数据正确性与事务处理

Posted on 2021-11-15
Words count in article: 4.1k | Reading time ≈ 14

数据正确性与事务处理

数据库除了提供数据的存储和查询功能，还提供**事务处理**功能。本章节就要来谈谈事务处理的相关细节。

OLTP和OLAP

首先我们要了解应用的两种模式：OLTP和OLAP

OLTP 即事务处理，OLAP即分析处理，两者在数据库里存储的数据以及对数据库的操作都有不同。

	OLTP(事务型应用)	OLAP(分析型应用)
数据	状态型	历时型
例子	账户余额、购物车、课程表等	购物历史、记录等
对数据库的要求	稳定性、正确性	注重处理能力
对数据库的主要操作	更新	复杂SQL查询

因此，对于事务型应用，我们要求数据库能保证**数据正确性**的同时提供**事务处理**功能。

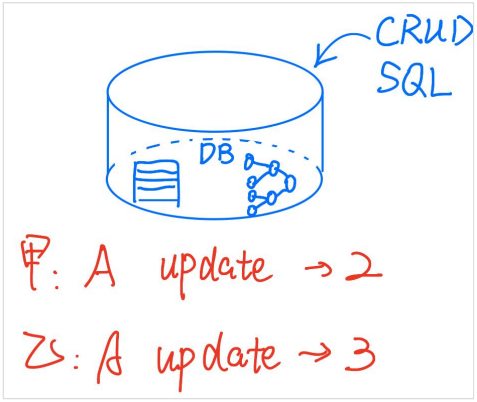
数据的正确性问题

导致数据出现错误的的原因主要有两个

一个是在数据并发的时候若不使用锁的时候，会发生LOST UPDATES，导致数据丢失

另一个是在处理事务的时候数据库突然发生故障导致数据丢失

我们举一个很简单的例子：



在数据库中，A所在的一列是索引，因此当我们修改的A的值的时候需要修改表和索引两个地方，使其保持一致。假设甲和乙几乎同时提交修改请求，如果没有锁的存在，可能表中收到的指令是：先把A修改成2再把A修改成3；而索引收到的指令是：先把A修改成3再把A修改成2. 这样的后果就是索引和表中的值都无法保持一致

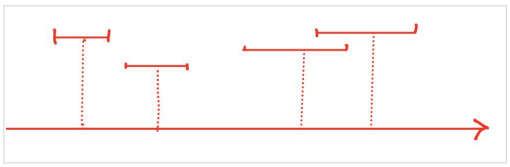
此外，数据库系统还可能在已经修改了表但还没有修改索引的时候突然奔溃，这也会使两个值不一样。

为了规避这些风险，我们需要一种机制去保证数据的正确性

数据库操作的原子性

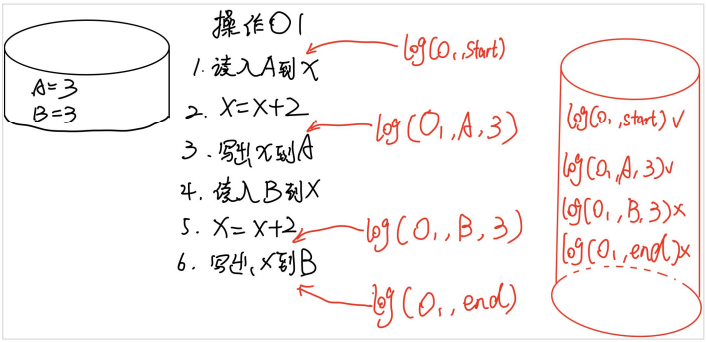
理论上，只要我们对数据库的操作满足原子性，我们就能保证数据的正确性。那么什么是原子性?

