# MySQL高级-03-授课笔记

## 一、MySQL存储过程和函数

### 1.存储过程和函数的概念

- 存储过程和函数是 事先经过编译并存储在数据库中的一段 SQL 语句的集合

### 2.存储过程和函数的好处

- 存储过程和函数可以重复使用，减轻开发人员的工作量。类似于java中方法可以多次调用
- 减少网络流量，存储过程和函数位于服务器上，调用的时候只需要传递名称和参数即可
- 减少数据在数据库和应用服务器之间的传输，可以提高数据处理的效率
- 将一些业务逻辑在数据库层面来实现，可以减少代码层面的业务处理

### 3.存储过程和函数的区别

- 函数必须有返回值
- 存储过程没有返回值

### 4.创建存储过程

- 小知识

```
/*
    该关键字用来声明sql语句的分隔符，告诉MySQL该段命令已经结束!
    sql语句默认的分隔符是分号，但是有的时候我们需要一条功能sql语句中包含分号，但是并不作为结束标识。
    这个时候就可以使用DELIMITER来指定分隔符了!
*/
-- 标准语法
DELIMITER 分隔符
```

- 数据准备

```
-- 创建db8数据库
CREATE DATABASE db8;

-- 使用db8数据库
USE db8;

-- 创建学生表
CREATE TABLE student(
    id INT PRIMARY KEY AUTO_INCREMENT,  -- 学生id
    NAME VARCHAR(20),                   -- 学生姓名
    age INT,                            -- 学生年龄
    gender VARCHAR(5),                  -- 学生性别
    score INT                           -- 学生成绩
);
-- 添加数据
INSERT INTO student VALUES (NULL,'张三',23,'男',95),(NULL,'李四',24,'男',98),
(NULL,'王五',25,'女',100),(NULL,'赵六',26,'女',90);
```

```
-- 按照性别进行分组，查询每组学生的总成绩。按照总成绩的升序排序
SELECT gender,SUM(score) getSum FROM student GROUP BY gender ORDER BY getSum
ASC;
```

- 创建存储过程语法

```
-- 修改分隔符为$
DELIMITER $

-- 标准语法
CREATE PROCEDURE 存储过程名称(参数...)
BEGIN
    sql语句;
END$

-- 修改分隔符为分号
DELIMITER ;
```

- 创建存储过程

```
-- 修改分隔符为$
DELIMITER $

-- 创建存储过程，封装分组查询学生总成绩的sql语句
CREATE PROCEDURE stu_group()
BEGIN
    SELECT gender,SUM(score) getSum FROM student GROUP BY gender ORDER BY getSum
ASC;
END$

-- 修改分隔符为分号
DELIMITER ;
```

## 5.调用存储过程

- 调用存储过程语法

```
-- 标准语法
CALL 存储过程名称(实际参数);

-- 调用stu_group存储过程
CALL stu_group();
```

## 6.查看存储过程

- 查看存储过程语法

```
-- 查询数据库中所有的存储过程 标准语法
SELECT * FROM mysql.proc WHERE db='数据库名称';
```

## 7.删除存储过程

- 删除存储过程语法

```
-- 标准语法
DROP PROCEDURE [IF EXISTS] 存储过程名称;

-- 删除stu_group存储过程
DROP PROCEDURE stu_group;
```

## 8.存储过程语法

### 8.1存储过程语法介绍

- 存储过程是可以进行编程的。意味着可以使用变量、表达式、条件控制语句、循环语句等，来完成比较复杂的功能!

### 8.2变量的使用

- 定义变量

```
-- 标准语法
DECLARE 变量名 数据类型 [DEFAULT 默认值];
-- 注意： DECLARE定义的是局部变量，只能用在BEGIN END范围之内

-- 定义一个int类型变量、并赋默认值为10
DELIMITER $

CREATE PROCEDURE pro_test1()
BEGIN
    DECLARE num INT DEFAULT 10;    -- 定义变量
    SELECT num;                    -- 查询变量
END$

DELIMITER ;

-- 调用pro_test1存储过程
CALL pro_test1();
```

- 变量的赋值1

```
-- 标准语法
SET 变量名 = 变量值;

-- 定义字符串类型变量，并赋值
DELIMITER $

CREATE PROCEDURE pro_test2()
BEGIN
    DECLARE NAME VARCHAR(10);    -- 定义变量
    SET NAME = '存储过程';        -- 为变量赋值
    SELECT NAME;                 -- 查询变量
END$

DELIMITER ;

-- 调用pro_test2存储过程
CALL pro_test2();
```

