第十二章　第十二次作业

1. (单选题) 当事务的隔离级别设置为 read committed 时

   A. 可以避免幻读　　B. 可以避免脏读

   C. 可以避免更新丢失　　D. 可以避免不可重复读

2. (单选题) 在以下描述中，关于 SQL-92 事务隔离级别的说法哪一项是正确的?

   A. 可串行化 (Serializable) 级别保证了所有事务的执行是串行化的，允许读取未提交数据。

   B. 可重复读 (Repeatable Read) 级别保证了事务在执行过程中读取的数据在整个事务中都不会发生变化，但允许其他事务修改数据的值。

   C. 已提交读 (Read Committed) 级别保证了事务只会读取已提交的数据，但在同一事务内，读取同一数据

   D. 未提交读 (Read Uncommitted) 级别保证事务只能读取其他事务已提交的数据，并且禁止读取未提交的数据。

3. (单选题) 在一个数据库管理系统中，假设有两个事务 T1 和 T2，T1 对数据项 X 加了共享锁 (S-lock)，而 T2 试图对同一数据项 X 加锁。以下哪种情况是允许的?

   A. 允许 T2 获取排他锁 (X-lock)，因为 T1 已经获得了共享锁 (S-lock)。

   B. 不允许 T2 获取共享锁 (S-lock)。

   C. T2 可以获取排他锁 (X-lock)，但需要等待 T1 的共享锁 (S-lock) 释放。

   D. 允许 T2 获取排他锁 (X-lock)，不需要等待。

4. (单选题) 在两阶段封锁协议 (2PL) 中，关于串行化性的描述，以下哪项是正确的?

   A. 两阶段封锁协议能够保证事务执行顺序是可串行化的，但不能避免死锁的发生。

   B. 两阶段封锁协议能够防止死锁的发生，同时保证事务执行顺序是可串行化的。

   C. 两阶段封锁协议无法保证事务执行顺序是可串行化的，但能够避免死锁。

   D. 两阶段封锁协议不保证事务执行顺序是可串行化的，也不能避免死锁。

5. (单选题) 在 MySQL 默认状态下，现有空表 t1，执行以下语句，处理结果是 () insert into t1 values(1,1); create table t2 as select * from t1; insert into t2 values(2,2); rollback;

   A. t1 表有 1 条数据 (1,1)，t2 表为空

   B. t1 和 t2 表均为空

   C. t1 表有 1 条数据 (1,1)，t2 表有 1 条数据 (1,1)

   D. t1 表有 1 条数据 (1,1)，t2 表有 2 条数据 (1,1) 和 (2,2)

6. (单选题) 在以下 SQL 事务管理相关的语句中，哪一项描述是正确的？START TRANSACTION; UPDATE account SET balance = balance - 100 WHERE account_id = 1; INSERT INTO transaction_log (account_id, transaction_type, amount) VALUES (1, 'Debit', 100); COMMIT;

   A. 事务中的 UPDATE 和 INSERT 操作在 COMMIT 之前就生效。

   B. 该事务会自动提交所有更改，并且不需要显式调用 COMMIT。

   C. 只有在 COMMIT 时，所有更改才会永久保存到数据库中。

   D. 如果在事务执行期间发生错误，数据库会自动回滚所有更改，即使没有调用 ROLLBACK。

7. (简答题) 请创建一个新表 instructor(ID varchar(10), name varchar(20), dept_name varchar(10), salary int)，设置 ID 为主键，并插入一条教师信息 ('12121', 'Wu', 'Comp. Sci.', 5000)，进行 MySQL 隔离级别验证实验。请按照如下步骤完成实验并回答相应问题，在完成实验之后请删除该 instructor 表。

   1）将 instructor 表中编号 12121 的名为 Wu 的教工的 salary 设置为 10000 元

   2）同时开两个命令行界面，称为 A 和 B，在 A、B 两个窗口都执行下面的语句，设定隔离级别为“读未提交”。

   set transaction_isolation='read-uncommitted';

   可以执行 select @@session.transaction_isolation; 验证设定是否已经成功。

   3）在 A 窗口执行如下语句：

   begin;

   select * from instructor;

   可以看到所有教工的信息，编号 12121 的名为 Wu 的教工的 salary 当前值应该为 10000

   4）在 B 窗口执行如下语句：

   begin;

   update instructor set salary = salary - 500 where id= 12121;

   select * from instructor;

