华中科技大学

数据库系统原理实践报告

专　　业：	计算机科学与技术
班　　级：	2301
学　　号：	U2023xxxxx
姓　　名：	Losyi
指导教师：	翟彬彬

分数	
教师签名	

2025 年　6 月 13　日

教师评分页

子目标	子目标评分
1	
2	
3	
4	
5	

总分	

目　录

1 课程任务概述

本次实践课程以 MySQL 为例，系统性地设计了一系列的实训任务，涉及数据库基础操作、数据查询、数据库设计等多方面内容。课程依托头歌实践教学平台，实验环境为 Linux 操作系统下的 MySQL 8.0.28。

实训任务共十六个，每个实训内有多个关卡，若所有关卡全部完成，将会获得头歌平台总分超过 100 分，然而课程考核并不要求所有关卡全部完成，最终程序检查满分只计头歌平台中的 100 分。课程任务的具体分解如下：

(1) 数据库数据对象的管理与编程

核心内容：掌握数据库、表、索引、视图、约束（主码 / 外码 / CHECK 等）、存储过程、函数、触发器、游标等对象的创建与管理语法。涉及实训 1：创建数据库及表，设置完整性约束。实训 2：使用 ALTER 语句修改表结构与约束。实训 7 - 10：视图、存储过程、触发器、用户自定义函数的开发与应用。

(2) 数据处理相关任务

核心内容：熟练完成数据查询、插入、删除、修改等操作。涉及单表与多表关联查询、复杂统计分析。涉及实训 3 - 5：基于金融场景数据库，完成基础查询、统计分析及相似性推荐查询。实训 6：实现数据的增删改操作

(3) 数据库系统内核实验

核心内容：掌握数据库安全性控制（用户权限管理）、完整性控制、恢复机制（备份与还原）、并发控制（事务隔离级别）等内核机制。涉及实训 11：数据库安全性控制（用户权限分配与管理）。实训 12：并发控制与事务隔离级别实验（模拟事务并发执行）。实训 13：数据库备份与恢复（逻辑备份 residents_bak.sql 及还原）。实训 16：存储管理（缓冲池管理器实现，涉及磁盘与内存数据调度）。

(4) 数据库的设计与实现

核心内容：从需求分析到逻辑模型设计，完成数据库建模与物理实现，包括 E-R 图设计、关系模式转换及 SQL 脚本生成。涉及实训 14：从概念模型到 MySQL 实现（基于 E-R 图设计数据库），影院管理系统设计（绘制 E-R 图、生成关系模式，利用 MySQL Workbench 建模工具导出 SQL 脚本。

(5) 数据库应用系统的开发（JAVA 篇）

核心内容：基于 JDBC 技术实现数据库应用开发，包括数据查询、用户认证、事务处理及业务逻辑实现。

2 任务实施过程与分析

本次实践课程在头歌平台进行，实践任务均在平台上提交代码，所有完成的任务、关卡均通过了自动测评。本次实践最终完成了课程平台中的第 10~11、1~7、14~15 实训任务，其中实训 5 和实训 14 跳过了两个子关卡、下面将重点针对其中的数据查询有关（实训 3-5）、数据库设计与实现（实训 14）、Java 数据库应用开发（实训 15）、任务阐述其完成过程中的具体工作。

2.1 基于金融应用的数据查询(Select)

本小节，即实训三，采用的是某银行的一个金融场景应用的模拟数据库 finance，该数据库共包含六个表，客户表、银行卡表、理财产品表、保险表、基金表、资产表。我们需要用 SQL 语句完成各关卡的给出的查询任务。

本任务已完成全部关卡（共 19 个）。

2.1.1 投资总收益前三名的客户

本次任务需要查询当前可用资产收益前三名的客户信息，包括客户姓名、身份证号及其总收益，并按收益降序输出。任务整体较为简单吗，但涉及了多个知识点，我设计了一条高效的 SQL 查询语句来完成这一任务。

```
SELECT
    c.c_name,c.c_id_card,t.total_income
FROM
client c
JOIN (
    SELECT pro_c_id,SUM(pro_income) AS total_income
    FROM property
    WHERE pro_status = '可用'
    GROUP BY pro_c_id
) t ON c.c_id = t.pro_c_id
ORDER BY t.total_income DESC
LIMIT 3;
```

该 SQL 语句的核心思路是先从 property 表中筛选出状态为"可用"的资产记录，按客户 ID 分组计算出每个客户的总收益，然后将这一结果与 client 表关联获取客户的基本信息。

这条 SQL 语句虽然简洁，但包含了多表连接、分组统计、派生表、子查询

和 TOP N 查询等多个关键技术点。在性能方面，该查询需要对 property 表进行一次全表扫描，执行分组聚合操作，然后与 client 表进行哈希连接，最后进行排序。

2.1.2 持有完全相同基金组合的客户

本次任务需要找出持有完全相同基金组合的客户对，如编号为 A 的客户持有的基金，编号为 B 的客户也持有，反过来，编号为 B 的客户持有的基金，编号为 A 的客户也持有。要求每对客户只显示一次（编号小的在前）。我设计了一个基于公共表表达式(CTE)的简单解决方案，通过以下步骤实现：

首先，创建了一个名为 client_funds 的 CTE，用于计算每个客户投资的基金组合。在这个 CTE 中，从 property 表中筛选出基金投资记录（pro_type=3），然后使用 GROUP_CONCAT 函数将每个客户持有的基金 ID 连接成一个有序字符串。这里特别使用了 DISTINCT 关键字确保基金 ID 不重复，并通过 ORDER BY 保证结果的一致性。接着，在主查询中，对这个 CTE 进行自连接，通过条件 c1.pro_c_id < c2.pro_c_id 确保每对客户只出现一次（编号小的在前），并且只选择基金组合完全相同的客户对。

