北京交通大学

BEIJING JIAOTONG UNIVERSITY

北京交通大学

课程名称：数据库系统原理

实验题目 ： 数据库系统原理Lab7

学号 ： 22281188

姓名 ： 江家玮

班级 ： 计科2204班

指导老师 ： 刘真老师

报告日期 ： 2025-05-21

目录结构：

```
22281188-江家炜-数据库LAB7/
├── database_scripts/
|   ├── DBLab3_22281188.sql          # 数据库模式定义
|   ├── DBLab3_Data_Insert.sql       # 初始数据插入
|   └── lab7_procedures_triggers.sql # 存储过程和触发器
├── templates/
|   └── passengers.html              # 前端页面
├── app.py                           # Python Flask应用
├── concurrency_tests/
|   ├── isolation_level_setup.sql    # 设置隔离级别和测试账户表
|   ├── dirty_read_test.py           # 脏读测试脚本
|   ├── non_repeatable_read_test.py # 不可重复读测试脚本
|   ├── lost_update_test.py          # 丢失更新测试脚本
|   └── README_Lab7.md               # 并发测试说明
└── README.md                        # 项目总说明
```

一、实验目的与要求

具体目标和要求如下：

1. **存储过程与触发器实验：**
    - 针对我的“航空公司乘客管理系统”应用场景，在MySQL数据库平台上实现以下操作的存储过程，每种至少一个：
        - 单表或多表查询
        - 数据插入
        - 数据删除
        - 数据修改
    - 通过Python Flask后端应用（B/S模式），调用所实现的后台存储过程，以验证其功能。
    - 在我的案例场景中，分别设计并实现由数据插入、数据更新、数据删除操作引发的触发器（前触发或后触发均可），并测试触发器的执行效果。
2. **并发模拟实验：**
    - 针对我的应用场景，搭建并发实验环境，模拟多用户并发访问数据库。
    - 设置MySQL的四种事务隔离级别（Read Uncommitted, Read Committed, Repeatable Read, Serializable）。
    - 通过编写Python脚本或使用多MySQL客户端，验证并发操作可能带来的数据不一致问题，包括：
        - 丢失修改 (Lost Update)
        - 不可重复读 (Non-Repeatable Read)
        - 读“脏”数据 (Dirty Read)

二、实验环境

- **操作系统：** macOS
- **数据库管理系统：** MySQL 8.0
- **后端编程语言与框架：** Python 3.12, Flask

- **数据库连接库：** `mysql-connector-python`
- **API测试工具：** Postman
- **并发测试环境：**
  - 多个MySQL命令行客户端实例
  - Python `threading` 模块编写的并发模拟脚本
- **文本编辑器/IDE：** Visual Studio Code
- **项目文件结构：**

```
AirlineApp_Lab7/
├── database_scripts/
|   ├── DBLab3_22281188.sql          # 数据库模式定义
|   ├── DBLab3_Data_Insert.sql       # 初始数据插入
|   └── lab7_procedures_triggers.sql # 存储过程和触发器
├── templates/
|   └── passengers.html              # 前端页面
├── app.py                           # Python Flask应用
├── concurrency_tests/
|   ├── isolation_level_setup.sql    # 设置隔离级别和测试账户表
|   ├── dirty_read_test.py           # 脏读测试脚本
|   ├── non_repeatable_read_test.py  # 不可重复读测试脚本
|   ├── lost_update_test.py          # 丢失更新测试脚本
|   └── README_Lab7.md               # 并发测试说明
└── README.md                        # 项目总说明
```

三、存储过程与触发器实验

在我的“航空公司乘客管理系统”中，数据库为`AirlineDB`。我针对乘客（Passenger）、预订（Booking）和航班（Flight）等核心实体设计并实现了相关的存储过程和触发器。

3.1 存储过程的实现

我实现了以下四个存储过程，覆盖了查询、插入、删除和修改操作。SQL脚本位于`database_scripts/lab7_procedures_triggers.sql`。

3.1.1 单表或多表查询存储过程（`GetPassengerAndBookings`）

- **功能说明：** 根据输入的乘客ID，查询该乘客的基本信息及其所有相关的预订记录（包括航班的起降时间、航线起终点等）。这是一个多表连接查询。
- **SQL定义：**

```
DELIMITER //
DROP PROCEDURE IF EXISTS GetPassengerAndBookings;
CREATE PROCEDURE GetPassengerAndBookings(
    IN p_passenger_id INT
)
BEGIN
    SELECT
        p.passenger_id, p.name AS passenger_name,
p.id_card_number, p.phone_number, p.email,
        b.booking_id, b.flight_id, f.departure_time,
f.arrival_time,
        r.origin, r.destination, b.seat_type, b.booking_date,
b.price AS booking_price, b.payment_status
    FROM Passenger p
    LEFT JOIN Booking b ON p.passenger_id = b.passenger_id
    LEFT JOIN Flight f ON b.flight_id = f.flight_id
    LEFT JOIN Route r ON f.route_id = r.route_id
    WHERE p.passenger_id = p_passenger_id;
END //
DELIMITER ;
```

3.1.2 数据插入存储过程（`AddPassengerViaSP`）

- **功能说明：** 接收乘客的各项信息作为输入参数，向`Passenger`表中插入一条新的乘客记录。存储过程返回新插入乘客的`passenger_id`。此过程依赖于`Passenger`表的`passenger_id`列已设置为`AUTO_INCREMENT`。
- **SQL定义：**

```
DELIMITER //
DROP PROCEDURE IF EXISTS AddPassengerViaSP;
CREATE PROCEDURE AddPassengerViaSP(
    IN p_name VARCHAR(100), IN p_id_card_number VARCHAR(18), IN
p_phone_number VARCHAR(20),
    IN p_email VARCHAR(100), IN p_frequent_flyer_number
VARCHAR(50), OUT p_new_passenger_id INT
)
BEGIN
    INSERT INTO Passenger (name, id_card_number, phone_number,
email, frequent_flyer_number)
    VALUES (p_name, p_id_card_number, p_phone_number, p_email,
p_frequent_flyer_number);
    SET p_new_passenger_id = LAST_INSERT_ID();
END //
DELIMITER ;
```

