第五章 存储过程、函数、触发器和视图

课前回顾

1. 查询有哪些分类

```
内连接： 衔接的多表必须保证数据一一对应才可能展示结果
左外连接：衔接的主表（ LEFT JOIN 左边的表为主表）为准，从表中没有的数据以NULL形式展示
右外连接：衔接的主表（ RIGHT JOIN 右边的表为主表）为准，从表中没有的数据以NULL形式展示
```

2. 子查询有哪些分类

```
在 SELECT ... FROM 之间：执行时机就是查询出结果后执行
在 FROM ... WHERE 之间：执行时机是一开始就先执行
在 WHERE 之后：执行时机是筛选数据时执行
```

章节内容

- 变量 **熟悉**
- 存储过程 **重点**
- 事务 **重点**
- 自定义函数 **熟悉**
- 触发器 **重点**
- 视图 **重点**

章节目标

- 熟悉变量的使用
- 掌握存储过程的使用
- 掌握事务的使用
- 熟悉自定义函数的使用
- 掌握触发器的使用
- 掌握视图的使用

第一节 变量

在MySQL中，变量分为四种类型，即局部变量、用户变量、会话变量和全局变量。其中局部变量和用户变量在实际应用中使用较多，会话变量和全局变量使用较少，因此作为了解即可。

1. 全局变量

MySQL全局变量会影响服务器整体操作，当服务启动时，它将所有全局变量初始化为默认值。要想更改全局变量，必须具有管理员权限。其作用域为服务器的整个生命周期。

```sql
-- 显示所有的全局变量
SHOW GLOBAL VARIABLES;

-- 设置全局变量的值的两种方式
SET GLOBAL sql_warnings = ON;        -- GLOBAL不能省略
SET @@GLOBAL.sql_warnings = OFF;

-- 查询全局变量的值的两种方式
SELECT @@GLOBAL.sql_warnings;
SHOW GLOBAL VARIABLES LIKE '%sql_warnings%';
```

2. 会话变量

MySQL会话变量是服务器为每个连接的客户端维护的一系列变量。其作用域仅限于当前连接，因此，会话变量是独立的。

```sql
-- 显示所有的会话变量
SHOW SESSION VARIABLES;

-- 设置会话变量的值的三种方式
SET SESSION auto_increment_increment = 1;
SET @@SESSION.auto_increment_increment = 2;
-- 当省略SESSION关键字时，默认缺省为SESSION，即设置会话变量的值
SET auto_increment_increment = 3;

-- 查询会话变量的值的三种方式
SELECT @@auto_increment_increment;
SELECT @@SESSION.auto_increment_increment;
-- SESSION关键字可省略,也可用关键字LOCAL替代
SHOW SESSION VARIABLES LIKE '%auto_increment_increment%';

SET @@LOCAL.auto_increment_increment = 1;
SELECT @@LOCAL.auto_increment_increment;
```

3. 用户变量

MySQL用户变量，MySQL中用户变量不用提前申明，在用的时候直接用“@变量名”使用就可以了。其作用域为当前连接。

```sql
-- 第一种用法，使用SET时可以用“=”或“:=”两种赋值符号赋值
SET @age = 19;

-- 第二种用法，使用SELECT时必须用“:=”赋值符号赋值
SELECT @age := 22;
SELECT @age := age FROM stu WHERE `name` = '张华';

-- 第三种用法，使用SELECT ... INTO语句赋值
SELECT age INTO @age FROM stu WHERE `name` = '张华';
SELECT @age;
```

4. 局部变量

MySQL局部变量，只能用在BEGIN/END语句块中，比如存储过程中的BEGIN/END语句块。

```
-- 定义局部变量
DECLARE age INT(3) DEFAULT 0;
-- 为局部变量赋值
SET age = 10;
SELECT age := 10;
SELECT 10 INTO age;
SELECT age;
```

第二节 存储过程

1. 概念

在大型数据库系统中，存储过程是一组为了完成特定功能而存储在数据库中的SQL语句集，一次编译后永久有效

2. 为什么要使用存储过程

- **运行速度快：**

  在存储过程创建的时候，数据库已经对其进行了一次解析和优化。存储过程一旦执行，在内存中就会保留一份这个存储过程，下次再执行同样的存储过程时，可以从内存中直接调用，所以执行速度会比普通SQL快。

- **减少网络传输：**

  存储过程直接就在数据库服务器上跑，所有的数据访问都在数据库服务器内部进行，不需要传输数据到其它服务器，所以会减少一定的网络传输。

- **增强安全性：**

  提高代码安全，防止SQL被截获、篡改。

3. 如何使用存储过程

语法

```
-- 声明分隔符
[DELIMITER $$]
CREATE PROCEDURE 存储过程名称 ([IN | OUT | INOUT] 参数名1 数据类型, [[IN | OUT | INOUT] 参数名2 数据类型, ..., [IN | OUT | INOUT] 参数名n 数据类型])
-- 语句块开始
BEGIN
-- SQL语句集
END[$$]
-- 还原分隔符
[DELIMITER ; ]


-- 调用存储过程
CALL 存储过程名(参数1,参数2,…);
```

