一、存储过程 `stored procedure`

1. 定义存储过程

示例1: 创建一个新表 `test_table` ，然后构建一个存储过程，实现把一定数量的数据插入到一个表中

示例2: 构建一个存储过程，实现把某个类别的产品数量写入到1个变量中

2. 查看存储过程

示例3：查看 `insert_many_row` 和 `sort_count_proc` 的定义

3. 修改与删除存储过程

示例4: 修改 `sort_count_proc` 定义

4. 存储过程的异常和错误处理

示例5：为错误状态定义名称

示例6：定义存储程序，往 `sort` 表插入数据行，如果插入成功，则设置会话变量为1；如果插入重复值，则设置会话变量为0.

二、 游标 `cursor`

基本语法

示例7: 定义存储过程，通过游标逐行更新某一类产品的所有产品的价格：如果大于1000，则上涨5%；否则，上涨10%。

练习1

三、触发器 `trigger`

1. 创建触发器

示例8: 定义触发器，实现对插入或者更新操作的以下约束：不允许 `instructor` 的薪水值高于150000

示例9：利用触发器实现 `sort` 表和 `subsort` 表之间的外键约束： `subsort` 表的 `sort_id` 参照 `sort` 表的 `sort_id` 。

2. 查看触发器

3. 删除触发器

练习2

四、事件调度器 `event scheduler`

1. 创建事件调度器语法

示例10：创建一个立即启动的事件。

示例11：创建一个每10秒执行的事件。

示例12：创建一个2020年5月19号起每天 `00:00` 执行的事件。

2. 查看事件

3. 修改事件

4. 删除事件

练习3

五、编程应用

1. 迭代查询：以前一次查询结果集作为条件，继续进行当前查询，直至没有新的结果集产生。

示例13：基于 `prereq` 表，查询 `course_id` 为 `'CS-10'` 的子孙课程。

2. 基于触发器实现日志表自动管理：当日志表发生变更时，检查日志表是否满足一定状态，如满足则对日志表进行进一步的处理。

示例14： `his_log` 表中的行不大于10000

3.基于用事件调度器实现定时管理日志表

示例15：每天凌晨定期清理30天前的 `his_log` 中的记录。

4. 基于存储过程和事件调度器实现定时检查外键约束。

   示例16：首先，定义存储过程，检查 `orders` 表的 `product_id` 都在 `product` 表中，如果不在，则将对应的行移动到 `orders_suspend` 表中（ `orders_suspend` 有 `orders` 所有字段，且有 `insert_time` 字段记录移动时间）。然后，定义事件，在每天00:00:00分执行该检查。

一、存储过程 `stored procedure`

对于 `SQL` 编程而言，存储对象是数据中的一个重要的对象，它是一组为了完成特定功能的 `SQL` 语句集，在经过一次编译后，再次调用就不需要重复编译，因此执行效率高。函数和存储过程有以下异同点

- 相同点：
  - 存储过程和函数的相同点在于，他们都是为了可重复地执行数据库 `SQL` 语句的集合，并且都是经过一次编译后可直接执行
- 不同点：
  - 语法中实现的标识符不同， `procedure` 和 `function`
  - 存储过程在创建时无返回值，而函数在定义时必须设置返回值
  - 存储过程没有返回值类型，且不能将结果直接赋值给变量；而函数定义时需要设置返回值类型，且在调用时必须将返回值赋值给变量
  - 存储过程必须通过 `CALL` 进行调用，不能在 `SELECT` 语句中调用；函数在 `SELECT` 语句中调用

1. 定义存储过程

```
-- 定义
create procedure 存储过程名(参数1, 参数2, ...)
[存储过程选项]
begin
存储过程语句块;
end;

-- 调用
call <procedure_name>(para1, ...);

-- 删除
drop procedure <procedure_name>;
```

参数: 每个参数由3部分组成，分别为输入输出类型、参数名称和参数类型， `[IN | OUT | INOUT] 参数名称 参数类型`

- `IN` 表示输入参数，即参数是在调用存储过程时传入到存储过程里面使用，传入的数据可以是直接数据，也可以是变量。
- `OUT` 表示输出参数，初始值为 `null` ，它是将存储过程中的值保持到 `out` 指定的参数中，返回给调用者。
- `INOUT` 表示既可输入也可输出的参数，即参数在调用时传入到存储过程，同时在存储过程中操作

之后，又可将数据返回给调用者。

存储过程选项:

- `language sql` ： 说明存储过程体由 `SQL` 语言组成
- `[not] deterministic` ： 存储过程的执行结果是否确定
  - `deterministic` :相同的输入对应相同的执行过程和结果
  - `not deterministic` :相同的输入可能得到不同的输出，默认情况
- `sql` 选项
  - `contains sql` ： 包含 `SQL` ，但不包含读或者写数据的语句，默认情况
  - `no sql` ： 不包含 `SQL`
  - `reads sql data` ： 包含读数据的 `SQL` 语句
  - `modifies sql data` ： 包含写数据的 `SQL` 语句
- `sql security`
  - `definer` ： 只有定义者有权执行
  - `invoker` ： 调用者可执行
- `comment  '注释'`