- 变量的赋值2

```
-- 标准语法
SELECT 列名 INTO 变量名 FROM 表名 [WHERE 条件];

-- 定义两个int变量，用于存储男女同学的总分数
DELIMITER $

CREATE PROCEDURE pro_test3()
BEGIN
    DECLARE men,women INT;  -- 定义变量
    SELECT SUM(score) INTO men FROM student WHERE gender='男';    -- 计算男同学总分数赋值给men
    SELECT SUM(score) INTO women FROM student WHERE gender='女';  -- 计算女同学总分数赋值给women
    SELECT men,women;            -- 查询变量
END$

DELIMITER ;

-- 调用pro_test3存储过程
CALL pro_test3();
```

### 8.3if语句的使用

- 标准语法

```
-- 标准语法
IF 判断条件1 THEN 执行的sql语句1;
[ELSEIF 判断条件2 THEN 执行的sql语句2;]
...
[ELSE 执行的sql语句n;]
END IF;
```

- 案例演示

```
/*
    定义一个int变量，用于存储班级总成绩
    定义一个varchar变量，用于存储分数描述
    根据总成绩判断：
        380分及以上     学习优秀
        320 ~ 380       学习不错
        320以下         学习一般
*/
DELIMITER $

CREATE PROCEDURE pro_test4()
BEGIN
    -- 定义总分数变量
    DECLARE total INT;
    -- 定义分数描述变量
    DECLARE description VARCHAR(10);
    -- 为总分数变量赋值
    SELECT SUM(score) INTO total FROM student;
    -- 判断总分数
    IF total >= 380 THEN
        SET description = '学习优秀';
    ELSEIF total >= 320 AND total < 380 THEN
```

```
        SET description = '学习不错';
    ELSE
        SET description = '学习一般';
    END IF;

    -- 查询总成绩和描述信息
    SELECT total,description;
END$

DELIMITER ;

-- 调用pro_test4存储过程
CALL pro_test4();
```

### 8.4参数的传递

- 参数传递的语法

```
DELIMITER $

-- 标准语法
CREATE PROCEDURE 存储过程名称([IN|OUT|INOUT] 参数名 数据类型)
BEGIN
    执行的sql语句;
END$
/*
    IN:代表输入参数，需要由调用者传递实际数据。默认的
    OUT:代表输出参数，该参数可以作为返回值
    INOUT:代表既可以作为输入参数，也可以作为输出参数
*/
DELIMITER ;
```

- 输入参数
    - 标准语法

```
DELIMITER $

-- 标准语法
CREATE PROCEDURE 存储过程名称(IN 参数名 数据类型)
BEGIN
    执行的sql语句;
END$

DELIMITER ;
```

    - 案例演示

```
/*
    输入总成绩变量，代表学生总成绩
    定义一个varchar变量，用于存储分数描述
    根据总成绩判断:
        380分及以上   学习优秀
        320 ~ 380     学习不错
        320以下       学习一般
*/
```

```
DELIMITER $

CREATE PROCEDURE pro_test5(IN total INT)
BEGIN
    -- 定义分数描述变量
    DECLARE description VARCHAR(10);
    -- 判断总分数
    IF total >= 380 THEN
        SET description = '学习优秀';
    ELSEIF total >= 320 AND total < 380 THEN
        SET description = '学习不错';
    ELSE
        SET description = '学习一般';
    END IF;

    -- 查询总成绩和描述信息
    SELECT total,description;
END$

DELIMITER ;

-- 调用pro_test5存储过程
CALL pro_test5(390);
CALL pro_test5((SELECT SUM(score) FROM student));
```

- 输出参数
  - 标准语法

```
DELIMITER $

-- 标准语法
CREATE PROCEDURE 存储过程名称(OUT 参数名 数据类型)
BEGIN
    执行的sql语句;
END$

DELIMITER ;
```

  - 案例演示

```
/*
    输入总成绩变量，代表学生总成绩
    输出分数描述变量，代表学生总成绩的描述
    根据总成绩判断：
        380分及以上   学习优秀
        320 ~ 380     学习不错
        320以下       学习一般
*/
DELIMITER $

CREATE PROCEDURE pro_test6(IN total INT,OUT description VARCHAR(10))
BEGIN
    -- 判断总分数
    IF total >= 380 THEN
        SET description = '学习优秀';
    ELSEIF total >= 320 AND total < 380 THEN
```

```
        SET description = '学习不错';
    ELSE
        SET description = '学习一般';
    END IF;
END$

DELIMITER ;

-- 调用pro_test6存储过程
CALL pro_test6(310,@description);

-- 查询总成绩描述
SELECT @description;
```