这个解决方案的关键在于使用 GROUP_CONCAT 函数将多行基金 ID 转换为可比较的字符串，大大简化了比较的实现过程。

```
WITH client_funds AS (
    -- 从 property 表中选择客户 ID 和该客户投资的基金组合
    SELECT pro_c_id,
        -- GROUP_CONCAT 函数用于将分组内的多行数据连接成一个字符串
        -- DISTINCT 确保连接的基金 ID 不重复
        -- ORDER BY pro_pif_id 按基金 ID 排序，保证结果的一致性
        GROUP_CONCAT(DISTINCT pro_pif_id ORDER BY pro_pif_id)
AS fund_combination
    FROM property
    WHERE  pro_type = 3
        -- 只筛选 pro_type 为 3 的记录，可理解为只关注基金投资
    GROUP BY pro_c_id
        -- 按客户 ID 分组，以便统计每个客户的基金组合
)
-- 主查询部分，用于找出具有相同基金组合的不同客户对
SELECT c1.pro_c_id AS c_id1,c2.pro_c_id AS c_id2
FROM  client_funds c1
    -- 从 CTE client_funds 中选择记录，别名为 c1
```

```
JOIN client_funds c2 ON c1.pro_c_id < c2.pro_c_id
    -- 将 CTE client_funds 进行自连接，别名为 c2
WHERE c1.fund_combination = c2.fund_combination;
    -- 筛选条件：只有当两个客户的基金组合相同时才会被选中
```

根据题目提示，我们还可以采用 NOT EXISTS 子查询的方式，通过双重否定来验证两个客户的基金组合完全相同，具体代码如下。

```
SELECT DISTINCT
    p1.pro_c_id AS c_id1,
    p2.pro_c_id AS c_id2
FROM property p1
JOIN property p2 ON p1.pro_c_id < p2.pro_c_id
WHERE
    p1.pro_type = 3 AND p2.pro_type = 3  -- 假设 3 代表基金类型
    AND NOT EXISTS (
        SELECT pro_pif_id
        FROM property p3
        WHERE p3.pro_c_id = p1.pro_c_id
        AND p3.pro_type = 3
        AND NOT EXISTS (
            SELECT 1 FROM property p4
            WHERE p4.pro_c_id = p2.pro_c_id
            AND p4.pro_type = 3
            AND p4.pro_pif_id = p3.pro_pif_id
        )
    )
    AND NOT EXISTS (
        SELECT pro_pif_id FROM property p5
        WHERE p5.pro_c_id = p2.pro_c_id
        AND p5.pro_type = 3
        AND NOT EXISTS (
            SELECT 1
            FROM property p6
            WHERE p6.pro_c_id = p1.pro_c_id
            AND p6.pro_type = 3
```

```
            AND p6.pro_pif_id = p5.pro_pif_id
        )
    );
```

该方案直接连接 property 表，使用两个对称的 NOT EXISTS 子查询结构，分别确保：第一个客户持有的所有基金第二个客户也都持有，且第二个客户持有的所有基金第一个客户也都持有。这种方案的优势是不需要处理长字符串，避免了 GROUP_CONCAT 的长度限制问题，但查询结构相对复杂，性能可能较差，特别是当数据量较大时。

两方案各有优缺点：第一种方案代码简洁，执行效率较高，但可能受限于 GROUP_CONCAT 的长度限制，第二种方案逻辑严谨，不受字符串长度限制，但查询复杂度高，执行效率可能较低。在实际应用中，如果基金组合数量较少，建议使用第一种方案，如果基金组合可能包含大量基金，则第二种方案更为稳妥。两种方案都通过 p1.pro_c_id < p2.pro_c_id 的条件确保了结果不重复，满足了题目要求。

2.1.3 以日历表格式显示每日基金购买总金额

本任务要求以日历表格式呈现 2022 年 2 月每周各交易日的基金购买总金额，规定 2 月 7 日（周一）为第 1 周首个交易日。实现方案通过 SQL 关联 property 与 fund 表，筛选 pro_type=3 的基金购买记录（时间范围 2022-02-07 至 2022-02-28），并利用 WEEKDAY<5 过滤非交易日。核心在于通过 WEEK(pro_purchase_time,1)-WEEK('2022-02-07',1)+1 计算相对周次，确保 2 月 7 日为第 1 周，再借助 IF(WEEKDAY=X)配合 SUM 实现按周一至周五的金额聚合，最终按周次分组排序输出。

```
SELECT
    WEEK(pro_purchase_time, 1) - WEEK('2022-02-07', 1) + 1 AS
week_of_trading,
    -- 返回 pro_purchase_time 的 ISO 周次（周一是每周第一天）
    SUM(IF(WEEKDAY(pro_purchase_time) = 0, pro_quantity * f_am
ount, NULL)) AS Monday,
    SUM(IF(WEEKDAY(pro_purchase_time) = 1, pro_quantity * f_am
ount, NULL)) AS Tuesday,
    SUM(IF(WEEKDAY(pro_purchase_time) = 2, pro_quantity * f_am
ount, NULL)) AS Wednesday,
    SUM(IF(WEEKDAY(pro_purchase_time) = 3, pro_quantity * f_am
ount, NULL)) AS Thursday,
```

```
    SUM(IF(WEEKDAY(pro_purchase_time) = 4, pro_quantity * f_am
ount, NULL)) AS Friday
FROM
    property p
JOIN
    fund f ON p.pro_pif_id = f.f_id
WHERE
    p.pro_type = 3
    AND p.pro_purchase_time BETWEEN '2022-02-07' AND '2022-02-
28'
    AND WEEKDAY(p.pro_purchase_time) < 5
GROUP BY
    WEEK(pro_purchase_time, 1) - WEEK('2022-02-07', 1) + 1
ORDER BY
    week_of_trading;
```

具体实现中，JOIN 操作通过 pro_pif_id=f_id 关联资产与基金表以获取单价与数量，WHERE 条件精准限定数据范围。周次计算采用 ISO 标准（周一为周首日），通过基准日差值避免自然年周次干扰，金额聚合时，IF(WEEKDAY=0,金额,NULL)等条件判断将每日数据按星期分类，SUM 自动忽略 NULL 值实现累加，形成行转列效果。例如 SUM(IF(WEEKDAY=0,...)) AS Monday 将所有周一的交易金额汇总至同一列，确保日历表格式。

该方案的技术关键点在于基准周次的差值计算逻辑，以及 WEEKDAY 与 SUM 的组合运用。对于大数据场景，建议在 pro_purchase_time 和 pro_type 建立索引以优化查询效率，其行转列的设计也为同类金融数据可视化提供了可复用模板，最终结果能准确呈现每周各交易日的基金购买情况，满足日历表的展示需求。

