

**数据库系统原理实践报告**

|  |  |
| --- | --- |
| 专 业： | 计算机科学与技术 |
| 班 级： | 大数据2102班 |
| 学 号： | U202112313 |
| 姓 名： | 陈东平 |
| 指导教师： | 潘鹏 |

|  |  |
| --- | --- |
| 分数 |  |
| 教师签名 |  |

2023年5月31日

**教师评分页**

|  |  |
| --- | --- |
| 子目标 | 子目标评分 |
| 1 |  |
| 2 |  |
| 3 |  |
| 4 |  |
| 5 |  |
| 6 |  |
|  |  |

|  |  |
| --- | --- |
| 总分 |  |

目 录

[1 课程任务概述 1](#_Toc136394219)

[2 任务实施过程与分析 2](#_Toc136394220)

[2.1 数据查询(Select) 2](#_Toc136394221)

[2.2 存储过程与事务 10](#_Toc136394222)

[2.3触发器 17](#_Toc136394223)

[2.4 数据库应用开发(JAVA篇) 19](#_Toc136394224)

[2.5数据库设计与实现 23](#_Toc136394225)

[2.6 数据库的索引B+树实现 25](#_Toc136394226)

[3 课程总结 29](#_Toc136394227)

[附录 30](#_Toc136394228)

# 1 课程任务概述

"数据库系统原理实践"是一门专门为配合"数据库系统原理"课程而设计的实践课程，它强调理论与实践的紧密结合。在这门课程中，我们使用OpenGauss数据库管理系统，需要在头歌平台上完成课程组为我们系统设计的一系列实训任务，从而深入理解和掌握数据库系统的各个方面：

数据对象的管理与编程：我们将深入探讨数据库、表、索引、视图、约束、存储过程、函数、触发器、游标等数据对象的管理和编程。这些结构和部件是数据库中基础的组成部份，因此需要我们分门别类对每一个组成部份系统性的学习。

数据处理相关任务：我们将通过实践学习数据查询，数据插入、删除与修改等基本的数据处理任务。借此我们可以熟悉数据库的基本操作，并理解其背后的原理。这一部分和SQL语言高度相关，因此我们也会同时学习到OpenGauss下的SQL语句的编写以及相关函数的应用，这一部分是数据库学习中最基础的，也是最接近应用的一个部分，需要我们仔细学习牢牢把握。

数据库系统内核的实验：我们将研究数据库的安全性控制，完整性控制，恢复机制，以及并发控制机制等核心技术。这一部分是数据库的进阶部分，与前面的基础部分相辅相成，更好的帮我们熟悉数据库系统。

数据库应用系统的开发：我们将使用JAVA 1.8进行数据库应用系统的开发实践，掌握如何在实际应用中使用数据库进行数据的访问及处理等等。

数据库综合实验：我们将设计并实现自己的数据库，以深入理解数据库的工作原理。例如在实验中，我们将搭建飞机票、电影票等系统。不仅如此，我们还会进行B+树的实验，从而更加理解数据库的索引机制。

本课程依托于头歌实践教学平台进行，实验环境为Linux操作系统下的OpenGauss2.1；在数据库应用开发环节，我们将使用JAVA 1.8；在B+树索引编写部分，我们将使用C++来进行编写。

综上所述，我们会在这门课中学习到数据库的基础结构，SQL语法，相关的内核和实际应用，同时深入理解和掌握数据库系统的原理和应用，为未来的学习和工作打下坚实的基础。

# 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任务书中的2.1～2.14所有的任务，下面将重点针对其中的2.3、2.6、2.7、2.11、2.13和2.14任务阐述其完成过程中的具体工作。

## 2.1 数据查询(Select)

本实训采用的是某银行的一个金融场景应用的模拟数据库，数据库中有client(客户表)、bank\_card(银行卡)、finances\_product(理财产品表)、insurance(保险表)、fund(基金表)、property(资产表)。在本小节中，我们需要用一句SELECT语句来完成查询任务，需要我们掌握SELECT语句、内置函数、排序等功能的调用，十分考验综合能力。在本章节我完成了所有的任务，获得满分。

### 2.1.1 任务6:商品收益的众数

任务要求：请用一条SQL语句完成以下查询任务：查询资产表中所有资产记录里商品收益的众数和它出现的次数，出现的次数命名为presence。众数是一组数据中出现次数最多的数值，有时众数在一组数中会有好几个。

相关知识：子查询；分组统计与COUNT()函数；ALL关键词或者MAX()函数;给表达式列命名。

过程分析：在这个任务中，我们要进行两次查询。在第一次查询中，我们可以获得每个收益的金额在所有投资中的次数，这个时候数据是根据pro\_income来归组的，同样金额的收入在同一组，使用Count函数就可以获得这一组里面的个数。之后，我们使用Max函数进一步获得我们在上一步得到的数量中的最大的数量，也就是所谓的众数。因为这个任务是把所有的众数都列出来，因此我们先对所有的数据根据pro\_income归组，然后组中的数量等于我们之前查询到的最大数量的就是众数了。最后我们选择这个收益的金额和数量（重命名为presence）输出即可，我们使用了子查询，分组统计和count，max函数来提高查询效率和降低查询难度。

|  |
| --- |
| SELECT pro\_income,COUNT(\*) AS presence FROM property  GROUP BY pro\_income  HAVING COUNT(\*) = (SELECT MAX(cnt) FROM (SELECT COUNT(\*) as cnt FROM property GROUP BY pro\_income)); |

源代码展示

### 2.1.2 任务10: 投资总收益前三名的客户

任务要求：使用一条SQL语句完成以下查询任务：查询当前总的可用资产收益(被冻结的资产除外)前三名的客户的名称、身份证号及其总收益，按收益降序输出，总收益命名为total\_income。不需要考虑并列排名情形。输出：c\_id,c\_name,c\_id\_card,total\_income。

注意事项：1. 本题目不需要考虑并列排名情形，这意味着前5名为10000，9900，9900，8000，8000时，也只需要给出10000，9900，9900即可。2. 背景介绍中已给出，pro\_status有”可用”或”冻结”两种取值，pro\_status=”冻结”表示这条资产记录处于冻结状态。

相关知识：多表连接；分组统计；衍生表(派生表)；子查询；top n查询。

过程分析：在这个查询中，分为外部和内部查询。在外部查询中，它从客户表中选择客户的姓名、身份证号以及他们的总收入。总收入是通过一个子查询计算得出的，这个子查询从属性表中选择所有状态为'可用'且与当前客户关联的收入记录，然后计算他们的总和。在外查询的WHERE子句中，它限制了查询结果只包括收入最高的前三名客户，这也是通过一个子查询实现的，与上面的子查询类似，最后将查询结果按总收入降序排序，这样收入最高的客户就会排在最前面。

查询语句优化：由于使用两次子查询会降低查询的效率，经过分析我发现可以把两个子查询合并成一个，也就是先进行property查询之后和client并起来即可，在下面的代码可以发现我们使用Join把我们子查询和外部查询连接起来了，这样子只需要两次查询就可以完成任务。

|  |
| --- |
| SELECT c\_name, c\_id\_card, (SELECT SUM(pro\_income) FROM property WHERE pro\_status = '可用' AND client.c\_id = property.pro\_c\_id) AS total\_income FROM client WHERE c\_id IN (SELECT pro\_c\_id FROM property WHERE pro\_status = '可用' GROUP BY pro\_c\_id ORDER BY SUM(pro\_income) DESC LIMIT 3) ORDER BY total\_income DESC; |

原始源代码展示

|  |
| --- |
| SELECT c.c\_name, c.c\_id\_card, p.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 ORDER BY total\_income DESC LIMIT 3) p ON c.c\_id = p.pro\_c\_id ORDER BY p.total\_income DESC; |

修改后的代码展示

### 2.1.3 任务12： 客户理财、保险与基金投资总额

任务要求：综合客户表(client)、资产表(property)、理财产品表(finances\_product)、保险表(insurance)和基金表(fund)，列出客户的名称、身份证号以及投资总金额（即投资本金，每笔投资金额=商品数量\*该产品每份金额)，投资总金额命名为total\_amount。查询结果按总金额降序排序。查询语句输出为：c\_name, c\_id\_card, total\_amount

注意事项：1.投资金额按类型需查询不同的表，投资总金额是客户购买的各类(理财,保险,基金)资产投资金额的总和。2. 不考虑资产是否可用，可用的和冻结的都考虑在内。3.需要列出所有客户的投资总金额，如果他/她没有购买过任何资产，那么他/她的投资总金额为0。

相关知识：分表合并；分组统计；对结果排序。

过程分析：查询涉及多个表和多个子查询。主体部分选择了客户的姓名和身份证号，并计算了总金额作为total\_amount。总金额的计算涉及多个子查询。第一个子查询计算客户在金融产品表（finances\_product）中的金额总和，通过关联属性表（property）确保仅计算与当前客户相关的金融产品金额。第二个子查询计算客户在保险表（insurance）中的金额总和，同样通过关联属性表（property）限定相关性。第三个子查询计算客户在基金表（fund）中的金额总和，也通过关联属性表（property）确保关联性。这里最重要的是使用COALESCE函数，它是用于处理子查询返回的空值的。因为sum可能在没有数据的时候返回null，但是null不能用在加法中，因此要用COALESCE函数将null转化为0，这样子就可以参与加法了。最后，查询结果按照总金额降序排序，以使金额最高的客户排在前面。

|  |
| --- |
| SELECT c\_name, c\_id\_card, ( COALESCE((SELECT SUM(pro\_quantity \* p\_amount) FROM finances\_product,property WHERE pro\_type = 1 AND pro\_pif\_id = p\_id AND pro\_c\_id = c\_id),0) + COALESCE((SELECT SUM(pro\_quantity \* i\_amount) FROM insurance,property WHERE pro\_type = 2 AND pro\_pif\_id = i\_id AND pro\_c\_id = c\_id),0) + COALESCE((SELECT SUM(pro\_quantity \* f\_amount) FROM fund,property WHERE pro\_type = 3 AND pro\_pif\_id = f\_id AND pro\_c\_id = c\_id),0)) AS total\_amount FROM client ORDER BY total\_amount DESC; |

源代码展示

### 2.1.4 任务15：基金收益两种方式排名

任务要求：使用一句sql语句，查询资产表中客户编号，客户基金投资总收益,基金投资总收益的排名(从高到低排名)。总收益相同时名次亦相同(即并列名次)。总收益命名为total\_revenue, 名次命名为rank。第一条SQL语句实现全局名次不连续的排名，第二条SQL语句实现全局名次连续的排名。不管哪种方式排名，收益相同时,客户编号小的排在前。查询语句输出为：pro\_c\_id,total\_revenue,rank。

注意事项：1.排名说明详见平台相关知识部分。2.实践前面介绍过，当数据库/表/列与SQL保留字同名时，需要用`（大键盘数字键1前一个按键，英文输入模式下）框起加以区分，例如sql关键字select，如果需要建立一个名为select的表，语句应该为 create table `select`;

相关知识：分组统计，对分组统计的结果进行排序；两种排名次需求及对应策略。

程序设计：查询主体部分选择了属性的客户ID，并使用COALESCE函数处理总收入的求和结果。COALESCE函数用于处理求和结果中可能存在的空值，将其替换为0。RANK()函数与窗口函数OVER一起使用，根据总收入的降序对查询结果进行排名。每个客户的排名作为rank列返回。之后使用查询的WHERE子句筛选出属性类型为3的属性，即pro\_type = 3。查询结果按照总收入的降序和客户ID的升序进行排序。首先，根据总收入的降序排列，使总收入最高的属性排在前面；在总收入相同时，按照客户ID的升序排列。在这里的重点是学会使用RANK和DENSE\_RANK，前者是同数同名次，总排名不连续；后者是

同数同名次，总排名连续的，只要学会使用这个，这个任务就很好解决。

|  |
| --- |
| SELECT pro\_c\_id, (COALESCE(SUM(pro\_income),0)) AS total\_revenue, RANK() OVER(ORDER BY total\_revenue DESC) AS rank FROM property WHERE pro\_type = 3 GROUP BY pro\_c\_id ORDER BY total\_revenue DESC,pro\_c\_id ASC; |

源代码展示

### 2.1.5 任务17：购买基金的高峰期

任务要求：使用一句sql语句，查询2022年2月购买基金的高峰期。至少连续三个交易日，所有投资者购买基金的总金额每天超过100万(含)，则称这段连续交易日为投资者购买基金的高峰期。只有交易日才能购买基金,但不能保证每个交易日都有投资者购买基金。2022年春节假期之后的第1个交易日为2月7日,周六和周日是非交易日，其余均为交易日。请列出高峰时段的日期和当日基金的总购买金额，按日期顺序排序。总购买金额命名为total\_amount。查询的输出为：pro\_purchase\_time,total\_amount