- 小知识

```
@变量名：  这种变量要在变量名称前面加上“@”符号，叫做用户会话变量，代表整个会话过程他都是有作用的，这个类似于全局变量一样。

@@变量名: 这种在变量前加上 "@@" 符号，叫做系统变量
```

### 8.5case语句的使用

- 标准语法1

```
-- 标准语法
CASE 表达式
WHEN 值1 THEN 执行sql语句1;
[WHEN 值2 THEN 执行sql语句2;]
...
[ELSE 执行sql语句n;]
END CASE;
```

- 标准语法2

```
-- 标准语法
CASE
WHEN 判断条件1 THEN 执行sql语句1;
[WHEN 判断条件2 THEN 执行sql语句2;]
...
[ELSE 执行sql语句n;]
END CASE;
```

- 案例演示

```
/*
    输入总成绩变量，代表学生总成绩
    定义一个varchar变量，用于存储分数描述
    根据总成绩判断：
        380分及以上   学习优秀
        320 ~ 380     学习不错
        320以下       学习一般
*/
DELIMITER $
```

```
CREATE PROCEDURE pro_test7(IN total INT)
BEGIN
    -- 定义变量
    DECLARE description VARCHAR(10);
    -- 使用case判断
    CASE
    WHEN total >= 380 THEN
        SET description = '学习优秀';
    WHEN total >= 320 AND total < 380 THEN
        SET description = '学习不错';
    ELSE
        SET description = '学习一般';
    END CASE;

    -- 查询分数描述信息
    SELECT description;
END$

DELIMITER ;

-- 调用pro_test7存储过程
CALL pro_test7(390);
CALL pro_test7((SELECT SUM(score) FROM student));
```

### 8.6while循环

- 标准语法

```
-- 标准语法
初始化语句;
WHILE 条件判断语句 DO
    循环体语句;
    条件控制语句;
END WHILE;
```

- 案例演示

```
/*
    计算1~100之间的偶数和
*/
DELIMITER $

CREATE PROCEDURE pro_test8()
BEGIN
    -- 定义求和变量
    DECLARE result INT DEFAULT 0;
    -- 定义初始化变量
    DECLARE num INT DEFAULT 1;
    -- while循环
    WHILE num <= 100 DO
        -- 偶数判断
        IF num%2=0 THEN
            SET result = result + num; -- 累加
        END IF;

        -- 让num+1
        SET num = num + 1;
```

```
    END WHILE;

    -- 查询求和结果
    SELECT result;
END$

DELIMITER ;

-- 调用pro_test8存储过程
CALL pro_test8();
```

### 8.7repeat循环

- 标准语法

```
-- 标准语法
初始化语句;
REPEAT
    循环体语句;
    条件控制语句;
    UNTIL 条件判断语句
END REPEAT;

-- 注意：repeat循环是条件满足则停止。while循环是条件满足则执行
```

- 案例演示

```
/*
    计算1~10之间的和
*/
DELIMITER $

CREATE PROCEDURE pro_test9()
BEGIN
    -- 定义求和变量
    DECLARE result INT DEFAULT 0;
    -- 定义初始化变量
    DECLARE num INT DEFAULT 1;
    -- repeat循环
    REPEAT
        -- 累加
        SET result = result + num;
        -- 让num+1
        SET num = num + 1;

        -- 停止循环
        UNTIL num>10
    END REPEAT;

    -- 查询求和结果
    SELECT result;
END$

DELIMITER ;

-- 调用pro_test9存储过程
CALL pro_test9();
```

### 8.8loop循环

- 标准语法

```
-- 标准语法
初始化语句;
[循环名称:] LOOP
    条件判断语句
        [LEAVE 循环名称;]
    循环体语句;
    条件控制语句;
END LOOP 循环名称;

-- 注意：loop可以实现简单的循环，但是退出循环需要使用其他的语句来定义。我们可以使用leave语句完成!
--       如果不加退出循环的语句，那么就变成了死循环。
```

- 案例演示

```
/*
    计算1~10之间的和
*/
DELIMITER $

CREATE PROCEDURE pro_test10()
BEGIN
    -- 定义求和变量
    DECLARE result INT DEFAULT 0;
    -- 定义初始化变量
    DECLARE num INT DEFAULT 1;
    -- loop循环
    l:LOOP
        -- 条件成立，停止循环
        IF num > 10 THEN
            LEAVE l;
        END IF;

        -- 累加
        SET result = result + num;
        -- 让num+1
        SET num = num + 1;
    END LOOP l;

    -- 查询求和结果
    SELECT result;
END$

DELIMITER ;

-- 调用pro_test10存储过程
CALL pro_test10();
```

