事务的使用方法

面向事务型应用的软件系统，如在线订票系统，电商购物系统等，依靠数据库管理系统的事务处理机制来保证系统的正确性。随着互联网和移动互联网地快速发展，这些应用软件系统遍布人们的衣食住行，它们需要在保证数据正确性的基础上重视用户的体验。通常，如果用户在点击请求按钮后，系统能够在3秒内做出应答，用户是比较满意的；如果系统在5秒内做出应答，用户是能够忍受的；但是如果系统在8秒后做出应答，则用户是不能忍受的。这是应用软件性能测试中常说的358原则。要想提高应用软件的请求处理性能，一方面可以优化数据库管理系统的日志机制和并发控制机制，另一方面是优化构建应用的事务逻辑。本小节主要介绍在软件应用层面如何合理使用事务来构建高效的业务逻辑。

短事务

以电影票订票业务为例来讲解如何合理地使用短事务。通常，用户使用在线购票APP购买电影票，它的业务流程包括：

- 首先选取一个电影的场次，获取该场次的空闲座位；
- 如果该场次没有空闲座位或者空闲座位不符合用户心意，则退出，否则用户选择想购买的座位，并点击提交按钮；
- 将座位分配给用户，订票成功。

为了便于说明，上述流程省略了用户付款过程，只专注于用户订票流程。如何来构建电影票预订业务的事务逻辑呢？最简单，最显而易见的一种方法是将上述所有流程封装在一个事务内，该业务的逻辑如下：

```
低效的电影票预订业务逻辑：
BEGIN TRANSACTION;
SELECT sid FROM MovieSeats WHERE mid=? AND mtime=? AND state = '未售' FOR UPDATE;
/*获取空闲座位*/
IF (sid == null || quitbutton == 'true')    /*如果没有空闲座位或者用户点击退出按钮则
回滚*/
    ROLLBACK;
ELSE
{
    /*用户选择座位*/
    UPDATE Movieseats SET state = '已售' where sid = ?; /*根据用户选择的座位号，将该
座位修改为“已售”*/
    COMMIT;  /*订票成功！*/
}
```

这种设计能够保证应用软件系统的正确性，不会出现多个用户预订到同一个座位的情况。但是，这种设计使得业务性能不够高效。当数据库管理系统收到该电影票预订事务时，在读取空闲座位后，为了保证系统正确性会对所有的空闲座位加写锁，直到所有更新操作结束之后才释放空闲座位上的写锁。假如这部电影非常火，同一时间有大量的用户在预订电影票，一旦某一个用户U获得空闲座位的写锁之后，其余用户只能等待直到用户U选好座位并释放空闲座位的写锁。如果用户U在选择座位时一直犹豫不决，同朋友商量了很久（比如5分钟）才确定，那么在此期间其他用户也将无法读取该电影场次的空闲座位，只能一直等待。当出现这种情况时，用户的体验是极差的，不可忍受的。显然，这种事务设计不够合理，一个用户的迟疑会影响其他所有人的订票流程。

那么，一个高效的事务设计应该是怎样的呢？通常，**事务应尽量的短小，只包含业务的核心处理逻辑，人机交互以及网络交互等要尽量的排除在事务之外。**因为，人机交互时间以及网络交互时延都是不可控的，一旦时间过长，就会出现阻塞从而影响应用软件性能。上例电影票订票业务中，高效的方法是只将用户选择好座位并点击提交按钮之后的流程封装成事务，也就是事务只包含流程的第三步，即更新用户选择座位的状态。该业务的逻辑如下：

高效的电影票预订业务逻辑：

```
SELECT sid FROM MovieSeats WHERE mid=? AND mtime=? AND state = '未售' ;   /*获取空闲座位*/
IF (sid == null || quitbutton == 'true')    /*如果没有空闲座位或者用户点击退出按钮则退出*/
    QUIT();
ELSE
{
    /*用户选择座位*/
    BEGIN TRANSACTION;
    UPDATE Movieseats SET state = '已售' where sid = ? AND state = '未售'; /*分配座位给用户*/
    IF (error)
    {
        ROLLBACK; /*用户选择的座位已出售。订票失败！*/
    }
    ELSE
    {
        COMMIT; /*订票成功！*/
    }
}
```