   5）在 A 窗口执行如下语句：

   select * from instructor;

   1. 请回答如下问题：

   Q1：是否可以看到编号 12121 的名为 Wu 的教工的 salary 值被 B 窗口修改但未提交的结果，回答是/否。

   否。

   Q2：这种现象的名称是？

   脏读。

验证一下

A 窗口执行的：

```
create table instructor(
    ID varchar(10),
    name varchar(20),
    dept_name varchar(10),
    salary int,
    primary key (ID)
);

insert into instructor
value ('12121', 'Wu', 'Comp. Sci.', 5000);

update instructor set salary = 10000 where ID = '12121' and name = 'Wu';

set transaction isolation level read uncommitted ;

begin ;
select * from instructor;
```

```
+-------+------+------------+--------+
| ID    | name | dept_name  | salary |
+-------+------+------------+--------+
| 12121 | Wu   | Comp. Sci. |  10000 |
+-------+------+------------+--------+
```

B 窗口执行的：

```
set transaction isolation level read uncommitted ;

begin ;
update instructor set salary = salary - 500 where ID = '12121';
select * from instructor;
```

```
+-------+------+------------+--------+
| ID    | name | dept_name  | salary |
+-------+------+------------+--------+
| 12121 | Wu   | Comp. Sci. |   9500 |
+-------+------+------------+--------+
```

A 窗口再执行：

```
select * from instructor;
```

```
+-------+------+------------+--------+
| ID    | name | dept_name  | salary |
+-------+------+------------+--------+
| 12121 | Wu   | Comp. Sci. |  10000 |
+-------+------+------------+--------+
```

6）在 B 窗口执行如下语句：

rollback;

7）在 A 窗口执行如下语句：

update instructor set salary = salary - 500 where id= 12121;

select * from instructor;

2. 请回答如下问题：

Q：当前编号 12121 的名为 Wu 的教工的 salary 值为多少?

9500

验证一下

```
+-------+------+------------+--------+
| ID    | name | dept_name  | salary |
+-------+------+------------+--------+
| 12121 | Wu   | Comp. Sci. |   9500 |
+-------+------+------------+--------+
```

8）在 A 窗口执行如下语句：

rollback;

select * from instructor;

9）在 A、B 两个窗口都执行下面的语句，设定隔离级别为“读已提交”。

set transaction_isolation='read-committed';

可以执行 select @@session.transaction_isolation; 验证设定是否已经成功。

10）在 A 窗口执行下面语句：

begin;

select * from instructor;

11）在 B 窗口执行下面语句：

begin;

select * from instructor;

update instructor set salary = salary - 500 where id= 12121;

12）在 A 窗口执行如下语句：

select * from instructor;

3. 请回答如下问题：

Q1：当前编号 12121 的名为 Wu 的教工的 salary 值为多少?

10000。

验证一下

```
+-------+------+------------+--------+
| ID    | name | dept_name  | salary |
+-------+------+------------+--------+
| 12121 | Wu   | Comp. Sci. |  10000 |
+-------+------+------------+--------+
```

Q2：是否可以看到编号 12121 的名为 Wu 的教工的 salary 值被 B 窗口修改但未提交的结果，回答是/否

否。

13）在 B 窗口执行下面语句：

commit;

select * from instructor;

14）在 A 窗口执行如下语句：

select * from instructor;

commit;

4. 请回答如下问题：

Q1：当前编号 12121 的名为 Wu 的教工的 salary 值为多少?

9500。

验证一下

```
+-------+------+------------+--------+
| ID    | name | dept_name  | salary |
+-------+------+------------+--------+
| 12121 | Wu   | Comp. Sci. |   9500 |
+-------+------+------------+--------+
```

Q2：是否可以看到编号 12121 的名为 Wu 的教工的 salary 值被 B 窗口修改并已提交的结果，回答是/否

是。

Q3：A 窗口当前两次的 select * from instructor; 查询结果是否一致?

不一致。

Q4：这种现象的名称是?

不可重复读。

15）在 A、B 两个窗口都执行下面的语句，设定隔离级别为 “可重复读”。

set transaction_isolation='repeatable-read';

可以执行 select @@session.transaction_isolation; 验证设定是否已经成功。

16）在 A 窗口执行下面语句：

begin;

select * from instructor;

16）在 B 窗口执行下面语句：

begin;

select * from instructor;

update instructor set salary = salary - 500 where id= 12121;

commit;

select * from instructor;

17）在 A 窗口执行如下语句：

select * from instructor;

5. 请回答如下问题：

Q1：当前编号 12121 的名为 Wu 的教工的 salary 值为多少?