### 8.9游标

- 游标的概念

    - 游标可以遍历返回的多行结果，每次拿到一整行数据
    - 在存储过程和函数中可以使用游标对结果集进行循环的处理
    - 简单来说游标就类似于集合的迭代器遍历
    - MySQL中的游标只能用在存储过程和函数中
- 游标的语法
    - 创建游标

```
-- 标准语法
DECLARE 游标名称 CURSOR FOR 查询sql语句;
```

    - 打开游标

```
-- 标准语法
OPEN 游标名称;
```

    - 使用游标获取数据

```
-- 标准语法
FETCH 游标名称 INTO 变量名1,变量名2,...;
```

    - 关闭游标

```
-- 标准语法
CLOSE 游标名称;
```

- 游标的基本使用

```
-- 创建stu_score表
CREATE TABLE stu_score(
    id INT PRIMARY KEY AUTO_INCREMENT,
    score INT
);

/*
    将student表中所有的成绩保存到stu_score表中
*/
DELIMITER $

CREATE PROCEDURE pro_test11()
BEGIN
    -- 定义成绩变量
    DECLARE s_score INT;
    -- 创建游标,查询所有学生成绩数据
    DECLARE stu_result CURSOR FOR SELECT score FROM student;

    -- 开启游标
    OPEN stu_result;

    -- 使用游标，遍历结果,拿到第1行数据
    FETCH stu_result INTO s_score;
    -- 将数据保存到stu_score表中
    INSERT INTO stu_score VALUES (NULL,s_score);

    -- 使用游标，遍历结果,拿到第2行数据
```

```
    FETCH stu_result INTO s_score;
    -- 将数据保存到stu_score表中
    INSERT INTO stu_score VALUES (NULL,s_score);

    -- 使用游标，遍历结果,拿到第3行数据
    FETCH stu_result INTO s_score;
    -- 将数据保存到stu_score表中
    INSERT INTO stu_score VALUES (NULL,s_score);

    -- 使用游标，遍历结果,拿到第4行数据
    FETCH stu_result INTO s_score;
    -- 将数据保存到stu_score表中
    INSERT INTO stu_score VALUES (NULL,s_score);

    -- 关闭游标
    CLOSE stu_result;
END$

DELIMITER ;

-- 调用pro_test11存储过程
CALL pro_test11();

-- 查询stu_score表
SELECT * FROM stu_score;


-- ============================================================
/*
    出现的问题：
        student表中一共有4条数据，我们在游标遍历了4次，没有问题！
        但是在游标中多遍历几次呢？就会出现问题
*/
DELIMITER $

CREATE PROCEDURE pro_test11()
BEGIN
    -- 定义成绩变量
    DECLARE s_score INT;
    -- 创建游标,查询所有学生成绩数据
    DECLARE stu_result CURSOR FOR SELECT score FROM student;

    -- 开启游标
    OPEN stu_result;

    -- 使用游标，遍历结果,拿到第1行数据
    FETCH stu_result INTO s_score;
    -- 将数据保存到stu_score表中
    INSERT INTO stu_score VALUES (NULL,s_score);

    -- 使用游标，遍历结果,拿到第2行数据
    FETCH stu_result INTO s_score;
    -- 将数据保存到stu_score表中
    INSERT INTO stu_score VALUES (NULL,s_score);

    -- 使用游标，遍历结果,拿到第3行数据
    FETCH stu_result INTO s_score;
    -- 将数据保存到stu_score表中
```

```
    INSERT INTO stu_score VALUES (NULL,s_score);

    -- 使用游标，遍历结果,拿到第4行数据
    FETCH stu_result INTO s_score;
    -- 将数据保存到stu_score表中
    INSERT INTO stu_score VALUES (NULL,s_score);

    -- 使用游标，遍历结果,拿到第5行数据
    FETCH stu_result INTO s_score;
    -- 将数据保存到stu_score表中
    INSERT INTO stu_score VALUES (NULL,s_score);

    -- 关闭游标
    CLOSE stu_result;
END$

DELIMITER ;

-- 调用pro_test11存储过程
CALL pro_test11();

-- 查询stu_score表,虽然数据正确，但是在执行存储过程时会报错
SELECT * FROM stu_score;
```