我们可以把原子性理解为：不管多少事务是有重合的，我们要把他们实现成好像一瞬间完成的一样(具体实现方法之后再讲)，投影在一个数轴上，这样在数轴上这些点是不会重合的，每一个操作都有先后顺序，不会互相干扰



那么，怎么样实现原子性呢？ 我们提出日志机制

undo日志



现在有一个O1操作如上图所示，一共要执行6个步骤。为了保证数据的准确性，我们在硬盘上另外开了一片区域来记录日志

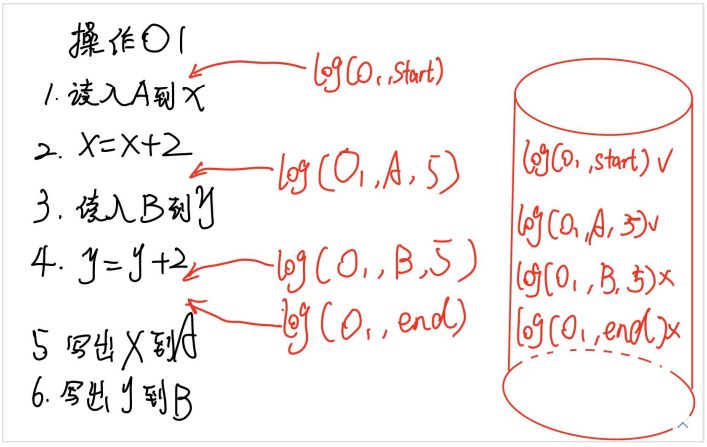
- 在O1操作开始前，会记录日志 `log(O1,start)` 代表O1操作已经开始
- 当我们发现要对硬盘上的数据进行修改的时候，也就是在第二步和第三步之间。则会记录A原始的值，即 `log(O1,A,3)`
- 同理在第五和第六步之间记录 `log(O1,B,3)`
- 最后在O1操作全部完成之后，记录 `log(O1,end)` 代表整个操作结束

这样的当运行到第三句话，发生了故障导致机器重启。那么当数据库系统再次启动的时候会执行恢复程序，即扫描一遍日志，查看是否有异常。当扫描到O1操作的日志的时候，发现 `log(O1,start)` 却没有结束日志。因此判定O1操作没有执行完成，执行回滚操作。由于我们记录了A的原始数据，因此回滚起来也非常方便。

redo日志

上面的这种记录日志的方式叫做undo日志,其主要的功能就是在操作到一半的情况下能够回滚回去。

还有一种记录日志的方式叫做redo日志。其原理如下图所示



运用这种形式操作的步骤也是不同的，是先把数据取出来再修改好，然后一并写回。不是像undo一样改一个写回一个。那么如果在3和4之间发生中断，此时还没有写回到数据表当中，在日志里只记录了两行，那么此时系统不需要做任何事情。数据表中A和B仍然为3

那么如果在5和6之间发生中断，这时候A的值已经被改成5，B的值仍然为3，而此时在日志里，`log(O1,A,5)`和`log(O1,B,5)`已经被记录下来，`log(O1,end)`也记录了下来。系统会判断操作已经结束，然后去检查数据表中的数据是否与日志匹配，结果发现B仍然是3，因此判定出现了异常现象，此时只要把数据表中的B修改成日志当中记录的5即可

所以undo是把数据表恢复成操作开始前的状态(回滚)，redo是把数据表恢复成操作结束后的状态

那么undo和redo有什么优缺点呢?

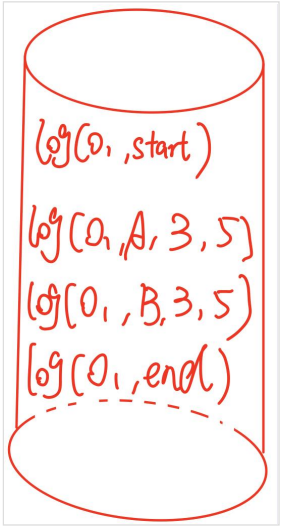
redo：

- 优点： 效率比较高，性能比较好，对数据的修改可以推迟到所有操作结束之后，记录日志和回写硬盘分开进行
- 缺点：对内存消耗比较高，当内存满了以后再添加会导致数据丢失。

undo:

- 优点：对内存的消耗比较小，没修改一个值内存就会直接释放
- 缺点：需要同步记录日志和修改数据的值，导致记录日志和回写硬盘交错进行，是随机访问硬盘的情况，性能较差。