说明：1.高峰期内每一天的金额都超过100万；2.注意高峰期指连续的交易日，而不是连续的日子，这意味着周四、周五、周一有可能构成高峰日。

相关知识：分组统计;子查询;复杂行筛选条件与分组统计后筛选条件；日期函数使用。

程序设计：首先，第一个WITH子句创建了名为daily\_amount的临时视图。这个视图从property和fund两个表格中通过JOIN操作连接数据，连接条件是property的pro\_pif\_id等于fund的f\_id。这个视图对pro\_purchase\_time进行分组并计算每天的购买总额，该总额是每一笔购买的商品数量(pro\_quantity)与f\_amount的乘积的总和。这个视图只包括pro\_type等于3，且pro\_purchase\_time在2022年2月1日至2022年2月28日期间的记录。由于我们的时间数据都是取自于pro\_purchase\_time中的，而交易都是发生在交易日的，因此在这一步可以保证在逻辑上连续的日期就是相邻的交易日，满足题目的要求。

然后，第二个WITH子句创建了另一个临时视图，名为three\_day\_amount。这个视图从daily\_amount视图中选取数据，并计算了连续三天的购买总额（three\_days）。这个计算通过当前天的购买总额(amount)与前两天的购买总额之和得到，如果前两天没有数据，则默认为0。此外，这个视图还提供了下一天的购买总额(next\_day\_amount)以及下两天的购买总额，因为对于某一天，连续三天的情况有三种，例如求第三天的连续三天情况，我们需要的数据有123，234和345天的购买总额。

最后，主查询从三天总购买量视图(three\_day\_amount)和日购买量视图(daily\_amount)中连接数据，连接条件是购买时间pro\_purchase\_time相同。该查询选取了购买时间和当天的购买总额，但只包括满足以下条件的记录：当天的购买总额(amount)大于或等于1000000，且连续三天的购买总额(three\_days)大于或等于3000000。最后，根据购买时间进行排序就得到最后的结果了。

程序测试结果见图2.3，流程图见图2.1和2.2。

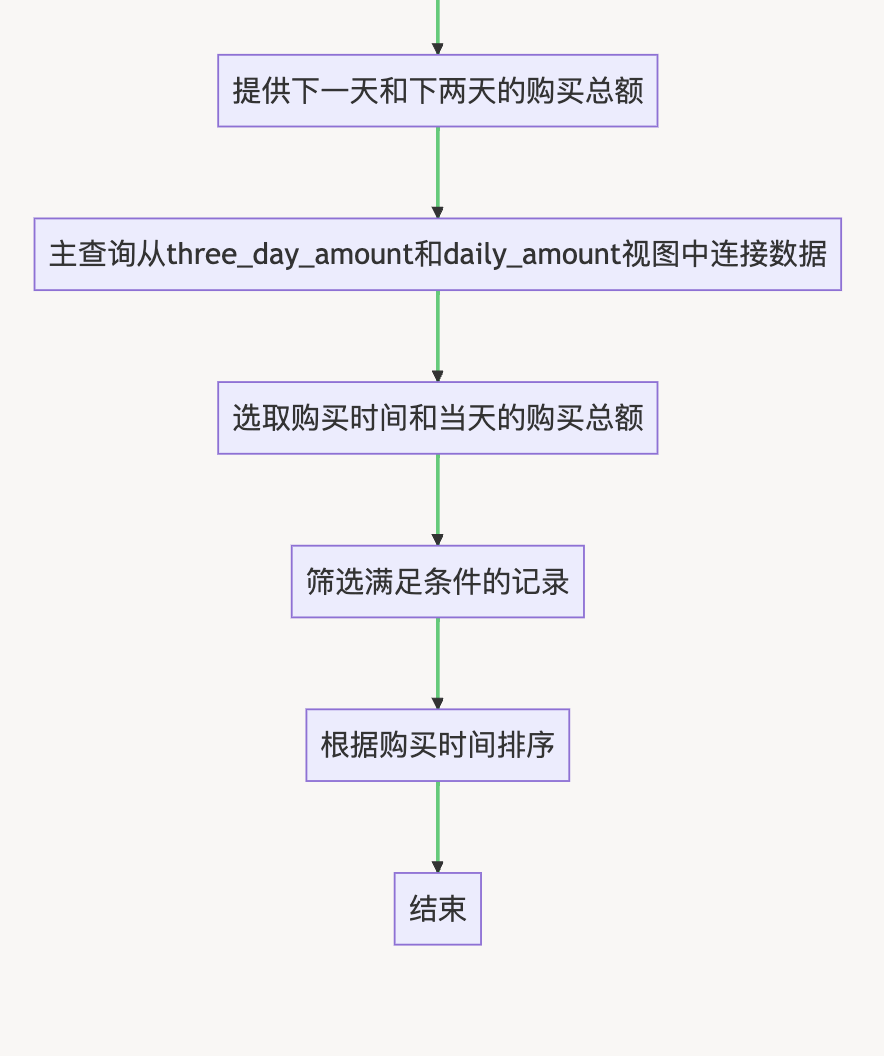
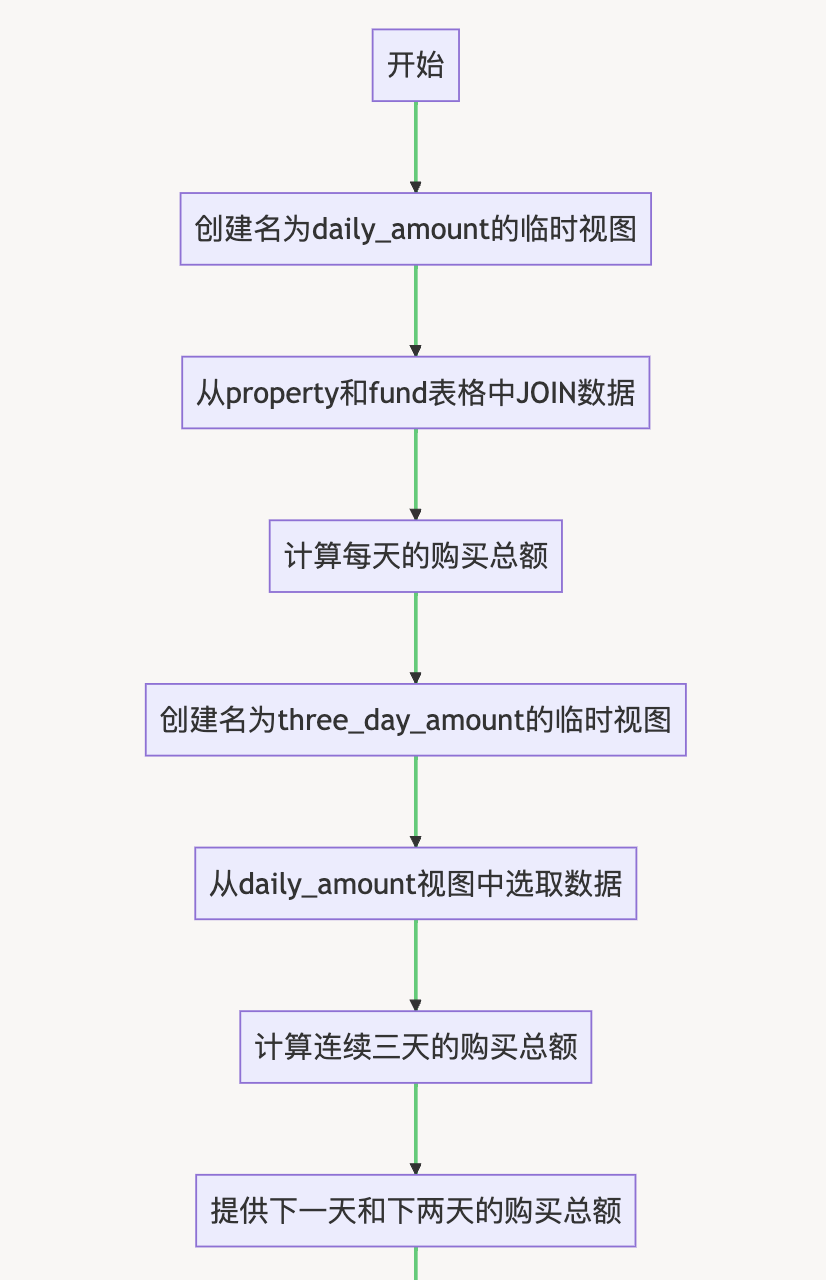


图2.1、图2.1: 购买基金的高峰期排名流程图

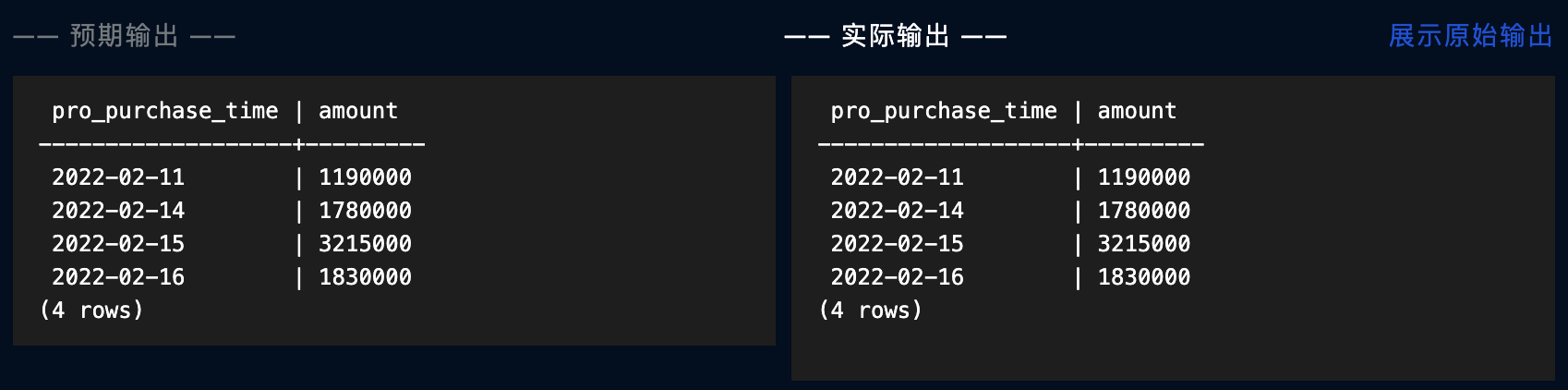


图2.3: 购买基金的高峰期排名测试输出结果

### 2.1.6 任务19：以日历表格式显示每日基金购买总金额

任务要求：使用一句sql语句，以日历表格式列出2022年2月每周每个交易日基金购买总金额，输出格式如表2-1：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| week\_of\_trading | Monday | Tuesday | Wednesday | Thursday | Friday |
| 1 |  |  |  |  |  |
| 2 |  |  |  |  |  |
| 3 |  |  |  |  |  |
| 4 |  |  |  |  |  |

表2.1：日历表格式示例

其中，列表中第1列为周次，2022年2月7日(星期一)为当月的第1个交易日，这一周记为第1周次。注意显示结果并不需要画表格线，只需按这个格式输出结果即可。

注意事项：2月7日星期一做为日历表的开始，2月28日星期一是2月所有交易日的结束。之后的日期既不应该用3月日期填补，也不应该为这些日期写0，因为它们是不存在的日期，在这些位置输出NULL(思考SQL查询输出中什么情况下会出现空值)

相关知识：分组统计；日期函数的应用；if或case函数的用法；有规律的行列转换。

程序设计：这个SQL查询从property和fund两个表格中通过JOIN操作连接数据，连接条件是property的pro\_pif\_id等于fund的f\_id。查询的主要目的是计算2022年2月每个交易周（pro\_purchase\_time的每周部分作为周交易索引）的每个工作日的购买总额，前提是pro\_type等于3。这里的周交易索引是基于pro\_purchase\_time的周部分减5来计算的，因为二月第一个交易日的那一周相当于是那一年的第六周。查询首先选取pro\_purchase\_time的周部分，以此作为每个交易周的索引（week\_of\_trading）。然后，通过分别判断pro\_purchase\_time转换为时间戳后的星期数（DOW，Day Of Week），计算每个工作日的购买总额。这是通过对pro\_quantity与f\_amount的乘积进行求和得到的，如果对应的工作日没有记录，则结果为0。最后，如果某一天的购买总额为0，则使用NULLIF函数将其转换为NULL。最后的查询结果按照周交易索引（week\_of\_trading）进行排序，这样子就可以生成一张类似于日历的表格，展示了2022年2月每个交易周（周一至周五）的每天购买总额。具体来说，这个查询的输出形式是：week\_of\_trading表示交易周的序号，monday、tuesday、wendnesday、thursday和friday列分别代表该周的周一到周五的购买总额，获取了2022年2月每个交易周的每个工作日的购买总额，这样的数据可能对于理解购买模式、识别特定日子或周的购买趋势以及预测未来的购买行为都非常有用。