- 游标的优化使用(配合循环使用)

```
/*
    当游标结束后，会触发游标结束事件。我们可以通过这一特性来完成循环操作
    加标记思想：
        1.定义一个变量，默认值为0(意味着有数据)
        2.当游标结束后，将变量值改为1(意味着没有数据了)
*/
-- 1.定义一个变量，默认值为0(意味着有数据)
DECLARE flag INT DEFAULT 0;
-- 2.当游标结束后，将变量值改为1(意味着没有数据了)
DECLARE EXIT HANDLER FOR NOT FOUND SET flag = 1;
```

```
/*
    将student表中所有的成绩保存到stu_score表中
*/
DELIMITER $

CREATE PROCEDURE pro_test12()
BEGIN
    -- 定义成绩变量
    DECLARE s_score INT;
    -- 定义标记变量
    DECLARE flag INT DEFAULT 0;
    -- 创建游标，查询所有学生成绩数据
    DECLARE stu_result CURSOR FOR SELECT score FROM student;
    -- 游标结束后，将标记变量改为1
    DECLARE EXIT HANDLER FOR NOT FOUND SET flag = 1;

    -- 开启游标
    OPEN stu_result;
```

```
    -- 循环使用游标
    REPEAT
        -- 使用游标，遍历结果,拿到数据
        FETCH stu_result INTO s_score;
        -- 将数据保存到stu_score表中
        INSERT INTO stu_score VALUES (NULL,s_score);
    UNTIL flag=1
    END REPEAT;

    -- 关闭游标
    CLOSE stu_result;
END$

DELIMITER ;

-- 调用pro_test12存储过程
CALL pro_test12();

-- 查询stu_score表
SELECT * FROM stu_score;
```

## 9.存储过程的总结

- 存储过程是 事先经过编译并存储在数据库中的一段 SQL 语句的集合。可以在数据库层面做一些业务处理
- 说白了存储过程其实就是将sql语句封装为方法，然后可以调用方法执行sql语句而已
- 存储过程的好处
    - 安全
    - 高效
    - 复用性强

## 10.存储函数

- 存储函数和存储过程是非常相似的。存储函数可以做的事情，存储过程也可以做到!
- 存储函数有返回值，存储过程没有返回值(参数的out其实也相当于是返回数据了)
- 标准语法
    - 创建存储函数

```
DELIMITER $

-- 标准语法
CREATE FUNCTION 函数名称([参数 数据类型])
RETURNS 返回值类型
BEGIN
    执行的sql语句;
    RETURN 结果;
END$

DELIMITER ;
```

    - 调用存储函数

```
-- 标准语法
SELECT 函数名称(实际参数);
```

- 删除存储函数

```
-- 标准语法
DROP FUNCTION 函数名称;
```

- 案例演示

```
/*
    定义存储函数，获取学生表中成绩大于95分的学生数量
*/
DELIMITER $

CREATE FUNCTION fun_test1()
RETURNS INT
BEGIN
    -- 定义统计变量
    DECLARE result INT;
    -- 查询成绩大于95分的学生数量，给统计变量赋值
    SELECT COUNT(*) INTO result FROM student WHERE score > 95;
    -- 返回统计结果
    RETURN result;
END$

DELIMITER ;

-- 调用fun_test1存储函数
SELECT fun_test1();
```

# 二、MySQL触发器

## 1.触发器的概念

- 触发器是与表有关的数据库对象，可以在 insert/update/delete 之前或之后，触发并执行触发器中定义的SQL语句。触发器的这种特性可以协助应用在数据库端确保数据的完整性 、日志记录 、数据校验等操作 。
- 使用别名 NEW 和 OLD 来引用触发器中发生变化的记录内容，这与其他的数据库是相似的。现在触发器还只支持行级触发，不支持语句级触发。

| 触发器类型 | OLD的含义 | NEW的含义 |
| --- | --- | --- |
| INSERT 型触发器 | 无 (因为插入前状态无数据) | NEW 表示将要或者已经新增的数据 |
| UPDATE 型触发器 | OLD 表示修改之前的数据 | NEW 表示将要或已经修改后的数据 |
| DELETE 型触发器 | OLD 表示将要或者已经删除的数据 | 无 (因为删除后状态无数据) |

## 2.创建触发器

- 标准语法

```
DELIMITER $

CREATE TRIGGER 触发器名称
BEFORE|AFTER INSERT|UPDATE|DELETE
ON 表名
[FOR EACH ROW]  -- 行级触发器
BEGIN
    触发器要执行的功能;
END$

DELIMITER ;
```

- 触发器演示。通过触发器记录账户表的数据变更日志。包含：增加、修改、删除
  - 创建账户表

```
-- 创建db9数据库
CREATE DATABASE db9;

-- 使用db9数据库
USE db9;

-- 创建账户表account
CREATE TABLE account(
    id INT PRIMARY KEY AUTO_INCREMENT,  -- 账户id
    NAME VARCHAR(20),                   -- 姓名
    money DOUBLE                        -- 余额
);
-- 添加数据
INSERT INTO account VALUES (NULL,'张三',1000),(NULL,'李四',2000);
```