2.2 数据查询（Select）-新增 2

本小节为实训五，本节使用的数据库结构较为简单，仅包含两个表：客户表与薪资表。我们需要用 SQL 语句完成各关卡的给出的查询任务。

本任务我完成了前四个关卡。

2.2.1 客户年度从各单位获得的酬劳总额

本任务要求统计客户年度从各单位获得的全职和兼职酬劳总额，并按总金额降序排序。我们可以结合条件聚合和左连接来实现查询。查询核心在于使用 EXTRACT (YEAR FROM w.w_time) 提取薪资年份，借助 CASE WHEN 表达式分别聚合 w_type=1（全职）和 w_type=2（兼职）的酬劳金额，并用 COALESCE

将空值转换为 0，最终按全职与兼职金额之和排序输出。

```
SELECT
    c.c_name,
    EXTRACT(YEAR FROM w.w_time) AS year,
    c.c_id_card,
    COALESCE(SUM(CASE WHEN w.w_type = 1 THEN w.w_amount ELSE 0
 END), 0) AS full_t_amount,
    COALESCE(SUM(CASE WHEN w.w_type = 2 THEN w.w_amount ELSE 0
 END), 0) AS part_t_amount
FROM
    client c
LEFT JOIN
    wage w ON c.c_id = w.w_c_id
WHERE
    w.w_c_id IS NOT NULL  -- 确保只统计有效客户关联的记录
GROUP BY
    c.c_id, c.c_name, c.c_id_card, EXTRACT(YEAR FROM w.w_time)
ORDER BY
    full_t_amount + part_t_amount DESC;  -- 按全职和兼职总金额降
序排序
```

具体实现中，LEFT JOIN 的运用保证了客户记录的完整性，避免遗漏无薪资数据的客户。CASE WHEN 条件聚合通过 “WHEN w_type=1 THEN w_amount ELSE 0 END” 的逻辑，将同一客户同一年份的全职与兼职酬劳分别累加，例如 SUM (CASE WHEN w_type=1...) 会汇总该客户所有全职记录的金额。EXTRACT 函数精准获取薪资时间的年份信息，确保按年度分组统计，COALESCE 处理聚合结果为空的情况（如某客户无全职收入时，SUM 结果为 NULL，经转换后显示为 0）。

2.2.2 对身份证号为 420108199702144323 的客户 2023 年的酬劳代扣税

本任务需为身份证号 420108199702144323 的客户计算并扣除 2023 年度个人所得税，根据题目提示，我们可以采用 CTE 与 UPDATE 语句结合的分步骤 SQL 方案实现完整税务处理流程。方案首先通过 client_income CTE 关联 client 和 wage 表，筛选指定身份证号及 2023 年薪资记录，用 SUM 汇总全年收入，接着在 tax_calculation CTE 中，运用 GREATEST 函数确保仅对超过 6 万元的收入部分按 20% 税率计税，最后通过 UPDATE 语句按每月薪资占比分摊全年税额，并依税额是否大于 0 设置税务标志 'Y' 或 'N'。

CTE 的运用将复杂计算拆解为可复用的逻辑块：client_income 通过 JOIN 精准定位目标客户及年度收入，避免无效数据干扰，tax_calculation 中的 GREATEST (total_salary - 60000, 0) 确保免税额度内金额不产生负税额，符合税务规则。UPDATE 环节的比例分摊逻辑（w.w_amount/tc.total_salary）保证每月扣税金额与收入占比匹配，例如某笔薪资占全年收入 10%，则扣除总税额的 10%，确保扣税公平性。该方案的技术核心在于分层计算与精准分摊：GREATEST 函数处理免税阈值，比例分摊避免税额均摊带来的误差，IF 条件实现税务标志的自动化标记。

```
-- 1. 计算客户2023年总收入
WITH client_income AS (
    SELECT
        c.c_id,
        SUM(w.w_amount) AS total_salary
    FROM client c
    JOIN wage w ON c.c_id = w.w_c_id
    WHERE
        c.c_id_card = '420108199702144323'
        AND YEAR(w.w_time) = 2023
    GROUP BY c.c_id
),
-- 2. 计算应缴税额
tax_calculation AS (
    SELECT c_id,total_salary,
        GREATEST(total_salary - 60000, 0) * 0.2 AS total_tax
    FROM client_income
)
-- 3. 更新每月扣税记录
UPDATE wage w
JOIN client c ON w.w_c_id = c.c_id
JOIN tax_calculation tc ON c.c_id = tc.c_id
SET
    w.w_amount = w.w_amount - (tc.total_tax * (w.w_amount / tc.
total_salary)),
    w.w_tax = IF(tc.total_tax > 0, 'Y', 'N')
WHERE c.c_id_card = '420108199702144323'
```

```
    AND YEAR(w.w_time) = 2023;
```

2.3 用户自定义函数

本小节为实训 10，只包含一个关卡“创建函数并在语句中使用它”，该任务要求我们写一个依据客户编号计算其在本金融机构的存储总额的函数,并在 SELECT 语句使用这个函数。查询存款超 100 万的客户信息。

本任务要求创建计算客户储蓄卡存款总额的自定义函数，并根据题目讲解，我们创建函数时，先通过 DELIMITER $$修改分隔符以支持多行语句定义，使用 CREATE FUNCTION 声明函数 get_deposit，参数 client_id 为整数类型，返回值定义为 NUMERIC(10,2)。函数内部通过 DECLARE total 声明局部变量，利用 SELECT SUM(b_balance)查询 bank_card 表中指定客户 ID 且类型为“储蓄卡”的余额总和，结果存入变量后返回。例如，当客户 ID 为 1 时，函数会筛选 b_c_id=1 且 b_type='储蓄卡'的记录，累加 b_balance 生成总额。

