42 | 电商系统的分布式事务调优

2019-08-27 刘超

你好，我是刘超。

今天的分享也是从案例开始。我们团队曾经遇到过一个非常严重的线上事故，在一次DBA完成单台数据库线上补丁后，系统偶尔会出现异常报警，我们的开发工程师很快就定位到了数据库异常问题。

具体情况是这样的，当玩家购买道具之后，扣除通宝时出现了异常。这种异常在正常情况下发生之后，应该是整个购买操作都需要撤销，然而这次异常的严重性就是在于玩家购买道具成功后，没有扣除通宝。

究其原因是由于购买的道具更新的是游戏数据库，而通宝是在用户账户中心数据库，在一次购买道具时，存在同时操作两个数据库的情况，属于一种分布式事务。而我们的工程师在完成玩家获得道具和扣除余额的操作时，没有做到事务的一致性，即在扣除通宝失败时，应该回滚已经购买的游戏道具。

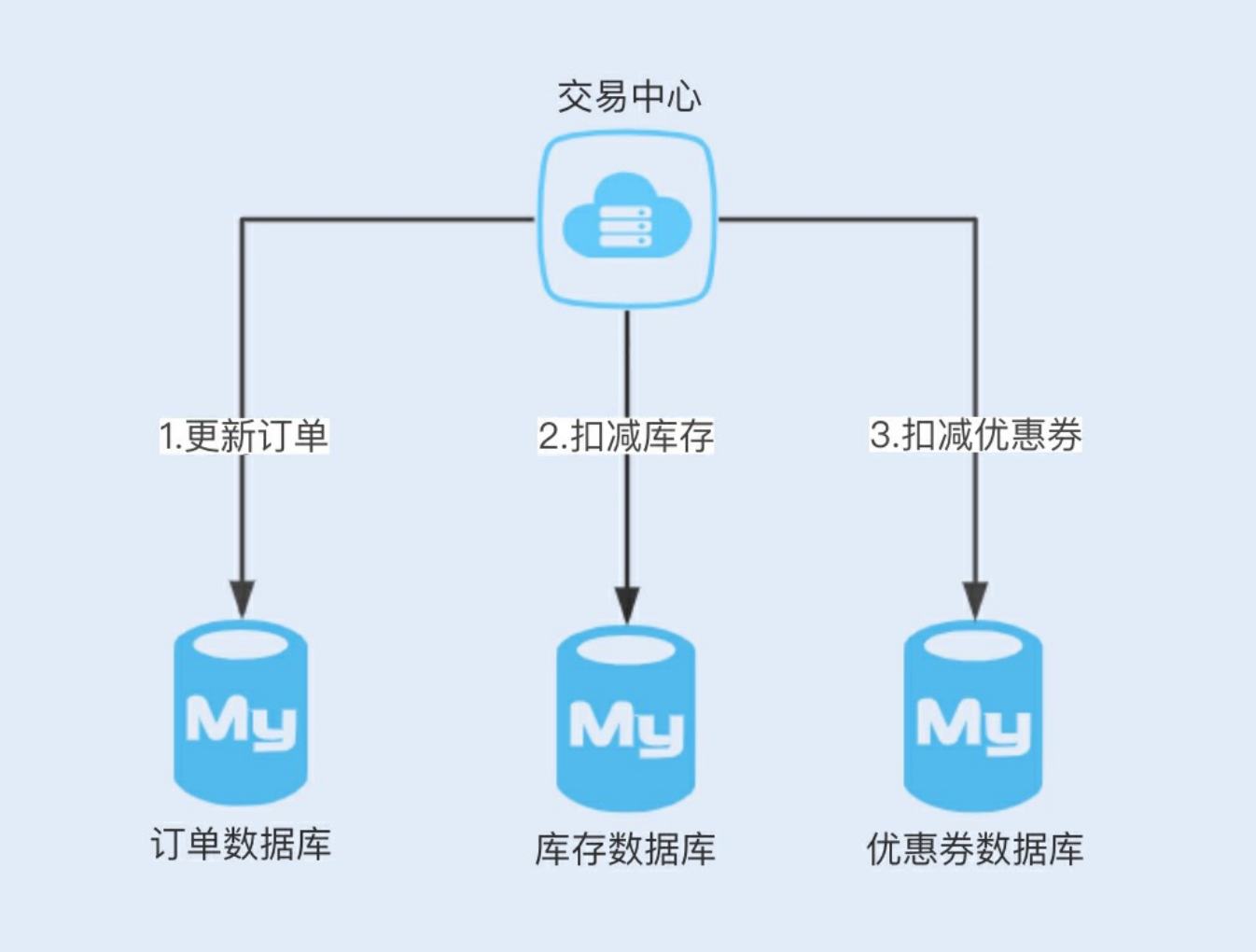
从这个案例中，我想你应该意识到了分布式事务的重要性。

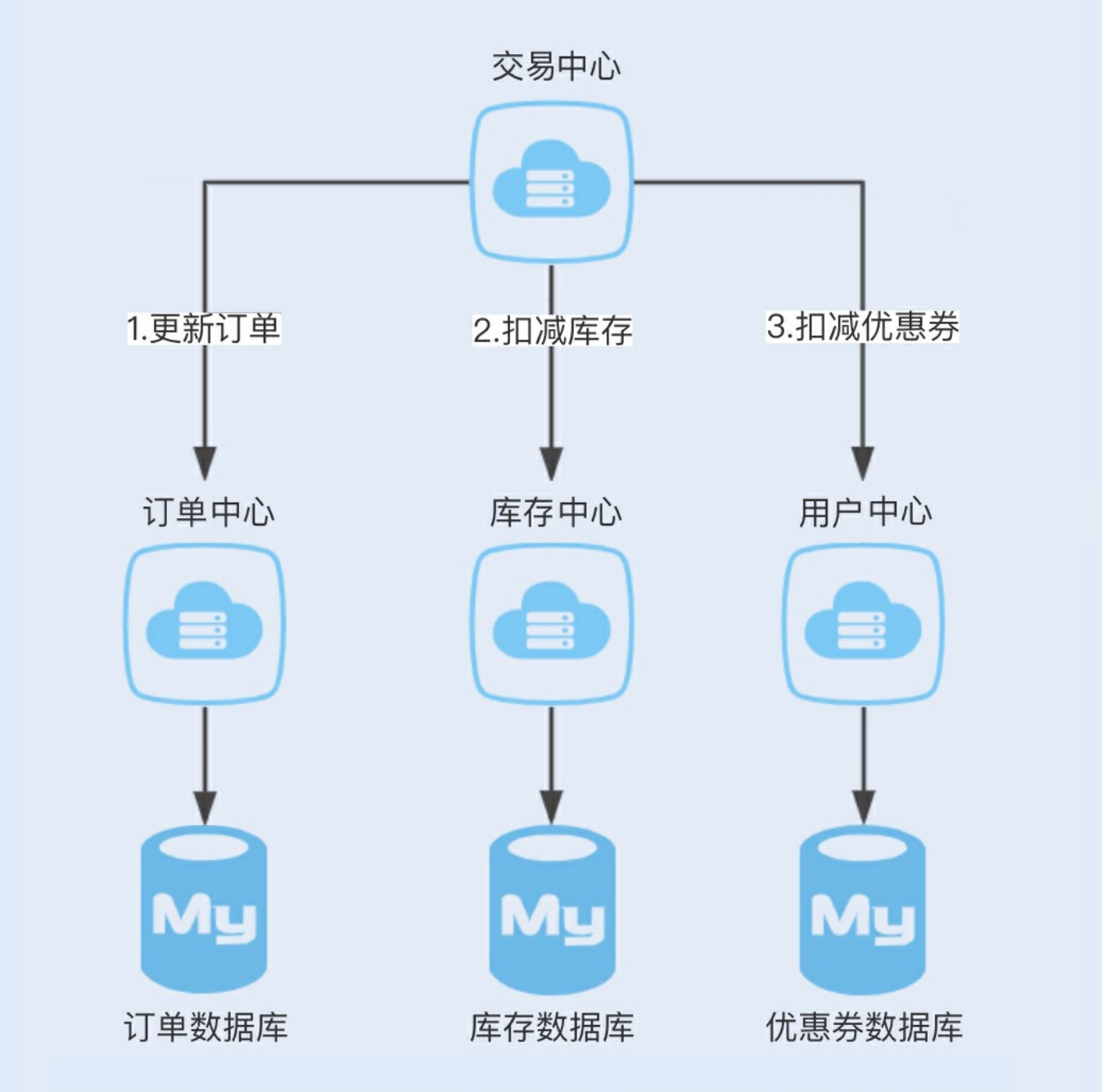
如今，大部分公司的服务基本都实现了微服务化，首先是业务需求，为了解耦业务；其次是为了减少业务与业务之间的相互影响。

电商系统亦是如此，大部分公司的电商系统都是分为了不同服务模块，例如商品模块、订单模块、库存模块等等。事实上，分解服务是一把双刃剑，可以带来一些开发、性能以及运维上的优