9000。

Q2：是否可以看到编号 12121 的名为 Wu 的教工的 salary 值被 B 窗口修改并已提交的结果，回答是/否

否。

Q3：A 窗口当前两次的 select * from instructor; 查询结果是否一致?

是。

18）在 A 窗口执行如下语句：

update instructor set salary = salary - 500 where id= 12121;

commit;

select * from instructor;

6. 请回答如下问题：

Q1：当前编号 12121 的名为 Wu 的教工的 salary 值为多少?

8500。

19）在 A 窗口执行如下语句：

begin;

select * from instructor;

20) 在 B 窗口执行下面语句：

begin;

insert into instructor values(99999,'test','Comp. Sci.',10000);

commit;

select * from instructor;

21）在 A 窗口执行如下语句：

select * from instructor;

7. 请回答如下问题：

Q1：是否可以看到 B 窗口新插入并已提交的结果，回答是/否

否。

验证一下

B 窗口：

```
+-------+------+------------+--------+
| ID    | name | dept_name  | salary |
+-------+------+------------+--------+
| 12121 | Wu   | Comp. Sci. |   8500 |
| 99999 | test | Comp. Sci. |  10000 |
+-------+------+------------+--------+
```

A 窗口：

```
+-------+------+------------+--------+
| ID    | name | dept_name  | salary |
+-------+------+------------+--------+
| 12121 | Wu   | Comp. Sci. |   8500 |
+-------+------+------------+--------+
```

22）在 A 窗口执行如下语句：

insert into instructor values(99999,'test','Comp. Sci.',10000);

8. 请回答如下问题：

Q1：A 窗口是否能插入此条数据，回答是/否

否。

验证一下

```
ERROR 1062 (23000): Duplicate entry '99999' for key 'PRIMARY'
```

Q2：这种现象的名称是?

幻读。

23）在 A 窗口执行如下语句：

update instructor set salary = 6666 where id=99999;

select * from instructor;

commit;

9. 请回答如下问题：

Q1：A 窗口是否更新了一条自己会话中查不到的数据，回答是/否

是。

验证一下

```
+-------+------+------------+--------+
| ID    | name | dept_name  | salary |
+-------+------+------------+--------+
| 12121 | Wu   | Comp. Sci. |   8500 |
| 99999 | test | Comp. Sci. |   6666 |
```

```
+-------+------+-----------+--------+
```

8. (简答题) 假设你正在开发一个在线售票系统，该系统需要处理多个用户同时购买同一场次电影票的情况。为了防止超卖现象（即卖出比实际可用数量更多的票），你需要实现一个安全的购票流程。请按照如下步骤完成实验并回答相应问题，在完成实验之后请删除该 Tickets 表。

注：mysql 中添加锁代码为

```
-- 共享锁
SELECT remaining_tickets INTO remaining FROM Tickets WHERE id = movie_id LOCK
↪  IN SHARE MODE;

-- 排他锁
SELECT remaining_tickets INTO remaining FROM Tickets WHERE id = movie_id FOR
↪  UPDATE;
```

步骤 1：创建测试环境

在数据库中创建一张表 Tickets ，包含字段 id （主键）、movie_name （电影名称）、remaining_tickets （剩余票数）。

```
CREATE TABLE Tickets (
    id INT AUTO_INCREMENT PRIMARY KEY,
    movie_name VARCHAR(255) NOT NULL,
    remaining_tickets INT NOT NULL
);

INSERT INTO Tickets (movie_name, remaining_tickets) VALUES
('Movie A', 10),
('Movie B', 10),
('Movie C', 10);
```