3.1.3 数据删除存储过程（`DeletePassengerViaSP`）

- **功能说明：** 根据输入的乘客ID，从`Passenger`表中删除对应的乘客记录。
- **SQL定义：**

```
DELIMITER //
DROP PROCEDURE IF EXISTS DeletePassengerViaSP;
CREATE PROCEDURE DeletePassengerViaSP(
    IN p_passenger_id INT
)
BEGIN
    DELETE FROM Passenger WHERE passenger_id = p_passenger_id;
END //
DELIMITER ;
```

3.1.4 数据修改存储过程（`UpdatePassengerEmailViaSP`）

- **功能说明：** 根据输入的乘客ID和新的电子邮件地址，更新该乘客的电子邮件信息。

- **SQL定义：**

```
DELIMITER //
DROP PROCEDURE IF EXISTS UpdatePassengerEmailViaSP;
CREATE PROCEDURE UpdatePassengerEmailViaSP(
    IN p_passenger_id INT,
    IN p_new_email VARCHAR(100)
)
BEGIN
    UPDATE Passenger
    SET email = p_new_email
    WHERE passenger_id = p_passenger_id;
END //
DELIMITER ;
```

3.2 在后端程序中调用存储过程

我对原有的`app.py`（Flask应用）进行了修改，添加了新的API端点（路径以`/api/sp/...`开头）来调用上述存储过程。

调用示例（以添加乘客`AddPassengerViaSP`为例）：

```
@app.route('/api/sp/passengers', methods=['POST'])
def add_passenger_sp():
... (参数校验) ...
    conn = get_db_connection()
    # ... (错误处理) ...
    cursor = conn.cursor()
    try:
        args = (
            data['name'], data['id_card_number'],
data['phone_number'],
            data.get('email') or None,
data.get('frequent_flyer_number') or None,
            0  # Placeholder for OUT parameter p_new_passenger_id
        )
        result_args = cursor.callproc('AddPassengerViaSP', args) # 调
用存储过程
        conn.commit() # 提交事务
```

```
        new_passenger_id = result_args[5] # 获取OUT参数的值 (索引从0开
始，第6个参数)
        return jsonify({"message": "乘客通过存储过程添加成功",
"passenger_id": new_passenger_id}), 201
    # ... (异常处理和关闭连接) ...
```

我使用Postman工具测试了这些新的API端点，例如：

```
GET /api/sp/passengers/1001/details

POST /api/sp/passengers (请求体包含新乘客数据)

PUT /api/sp/passengers/1001/email (请求体包含新邮箱)

DELETE /api/sp/passengers/<id>
```

所有测试均返回了预期的结果，表明存储过程已能被后端应用正确封装和调用。

3.3 触发器的实现与测试

我设计并实现了三个AFTER触发器。

3.3.1 插入操作触发器（`AfterBookingInsert`）

功能说明： 当向Booking表成功插入一条新的预订记录后，此触发器自动触发，将对应航班（Flight表）的booked_seats（已预订座位数）加1。

SQL定义：

```
DELIMITER //
CREATE TRIGGER AfterBookingInsert
AFTER INSERT ON Booking
FOR EACH ROW
BEGIN
    UPDATE Flight SET booked_seats = booked_seats + 1 WHERE flight_id
= NEW.flight_id;
END //
DELIMITER ;
```

测试方法： 在MySQL客户端直接向Booking表插入一条新记录，然后查询对应Flight记录的booked_seats字段，验证其值是否增加了1。测试结果符合预期。

3.3.2 更新操作触发器（`AfterPassengerEmailUpdate`）

功能说明： 当Passenger表中某条记录的email字段被更新后（且新旧值不同），此触发器自动触发，将旧邮箱、新邮箱及变更时间等信息记录到PassengerEmailAudit审计表中。

审计表PassengerEmailAudit定义：

```
CREATE TABLE IF NOT EXISTS PassengerEmailAudit (
    audit_id INT AUTO_INCREMENT PRIMARY KEY, passenger_id INT,
    old_email VARCHAR(100), new_email VARCHAR(100),
    change_timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    changed_by VARCHAR(255) DEFAULT 'DB_TRIGGER'
);
```

触发器SQL定义：

```
DELIMITER //
CREATE TRIGGER AfterPassengerEmailUpdate
AFTER UPDATE ON Passenger
FOR EACH ROW
BEGIN
    IF OLD.email <> NEW.email OR (OLD.email IS NULL AND NEW.email IS
NOT NULL) OR (OLD.email IS NOT NULL AND NEW.email IS NULL) THEN
        INSERT INTO PassengerEmailAudit (passenger_id, old_email,
new_email)
        VALUES (OLD.passenger_id, OLD.email, NEW.email);
    END IF;
END //
DELIMITER ;
```

测试方法： 在MySQL客户端直接更新Passenger表中某条记录的email字段，然后查询PassengerEmailAudit表，验证是否生成了相应的审计记录。测试结果符合预期。

3.3.3 删除操作触发器（`AfterBookingDelete`）

功能说明： 当从Booking表成功删除一条预订记录后，此触发器自动触发，将对应航班（Flight表）的booked_seats减1。

SQL定义：

```
DELIMITER //
CREATE TRIGGER AfterBookingDelete
AFTER DELETE ON Booking
FOR EACH ROW
BEGIN
    UPDATE Flight SET booked_seats = booked_seats - 1 WHERE flight_id
= OLD.flight_id;
END //
DELIMITER ;
```

测试方法： 在MySQL客户端先插入一条Booking记录（会触发AfterBookingInsert），然后删除该记录，再查询对应Flight记录的booked_seats字段，验证其值是否恢复到删除前的值减1。测试结果符合预期。

四、并发模拟实验

4.1 并发实验环境搭建

我主要通过以下方式搭建并发实验环境：

多MySQL客户端： 打开多个MySQL命令行客户端窗口，每个窗口代表一个独立的用户会话。

Python并发脚本： 使用位于 `concurrency_tests/` 目录下的Python脚本（`dirty_read_test.py`，`non_repeatable_read_test.py`，`lost_update_test.py`），这些脚本利用threading模块创建多个线程来模拟并发用户操作。

在进行具体测试前，我通过执行`concurrency_tests/isolation_level_setup.sql`脚本，在AirlineDB数据库中创建了用于部分并发场景测试的 Accounts 表并插入了初始数据。对于涉及航班座位的测试，则直接使用Flight表中的booked_seats字段。