势，但同时也会增加业务开发的逻辑复杂度。其中最为突出的就是分布式事务了。

通常，存在分布式事务的服务架构部署有以下两种：同服务不同数据库，不同服务不同数据库。我们以商城为例，用图示说明下这两种部署：





通常，我们都是基于第二种架构部署实现的，那我们应该如何实现在这种服务架构下，有关订单提交业务的分布式事务呢？

分布式事务解决方案

我们讲过，在单个数据库的情况下，数据事务操作具有ACID四个特性，但如果在一个事务中操作多个数据库，则无法使用数据库事务来保证一致性。

也就是说，当两个数据库操作数据时，可能存在一个数据库操作成功，而另一个数据库操作失败的情况，我们无法通过单个数据库事务来回滚两个数据操作。

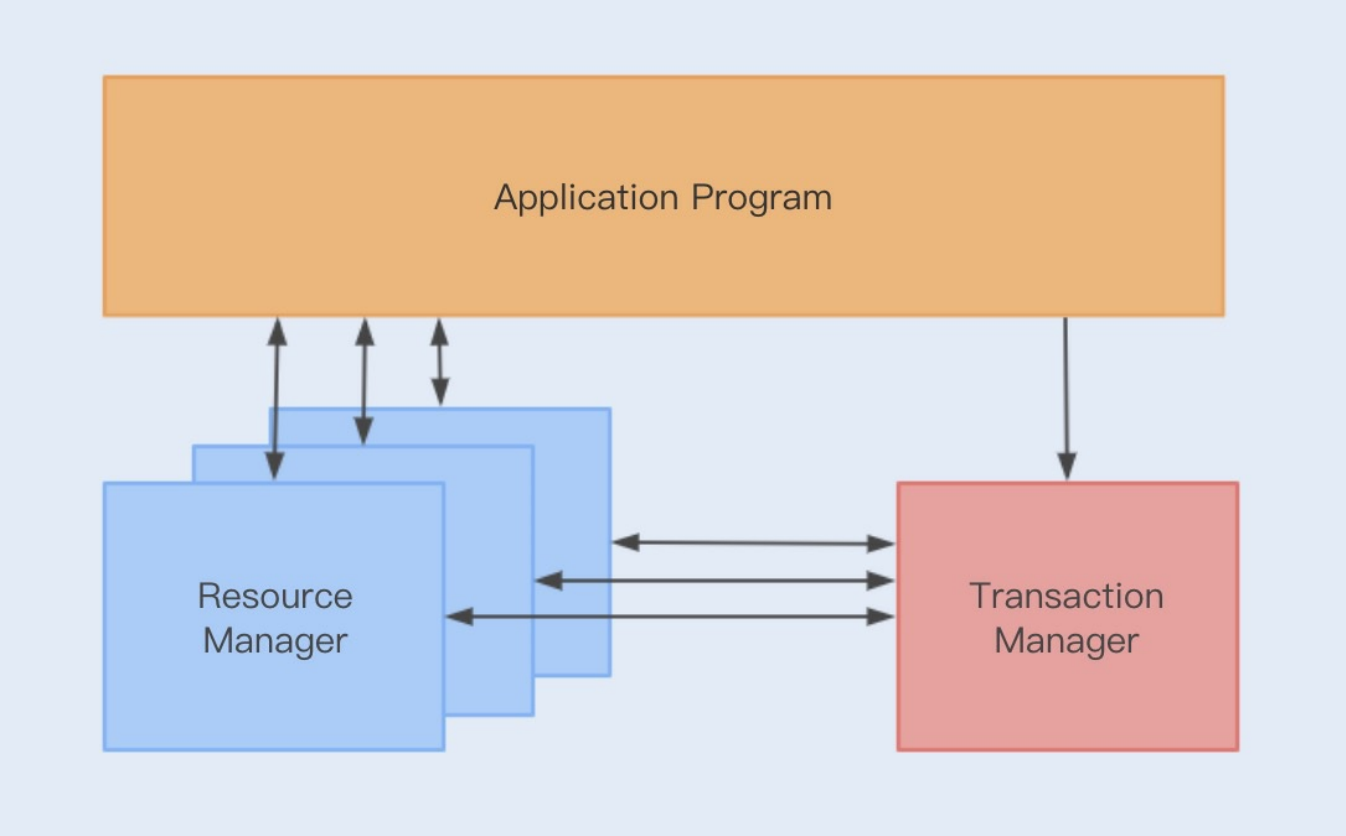
而分布式事务就是为了解决在同一个事务下，不同节点的数据库操作数据不一致的问题。在一个事务操作请求多个服务或多个数据库节点时，要么所有请求成功，要么所有请求都失败回滚回去。通常，分布式事务的实现有多种方式，例如XA协议实现的二阶提交（2PC）、三阶提交(3PC)，以及TCC补偿性事务。

在了解2PC和3PC之前，我们有必要先来了解下XA协议。XA协议是由X/Open组织提出的一个分布式事务处理规范，目前MySQL中只有InnoDB存储引擎支持XA协议。

1. XA规范

在XA规范之前，存在着一个DTP模型，该模型规范了分布式事务的模型设计。

DTP规范中主要包含了AP、RM、TM三个部分，其中AP是应用程序，是事务发起和结束的地方；RM是资源管理器，主要负责管理每个数据库的连接数据源；TM是事务管理器，负责事务的全局管理，包括事务的生命周期管理和资源的分配协调等。



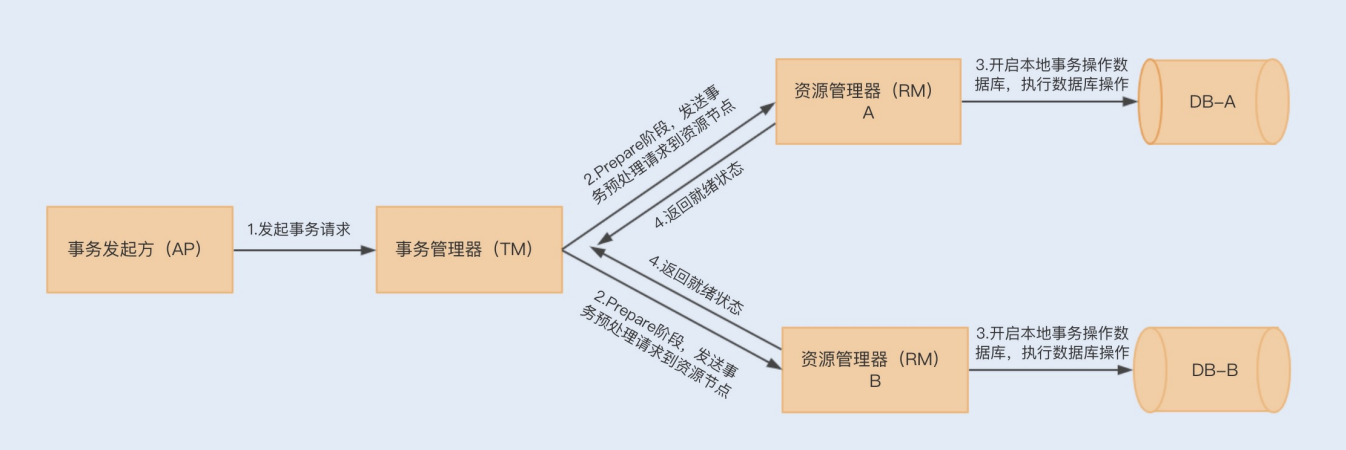
XA则规范了TM与RM之间的通信接口，在TM与多个RM之间形成一个双向通信桥梁，从而在多个数据库资源下保证ACID四个特性。