示例1: 创建一个新表 `test_table` ，然后构建一个存储过程，实现把一定数量的数据插入到一个表中

```
USE purchase;
CREATE TABLE test_table (id INT PRIMARY KEY AUTO_INCREMENT,
    a VARCHAR(10),
    b VARCHAR(10));
```

```
 DROP PROCEDURE IF EXISTS insert_many_rows;

 DELIMITER $$
 CREATE PROCEDURE insert_many_rows (IN loops INT)
     BEGIN
     DECLARE v1 INT;
     SET v1 = loops;
     WHILE v1 > 0 DO
         INSERT INTO test_table(a, b) VALUES ('qpq', 'rst');
         SET v1 = v1 - 1;
     END WHILE;
     END;
 $$
 DELIMITER ;
```

```
-- 调用insert_many_row
CALL insert_many_rows(100);
SELECT * FROM test_table;
```

示例2: 构建一个存储过程，实现把某个类别的产品数量写入到1个变量中

```
USE purchase;
DESC product;

delimiter $$
CREATE PROCEDURE sort_count_proc (IN v_sort_id VARCHAR(5), OUT v_product_count
INTEGER)
READS SQL DATA
BEGIN
  SELECT COUNT(*) INTO v_product_count
  FROM product
  WHERE sort_id = v_sort_id;
END;
$$
delimiter ;
```

```
-- 调用sort_count_proc
SET @v_sort_id = '11';
SET @v_product_count = 0;
CALL sort_count_proc(@v_sort_id, @v_product_count);
SELECT @v_product_count;
```

2. 查看存储过程

创建完存储过程后，可以使用 `MySQL` 专门提供的语句查看存储过程

```
-- 查看存储过程的创建语句
SHOW CREATE PROCEDURE <proc_name>

-- 根据指定的模式查看所有符合要求的存储过程
SHOW PROCEDURE STATUS [LIKE 匹配模式];

-- 直接在information_schema.routines中查询
SELECT *
FROM information_schema.routines;
```

示例3：查看 `insert_many_row` 和 `sort_count_proc` 的定义

```
SHOW CREATE PROCEDURE insert_many_rows;
SHOW CREATE PROCEDURE sort_count_proc;

SHOW PROCEDURE STATUS LIKE 'insert%';
SHOW PROCEDURE STATUS LIKE 'sort_count%';
```

3. 修改与删除存储过程

在 `MySQL` 中可以使用 `ALTER` 语句修改存储过程的特性，其基本的语法格式如下

```
ALTER PROCEDURE <proc_name> [properties];
```

上述语句中的特征指的是存储过程中需要修改的部分，注意， `ALTER PROCEDURE` 不能修改存储过程的参数或程序题。

- `properties` 选项有以下：
  - `comment`
  - `language sql`
  - `contain sql`
  - `no sql`
  - `reads sql data`
  - `modifies sql data`
  - `sql security defineer`
  - `sql security invoker`

示例4: 修改 `sort_count_proc` 定义

```
ALTER PROCEDURE sort_count_proc
SQL SECURITY INVOKER
COMMENT '统计某一个类别下的产品数量';
```

4. 存储过程的异常和错误处理

在存储过程执行过程中，可对某些特定的错误代码、警告或异常进行定义，然后针对这些错误添加处理程序进行进一步的处理。

- 自定义错误名称

```
DECLARE 异常名称 CONDITION FOR [错误类型];
```

在上述语法中，错误类型由两种可选值，分别为 `mysql_error_code` 和 `SQLSTATE[VALUE] sqlstate_value` 。前者是数值类型表示的错误代码，如 `1148` ；后者是5个字符长度的错误代码，如 `SQLSTATE '42000'` 。

示例5：为错误状态定义名称

```
DELIMITER $$
CREATE PROCEDURE exp_proc_1()
BEGIN
  DECLARE command_not_allowed CONDITION FOR SQLSTATE '42000';  -- SQLSTATE
END
$$
DELIMITER ;

-- 或者
DELIMITER $$
CREATE PROCEDURE exp_proc_2()
BEGIN
  DECLARE command_not_allowed CONDITION FOR 1148; -- MYSQL_ERROR_CODE
END
$$
DELIMITER ;
```

- 错误的处理程序

```
DECLARE 错误处理方式 HANDLER FOR 错误类型 [, 错误类型] ... 程序语句块;
```

错误类型 包括 `CONTINUE` （遇到错误不处理，继续执行），另一种是 `EXIT` （遇到错误时马上退出）； *程序语句段* 表示遇到定义的错误时，需要执行的存储过程代码块。 `FOR` 后的错误类型可选值有以下几种：

- `MYSQL_ERROR_CODE`
- `SQLSTATE`
- `SQLWARING` 表示所有以01开头的 `SQLSTATE` 代码
- `NOT FOUND` 表示所有以02开头的 `SQLSTATE` 代码
- `SQLEXCEPTION` 表示所有以01或02开头外的所有 `SQLSTATE` 代码

示例6：定义存储程序，往 `sort` 表插入数据行，如果插入成功，则设置会话变量为1；如果插入重复值，则设置会话变量为0.