4.2 MySQL事务隔离级别简介

MySQL支持四种标准的事务隔离级别，用于控制事务并发执行时数据的可见性和一致性：

> READ UNCOMMITTED (读未提交)
>
> READ COMMITTED (读已提交)
>
> REPEATABLE READ (可重复读) (MySQL InnoDB存储引擎的默认隔离级别)
>
> SERIALIZABLE (可串行化)

在每个测试会话或Python脚本的数据库连接中，我使用 SET SESSION TRANSACTION ISOLATION LEVEL <LEVEL_NAME>; 来设置所需的隔离级别进行测试。

4.3 并发操作带来的数据不一致问题验证

4.3.1 读“脏”数据（Dirty Read）

概念：事务T1读取了事务T2已修改但尚未提交的数据。

测试方法：运行 dirty_read_test.py。该脚本模拟事务A更新数据但不提交，事务B在不同隔离级别下读取该数据，然后事务A回滚。

预期现象：

在 READ UNCOMMITTED 级别下，事务B应能读取到事务A未提交的修改（脏数据）。

在 READ COMMITTED 级别下，事务B应读取到事务A修改前已提交的数据，避免脏读。

4.3.2 不可重复读（Non-Repeatable Read）

概念：事务T1在事务内两次读取同一行数据，结果不一致，因为事务T2在此期间修改了该行并提交。

测试方法：运行 non_repeatable_read_test.py。脚本模拟事务A两次读取数据，期间事务B修改并提交该数据。

预期现象：

在 READ COMMITTED 级别下，事务A的第二次读取应看到事务B提交的修改，发生不可重复读。

在 REPEATABLE READ 级别下，事务A两次读取的结果应一致，避免不可重复读。

4.3.3 丢失修改（Lost Update）

概念：两个事务基于同一旧值进行修改并写回，导致一个事务的更新被另一个事务覆盖。

测试方法：运行 lost_update_test.py。脚本模拟两个线程并发执行“读-计算-写回”操作更新航班的 booked_seats，并与原子操作进行对比。

预期现象：

在使用非原子的“读-改-写”逻辑，并在 READ COMMITTED 或特定情况下的 REPEATABLE READ 级别时，可能发生丢失修改，即 booked_seats 的最终增量小于预期。

使用数据库原子操作 UPDATE Flight SET booked_seats = booked_seats + 1 应能避免丢失修改。

4.3.4 幻读（Phantom Read）

概念：事务T1执行范围查询，事务T2在该范围插入新行并提交，T1再次查询发现“幻影”行。

手动测试思路：在两个MySQL客户端模拟。会话1（REPEATABLE READ）执行范围查询，会话2插入符合条件的新行并提交，会话1再次查询。

分析：MySQL InnoDB的 REPEATABLE READ 通过MVCC和Next-Key Locking在很大程度上能避免幻读。完全避免需 SERIALIZABLE 级别。

五、测试结果

5.1 数据库初始化

创建存储过程和触发器

- 执行 `database_scripts/lab7_procedures_triggers.sql` 文件的全部内容。这将创建实验七所需的存储过程和触发器。

```
mysql> source /Users/bananapig/Desktop/22281188-江家玮-数据库Lab5/database_scripts/lab7_procedures_triggers.sql
Query OK, 0 rows affected (0.001 sec)

Query OK, 0 rows affected (0.000 sec)

Query OK, 0 rows affected (0.000 sec)

Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
Query OK, 0 rows affected (0.000 sec)

Query OK, 0 rows affected (0.000 sec)

Query OK, 0 rows affected (0.000 sec)

Query OK, 0 rows affected (0.000 sec)

Query OK, 0 rows affected (0.000 sec)

Query OK, 0 rows affected, 1 warning (0.008 sec)

Query OK, 0 rows affected (0.017 sec)
```

- **检查：** 执行完每个脚本后，检查是否有错误信息。执行一些简单的`SELECT`语句来确认表已创建并且数据已插入，例如：

```sql
SELECT * FROM Passenger LIMIT 5;
SELECT * FROM Flight LIMIT 5;
SHOW PROCEDURE STATUS WHERE Db = 'AirlineDB'; -- 查看已创建的存储过程
SHOW TRIGGERS FROM AirlineDB; -- 查看已创建的触发器
```

```
mysql> SELECT * FROM Passenger LIMIT 5;
+--------------+--------------+--------------------+--------------+-------------------------+----------------------+
| passenger_id | name         | id_card_number     | phone_number | email                   | frequent_flyer_number |
+--------------+--------------+--------------------+--------------+-------------------------+----------------------+
|         1001 | Jiang Jiawei | 310101199001011234 | 13800138000  | byjiaweijiang@gmail.com | MU1234567            |
|         1002 | Xiao Jiang   | 440101199202022345 | 13900139000  | 22281188@bjtu.edu.cn    | CZ654321             |
|         1003 | Coconut      | 441802200401300000 | 15119968067  | vgmwg303393@outlook.com | CN00000              |
|         1004 | User1        | 441802200401300001 | 13922609999  | 376234982@qq.com        | CN0000001            |
+--------------+--------------+--------------------+--------------+-------------------------+----------------------+
4 rows in set (0.004 sec)
```

```
mysql> SELECT * FROM Flight LIMIT 5;
+-----------+------------+----------+---------------------+---------------------+----------------------------------------+----------------------------------------+-------------+--------------+----------------+
| flight_id | airline_id | route_id | departure_time      | arrival_time        | departure_location                     | destination_location                   | total_seats | booked_seats | aircraft_model |
+-----------+------------+----------+---------------------+---------------------+----------------------------------------+----------------------------------------+-------------+--------------+----------------+
|      3001 |          1 |      201 | 2025-07-15 08:00:00 | 2025-07-15 10:10:00 | Shanghai Hongqiao International Airport | Beijing Capital International Airport  |         180 |            0 | A320           |
|      3002 |          2 |      202 | 2025-07-15 09:30:00 | 2025-07-15 11:30:00 | Guangzhou Baiyun International Airport | Shanghai Hongqiao International Airport |         220 |            0 | B737           |
+-----------+------------+----------+---------------------+---------------------+----------------------------------------+----------------------------------------+-------------+--------------+----------------+
2 rows in set (0.001 sec)
```

```
mysql> SHOW TRIGGERS FROM AirlineDB;
| Trigger                  | Event  | Table     | Statement
                                   | Timing | Created                | sql_mode                                                  | Definer        | character_set_client | collation_connection | Database Collation |
| AfterBookingInsert       | INSERT | Booking   | BEGIN
    UPDATE Flight
    SET booked_seats = booked_seats + 1
    WHERE flight_id = NEW.flight_id;
END                                                                                | AFTER  | 2025-05-21 15:19:31.92 | ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,ERROR_F
OR_DIVISION_BY_ZERO,NO_ENGINE_SUBSTITUTION | root@localhost | utf8mb4              | utf8mb4_0900_ai_ci   | utf8mb4_unicode_ci |
| AfterBookingDelete       | DELETE | Booking   | BEGIN
    UPDATE Flight
    SET booked_seats = booked_seats - 1
    WHERE flight_id = OLD.flight_id;
END                                                                                | AFTER  | 2025-05-21 15:19:31.93 | ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,ERROR_F
OR_DIVISION_BY_ZERO,NO_ENGINE_SUBSTITUTION | root@localhost | utf8mb4              | utf8mb4_0900_ai_ci   | utf8mb4_unicode_ci |
| AfterPassengerEmailUpdate | UPDATE | Passenger | BEGIN
    IF OLD.email <> NEW.email OR (OLD.email IS NULL AND NEW.email IS NOT NULL) OR (OLD.email IS NOT NULL AND NEW.email IS NULL) THEN
        INSERT INTO PassengerEmailAudit (passenger_id, old_email, new_email)
        VALUES (OLD.passenger_id, OLD.email, NEW.email);
    END IF;
END | AFTER  | 2025-05-21 15:19:31.93 | ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,ERROR_FOR_DIVISION_BY_ZERO,NO_ENGINE_SUBSTITUTION | root@localhost | utf8mb4              | utf8mb4_0900_ai_ci   | utf8mb4_unicode_ci |
3 rows in set (0.008 sec)
```

