MySQL查询性能的优化

    MySQL查询性能的优化涉及多个方面，其中包括库表结构、建立合理的索引、设计合理的查询。库表结构包括如何设计表之间的关联、表字段的数据类型等。这需要依据具体的场景进行设计。如下我们从数据库的索引和查询语句的设计两个角度介绍如何提高MySQL查询性能。

数据库索引

    索引是存储引擎中用于快速找到记录的一种数据结构。索引有多种分类方式，按照存储方式可以分为：聚簇索引和非聚簇索引；按照数据的唯一性可以分为：唯一索引和非唯一索引；按照列个数可以分为：单列索引和多列索引等。索引也有多种类型：B-Tree索引、Hash索引、空间数据索引（R-Tree）、全文索引等。

B-Tree索引

    在利用B-Tree索引进行查询的过程中，有几点注意事项，我们以表A进行说明。其中表A的定义如下：

1. create table A(id int auto\_increment primary key, name varchar(10), age tinyint,
2. sex enum('男','女'), birth datatime, key(name,age,sex));

复制代码

id为主键，并在name,age,sex列上建立了索引。

* 全值匹配：指和索引中的所有列进行匹配，例如查找name='Jone' and age=13 and sex='男'的人；
* 匹配最左前缀：指用索引的第一列name，如where name='Jone'，该查询只使用了索引的第一列
* 匹配列前缀：匹配索引列值的开头，如where name like 'J%'，查找名字以J开头的人；
* 匹配范围值：例如查找年龄在10-30之间的Jone，where name='Jone' and age between 10 and 30;
* 只访问索引的查询：如果在select中选择的字段都是索引中的字段，那么就不需要访问数据行，从而提高查询速度。
* 如果不是按照索引的最左列进行查找，则无法使用索引，如当仅查找表A中年龄为15岁的人时则无法使用索引；
* 不能跳过索引中的列，如查找表A中名字为Jone且为男性的人，则索引只能使用name列，无法使用sex列；
* 查询中索引的某列是范围查询，则该列后的查询条件将不能使用索引。

Hash索引与B-Tree的区别：

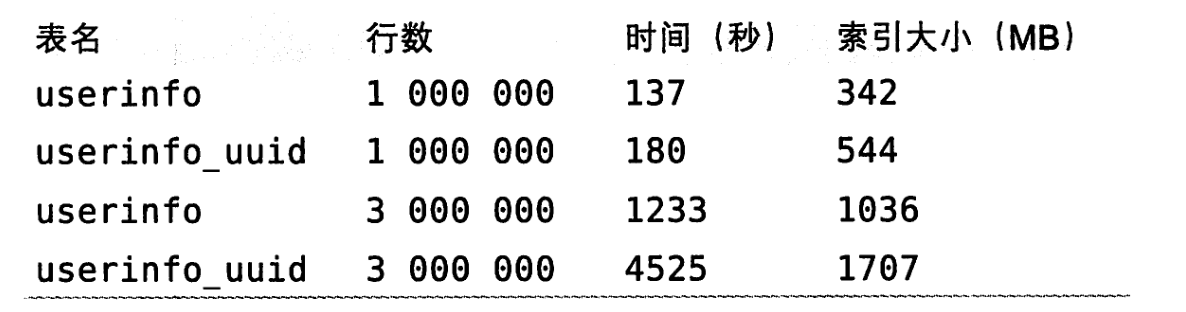
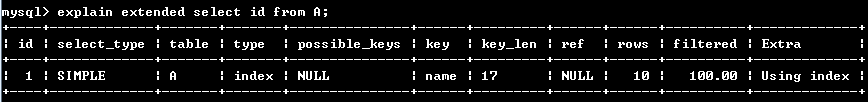
1. Hash索引指包含哈希值（根据key中的列计算）和行指针，而B-Tree存储的是列值。所以Hash不能使用索引来避免读取数据行；
2. Hash索引数据不是按照索引值顺序存储的，所以无法用于排序；
3. Hash索引不支持部分索引列匹配查找，因为Hash值是根据索引中的全部列计算出来的；
4. Hash索引只支持等值比较查询，包括=、in()、<=>。不支持范围查询。

索引的优点

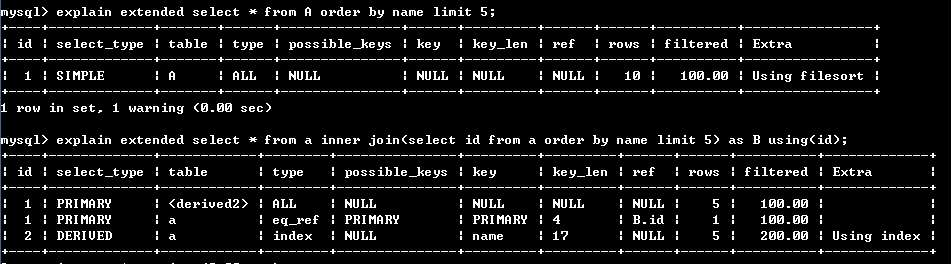
    索引不仅仅可以让服务器快速定位到表的指定位置，而且还有以下优点：

1. B-Tree索引按照列的顺序存储数据，所以可以用来做Order by和group by操作，避免排序和临时表
2. B-Tree索引中存储索引列的值，所以当select的值在索引中时，可以避免访问数据行
3. 索引可以有效减少服务器扫描的数据量。

高性能的索引策略  
    正确地创建和使用索引是实现高性能查询的基础。前面已经介绍了各种类型的索引以及对应的优缺点。高效地选择和使用索引有很多种方式，其中有些是针对特殊案例的优化，有些则是针对特定行为的优化。

1. 独立的列：指索引不能是表达式的一部分，也不能是函数的参数。如：select \* from A where id+1=5; 则无法使用主键索引。
2. 前缀索引和索引选择性：有时需要索引很长的字符串，索引会占用很大的空间，通常可以索引开始的部分字符来节约索引空间，提高索引效率，但也会降低索引的选择性。索引的选择性=不重复索引值/数据表的记录总数。索引的选择性越高查询效率越高。
3. 多列索引：首先需要说明在多列上创建索引不等同于给这些列的每一列单独建立索引。当执行查询的时候，MySQL只能使用一个索引。如果你有三个单列的索引，MySQL会试图选择一个限制最严格的索引。即使是限制最严格的单列索引，它的限制能力也肯定远远低于这三个列上的多列索引。比如我们想查询表A中id为3或者名字首字母为A的人，sql语句的两种写法对比，其中第二种写法比第一种减少对表的扫描次数：
4. 多列索引中索引列的顺序也十分重要，在设计索引的顺序时也需要考虑如何更好地满足排序和分组的需要（B-Tree）。在一个多列的B-Tree索引中，索引列的顺序意味着索引首先按照最左列进行排序，其次是第二列等等。确定索引列的顺序有一个经验法则：将选择性最高的列放到索引最前列。当然如果需要考虑对表的排序的情况就需要另当考虑了。
5. 聚簇索引：不是一种单独的索引类型，而是一种数据存储方式，具体的细节依赖于其实现方式，InnoDB的聚簇索引实际上在同一个结构中保存了B-Tree索引和数据行，一个表只能有一个聚簇索引。聚簇索引的优（1-3）缺（4-7）点如下：  
   1. 可以把相关数据保存在一起。例如实现电子邮箱时，可以根据用户ID来聚集数据，这样只需要从磁盘读取少数的数据页就能够获取某个用户的全部邮件。如果没有聚簇索引，则每封邮件都可能导致一次磁盘I/O；
   2. 数据访问更快。聚簇索引将索引和数据保存在同一个B-Tree中，因此聚簇索引中获取数据通常比在非聚簇索引中查找要快；
   3. 使用覆盖索引扫描的查询可以直接使用页节点中的主键值；
   4. B-Tree索引插入速度严重依赖于插入顺序。按照聚簇索引列中值的顺序插入是加载数据到InnoDB表中速度最快的；
   5. 更新聚簇索引列的代价很高，因为会强制InnoDB将每个被更新的行移动到新的位置；
   6. 被插入的新行在移动时，可能面临“页分裂”的问题。页分裂问题是聚簇索引要求必须将这一行插入到某个已满的页中时，存储引擎会将该页分裂成两个页面来容纳该行，也就是一次页分裂操作，导致表占用更多的磁盘空间；
   7. 聚簇索引可能导致全表扫描变慢，尤其是行比较稀疏，或者由于页分裂导致数据存储不连续的时候。
   8. 如上是盗取的一个向InnoDB表中插入数据的时间和索引大小的图，其中userinfo表和userinfo\_uuid表唯一的区别是userinfo表以id为主键，而userinfo\_uuid表以uuid为主键，而插入100万和300万数据的顺序是按照id列的顺序插入的，由上图可知，当插入300万数据行时，userinfo\_uuid表由于不是按照主键（uuid）的顺序插入的数据，会导致大量的页分裂，从而插入需要更多的时间、索引占用更大的空间。
6. 覆盖索引：大家都会根据where的条件建立合适的索引，这只是索引优化的一个方面。优秀的索引还应该考虑整个查询。MySQL可以使用索引直接获取列的数据，这样就不需要读取数据行了。如果索引包含（覆盖）所有需要查询的字段值，我们就称之为覆盖索引。当查询是一个索引覆盖查询时，Extra列可以看到Using index的信息。  
   