测试结果见图2.4，流程图见图2.5。



图2.4: 以日历表格式显示每日基金购买总金额测试输出结果

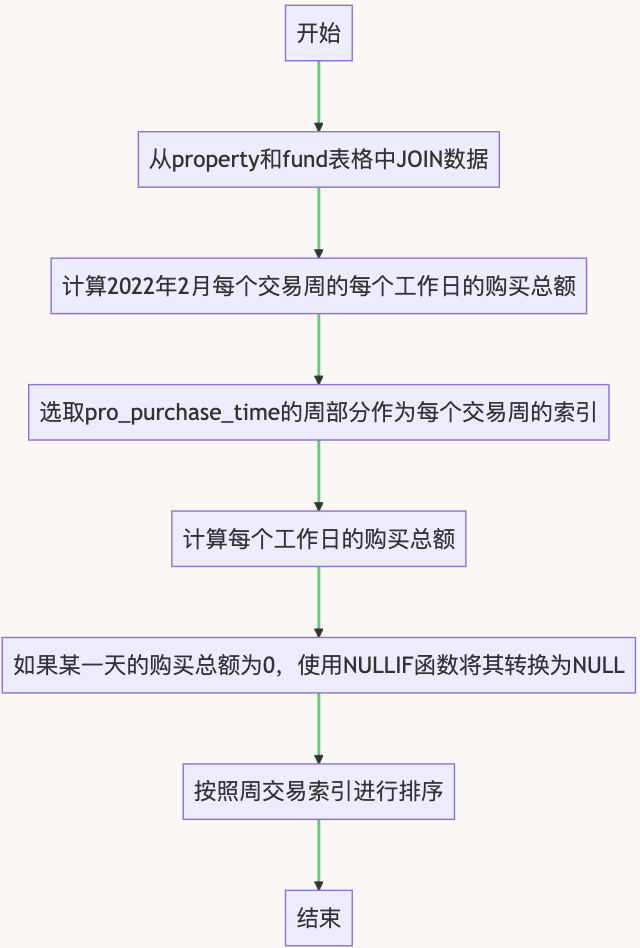


图2.5：以日历表格式显示每日基金购买总金额程序流程图

### 2.1.7 任务25：查找相似的理财客户

任务要求：在某些推荐方法中，需要查找某位客户在理财行为上相似的其他客户，不妨设其定义为：对于A客户，其购买的理财产品集合为{P}，另所有买过{P}中至少一款产品的其他客户集合为{B}，则{B}中每位用户购买的{P}中产品的数量为他与A客户的相似度值。将{B}中客户按照相似度值降序排列，得到A客户的相同相似度值则按照客户编号升序排列，这样取前两位客户即为A客户的相似理财客户列表。使用一句sql语句，查询每位客户的相似度排名值小于3的相似客户列表，以及该每位客户和他的每位相似客户的共同持有的理财产品数、相似度排名值。

注意事项：结果输出要求：要求结果先按照左边客户编号升序排列，同一个客户的相似客户则按照该客户的相似客户列表本身的顺序排列。

相关知识：嵌套查询；rank() over(partition by ...order by)的使用。

程序设计：基于相似客户寻找的方法，我发现至少要经过三次查询，分别查询该用户购买的理财产品，购买这些理财产品的相似人群，最后一次查询将这些人归组并且排序输出。首先，查询在内部SELECT语句中，对从property表中取出的所有pro\_c\_id进行配对，只要这两个pro\_c\_id不同，并且C表的pro\_type等于1。然后，查询在A表（也是property表的别名）中寻找B表中pro\_c\_id所关联的所有唯一的pro\_pif\_id（A表中的pro\_type也必须为1）。如果C表的pro\_pif\_id在这些结果中，那么这个配对就被选中。对于每个被选中的配对，我们查询计算了它们共享的唯一pro\_pif\_id的数量（命名为common）。然后，查询对于每个B表的pro\_c\_id，使用rank() over(partition by … order by)的格式，这样子可以对每一个用户的相似人群分别进行排序而不是排到一起去。这个排名命名为crank，排名按照common的值降序、C表的pro\_c\_id升序进行，也就是拥有类似理财产品越多且c\_id越小的用户排的越前。最后，查询把内部SELECT语句的结果包装在一个外部SELECT语句中，以便对crank进行过滤。结果只包含了那些crank值小于3的记录。

查询语句优化：我用JOIN操作替代了原本的子查询。这样做的优点是可以利用数据库的查询优化器，执行更有效的数据检索。JOIN操作通过连接两个表的字段来组合数据，从而避免了子查询所需的额外处理，减少了一次子查询，提高了查询的速度。

|  |
| --- |
| SELECT pac,pbc,common,crank FROM (SELECT B.pro\_c\_id AS pac, C.pro\_c\_id AS pbc ,COUNT(DISTINCT C.pro\_pif\_id) AS common,(RANK() OVER(partition by B.pro\_c\_id order by COUNT(DISTINCT C.pro\_pif\_id) DESC,C.pro\_c\_id ASC))AS crank FROM property AS B,property AS C WHERE B.pro\_c\_id != C.pro\_c\_id AND C.pro\_type = 1 AND C.pro\_pif\_id in (SELECT DISTINCT pro\_pif\_id FROM property AS A WHERE A.pro\_c\_id = B.pro\_c\_id AND A.pro\_type = 1) GROUP BY B.pro\_c\_id,C.pro\_c\_id ORDER BY B.pro\_c\_id ASC, crank ASC) WHERE crank < 3; |

优化前sql语句

|  |
| --- |
| SELECT pac, pbc, common, crank FROM (SELECT B.pro\_c\_id AS pac, C.pro\_c\_id AS pbc, COUNT(DISTINCT C.pro\_pif\_id) AS common, RANK() OVER(PARTITION BY B.pro\_c\_id ORDER BY COUNT(DISTINCT C.pro\_pif\_id) DESC, C.pro\_c\_id ASC) AS crank FROM property AS B JOIN property AS C ON B.pro\_c\_id != C.pro\_c\_id AND C.pro\_type = 1 JOIN property AS A ON A.pro\_c\_id = B.pro\_c\_id AND A.pro\_type = 1 AND C.pro\_pif\_id = A.pro\_pif\_id GROUP BY B.pro\_c\_id, C.pro\_c\_id) subquery WHERE crank < 3 ORDER BY pac ASC, crank ASC; |

优化后sql语句

## 2.2 存储过程与事务

存储过程（Stored Procedure）是一种在数据库中存储复杂程序，以便外部程序调用的一种数据库对象。存储过程是为了完成特定功能的 SQL 语句集，经编译创建并保存在数据库中，用户可通过指定存储过程的名字并给定参数（需要时）来调用执行。简单的说存储过程就是具有名字的一段代码，用来完成一个特定的功能。在本部分，我完成了所有的练习并取得满分。

### 2.2.1 任务1：使用流程控制语句的存储过程

任务要求：创建存储过程sp\_fibonacci(in m int)，向表fibonacci插入斐波拉契数列的前m项，及其对应的斐波拉契数。fibonacci表初始值为一张空表。请保证存储过程可以多次运行而不出错。

注意事项：数据库中有表fibonacci，用来储存斐波拉契数列的前n项，如表2.2所示：（测试使用m范围为1<=m<=92）

| **列名** | **类型** | **说明** |
| --- | --- | --- |
| n | int | 斐波拉契数列的第n项,主码 |
| fibn | bigint | 第n项的值 |

表2.2：斐波拉契数列表

相关知识：变量的定义与赋值；复合语句与流程控制语句；存储过程的定义；存储过程的创建和调用；存储过程的查询和删除。

程序设计：这些语句创建了一个名为sp\_fibonacci的存储过程，其目的是计算斐波那契数列的前m个项，并将这些项插入到名为fibonacci的表中。斐波那契数列是这样的数列：每一项都是前两项的和，其中第一项和第二项分别为0和1。在这个过程中，我首先声明了四个整数变量a、b、fib和i，其中a和b是用来计算斐波那契数列的，fib用来保存计算结果，i用来追踪已计算的项数。这里要考虑特殊情况，也就是m为0或者1的情况，我这里采用了如下的设计：如果输入参数m比0大，那么就会将0（斐波那契数列的第一项）插入到fibonacci表中。如果m比1大，就会将1（斐波那契数列的第二项）插入到fibonacci表中。然后使用一个循环，计算并插入其余的斐波那契数列项。在每次循环中，它都会更新a和b的值，将当前的斐波那契数插入到fibonacci表中，然后i的值增加1，以便在下次循环中计算下一个斐波那契数。

在完成任务的过程中，我学会了在OpenGauss下如何创建储存过程，包括使用DECLARE完成变量的定义，变量在语句中的赋值，IF ELSE语句的应用以及WHILE语句的应用，这些都拓展了我对数据库的狭隘认知，让我知道数据库其实可以完成很多各种各样的任务，而并不是SELECT那些那么简单。

输出的测试图见图2.6和图2.7，程序设计的流程图见图2.8。

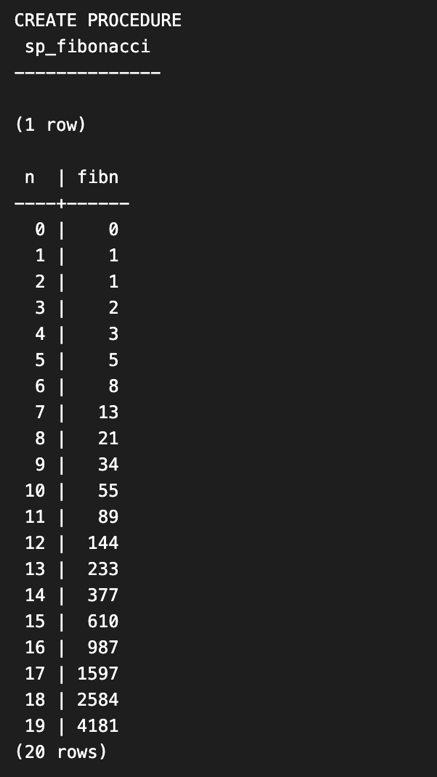
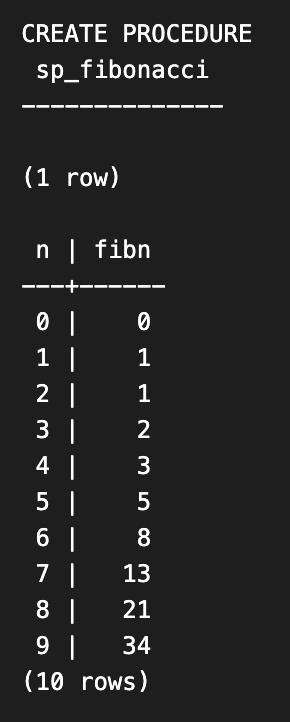
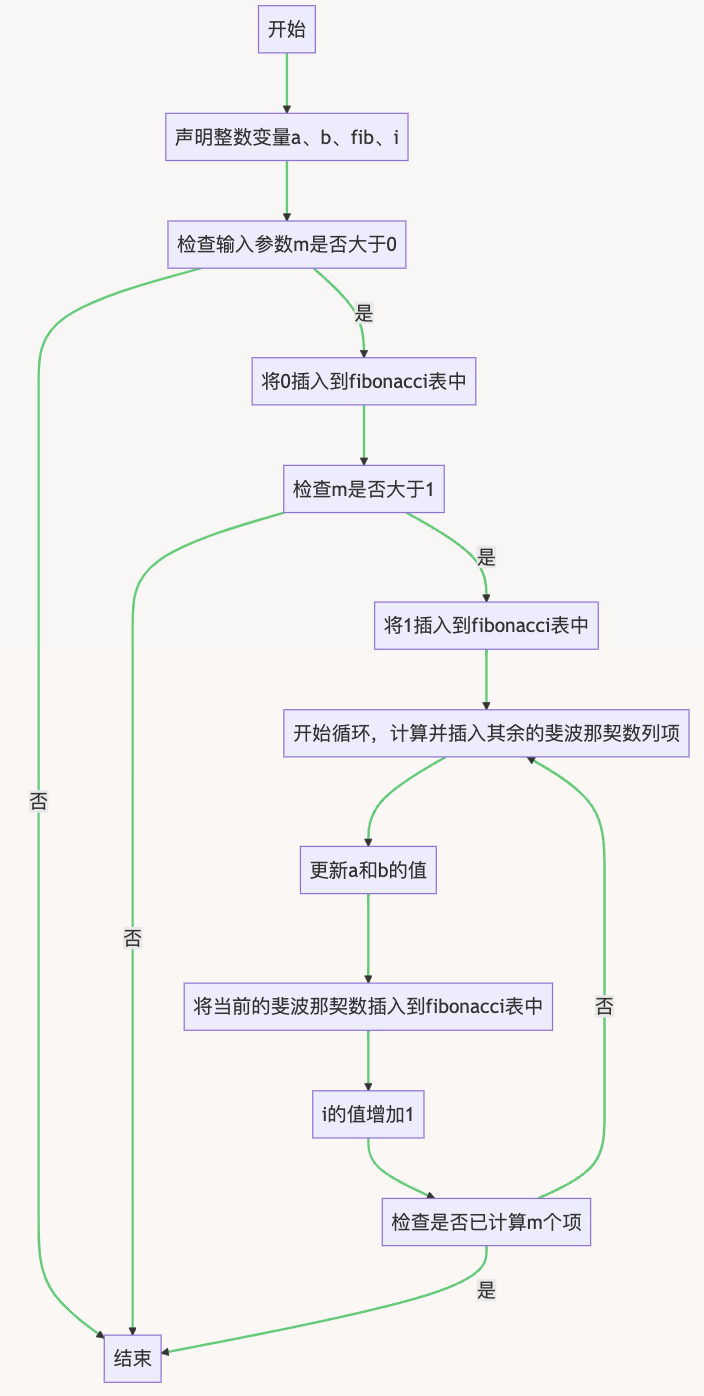
 