应用层面，通过 SELECT c_id_card, c_name, get_deposit(c_id) AS total_deposit 调用函数，结合 WHERE get_deposit(c_id) >= 1000000 筛选存款超 100 万的客户，并按 total_deposit 降序排列。

```
delimiter $$
create function get_deposit(client_id int)
returns numeric(10,2)
begin
    declare total numeric(10,2);
    select sum(b_balance) into total
    from bank_card
    where b_c_id = client_id and b_type = '储蓄卡';
    return total;
end$$
delimiter ;
/*  应用该函数查询存款总额在 100 万以上的客户身份证号，姓名和存储总额(total_deposit)，结果依存款总额从高到代排序  */
select c_id_card, c_name, get_deposit(c_id) as total_deposit
from client
where get_deposit(c_id) >= 1000000
order by total_deposit desc;
```

2.4 视图

本小节为实训 7，仍然使用之前的金融场景数据库，为完成本关任务，我们需要学习如何创建和使用视图。

本小节我完成了全部关卡（共 2 个）。

2.4.1 创建所有保险资产的详细记录视图

本关需要创建包含所有保险资产记录的详细信息的视图 v_insurance_detail，包括购买客户的名称、客户的身份证号、保险名称、保障项目、商品状态、商品数量、保险金额、保险年限、商品收益和购买时间。本质是在创建视图的语法 AS 后完成一个查询任务。

分析业务需求，明确需要整合客户、保险产品和购买记录三方面数据。通过 INNER JOIN 关联 client、insurance 和 property 三张表，其中 property 表的 pro_type=2 标识保险类资产。视图包含客户姓名、身份证号、保险名称、保障项目等关键字段，并按客户姓名和购买时间排序，便于业务查询。

```
CREATE VIEW v_insurance_detail AS
SELECT
    c.c_name,
    c.c_id_card,
    i.i_name,
    i.i_project,
    p.pro_status,
    p.pro_quantity,
    i.i_amount,
    i.i_year,
    p.pro_income,
    p.pro_purchase_time
FROM  property p
JOIN client c ON p.pro_c_id = c.c_id
JOIN insurance i ON p.pro_pif_id = i.i_id
WHERE p.pro_type = 2  -- 2表示保险类型
ORDER BY c.c_name, p.pro_purchase_time;
```

2.4.2 基于视图的查询

本关基于视图 v_insurance_detail 查询每位客户保险资产的总额和保险总收益。我们将视图看作一个表，本质上仍然是一个查询任务。

具体代码通过分组统计实现：使用 SUM (i_amount * pro_quantity) 计算保险

投资总额，SUM (pro_income) 计算总收益，按客户姓名和身份证号分组，结果依保险投资总额降序排列。

```
SELECT
    c_name,
    c_id_card,
    SUM(i_amount * pro_quantity) AS insurance_total_amount,
    SUM(pro_income) AS insurance_total_revenue
FROM
    v_insurance_detail
GROUP BY
    c_name, c_id_card
ORDER BY
    insurance_total_amount DESC;
```

2.5 数据库设计与实现

本小节主要包括数据库设计方面的内容，具体而言我们学习和完成以下内容。
需求分析：根据业务需求，确定应用系统所涉及的数据(信息)，以及处理需求。
概念结构设计：对数据建模，主要是用 ER 图描述数据及数据间的关系。
逻辑结构设计：最主要的工作就是把 ER 图转换成关系模式。
建模工具使用：将一个建好的模型文件，利用 MySQL Workbench 的 forward engineering 功能，自动转换成 SQL 脚本。
本任务我完成了第一个关卡。

2.5.1 从概念模型到 MySQL 实现

本关的任务，是根据一个已建好的逻辑模型，完成 MySQL 的实现。逻辑 ER 图如图 2-1 所示。我们需要根据题目信息和所给 ER 图，给出在 MySQL 实现 flight_booking 的语句，包括建库，建表，创建主码，外码，索引，指定缺省，不能为空等约束。所有索引采用 BTREE。

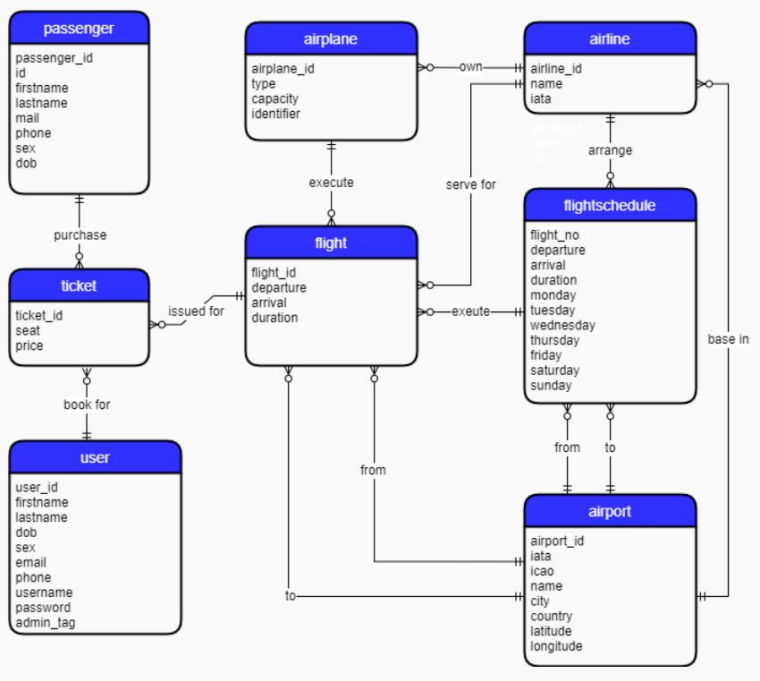

图 2-1 机票订票系统概念模型 ER 图

具体实现流程：理解 ER 图实体关系→设计表结构→处理关键字段冲突→实现约束与索引。ER 图涉及 9 个实体（用户、旅客、机场等）及复杂关联（如航空公司与机场的一对多、航班与飞机的多对一），需重点处理外码命名冲突（如出发 / 到达机场字段 from、to）和关键字段兼容性。下面，我们以第一个表为例进行说明代码构建过程。

1.用户(user)

用户分两类，普通用户可以订票，管理用户有权限维护和管理整个系统的运营。为简单起见，两类用户合并，用 admin_tag 标记区分。用户的属性(包括业务约束)有：

用户编号: user_id int 主码,自动增加

名字: firstname varchar(50) 不可为空

姓氏: lastname varchar(50) 不可为空

生日: dob date 不可为空

性别: sex char(1) 不可为空

邮箱: email varchar(50)