当然覆盖查询还是有很多陷阱可能导致无法实现优化的。MySQL查询优化器会在执行查询前判断是否有一个索引能够进行覆盖，覆盖where条件中的字段和select的字段。如果不能覆盖，则还是需要扫描数据行。



因为InnoDB表中非聚簇索引中存储主键值，所以我们先根据条件获取主键值，然后再根据主键值进行查询，这种方式叫做延迟关联。

1. 使用索引扫描来做排序。如果EXPLAIN出来的type列值为index，说明MySQL使用了索引扫描来做排序。扫描索引本身是很快的，但是如果索引不能覆盖查询所需的全部列，那就不得不每扫描一条索引记录就回表查询一次对应的行。这基本都是随机I/O，因此按索引顺序读取的速度通常要比顺序地全表扫描慢，尤其是I/O密集型的工作负载时。因此MySQL设计索引时应尽可能的满足排序和查找。只有索引列顺序和order by子句的顺序完全一致，并且所有列的排序方向都一致时，MySQL才能使用索引来对结果做排序。如果查询关联多张表，则只有order by子句引用的字段全部为第一个表时，才能使用索引排序。

如上是分别使用主键id排序和name排序的查询，可以看出使用id排序的查询使用了索引排序，而name排序的查询使用的是filesort。

  总结:  
    总的来说编写查询语句时，应尽可能选择合适的索引以避免单行查找，尽可能的使用原生顺序从而避免额外的排序操作，并尽可能使用索引覆盖查询。我们通过响应时间来对查询进行分析，找出消耗时间最长的查询或者给服务器带来压力最大的查询，然后检查查询的schema、SQL和索引结构，判断是否有查询扫描了太多的行，是否做了很多额外的排序或者使用了临时表，是否使用了随机I/O访问数据，或者太多回表查询哪些不在索引中的列的操作。  
查询设计  
    在发现查询效率不高时，首先就需要考虑查询语句的设计是否合理。如下将会介绍一些查询优化技巧，然后在介绍一些MySQL优化器内部的机制，并展示MySQL是如何执行查询的。最后探索查询优化的模式，以帮助MySQL更有效地执行查询。  
  
优化数据访问  
    查询性能低下的最基本原因是访问的数据太多了。因此大部分的性能低下查询都可以通过减少访问的数据量进行优化。减少数据访问量通常意味着访问了太多的行，但有时也可能是访问了太多的列。在查询时如果仅需要查询结果集中的前某些行，则最简单的方式是在查询语句的最后加上limit。在进行多表关联查询时应尽量避免使用select \*，因为它返回表的所有列，但是这些列可能并不都是必须的。除了请求了不需要的数据，还需要查看MySQL是否在扫描额外的记录，其中可以通过扫描行数和返回行数进行衡量。如果发现查询中需要扫描大量的数据但是只返回少数的行，通常可以：

1. 使用索引覆盖扫描，把所有需要的列都放入索引，这样存储引擎无须回表获取对应行就可以返回结果；
2. 改变库表结构；
3. 重写这个复杂的查询，让MySQL优化器能够以更优的方式执行这个查询。

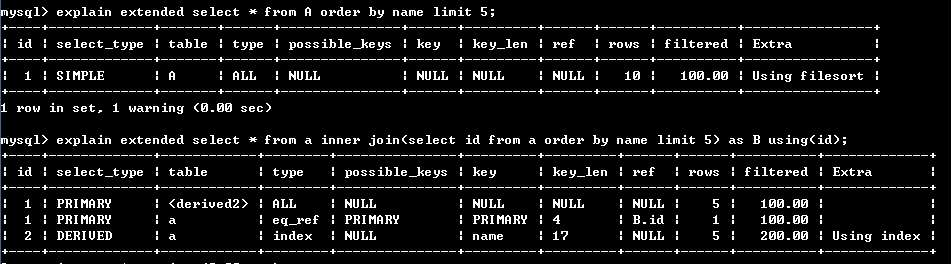
重构查询方式  
    设计查询的时候一个需要考虑的重要问题是，是否需要将一个复杂的查询分成多个简单的查询。在传统的实现中总是强调数据库层完成尽可能多的工作，这样的逻辑在于以前总是认为网络通信、查询解析和优化是一件代价很高的事情。但是这样的想法对于MySQL并不适用，MySQL从设计上连接和断开连接都很轻量级，在返回一个小的查询结果方面很高效。  
分解关联查询：很多高性能的应用都会对关联查询进行分解，简单地说就是对每个表进行一次单表查询，然后将结果在应用程序中进行关联。如下图所示：  
  
    查询计算机1班学生的所有成绩，我们可以将上过程分解为三个子步骤，如下：  
  
    那么这么分解的好处又在哪里呢？首先是让缓存的效率更高。许多应用程序可以方便的缓存单表查询对应的结果对象。如已经缓存了计算机1班对应的id为1，tb\_student表中1班的学生有1号和5号，从而可以从成绩表中查询1号和5号学生的成绩；其次查询分解后，执行单个查询可以减少锁竞争；再次查询本身效率也会有所提升。如上使用in()代替关联查询，可以让MySQL按照ID顺序进行查询，这可能比随机的关联更加高效；最后分解关联查询可以减少冗余记录的查询，在应用层做关联查询时，意味着对于某条记录应用只需要查询一次，而在数据库中做关联查询，则可能需要重复地访问一部分数据。  
查询执行的基础  
    当希望MySQL能够以较高的性能运行查询时，最好的办法就是弄清楚MySQL是如何优化和执行查询的。如下图展示了向MySQL发送一个请求时MySQL具体的操作过程：

1. 首先服务器接收到一条客户端请求，先检查查询缓存，如果命中缓存，则立刻返回缓存中的数据，否则进入下一阶段；
2. 服务器进行SQL解析、预处理，再由优化器生成对应的执行计划；
3. MySQL根据优化器生成的执行计划，调用存储引擎的API执行查询；
4. 将结果返回给客户端。

第一步是MySQL客户端/服务器通信，二者之间通信协议是“半双工”的，也就是说在某一时刻只能有一方在发送数据。在任何一个时刻MySQL连接都有一个状态，该状态表示MySQL当前的工作，通过SHOW FULL PROCESSLIST命令查询状态。其中状态有Sleep、Query、Locked、Analyzing and statistics、Coping to tmp table、Sorting result、Sending data.  
  