这里强调一下，JTA是基于XA规范实现的一套Java事务编程接口，是一种两阶段提交事务。我们可以通过源码简单了解下JTA实现的多数据源事务提交。

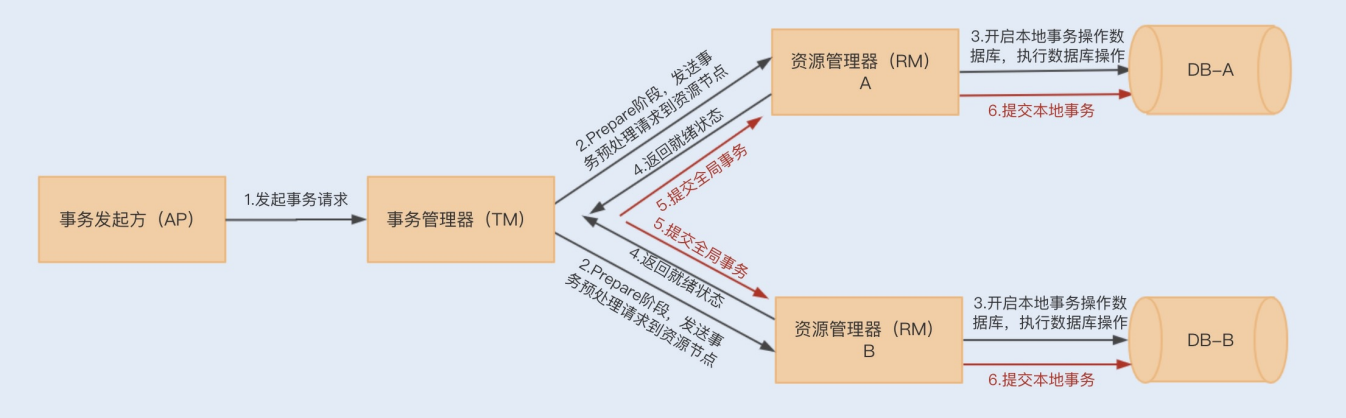
2. 二阶提交和三阶提交

XA规范实现的分布式事务属于二阶提交事务，顾名思义就是通过两个阶段来实现事务的提交。

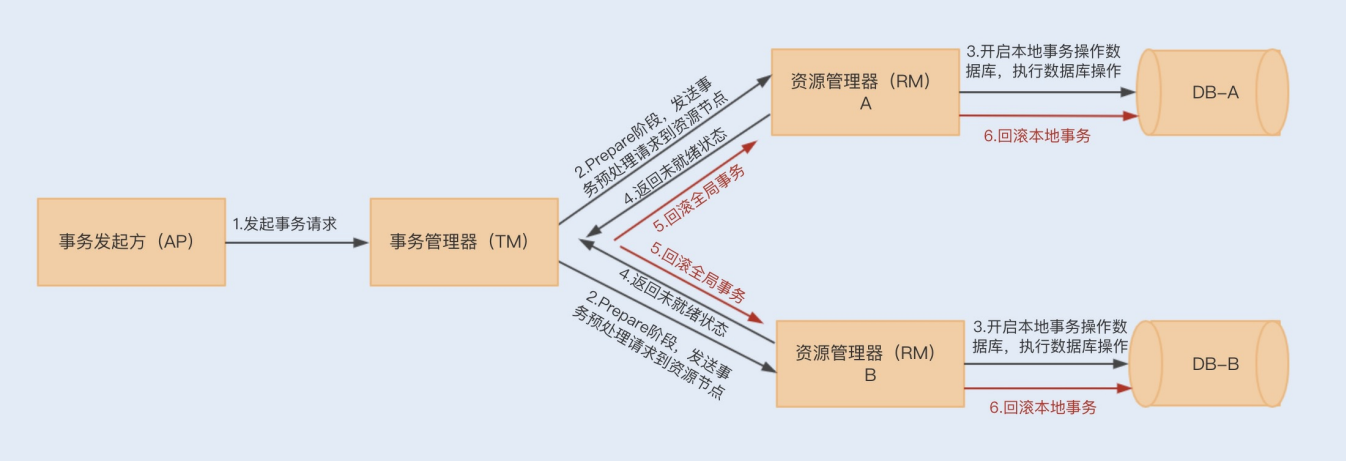
在第一阶段，应用程序向事务管理器（TM）发起事务请求，而事务管理器则会分别向参与的各个资源管理器（RM）发送事务预处理请求（Prepare），此时这些资源管理器会打开本地数据库事务，然后开始执行数据库事务，但执行完成后并不会立刻提交事务，而是向事务管理器返回已就绪（Ready）或未就绪（Not Ready）状态。如果各个参与节点都返回状态了，就会进入第二阶段。



到了第二阶段，如果资源管理器返回的都是就绪状态，事务管理器则会向各个资源管理器发送提交（Commit）通知，资源管理器则会完成本地数据库的事务提交，最终返回提交结果给事务管理器。