联系电话: phone varchar(30)

用户名: username varchar(20) 不可空,不可有重

密码: password char(32) 不可空

管理员标志: admin_tag tinyint 缺省值 0(非管理员),不能空

```
DROP TABLE IF EXISTS user;
```

```
CREATE TABLE user (
    user_id INT AUTO_INCREMENT PRIMARY KEY,
    firstname VARCHAR(50) NOT NULL,
    lastname VARCHAR(50) NOT NULL,
    dob DATE NOT NULL,
    sex CHAR(1) NOT NULL,
    email VARCHAR(50),
    phone VARCHAR(30),
    username VARCHAR(20) NOT NULL UNIQUE,
    password CHAR(32) NOT NULL,
    admin_tag TINYINT NOT NULL DEFAULT 0
) ENGINE=InnoDB CHARACTER SET=utf8mb4 COLLATE=utf8mb4_unicode_
ci ROW_FORMAT=Dynamic;
```

该表的设计采用自增 INT 类型主键 user_id，确保唯一性且插入高效，其数值范围完全满足用户规模需求。姓名字段分离为 firstname 和 lastname，使用 VARCHAR(50)支持超长姓名并设置 NOT NULL 约束保证数据完整性，这种设计兼顾了东西方命名习惯。关键业务字段 username 设置为 VARCHAR(20)并添加 UNIQUE 约束防止重复注册，password 使用 CHAR(32)并强制非空以杜绝安全隐患。权限控制字段 admin_tag 采用 TINYINT 类型，默认值设为 0 确保新建用户默认为普通用户，NOT NULL 约束强制显式指定权限状态。基础属性如 sex 和 dob 都设置 NOT NULL 约束避免业务逻辑异常。在技术实现上选用 InnoDB 存储引擎以支持事务和行级锁定，特别适合高并发的用户注册/登录场景，字符集配置为 utf8mb4 配合 utf8mb4_unicode_ci 排序规则，完整支持 emoji 和所有 Unicode 字符的同时实现大小写不敏感的用户名比较，ROW_FORMAT 设置为 Dynamic 优化存储空间利用率并支持大字段高效存储。创建语句首先使用 DROP TABLE IF EXISTS 确保安全删除已有表结构，完整的 CREATE TABLE 语句包含所有字段定义和表级配置。

本关的难点是各种细节，包括约束的建立，命名，数据库的建立和删除等，下面是我完成过程中出现的一些经典错误。

(1)忘记在创建数据库前检查是否已存在同名数据库：由于没有删除掉之前创建的数据库，导致我对代码进行修改后，运行结果仍与之前的错误相同。

(2)外键缺失约束：在最初的 flight 表设计中，遗漏了对 flightschedule(flightno)的外键约束，缺少此约束会导致数据不一致，可能产生无调度记录的航班。

(3)表创建顺序问题：尝试在创建被引用表之前创建引用表。如果外键约束立

即生效，会导致创建失败。

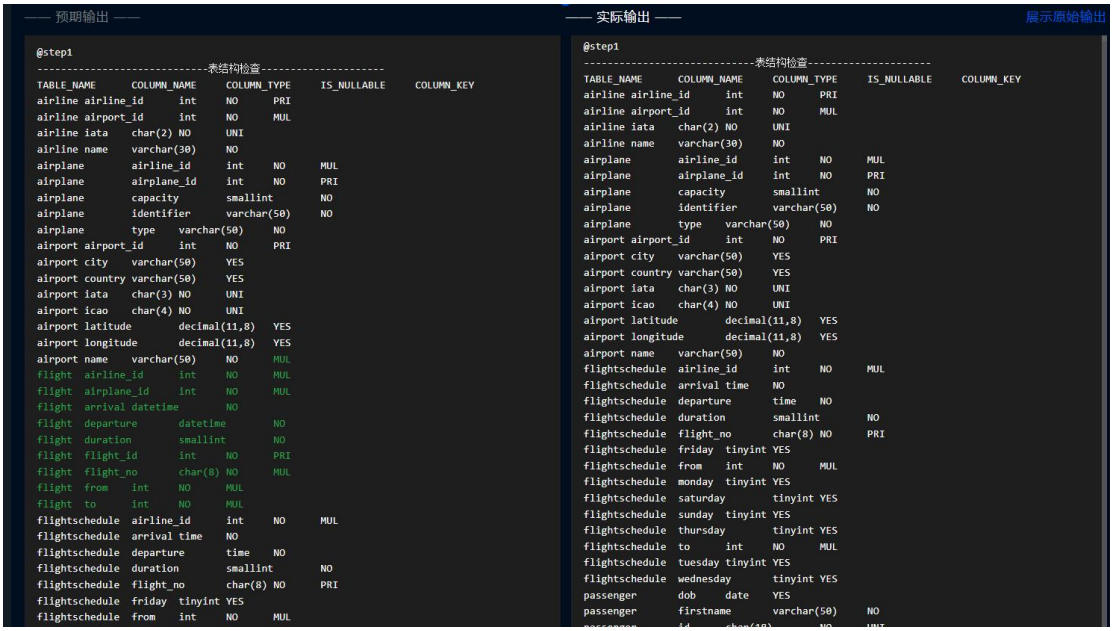

图 2-2 错误的输出截图

2.5.2 制约因素分析与设计

本数据库的设计充分考虑了现实中的各种制约因素。例如，在安全方面，通过 user 表 admin_tag 字段区分普通用户与管理员（默认 0 为非管理员），管理员可维护系统数据（如航班调度），普通用户仅能购票，符合《网络安全法》中“最小权限原则”，避免越权操作引发法律风险。旅客身份证号(CHAR(18))设置唯一约束，既满足业务需求又符合个人信息保护要求。安全方面，机场表记录 latitude 和 longitude（十进制坐标，精度 11,8），支持地图可视化与飞行路径规划，避免因机场位置误差导致导航系统故障，间接保障航空安全。社会文化方面，航班常规调度表 flightschedule 通过 monday-sunday 字段标记每周班次，支持法定节假日（如春节、国庆）的航班调整，符合民航局对特殊日期运力调配的要求，避免因调度冲突导致航班延误或安全事故。所有表使用 utf8mb4 字符集与 unicode_ci 校对规则，支持中、英等多语言机场名称（如“北京首都机场”“London Heathrow”），避免因字符集不兼容导致文化地名显示乱码，提升国际用户使用体验。最终实现的数据库架构既符合工程认证规范，又为机票订票系统的稳定运行提供了全面的制约因素解决方案。