第二步是查寻缓存。在解析一个查询语句之前，如果查询缓存是打开的，那么MySQL会优先检查这个查询是否命中查询缓存中的数据。通常是通过一个对大小写敏感的Hash查找实现。如果命中，那么在返回结果前MySQL会检查一次用户权限，该过程无须解析查询SQL语句。如果未命中，则解析SQL语句。  
  
第三步是查询优化处理。包括解析SQL、预处理、优化SQL执行计划，其中出现任何错误都会终止查询。首先，MySQL通过关键字将SQL语句进行解析，并生成一棵对应的“解析树”。查询优化器负责将解析树转化成执行计划，优化器的作用就是找到查询的较优执行计划。MySQL使用基于成本的优化器，它将尝试预测一个查询使用某种执行计划时的成本（SHOW STATUS LIKE 'Last\_query\_cost'），并选择成本最小的一个。查询优化器是一个非常复杂的部件，它使用了很多优化策略来生成一个最优的执行计划。优化策略分为：静态优化和动态优化。静态优化可以直接对解析树进行分析，并完成优化。例如，优化器可以通过简单的代数变换将where条件转换成另一种等价形式，静态优化不依赖于特别的数值，如where中带入的常数。静态优化在第一次完成后就一直有效，即使使用不同的参数重复执行也不会发生变化，可以认为是一种“编译时优化”。动态优化是上下文相关的，如where条件中取值、索引条目对应的数据行数等，是一种“运行时优化”。如下是MySQL能够处理的优化类型：

1. 重新定义关联表的顺序：数据表的关联并不总是按照查询中指定的顺序进行。
2. 将外连接转化为内连接：并不是OUTER JOIN语句都必须以外连接的方式执行。如where条件、库表结构都可能会让外连接等价于一个内连接；
3. 使用等价变换：MySQL使用等价变换来规范表达式。如（a<b and b=c） and a=10则会改写为a=10 and b>10 and b=c；
4. 优化count()、min()、max()
5. 覆盖索引扫描：当索引中的列包含所需要的列时，MySQL使用索引返回需要的数据，不需要查询对应的行数据；
6. 子查询优化：将子查询转化一种效率更高的形式，从而减少多个查询多次对数据的访问；
7. 提前终止查询：使用limit时，发现已经满足查询需求时，MySQL能够立刻终止查询；
8. 列表in()比较：MySQL中in()不等同于多个or条件的子句，因为MySQL首先对in()中的数据进行排序，然后通过二分查找的方式来确定列表中的值是否满足条件，该时间复杂度为o(logn)，而多个or查询的时间复杂度为o(n)。

    当MySQL需要对选择的数据进行排序时，如果无法使用索引进行排序，那么MySQL在数据量小则在内存中进行排序，如果数据量大则需要磁盘进行排序，不过MySQL将这一过程统一称为文件排序（filesort）。如果需要排序的数据量小于“排序缓冲区”，MySQL使用内存进行“快速排序”操作，如果内存不够排序，MySQL先对数据进行分块，然后对每个独立的块使用“快速排序”,并将各块排序结果放入磁盘，然后将各个排好序的块进行合并（merge）。在关联查询的时候如果需要排序，MySQL会分两种情况来处理这样的文件排序，如果order by子句中的所有列都来自关联的第一个表，那么MySQL在关联处理第一个表的时候就进行文件排序，则MySQL的EXPLAIN结果的extra字段就会有“using filesort”。除此之外的其他情况，MySQL都会先将关联结果放到一个临时表中，，然后在所有关联都结束后再进行文件排序，此时的MySQL的EXPLAIN结果的extra字段值为“Using temporary;Using filesort”。如果查询中有limit的话，limit也会在排序之后应用，所以即使返回较少的数据，临时表和需要排序的数量仍会非常大（MySQL5.6的limit子句在此处已经做了改进）。  
  
第四步是查询执行引擎。MySQL根据执行计划给出的指令逐步执行，在该过程中，有大量的操作需要通过调用存储引擎实现的接口来完成，也就是“Handler API”。MySQL在优化阶段就为每个表创建一个handler实例，优化器根据这些实例的接口获取表的相关信息。  
  
最后一步就是将查询的结果返回给客户端。MySQL将结果集返回客户端是一个增量、逐步返回的过程。一旦服务器处理完最后一个关联表，开始生成第一条结果时，MySQL就可以开始想客户端逐步返回结果。这样有两个好处：一是服务器端无须存储太多的结果；二是结果集中的每一行都会以一个满足MySQL客户端/服务器通信协议的封包发送，再通过TCP协议进行传输，从而是客户端可以在第一时间获得返回的结果。  
  
优化特定类型的查询

1. 优化count()查询。如果指定了列，则查询该列不为null的行数，如果为count(\*)则查询总行数。
2. 优化关联查询，确保on或者using子句中的列上有索引。确保group by和order by的表达式只涉及一个表中的列，这样MySQL才有可能使用索引来优化整个过程。
3. 优化group by和distinct。MySQL使用同样的方法优化这两类查询，通常是利用索引的顺序性进行优化。但是如果无法使用索引，group by使用两种策略来完成：使用临时表或者文件排序来做分组。
4. 优化limit分页，使用延迟关联的方式来优化limit分页；
5. 优化UNION查询。MySQL通过创建并填充临时表的方式来执行UNION查询，因此需要手工的将where、limit、order by等子句“下推”到UNION的各个子查询中，除非确实需要服务器消除重复的行，否则一定要使用UNION ALL，如果没有ALL关键字，MySQL会给临时表加上distinct，从而对临时表的数据做唯一性检查，这样代价非常高。

总结  
    综上所有的内容可知，创建高性能应用程序要考虑schema、索引、查询语句以及查询优化等问题。理解查询是如何被执行的以及时间都消耗在哪些地方，从而针对耗时大的查询语句进行改进。

# X/OpenDTP事务模型

X/Open Distributed Transaction Processing Reference Model

X/Open是一个组织机构，定义出的一套分布式事务标准， 定义了规范的API接口

2PC（two -phase-commit）, 用来保证分布式事务的完整性

J2EE 遵循了X/open DTP规范，设计并实现了java里面的分布式事务编程接口规范-JTA

XA是X/Open DTP定义的中间件与数据库之间的接口规范。 XA接口函数由数据库厂商提供

## X/OpenDTP角色

APapplication

RM resouces manager资源管理器。 数据库

TM transaction manager事务管理器，事务协调者

# 2PC（two -phase-commit）

(CAP)