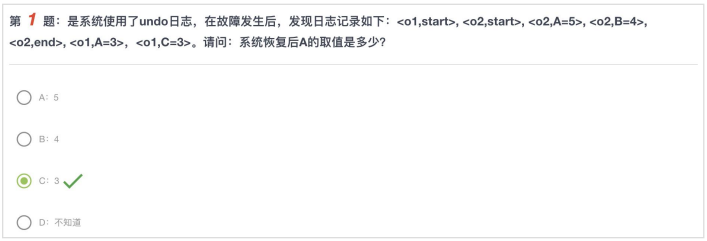
在数据库中，我们会把这两种日志结合在一起，称为 undo/redo 日志。这种日志不但包含了每个元素修改之前的值，还包含了元素修改之后的值。这样就可以规避undo/redo日志各自的缺点。如下：



一旦发生故障，就有两种选择，如果发现日志已经完整了，我们可执行redo操作；如果日志并不完整，可以执行undo操作。

运用undo/redo日志，我们看到每个操作都可以看做是时间轴上的某一个点，即使发生故障，故障也可以看做是时间轴上的一个点，只不过之后会做undo/redo罢了，也就是我们利用日志可以保证在发生系统故障的时候数据操作的原子性

例题



这个是Undo日志，O2已经完成了但是O1还没有完成，因此undo会把A回滚成原来的样子，即3

对于B，在Undo日志下，我们要 先记录元素原本的值，然后再讲修改后的值写回硬盘

谁最后结束，磁盘上的值就是谁操作的值。

对于D， 操作o1 是最后结束的，因此最后应该 `write(A=5)` 而不是 `write(A=6)`

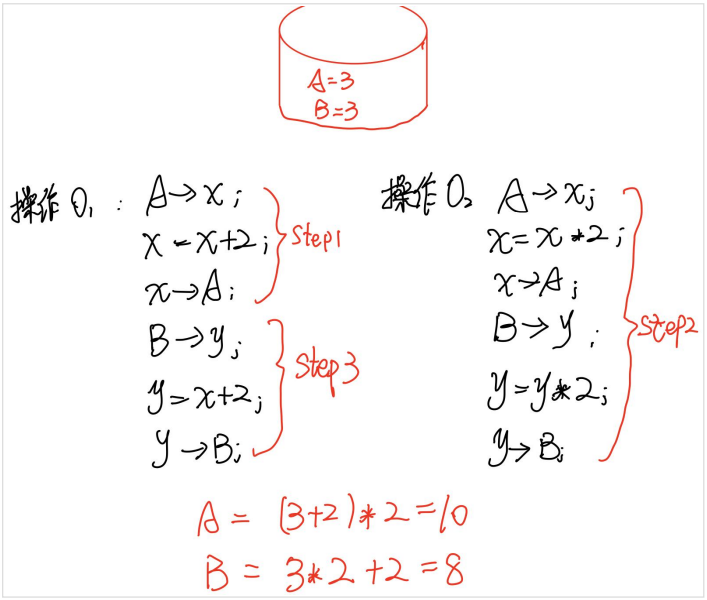
对于C，由于undo的存在，在执行到一般的时候，可能会发生回滚导致已经结束的CRUD操作撤销。不一定会顺利完成

课程中提到了Undo/Redo日志，但没有介绍其具体工作机制。请思考它的具体工作机制。假设有如下一个操作，请使用你想到的Undo/Redo日志机制阐述整个操作的执行过程。

```
Begin
        read A from Disk into x;
        x = x+5;
        write x to A on Disk;
        read B from Disk into y;
        y = y+x;
        write y to B on Disk;
End
```