2.5.3 工程师责任及其分析

在机票预订系统的数据库设计中，工程师需兼顾技术实现与社会、安全、法律等多维影响。从社会层面看，系统涉及用户身份证号、联系方式等敏感信息，通过 admin_tag 字段区分用户权限，以严格的访问控制降低数据泄露风险，落实隐私保护责任。

健康安全方面，机场经纬度采用 DECIMAL(11,8)精确记录，为航班导航提供数据支撑，航班调度表与实际航班表分离设计，确保计划灵活调整，避免因数据误差导致航班延误。这种严谨的数据建模间接保障了航空运输安全。法律合规上，旅客身份证号设为 CHAR(18)并添加唯一约束，契合国内身份证标准，航空公司与机场的 IATA/ICAO 编码遵循国际规范，规避数据格式引发的法律风险。设计需贴合行业法规，从结构到逻辑确保合法合规。文化适配同样重要，用户姓名字段设计兼容多元命名习惯，采用 utf8mb4 字符集支持多语言输入，避免字符编码导致的数据显示问题，体现对文化多样性的考量。

综上，工程师在技术落地中需肩负社会责任，既是技术方案的实施者，也是社会价值的维护者。未来实践应强化责任意识，将技术能力与社会担当结合，推动系统设计兼具高效性与伦理合规性。

2.6 数据库应用开发(JAVA 篇)

本任务需要我们了解 JDBC（Java DataBase Connectivity,java 数据库连接）JDBC 和使用方法，学习如何使用 JDBC 进行编程，具体包括数据库的连接，查询的实现，从结果集中提取数据。我们需要正确使用 JDBC，对金融应用场景数据库 finance 的 client 表进行查询、修改等操作。

本小节我完成了全部关卡（共 7 个）。

2.6.1 JDBC 体系结构和简单的查询

本关本需要查询 client 表中邮箱非空的客户信息，列出客户姓名，邮箱和电话。为了完成本关任务，我们需要掌握使用步骤：

(1) 导入包：需要包含包含数据库编程所需的 JDBC 类的包。大多数情况下，使用 import java.sql.*就足够了。

(2) 注册 JDBC 驱动程序：要求您初始化驱动程序，以便您可以打开与数据库的通信通道。使用 `Class.forName("com.mysql.cj.jdbc.Driver")`

(3) 打开连接：需要使用 DriverManager.getConnection()方法创建一个 Connection 对象，该对象表示与数据库的物理连接。本关卡对应的 url 为 `"jdbc:mysql://localhost:3306/finance?useSSL=false&serverTimezone=UTC"`

(4) 执行查询：需要使用类型为 Statement 的对象来构建和提交 SQL 语句到数据库。

(5) 从结果集中提取数据：需要使用相应的 ResultSet.getXXX（）方法从结果集中检索数据。

(6) 释放资源：需要明确地关闭所有数据库资源，而不依赖于 JVM 的垃圾收集。同时，我们需要使用 try\catch 处理异常，还要注意使用\t 控制格式以符合题

目要求，最终得代码如下。

```
public static void main(String[] args) {
        Connection connection = null;
        Statement statement = null;
        ResultSet resultSet = null;
        try {
            // 1. 加载 JDBC 驱动
            Class.forName("com.mysql.cj.jdbc.Driver");
            // 2. 建立数据库连接
            String url = "jdbc:mysql://localhost:3306/finance?
useSSL=false&serverTimezone=UTC";
            String username = "root";
            String password = "123123";
            connection = DriverManager.getConnection(url, user
name, password);
            // 3. 创建 Statement 对象
            statement = connection.createStatement();
            // 4. 执行 SQL 查询
            String sql = "SELECT c_name, c_mail, c_phone FROM
client WHERE c_mail IS NOT NULL";
            resultSet = statement.executeQuery(sql);
            // 5. 输出表头（注意制表符数量）
            System.out.println("姓名\t 邮箱\t\t\t\t 电话");
            // 6. 处理结果集
            while (resultSet.next()) {
                String name = resultSet.getString("c_name");
                String mail = resultSet.getString("c_mail");
                String phone = resultSet.getString("c_phone");
                System.out.println(name + "\t" + mail + "\t\t"
 + phone);
            }
        } catch (ClassNotFoundException e) {
            System.out.println("Sorry,can`t find the JDBC Driv
er!");
            e.printStackTrace();
```

```
        } catch (SQLException throwables) {
            throwables.printStackTrace();
        } finally {
            try {
                if (resultSet != null) resultSet.close();
                if (statement != null) statement.close();
                if (connection != null) connection.close();
            } catch (SQLException throwables) {
                throwables.printStackTrace();
            }
        }
    }
```

2.6.2 客户修改密码

本关要求编写修改客户登录密码的的方法。客户修改密码通常需要确认客户身份，即客户需提供用户名(以邮箱为用户名)和密码，同时还需要输两次新密码，以免客户实际输入的密码与心中想的不一致，只有当所有条件(合法的客户，两次密码输入一致)时才修改密码。

在实现客户密码修改功能时，构建了双层验证机制以确保安全性与准确性。`"SELECT c_id FROM client WHERE c_mail = ? AND c_password = ?"`预编译查询验证用户身份，利用 PreparedStatement 的 setString 方法绑定邮箱和旧密码参数，有效防范 SQL 注入风险。当查询无结果时，进一步执行 SELECT c_id FROM client WHERE c_mail = ?单独校验邮箱存在性，以此明确区分“用户不存在”（返回 2）和“密码错误”（返回 3）的场景，保证错误提示的精准性。

在通过了上述验证流程后，我们同样使用 PreparedStatement 传入更新语句`"UPDATE client SET c_password = ? WHERE c_mail = ?"`新密码与邮箱均采用参数化方式传入，确保特殊字符正确处理的同时避免注入漏洞。通过 executeUpdate()的返回值判断受影响行数，若大于 0 则返回 1 表示密码修改成功，异常时返回 - 1，形成覆盖“成功 - 用户不存在 - 密码错误 - 异常”的完整业务状态码体系。