  - 创建日志表

```
-- 创建日志表account_log
CREATE TABLE account_log(
    id INT PRIMARY KEY AUTO_INCREMENT,  -- 日志id
    operation VARCHAR(20),              -- 操作类型 (insert update delete)
    operation_time DATETIME,            -- 操作时间
    operation_id INT,                   -- 操作表的id
    operation_params VARCHAR(200)       -- 操作参数
);
```

  - 创建INSERT触发器

```
-- 创建INSERT触发器
DELIMITER $

CREATE TRIGGER account_insert
AFTER INSERT
ON account
FOR EACH ROW
BEGIN
    INSERT INTO account_log VALUES (NULL,'INSERT',NOW(),new.id,CONCAT('插入后{id=',new.id,',name=',new.name,',money=',new.money,'}'));
END$
```

```
DELIMITER ;

-- 向account表添加记录
INSERT INTO account VALUES (NULL,'王五',3000);

-- 查询account表
SELECT * FROM account;

-- 查询日志表
SELECT * FROM account_log;
```

- 创建UPDATE触发器

```
-- 创建UPDATE触发器
DELIMITER $

CREATE TRIGGER account_update
AFTER UPDATE
ON account
FOR EACH ROW
BEGIN
    INSERT INTO account_log VALUES (NULL,'UPDATE',NOW(),new.id,CONCAT('修改前
{id=',old.id,',name=',old.name,',money=',old.money,'}','修改后
{id=',new.id,',name=',new.name,',money=',new.money,'}'));
END$

DELIMITER ;

-- 修改account表
UPDATE account SET money=3500 WHERE id=3;

-- 查询account表
SELECT * FROM account;

-- 查询日志表
SELECT * FROM account_log;
```

- 创建DELETE触发器

```
-- 创建DELETE触发器
DELIMITER $

CREATE TRIGGER account_delete
AFTER DELETE
ON account
FOR EACH ROW
BEGIN
    INSERT INTO account_log VALUES (NULL,'DELETE',NOW(),old.id,CONCAT('删除前
{id=',old.id,',name=',old.name,',money=',old.money,'}'));
END$

DELIMITER ;

-- 删除account表数据
DELETE FROM account WHERE id=3;
```

```sql
-- 查询account表
SELECT * FROM account;

-- 查询日志表
SELECT * FROM account_log;
```

### 3.查看触发器

```sql
-- 标准语法
SHOW TRIGGERS;

-- 查看触发器
SHOW TRIGGERS;
```

### 4.删除触发器

```sql
-- 标准语法
DROP TRIGGER 触发器名称;

-- 删除DELETE触发器
DROP TRIGGER account_delete;
```

### 5.触发器的总结

- 触发器是与表有关的数据库对象
- 可以在 insert/update/delete 之前或之后，触发并执行触发器中定义的SQL语句
- 触发器的这种特性可以协助应用在数据库端确保数据的完整性 、日志记录 、数据校验等操作
- 使用别名 NEW 和 OLD 来引用触发器中发生变化的记录内容

## 三、MySQL事务

### 1.事务的概念

- 一条或多条 SQL 语句组成一个执行单元，其特点是这个单元要么同时成功要么同时失败，单元中的每条 SQL 语句都相互依赖，形成一个整体，如果某条 SQL 语句执行失败或者出现错误，那么整个单元就会回滚，撤回到事务最初的状态，如果单元中所有的 SQL 语句都执行成功，则事务就顺利执行。

### 2.事务的数据准备

```sql
-- 创建db10数据库
CREATE DATABASE db10;

-- 使用db10数据库
USE db10;

-- 创建账户表
CREATE TABLE account(
    id INT PRIMARY KEY AUTO_INCREMENT,  -- 账户id
    NAME VARCHAR(20),           -- 账户名称
    money DOUBLE                -- 账户余额
);
-- 添加数据
INSERT INTO account VALUES (NULL,'张三',1000),(NULL,'李四',1000);
```

### 3.未管理事务演示

```
-- 张三给李四转账500元
-- 1.张三账户-500
UPDATE account SET money=money-500 WHERE NAME='张三';
-- 2.李四账户+500
出错了...
UPDATE account SET money=money+500 WHERE NAME='李四';

-- 该场景下，这两条sql语句要么同时成功，要么同时失败。就需要被事务所管理!
```

