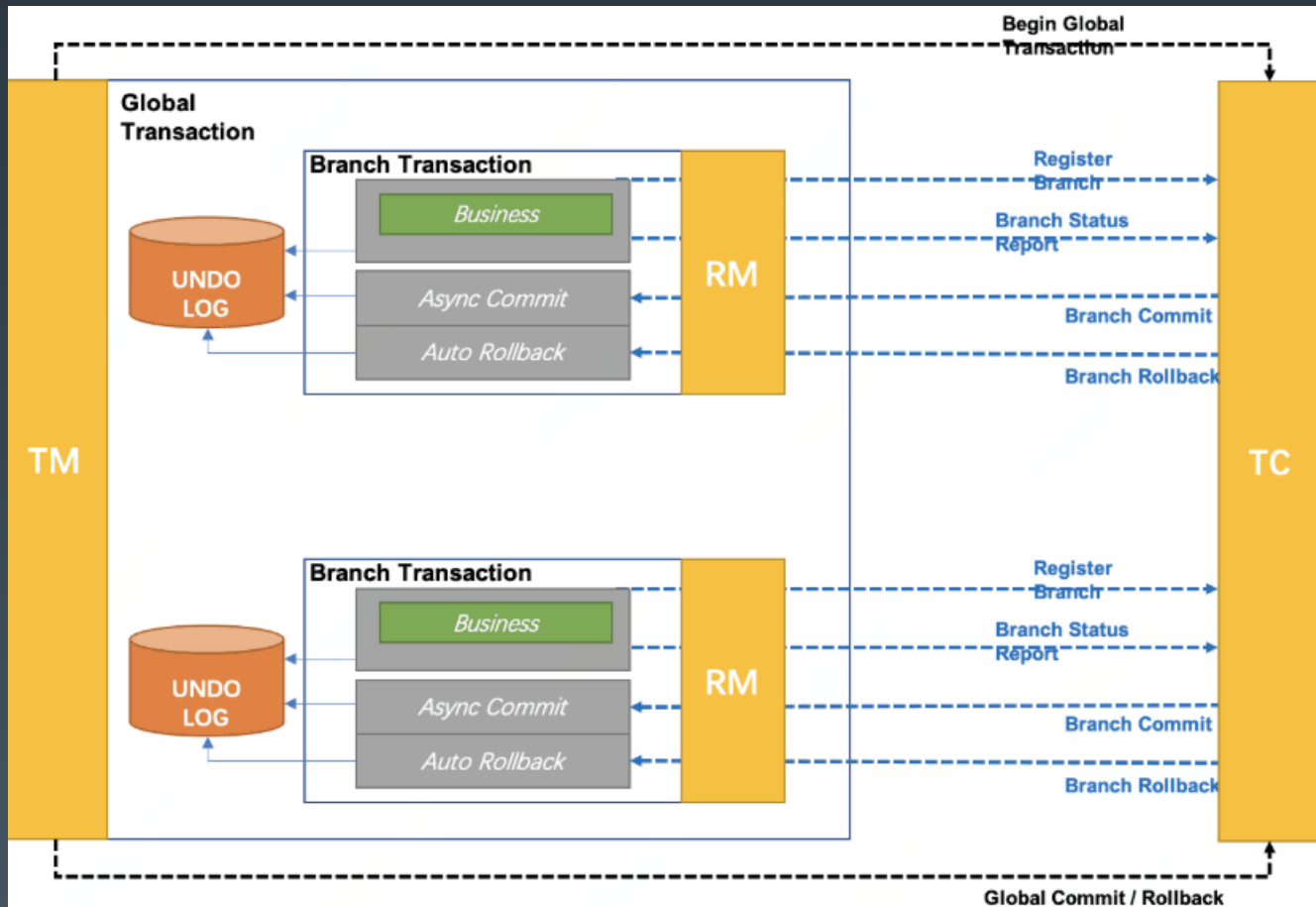
五、BASE 分布式事务



五、BASE 分布式事务



五、BASE 分布式事务



五、BASE 分布式事务

TC 与 RM, TM 维持 netty 长连接通信, 其本身可以部署成一个集群(需要保证 TM, RM 连接到同一个节点, 目前 seata 并没有支持, 如果发现连接不是同一个节点 TC, 只能通过定时任务来通知)