图2.6、图2.7: 流程控制过程的测试结果 图2.8: 流程控制的流程图

### 2.2.2 任务2：使用游标的存储过程

任务要求：医院的某科室有科室主任1名(亦为医生)，医生若干(至少2名，不含主任)，护士若干(至少4人)，现在需要编写一存储过程，自动安排某个连续期间的大夜班(即每天00:00-8:00时间段)的值班表，排班规则为：

1. 每个夜班安排1名医生，2名护士；
2. 值班顺序依工号顺序循环轮流安排(即排至最后1名后再从第1名接着排)；
3. 科室主任参与轮值夜班，但不安排周末(星期六和星期天)的夜班，当周末轮至科主任时，主任的夜班调至周一，由排在主任后面的医生依次递补值周末的夜班。

存储过程的名字为sp\_night\_shift\_arrange,它带两个输入参数：start\_date, end\_date，分别指排班的起始时间和结束时间。排班结果直接写入表night\_shift\_schedule。

注意事项：不用考虑其它信息(比如科室之类的)，在生产环境中，只需在where短语中施加条件限制即可明确选出所需科室的医护人员。这里，且把表中全部人员视为该科室人员。

相关知识：存储过程的定义(上一关已介绍);游标的定义和相关的语句使用，例如Open，Fetch，Next等

程序设计：在定义阶段，我们首先定义了两个游标，doctor\_cursor和nurse\_cursor，分别用于从employee表中选取医生和护士的名字。医生包括主任医生（e\_type=1）和普通医生（e\_type=2）。这些人员按照他们的编号（e\_id）进行排序，方便后续游标的遍历。之后定义了一个日期变量currentdate，初始值为输入的开始日期，是用来遍历题目给定的日期区间的。同时定义了三个字符变量，doctor\_name，nurse1\_name，nurse2\_name，用于存储当前日期的值班医生和两个值班护士的名字。

之后，程序在开始日期和结束日期之间进行循环。在每次循环中，首先检查是否需要主任医生值班（zhuguan=TRUE），并且当前日期是星期一（EXTRACT(DOW FROM currentdate) = 1）。如果满足这两个条件，那么游标将不进行遍历，直接让主任医生值班。Zhuguan默认是FALSE，只有在周六日排班到他的时候才会变成TRUE，相当于就是让他在周一补值班。

如果上述的IF语句不成立，就从doctor\_cursor游标中获取下一个医生的名字。如果没有找到（即所有医生都已经被选取过），那么就关闭再重新打开doctor\_cursor游标，这样子游标回到开头，并获取第一个医生的名字。对于两个值班护士，也是同样的逻辑。从nurse\_cursor游标中获取下一个护士的名字，如果没有找到，那么就关闭再重新打开nurse\_cursor游标，并获取第一个护士的名字。

如果当前的值班医生是主任医生，并且当前日期是周末（星期六或星期日），那么主任医生将不需要继续值班，而是调到周一补值班，把zhuguan变为TRUE，然后从doctor\_cursor游标中获取下一个医生的名字，如果游标到头了就重新关闭再打开游标就好。

最后，将当前日期、值班医生的名字、两个值班护士的名字插入到night\_shift\_schedule表中。然后将当前日期加1，进入下一次循环。

在结束日期之后，关闭两个游标。由于代码太长，因此我放在目录，这里只放测试的图片，见图2.9。我还画了相关的流程图，见图2.10。



图2.9: 使用游标的存储过程的测试结果

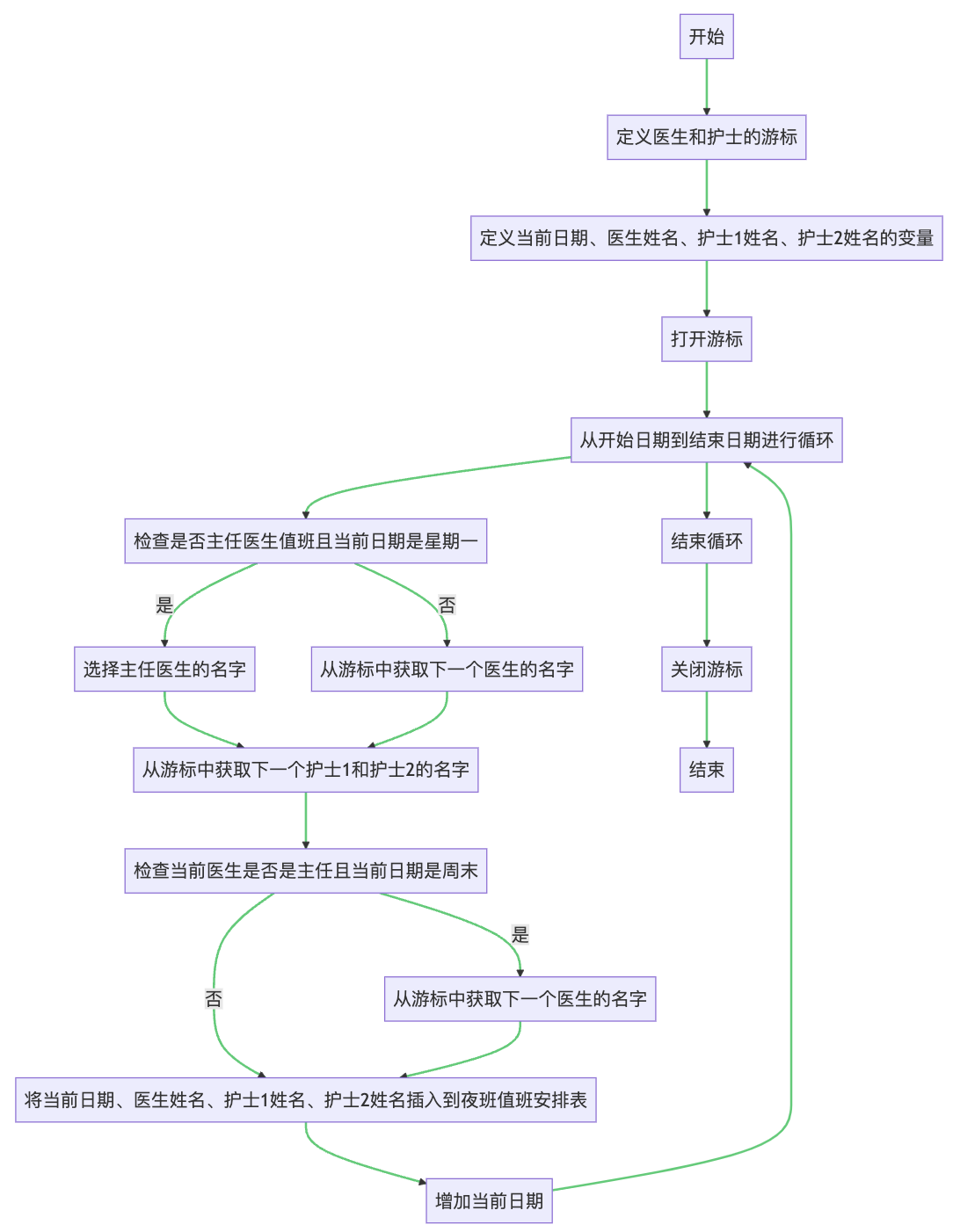


图2.10: 使用游标的存储过程的流程图

### 2.2.3 任务3：使用事务的存储过程

任务要求：在金融应用场景数据库中，编程实现一个转账操作的存储过程sp\_transfer，实现从一个帐户向另一个帐户转账。该过程有5个输入参数：applicant\_id 付款人编号；source\_card\_id 付款卡号；receiver\_card\_id 收款人编号；dest\_card\_id 收款卡号；amount 转账金额；还有1个整型输出参数：return\_code 1：正常转账；0:转账不成功。

转账操作涉及对表bank\_card的操作(在生产环境中，至少还要记录转账操作本身相关的信息至转账表，在实验环境中没有设计这样的表，从略；另外，生产环境中，当银行卡被冻结，或被卡主挂失后，都不能进行转账，在实验环境中，没有设计相应的字段 ，故也从略)。

注意事项：仅当转款人是转出卡的持有人时，才可转出；仅当收款人是收款卡的持有人时，才可转入；储蓄卡之间可以相互转账；允许储蓄卡向信用卡转账，称为信用卡还款(允许替它人还款)，还款可以超过信用卡余额，此时，信用卡余额为负数；信用卡不能向储蓄卡转账；转账金额不能超过储蓄卡余额；

相关知识：事务的定义和应用；存储过程的定义和应用；IF ELSIF ELSE格式的使用。

程序设计：首先，定义一个输出参数return\_code，并将其初始值设为0。这个参数用于表示转账操作的结果。如果转账成功，return\_code将被设置为1；如果转账失败，return\_code将保持为0。

接下来，进行一系列的检查以确保转账操作的合法性。首先，检查输入的源银行卡号source\_card\_id是否属于申请人applicant\_id，这是通过在bank\_card表中查询source\_card\_id对应的记录，并检查其b\_c\_id字段是否等于applicant\_id来实现的。然后，检查输入的目标银行卡号dest\_card\_id是否属于接收人receiver\_id，这也是通过在bank\_card表中进行查询来实现的。最后，检查源银行卡的余额是否大于等于转账金额amount，这是通过查询source\_card\_id对应的记录，并检查其b\_balance字段是否大于等于amount来实现的。

如果上述所有检查都通过，那么就会进一步检查源银行卡和目标银行卡的类型。这是通过查询source\_card\_id和dest\_card\_id对应的记录，并检查其b\_type字段来实现的。

在这里继续分为两种情况，如果源银行卡和目标银行卡都是储蓄卡，那么就会进行转账操作。首先，从源银行卡的余额中减去转账金额，这是通过更新source\_card\_id对应的记录，并将其b\_balance字段减去amount来实现的。然后，将转账金额加到目标银行卡的余额中，这是通过更新dest\_card\_id对应的记录，并将其b\_balance字段加上amount来实现的。如果源银行卡是储蓄卡，而目标银行卡是信用卡，那么也会进行转账操作。首先，从源银行卡的余额中减去转账金额，这和上述操作相同。然后，将转账金额从目标银行卡的已透支金额中减去，这是通过更新dest\_card\_id对应的记录，并将其b\_balance字段减去amount来实现的。

在转账操作完成后，将return\_code设置为1，表示转账成功，然后返回是否转账成功。在失败的情况下，比如源银行卡不属于申请人，或者目标银行卡不属于接收人，或者源银行卡的余额小于转账金额，那么存储过程将结束，return\_code保持为0，表示转账失败。