5.2 测试存储过程和触发器

A. 通过MySQL客户端手动测试

可以直接在MySQL客户端中调用存储过程并观察触发器的行为，如`lab7_procedures_triggers.sql`文件末尾的注释示例所示。

测试存储过程 `GetPassengerAndBookings`：

```
USE AirlineDB;
CALL GetPassengerAndBookings(1001);
```

观察返回的乘客及其预订信息。

1.

```
mysql> USE AirlineDB;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> CALL GetPassengerAndBookings(1001);
+--------------+----------------+-----------------+--------------+-------------------------+------------+-----------+---------------------+---------------------+-------------------------------+-------------------------------+----------
--------+---------------------+---------------+----------------+
| passenger_id | passenger_name | id_card_number  | phone_number | email                   | booking_id | flight_id | departure_time      | arrival_time        | origin                        | destination                   | sea
t_type | booking_date        | booking_price | payment_status |
+--------------+----------------+-----------------+--------------+-------------------------+------------+-----------+---------------------+---------------------+-------------------------------+-------------------------------+----------
--------+---------------------+---------------+----------------+
|         1001 | Jiang Jiawei   | 310101199001011234 | 13800138000  | byjiaweijiang@gmail.com |       4001 |      3001 | 2025-07-15 08:00:00 | 2025-07-15 10:10:00 | Shanghai Hongqiao International Airport | Beijing Capital International Airport | Eco
nomy   | 2025-06-10 14:30:00 |        750.00 | Paid           |
+--------------+----------------+-----------------+--------------+-------------------------+------------+-----------+---------------------+---------------------+-------------------------------+-------------------------------+----------
--------+---------------------+---------------+----------------+
1 row in set (0.006 sec)

Query OK, 0 rows affected (0.006 sec)
```

测试触发器 `AfterBookingInsert`：

```
-- 1. 查看航班3001的初始booked_seats
SELECT flight_id, booked_seats FROM Flight WHERE flight_id = 3001;
-- 2. 插入一条新的预订（确保booking_id唯一，例如5001）
INSERT INTO Booking (booking_id, passenger_id, flight_id, seat_type,
booking_date, price, payment_status)
VALUES (5001, 1002, 3001, 'Economy', NOW(), 750.00, 'Paid');
-- 3. 再次查看booked_seats，应已增加1
SELECT flight_id, booked_seats FROM Flight WHERE flight_id = 3001;
```

```
mysql> SELECT flight_id, booked_seats FROM Flight WHERE flight_id = 3001;
+-----------+--------------+
| flight_id | booked_seats |
+-----------+--------------+
|      3001 |            0 |
+-----------+--------------+
1 row in set (0.001 sec)

mysql> INSERT INTO Booking (booking_id, passenger_id, flight_id, seat_type, booking_date, price, payment_status)
    -> VALUES (5001, 1002, 3001, 'Economy', NOW(), 750.00, 'Paid');
Query OK, 1 row affected (0.010 sec)

mysql> SELECT flight_id, booked_seats FROM Flight WHERE flight_id = 3001;
+-----------+--------------+
| flight_id | booked_seats |
+-----------+--------------+
|      3001 |            1 |
+-----------+--------------+
1 row in set (0.001 sec)
```

测试存储过程 `UpdatePassengerEmailViaSP` 和触发器 `AfterPassengerEmailUpdate`：

```
-- 1. 查看乘客1001的当前邮箱和PassengerEmailAudit表关于此乘客的记录
SELECT email FROM Passenger WHERE passenger_id = 1001;
SELECT * FROM PassengerEmailAudit WHERE passenger_id = 1001 ORDER BY
change_timestamp DESC;
-- 2. 调用存储过程更新邮箱
CALL UpdatePassengerEmailViaSP(1001,
'new.sp.email.updated@example.com');
-- 3. 再次查看邮箱和审计表，应有新记录
SELECT email FROM Passenger WHERE passenger_id = 1001;
SELECT * FROM PassengerEmailAudit WHERE passenger_id = 1001 ORDER BY
change_timestamp DESC;
```

```
mysql> SELECT * FROM PassengerEmailAudit WHERE passenger_id = 1001 ORDER BY change_timestamp DESC;
+----------+--------------+-------------------------+---------------------------------+---------------------+------------+
| audit_id | passenger_id | old_email               | new_email                       | change_timestamp    | changed_by |
+----------+--------------+-------------------------+---------------------------------+---------------------+------------+
|        1 |         1001 | byjiaweijiang@gmail.com | new.sp.email.updated@example.com | 2025-05-21 15:26:47 | DB_TRIGGER |
+----------+--------------+-------------------------+---------------------------------+---------------------+------------+
1 row in set (0.002 sec)
```