此外注意异常处理，在 try 块中依次创建 checkStmt 和 updateStmt 处理验证与更新操作，验证时通过 ResultSet.next()直接判断结果以减少对象创建，finally 块按 ResultSet→Statement 的顺序关闭资源，每个关闭操作均单独捕获 SQLException，防止因某资源关闭失败导致的连接泄漏。

```
public static int passwd(Connection connection,
                         String mail,
                         String password,
                         String newPass) {
        PreparedStatement checkStmt = null;
        PreparedStatement updateStmt = null;
        ResultSet rs = null;

        try {
            // 1. 验证用户是否存在且密码正确
            String checkSql = "SELECT c_id FROM client WHERE c
_mail = ? AND c_password = ?";
            checkStmt = connection.prepareStatement(checkSql);
            checkStmt.setString(1, mail);
            checkStmt.setString(2, password);
            rs = checkStmt.executeQuery();

            if (!rs.next()) {
                // 用户不存在或密码错误
                String checkUserSql = "SELECT c_id FROM client
 WHERE c_mail = ?";
                PreparedStatement userCheckStmt = connection.p
repareStatement(checkUserSql);
                userCheckStmt.setString(1, mail);
                ResultSet userRs = userCheckStmt.executeQuery()
;

                if (!userRs.next()) {
                    return 2; // 用户不存在
                } else {
                    return 3; // 密码不正确
                }
            }

            // 2. 更新密码
```

```
            String updateSql = "UPDATE client SET c_password =
 ? WHERE c_mail = ?";
            updateStmt = connection.prepareStatement(updateSql)
;
            updateStmt.setString(1, newPass);
            updateStmt.setString(2, mail);

            int rowsAffected = updateStmt.executeUpdate();

            if (rowsAffected > 0) {
                return 1; // 密码修改成功
            } else {
                return -1; // 更新失败
            }
        } catch (SQLException e) {
            e.printStackTrace();
            return -1; // 程序异常
        } finally {
            // 关闭资源
            try {
                if (rs != null) rs.close();
                if (checkStmt != null) checkStmt.close();
                if (updateStmt != null) updateStmt.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
```

2.6.3 事务与转账操作

本关需要编写一个银行卡转账的方法，转帐方法只接受转出卡号，转入卡号和转账金额三个参数。由调用者保证转账金额为正数。本方法需开启手工事务，并正确使用 commit 和 rollback。

转账操作的核心实现分为四个关键步骤，通过严格的账户验证和差异化的金

额处理确保交易安全可靠。首先使用 SELECT...FOR UPDATE 锁定查询记录，检查转出账户是否存在且非信用卡类型，同时验证转入账户是否存在，这两个查询通过预编译语句防止 SQL 注入，其中转出账户检查额外排除信用卡类型（信用卡不可作为转出源）。余额验证阶段直接比较转出账户当前余额与转账金额，不足时立即终止流程。

转账执行阶段采用差异化处理策略：对于转出账户统一执行 b_balance - ?的扣款操作，转入账户则根据类型区分处理——普通账户直接增加余额（b_balance + ?），信用卡账户则采用三段式逻辑：当信用卡存在透支（余额为正）时，转账金额优先偿还欠款，若偿还后仍有剩余则转为存款（更新为负值），无透支时直接作为存款处理。所有更新操作均使用参数化查询确保数据安全，金额计算在应用层完成后再通过预编译语句传入，避免数据库端计算错误。最终通过显式调用 connection.commit()提交事务，保证两个账户的余额更新作为原子操作同时生效。这种实现既满足基础转账需求，也正确处理了信用卡还款的特殊业务场景。

本关易错点在于信用卡的余额代表的是透支透支金额，若偿还后仍有剩余则转为存款，存款应为负值。

由于代码过长，具体代码请见附录 1。

2.6.4 把稀疏表格转为键值对存储

本关需要将一个稀疏的表中有保存数据的列值，以键值对(列名，列值)的形式转存到另一个表中，这样可以直接丢失没有值列。

在处理稀疏表转换工作时，可以采用了双层遍历的方式来处理原始数据。具体而言，首先执行一条 SQL 查询，即 SELECT * FROM entrance_exam，以此获取所有学生的成绩记录。在处理每一行数据时，通过循环检查各个科目的成绩是否有效（借助 rs.wasNull()方法），避免了为每个科目单独编写判断逻辑。

在数据转换环节，对于每个有效的成绩值，将其封装为包含学号、科目名称和成绩值的三元组，并通过专门的 insertSC 方法将这些三元组写入目标表。该方法构建参数化查询，依次绑定学号、科目名称和成绩值三个参数，完成单条数据的插入操作。这种设计使得列值检查与数据写入操作相互独立，当需要调整科目或添加新科目时，只需修改预先定义的科目数组即可。主循环中，通过 wasNull()方法精确过滤掉空值，确保只有有效的成绩数据被转换和存储，从而实现了从宽表结构到键值对结构的无损压缩转换。

其中 insertSC 方法的实现代码如下。

```
    public static int insertSC(Connection connection, int sno,
String colName, int colValue) {
        PreparedStatement pstmt = null;
        try {
```

```
            String sql = "INSERT INTO sc (sno, col_name, col_v
alue) VALUES (?, ?, ?)";
            pstmt = connection.prepareStatement(sql);
            pstmt.setInt(1, sno);
            pstmt.setString(2, colName);
            pstmt.setInt(3, colValue);
            return pstmt.executeUpdate();
        } catch (SQLException e) {
            e.printStackTrace();
            return 0;
        } finally {
            if (pstmt != null) {
                try {
                    pstmt.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            }
        }
    }
```

3 课程总结

本课程以 MySQL 为例，系统性地设计了 16 个实训任务，涵盖数据库对象管理、数据查询与处理、系统内核实验、数据库设计及 Java 应用开发等内容，我完成了实训 1-7，10-11，14-15，其中实训 5 和实训 14 跳过了部分关卡。重点完成了数据查询的有关实训。