总的来说，程序设计考虑了多种转账操作的各种可能情况，包括源银行卡和目标银行卡的所有者是否正确，源银行卡的余额是否足够，以及源银行卡和目标银行卡的类型是否符合转账规则。这些检查都是为了确保转账操作的合法性和正确性。此外还考虑了储蓄卡和信用卡的不同点，也就是，对于储蓄卡，余额表示的是卡上的存款金额，因此在转账时，需要从源银行卡的余额中减去转账金额，然后将转账金额加到目标银行卡的余额中。而对于信用卡，余额表示的是已透支的金额，因此在转账时，需要从源银行卡的余额中减去转账金额，然后将转账金额从目标银行卡的已透支金额中减去。

对于程序的测试结果可以见图2.11，流程图见图2.12。

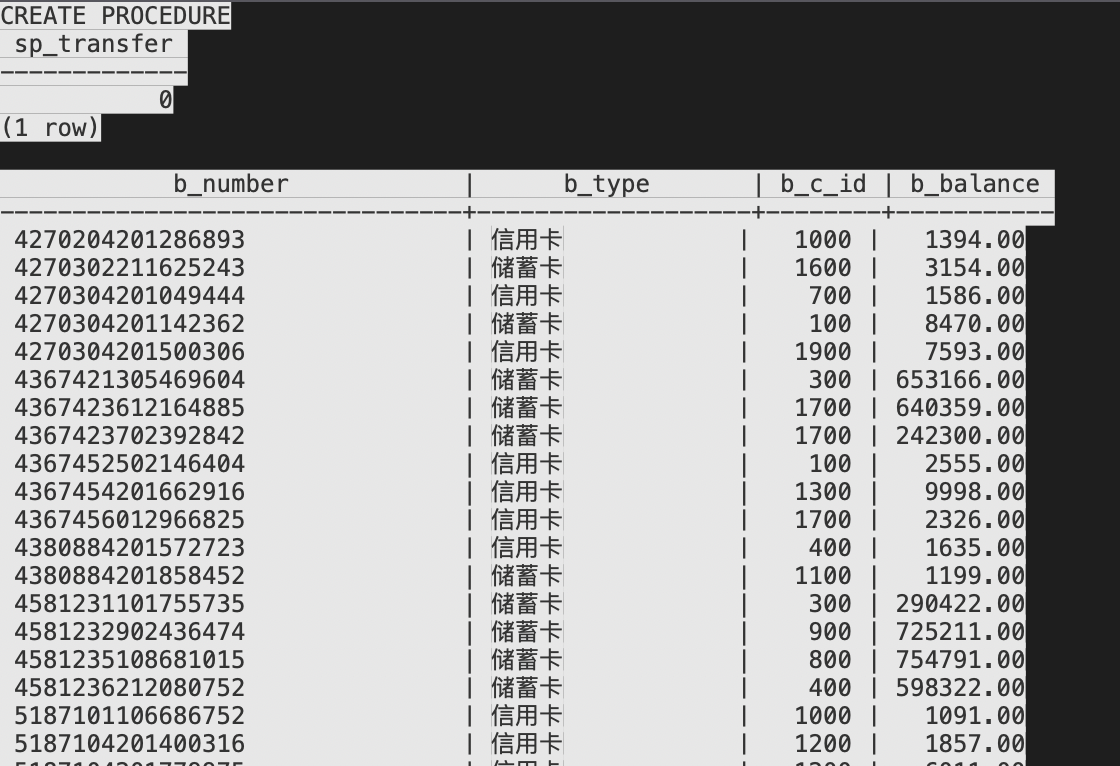


图2.11：使用事务的存储过程的测试结果

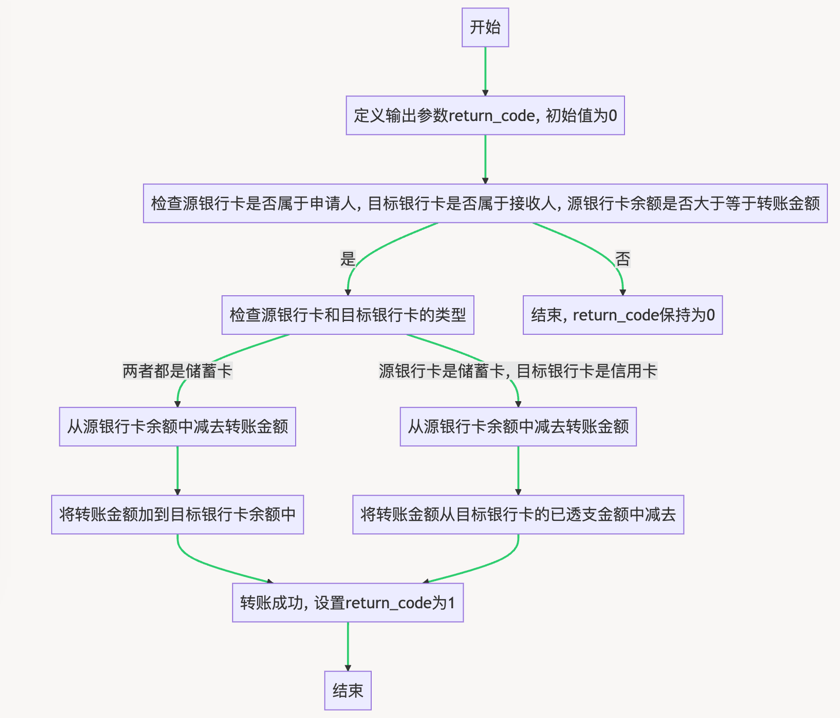


图2.12：使用事务的存储过程的流程图

## 2.3触发器

触发器是与某个表绑定的命名存储对象，与存储过程一样，它由一组语句组成，当这个表上发生某个操作(insert,delete,update)时，触发器被触发执行。当primary key,foreigh key, check等约束都无法实现某个复杂的业务规则时，可以考虑用触发器来实现。在这一部分我们学习了如何创建触发器，以及触发的位置与顺序对触发器的影响。在这一部分，我完成了所有的实验并且取得了满分。

### 2.3.1 任务1：为投资表property实现业务约束规则

任务要求：补充代码，实现本任务所要求的完整性业务规则。当插入的数据不符合要求时，拒绝数据的插入，并反馈出错信息：

(1) pro\_type数据不合法时，显示:

type x is illegal!

这里，x系指试图插入的pro\_type值。

(2) pro\_type = 1,但pro\_pif\_id不是finances\_product表中的某个主码值，显示:

finances product #x not found!

这里,x系指试图插入的pro\_pif\_id的值。

(3) pro\_type = 2,但pro\_pif\_id不是insurance表中的某个主码值，显示:

insurance #x not found!

这里,x系指试图插入的pro\_pif\_id的值。

(3) pro\_type = 3,但pro\_pif\_id不是fund表中的某个主码值，显示:

fund #x not found!

这里,x系指试图插入的pro\_pif\_id的值。

注意事项：可以查阅OpenGauss的字符串函数，构造出错信息；（提示使用concat函数；当数据不合法时，用raise exception语句抛出异常，并设置出错信息：“raise exception '%',msg;”，其中，msg需替换成你想要显示的提示信息(不超过128个字符)。

相关知识： OpenGauss的流程控制编程(参见存储过程实训)；触发器的基本知识；触发器的创建；触发触发器的时机；触发触发器的事件；触发器内的特殊表。

程序设计：触发器函数TRI\_INSERT\_FUNC()的设计是在插入新数据到property表之前，检查新数据的合法性，主要是检查pro\_type、pro\_pif\_id的值。具体来说，它检查了以下几点：

首先先用一个IF语句检测新数据的pro\_type字段是否在1,2,3这三个值中，如果不在，函数会抛出一个异常，异常信息是"type [pro\_type] is illegal!"。

如果新数据的pro\_type字段值为1，函数会检查新数据的pro\_pif\_id字段值是否在finances\_product表的p\_id字段值中。如果不在，函数会抛出一个异常，异常信息是"finances product #[pro\_pif\_id] not found!"。

如果新数据的pro\_type字段值为2，函数会检查新数据的pro\_pif\_id字段值是否在insurance表的i\_id字段值中。如果不在，函数会抛出一个异常，异常信息是"insurance #[pro\_pif\_id] not found!"。

如果新数据的pro\_type字段值为3，函数会检查新数据的pro\_pif\_id字段值是否在fund表的f\_id字段值中。如果不在，函数会抛出一个异常，异常信息是"fund #[pro\_pif\_id] not found!"。

在所有检查都通过后，函数才会允许插入的元组插入。

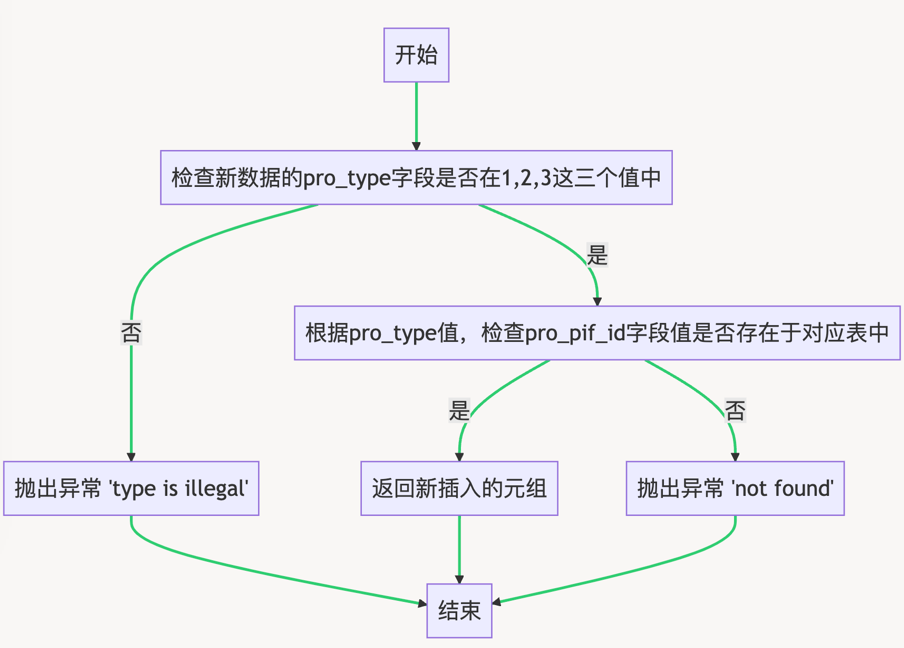


图2.13：实现业务约束规则的流程图

触发器before\_property\_inserted的作用是在每次向property表插入新数据之前，调用触发器函数TRI\_INSERT\_FUNC()，以执行上述的检查操作，这样就可以确保插入到property表的数据都是合法的了。

为投资表property实现业务约束规则的流程图见图2.13。

## 2.4 数据库应用开发(JAVA篇)

这部分一共有6个关卡，要求结合 java 和数据库语言实现一个简单的数据库应用，是关于金融行业的。在这个过程中，我们会学习到JDBC的语句和结构，连接上在线数据库进行编程。在这一部分，我完成了所有的题目并且取得了满分。

### 2.4.1 任务2：用户登录

任务要求：编程体验客户登录功能，程序先后提示客户输用户名和密码：客户的邮箱(c\_mail)充当用户名，而不是编号(c\_id)，因为通常邮箱更容易记住。随后根据客户的输入,输出以下两类信息之一：“登录成功”或者“用户名或密码错误！”

相关知识：正确使用JDBC；条件不确定的查询。

程序设计：首先先创建Connection、Statement和ResultSet对象，这些对象分别是用于建立数据库连接、执行SQL语句和处理查询结果的。之后使用Scanner对象读取用户的用户名和密码，然后加载PostgreSQL的JDBC驱动：这是通过调用Class.forName方法并传入驱动的类名来完成的。在连接完数据库之后，我们定义SQL查询语句，这个语句用于从client表中查询与输入的用户名匹配的记录。之后执行SQL查询并获取结果，如果找到了匹配的记录，并且记录中的密码与输入的密码相同，那么就打印出"登录成功"的消息；否则，就打印出"用户名或密码错误"的消息。最后在finally块中关闭ResultSet、Statement和Connection对象，这是为了释放数据库资源。

虽然这段这段代码完成了任务要求的主要目的——验证用户的用户名和密码，但是它没有进行任何形式的输入验证或错误处理，也没有使用预编译的SQL语句来防止SQL注入攻击，因此在实际使用中需要进行改进。代码部分太长了，因此不作展示。

### 2.4.2 任务5：客户修改密码

任务要求：实现修改密码的方法。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，方法返回一个整数：1 - 密码修改成功；2 - 用户不存在；3 - 密码不正确；-1 - 程序异常(如没能连接到数据库等）。