在第二阶段中，如果任意资源管理器返回了未就绪状态，此时事务管理器会向所有资源管理器发送事务回滚（Rollback）通知，此时各个资源管理器就会回滚本地数据库事务，释放资源，并返回结果通知。



但事实上，二阶事务提交也存在一些缺陷。

第一，在整个流程中，我们会发现各个资源管理器节点存在阻塞，只有当所有的节点都准备完成之后，事务管理器才会发出进行全局事务提交的通知，这个过程如果很长，则会有很多节点长时间占用资源，从而影响整个节点的性能。

一旦资源管理器挂了，就会出现一直阻塞等待的情况。类似问题，我们可以通过设置事务超时时间来解决。

第二，仍然存在数据不一致的可能性，例如，在最后通知提交全局事务时，由于网络故障，部分节点有可能收不到通知，由于这部分节点没有提交事务，就会导致数据不一致的情况出现。

而三阶事务（3PC）的出现就是为了减少此类问题的发生。

3PC把2PC的准备阶段分为了准备阶段和预处理阶段，在第一阶段只是询问各个资源节点是否可以执行事务，而在第二阶段，所有的节点反馈可以执行事务，才开始执行事务操作，最后在第三阶段执行提交或回滚操作。并且在事务管理器和资源管理器中都引入了超时机制，如果在第三阶段，资源节点一直无法收到来自资源管理器的提交或回滚请求，它就会在超时之后，继续提交事务。

所以3PC可以通过超时机制，避免管理器挂掉所造成的长时间阻塞问题，但其实这样还是无法解决在最后提交全局事务时，由于网络故障无法通知到一些节点的问题，特别是回滚通知，这样会导致事务等待超时从而默认提交。

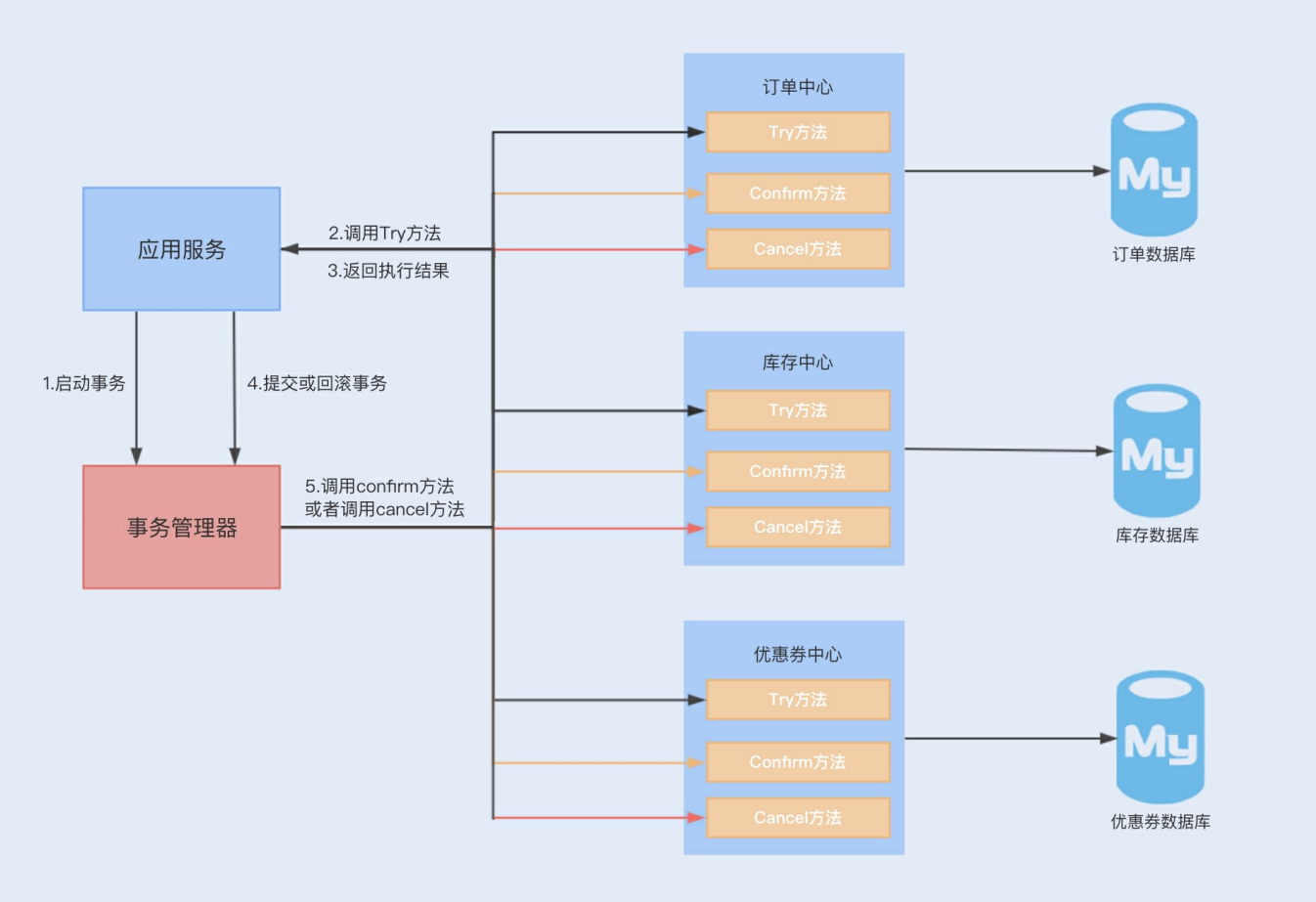
3. 事务补偿机制（TCC）

以上这种基于XA规范实现的事务提交，由于阻塞等性能问题，有着比较明显的低性能、低吞吐的特性。所以在抢购活动中使用该事务，很难满足系统的并发性能。

除了性能问题，JTA只能解决同一服务下操作多数据源的分布式事务问题，换到微服务架构下，可能存在同一个事务操作，分别在不同服务上连接数据源，提交数据库操作。

而TCC正是为了解决以上问题而出现的一种分布式事务解决方案。TCC采用最终一致性的方式实现了一种柔性分布式事务，与XA规范实现的二阶事务不同的是，TCC的实现是基于服务层实现的一种二阶事务提交。