> `SQLSTATE '23000'` 或者 `1062` 表示插入行时，表中已包含重复键，因此不能插入。

```
ALTER TABLE sort MODIFY sort_id CHAR(2) PRIMARY KEY;

DELIMITER $$
CREATE PROCEDURE proc_demo(v_sortid, v_sort_name)
BEGIN
  DECLARE CONTINUE HANDLER FOR SQLSTATE '23000'
    SET @is_success = 0; -- 如果遇到23000错误，则执行set @num=1，并继续之后的语句
  SET @is_success = 1;
  INSERT INTO sort(sort_id, sort_name) VALUES (v_sortid, v_sortname);
END
$$
DELIMITER ;

-- 分别执行两次，第一次成功;第二次未成功，但未报错，说明已经处理了异常。
call proc_demo('99', '其它');
select @is_success;
```

二、游标 `cursor`

在存储过程或自定义函数中的查询可能返回多条记录，可以使用游标来逐行读取查询结果分别处理。游标的使用包括声明游标、打开游标、使用游标和关闭游标。游标必须在处理程序之间声明，在变量和条件之后声明。游标用于标识数据的读取位置，可以和 `python` 中元组的下标对照理解。游标只能在存储过程或函数中使用。游标有以下特性：

- 只读的，即不能更新游标指向的结果集
- 不滚动的，即不能直接跳过一些行

基本语法

- 声明游标 `declare 游标名 cursor for select 语句`

使用 `declare` 语句声明游标后，此时与游标对应的 `select` 语句并没有执行， `mysql` 服务器内存中并不存在与 `select` 语句对应的结果集。

- 打开游标 `open 游标名`
  此时对应的 `select` 语句被执行， `mysql` 服务器内存中存在与 `select` 语句对应的结果集，此时结果集存储在临时表中。
- 从游标中提出数据 `fetch 游标名 into 变量名1, 变量名2, ...`
  每提取一条记录，游标移到下一条记录的开头。当取出最后一条记录后，如果再次执行 `fetch` 语句，则产生 `"ERROR 1329(02000):No data to fetch"` 。遇到该异常，终止读取游标程序。
- 关闭游标 `close 游标名`
  释放游标打开的数据集，以节省 `mysql` 服务器的内存空间。如果没有被明确关闭，则它将再被打开的 `begin-end` 语句块的末尾关闭。

示例7: 定义存储过程，通过游标逐行更新某一类产品的所有产品的价格：如果大于1000，则上涨5%；否则，上涨10%。

```
DROP PROCEDURE IF EXISTS update_price_proc;

delimiter $$
CREATE PROCEDURE update_price_proc (IN v_sort_name VARCHAR(20))
MODIFIES SQL DATA
BEGIN
  -- 定义变量
  DECLARE v_product_id INT;
  DECLARE v_price DECIMAL(8, 2);
  DECLARE state CHAR(20);
  -- 定义游标
  DECLARE price_cur CURSOR FOR
        SELECT product_id, price
        FROM product natural JOIN sort
        WHERE sort_name = v_sort_name;
  -- 定义异常处理: continue 发生错误继续运行， exit 发生错误终止程序
  DECLARE CONTINUE HANDLER FOR 1329 SET state = 'Error';

  OPEN price_cur; -- 打开游标
  REPEAT
    FETCH price_cur INTO v_product_id, v_price;  -- 移动游标，获取数据
      IF (v_price > 1000) THEN
        SET v_price = v_price * 1.05;
      ELSE
        SET v_price = v_price * 1.1;
      END IF;

      UPDATE product
      SET price = v_price
      WHERE product_id = v_product_id;
    UNTIL state = 'Error'  -- 如果没发生异常，则state为null；如果发生1329异常，state的值为error，此时终止repeat
  END REPEAT;

  CLOSE price_cur;  -- 关闭游标
```

```
35    END;
36    $$
37    delimiter ;
```

- 调用存储过程

```
1   SELECT product_id, price
2   FROM product natural JOIN sort
3   WHERE sort_name = '办公机器设备';
4
5   CALL update_price_proc('办公机器设备');
6   SELECT * FROM sort WHERE sort_name='办公机器设备';
```

练习1

1. 定义存储过程 `product_count_proc` ，输入参数 `v_sort_id` ，输出参数 `v_sort_count` ，语句块中查询给定类别为 `v_sort_id` 的产品数量，保存至 `v_sort_count` 。
2. 定义存储过程 `delete_expired_records_proc` ，无参数，语句块实现对 `operate_log` 30天前插入的记录的删除。

```
1   CREATE TABLE operate_log (id int primary key auto_increment,
2                             user_id varchar(50) not null,
3                             content varchar(255) not null default '',
4                             operate_time timestamp default current_timestamp());
```

3. 定义存储过程 `update_remark_proc` ，通过定义游标，逐行更新 `orders` 表中的价格 `remark` ：如果 `quantity<10` ，更新 `remark` 的值为 `'小批量订单'` ；如果 `quantity` 在10和50之间，更新 `remark` 的值为 `'中批量订单'` ；如果 `quantity>50` ，更新 `remark` 的值为 `'大批量订单'` 。