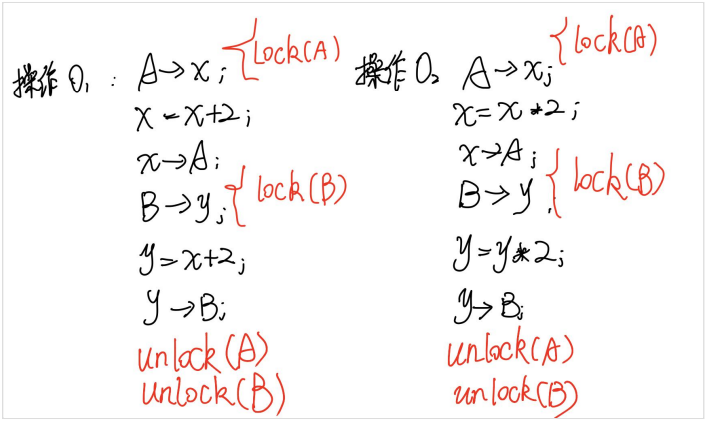
并发控制机制

然而，我们还没有解决如何在并发时保持数据的原子性。因此当两个线程同时处理两个操作的时候，可能会出现数据错误的情况，如下：



假设CPU一开始收到O2的前半部分，修改了A的值为5，这时候发生了中断，转而去执行操作O2的所有指令，这时A变成了10，B变成了6；执行完O2后回过来执行O1，最终B为8

我们发现本应该是相同的A和B，但是当线程之间发生调度的时候，会出现AB不相等的情况。为了解决这个问题，可以使用**锁**或者**时间戳**机制



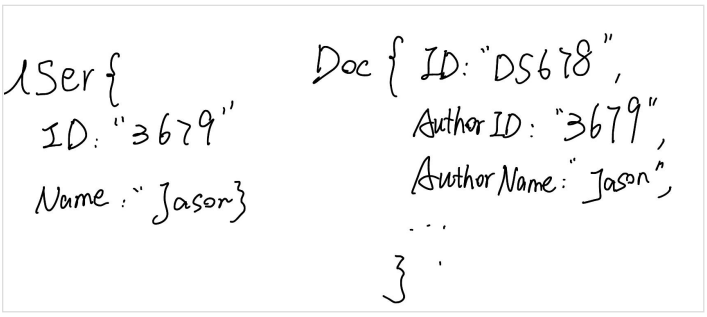
如上图，在操作1读取A之前，就要给A上锁，这样即使发生了线程调度，在运行了三句指令后去执行操作O2，CPU会发现A上的锁还没有解开，因此无法读取A的值。

这样一来，只有当操作O1完全结束，将锁完全释放之后，才能继续执行操作O2

上面所讲的日志和锁保证了数据库内单步操作的原子性，但是在事务型应用中，事务是由一系列的访问构成的，我们要保证整个事务的正确性，因此需要数据库提供事务处理接口。接下来我们我们就来学习这方面的知识。

应用层面的数据正确性

我们首先用一个例子来说明为什么应用中的事务逻辑会出现正确性问题：



那么当我们要修改账户的名称为Bob的时候，编写的程序应该如下：

```
Update —> User.ID = '3679'
Set name = 'Bob';
For(Doc.AuthorID ='3679'){
        set Doc.AuthorName = 'Bob',}
END for
```

这时候，如果在第二行和第三行之间计算机发生了故障，软件重启后，就发现仅仅修改了User文档中的名字，而没有吧博客文章文档里面的每个名字都进行修改。这就出现异常了，但是这和数据库本身并没有问题，因此我们要提出新的机制来解决应用层面的数据正确性问题。