## 阶段一：提交事务请求（投票）

1. TM向所有的AP发送事务内容，询问是否可以执行事务的提交操作，并等待各个AP的响应
2. 执行事务

各个AP节点执行事务操作，将undo和redo信息记录到事务日志中，尽量把提交过程中所消耗时间的操作和准备都提前完成后确保后续

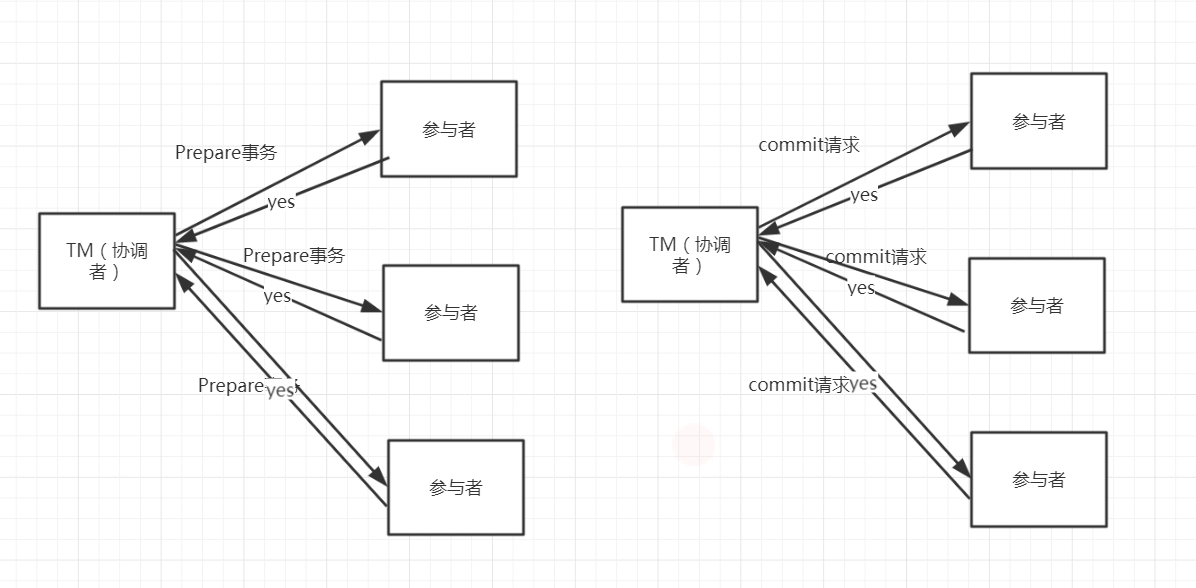
事务提交的成功率

1. 各个AP向TM反馈事务询问的响应

各个AP成功执行了事务操作，那么反馈给TMyes的response；如果AP没有成功执行事务，就反馈TM no的response

## 阶段二：执行事务提交

### 执行提交事务

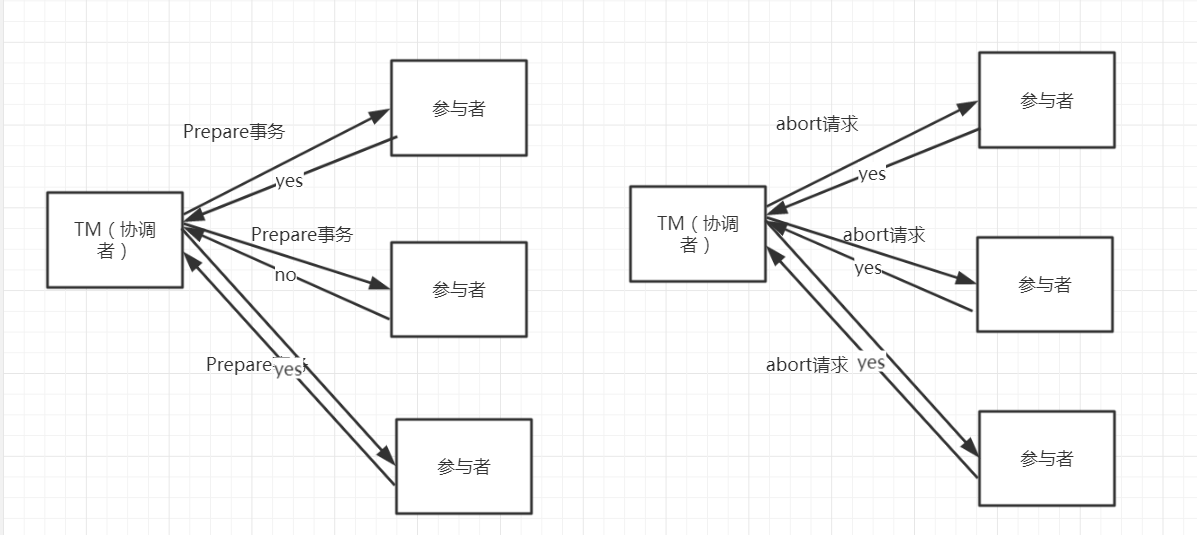


假设一个事务的提交过程总共需要30s， 其中prepare操作需要28（事务日志落地磁盘及各种io操作），而真正commit只需要2s

那么，commit阶段发生错误的概率和prepare相比， 2/28 (<10%) .只要第一个阶段成功，那么commit阶段出现失败的概率就非常小

大大增加了分布式事务的成功概率

### 中断事务提交



# 2pc存在的问题

1. 数据一致性问题
2. 同步阻塞

# 3PC(three phase commit)

阶段一：canCommit

阶段二：preCommit

阶段三：doCommit

# 分布式事务的实现

JOTM(java open transaction manager)

Atomikos

# 互联网的分布式事务的解决方案

什么是分布式事务？

到此为止，所介绍的事务都是基于单数据库的本地事务，目前的数据库仅支持单库事务，并不支持跨库事务。而随着微服务架构的普及，一个大型业务系统往往由若干个子系统构成，这些子系统又拥有各自独立的数据库。往往一个业务流程需要由多个子系统共同完成，而且这些操作可能需要在一个事务中完成。在微服务系统中，这些业务场景是普遍存在的。此时，我们就需要在数据库之上通过某种手段，实现支持跨数据库的事务支持，这也就是大家常说的“分布式事务”。

这里举一个分布式事务的典型例子——用户下单过程。

当我们的系统采用了微服务架构后，一个电商系统往往被拆分成如下几个子系统：商品系统、订单系统、支付系统、积分系统等。整个下单的过程如下：

用户通过商品系统浏览商品，他看中了某一项商品，便点击下单

此时订单系统会生成一条订单

订单创建成功后，支付系统提供支付功能

当支付完成后，由积分系统为该用户增加积分

上述步骤2、3、4需要在一个事务中完成。对于传统单体应用而言，实现事务非常简单，只需将这三个步骤放在一个方法A中，再用Spring的@Transactional注解标识该方法即可。Spring通过数据库的事务支持，保证这些步骤要么全都执行完成，要么全都不执行。但在这个微服务架构中，这三个步骤涉及三个系统，涉及三个数据库，此时我们必须在数据库和应用系统之间，通过某项黑科技，实现分布式事务的支持。

CAP理论

CAP理论说的是：在一个分布式系统中，最多只能满足C、A、P中的两个需求。

CAP的含义：

C：Consistency 一致性

同一数据的多个副本是否实时相同。

A：Availability 可用性

可用性：一定时间内 & 系统返回一个明确的结果 则称为该系统可用。

P：Partition tolerance 分区容错性

将同一服务分布在多个系统中，从而保证某一个系统宕机，仍然有其他系统提供相同的服务。

CAP理论告诉我们，在分布式系统中，C、A、P三个条件中我们最多只能选择两个。那么问题来了，究竟选择哪两个条件较为合适呢？

对于一个业务系统来说，可用性和分区容错性是必须要满足的两个条件，并且这两者是相辅相成的。业务系统之所以使用分布式系统，主要原因有两个：

提升整体性能

当业务量猛增，单个服务器已经无法满足我们的业务需求的时候，就需要使用分布式系统，使用多个节点提供相同的功能，从而整体上提升系统的性能，这就是使用分布式系统的第一个原因。

实现分区容错性

单一节点 或 多个节点处于相同的网络环境下，那么会存在一定的风险，万一该机房断电、该地区发生自然灾害，那么业务系统就全面瘫痪了。为了防止这一问题，采用分布式系统，将多个子系统分布在不同的地域、不同的机房中，从而保证系统高可用性。

这说明分区容错性是分布式系统的根本，如果分区容错性不能满足，那使用分布式系统将失去意义。

此外，可用性对业务系统也尤为重要。在大谈用户体验的今天，如果业务系统时常出现“系统异常”、响应时间过长等情况，这使得用户对系统的好感度大打折扣，在互联网行业竞争激烈的今天，相同领域的竞争者不甚枚举，系统的间歇性不可用会立马导致用户流向竞争对手。因此，我们只能通过牺牲一致性来换取系统的可用性和分区容错性。这也就是下面要介绍的BASE理论。

BASE理论

CAP理论告诉我们一个悲惨但不得不接受的事实——我们只能在C、A、P中选择两个条件。而对于业务系统而言，我们往往选择牺牲一致性来换取系统的可用性和分区容错性。不过这里要指出的是，所谓的“牺牲一致性”并不是完全放弃数据一致性，而是牺牲强一致性换取弱一致性。下面来介绍下BASE理论。