示例

使用存储过程完成银行转账业务

```
-- 创建存储过程
CREATE PROCEDURE transfer(IN transferFrom BIGINT, IN transferTo BIGINT, IN transferMoney BIGINT(20))
BEGIN
  UPDATE account SET balance = balance - transferMoney WHERE account = transferFrom;
  UPDATE account SET balance = balance + transferMoney WHERE account = transferTo;
END

-- 调用存储过程
CALL transfer(123456, 123457, 2000);
```

如果转账账户余额不足，上面的SQL代码依然可以正常执行，只是执行完后，转账账户的余额变为了负数。这显然不符合常理。因此需要修正。

```
DROP PROCEDURE IF EXISTS transfer;
CREATE PROCEDURE transfer(IN transferFrom BIGINT, IN transferTo BIGINT, IN transferMoney BIGINT(20))
BEGIN
  -- 定义变量表示执行结果：0-失败，1-成功
  DECLARE result TINYINT(1) DEFAULT 0;
  -- 转账账户必须保证余额大于等于转账金额
  UPDATE account SET balance = balance - transferMoney WHERE account = transferFrom AND balance ≥ transferMoney;
  -- 检测受影响的行数是否为1，为1表示更新成功
  IF ROW_COUNT() = 1 THEN
    -- 目标账号余额增加
    UPDATE account SET balance = balance + transferMoney WHERE account = transferTo;
    IF ROW_COUNT() = 1 THEN
      -- 更新结果为1
      SET result = 1;
    END IF;
  END IF;
  -- 查询结果
  SELECT result;
END
```

如果转账账户已经将钱转出去，而在执行目标账户增加余额的时候出现了异常或者目标账户输入错误，此时应该怎么办呢?

MySQL对数据的操作提供了事务的支持，用来保证数据的一致性,可以有效的解决此类问题。

4. 事务

4.1 什么是事务

事务(Transaction)是访问并可能操作各种数据项的一个数据库操作序列，这些操作要么全部执行,要么全部不执行，是一个不可分割的工作单位。事务由事务开始与事务结束之间执行的全部数据库操作组成。

4.2 事务的特性(ACID)

- **原子性（Atomicity）**

  事务的各元素是不可分的（原子的）,它们是一个整体。要么都执行，要么都不执行。

- **一致性（Consistency）**

  当事务完成时，必须保证所有数据保持一致状态。当转账操作完成时，所有账户的总金额应该保持不变，此时数据处于一致性状态；如果总金额发生了改变，说明数据处于非一致性状态。

- **隔离性（Isolation）**

  对数据操作的多个并发事务彼此独立，互不影响。比如张三和李四同时都在进行转账操作，但彼此都不影响对方。

- **持久性（Durability）**

  对于已提交事务，系统必须保证该事务对数据库的改变不被丢失，即使数据库出现故障

4.3 事务解决银行转账问题

```
DROP PROCEDURE IF EXISTS transfer;
CREATE PROCEDURE transfer(IN transferFrom BIGINT, IN transferTo BIGINT, IN transferMoney BIGINT(20))
BEGIN
    -- 定义变量表示执行结果，默认为1
    DECLARE result TINYINT(1) DEFAULT 1;
    -- 声明SQLEXCEPTION处理器，当有SQLEXCEPTION发生时，错误标识符的值设为0
    -- 发生SQLEXCEPTION时的处理方式：CONTINUE，EXIT
    -- CONTINUE表示即使有异常发生，也会执行后面的语句
    -- EXIT表示，有异常发生时，直接退出当前存储过程
    DECLARE CONTINUE HANDLER FOR SQLEXCEPTION SET result = 0;
    -- 开启事务
    START TRANSACTION;
    UPDATE account SET balance = balance - transferMoney WHERE account = transferFrom AND balance ≥ transferMoney;
    -- 设置result的值为SQL执行后受影响的行数
    SET result = ROW_COUNT();
    IF result = 1 THEN
      UPDATE account SET balance = balance + transferMoney WHERE account = transferTo;
      SET result = ROW_COUNT();
    END IF;
    -- 如果result的值为1，表示所有操作都成功，提交事务
    IF result = 1 THEN COMMIT;
    -- 否则，表示操作存在失败的情况，事务回滚，数据恢复到更改之前的状态
    ELSE ROLLBACK;
    END IF;
    SELECT result;
END
```