用标志位防止数据异常

和记录日志类似，我们也有一种机制，能让程序记起来自己在哪里中断，已经执行了什么，还未执行什么。

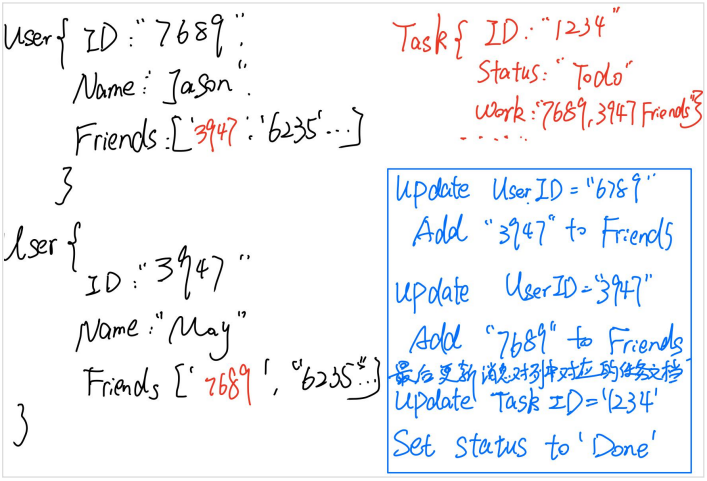
因此我们可以在User文档中新增一个标志位namesync,当user名字被修改的时候，会将其设为ToDo，用于告诉系统还有工作没做完。当所有博客文章的作者名称都修改完之后，才会更新标志位为 Done，如下：

```
Update —> User.ID = '3679'
Set name = 'Bob';
Set namesync = 'ToDo'   //设置标志位为To DO
For(Doc.AuthorID ='3679'){
        set Doc.AuthorName = 'Bob',}
END for
Update ->User.ID = '3679' Set namesync ='Done' //重新更新标志位为Done
```

这样，当故障发生，系统重启之后，会去检查namesync的值，如果是todo，那么就会重新执行一遍改名操作。最后再将namesync置为Done即可

用消息队列防止数据异常

之前我们是在一个现有文档里面添加一个标志位来保证正确性。现在我们可以新建一个任务文档来存放将要做的任务，我们称之为消息队列。示意图如下：



当我们要做这个互相加好友的操作的时候，消息队列中新插入一个任务，把status置为Todo，意为还未完成。然后，当双方都完成更新操作，才将status置为Done

注意了，若要用消息队列来保证应用的数据正确性，首先要满足一个前提——操作的幂等性，也就是不管操作几次，最终的结果是一样的。

比如，向集合里插入一个元素、从集合里删除一个元素、对一个元素进行赋值，都属于幂等操作。但是对一个数据进行自增，这就不幂等了因为每次操作和原来的值都不一样

事务处理的概念

上面两种操作是在APP层面保证应用的数据正确性，但是这给程序员增加了不少负担。

此外，可能有多个程序员编写多个程序在访问同一张表。一个程序员可能能让自己的程序保持数据的正确性，但是他不能要求别的程序员也这样。因此程序会变得非常复杂。

因此，我们想能不能把这些工作交给数据库去做呢？这就叫做事务处理。

什么是事务(Transaction)？我在mysql博客中已经介绍过了，事务就是一段可以实现特定功能的SQL代码，在执行失误的时候，所有的语句都必须成功执行，否则事务就会失败。

Transaction有几个特性：

第一个是原子性(Atomicity)，也就是不可分割的，一个事务不管其包含多少语句都是一个完整的整体。除非事务中所有的语句都成功执行，Transaction才算执行成功，否则所有修改都会回滚。

第二个特性是一致性(Consistency)，也就是说我们使用了Transaction之后，我们的数据库永远保持一致。

第三个特性是隔离性(Isolation)。也就是说 Transaction之间是相互隔离的，特别是他们要修改同一个数据的时候。他们之间不受影响。如果多个Transaction要修改同一个数据，这条记录就会被锁定，每次只能有一个Transaction有权修改。其他的Transaction需要等待这个Transaction执行完毕

最后一个是持久性(Durability), 意思是一但一个Transaction被提交，它的修改就是永久性的，无法撤销，其他任何崩溃的情况（停电、宕机),也不会影响数据的修改

我们称这四个属性为 ACID

使用了事务，就好像每个事务都只是数轴上的一个点，不会互相干扰。

合理使用事务

现在我们举几个例子来说明事务的功能

首先是订电影票的流程:

- 获取空闲座位
- 展示座位
- 用户选择座位
- 用户提交订票请求
- 将座位分配给用户

一种朴素的思想是将这五步都包含在一个Transaction里面，虽然能保证事务的完整性，但是这就使得该订票平台每次只能有一个人订票。因为数据库在一开始就会把所有的位置上锁，当整个订票过程结束后才解锁。这样就无法满足高并发的情况

因此更好的方法是在第四步和第五步之间 Beigin Transaction。但当电影特别火的时候，还是有可能同时多个人抢一个座位的情况。因此我们不能简单的将座位直接分配给客户，而是在分配前做一个判断