TCC分为三个阶段，即Try、Confirm、Cancel三个阶段。



- Try阶段：主要尝试执行业务，执行各个服务中的Try方法，主要包括预留操作；
- Confirm阶段：确认Try中的各个方法执行成功，然后通过TM调用各个服务的Confirm方法，这个阶段是提交阶段；
- Cancel阶段：当在Try阶段发现其中一个Try方法失败，例如预留资源失败、代码异常等，则会触发TM调用各个服务的Cancel方法，对全局事务进行回滚，取消执行业务。

以上执行只是保证Try阶段执行时成功或失败的提交和回滚操作，你肯定会想到，如果在Confirm和Cancel阶段出现异常情况，那TCC该如何处理呢？此时TCC会不停地重试调用失败的Confirm或Cancel方法，直到成功为止。

但TCC补偿性事务也有比较明显的缺点，那就是对业务的侵入性非常大。

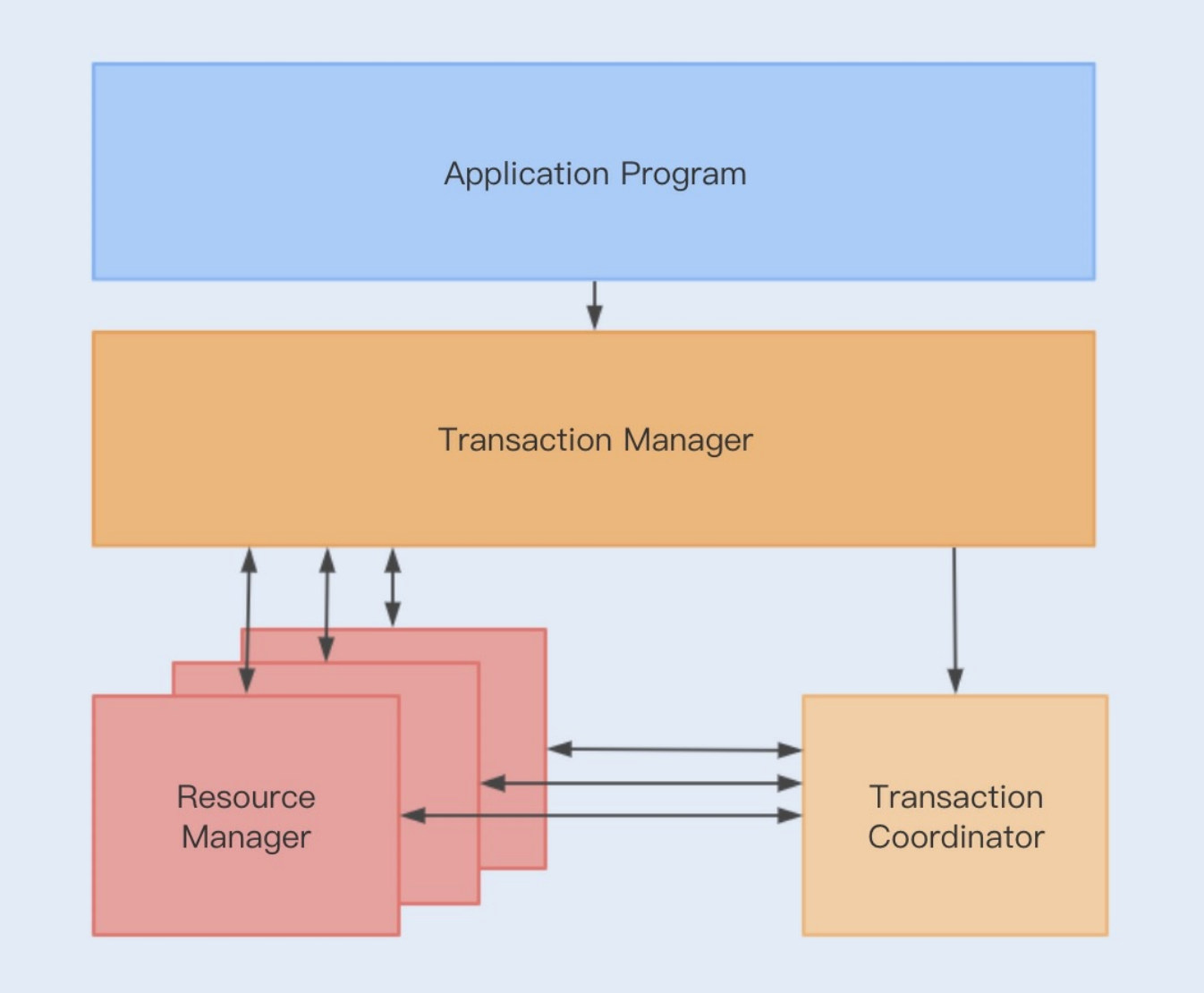
首先，我们需要在业务设计的时候考虑预留资源；然后，我们需要编写大量业务性代码，例如Try、Confirm、Cancel方法；最后，我们还需要为每个方法考虑幂等性。这种事务的实现和维护成本非常高，但综合来看，这种实现是目前大家最常用的分布式事务解决方案。