5. 存储过程输出

```
DROP PROCEDURE IF EXISTS transfer;
CREATE PROCEDURE transfer(IN transferFrom BIGINT, IN transferTo BIGINT, IN transferMoney BIGINT(20), OUT result TINYINT(1))
BEGIN
    -- 为SQL异常声明一个持续处理的处理器，一旦出现异常，则将result的值更改为0
    DECLARE CONTINUE HANDLER FOR SQLEXCEPTION SET result = 0;
    -- 开启事务
    START TRANSACTION;
    UPDATE account SET balance = balance - transferMoney WHERE account = transferFrom AND balance ≥ transferMoney;
    -- 设置result的值为SQL执行后受影响的行数
    SET result = ROW_COUNT();
    IF result = 1 THEN
      UPDATE account SET balance = balance + transferMoney WHERE account = transferTo;
      SET result = ROW_COUNT();
    END IF;
    -- 如果result的值为1，表示所有操作都成功，提交事务
    IF result = 1 THEN COMMIT;
    -- 否则，表示操作存在失败的情况，事务回滚，数据恢复到更改之前的状态
    ELSE ROLLBACK;
    END IF;
END

-- 调用存储过程
CALL transfer(123456, 123457, 2000, @rs);
SELECT @rs;
```

第三节 自定义函数

1. 概念

函数就是在大型数据库系统中，一组为了完成特定功能而存储在数据库中的SQL 语句集，一次编译后永久有效

2. 自定义函数

MySQL本身提供了一些内置函数，这些函数给我们日常的开发和数据操作带来了很大的便利，比如聚合函数SUM()、AVG()以及日期时间函数等。但这并不能完全满足开发的需要，有时我们需要一个函数来完成一些复杂功能的实现，而MySQL中又没有这样的函数，因此，我们需要自定义函数来实现。

3. 如何使用自定义函数

语法

```
CREATE FUNCTION 函数名称 (参数名1 数据类型, 参数名2 数据类型, ..., 参数名n 数据类型])
RETURNS 数据类型
-- 函数特征:
```

```
-- DETERMINISTIC：不确定的
-- NO SQL：没有SQL语句，当然也不会修改数据
-- READS SQL DATA：  只是读取数据，不会修改数据
-- MODIFIES SQL DATA：要修改数据
-- CONTAINS SQL：包含了SQL语句
DETERMINISTIC | NO SQL | READS SQL DATA | MODIFIES SQL DATA | CONTAINS SQL
-- 语句块开始
BEGIN
    -- SQL语句集
    RETURN 结果;
-- 语句块结束
END
```

示例：使用函数实现求score表中的成绩最大差值

```
CREATE FUNCTION getMaxDiff()
RETURNS DOUBLE(5, 2)
DETERMINISTIC
BEGIN
  RETURN (SELECT MAX(score) - MIN(score) FROM score);
END

-- 调用函数
SELECT getMaxDiff();
```

4. 循环结构

```
WHILE 循环条件 DO
   -- SQL语句集
END WHILE;

REPEAT
     -- SQL语句集
UNTIL 循环终止条件 END REPEAT;

标号：LOOP
       -- SQL语句集
        IF 循环终止条件 THEN LEAVE 标号;
        END IF;
    END LOOP;
```

示例：使用函数实现求0~给定的任意整数的累加和

```
DROP FUNCTION IF EXISTS getTotal;
CREATE FUNCTION getTotal(maxNum INT(11))
RETURNS INT(11)
NO SQL
BEGIN
    DECLARE i INT(11) DEFAULT 0;
    DECLARE total INT(11) DEFAULT 0;
    WHILE i ≤ maxNum DO
         SET total = total + i;
         SET i = i + 1;
```

```
    END WHILE;
  RETURN total;
END

-- 调用函数
SELECT getTotal(100);
```

练习: 使用函数实现生成一个指定长度的随机字符串

思路:

1. 定义变量保存字符和数字组成字符串
2. 定义变量保存生成的字符串
3. 循环获取随机数，使用字符串截取的方式获得随机字符，并使用字符串拼接函数完成组装

第四节 触发器

1. 概念

触发器（trigger）是用来保证数据完整性的一种方法，由事件来触发，比如当对一个表进行增删改操作时就会被激活执行。经常用于加强数据的完整性约束和业务规则

2. 如何定义触发器

```
DROP TRIGGER [IF EXISTS] 触发器名称;
-- 创建触发器
-- 触发时机为BEFORE或者AFTER
-- 触发事件，为INSERT 、 UPDATE或者DELETE
CREATE TRIGGER 触发器名称 {BEFORE|AFTER} {INSERT|UPDATE|DELETE} ON 表名 FOR EACH ROW
BEGIN
-- 执行的SQL操作
END
```