TC 存储全局事务信息, 分支事务信息, 分支 lock 信息

global_table	
xid	varchar(128)
transaction_id	bigint(20)
status	tinyint(4)
application_id	varchar(32)
transaction_service_group	varchar(32)
transaction_name	varchar(128)
timeout	int(11)
begin_time	bigint(20)
application_data	varchar(2000)
gmt_create	datetime
gmt_modified	datetime

branch_table	
branch_id	bigint(20)
xid	varchar(128)
transaction_id	bigint(20)
resource_group_id	varchar(32)
resource_id	varchar(256)
branch_type	varchar(8)
status	tinyint(4)
client_id	varchar(64)
application_data	varchar(2000)
gmt_create	datetime(6)
gmt_modified	datetime(6)

lock_table	
row_key	varchar(128)
xid	varchar(96)
transaction_id	bigint(20)
branch_id	bigint(20)
resource_id	varchar(256)
table_name	varchar(32)
pk	varchar(36)
gmt_create	datetime
gmt_modified	datetime

其通过 XID 来进行关联, 在进行一次分布式事务的时候, 有一套全局事务信息, 多条分支事务信息, 以及多条分支 lock 信息

五、BASE 分布式事务

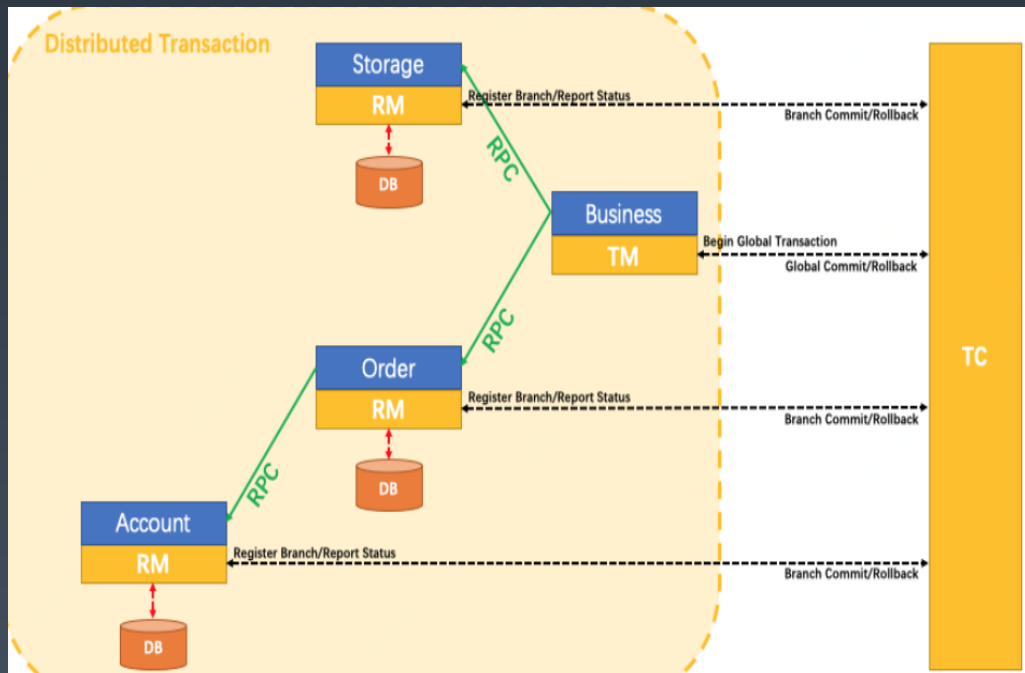
整体机制：

一阶段：执行业务 sql，同时生成 sql 方向日志存储在 undo 日志表中

二阶段提交：如果无异常，由 TM 向 TC 发送事务完成请求，TC 下发提交命令给 RM，完成提日志的清理，同时向 TC 上报分支完成

二阶段回滚：有异常，TM 向 TC 发送事务回滚请求

TC 下发回滚命令到各 RM，RM 通过 undo 日志表获取回滚记录，生成反向 SQL 语句，执行后，向 TC 上报分支回滚



写隔离：分支事务提交前，都通过 lock 表获取全局锁，如果获取不到就报错，回滚本地事务，释放本地锁

读隔离：先去 lock 表查看，当前的语句是否有全局锁，如果有通过 Selector for update 语句代理

五、BASE 分布式事务

Hmily 是一个高性能分布式事务框架，开源于2017年，目前有2800个 Star，基于 TCC 原理实现，使用 Java 语言开发（JDK1.8+），天然支持 Dubbo、SpringCloud、Motan 等微服务框架的分布式事务。