> - 获取空闲座位
> - 展示座位
> - 用户选择座位
> - 用户提交订票请求
>
> Begin Transaction
>
> - 座位是否空闲
>   - IF空闲，分配给用户
>   - ELSE不空闲，数据库abort，前端return SORRY
>
> End Transaction¢

因此，我们引出了一个创建事务的原则：事务应该是短小的。否则会导致整个程序的性能下降

现在我们用另外一个例子：

购买商品的过程：

- IF 账户余额 <= 商品价格，Then 取消
- 调用商品运输服务
- 账户余额 -= 商品价格

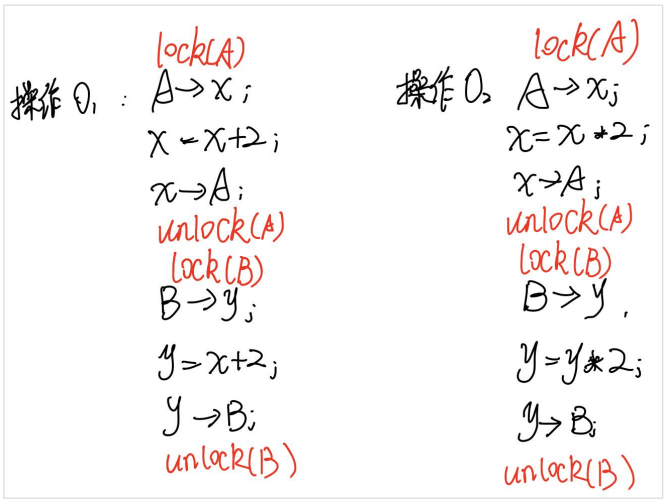
我们当然可以把这三步都放在一个Transaction里面，但是我们会发现，好像第二步和第一步、第三步的关系没有那么紧密，他并没有处理的用户账户相关的问题。因此更好的解决办法如下：

> Begin Transaction
>
> - IF 账户余额 <= 商品价格，Then 取消
> - 账户余额 -= 商品价格
>
> End Transaction
>
> - 调用商品运输服务

我们要根据不同的情境来设计不同的事务

例题

我们要注意一句话：在拿到所有锁之前，不能释放任何一个锁。比如下面这个例子



当O1运行到 `x->A` 后，如果释放了锁，这时候如果发生了线程调度，调到O2，马上锁住并全部运行完成。这就会造成数据不同步。因此即使手上有1000把锁，如果事务中还有锁没拿到，那么就一把锁也无法释放。

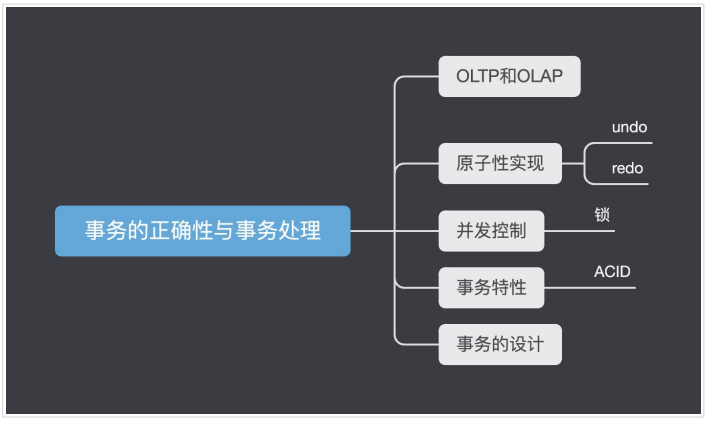
因此，这也告诉我们事务要设计的短一些。

对于A，在读取A元素之前必须先给A上锁，因此错误

对于C，同理

对于D，我们要记住：先Start再锁，先End再释放。因为如果先锁再记录日志，这种逻辑很怪，因为在锁的时候我并不知道我的目标元素是什么

总结



----------本文结束，感谢您的阅读----------

数据管理系统

关系数据库设计

《数据科学与工程算法基础》实践报告