上述业务逻辑中，所有的用户可以同时获取空闲座位并选择座位，不会出现因某一用户迟疑而导致其他用户被阻塞的情况。但是，存在不同用户选择同一座位的情况。这种并发请求的正确性由数据库管理系统的并发控制技术来保证，即同一时刻只会有一个用户订票成功，而其他用户订票失败，从而保证了不同的用户不会购买到相同的座位。在电影票预定业务逻辑2中，虽然可能存在订票失败的情况，但是它的性能远远高于电影票预定业务逻辑1，且用户的体验也会更好。

事务拆分

以电商平台的购物业务为例来讲解如何进行事务拆分。通常，用户通过电商APP进行购物、支付和获取货物，它的大致业务流程包括：

- 用户选取喜欢的商品，获取商品的价格和库存量；
- 用户选择购买商品数量，点击提交；
- 查询用户账户余额，如果余额不足则取消，否则生成购物单；
- 商家处理商品出库，生成商品物流信息单；
- 用户进行商品支付交易，完成商品购买。

如果所有的业务流程都封装在一个事务中，它的业务逻辑如下：

```
低效的购物业务逻辑：
BEGIN TRANSACTION;
SELECT price, stock FROM Goods WHERE id=? FOR UPDATE;   /*获取所选商品的价格和库存量
*/
/*用户设置购买商品数量，商品收件地址、收件人等信息，然后点击提交按钮*/
SELECT balance FROM Account WHERE uid=?  FOR UPDATE; /*读取用户账户中的余额*/
IF (balance <  price * num)    /*如果账户余额不足，则回滚*/
    ROLLBACK;  /*购物失败*/
ELSE
{
    UPDATE Goods SET stock = stock - num where id = ?; /*更新商品的库存量*/
    INSERT INTO Order VALUES('','','','','','','','')  /*新增一条商品订单*/
    INSERT INTO Logistics VALUES('','','','','','','',''); /*新增一条物流信息*/
    /*等待商品出库之后，修改订单的状态*/
    UPDATE Order SET state = '出库' WHERE oid=?;
    UPDATE Account SET balance = balance - price * num WHERE uid = ? /*用户进行支付
*/
    COMMIT;
}
```

这种事务的设计方式并不合理，其中商品出库，商家添加商品物流信息步骤与用户购物的业务逻辑并不紧密，它属于用户购买和支付之后的商品运输流程。将运输流程加入购物流程中，增加了事务的长度，也使得相关数据上的锁持有更长的时间，从而影响了整个应用软件购物功能的性能。

一种更合适的方式，是将上述一个大事务拆分成两个小事务，即商品事务以及商品物流事务。它的业务逻辑为：

```
高效的购物业务逻辑：

商品购买事务：
BEGIN TRANSACTION;
SELECT price, stock FROM Goods WHERE id=? FOR UPDATE;   /*获取所选商品的价格和库存量
*/
/*用户设置购买商品数量，商品收件地址、收件人等信息，然后点击提交按钮*/
SELECT balance FROM Account WHERE uid=?  FOR UPDATE; /*读取用户账户中的余额*/
IF (balance <  price * num)    /*如果账户余额不足，则回滚*/
    ROLLBACK;  /*购物失败*/
ELSE
{
    UPDATE Goods SET stock = stock - num where id = ?; /*更新商品的库存量*/
    INSERT INTO Order VALUES('','','','','','','');  /*新增一条商品订单*/
    UPDATE Account SET balance = balance - price * num WHERE uid = ? /*用户进行支付
*/
    COMMIT;
}

商品物流事务：
BEGIN TRANSACTION;
SELECT * FROM Order WHERE date = ? AND state = '未出库' ; /*获取某一天所有未出库的订
单，通常获取前一天的订单*/
```

```
INSERT INTO Logistics VALUES('','','','','','','',''); /*批量生成商品的物流信息*/
/*等待商品出库之后，修改订单的状态*/
UPDATE Order SET state = '出库' WHERE date = ? AND state = '未出库';
COMMIT;
```

上述方式是将业务逻辑拆分成多个内部逻辑紧密相关的子业务，每个子业务对应一类事务，不同类事务之间按顺序进行，如商品要先购买，支付成功之后，才进行物流运输。这种事务拆封的方式不仅能够使业务逻辑更加清晰，而且能够提高整个业务的处理性能。

在真实的应用开发中，程序员会遇到各种各样的，比上述两个例子更加复杂的业务逻辑。如何合理地设计事务，实现高效的应用软件呢？通常，可以遵循两个规则：（1）事务尽量短小；（2）事务内部的操作在业务上要紧密相关。基于这两个规则之后，再结合实际情况进行调整和优化。