支持嵌套事务(Nested transaction support)等复杂场景

支持 RPC 事务恢复，超时异常恢复等，具有高稳定性

基于异步 Confirm 和 Cancel 设计，相比其他方式具有更高性能

基于 SPI 和 API 机制设计，定制性强，具有高扩展性

本地事务的多种存储支持：redis/mongodb/zookeeper/file/mysql

事务日志的多种序列化支持：java/hessian/kryo/protostuff

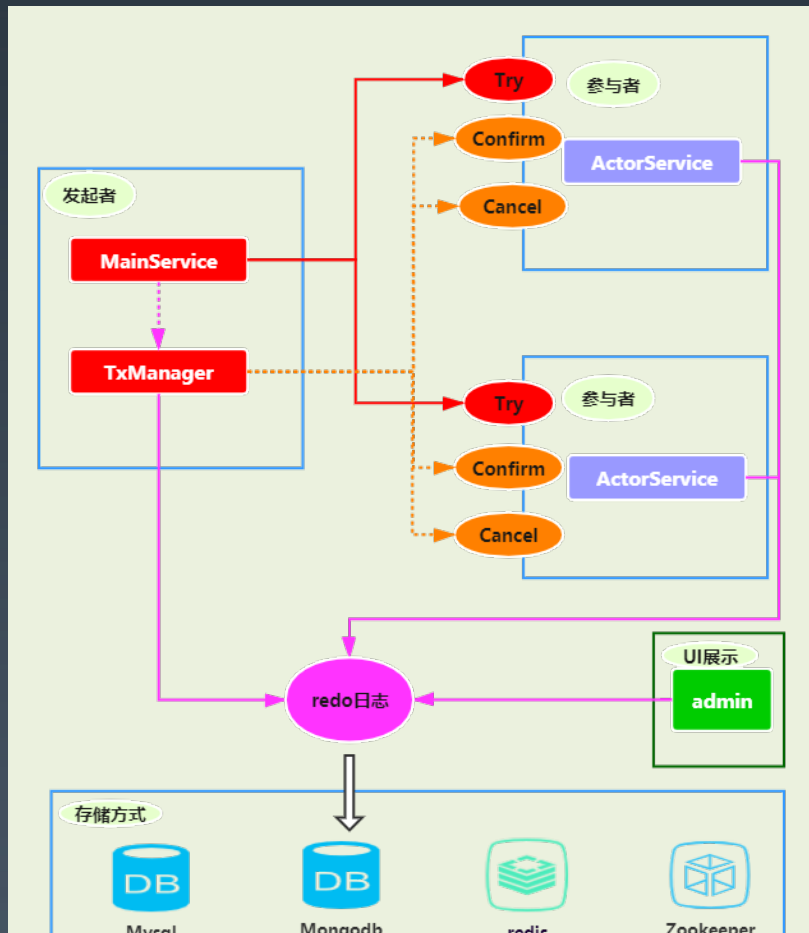
基于高性能组件 disruptor 的异步日志性能良好

实现了 SpringBoot-Starter，开箱即用，集成方便

采用 Aspect AOP 切面思想与 Spring 无缝集成，天然支持集群

实现了基于 VUE 的 UI 界面，方便监控和管理

五、BASE 分布式事务



MainService : 事务发起者 (业务服务)

TxManage : 事务协调者

ActorService : 事务参与者 (多个业务服务)