测试触发器 `AfterBookingDelete`：

```
-- (接步骤3，此时航班3001的booked_seats应为初始值+1)
-- 1. 删除之前插入的预订记录 (booking_id = 5001)
DELETE FROM Booking WHERE booking_id = 5001;
-- 2. 再次查看booked_seats，应已减少1，恢复到初始值
SELECT flight_id, booked_seats FROM Flight WHERE flight_id = 3001;
```

```
mysql> DELETE FROM Booking WHERE booking_id = 5001;
Query OK, 1 row affected (0.002 sec)

mysql> SELECT flight_id, booked_seats FROM Flight WHERE flight_id = 3001;
+-----------+--------------+
| flight_id | booked_seats |
+-----------+--------------+
|      3001 |            0 |
+-----------+--------------+
1 row in set (0.001 sec)
```

B. 通过API端点测试（主要测试方式）

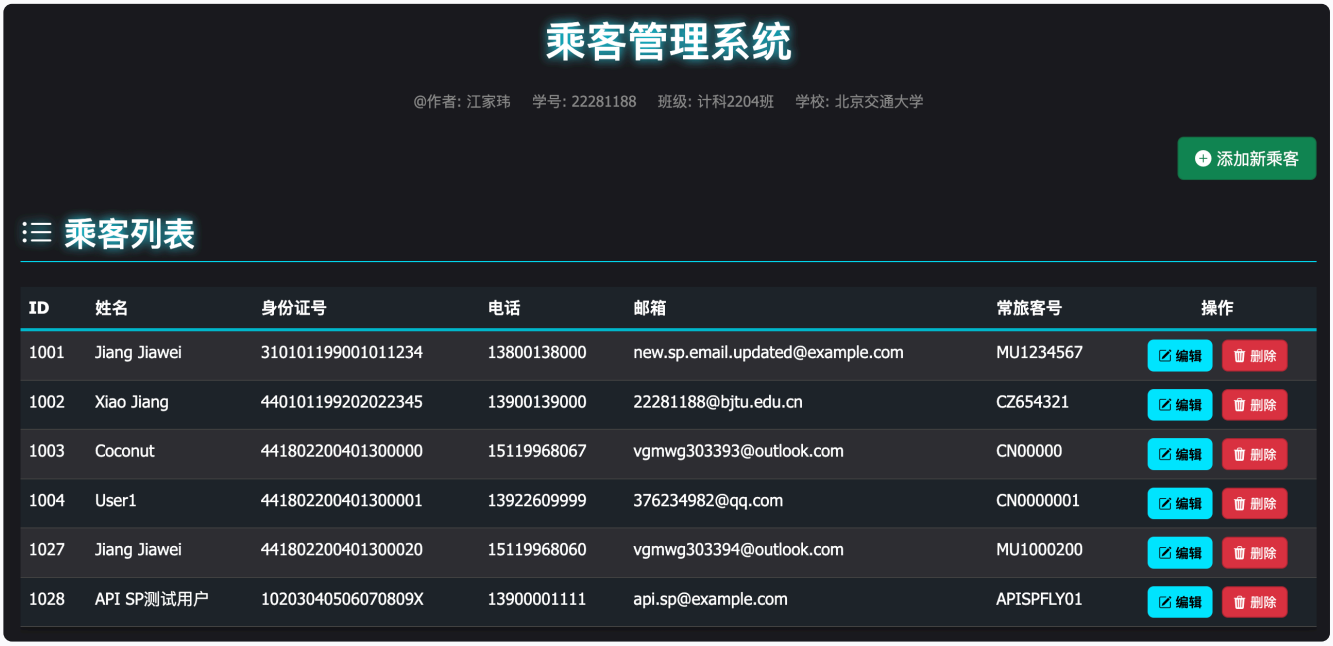
使用Postman或curl测试`app.py`中定义的与存储过程相关的API端点。确保Flask应用正在运行。

- **查询乘客详情（调用 `GetPassengerAndBookings`）**
  - **Method:** `GET`
  - **URL:** `http://127.0.0.1:5000/api/sp/passengers/1001/details`（将1001替换为存在的乘客ID）
  - **预期：** 返回包含乘客信息及其预订列表的JSON。
- **添加新乘客（调用 `AddPassengerViaSP`）**
  - **Method:** `POST`
  - **URL:** `http://127.0.0.1:5000/api/sp/passengers`
  - **Headers:** `Content-Type: application/json`
  - Body (raw JSON):

```
{
    "name": "API SP测试用户",
    "id_card_number": "10203040506070809X",
    "phone_number": "13900001111",
    "email": "api.sp@example.com",
    "frequent_flyer_number": "APISPFLY01"
}
```

  - **预期：** 返回成功消息和新乘客的`passenger_id`（HTTP 201）。同时，可以去数据库检查`Passenger`表是否添加了此记录。

成功添加，但会HTTP 201，数据库表格中成功添加了记录



```
127.0.0.1 - - [21/May/2025 15:41:57] "POST /api/passengers HTTP/1.1" 201 -
127.0.0.1 - - [21/May/2025 15:41:57] "GET /api/passengers HTTP/1.1" 200 -
127.0.0.1 - - [21/May/2025 15:42:41] "POST /api/passengers HTTP/1.1" 201 -
127.0.0.1 - - [21/May/2025 15:42:41] "GET /api/passengers HTTP/1.1" 200 -
```

```
mysql> SELECT * FROM Passenger
    -> ;
+--------------+-------------------+--------------------+--------------+---------------------------------+----------------------+
| passenger_id | name              | id_card_number     | phone_number | email                           | frequent_flyer_number |
+--------------+-------------------+--------------------+--------------+---------------------------------+----------------------+
|         1001 | Jiang Jiawei      | 310101199001011234 | 13800138000  | new.sp.email.updated@example.com | MU1234567            |
|         1002 | Xiao Jiang        | 440101199202022345 | 13900139000  | 22281188@bjtu.edu.cn            | CZ654321             |
|         1003 | Coconut           | 441802200401300000 | 15119968067  | vgmwg303393@outlook.com         | CN00000              |
|         1004 | User1             | 441802200401300001 | 13922609999  | 376234982@qq.com                | CN0000001            |
|         1027 | Jiang Jiawei      | 441802200401300020 | 15119968060  | vgmwg303394@outlook.com         | MU1000200            |
|         1028 | API SP测试用户    | 10203040506070809X | 13900001111  | api.sp@example.com              | APISPFLY01           |
+--------------+-------------------+--------------------+--------------+---------------------------------+----------------------+
6 rows in set (0.000 sec)
```