三、触发器 `trigger`

触发器的行为由数据操纵行为（例如插入、更新和删除）自动触发，因此一旦定义好，即可实现自动管理数据表。一些数据库的完整约束（如主键约束、外键约束和用户自定义约束等）可以基于触发器实现。触发器可用于跟踪用户对数据库的操作，审计用户操作数据库的语句，将用户的数据操纵写入预定的审计表。触发器可以同步实时地复制表中的数据。触发器可以自动计算数据值，并根据数据值进行特定的处理。

需要注意的是，在 `INNODB` 表上的触发器中的语句和触发语句是在同一个事务中完成的，所以它们执行的操作是原子的，触发语句和触发器操作会同时失败或成功。

对于具有相同触发器动作时间和事件的给定表，不能有两个触发器。例如，对于同一个表，不能有两个 `before update` 触发器，但可以有1个 `before update` 触发器和1个 `before insert` 触发器，或1个 `before update` 触发器和1个 `after update` 触发器。

`MySQL` 没有提供 `ALTER TRIGGER` 语句。如果需要修改，则应先 `DROP TRIGGER` ，然后重新定义 `CREATE TRIGGER` 。

1. 创建触发器

基本语法

```
CREATE TRIGGER 触发器名 触发时机 触发事件 ON 表名 FOR EACH ROW
BEGIN
  触发程序
END;
```

- 触发时间: `before / after` ，在事件发生前或后做触发程序
- 触发事件: `insert / update / delete` ，表上的触发事件
  - `insert` :将新行插入表时激活触发器，可以通过 `insert` ，`load data` 和 `replace` 触发
  - `update` :更新某一行时激活触发器，可以通过 `update` 或 `replace` 触发
  - `delete` :从表中删除某一行时激活触发器，可以通过 `delete` 或 `replace` 触发
- `for each row` :行级触发器(`mysql` 目前仅支持行级触发器，不支持语句级别的触发器,如 `create table`)
- 触发程序中的 `select` 语句不能产生结果集
- 触发程序中可以使用 `old` 和 `new` 关键字区别更新前后的行值
- `old` 是只读的，可以引用，但不能更改。在 `before` 触发程序中，可使用 `"set new.col_name = value"` 更改 `new` 值。但在 `after` 触发程序中，不能使用 `"set new.col_name = value"` .

示例8: 定义触发器，实现对插入或者更新操作的以下约束：不允许 `instructor` 的薪水值高于150000

```
USE PURCHASE;
CREATE TABLE `instructor` (`id` CHAR(5) PRIMARY KEY,
                           `name` VARCHAR(20),
                           `dept_name` VARCHAR(20),
                           `salary` DECIMAL(8, 2));

INSERT INTO instructor (id, `name`, dept_name, salary)
VALUES ('10101', 'Srinivasan', 'Comp. Sci.', '65000.00'),
       ('12121', 'Wu', 'Finance', '90000.00'),
       ('15151', 'Mozart', 'Music', '40000.00'),
       ('22222', 'Einstein', 'Physics', '95000.00'),
       ('32343', 'EI Said', 'History', '60000.00'),
       ('33456', 'Gold', 'Physics', '87000.00'),
       ('45565', 'Katz', 'Comp. Sci.', '75000.00'),
       ('58583', 'Califieri', 'History', '62000.00'),
       ('76543', 'Singh', 'Finance', '80000.00'),
       ('76766', 'Crick', 'Biology', '72000.00'),
       ('83821', 'Brandt', 'Comp. Sci.', '92000.00'),
       ('98345', 'Kim', 'Elec. Eng.', '80000.00');
```

- 分析：对于任意的数据操作，不允许更新后的薪水高于15000。
  - 插入一条或多条新记录，其中某行的薪水值高于15000
  - 修改一条或多条记录，其中修改后某行的薪水值高于15000

```
-- update
delimiter $$
create trigger instru_update_before_trigger before update on instructor for each
row
begin
    if (new.salary > 150000) then -- new为更新前的行值
        set new.salary = old.salary;  -- 如果高于150000，则重新更新为原来的值
        insert into mytable values(0); -- 因为mytable未经定义，触发异常，更新操作失败。
    end if;
end;
$$
delimiter ;
```

```
-- insert
delimiter $$
create trigger instru_insert_before_trigger before insert on instructor for each
row
begin
    if (new.salary > 150000) then
        insert into mytable values(0);  -- 因为mytable未经定义，触发异常，更新操作失败。
    end if;
end;
$$
delimiter ;
```

- 查看已经创建的触发器

```
show triggers; -- 查看触发器

SELECT *
FROM information_schema.triggers
WHERE trigger_name = 'instructor'; -- 查看instructor表中的所有触发器
```

- 根据插入条件触发

```
select * from instructor;

update instructor
set salary=185000
where id='10101';

insert into instructor
values('11111', 'Steve', 'Finance', 160000);

delete from instructor
where id='11111';
```