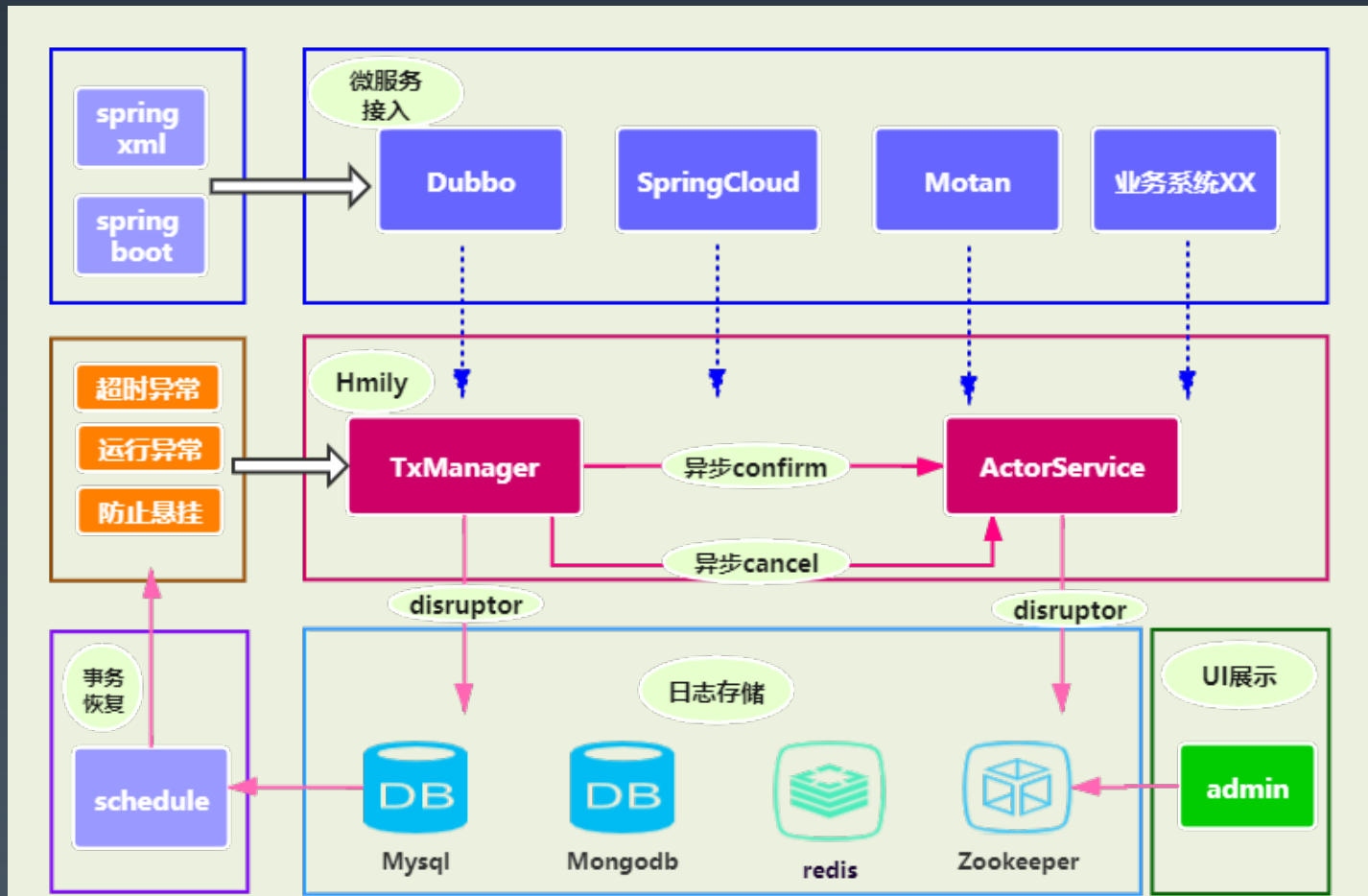
Try : 事务执行

Confirm : 事务确认

Cancel : 事务回滚

Redo 日志 : 可以选择任意一种进行存储

五、BASE 分布式事务



五、BASE 分布式事务

ShardingSphere 对分布式事务的支持

由于应用的场景不同，需要开发者能够合理的在性能与功能之间权衡各种分布式事务。