- **更新乘客邮箱（调用 `UpdatePassengerEmailViaSP`）**
  - **Method:** `PUT`
  - **URL:** `http://127.0.0.1:5000/api/sp/passengers/1001/email` (将1001替换为存在的乘客ID)
  - **Headers:** `Content-Type: application/json`
  - Body (raw JSON):

```
{
    "email": "updated.api.sp@example.com"
}
```

  - **预期：** 返回成功消息。去数据库检查`Passenger`表对应乘客的邮箱是否更新，并检查`PassengerEmailAudit`表是否有新的审计记录。
    成功，并且表格更新：

ID	姓名	身份证号	电话	邮箱	常旅客号	操作
1001	Jiang Jiawei	310101199001011234	13800138000	new.sp.email.updated@example.com	MU1234567	编辑 删除
1002	Xiao Jiang	440101199202022345	13900139000	22281188@bjtu.edu.cn	CZ654321	编辑 删除
1003	Coconut	441802200401300000	15119968067	vgmwg303393@outlook.com	CN00000	编辑 删除
1004	User1	441802200401300001	13922609999	376234982@qq.com	CN0000001	编辑 删除
1027	Jiang Jiawei	441802200401300020	15119968060	vgmwg303394@outlook.com	MU1000200	编辑 删除
1028	API SP测试用户	10203040506070809X	13900001111	updated.api.sp@example.com	APISPFLY01	编辑 删除

```
mysql> SELECT * FROM Passenger;
+--------------+-------------------+--------------------+--------------+---------------------------------+----------------------+
| passenger_id | name              | id_card_number     | phone_number | email                           | frequent_flyer_number |
+--------------+-------------------+--------------------+--------------+---------------------------------+----------------------+
|         1001 | Jiang Jiawei      | 310101199001011234 | 13800138000  | new.sp.email.updated@example.com | MU1234567            |
|         1002 | Xiao Jiang        | 440101199202022345 | 13900139000  | 22281188@bjtu.edu.cn            | CZ654321             |
|         1003 | Coconut           | 441802200401300000 | 15119968067  | vgmwg303393@outlook.com         | CN00000              |
|         1004 | User1             | 441802200401300001 | 13922609999  | 376234982@qq.com                | CN0000001            |
|         1027 | Jiang Jiawei      | 441802200401300020 | 15119968060  | vgmwg303394@outlook.com         | MU1000200            |
|         1028 | API SP测试用户    | 10203040506070809X | 13900001111  | updated.api.sp@example.com      | APISPFLY01           |
+--------------+-------------------+--------------------+--------------+---------------------------------+----------------------+
6 rows in set (0.001 sec)
```

审计表也更新了：

```
mysql> SELECT * FROM PassengerEmailAudit;
+----------+--------------+-------------------------+---------------------------------+---------------------+------------+
| audit_id | passenger_id | old_email               | new_email                       | change_timestamp    | changed_by |
+----------+--------------+-------------------------+---------------------------------+---------------------+------------+
|        1 |         1001 | byjiaweijiang@gmail.com | new.sp.email.updated@example.com | 2025-05-21 15:26:47 | DB_TRIGGER |
|        2 |         1028 | api.sp@example.com      | updated.api.sp@example.com      | 2025-05-21 15:44:30 | DB_TRIGGER |
+----------+--------------+-------------------------+---------------------------------+---------------------+------------+
2 rows in set (0.001 sec)
```

- **删除乘客（调用 `DeletePassengerViaSP`）**
  - **Method:** `DELETE`
  - **URL:** `http://127.0.0.1:5000/api/sp/passengers/<passenger_id_to_delete>`
  - **预期：** 返回成功消息。去数据库检查`Passenger`表是否已删除该记录。

5.3 测试并发控制

这部分主要通过运行`concurrency_tests/`目录下的Python脚本来模拟和观察。

1. **准备测试表和数据：**
   - 在MySQL客户端中，执行

     ```
     concurrency_tests/isolation_level_setup.sql
     ```

     脚本。这将创建

```
Accounts
```

表并插入一些初始数据，供并发测试使用。

```
USE AirlineDB;
```

```
mysql> source /Users/bananapig/Desktop/22281188-江家玮-数据库Lab5/concurrency_tests/isolation_level_setup.sql
Query OK, 0 rows affected (0.000 sec)

Query OK, 0 rows affected (0.000 sec)

Query OK, 0 rows affected (0.000 sec)

Query OK, 0 rows affected (0.000 sec)

+---------------------------------+
| @@SESSION.transaction_isolation |
+---------------------------------+
| REPEATABLE-READ                 |
+---------------------------------+
1 row in set (0.001 sec)

+---------------------------------+
| @@GLOBAL.transaction_isolation |
+---------------------------------+
| REPEATABLE-READ                 |
+---------------------------------+
1 row in set (0.000 sec)
```

2. **运行并发测试脚本：**
   - 打开新的终端窗口（不要关闭Flask应用的终端）。
   - 导航到 `AirlineApp_Lab7/concurrency_tests/` 目录。
   - 逐个运行脚本：
     - 脏读测试：