3. 触发器类型

触发器类型	NEW和OLD的使用
INSERT触发器	NEW表示将要或者已经新增的数据
UPDATE触发器	OLD表示将要或者已经修改的数据，NEW表示将要修改的数据
DELETE触发器	OLD表示将要或者已经删除的数据

4. 触发器使用场景

场景一

现有商品表goods和订单表order，每一个订单的生成都意味着商品数量的减少，请使用触发器完成这一过程。

```sql
-- 如果存在增加订单的触发器，就将其删除掉
DROP TRIGGER IF EXISTS addOrder;
-- 创建触发器
CREATE TRIGGER addOrder AFTER INSERT ON `order` FOR EACH ROW
BEGIN
    UPDATE goods SET number = number - NEW.sale_count WHERE id=NEW.goods_id;
END

-- 测试代码
INSERT INTO `order` (`goods_id`, `sales_id`, `sale_count`, `created_time`, `state`)
VALUES (1, 1, 6, '2021-08-16', 1);
```

场景二

现有商品表goods和订单order，每一个订单的取消都意味着商品数量的增加，请使用触发器完成这一过程。

```sql
-- 如果存在取消订单的触发器，就将其删除掉
DROP TRIGGER IF EXISTS deleteOrder;
-- 创建触发器
CREATE TRIGGER deleteOrder AFTER DELETE ON `order` FOR EACH ROW
BEGIN
    UPDATE goods SET number = number + OLD.sale_count WHERE id=OLD.goods_id;
END

-- 测试代码
DELETE FROM `order` WHERE id=350001;
```

场景三

现有商品表goods和订单表order，每一个订单购买数量的更新都意味着商品数量的变动，请使用触发器完成这一过程。

```sql
-- 如果存在更新订单的触发器，就将其删除掉
DROP TRIGGER IF EXISTS updateOrder;
-- 创建触发器
CREATE TRIGGER updateOrder AFTER UPDATE ON `order` FOR EACH ROW
BEGIN
    DECLARE changeNum INT(11) DEFAULT 0;
    SET changeNum = NEW.sale_count - OLD.sale_count;
    UPDATE goods SET number = number - changeNum WHERE id=OLD.goods_id;
END

-- 测试代码
UPDATE `order` SET sale_count = sale_count + 2 WHERE id=20;
UPDATE `order` SET sale_count = sale_count - 4 WHERE id=20;
```

第五节 视图

1. 概念

视图是一张虚拟表，本身并不存储数据，当SQL操作视图时所有数据都是从其他表中查出来

2. 如何使用视图

```
-- 创建视图
CREATE VIEW 视图名称 AS SELECT 列1[,列2,…] FROM 表名 WHERE 条件;
-- 更新视图
CREATE OR REPLACE VIEW 视图名称 AS SELECT 列1[,列2,…] FROM 表名 WHERE 条件;
-- 删除视图
DROP VIEW IF EXISTS 视图名称;
```

3. 为什么使用视图

定制用户数据，聚焦特定的数据。例如：如果频繁获取销售人员编号、姓名和代理商名称，可以创建视图

```
-- 删除视图
DROP VIEW IF EXISTS salesInfo;
-- 创建或者更新视图
CREATE OR REPLACE VIEW salesInfo AS
SELECT
    a.id,
    a.`name` saleName,
    b.`name` agentName
FROM
    sales a,
    agent b
WHERE
    a.agent_id = b.id;

-- 测试代码
SELECT id, saleName FROM salesInfo;
```

简化数据操作。例如：进行关联查询时，涉及到的表可能会很多，这时写的SQL语句可能会很长，如果这个动作频繁发生的话，可以创建视图

```
DROP VIEW IF EXISTS searchOrderDetail;

CREATE OR REPLACE VIEW searchOrderDetail AS
SELECT
    a.id regionId,
    a.`name` regionName,
    b.id agentId,
    b.`name` agentName,
    c.id saleId,
    c.`name` saleName,
    d.sale_count saleCount,
    d.created_time createdTime,
    e.`name` goodsName
FROM
    region a,
    agent b,
```

```
    sales c,
    `order` d,
    goods e
WHERE
    a.id = b.region_id
AND b.id = c.agent_id
AND c.id = d.sales_id
AND d.goods_id = e.id;

-- 测试代码
SELECT * FROM searchOrderDetail;
```

提高安全性能。例如：用户密码属于隐私数据，用户不能直接查看密码。可以使用视图过滤掉这一字段

```
DROP VIEW IF EXISTS userInfo;
CREATE OR REPLACE VIEW userInfo AS
SELECT
    username,
    salt,
    failure_times,
    last_log_time
FROM
    `user`;

SELECT username, salt FROM userInfo;
```

注意：**视图并不能提升查询速度，只是方便了业务开发，但同时也加大了数据库服务器的压力，因此，需要合理的使用视图**