相关知识：JDBC;Statement或PreparedStatement类的应用。

程序设计：这段Java代码主要实现了一个密码修改功能，它连接到一个PostgreSQL数据库并根据用户的输入来修改密码。首先，我定义了一个名为passwd的方法，这个方法接收一个数据库连接对象、一个邮箱地址（也是登录名）、一个旧密码和一个新密码作为参数。这个方法首先创建一个PreparedStatement对象和一个ResultSet对象，然后执行一个SQL查询语句，这个语句用于从client表中查询与输入的邮箱地址匹配的记录。这条语句是"select \* from client where c\_mail = ?;"，使用了PreparedStatement和SetString的方法来传递变量。

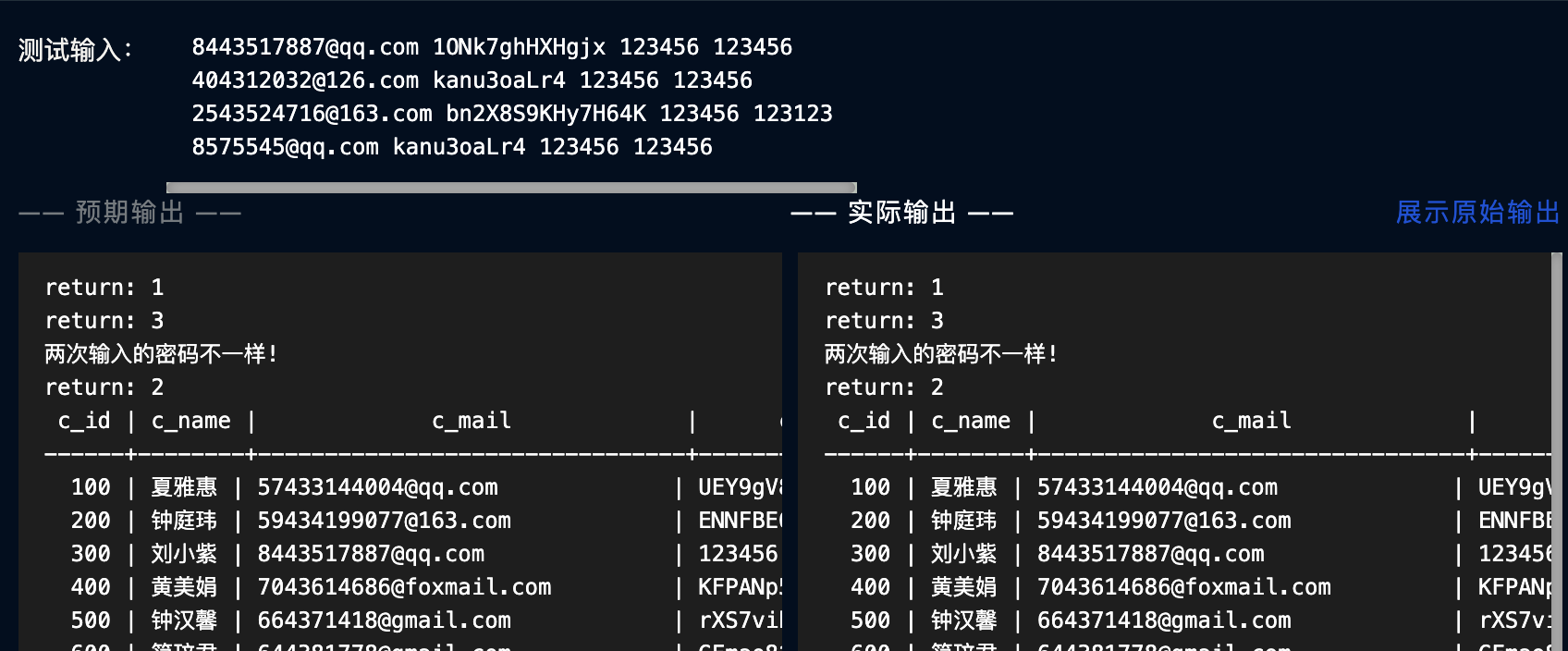


图2.14：客户修改密码的测试结果

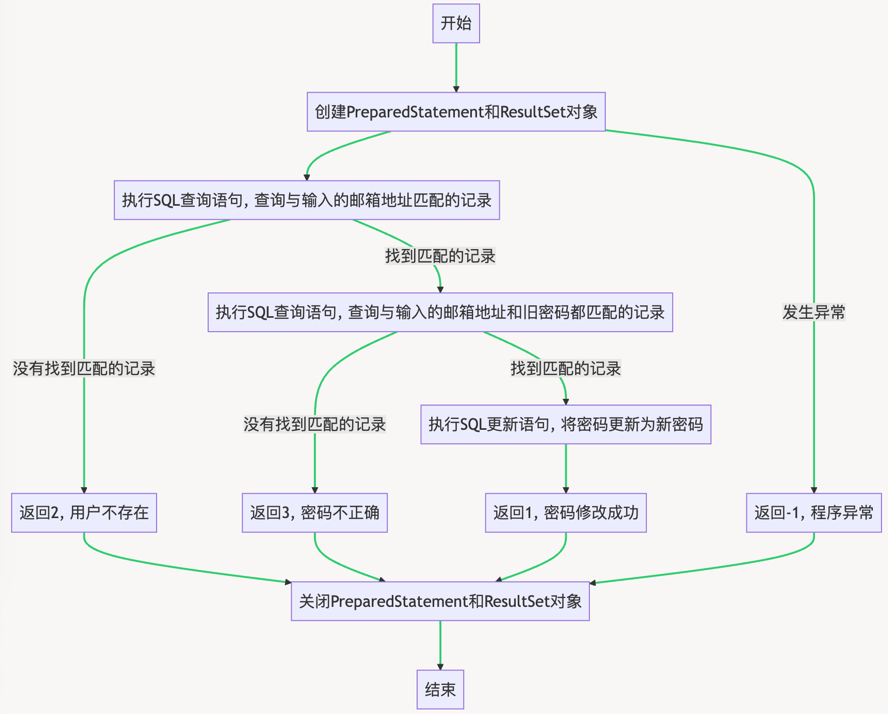


图2.15：客户修改密码的流程图

如果没有找到匹配的记录，那么方法返回2，表示用户不存在。如果找到了匹配的记录，那么它执行另一个SQL查询语句，这个语句用于从client表中查询与输入的邮箱地址和旧密码都匹配的记录。这条语句是"select \* from client where c\_mail = ? and c\_password = ?;"，和上文一样是用SetString传递变量的。如果没有找到匹配的记录，那么方法返回3，表示密码不正确。如果找到了匹配的记录，那么它执行一个SQL更新语句，这个语句用于将client表中对应记录的密码更新为新密码，然后方法返回1，表示密码修改成功。如果在执行这些操作的过程中发生了任何异常，那么方法返回-1，表示程序异常。

程序的测试见图2.14，流程图见图2.15。

### 2.4.3 任务6：事务与转账操作

任务要求：实现一个银行卡转账的方法，方法返回boolean值，true表示转帐成功，false表示转账失败，并不需要细分或解释失败的原因。

下列任一情形都不可转账(转账失败的原因)：转出或转入帐号不存在；转出账号是信用卡；转出帐号余额不足。

相关知识：JDBC的基本知识(参见2.3.1);JDBC的事务处理。

程序设计：我们需要设计的方法接收四个参数：一个数据库连接对象，转出账号，转入账号，以及转账金额。首先，我们创建了PreparedStatement和ResultSet对象，并将数据库连接的自动提交模式设置为false，这是为了确保转账操作的原子性，即要么全部成功，要么全部失败，不会出现只完成部分操作的情况。在这里我们采用先执行SQL语句，如果出现问题再进行回滚的操作，非常具有数据库的特色。

然后，执行了一系列SQL语句来完成转账操作。首先，从转出账号的余额中扣除转账金额。然后，如果转入账号是储蓄卡，就将转账金额加到其余额中；如果转入账号是信用卡，就将其余额减去转账金额。这里需要注意的是，信用卡的余额实际上表示的是已透支的金额，所以在转账时需要减去转账金额。

接下来，查询转出账号的信息，如果查询结果为空，或者转出账号的余额小于0，就认为转账失败，回滚事务，返回false。否则，查询转入账号的信息，如果查询结果为空，也认为转账失败，回滚事务，返回false。如果以上所有操作都成功完成，就提交事务，返回true。最后，关闭了PreparedStatement和ResultSet对象，并处理了可能出现的异常。如果在执行过程中出现任何异常，都会回滚事务，返回false。

总的来说，在这里我们把自动提交关闭来确保操作的原子性对我来说还是非常新颖的，让我深刻认识到了数据库操作的原子性，避免了可能出现的数据不一致问题。同时，通过参数化查询，防止了SQL注入攻击，提高了代码的安全性。

程序的测试结果见图2.15，程序的流程图见图2.16。

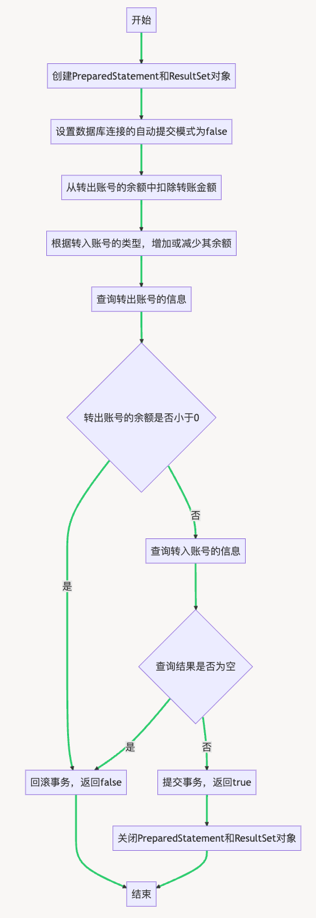


图2.16：事务与转账操作的测试结果 图2.17：事务与转账操作的流程图

### 2.4.4 任务7：把稀疏表格转为键值对存储

任务要求：一个表有很多列，但只有少数列存储有值，这样的表称为稀疏表。举个例子，大学里有几千名学生和几百门课程，为记录学生的成绩，设计这样一张表：每门课程占一列，每个学生占一行。但是，绝大多数学生只会选修这几百门课程中的一小部分，因此，整个表只会稀疏地存储了一些数据，这将产生大量的闲置空间。

一种更有效的方法是以(学号, 课程，成绩)这样的三元组形式存储。因此我们在这里就是把这种稀疏表使用SQL语句转化成三元组表。

相关知识：JDBC的相关知识；综合利用查询和插入操作完成数据的转存。

程序设计：首先，我完善了"insertSC"的函数，该函数接收一个数据库连接对象，一个学生编号，一个类型和一个分数作为参数。这个函数的主要目的是将这些参数插入到名为"sc"的数据库表中。它首先创建一个预处理的SQL语句："insert into sc values (?,?,?);"，然后通过SetInt和SetValue设置参数值，并执行更新操作。如果插入成功，函数将返回更新的行数。

在主函数"main"中，首先加载了数据库驱动，并建立了到数据库的连接。然后，创建了一个语句对象，并执行了一个查询操作，从名为"entrance\_exam"的表中获取所有的数据，这里使用的是next方法进行遍历。在获取到的数据集中，它遍历每一行数据，获取了学生编号和各科的分数。如果某科的分数不为0，就调用"insertSC"函数，将学生编号，科目类型和分数插入到"sc"表中。