       ```
       python dirty_read_test.py
       ```

       READ UNCOMMITTED 测试部分：事务B在事务A提交前回滚前，读取到了事务A未提交的修改（900.00）。这**成功演示了脏读**。
       READ COMMITTED 测试部分: 事务B在事务A回滚前，读取到的是账户原始的、已提交的值（1000.00），而不是事务A未提交的修改。这表明 `READ COMMITTED` 隔离级别**成功避免了脏读**。

```
bananapig@BananadeMacBook-Pro 22281188-江家玮-数据库Lab5 % python3 concurrency_tests/dirty_read_test.py
Account 1 balance reset to 1000.0

--- Testing Dirty Read with READ UNCOMMITTED for Transaction B ---
Transaction A (Isolation: REPEATABLE READ): Starting.
Transaction A: Original balance for account 1 is 1000.00.
Transaction A: Updated balance to 900.00 (UNCOMMITTED).
Transaction B (Isolation: READ UNCOMMITTED): Starting.
Transaction B: Read balance for account 1 as 900.00.
Transaction B: Finished.
Transaction A: Rolled back changes.
Transaction A: Final balance after rollback is 1000.00.
Transaction A: Finished.
Dirty Read Test (READ UNCOMMITTED for B) finished.
Expected for READ UNCOMMITTED: Transaction B reads the uncommitted update from A.

--- Resetting balance for next test ---
Account 1 balance reset to 1000.0

--- Testing Dirty Read with READ COMMITTED for Transaction B ---
Transaction A (Isolation: REPEATABLE READ): Starting.
Transaction A: Original balance for account 1 is 1000.00.
Transaction A: Updated balance to 900.00 (UNCOMMITTED).
Transaction B (Isolation: READ COMMITTED): Starting.
Transaction B: Read balance for account 1 as 1000.00.
Transaction B: Finished.
Transaction A: Rolled back changes.
Transaction A: Final balance after rollback is 1000.00.
Transaction A: Finished.
Dirty Read Test (READ COMMITTED for B) finished.
Expected for READ COMMITTED: Transaction B reads the original committed balance, not A's uncommitted update.
```

- 不可重复读测试：

```
python non_repeatable_read_test.py
```

READ COMMITTED 测试部分：事务A在同一事务内两次读取同一数据，但由于事务B在此期间修改并提交了数据，导致事务A两次读取的结果不一致。这**成功演示了不可重复读**。
REPEATABLE READ 测试部分: 事务A在同一事务内两次读取同一数据，尽管事务B在此期间修改并提交了数据，但事务A两次读取的结果仍然一致。这表明 `REPEATABLE READ` 隔离级别**成功避免了不可重复读**。

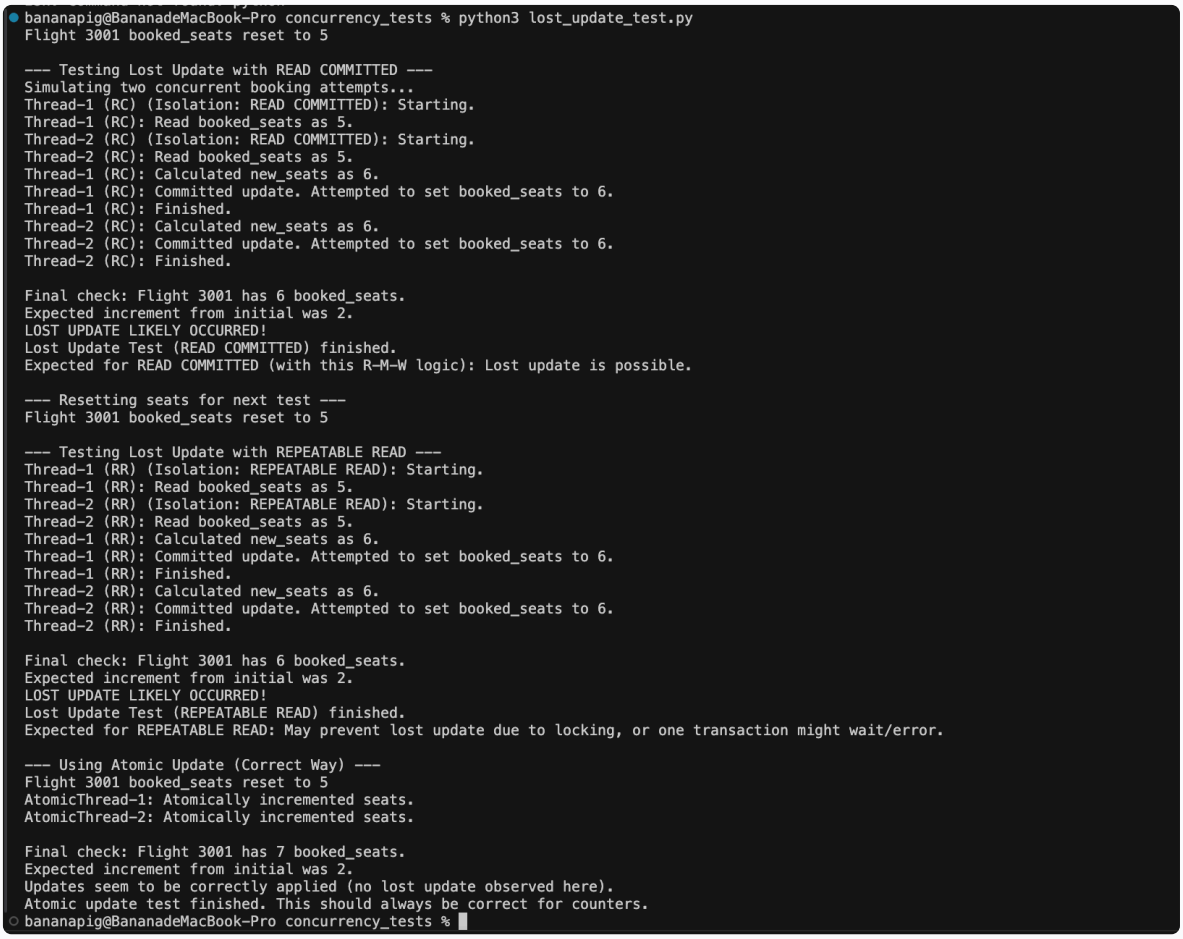
```
bananapig@BananadeMacBook-Pro concurrency_tests % python3 non_repeatable_read_test.py
Account 1 balance reset to 1000.0

--- Testing Non-Repeatable Read with READ COMMITTED for Transaction A ---
Transaction A (Isolation: READ COMMITTED): Starting.
Transaction A: First read of balance for account 1 is 1000.00.
Transaction B (Isolation: READ COMMITTED): Starting.
Transaction B: Updated balance from 1000.00 to 1200.00.
Transaction B: Committed changes.
Transaction B: Finished.
Transaction A: Second read of balance for account 1 is 1200.00.
Transaction A: Balances are NOT repeatable (Non-Repeatable Read occurred).
Transaction A: Finished.
Non-Repeatable Read Test (READ COMMITTED for A) finished.
Expected for READ COMMITTED: Transaction A sees different balances (Non-Repeatable Read).

--- Resetting balance for next test ---
Account 1 balance reset to 1000.0

--- Testing Non-Repeatable Read with REPEATABLE READ for Transaction A ---
Transaction A (Isolation: REPEATABLE READ): Starting.
Transaction A: First read of balance for account 1 is 1000.00.
Transaction B (Isolation: READ COMMITTED): Starting.
Transaction B: Updated balance from 1000.00 to 1200.00.
Transaction B: Committed changes.
Transaction B: Finished.
Transaction A: Second read of balance for account 1 is 1000.00.
Transaction A: Balances are repeatable.
Transaction A: Finished.
Non-Repeatable Read Test (REPEATABLE READ for A) finished.
Expected for REPEATABLE READ: Transaction A sees the same balance (Repeatable Read).
```