步骤 2：编写购票存储过程及并发脚本存储过程

```
^^IDELIMITER $$

-- 创建一个没有使用锁机制的存储过程 BuyTicketsNoLock
drop PROCEDURE if exists BuyTicketsNoLock;
CREATE PROCEDURE BuyTicketsNoLock(IN movie_id INT, IN tickets_to_buy INT)
BEGIN

    DECLARE remaining INT;
    DECLARE new_remaining INT;
 START TRANSACTION;
    -- 不使用锁，直接查询剩余票数
```

```
    SELECT remaining_tickets INTO remaining FROM Tickets WHERE id = movie_id;

    -- 检查是否有足够的票
    IF remaining >= tickets_to_buy THEN
        SET new_remaining = remaining - tickets_to_buy;

        -- 更新剩余票数，这里也没有使用锁
        UPDATE Tickets SET remaining_tickets = new_remaining WHERE id =
        ↪  movie_id;

        SELECT CONCAT('成功购买 ', tickets_to_buy, ' 张票，剩余 ',
        ↪  new_remaining, ' 张') AS result;
    ELSE
        SELECT '购票失败：库存不足' AS result;
    END IF;
 COMMIT;
END$$

DELIMITER ;



DELIMITER $$
drop PROCEDURE if exists init;
CREATE PROCEDURE init()
BEGIN
 start TRANSACTION;
 update Tickets set remaining_tickets = 10;
 select "init finished.";
 commit;
END$$

DELIMITER ;
```

并发脚本，python 示例：

依赖库：

```
pip install pymysql
pip install mysql-connector-python
```

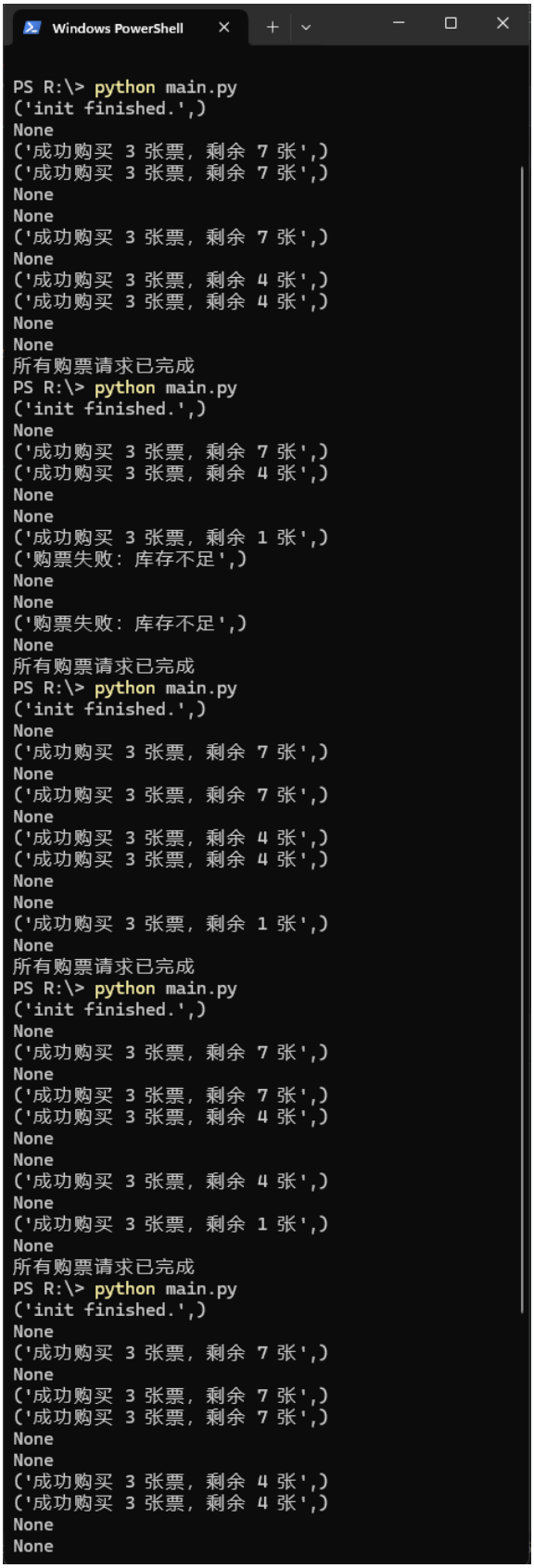
python 脚本：

```
import threading
import mysql.connector
```

```
from mysql.connector import Error


def buy_tickets(movie_id, tickets_to_buy):
    try:
        connection = mysql.connector.connect(host='localhost',
                                             database='TicketSystem',
                                             user='your_username',
                                             password='your_password',
                                             port='3306')

        if connection.is_connected():
            cursor = connection.cursor()
            cursor.callproc('BuyTicketsNoLock', [movie_id, tickets_to_buy])
            for result in cursor.stored_results():
                print(result.fetchall())
    except Error as e:
        print("Error while connecting to MySQL", e)
    finally:
        if (connection.is_connected()):
            cursor.close()
            connection.close()


def init():
    try:
        connection = mysql.connector.connect(host='localhost',
                                             database='TicketSystem',
                                             user='your_username',
                                             password='your_password',
                                             port='3306')

        if connection.is_connected():
            cursor = connection.cursor()
            cursor.callproc('init', [])
            for result in cursor.stored_results():
                print(result.fetchall())
    except Error as e:
        print("Error while connecting to MySQL", e)
    finally:
        if (connection.is_connected()):
            cursor.close()
```

```
            connection.close()


init()
# 创建线程列表
threads = []

# 添加线程到线程列表中
for i in range(5):  # 假设我们想要模拟 5 个并发请求
    thread = threading.Thread(target=buy_tickets, args=(1, 3))
    threads.append(thread)

# 开始所有线程
for thread in threads:
    thread.start()

# 等待所有线程完成
for thread in threads:
    thread.join()

print(" 所有购票请求已完成")
```