主要工作方面，我主要学习并完成以下方面的实训。

(1) 数据库基础操作：学习并实现数据库，表，约束等的创建，修改，删除等操作，完成了实训 1-2 的全部关卡。

(2) 数据查询：基于金融场景数据库，从简到难，逐步学习和熟悉 SQL 查询语句，其中有 SELECT,WHERE,GROUP,LIMIT 这些理论课中学习过的语法，也有实训过程中题目所讲解的，比如和排名，时间等有关的函数。还有实训过程自己通过各种途径所了解和学习的知识，比如 CTE 表达式的使用，GROUP_CONCAT 函数的妙用。最终完成了实训 3-实训 7（其中实训 5 跳过了两个关卡）。

(3) 数据库设计方面：这方面我只进行了简单了解学习数据库设计的有关概念，最终实现将机票订票系统概念模型转换为 MySQL 表结构，建立符合业务规则的索引与约束，完成了实训 14 的第一个关卡。

(4) JDBC 应用开发：学习了 JDBC 的有关概念和使用方法，最终能够使用 JDBC 实现对数据库的简单操作，完成了实训 15 的全部关卡。

本次实训主要集中在数据查询的有关工作上，缺乏广度，对对 DBMS 原理，数据库设计方面，和内核有关方面的掌握情况有所欠缺，未来有时间会加强这些方面的学习。此外，个人认为实践课程在这些方面的引导讲解不如数据查询，JDBC 实训充分，学习难度较大，未来实训课程也可以在这方面做些改进，

附录

附录 1　2.6.3 事务与转账操作的实现代码

```
  public static boolean transferBalance(Connection connection,
                                String sourceCard,
                                String destCard,
                                double amount) {
        PreparedStatement sourceCheckStmt = null;
        PreparedStatement destCheckStmt = null;
        PreparedStatement updateSourceStmt = null;
        PreparedStatement updateDestStmt = null;
        ResultSet sourceRs = null;
        ResultSet destRs = null;

        try {
            // 开启事务
            connection.setAutoCommit(false);

            // 1. 检查转出账户是否存在且不是信用卡
            String sourceCheckSql = "SELECT b_type, b_balance
FROM bank_card WHERE b_number = ? FOR UPDATE";
            sourceCheckStmt = connection.prepareStatement(sour
ceCheckSql);
            sourceCheckStmt.setString(1, sourceCard);
            sourceRs = sourceCheckStmt.executeQuery();

            if (!sourceRs.next()) {
                return false; // 转出账户不存在
            }

            String sourceType = sourceRs.getString("b_type");
            double sourceBalance = sourceRs.getDouble("b_balan
ce");
```

```
            if ("信用卡".equals(sourceType)) {
                return false; // 转出账户是信用卡
            }

            // 2. 检查转入账户是否存在
            String destCheckSql = "SELECT b_type, b_balance FR
OM bank_card WHERE b_number = ? FOR UPDATE";
            destCheckStmt = connection.prepareStatement(destCh
eckSql);
            destCheckStmt.setString(1, destCard);
            destRs = destCheckStmt.executeQuery();

            if (!destRs.next()) {
                return false; // 转入账户不存在
            }

            String destType = destRs.getString("b_type");
            double destBalance = destRs.getDouble("b_balance")
;

            // 3. 检查转出账户余额是否充足
            if (sourceBalance < amount) {
                return false; // 余额不足
            }

            // 4. 执行转账
            // 更新转出账户（扣款）
            String updateSourceSql = "UPDATE bank_card SET b_b
alance = b_balance - ? WHERE b_number = ?";
            updateSourceStmt = connection.prepareStatement(upd
ateSourceSql);
            updateSourceStmt.setDouble(1, amount);
            updateSourceStmt.setString(2, sourceCard);
            updateSourceStmt.executeUpdate();
```

```
            // 更新转入账户
            String updateDestSql;
            if ("信用卡".equals(destType)) {
                // 信用卡还款：先补齐透支款项，剩余金额转为存款
                if (destBalance > 0) {
                    // 有透支，先还清欠款
                    double remainingAmount = destBalance - amo
unt;
                    if (remainingAmount < 0) {
                        // 还清欠款后还有余额，转为存款
                        updateDestSql = "UPDATE bank_card SET
b_balance = ? WHERE b_number = ?";
                        updateDestStmt = connection.prepareSta
tement(updateDestSql);
                        updateDestStmt.setDouble(1, remainingA
mount);
                        updateDestStmt.setString(2, destCard);
                    } else {
                        // 只还部分欠款
                        updateDestSql = "UPDATE bank_card SET
b_balance = b_balance - ? WHERE b_number = ?";
                        updateDestStmt = connection.prepareSta
tement(updateDestSql);
                        updateDestStmt.setDouble(1, amount);
                        updateDestStmt.setString(2, destCard);
                    }
                } else {
                    // 无透支，直接转为存款
                    updateDestSql = "UPDATE bank_card SET b_ba
lance = b_balance - ? WHERE b_number = ?";
                    updateDestStmt = connection.prepareStateme
nt(updateDestSql);
                    updateDestStmt.setDouble(1, amount);
                    updateDestStmt.setString(2, destCard);
                }
```

```
            } else {
                // 普通转账
                updateDestSql = "UPDATE bank_card SET b_balanc
e = b_balance + ? WHERE b_number = ?";
                updateDestStmt = connection.prepareStatement(u
pdateDestSql);
                updateDestStmt.setDouble(1, amount);
                updateDestStmt.setString(2, destCard);
            }
            updateDestStmt.executeUpdate();

            // 提交事务
            connection.commit();
            return true;

        } catch (SQLException e) {
            try {
                // 回滚事务
                connection.rollback();
            } catch (SQLException ex) {
                ex.printStackTrace();
            }
            return false;
        } finally {
            // 恢复自动提交
            try {
                connection.setAutoCommit(true);
            } catch (SQLException e) {
                e.printStackTrace();
            }

            // 关闭资源
            try {
                if (sourceRs != null) sourceRs.close();
                if (destRs != null) destRs.close();
```

```
                if (sourceCheckStmt != null) sourceCheckStmt.c
lose();
                if (destCheckStmt != null) destCheckStmt.close
();
                if (updateSourceStmt != null) updateSourceStmt.
close();
                if (updateDestStmt != null) updateDestStmt.clo
se();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
```