BA：Basic Available 基本可用

整个系统在某些不可抗力的情况下，仍然能够保证“可用性”，即一定时间内仍然能够返回一个明确的结果。只不过“基本可用”和“高可用”的区别是：

“一定时间”可以适当延长

当举行大促时，响应时间可以适当延长

给部分用户返回一个降级页面

给部分用户直接返回一个降级页面，从而缓解服务器压力。但要注意，返回降级页面仍然是返回明确结果。

S：Soft State：柔性状态

同一数据的不同副本的状态，可以不需要实时一致。

E：Eventual Consisstency：最终一致性

同一数据的不同副本的状态，可以不需要实时一致，但一定要保证经过一定时间后仍然是一致的。

酸碱平衡

ACID能够保证事务的强一致性，即数据是实时一致的。这在本地事务中是没有问题的，在分布式事务中，强一致性会极大影响分布式系统的性能，因此分布式系统中遵循BASE理论即可。但分布式系统的不同业务场景对一致性的要求也不同。如交易场景下，就要求强一致性，此时就需要遵循ACID理论，而在注册成功后发送短信验证码等场景下，并不需要实时一致，因此遵循BASE理论即可。因此要根据具体业务场景，在ACID和BASE之间寻求平衡。

分布式事务协议

下面介绍几种实现分布式事务的协议。

两阶段提交协议 2PC

分布式系统的一个难点是如何保证架构下多个节点在进行事务性操作的时候保持一致性。为实现这个目的，二阶段提交算法的成立基于以下假设：

该分布式系统中，存在一个节点作为协调者(Coordinator)，其他节点作为参与者(Cohorts)。且节点之间可以进行网络通信。

所有节点都采用预写式日志，且日志被写入后即被保持在可靠的存储设备上，即使节点损坏不会导致日志数据的消失。

所有节点不会永久性损坏，即使损坏后仍然可以恢复。

1. 第一阶段（投票阶段）：

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。

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）

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。

2. 第二阶段（提交执行阶段）：

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：

协调者节点向所有参与者节点发出”正式提交(commit)”的请求。

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。

参与者节点向协调者节点发送”完成”消息。

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：

协调者节点向所有参与者节点发出”回滚操作(rollback)”的请求。

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。

参与者节点向协调者节点发送”回滚完成”消息。

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：

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。

参与者发生故障。协调者需要给每个参与者额外指定超时机制，超时后整个事务失败。（没有多少容错机制）

协调者发生故障。参与者会一直阻塞下去。需要额外的备机进行容错。（这个可以依赖后面要讲的Paxos协议实现HA）

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。

为此，Dale Skeen和Michael Stonebraker在“A Formal Model of Crash Recovery in a Distributed System”中提出了三阶段提交协议（3PC）。

三阶段提交协议 3PC

与两阶段提交不同的是，三阶段提交有两个改动点。

引入超时机制。同时在协调者和参与者中都引入超时机制。

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。

1. 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。

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。

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

2. 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。

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。

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。

3. 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。

该阶段进行真正的事务提交，也可以分为以下两种情况。

3.1 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。

3.2 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。

分布式事务的解决方案

分布式事务的解决方案有如下几种：

全局消息

基于可靠消息服务的分布式事务

TCC

最大努力通知

方案1：全局事务（DTP模型）

全局事务基于DTP模型实现。DTP是由X/Open组织提出的一种分布式事务模型——X/Open Distributed Transaction Processing Reference Model。它规定了要实现分布式事务，需要三种角色：

AP：Application 应用系统

它就是我们开发的业务系统，在我们开发的过程中，可以使用资源管理器提供的事务接口来实现分布式事务。

TM：Transaction Manager 事务管理器

分布式事务的实现由事务管理器来完成，它会提供分布式事务的操作接口供我们的业务系统调用。这些接口称为TX接口。

事务管理器还管理着所有的资源管理器，通过它们提供的XA接口来同一调度这些资源管理器，以实现分布式事务。

DTP只是一套实现分布式事务的规范，并没有定义具体如何实现分布式事务，TM可以采用2PC、3PC、Paxos等协议实现分布式事务。

RM：Resource Manager 资源管理器

能够提供数据服务的对象都可以是资源管理器，比如：数据库、消息中间件、缓存等。大部分场景下，数据库即为分布式事务中的资源管理器。

资源管理器能够提供单数据库的事务能力，它们通过XA接口，将本数据库的提交、回滚等能力提供给事务管理器调用，以帮助事务管理器实现分布式的事务管理。

XA是DTP模型定义的接口，用于向事务管理器提供该资源管理器(该数据库)的提交、回滚等能力。

DTP只是一套实现分布式事务的规范，RM具体的实现是由数据库厂商来完成的。

有没有基于DTP模型的分布式事务中间件？

DTP模型有啥优缺点？

方案2：基于可靠消息服务的分布式事务

这种实现分布式事务的方式需要通过消息中间件来实现。假设有A和B两个系统，分别可以处理任务A和任务B。此时系统A中存在一个业务流程，需要将任务A和任务B在同一个事务中处理。下面来介绍基于消息中间件来实现这种分布式事务。

在系统A处理任务A前，首先向消息中间件发送一条消息

消息中间件收到后将该条消息持久化，但并不投递。此时下游系统B仍然不知道该条消息的存在。

消息中间件持久化成功后，便向系统A返回一个确认应答；

系统A收到确认应答后，则可以开始处理任务A；

任务A处理完成后，向消息中间件发送Commit请求。该请求发送完成后，对系统A而言，该事务的处理过程就结束了，此时它可以处理别的任务了。

但commit消息可能会在传输途中丢失，从而消息中间件并不会向系统B投递这条消息，从而系统就会出现不一致性。这个问题由消息中间件的事务回查机制完成，下文会介绍。

消息中间件收到Commit指令后，便向系统B投递该消息，从而触发任务B的执行；

当任务B执行完成后，系统B向消息中间件返回一个确认应答，告诉消息中间件该消息已经成功消费，此时，这个分布式事务完成。

上述过程可以得出如下几个结论：

1. 消息中间件扮演者分布式事务协调者的角色。

2. 系统A完成任务A后，到任务B执行完成之间，会存在一定的时间差。在这个时间差内，整个系统处于数据不一致的状态，但这短暂的不一致性是可以接受的，因为经过短暂的时间后，系统又可以保持数据一致性，满足BASE理论。

上述过程中，如果任务A处理失败，那么需要进入回滚流程，如下图所示：

若系统A在处理任务A时失败，那么就会向消息中间件发送Rollback请求。和发送Commit请求一样，系统A发完之后便可以认为回滚已经完成，它便可以去做其他的事情。

消息中间件收到回滚请求后，直接将该消息丢弃，而不投递给系统B，从而不会触发系统B的任务B。

此时系统又处于一致性状态，因为任务A和任务B都没有执行。

上面所介绍的Commit和Rollback都属于理想情况，但在实际系统中，Commit和Rollback指令都有可能在传输途中丢失。那么当出现这种情况的时候，消息中间件是如何保证数据一致性呢？——答案就是超时询问机制。

系统A除了实现正常的业务流程外，还需提供一个事务询问的接口，供消息中间件调用。当消息中间件收到一条事务型消息后便开始计时，如果到了超时时间也没收到系统A发来的Commit或Rollback指令的话，就会主动调用系统A提供的事务询问接口询问该系统目前的状态。该接口会返回三种结果：

提交

若获得的状态是“提交”，则将该消息投递给系统B。

回滚

若获得的状态是“回滚”，则直接将条消息丢弃。

处理中

若获得的状态是“处理中”，则继续等待。

消息中间件的超时询问机制能够防止上游系统因在传输过程中丢失Commit/Rollback指令而导致的系统不一致情况，而且能降低上游系统的阻塞时间，上游系统只要发出Commit/Rollback指令后便可以处理其他任务，无需等待确认应答。而Commit/Rollback指令丢失的情况通过超时询问机制来弥补，这样大大降低上游系统的阻塞时间，提升系统的并发度。