### 4.管理事务演示

- 操作事务的三个步骤
    1. 开启事务：记录回滚点，并通知服务器，将要执行一组操作，要么同时成功、要么同时失败
    2. 执行sql语句：执行具体的一条或多条sql语句
    3. 结束事务(提交|回滚)
        - 提交：没出现问题，数据进行更新
        - 回滚：出现问题，数据恢复到开启事务时的状态
- 开启事务

```
-- 标准语法
START TRANSACTION;
```

- 回滚事务

```
-- 标准语法
ROLLBACK;
```

- 提交事务

```
-- 标准语法
COMMIT;
```

- 管理事务演示

```
-- 开启事务
START TRANSACTION;

-- 张三给李四转账500元
-- 1.张三账户-500
UPDATE account SET money=money-500 WHERE NAME='张三';
-- 2.李四账户+500
-- 出错了...
UPDATE account SET money=money+500 WHERE NAME='李四';

-- 回滚事务(出现问题)
ROLLBACK;

-- 提交事务(没出现问题)
COMMIT;
```

## 5.事务的提交方式

- 提交方式
  - 自动提交(MySQL默认为自动提交)
  - 手动提交
- 修改提交方式
  - 查看提交方式

```
-- 标准语法
SELECT @@AUTOCOMMIT;  -- 1代表自动提交    0代表手动提交
```

  - 修改提交方式

```
-- 标准语法
SET @@AUTOCOMMIT=数字;

-- 修改为手动提交
SET @@AUTOCOMMIT=0;

-- 查看提交方式
SELECT @@AUTOCOMMIT;
```

## 6.事务的四大特征(ACID)

- 原子性(atomicity)
  - 原子性是指事务包含的所有操作要么全部成功，要么全部失败回滚，因此事务的操作如果成功就必须要完全应用到数据库，如果操作失败则不能对数据库有任何影响
- 一致性(consistency)
  - 一致性是指事务必须使数据库从一个一致性状态变换到另一个一致性状态，也就是说一个事务执行之前和执行之后都必须处于一致性状态
  - 拿转账来说，假设张三和李四两者的钱加起来一共是2000，那么不管A和B之间如何转账，转几次账，事务结束后两个用户的钱相加起来应该还得是2000，这就是事务的一致性
- 隔离性(isolcation)
  - 隔离性是当多个用户并发访问数据库时，比如操作同一张表时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，多个并发事务之间要相互隔离
- 持久性(durability)
  - 持久性是指一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的，即便是在数据库系统遇到故障的情况下也不会丢失提交事务的操作

## 7.事务的隔离级别

- 隔离级别的概念
  - 多个客户端操作时,各个客户端的事务之间应该是隔离的，相互独立的,不受影响的。
  - 而如果多个事务操作同一批数据时，则需要设置不同的隔离级别,否则就会产生问题。
  - 我们先来了解一下四种隔离级别的名称,再来看看可能出现的问题
- 四种隔离级别

| | | | 会引发的问题 |
|---|---|---|---|
| 1 | 读未提交 | read uncommitted | 脏读、不可重复读、幻读 |
| 2 | 读已提交 | read committed | 不可重复读、幻读 |
| 3 | 可重复读 | repeatable read | 幻读 |
| 4 | 串行化 | serializable | 无 |

- 可能引发的问题

| 问题 | 现象 |
|---|---|
| 脏读 | 是指在一个事务处理过程中读取了另一个未提交的事务中的数据，导致两次查询结果不一致 |
| 不可重复读 | 是指在一个事务处理过程中读取了另一个事务中修改并已提交的数据，导致两次查询结果不一致 |
| 幻读 | select 某记录是否存在，不存在，准备插入此记录，但执行 insert 时发现此记录已存在，无法插入。或不存在执行delete删除，却发现删除成功 |

- 查询数据库隔离级别

```
-- 标准语法
SELECT @@TX_ISOLATION;
```

- 修改数据库隔离级别

```
-- 标准语法
SET GLOBAL TRANSACTION ISOLATION LEVEL 级别字符串;

-- 修改数据库隔离级别为read uncommitted
SET GLOBAL TRANSACTION ISOLATION LEVEL read uncommitted;

-- 查看隔离级别
SELECT @@TX_ISOLATION;    -- 修改后需要断开连接重新开
```

## 8.事务隔离级别演示

- 脏读的问题
  - 窗口1

```
-- 查询账户表
select * from account;

-- 设置隔离级别为read uncommitted
set global transaction isolation level read uncommitted;

-- 开启事务
start transaction;

-- 转账
update account set money = money - 500 where id = 1;
```

```
update account set money = money + 500 where id = 2;

-- 窗口2查询转账结果 ,出现脏读(查询到其他事务未提交的数据)

-- 窗口2查看转账结果后，执行回滚
rollback;
```