> 注意：
>
> - 触发器不能调用将数据返回客户端的存储过程，也不能使用采用 `CALL` 语句的动态 `SQL`

- 触发器不能使用以显式或隐式方式开始或接受事务的语句，如 `start stransaction` ， `commit` 或 `rollback` 。
- 使用 `old` 和 `new` 关键字可访问受触发器影响的列
  - 在 `insert` 触发器中，仅能使用 `new.col_name`
  - 在 `delete` 触发器中，仅能使用 `old.col_name`
  - 在 `update` 触发器中，可使用 `old.col_name` 来引用更新前的某一行的列， `new.col_name` 来引用更新后的某一行的列

示例9：利用触发器实现 `sort` 表和 `subsort` 表之间的外键约束： `subsort` 表的 `sort_id` 参照 `sort` 表的 `sort_id` 。

- 分析：需要分别定义从表和主表的更新和删除行为（ `before` ）
  - 主表：
    - 删除一行时，检查从表中是否存在参照值
    - 更新一行时，检查从表中是否存在参照行值，可进一步定义从表的参照值是否也对应更新（cascade）
  - 从表：
    - 新增一行时，检查主表是否存在被参照值，如果不存在，则插入失败。
    - 更新一行时，检查主表是否存在被参照值，如果不存在，则更新失败。

```
-- sort表上的更新，on update cascade
DROP TRIGGER IF EXISTS sort_update_after_trigger;

DELIMITER $$
CREATE TRIGGER sort_update_after_trigger AFTER UPDATE ON sort FOR EACH ROW -- 此处应该为after，不能为before，与subsort上的before trigger会冲突
BEGIN
  UPDATE subsort
  SET sort_id = new.sort_id
  WHERE sort_id = old.sort_id;
END;
$$
DELIMITER ;

-- sort表上的删除，on delete cascade
DROP TRIGGER IF EXISTS sort_delete_before_trigger;

DELIMITER $$
CREATE TRIGGER sort_delete_before_trigger BEFORE DELETE ON sort FOR EACH ROW
BEGIN
  DELETE FROM subsort
  WHERE sort_id = old.sort_id;
END;
$$
DELIMITER ;

-- subsort表上的插入
DROP TRIGGER IF EXISTS subsort_insert_before_trigger;
```

```
28
29  DELIMITER $$
30  CREATE TRIGGER subsort_insert_before_trigger BEFORE INSERT ON subsort FOR EACH
    ROW
31  BEGIN
32    SELECT COUNT(*) INTO @row_count
33    FROM sort
34    WHERE sort_id=new.sort_id;
35
36    IF (@row_count = 0) THEN
37      INSERT INTO mytable VALUES (0);
38    END IF;
39  END;
40  $$
41  DELIMITER ;
42
43  -- subsort表上的更新
44  DROP TRIGGER IF EXISTS subsort_update_before_trigger;
45
46  DELIMITER $$
47  CREATE TRIGGER subsort_update_before_trigger BEFORE UPDATE ON subsort FOR EACH
    ROW
48  BEGIN
49    SELECT COUNT(*) INTO @row_count
50    FROM sort
51    WHERE sort_id=new.sort_id;
52    IF (@row_count = 0) THEN
53      INSERT INTO mytable VALUES (0);
54    END IF;
55  END;
56  $$
57  DELIMITER ;
```

```
1   -- 如果subsort和sort表有外键约束，先删除
2   SHOW CREATE TABLE SORT;
3   SHOW CREATE TABLE SUBSORT;
4
5   SELECT * FROM SORT;
6   SELECT * FROM SUBSORT WHERE SORT_ID = 93;
7
8   -- 插入subsort
9   insert into subsort(subsort_id, subsort_name, sort_id)
10  values (9901, 'test', 93); -- 插入失败，  error code 1146
11
12  insert into sort (sort_id, sort_name)
13  values (93, 'test-sort'); -- 执行之后
14
15  insert into subsort(subsort_id, subsort_name, sort_id)
16  values (9901, 'test', 93); -- 插入成功
17
18  -- 更新sort
19  set sql_safe_updates=0;
```

```
20
21  update sort
22  set sort_id = 94
23  where sort_name = 'test-sort';
24
25  select * from sort;
26  select * from subsort where subsort_name = 'test';
27
28  -- 更新subsort
29  update subsort
30  set sort_id = 95
31  where subsort_name = 'test'; -- 执行失败, error 1146
32
33  -- 删除sort
34  delete from sort
35  where sort_name = 'test-sort';
36
37  select * from sort;
38  select * from subsort where subsort_name = 'test';
```

2. 查看触发器

```
1   SHOW TRIGGERS FROM purchase; -- 查看purchase中的触发器
2
3   SELECT *
4   FROM information_schema.triggers
5   WHERE EVENT_OBJECT_TABLE = 'sort';
6
7   SELECT *
8   FROM information_schema.triggers
9   WHERE EVENT_OBJECT_SCHEMA = 'purchase';
10
11  SHOW CREATE TRIGGER sort_update_after_trigger;
```

3. 删除触发器

```
1   DROP TRIGGER sort_update_after_trigger;
```

练习2

定义触发器 `move_product_records_trigger` ，实现以下功能：删除 `product` 表中的记录后，将被删除的记录插入到 `product_his` 中，其中 `product_his` 与 `product` 有完全相同的属性，且有 `insert_time` 记录插入时间。