4. 业务无侵入方案——Seata(Fescar)

Seata是阿里去年开源的一套分布式事务解决方案，开源一年多已经有一万多star了，可见受欢迎程度非常之高。

Seata的基础建模和DTP模型类似，只不过前者是将事务管理器分得更细了，抽出一个事务协调器（Transaction Coordinator 简称TC），主要维护全局事务的运行状态，负责协调并驱动全局

事务的提交或回滚。而TM则负责开启一个全局事务，并最终发起全局提交或全局回滚的决议。如下图所示：



按照Github中的说明介绍，整个事务流程为：

- TM 向 TC 申请开启一个全局事务，全局事务创建成功并生成一个全局唯一的 XID；
- XID 在微服务调用链路的上下文中传播；
- RM 向 TC 注册分支事务，将其纳入 XID 对应全局事务的管辖；
- TM 向 TC 发起针对 XID 的全局提交或回滚决议；
- TC 调度 XID 下管辖的全部分支事务完成提交或回滚请求。

Seata与其它分布式最大的区别在于，它在第一提交阶段就已经将各个事务操作commit了。Seata认为在一个正常的业务下，各个服务提交事务的大概率是成功的，这种事务提交操作可以节约两个阶段持有锁的时间，从而提高整体的执行效率。

那如果在第一阶段就已经提交了事务，那我们还谈何回滚呢？

Seata将RM提升到了服务层，通过JDBC数据源代理解析SQL，把业务数据在更新前后的数据镜像组织成回滚日志，利用本地事务的 ACID 特性，将业务数据的更新和回滚日志的写入在同一个

本地事务中提交。

如果TC决议要全局回滚，会通知RM进行回滚操作，通过XID找到对应的回滚日志记录，通过回滚记录生成反向更新SQL，进行更新回滚操作。

以上我们可以保证一个事务的原子性和一致性，但隔离性如何保证呢？

Seata设计通过事务协调器维护的全局写排它锁，来保证事务间的写隔离，而读写隔离级别则默认为未提交读的隔离级别。

总结

在同服务多数据源操作不同数据库的情况下，我们可以使用基于XA规范实现的分布式事务，在Spring中有成熟的JTA框架实现了XA规范的二阶事务提交。事实上，二阶事务除了性能方面存在严重的阻塞问题之外，还有可能导致数据不一致，我们应该慎重考虑使用这种二阶事务提交。

在跨服务的分布式事务下，我们可以考虑基于TCC实现的分布式事务，常用的中间件有TCC-Transaction。TCC也是基于二阶事务提交原理实现的，但TCC的二阶事务提交是提到了服务层实现。TCC方式虽然提高了分布式事务的整体性能，但也给业务层带来了非常大的工作量，对应用服务的侵入性非常强，但这是大多数公司目前所采用的分布式事务解决方案。

Seata是一种高效的分布式事务解决方案，设计初衷就是解决分布式带来的性能问题以及侵入性问题。但目前Seata的稳定性有待验证，例如，在TC通知RM开始提交事务后，TC与RM的连接断开了，或者RM与数据库的连接断开了，都不能保证事务的一致性。

思考题

Seata在第一阶段已经提交了事务，那如果在第二阶段发生了异常要回滚到Before快照前，别的线程若是更新了数据，且业务走完了，那么恢复的这个快照不就是脏数据了吗？但事实上，Seata是不会出现这种情况的，你知道它是怎么做到的吗？

期待在留言区看到你的答案。也欢迎你点击“请朋友读”，把今天的内容分享给身边的朋友，邀请他一起讨论。

精选留言

QQ怪 2

不太理解seata默认隔离级别为啥是未提交读，不怕脏读？还是为了保证性能才做的妥协？

2019-08-28

作者回复

默认情况下，seata认为大多数分布式业务涉及到脏读的可能性比较小，所以保证了大多数场景下的高效性。

如果需要达到全局的 读已提交，seata也提供了相应的机制来达到目的。

2019-08-28

G 1

目前主流的做法不是通过异步消息来实现的吗。下单同步调用扣减库存接口。然后业务线监听订单状态接口实现业务。对于扣减库存如果发生超时，下单失败。商品中心监听费单消息，加回库存。来实现最终一致性。其他业务类同

2019-09-18

作者回复

MQ实现的分布式事务也是TCC的一种实现方式，也是主流的一种分布式解决方案

2019-09-19

任鹏斌 1

老师有个问题阿里的开源分布式方案是事务管理器是单点的，如果挂掉了会不会引起事务不一

致？

2019-09-03

作者回复

会的，我在文中已经提到了

2019-09-04

-W.LI-

1

老师好!

Seata 设计通过事务协调器维护的全局写排它锁，来保证事务间的写隔离，而读写隔离级别则默认为未提交读的隔离级别。

这个全局写排他锁支持那几种锁啊?