下面来说一说消息投递过程的可靠性保证。

当上游系统执行完任务并向消息中间件提交了Commit指令后，便可以处理其他任务了，此时它可以认为事务已经完成，接下来消息中间件一定会保证消息被下游系统成功消费掉！那么这是怎么做到的呢？这由消息中间件的投递流程来保证。

消息中间件向下游系统投递完消息后便进入阻塞等待状态，下游系统便立即进行任务的处理，任务处理完成后便向消息中间件返回应答。消息中间件收到确认应答后便认为该事务处理完毕！

如果消息在投递过程中丢失，或消息的确认应答在返回途中丢失，那么消息中间件在等待确认应答超时之后就会重新投递，直到下游消费者返回消费成功响应为止。当然，一般消息中间件可以设置消息重试的次数和时间间隔，比如：当第一次投递失败后，每隔五分钟重试一次，一共重试3次。如果重试3次之后仍然投递失败，那么这条消息就需要人工干预。

有的同学可能要问：消息投递失败后为什么不回滚消息，而是不断尝试重新投递？

这就涉及到整套分布式事务系统的实现成本问题。

我们知道，当系统A将向消息中间件发送Commit指令后，它便去做别的事情了。如果此时消息投递失败，需要回滚的话，就需要让系统A事先提供回滚接口，这无疑增加了额外的开发成本，业务系统的复杂度也将提高。对于一个业务系统的设计目标是，在保证性能的前提下，最大限度地降低系统复杂度，从而能够降低系统的运维成本。

不知大家是否发现，上游系统A向消息中间件提交Commit/Rollback消息采用的是异步方式，也就是当上游系统提交完消息后便可以去做别的事情，接下来提交、回滚就完全交给消息中间件来完成，并且完全信任消息中间件，认为它一定能正确地完成事务的提交或回滚。然而，消息中间件向下游系统投递消息的过程是同步的。也就是消息中间件将消息投递给下游系统后，它会阻塞等待，等下游系统成功处理完任务返回确认应答后才取消阻塞等待。为什么这两者在设计上是不一致的呢？

首先，上游系统和消息中间件之间采用异步通信是为了提高系统并发度。业务系统直接和用户打交道，用户体验尤为重要，因此这种异步通信方式能够极大程度地降低用户等待时间。此外，异步通信相对于同步通信而言，没有了长时间的阻塞等待，因此系统的并发性也大大增加。但异步通信可能会引起Commit/Rollback指令丢失的问题，这就由消息中间件的超时询问机制来弥补。

那么，消息中间件和下游系统之间为什么要采用同步通信呢？

异步能提升系统性能，但随之会增加系统复杂度；而同步虽然降低系统并发度，但实现成本较低。因此，在对并发度要求不是很高的情况下，或者服务器资源较为充裕的情况下，我们可以选择同步来降低系统的复杂度。

我们知道，消息中间件是一个独立于业务系统的第三方中间件，它不和任何业务系统产生直接的耦合，它也不和用户产生直接的关联，它一般部署在独立的服务器集群上，具有良好的可扩展性，所以不必太过于担心它的性能，如果处理速度无法满足我们的要求，可以增加机器来解决。而且，即使消息中间件处理速度有一定的延迟那也是可以接受的，因为前面所介绍的BASE理论就告诉我们了，我们追求的是最终一致性，而非实时一致性，因此消息中间件产生的时延导致事务短暂的不一致是可以接受的。

方案3：最大努力通知（定期校对）

最大努力通知也被称为定期校对，其实在方案二中已经包含，这里再单独介绍，主要是为了知识体系的完整性。这种方案也需要消息中间件的参与，其过程如下：

上游系统在完成任务后，向消息中间件同步地发送一条消息，确保消息中间件成功持久化这条消息，然后上游系统可以去做别的事情了；

消息中间件收到消息后负责将该消息同步投递给相应的下游系统，并触发下游系统的任务执行；

当下游系统处理成功后，向消息中间件反馈确认应答，消息中间件便可以将该条消息删除，从而该事务完成。

上面是一个理想化的过程，但在实际场景中，往往会出现如下几种意外情况：

消息中间件向下游系统投递消息失败

上游系统向消息中间件发送消息失败

对于第一种情况，消息中间件具有重试机制，我们可以在消息中间件中设置消息的重试次数和重试时间间隔，对于网络不稳定导致的消息投递失败的情况，往往重试几次后消息便可以成功投递，如果超过了重试的上限仍然投递失败，那么消息中间件不再投递该消息，而是记录在失败消息表中，消息中间件需要提供失败消息的查询接口，下游系统会定期查询失败消息，并将其消费，这就是所谓的“定期校对”。

如果重复投递和定期校对都不能解决问题，往往是因为下游系统出现了严重的错误，此时就需要人工干预。

对于第二种情况，需要在上游系统中建立消息重发机制。可以在上游系统建立一张本地消息表，并将 任务处理过程 和 向本地消息表中插入消息 这两个步骤放在一个本地事务中完成。如果向本地消息表插入消息失败，那么就会触发回滚，之前的任务处理结果就会被取消。如果这量步都执行成功，那么该本地事务就完成了。接下来会有一个专门的消息发送者不断地发送本地消息表中的消息，如果发送失败它会返回重试。当然，也要给消息发送者设置重试的上限，一般而言，达到重试上限仍然发送失败，那就意味着消息中间件出现严重的问题，此时也只有人工干预才能解决问题。

对于不支持事务型消息的消息中间件，如果要实现分布式事务的话，就可以采用这种方式。它能够通过重试机制+定期校对实现分布式事务，但相比于第二种方案，它达到数据一致性的周期较长，而且还需要在上游系统中实现消息重试发布机制，以确保消息成功发布给消息中间件，这无疑增加了业务系统的开发成本，使得业务系统不够纯粹，并且这些额外的业务逻辑无疑会占用业务系统的硬件资源，从而影响性能。

因此，尽量选择支持事务型消息的消息中间件来实现分布式事务，如RocketMQ。

方案4：TCC（两阶段型、补偿型）

TCC即为Try Confirm Cancel，它属于补偿型分布式事务。顾名思义，TCC实现分布式事务一共有三个步骤：

Try：尝试待执行的业务

这个过程并未执行业务，只是完成所有业务的一致性检查，并预留好执行所需的全部资源

Confirm：执行业务

这个过程真正开始执行业务，由于Try阶段已经完成了一致性检查，因此本过程直接执行，而不做任何检查。并且在执行的过程中，会使用到Try阶段预留的业务资源。

Cancel：取消执行的业务

若业务执行失败，则进入Cancel阶段，它会释放所有占用的业务资源，并回滚Confirm阶段执行的操作。

下面以一个转账的例子来解释下TCC实现分布式事务的过程。

假设用户A用他的账户余额给用户B发一个100元的红包，并且余额系统和红包系统是两个独立的系统。

Try

创建一条转账流水，并将流水的状态设为交易中

将用户A的账户中扣除100元（预留业务资源）

Try成功之后，便进入Confirm阶段

Try过程发生任何异常，均进入Cancel阶段

Confirm

向B用户的红包账户中增加100元

将流水的状态设为交易已完成

Confirm过程发生任何异常，均进入Cancel阶段

Confirm过程执行成功，则该事务结束

Cancel

将用户A的账户增加100元

将流水的状态设为交易失败

在传统事务机制中，业务逻辑的执行和事务的处理，是在不同的阶段由不同的部件来完成的：业务逻辑部分访问资源实现数据存储，其处理是由业务系统负责；事务处理部分通过协调资源管理器以实现事务管理，其处理由事务管理器来负责。二者没有太多交互的地方，所以，传统事务管理器的事务处理逻辑，仅需要着眼于事务完成（commit/rollback）阶段，而不必关注业务执行阶段。

TCC全局事务必须基于RM本地事务来实现全局事务

TCC服务是由Try/Confirm/Cancel业务构成的，

其Try/Confirm/Cancel业务在执行时，会访问资源管理器（Resource Manager，下文简称RM）来存取数据。这些存取操作，必须要参与RM本地事务，以使其更改的数据要么都commit，要么都rollback。