- 窗口2

```
-- 查询隔离级别
select @@tx_isolation;

-- 开启事务
start transaction;

-- 查询账户表
select * from account;
```

- 解决脏读的问题和演示不可重复读的问题
  - 窗口1

```
-- 设置隔离级别为read committed
set global transaction isolation level read committed;

-- 开启事务
start transaction;

-- 转账
update account set money = money - 500 where id = 1;
update account set money = money + 500 where id = 2;

-- 窗口2查看转账结果，并没有发生变化(脏读问题被解决了)

-- 执行提交事务。
commit;

-- 窗口2查看转账结果，数据发生了变化(出现了不可重复读的问题，读取到其他事务已提交的数据)
```

  - 窗口2

```
-- 查询隔离级别
select @@tx_isolation;

-- 开启事务
start transaction;

-- 查询账户表
select * from account;
```

- 解决不可重复读的问题
  - 窗口1

```
-- 设置隔离级别为repeatable read
set global transaction isolation level repeatable read;
```

```
-- 开启事务
start transaction;

-- 转账
update account set money = money - 500 where id = 1;
update account set money = money + 500 where id = 2;

-- 窗口2查看转账结果，并没有发生变化

-- 执行提交事务
commit;

-- 这个时候窗口2只要还在上次事务中，看到的结果都是相同的。只有窗口2结束事务，才能看到变化(不可重复读的问题被解决)
```

- 窗口2

```
-- 查询隔离级别
select @@tx_isolation;

-- 开启事务
start transaction;

-- 查询账户表
select * from account;

-- 提交事务
commit;

-- 查询账户表
select * from account;
```

- 幻读的问题和解决
  - 窗口1

```
-- 设置隔离级别为repeatable read
set global transaction isolation level repeatable read;

-- 开启事务
start transaction;

-- 添加一条记录
INSERT INTO account VALUES (3,'王五',1500);

-- 查询账户表，本窗口可以查看到id为3的结果
SELECT * FROM account;

-- 提交事务
COMMIT;
```

  - 窗口2

```
-- 查询隔离级别
select @@tx_isolation;
```

```
-- 开启事务
start transaction;

-- 查询账户表，查询不到新添加的id为3的记录
select * from account;

-- 添加id为3的一条数据，发现添加失败。出现了幻读
INSERT INTO account VALUES (3,'测试',200);

-- 提交事务
COMMIT;

-- 查询账户表，查询到了新添加的id为3的记录
select * from account;
```

- 解决幻读的问题

```
/*
    窗口1
*/
-- 设置隔离级别为serializable
set global transaction isolation level serializable;

-- 开启事务
start transaction;

-- 添加一条记录
INSERT INTO account VALUES (4,'赵六',1600);

-- 查询账户表，本窗口可以查看到id为4的结果
SELECT * FROM account;

-- 提交事务
COMMIT;



/*
    窗口2
*/
-- 查询隔离级别
select @@tx_isolation;

-- 开启事务
start transaction;

-- 查询账户表，发现查询语句无法执行，数据表被锁住！只有窗口1提交事务后，才可以继续操作
select * from account;

-- 添加id为4的一条数据，发现已经存在了，就不会再添加了！幻读的问题被解决
INSERT INTO account VALUES (4,'测试',200);

-- 提交事务
COMMIT;
```

## 9.隔离级别总结

| | 隔离级别 | 名称 | 出现脏读 | 出现不可重复读 | 出现幻读 | 数据库默认隔离级别 |
|---|---|---|---|---|---|---|
| 1 | **read uncommitted** | 读未提交 | 是 | 是 | 是 | |
| 2 | **read committed** | 读已提交 | 否 | 是 | 是 | Oracle / SQL Server |
| 3 | **repeatable read** | 可重复读 | 否 | 否 | 是 | MySQL |
| 4 | **serializable** | 串行化 | 否 | 否 | 否 | |

> 注意：隔离级别从小到大安全性越来越高，但是效率越来越低 , 所以不建议使用READ UNCOMMITTED 和 SERIALIZABLE 隔离级别.

## 10.事务的总结

- 一条或多条 SQL 语句组成一个执行单元，其特点是这个单元要么同时成功要么同时失败。例如转账操作
- 开启事务：start transaction;
- 回滚事务：rollback;
- 提交事务：commit;
- 事务四大特征
  - 原子性
  - 持久性
  - 隔离性
  - 一致性
- 事务的隔离级别
  - read uncommitted(读未提交)
  - read committed (读已提交)
  - repeatable read (可重复读)
  - serializable (串行化)