四、事件调度器 `event scheduler`

事件调度器可以用作定时执行某些特定任务（如：删除记录、对数据进行汇总等），以取代原先只能由操作系统发起的计划任务。相比与 `Linux` 下的 `crontab` 或者 `Windows` 下的分级别任务计划， `MySQL` 的事件调度器可以精确到每秒钟执行一次任务。通常，我们会把复杂的 `SQL` 封装到一个存储过程中，这样事件在执行的时候只需要简单的 `CALL` 调用。

事件调度器又称为临时触发器，因为事件调度器是基于特定时间周期出发来执行任务的，而触发器是基于表上的数据操纵来执行任务。

1. 创建事件调度器语法

```
CREATE EVENT event_name
ON SCHEDULE schedule
[ON COMPLETION [NOT] PRESERVE]
[ENABLE | DISABLE]
[COMMENT '']
DO sql_statement;
```

- `event_name` ：必须是当前数据库中唯一的，同一个数据库不能有相同名称的event。
- `ON SCHEDULE` ： 计划
  - `AT 时间戳` ，用来完成在某一时刻单次执行某计划任务
  - `EVERY 时间的数量时间单位 [STARTAS 时间戳] [ENDS 时间戳]` ，用来完成重复执行的计划任务。
    - 时间戳可以是任意的 `TIMESTAMP` 和 `DATETIME` 数据类型，需要大于当前时间。
    - 时间的数值可以是任意非空的整数形式，时间单位是关键词：`YEAR,MONTH,DAY,HOUR,MINUTE或者SECOND`
- `ON COMPLETION` ：表示单次计划认为执行完之后或者当重复性的任务执行到了 `ENDS` 阶段。 `PRESERVE` 可以使事件在执行完毕后不会被 `DROP` 掉，建议使用该参数，以便于查看 `event` 具体信息。默认为 `ON COMPLETION NOT PRESERVE` 。
- `[ENABLE | DISABLE]` ：前者表示这个事件处于有效状态，即在指定时间将执行任务；后者表示这个事件处于失效状态，即在指定时间步执行任务。默认为 `ENABLE` 。
- `[COMMENT '']` ： 注释，最大长度为64字节
- `DO sql_statement` ： 事件需要执行的 `SQL` 语句或调用存储过程。这里 `SQL` 语句可以是单条或复条语句，多条语句写在 `BEGIN...END` 语句块。

```
USE PURCHASE;

SHOW VARIABLES LIKE '%event_scheduler%'; -- 如果为off，则执行下列语句开启事件调度器
SET GLOBAL event_scheduler = 1; -- 开启mysql事件调度器功能

CREATE TABLE demo_tb(id int primary key auto_increment,
                     name varchar(20),
                      insert_time timestamp default current_timestamp()); -- 示例表 demo_tb
```

示例10：创建一个立即启动的事件。

```
TRUNCATE demo_tb;
DROP EVENT IF EXISTS immediate_event;
DELIMITER $$
CREATE EVENT immediate_event
ON SCHEDULE AT now()
ON COMPLETION NOT PRESERVE
ENABLE
DO
BEGIN
  insert into demo_tb(name) values('demo');
END;
$$
delimiter ; -- 创建成功，执行完毕任务后，该事件被删除

SHOW EVENTS;

select * from demo_tb;
```

示例11：创建一个每10秒执行的事件。

```
DROP EVENT IF EXISTS interval_event;
DELIMITER $$
CREATE EVENT interval_event
ON schedule EVERY 10 SECOND
ON COMPLETION NOT PRESERVE
ENABLE
DO
BEGIN
  insert into demo_tb(name) values('demo_10S');
END;
$$
DELIMITER ;

SHOW EVENTS;
SHOW EVENTS FROM purchase; -- 查看purchase上的所有事件
SELECT * FROM information_schema.events; -- 查看所有事件
SHOW CREATE EVENT interval_event; -- 查看定义

SELECT * FROM demo_tb;
ALTER EVENT interval_event DISABLE; -- 临时关闭事件
```

示例12：创建一个2020年5月19号起每天 `00:00` 执行的事件。

```
DROP EVENT IF EXISTS repeat_event_from;
DELIMITER $$
CREATE EVENT repeat_event_from
ON SCHEDULE EVERY 1 DAY STARTS timestamp('2020-05-19 00:00:00')
ON COMPLETION PRESERVE
ENABLE
DO
BEGIN
  insert into demo_tb(name) values('demo_INTERAL_0000');
END;
$$
DELIMITER ;
```

```
SELECT timestamp('2020-12-21') + INTERVAL 1 DAY;
SELECT '2020-12-21 12:00:00' + INTERVAL 1 DAY;
SELECT DATE_ADD('2020-12-21 00:00:01', INTERVAL 1 DAY);
```