在这个过程中，也考虑了各科分数为0或者分数不存在的情况，只有当分数存在的时候，才会插入到"sc"表中。具体测试结果见图2.18，流程图见图2.19。

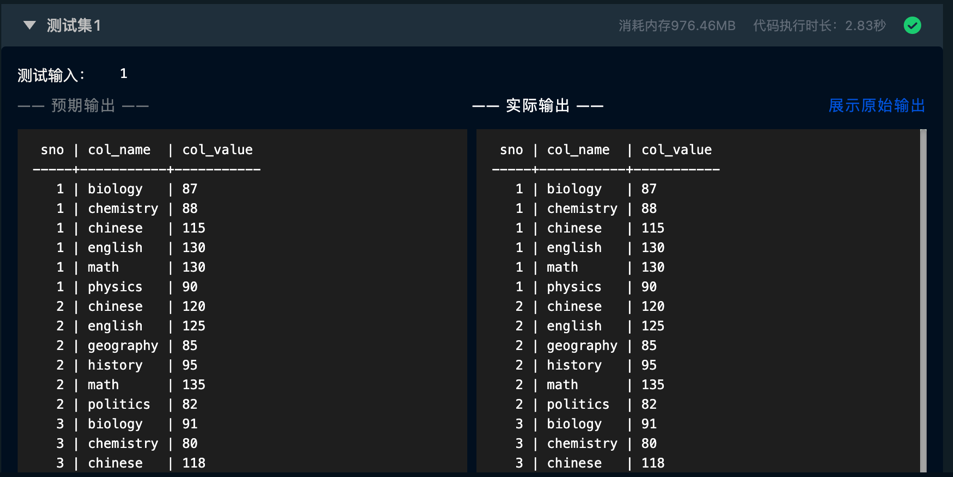
****

图2.18：把稀疏表格转为键值对存储的测试结果

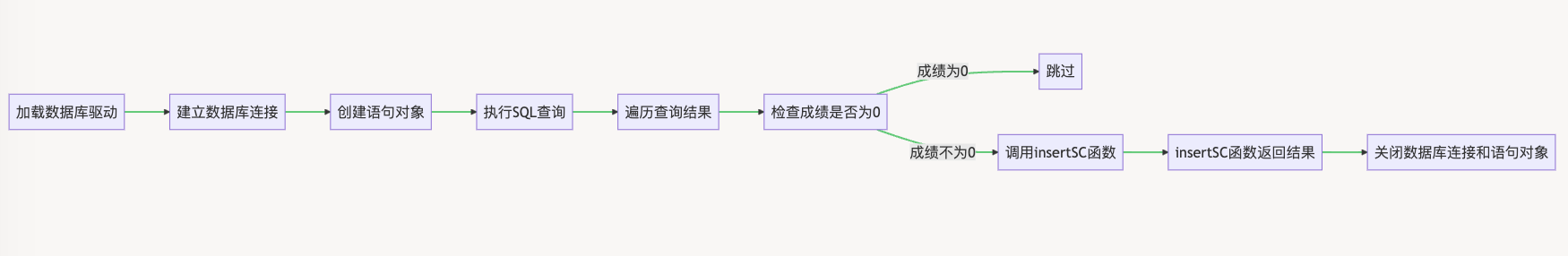


图2.19：把稀疏表格转为键值对存储的流程图

## 2.5数据库设计与实现

在这一部分，我们要学习如何从概念模型到OpenGauss实现、如何从需求分析到逻辑模型以及如何使用类似于PowerDesigner这类建模软件自动生成sql语句。在这一部分，我完成了所有的实验并且取得了满分。

### 2.5.1 任务2：从需求分析到逻辑模型

任务要求：设计一个影院管理系统。影院对当前的放映厅和电影进行排片，顾客到来后，可以购买任一排场的电影票，进入对应放映厅观看。系统中有以下实体集：

电影(movie)：属性有标识号(movie\_ID)、电影名(title)、类型(type)、时长(runtime)、首映日期(release\_date)、导演姓名(director)、主演姓名(starring)。

顾客(customer)：属性有标识号(c\_ID)、姓名(name)、手机号(phone)。

放映厅(hall)：属性有标识号(hall\_ID)、放映模式(mode)、容纳人数(capacity)、位置(location)。

排场(schedule)：属性有标识号(schedule\_ID)、日期(date)、时间(time)、票价(price)、票数(number)。

电影票(ticket)：属性有标识号(ticket\_ID)、座位号(seat\_num)。

实体间的关系描述如下：

①. 顾客和电影票有一对多的购买关系。每位顾客可以买多张电影票，每张电影票被一位顾客购买。

②. 电影票和排场有多对一的属于关系。一张电影票只属于一个排场，一个排场有多张电影票。

③. 排场和电影有一对多的放映关系。每个排场放一部电影，每部电影可以在多个排场放映。

④. 排场和放映厅有一对多的位于关系。每个排场位于一个放映厅，每个放映厅可以安排多个排场。

请按照上述要求完成：1.画出E-R图。2.给出对应的关系模式。

程序设计：在这一部分，我使用了draw.io来绘制了ER图。需要注意的是一些外码是没有在任务要求中直接给出的，需要自己后续自己领悟，例如一些主码和外码都是需要自己定义的。

以下是我设计的关系模型，ER图见图2.20：

|  |
| --- |
| 顾客(c\_ID, name, phone), primary key:(c\_ID);  电影票(ticket\_ID, seat\_num, schedule\_ID), primary key(ticket\_ID), foreign key(schedule\_ID,c\_ID);  排场(schedule\_ID, date, time, price, number, movie\_ID, hall\_ID), primary key:(schedule\_ID), foreign key(movie\_ID, hall\_ID);  放映厅(hall\_ID, mode, capacity, location), primary key:(hall\_ID);  电影(movie\_ID, title, type, runtime, release\_date, director, starring), primary key:(movie\_ID); |

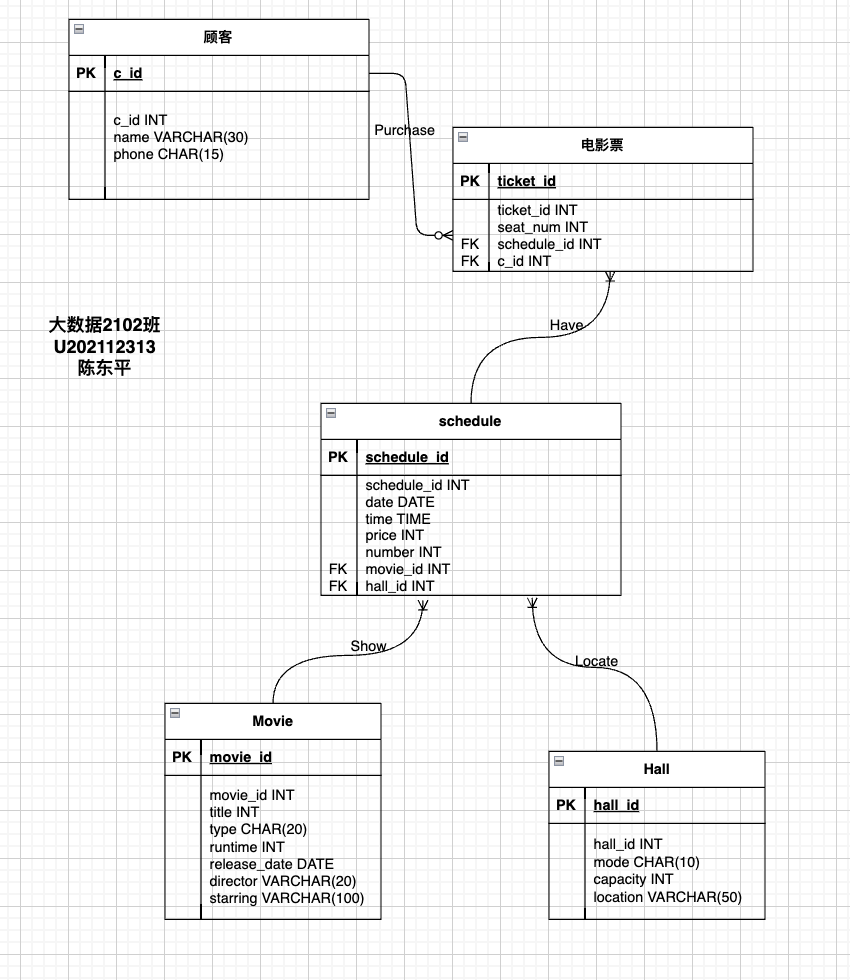


图2.20：影院管理系统的ER图

## 2.6 数据库的索引B+树实现

索引是数据库中的重要的组成部分，能够免于遍历数据的耗时，快速的定位记录。而B+树作为数据库索引所用数据结构，能够使得数据库在较小的磁盘访问次数内获取记录，有着稳定且高效的速度。作为数据库基础知识中不可缺少的一环，该项目将设计并实现一个完整的索引B+树其结构设计如图2.21所示。我在这一部分完成了所有的任务并且取得了满分。

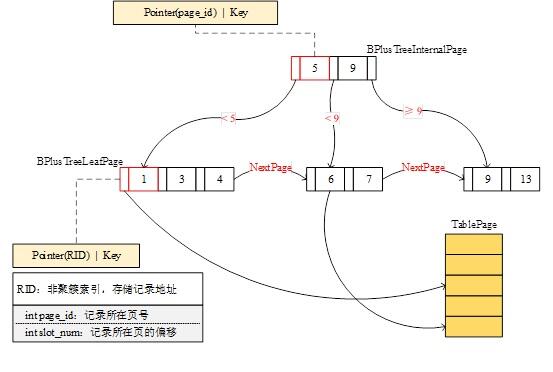


图2.21：B+树图示

### 2.6.1 任务4：B+树索引Insert

任务要求：在完成B+树相关数据结构的设计后，请完成B+树索引的插入功能。实现B+树迭代器。

注意事项：在我们的实验中， B+树索引仅存在Unique Key，即索引中存储的key值不重复。一颗完整的B+树由内部结点和叶结点组成，随着记录的插入，结点不断分裂增加，但仍需保持B+树的完整性和一致性。

可以使用BufferPoolManager类来对缓冲区进行操作，比如通过page\_id获得对应页，Pin操作防止该页被移出缓冲区，new创建页，delete删除页。

相关知识：B+树的基本特性，对记录插入时B+树结构的变化过程和细节有所了解；B+树迭代器的基本功能；C++的相关知识。

程序设计：在本部分中，我完成了相关的20个函数内容的编写，这里挑出我觉得比较具有代表性的函数详细讲解，并且会附上相应的代码，相应的我会放到附录里面并且标明对应关系。

（1）void BPLUSTREE\_TYPE::InsertIntoParent(BPlusTreePage \*old\_node, const KeyType &key, BPlusTreePage \*new\_node)：

函数输入：old\_node（分裂过的原节点）、key（新索引的键值）和new\_node（分裂后生成的新节点）

函数说明：首先，函数获取old\_node的父节点ID。如果父节点ID无效，说明old\_node是根节点。在这种情况下，函数会创建一个新的根节点，并调用PopulateNewRoot函数快速添加对原根节点和分裂节点的索引。然后，它会更新根节点的页面ID，并将新节点的父节点ID设置为新的根节点ID。最后，它会将新节点和新的根节点从缓冲池中解锁。

如果父节点ID有效，函数会获取父节点的页面，并在父节点中插入新元素。如果插入新元素后父节点的大小超过了最大值，函数会调用Split函数分裂父节点，并递归调用InsertIntoParent函数将新的内部节点插入到父节点的父节点中。然后，它会将新节点的父节点ID设置为父节点的ID，并将父节点和新节点从缓冲池中解锁。

这个函数的主要目的是保持B+树的平衡。当一个节点分裂时，它会创建一个新的节点，并将一部分键移动到新节点中。然后，它需要在父节点中插入一个新的键，该键是新节点中的最小键，以及一个指向新节点的指针。如果父节点没有足够的空间存储新的键和指针，父节点也需要分裂。这个过程可能会一直递归到根节点，可能需要创建一个新的根节点。

需要注意的是，在整个函数过程中，我们多次使用了BufferPoolManager来获取原始的数据和BPlusTree的数据，并且用它来标记脏页修改数据，每次只访问一个页面，访问完之后关闭再打开下一个，这样子可以减轻缓存的压力。