这一点不难理解，考虑一下如下场景：

假设图中的服务B没有基于RM本地事务（以RDBS为例，可通过设置auto-commit为true来模拟），那么一旦[B:Try]操作中途执行失败，TCC事务框架后续决定回滚全局事务时，该[B:Cancel]则需要判断[B:Try]中哪些操作已经写到DB、哪些操作还没有写到DB：假设[B:Try]业务有5个写库操作，[B:Cancel]业务则需要逐个判断这5个操作是否生效，并将生效的操作执行反向操作。

不幸的是，由于[B:Cancel]业务也有n（0<=n<=5）个反向的写库操作，此时一旦[B:Cancel]也中途出错，则后续的[B:Cancel]执行任务更加繁重。因为，相比第一次[B:Cancel]操作，后续的[B:Cancel]操作还需要判断先前的[B:Cancel]操作的n（0<=n<=5）个写库中哪几个已经执行、哪几个还没有执行，这就涉及到了幂等性问题。而对幂等性的保障，又很可能还需要涉及额外的写库操作，该写库操作又会因为没有RM本地事务的支持而存在类似问题。。。可想而知，如果不基于RM本地事务，TCC事务框架是无法有效的管理TCC全局事务的。

反之，基于RM本地事务的TCC事务，这种情况则会很容易处理：[B:Try]操作中途执行失败，TCC事务框架将其参与RM本地事务直接rollback即可。后续TCC事务框架决定回滚全局事务时，在知道“[B:Try]操作涉及的RM本地事务已经rollback”的情况下，根本无需执行[B:Cancel]操作。

换句话说，基于RM本地事务实现TCC事务框架时，一个TCC型服务的cancel业务要么执行，要么不执行，不需要考虑部分执行的情况。

TCC事务框架应该提供Confirm/Cancel服务的幂等性保障

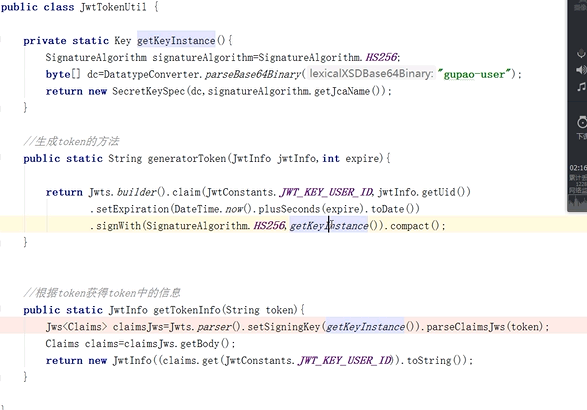
一般认为，服务的幂等性，是指针对同一个服务的多次(n>1)请求和对它的单次(n=1)请求，二者具有相同的副作用。

在TCC事务模型中，Confirm/Cancel业务可能会被重复调用，其原因很多。比如，全局事务在提交/回滚时会调用各TCC服务的Confirm/Cancel业务逻辑。执行这些Confirm/Cancel业务时，可能会出现如网络中断的故障而使得全局事务不能完成。因此，故障恢复机制后续仍然会重新提交/回滚这些未完成的全局事务，这样就会再次调用参与该全局事务的各TCC服务的Confirm/Cancel业务逻辑。

既然Confirm/Cancel业务可能会被多次调用，就需要保障其幂等性。

那么，应该由TCC事务框架来提供幂等性保障？还是应该由业务系统自行来保障幂等性呢？

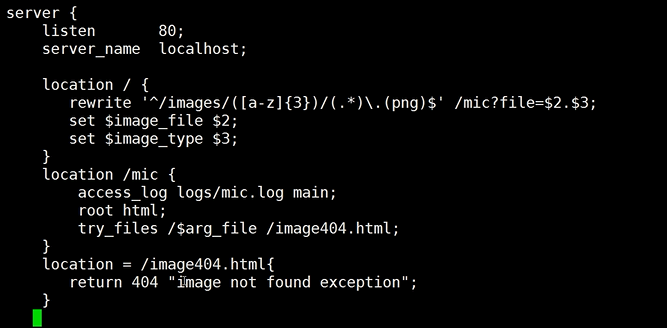
个人认为，应该是由TCC事务框架来提供幂等性保障。如果仅仅只是极个别服务存在这个问题的话，那么由业务系统来负责也是可以的；然而，这是一类公共问题，毫无疑问，所有TCC服务的Confirm/Cancel业务存在幂等性问题。TCC服务的公共问题应该由TCC事务框架来解决；而且，考虑一下由业务系统来负责幂等性需要考虑的问题，就会发现，这无疑增大了业务系统的复杂度。

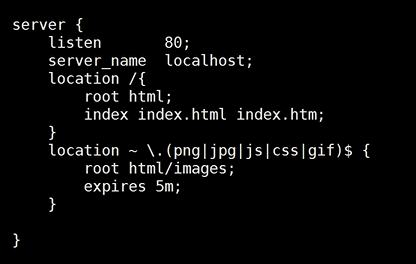


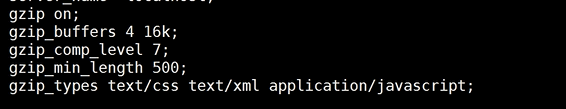
获取token后向请求头添加



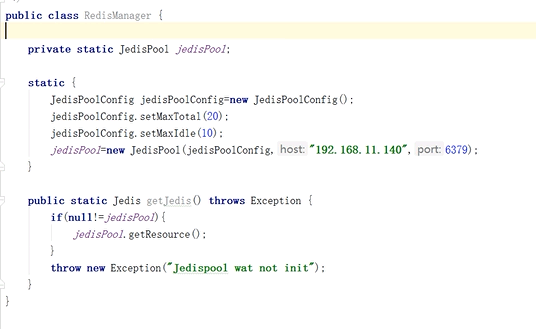
Nginx配置rewrite







Redis实现分布式锁



零拷贝实现

"狼哥，面试又跪了，碰到了知识盲区"  
"哪个？"  
"一面还可以，二面面试官问我零拷贝的原理，懵逼了...这块内容没去研究过"  
"哦，这个知识点，我之前应该有讲过，你没注意到？"  
"这东西工作中用不到，可能被我忽略了"  
"啧啧啧..."  
"哎，有空和我大概讲讲？"

"先从简单开始，实现下这个场景：从一个文件中读出数据并将数据传到另一台服务器上？"

"为啥写这个？"

"你先写"

"行..."

1分钟后

"我写了伪代码"

File.read(file, buf, len);

Socket.send(socket, buf, len);

"这里涉及到了几次数据拷贝？"

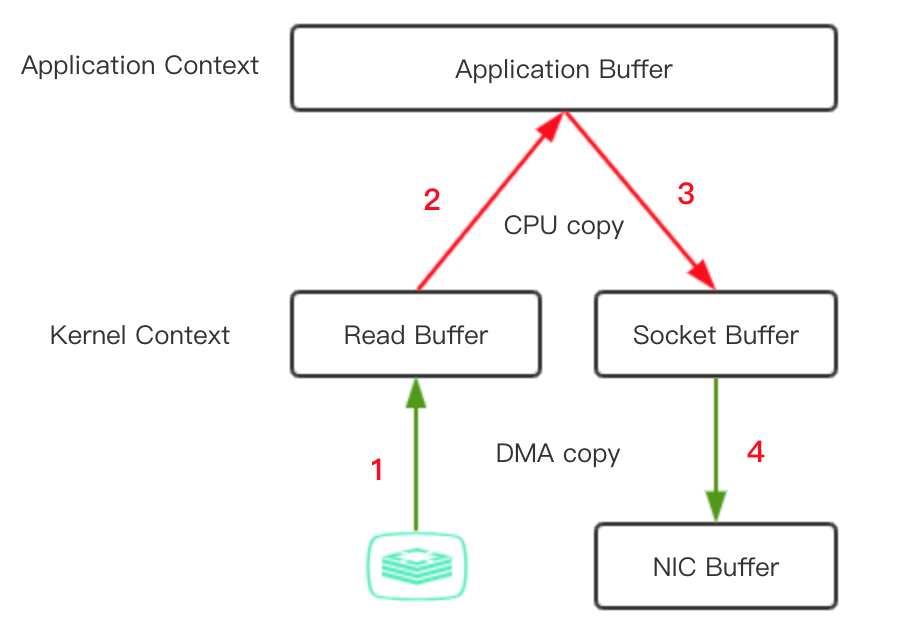
"2次？磁盘拷贝到内存，内存拷贝到Socket？"