> `SCHEDULE` 示例
>
> - `EVERY 1 HOUR STARTS current_timestamp() ENDS current_timestamp()+interval 1 day` 截止明天这个时间点，每小时执行一次。
> - `AT current_timestamp() + interval 30 second` 30秒后执行一次事件
> - `EVERY 1 DAY STARTS DATE_ADD(CURDATE(), INTERVAL 1 DAY)` 从明天 `00:00` 起，每天重复执行一次时间

2. 查看事件

```
SHOW EVENTS FROM purchase;  -- 查看purchase数据库中的所有事件

SELECT *
FROM information_schema.events; -- 查看所有事件

SHOW CREATE EVENT interval_event; -- 查看定义
```

3. 修改事件

```
ALTER EVENT repeat_event_from DISABLE;
ALTER EVENT repeat_event_from ON COMPLETION NOT PRESERVE;
```

4. 删除事件

```
DROP EVENT IF EXISTS repeat_event_from;
```

练习3

创建事件 `check_product_event` ，每天凌晨执行，检查 `product` 中的记录，如果对应的 `sort_id` 为空或者 `sort_id` 不在 `sort` 表中，则删除对应记录。

五、编程应用

1. 迭代查询：以前一次查询结果集作为条件，继续进行当前查询，直至没有新的结果集产生。

- 创建 `prereq` 表

```
USE purchase;
CREATE TABLE prereq (course_id varchar(30) primary key,
                     prereq_id varchar(30));

INSERT INTO prereq(course_id, prereq_id)
VALUES ('BIO-301', 'BIO-101'),
('BIO-399', 'BIO-101'),
('CS-190', 'CS-101'),
('CS-315', 'CS-101'),
('CS-319', 'CS-101'),
('CS-347', 'CS-101'),
('EE-181', 'PHY-101'),
('CS-101', 'CS-10'),
('CS-10', 'CS-1');
```

示例13：基于 `prereq` 表，查询 `course_id` 为 `'CS-10'` 的子孙课程。

```
-- mysql8.0以上版本可以用with recursive实现迭代查询
-- 查询课程号'CS-10'机器所有的子课程
with recursive tree_course(course_id, prereq_id, lev) as (
  select course_id, prereq_id, 0 as lev
  from prereq
  where course_id = 'CS-10'
  union all
  select s.course_id, s.prereq_id, p.lev + 1 as lev
  from tree_course p join prereq s on p.course_id = s.prereq_id)
select * from tree_course;

select * from prereq;
```

```
-- mysql5.7之前的版本
DROP PROCEDURE IF EXISTS purchase.tree_prereq_proc;

DELIMITER $$
CREATE PROCEDURE tree_prereq_proc(v_course_id varchar(30))
MODIFIES SQL DATA
BEGIN
  DECLARE v_lev INT DEFAULT 0;
  DROP TEMPORARY TABLE IF EXISTS tree_prereq; -- 要点1
  CREATE TEMPORARY TABLE tree_prereq(course_id varchar(30),
                   prereq_id varchar(30),
                   lev int); -- 用于存储结果
  REPEAT
```

```
    INSERT INTO tree_prereq(course_id, prereq_id, lev)
      SELECT course_id, prereq_id, v_lev as lev
      FROM prereq
      WHERE course_id = v_course_id; -- 首轮节点对应的子节点

    WHILE row_count() > 0 DO -- 要点2
      DROP TEMPORARY TABLE if exists temp;
      CREATE TEMPORARY TABLE temp -- 要点3
        SELECT * FROM tree_prereq WHERE lev = v_lev;
      SET v_lev = v_lev + 1; -- 更新层级
      INSERT INTO tree_prereq(course_id, prereq_id, lev)
        SELECT distinct s.course_id, s.prereq_id, v_lev as lev
        FROM temp p JOIN prereq s ON p.course_id = s.prereq_id; -- 要点4
    END WHILE;
  END;
  $$
  DELIMITER ;

  --
  CALL tree_prereq_proc('CS-10');
  SELECT * FROM tree_prereq;
```

要点

1. 初始化结果集：存储过程中的数据定义语句直接提交，需先删除已经存在的 `tree_prereq` ，否则建表执行不成功
2. `row_count()` 可得到前一个 `SQL` 更新语句影响到的行数，因此，若 `INSERT` 语句没有新行插入到 `tree_prereq` 表中，则 `row_count()=0` ，跳出循环。类似的， `found_rows()` 可得到前一个 `SQL` 查询语句的返回行数
3. 创建当前轮循环的种子行
4. `temp p JOIN prereq s ON p.course_id = s.prereq_id` 其中 `temp` 中的行作为父节点表， `prereq` 中的行作为子节点表。

- 练习：
  - 基础：请基于存储过程查询 `CS-190` 的所有先行课程。
  - 进阶：请基于存储过程、临时表、预处理语句等技术构建通用迭代查询程序，即输入 `表名称、节点编号列、父节点编号列、初始节点编号` ，得到关于该初始节点编号下的 `所有子孙节点` 。

2. 基于触发器实现日志表自动管理：当日志表发生变更时，检查日志表是否满足一定状态，如满足则对日志表进行进一步的处理。

示例14： `his_log` 表中的行不大于10000

```
  USE PURCHASE;
  CREATE TABLE his_log(id int primary key auto_increment,
                       userid char(50) not null,
                       operate_time timestamp default current_timestamp());