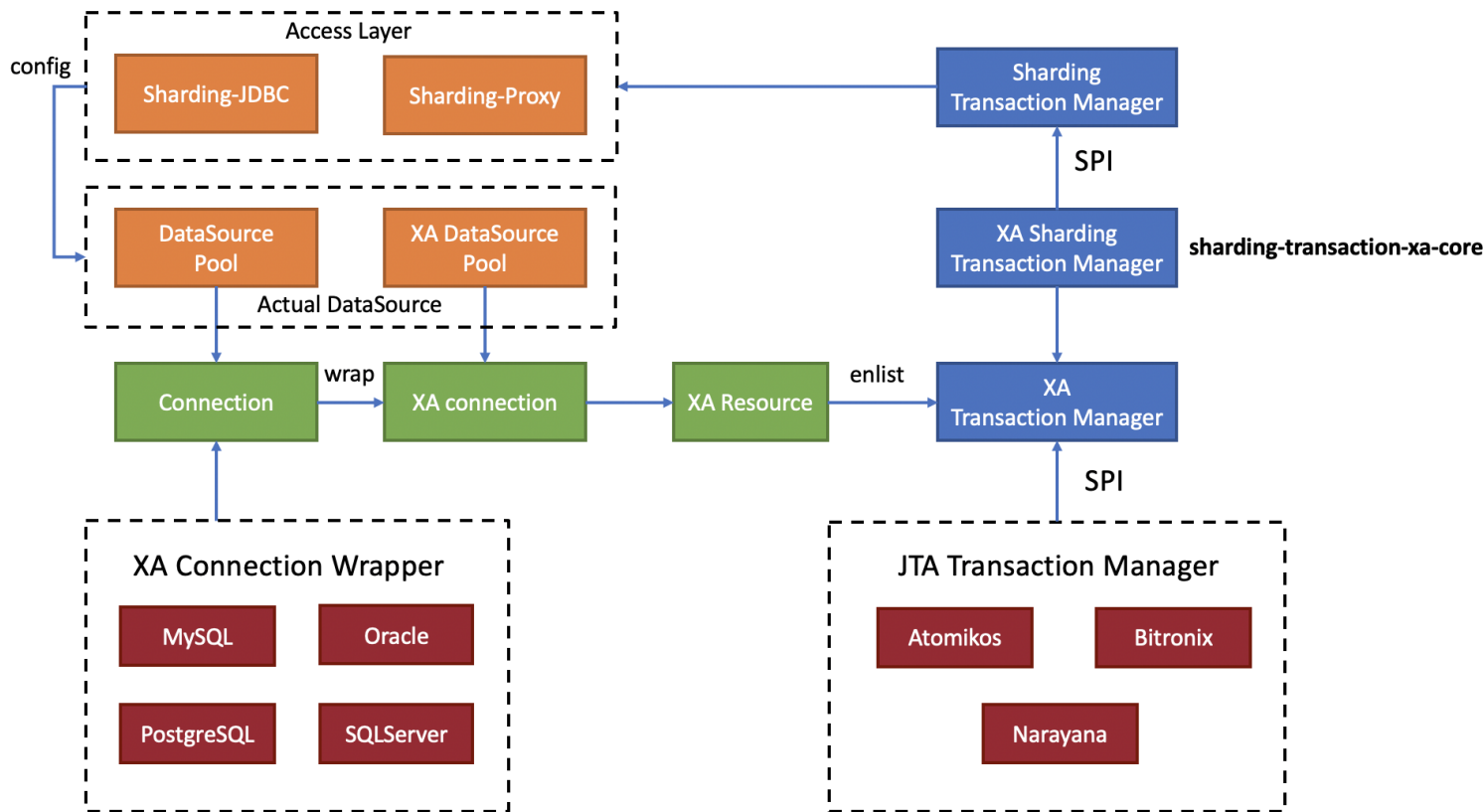
强一致的事务与柔性事务的 API 和功能并不完全相同，在它们之间并不能做到自由的透明切换。在开发决策阶段，就不得不在强一致的事务和柔性事务之间抉择，使得设计和开发成本被大幅增加。

基于XA的强一致事务使用相对简单，但是无法很好的应对互联网的高并发或复杂系统的长事务场景；柔性事务则需要开发者对应用进行改造，接入成本非常高，并且需要开发者自行实现资源锁定和反向补偿。