步骤 3：运行脚本并观察结果多次运行并发脚本（参考上面的 python 脚本或自己编写，不限语言）并观察结果，解释结果出现的原因（最好附带结果截图）。



由于没有锁，多次执行出现了冲突。

在这个示例中，先进行了一次查询剩余票数，再进行了一次更新剩余票数。一次查询可以看作一次 read，一次更新可以看作一次 read 加一次 write。

当一个线程 read 了旧的结果，还没 write 的时候，另一个线程也 read 了旧的结果，两个线程都认为还有剩余的票，于是都认为购买成功了。

步骤 4：添加锁修改存储过程代码，分别添加共享锁和排他锁并多次运行并发脚本，观察结果并说明原因（最好附带结果截图）。

添加共享锁：

```
-- 使用共享锁，查询剩余票数
SELECT remaining_tickets INTO remaining FROM Tickets WHERE id = movie_id LOCK
↪  IN SHARE MODE ;
```

```
cursor.callproc('BuyTicketsLock', [movie_id, tickets_to_buy])
```

```
PS R:\> python main.py
('init finished.',)
None
('成功购买 3 张票，剩余 7 张',)
('成功购买 3 张票，剩余 4 张',)
None
('购票失败：库存不足',)
('成功购买 3 张票，剩余 1 张',)
None
None
None
('购票失败：库存不足',)
None
所有购票请求已完成
PS R:\> python main.py
('init finished.',)
None
('成功购买 3 张票，剩余 7 张',)
None
Exception in thread Thread-5 (buy_tickets):
('成功购买 3 张票，剩余 4 张',)
Traceback (most recent call last):
pymysql.err.OperationalError: (1213, 'Deadlock f
ound when trying to get lock; try restarting tra
nsaction')
('成功购买 3 张票，剩余 1 张',)
None
('购票失败：库存不足',)
None
所有购票请求已完成
```