"emmm，怪不得挂了，一点不冤。"

"这种方式一共涉及了4次数据拷贝，知道用户态和内核态的区别吗？"

"了解"

"行，文字有点干瘪，你先看这个图"



1、应用程序中调用read() 方法，这里会涉及到一次上下文切换（用户态->内核态），底层采用DMA（direct memory access）读取磁盘的文件，并把内容存储到内核地址空间的读取缓存区。

2、由于应用程序无法读取内核地址空间的数据，如果应用程序要操作这些数据，必须把这些内容从读取缓冲区拷贝到用户缓冲区。这个时候，read() 调用返回，且引发一次上下文切换（内核态->用户态），现在数据已经被拷贝到了用户地址空间缓冲区，这时，如果有需要，应用程序可以操作修改这些内容。

3、我们最终目的是把这个文件内容通过Socket传到另一个服务中，调用Socket的send()方法，这里又涉及到一次上下文切换（用户态->内核态），同时，文件内容被进行第三次拷贝，被再次拷贝到内核地址空间缓冲区，但是这次的缓冲区与目标套接字相关联，与读取缓冲区没有半点关系。

4、send()调用返回，引发第四次的上下文切换，同时进行第四次的数据拷贝，通过DMA把数据从目标套接字相关的缓存区传到协议引擎进行发送。

"在整个过程中，过程1和4是由DMA负责，并不会消耗CPU，只有过程2和3的拷贝需要CPU参与，整明白了？"

"我消化一下..."

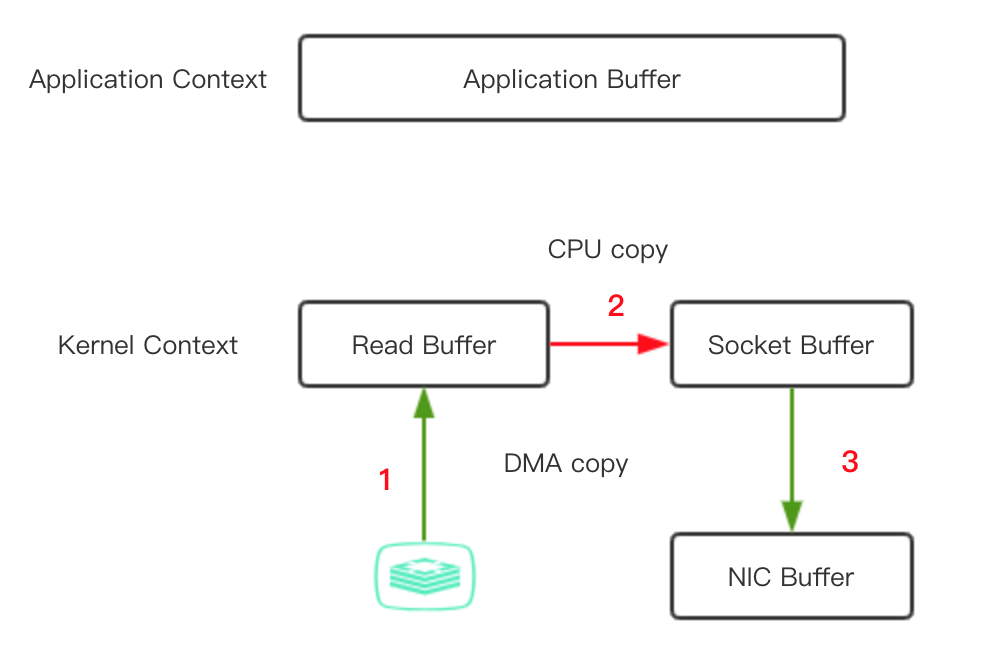
半小时后...

"狼哥，这个过程，感觉好几次的数据拷贝都是多余的，很影响性能啊"

"对，所以才有了零拷贝技术"

"具体咋实现？"

"慢慢来，如果在应用程序中，不需要操作内容，过程2和3就是多余的，如果可以直接把内核态读取缓存冲区数据直接拷贝到套接字相关的缓存区，是不是可以达到优化的目的？"



这种实现，可以有以下几点改进：

* 上下文切换的次数从四次减少到了两次
* 数据拷贝次数从四次减少到了三次（其中DMA copy 2次，CPU copy 1次）

"怎么实现？"

"在Java中，正好FileChannel的transferTo() 方法可以实现这个过程，该方法将数据从文件通道传输到给定的可写字节通道， 上面的file.read()和 socket.send() 调用动作可以替换为 transferTo() 调用"

public void transferTo(long position, long count, WritableByteChannel target);

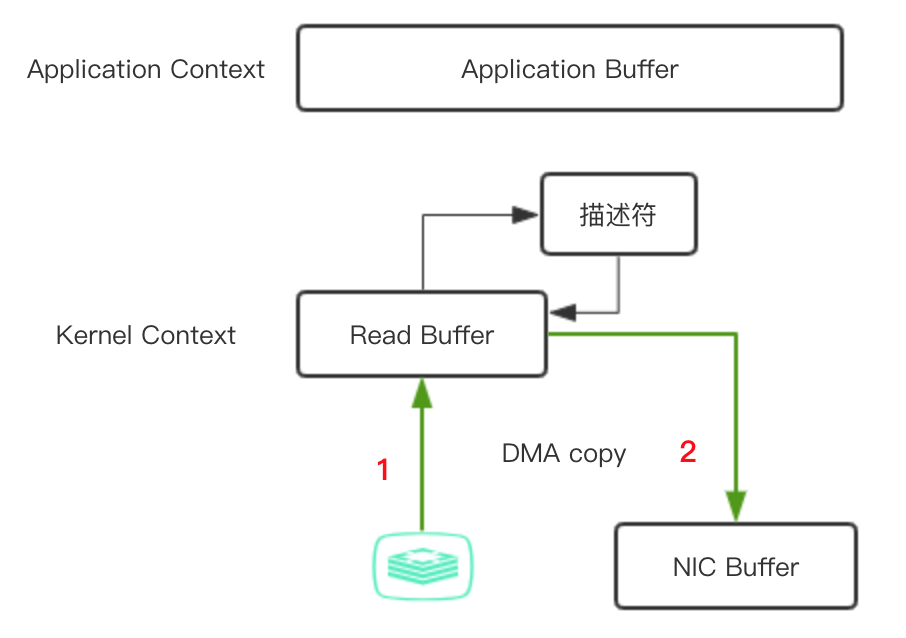
在 UNIX 和各种 Linux 系统中，此调用被传递到 sendfile() 系统调用中，最终实现将数据从一个文件描述符传输到了另一个文件描述符。

"确实改善了很多，但还没达到零拷贝的要求，还有其它黑技术吗？"

"对的，如果底层网络接口卡支持收集操作的话，就可以进一步的优化。"

"怎么优化？"

在 Linux 内核 2.4 及后期版本中，针对套接字缓冲区描述符做了相应调整，DMA自带了收集功能，对于用户方面，用法还是一样的，但是内部操作已经发生了改变：



* 第一步，transferTo() 方法引发 DMA 将文件内容拷贝到内核读取缓冲区。
* 第二步，把包含数据位置和长度信息的描述符追加到套接字缓冲区，避免了内容整体的拷贝，DMA 引擎直接把数据从内核缓冲区传到协议引擎，从而消除了最后一次 CPU参与的拷贝动作。