```

```
CREATE TRIGGER his_log_constant_rows_trigger BEFORE INSERT ON his_log FOR EACH ROW
BEGIN
  SELECT COUNT(*) INTO @num_rows
  FROM his_log;
  IF (@num_rows >= 10000) THEN
    DELETE FROM his_log
    ORDER BY operate_time LIMIT 1;
  END IF;
END;
```

3.基于用事件调度器实现定时管理日志表

示例15：每天凌晨定期清理30天前的 `his_log` 中的记录。

```
DELIMITER $$
CREATE EVENT delete_log_event
ON SCHEDULE EVERY 1 DAY STARTS CURDATE() + INTERVAL 1 DAY
ON COMPLETION PRESERVE
ENABLE
DO
BEGIN
  DELETE FROM his_log
  WHERE DATE_ADD(operate_time, INTERVAL 30 DAY) < CURRENT_TIMESTAMP();
END;
$$
DELIMITER ;
```

4. 基于存储过程和事件调度器实现定时检查外键约束。

> 在日常管理（如订货管理）中，存在一天某一段时间（如9:00至22:00）频繁有对表的写入、更新和读取操作，如果直接定义该表上的外键约束，则在变更该表记录时也会同时检查变更之后是否满足外键约束，从而影响到数据库的效率。而在一些时间段（如00:00至6:00）则只有极少量的变更和读操作。因此，可以结合利用事件和存储过程，在低频变更和读取表数据时间段检查该表相关的外键约束，实现动态表管理。

示例16：首先，定义存储过程，检查 `orders` 表的 `product_id` 都在 `product` 表中，如果不在，则将对应的行移动到 `orders_suspend` 表中（`orders_suspend` 有 `orders` 所有字段，且有 `insert_time` 字段记录移动时间）。然后，定义事件，在每天00:00:00分执行该检查。

```
USE purchase;
-- 定义表orders_suspend
CREATE TABLE orders_suspend SELECT * FROM orders WHERE 1 = 0;
ALTER TABLE orders_suspend ADD insert_time TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP();

-- 定义存储过程check_orders_ref_proc: 方法1
DELIMITER $$
```

```
CREATE PROCEDURE check_orders_ref1_proc()
MODIFIES SQL DATA
BEGIN
  DECLARE v_diff INT DEFAULT 0;
  -- 检查是否有不符合外键约束的记录
  SELECT COUNT(*) INTO v_diff
  FROM orders a LEFT JOIN product b ON a.product_id = b.product_id
  WHERE b.product_id IS NULL;
  -- 如果有，则转移问题记录
  IF (v_diff > 0) THEN
    -- 将不符合外键约束的行移至orders_suspend
    INSERT INTO orders_suspend (order_id, product_id, quantity, user_name, order_date, consignee, delivery_address, phone, email, remark, insert_time)
    SELECT a.order_id, a.product_id, a.quantity, a.user_name, a.order_date, a.consignee, a.delivery_address, a.phone, a.email, a.remark, current_timestamp()
    FROM orders a LEFT JOIN product b ON a.product_id = b.product_id
    WHERE b.product_id IS NULL;
    -- 删除orders表中不符合外键约束的行
    DELETE FROM orders
    WHERE product_id NOT IN (SELECT product_id FROM product);
  END IF;
END;
$$
DELIMITER ;

-- 定义存储过程check_orders_ref_proc: 方法2，保存中间结果，更加高效
DELIMITER $$
CREATE PROCEDURE check_orders_ref2_proc()
MODIFIES SQL DATA
BEGIN
  -- 创建临时表orders_temp，保存不符合外键约束的中间结果
  DROP TEMPORARY TABLE IF EXISTS orders_temp;
  CREATE TEMPORARY TABLE orders_temp
  SELECT COUNT(*) INTO v_diff
  FROM orders a LEFT JOIN product b ON a.product_id = b.product_id
  WHERE b.product_id IS NULL;
  -- row_count()返回上一次数据操纵语句的影响行数，found_rows()返回上一次查询的返回行数
  IF (row_count() > 0) THEN
  -- 将不符合外键约束的行移至orders_suspend
    INSERT INTO orders_suspend (order_id, product_id, quantity, user_name, order_date, consignee, delivery_address, phone, email, remark, insert_time)
    SELECT order_id, product_id, quantity, user_name, order_date, consignee, delivery_address, phone, email, remark, current_timestamp()
    FROM orders_temp;
    -- 删除orders表中不符合外键约束的行
    DELETE FROM orders
    WHERE product_id IN (SELECT product_id FROM orders_temp);
  END IF;
END;
$$
DELIMITER ;

```

```
-- 定义事件repeate_move_orders_event
DELIMITER $$
CREATE EVENT repeate_move_orders_event
ON SCHEDULE EVERY 1 DAY STARTS DATE_ADD(curdate(), INTERVAL 1 DAY)
ON COMPLETION PRESERVE
ENABLE
BEGIN
  CALL check_orders_ref_proc2();
END;
$$
DELIMITER ;
```