能看到可能会出现死锁。

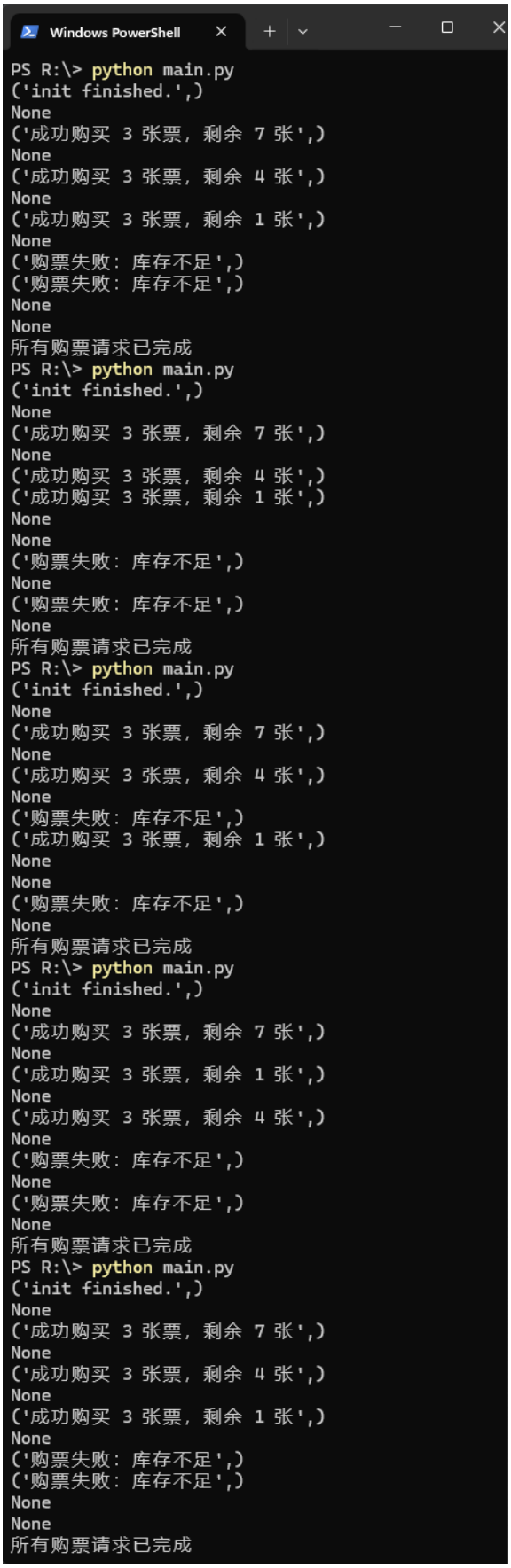
在这个示例中，先进行了一次查询剩余票数，再进行了一次更新剩余票数。一次查询可以看作一次 read，一次更新可以看作一次 read 加一次 write。

由于使用的是共享锁，每个线程在第一次 read 时都能拿到锁，第二次 read 也没问题，而在 write 时，由于 MySQL 不允许脏写，所以每个线程都在等待除了自己以外的其他线程把这个锁释放，于是就产生了死锁。

添加排他锁：

```
-- 使用排他锁，查询剩余票数
SELECT remaining_tickets INTO remaining FROM Tickets WHERE id = movie_id FOR
↪  UPDATE ;
```

```
cursor.callproc('BuyTicketsLock', [movie_id, tickets_to_buy])
```



能看到没有死锁了。

因为排他锁只有一个线程能拿到，当这个线程把锁释放后其他线程才能拿到锁，所以就不会死锁了。

9. (简答题) 教材 13.15 请考虑以下两个事务:

T_{13}: read(A);

read(B);

if A = 0 then B:= B+ 1;

write(B).

T_{14}: read(B);

read(A);

if B = 0 then A:= A+ 1;

write(A).

令一致性需求为 $A = 0 \vee B = 0$，初值是 A=B=0。

a. 请说明包括这两个事务的每一个串行执行都保持了数据库的一致性。

由于只有两个事务，所以串行执行只有两种情况：先 T_{13} 再 T_{14}，或者先 T_{14} 再 T_{13}。

如果先 T_{13} 再 T_{14}，那么执行完 T_{13} 后，$A = 0, B = 1$，再执行 T_{14}，还是 $A = 0, B = 1$，始终保持一致性。

如果先 T_{14} 再 T_{13}，那么执行完 T_{14} 后，$A = 1, B = 0$，再执行 T_{13}，还是 $A = 1, B = 0$，始终保持一致性。

b. 请给出 T_{13} 和 T_{14} 的一次并发执行，它产生了不可串行化的调度。

```
T13:                              T14:
read(A);    // A 读取为 0
read(B);    // B 读取为 0
if A = 0 then B := B + 1;
                                  read(B);    // B 读取为 0
                                  read(A);    // A 读取为 0
                                  if B = 0 then A := A + 1;
write(B);   // B 写入为 1
                                  write(A);   // A 写入为 1
```

最终的结果是 $A = 1, B = 1$，和任何一个串行执行的结果都不同，所以不可串行化。

c. 存在产生可串行化调度的 T_{13} 和 T_{14} 的并发执行吗?

不存在。

10. (简答题) 教材 14.14

请解释为什么 undo-list 中事务的日志记录必须由后往前处理，而执行重做时日志记录则由前往后处理。

因为 undo 是按照时间倒序把新值恢复为旧值，比如值的改变是 $A \to B \to C$，恢复的时候就要 $C \to B \to A$，顺序乱了就无法正确恢复。

而 redo 的时候是按照时间顺序把旧值设置为新值，日志记录是按照时间顺序从前往后的，那么 redo 也应按照日志记录从前往后。