该函数的流程图如图2.22所示。

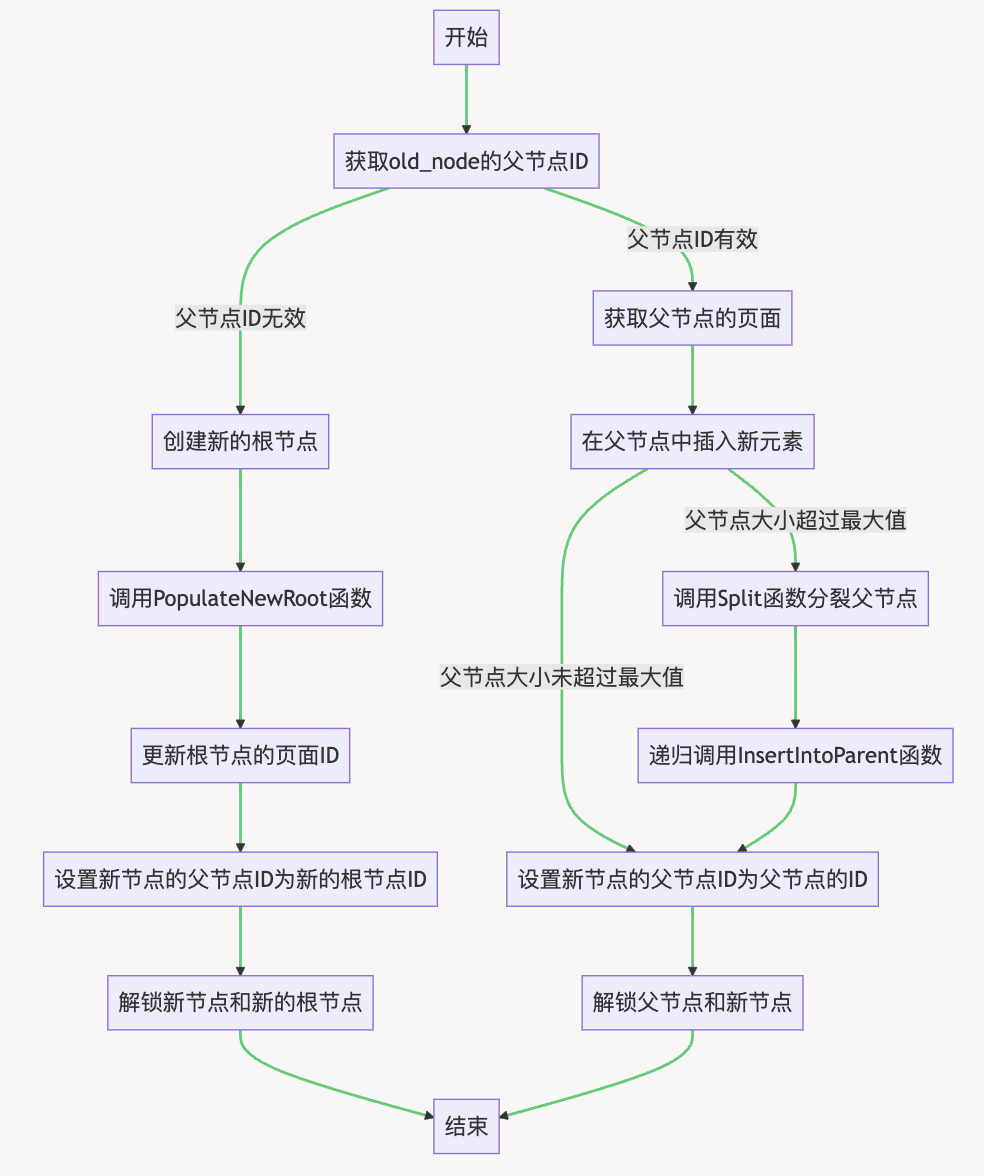


图2.22：InsertIntoParent函数设计流程图

（2）Page \*BPLUSTREE\_TYPE::FindLeafPage(const KeyType &key, bool leftMost)：

函数输入：key（需要查找的键值）和leftMost（一个bool，当为true时，函数会直接返回最左叶节点，这在遍历输出索引记录时特别有用）

函数说明：首先，函数检查B+树是否为空。如果为空，函数直接返回空指针。如果不为空，函数会获取根节点的页面，并将其转换为BPlusTreePage类型的指针。

然后，函数进入一个循环，持续进行直到找到目标叶子节点。在每次循环中，函数首先将当前页面转换为内部页面，要使用BufferPoolManager不停的关闭和打开读取页面，编写起来还是有点繁琐的。如果leftMost为true，函数会获取内部页面的第一个值作为下一个页面的ID。否则，函数会使用Lookup函数查找键值，并获取相应的页面ID。然后，函数会解锁当前页面，并将下一个页面的ID设置为当前页面ID。最后，函数会获取当前页面的数据，并将其转换为BPlusTreePage类型的指针。

当函数找到目标叶子节点时，它会返回该节点的页面。这个函数的主要目的是在B+树中查找键值，这是许多数据库操作的基础，例如插入、删除和搜索，因此掌握这个函数相当重要。

该函数的流程图如图2.23所示。

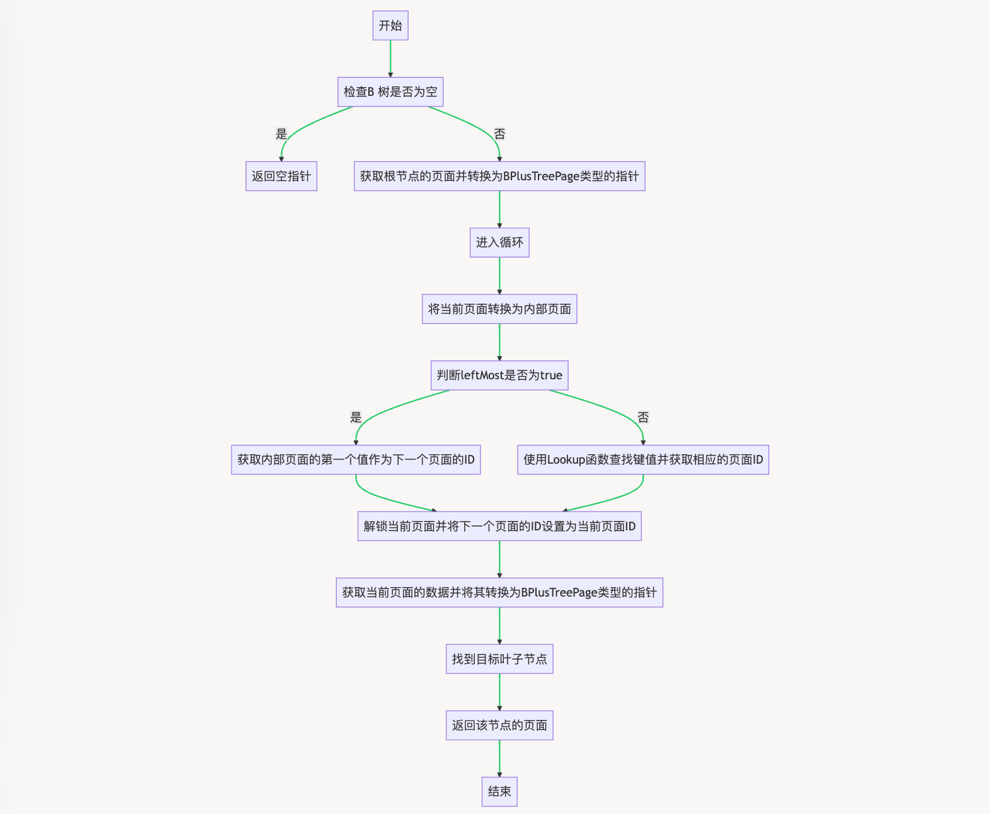


图2.23：FindLeafPage函数设计流程图

# 3 课程总结

数据库原理实践课程的学习，对我来说是一次非常难得的一段冒险，我从对数据库没有一点了解到头歌测试全部通过并且满分，掌握了很多数据库相关的知识。这次的实践课程非常有意思，虽然头歌平台老是“Fail connect to host：5432”，但还是让我有机会很好去理解数据库管理系统的基础和功能。通过深入实际的数据库操作，我掌握了一些数据库管理系统的基础知识和应用技巧，成功完成了所有预定的任务，包括数据库和表的定义，表的完整性约束的创建和修改，数据查询、插入、修改和删除，视图的创建和使用，存储过程和触发器的使用，以及用户自定义函数、安全性控制、并发控制和事务的隔离级别的理解和实践。

在这个过程中，我充分理解了数据库系统设计的各个环节。我从如何定义数据库和表开始，逐步理解了如何创建和修改表的完整性约束，让我深刻理解了完整性约束对于维护数据的正确性和一致性的重要性；通过进行各种数据查询和修改，学习了数据操作的各种方法，熟练掌握了复杂的嵌套查询和条件查询；通过创建和使用视图以及存储过程，理解了这些工具如何简化常见的数据库操作，提高效率；通过JDBC的实验，我了解到数据库独有的回滚操作，以及如何让数据库的操作保持原子性。在学习过程中，我不断地练习编写SQL语句，逐步从基本的数据查询和修改，到复杂的数据分析，再到优化查询性能，层层深入。我逐渐理解到，SQL语句的效率直接关系到数据库应用的性能，因此在编写SQL语句时，不仅要考虑其正确性，也要考虑其效率，例如少用笛卡尔积以及多表查询，可以多用Join来连接表。同时，我也深入研究了数据库的并发控制和事务隔离级别，这让我明白了数据库管理系统如何处理多个用户同时访问和修改数据的情况。同时，我也学会了如何为了保证数据的一致性和完整性，数据库管理系统必须通过并发控制和事务管理来协调多个用户的操作。

此次实践课程的最后一部分，也是对我来说最具挑战性的部分，是实现了一个基于B+树的索引结构。这个任务对我来说既是一个挑战，因为我的C++其实学的很烂。通过设计和实现B+树，我理解了索引结构的工作原理和性能优势。同时，这也是我首次接触到数据库中这种高级的数据结构，之前一直都停留在用的阶段，现在让我实现我才知道这玩意有多难。

回顾整个课程实践过程，我认为我达到了预定的学习目标，并从中获得了丰富的知识和经验。然而，我也深知，作为一名大数据专业的学生，在数据库方面我仍有许多需要学习和提高的地方。我需要更深入地理解SQL语句，提高查询性能和效率，尤其是在处理大型数据集时。同时，我需要进一步了解并实践数据库设计，以提高我的设计能力。我还需要更深入地理解数据库的安全性和并发性，以便在未来的项目中更好地应用这些知识。

总的来说，这次课程实践是一次非常宝贵的学习和成长的机会。我感谢课程组的所有老师，特别是潘鹏老师，当我有问题在群里提问时总是飞速给出解答，而且很照顾同学的身体情况，愿意给我们开网课上课，您的专业知识和热心指导使我受益匪浅。期待在未来的学习和研究中，我能继续应用和扩展我在这次课程实践中获得的知识和经验。

# 附录

2.6.1的代码：

|  |
| --- |
| INDEX\_TEMPLATE\_ARGUMENTS  void BPLUSTREE\_TYPE::InsertIntoParent(BPlusTreePage \*old\_node, const KeyType &key, BPlusTreePage \*new\_node) {  page\_id\_t parentId = old\_node->GetParentPageId();  if (parentId == INVALID\_PAGE\_ID) {  Page \*const newPage = buffer\_pool\_manager\_->NewPage(&root\_page\_id\_);  B\_PLUS\_TREE\_INTERNAL\_PAGE \*newRoot = reinterpret\_cast<B\_PLUS\_TREE\_INTERNAL\_PAGE \*>(newPage->GetData());  newRoot->Init(root\_page\_id\_, INVALID\_PAGE\_ID,internal\_max\_size\_);  newRoot->PopulateNewRoot(old\_node->GetPageId(), key, new\_node->GetPageId());  old\_node->SetParentPageId(root\_page\_id\_);  new\_node->SetParentPageId(root\_page\_id\_);  UpdateRootPageId();  buffer\_pool\_manager\_->UnpinPage(new\_node->GetPageId(), true);  buffer\_pool\_manager\_->UnpinPage(newRoot->GetPageId(), true);  return;  } else {  Page \*page = buffer\_pool\_manager\_->FetchPage(parentId);  B\_PLUS\_TREE\_INTERNAL\_PAGE \*parentPage = reinterpret\_cast<B\_PLUS\_TREE\_INTERNAL\_PAGE \*>(page->GetData());  new\_node->SetParentPageId(parentId);  parentPage->InsertNodeAfter(old\_node->GetPageId(), key, new\_node->GetPageId());  if (parentPage->GetSize() > parentPage->GetMaxSize()) {  B\_PLUS\_TREE\_INTERNAL\_PAGE \*newLeafPage = Split(parentPage);  InsertIntoParent(parentPage, newLeafPage->KeyAt(0), newLeafPage);  }  buffer\_pool\_manager\_->UnpinPage(parentPage->GetPageId(), true);  buffer\_pool\_manager\_->UnpinPage(new\_node->GetPageId(), true);  return;  }  } |

2.6.2的代码：

|  |
| --- |
| INDEX\_TEMPLATE\_ARGUMENTS  Page \*BPLUSTREE\_TYPE::FindLeafPage(const KeyType &key, bool leftMost) {  if (IsEmpty()) return nullptr;  Page \*page = buffer\_pool\_manager\_->FetchPage(root\_page\_id\_);  BPlusTreePage \*pointer = reinterpret\_cast<BPlusTreePage \*>(page->GetData());  page\_id\_t next;  for (page\_id\_t cur = root\_page\_id\_; !pointer->IsLeafPage();) {  B\_PLUS\_TREE\_INTERNAL\_PAGE \*internalPage = static\_cast<B\_PLUS\_TREE\_INTERNAL\_PAGE \*>(pointer);  if (leftMost) {  next = internalPage->ValueAt(0);  } else {  next = internalPage->Lookup(key, comparator\_);  }  buffer\_pool\_manager\_->UnpinPage(cur, false);  cur = next;  page = buffer\_pool\_manager\_->FetchPage(cur);  pointer = reinterpret\_cast<BPlusTreePage \*>(page->GetData());  }  return page;  } |