行锁，表锁，间隙锁，元数据锁别的记不起来了

如果支持的锁粒度不够吞吐量也会降低很多吧。

2019-08-27

作者回复

赞，全局写排它锁是根据resourceId + table + pks实现。

2019-08-28

-W.LI-

1

课后习题:全局写锁，第一阶段没有准确的提交或者回滚前，后续业务无法持有锁。我本来还想问下老师这个是怎么做到的，不过老师最后写了嘿嘿省的问了。默认读为提交，不怕脏读么?

TCC协议具体每一步怎么做讲一下么老师?

已订单支付为例，

try:尝试预扣处理,怎么预扣呢。用redis锁库存还是直接怎么锁。(抢购装备，游戏币和装备都在try阶段锁定。冲突大的后try，提交的时候冲突大的先commit?)。

try阶段，如果同时有多个事务进行try操作都能try成功么?如果支持try成功感觉有可能出现课后问题的情况。try这一步很重要啊，需要保证try以后，一定能提交成功，也一定能回滚。会不会有万一的?万一兜底解决是人工处理么?

2019-08-27

作者回复

对的，我们首先要使用重试机制，其次保证记录日志。

2019-08-28

天涯xj

0

为什么JTA不能适合单一数据源的分布式事务？

2019-10-30

Demon.Lee

0

TCC 也是基于二阶事务提交原理实现的，但 TCC 的二阶事务提交是提到了服务层实现。

Seata将RM也提升到了服务层实现。

--------老师，这两种服务层实现，应该不是一回事吧

2019-09-22

作者回复

是的，seata的RM提升到了服务层

2019-09-25

Gred 0

思考题：为什么SeaTac不会产生脏数据，第一点因为全局排它锁，全局排它锁的结构是【resourceId + table + pks】，精确到某个资源某张表的某个条数据。第二点，如果A事务对于【Select * from table where id = pks for update】数据执行失败，且未回滚【resourceId+table+pk】，这时全局锁尚未释放。B事务申请相同资源的全局锁会失败。该条数据只能看不能处理，也说不上产生脏数据了。

2019-09-18

再续啸傲 0

文中“如果 RM 决议要全局回滚，会通知 RM 进行回滚操作”，按照学习后的理解，应该是TC决定是否要进行全局回滚，不知道我理解的是否有偏差，忘老师指正

2019-09-04

作者回复

对的，已修正

2019-09-04

Jxin 0

1.第一次听到TCC,感觉也是两阶段提交的思想，只是把询问变成try操作。

2.至于补偿，感觉mq的方式更像是在补偿。（在这里，因为mq是最终一致所以个人觉得更像补偿）

3.分布式事务框架还有个lcn，事务的搬运工。

2019-09-04

N 0

老师您好，微服务间在事务中通过dubbo调用的方式是不是也可以实现分布式事务？

2019-09-01

作者回复

可以的，Seata中有个基于dubbo实现的分布式事务的例子，有兴趣可以自己参考手动实践一下

2019-09-02

灿烂明天 0

老师好，我看网上有些是用mq消息中间件来解决分布式事务的，其实这个方案能不能解决分布式事务问题的？他的思想是基于tcc的吗？

2019-08-28

作者回复

可以的，目前很多团队用过MQ实现分布式事务，也是基于TCC的思想实现。

2019-08-28

许童童 0

分页式事务中常用的方法：

1.二阶段提交

2.三阶段提交
3.TCC事务
4.Seata（有待验证）

2019-08-27

晓杰 0

全局写锁，如果线程还没有提交或者回滚事务，其他线程无法获得锁
老师，默认的隔离级别是读未提交，不是会发生脏读吗，这里是不是有问题

2019-08-27

作者回复

默认情况下，seata认为大多数分布式业务涉及到脏读的可能性比较小，所以保证了大多数场景下的高效性。

2019-08-28

疯狂咸鱼 0

老师，会讲Paxos算法么，面试经常会问道

2019-08-27

作者回复

可以考虑加餐

2019-08-30

疯狂咸鱼 0

老师是全能！

2019-08-27

JackJin 0

seata一阶段提交拿 全局锁 尝试被限制在一定范围内，超出范围将放弃，并回滚本地事务，释放本地锁。
这句话不理解，麻烦老师解答一下！

2019-08-27

作者回复

这句话在哪看到的呢，没有太理解。seata是根据一个全局事务ID进行上下文传播的。

2019-08-28

JackJin 0

一阶段本地事务提交前，需要确保先拿到 全局锁 。
拿不到 全局锁 ，不能提交本地事务。
拿 全局锁 的尝试被限制在一定范围内，超出范围将放弃，并回滚本地事务，释放本地锁。
以一个示例来说明：

两个全局事务 tx1 和 tx2，分别对 a 表的 m 字段进行更新操作，m 的初始值 1000。

tx1 先开始，开启本地事务，拿到本地锁，更新操作 m = 1000 - 100 = 900。本地事务提交前，先拿到该记录的 全局锁 ，本地提交释放本地锁。 tx2 后开始，开启本地事务，拿到本地锁，更

新操作 m = 900 - 100 = 800。本地事务提交前，尝试拿该记录的 全局锁 ，tx1 全局提交前，该记录的全局锁被 tx1 持有，tx2 需要重试等待 全局锁 。

2019-08-27

LW

0

思考题：全局事务ID，应该是参照数据库的事务ID来实现一致性的吧

2019-08-27

作者回复

对的，赞

2019-08-28

密码123456

0

通过查询未提交读的事务版本号？

2019-08-27

作者回复

差不多，由事务协调器维护的全局写排他锁，来保证事务间的写隔离。

2019-08-28