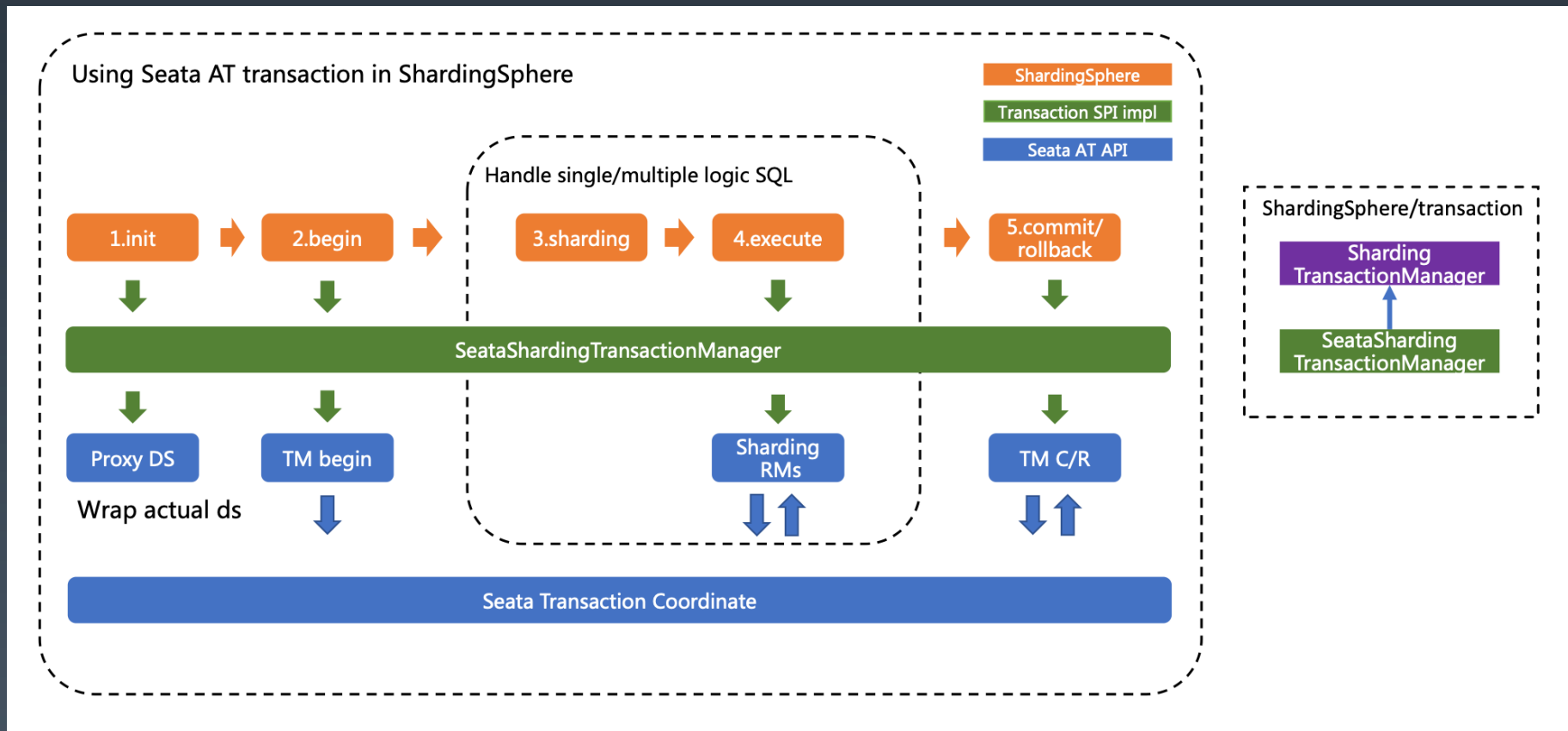
目标

整合现有的成熟事务方案，为本地事务、两阶段事务和柔性事务提供统一的分布式事务接口，并弥补当前方案的不足，提供一站式的分布式事务解决方案是 Apache ShardingSphere 分布式事务模块的主要设计目标。

五、BASE 分布式事务



五、BASE 分布式事务



七、数据库框架与中间件

Java 框架层面：

- TDDL
- Apache ShardingSphere-JDBC

中间件层面：

- Drds
- Apache ShardingSphere-Proxy
- MyCat/DBLE
- Cobar
- Vitness
- KingShard

七、数据库框架与中间件

数据迁移工具：

- 1、canal+outer , DataX
- 2、kettle、ETL 相关
- 3、Apache ShardingSphere-Scaling

闭源的有不少商业化 DTS 服务

七、数据库框架与中间件

框架和中间件的比较与选择。

开源，闭源？

成熟度，定制化？

性能与稳定性？

THANKS! |  极客大学