- 丢失修改测试：

```
python lost_update_test.py
```

READ COMMITTED 测试部分 (模拟非原子“读-改-写”): 两个线程都基于初始值5进行计算，并都尝试将结果更新为6。由于并发执行，最终结果是6，而不是期望的7 (5+1+1)。一个更新操作的效果丢失了。这**成功演示了丢失修改**。

REPEATABLE READ 测试部分 (模拟非原子“读-改-写”)：当使用数据库的原子操作 `UPDATE Flight SET booked_seats = booked_seats + 1 ...` 时，数据库正确处理了并发，最终结果是7，没有发生丢失修改。这**成功演示了如何避免丢失修改**。



11

六、实验总结与体会

通过本次Lab7实验，我对数据库的存储过程、触发器以及并发控制机制有了更为深入和实践性的理解。

存储过程与触发器：
我成功地为我的“航空公司乘客管理系统”设计并实现了满足查询、插入、删除、修改操作的存储过程。通过将常用的数据库操作封装在存储过程中，可以简化应用层代码，提高数据库操作的执行效率（预编译），并增强数据库操作的模块化和安全性。例如，`GetPassengerAndBookings`存储过程将复杂的多表查询逻辑封装起来，应用层只需简单调用即可。

我还实现了针对数据插入、更新和删除操作的触发器。触发器能够在特定数据库事件发生时自动执行预定义的逻辑，非常适合用于实现数据校验、审计跟踪（如我实现的AfterPassengerEmailUpdate触发器，记录乘客邮箱的变更历史到PassengerEmailAudit表）或维护数据一致性/冗余（如AfterBookingInsert和AfterBookingDelete触发器，自动更新Flight表的booked_seats）。我体会到触发器虽然功能强大，但也需要谨慎设计和使用，以避免因触发器链或复杂逻辑导致难以调试的性能问题或意外行为。

将存储过程通过Flask后端API暴露给前端调用，是典型的B/S架构实践。这使得核心业务逻辑可以更靠近数据源，集中在后端和数据库层面进行管理和优化。

遇到的问题与解决方法：
在实验过程中，我遇到了Passenger表passenger_id未能自动递增导致前端添加功能失效的问题。通过ALTER TABLE Passenger MODIFY COLUMN passenger_id INT NOT NULL AUTO_INCREMENT COMMENT '乘客ID';语句成功为该列添加了自增属性，解决了此问题。

此外，前端JavaScript中存在一个form变量未定义的ReferenceError，通过仔细检查HTML和JavaScript代码，确保了const form = document.getElementById('passengerForm');的正确声明和执行时机，并修改了部分事件绑定方式，最终修复了前端交互问题。
在并发测试时，精确控制线程的执行时序以稳定复现并发问题是一个挑战。通过在脚本中适当加入time.sleep()，并多次运行观察，可以更清晰地看到预期的并发现象。

六、附录（部分关键代码参考）

6.1 lab7_procedures_triggers.sql

```
-- 存储过程: GetPassengerAndBookings
DELIMITER //
CREATE PROCEDURE GetPassengerAndBookings(IN p_passenger_id INT) BEGIN
/* ...查询逻辑... */ END //
DELIMITER ;

-- 触发器: AfterBookingInsert
DELIMITER //
CREATE TRIGGER AfterBookingInsert AFTER INSERT ON Booking FOR EACH
ROW BEGIN UPDATE Flight SET booked_seats = booked_seats + 1 WHERE
flight_id = NEW.flight_id; END //
DELIMITER ;
```

6.2 app.py 调用存储过程（摘要）

调用 GetPassengerAndBookings 存储过程的Flask路由

```
@app.route('/api/sp/passengers/<int:passenger_id>/details', methods=
['GET'])
def get_passenger_details_sp(passenger_id):
    # ... conn = get_db_connection(); cursor =
conn.cursor(dictionary=True) ...
    cursor.callproc('GetPassengerAndBookings', (passenger_id,))
    results = []
    for result in cursor.stored_results():
        results.extend(result.fetchall())
    # ... close connection ...
    return jsonify(results)
```

6.3 concurrency_tests/lost_update_test.py (核心逻辑摘要)

模拟“读-改-写”导致丢失更新的函数

```
def simulate_booking_lost_update(thread_name, isolation_level):
    # ... conn.start_transaction() ...
    # 1. Read
    cursor.execute("SELECT booked_seats FROM Flight WHERE flight_id =
%s", (FLIGHT_ID_TO_TEST,))
    current_seats_read = cursor.fetchone()[0]
    # time.sleep(0.5) # 模拟处理时间
    # 2. Modify in application memory
    new_seats_calculated = current_seats_read + 1
    # 3. Write back
    cursor.execute("UPDATE Flight SET booked_seats = %s WHERE
flight_id = %s",
                   (new_seats_calculated, FLIGHT_ID_TO_TEST))
    conn.commit()
    # ... (异常处理和连接关闭) ...
```

使用原子操作避免丢失更新的函数

```
def atomic_increment(thread_name):

    # ... conn = get_db_connection(); cursor = conn.cursor() ...

   cursor.execute("UPDATE Flight SET booked_seats = booked_seats + 1
WHERE flight_id = %s", (FLIGHT_ID_TO_TEST,))
   conn.commit()
   #  ... (连接关闭) ...
```