语句在sysql中不区分大小写，注意区分关键字

目录

视图/存储过程/触发器

视图

- 介绍
  视图是一种虚拟存在的表，视图中的数据并不在数据库中实际存在，行和列的数据来自定义视图的查询中使用的表，并且实在是使用视图动态生成的。
  也就是视图只是保存了查询SQL逻辑，而不是一个静态的结果。
- 创建

```
create [or replace] view 视图名称 [(列名列表)] as select语句 [with | cascaded | local | check option]
```

- 查询

  查看创建视图语句

```
show create view 视图名称;
```

  查看视图数据

```
select * from 视图名称...;
```

- 修改
  创建语句中的第一条，但是需要加入or replac

```
create or replace view 视图名称 [(列名列表)] as select语句 [with | cascaded | local | check option]
```

  方法二：

```
alter view 视图名称 [(列名列表)] as select语句 [with | cascaded | local | check option]
```

- 删除

```
drop view [if exists] 视图名称 [,视图名称] ...
```

- *插入新数据，对于视图来说，他不是一个真正的表，对视图插入数据的语法会在视图本身所依赖的表中实现*

视图的检查选项

1. 当使用with check option子句创建视图时，mysql会通过视图检查正在更改的每个行，例如 **插入**，**更新**，以确保符合视图的定义。
2. mysql允许基于另一个视图创建视图，它会检查依赖图中的规则以保持一致性。

为了确定检查的范围，mysql提供了两个选项: cascaded 和 local，默认值为cascaded

没有with check option语句

> 不会考察添加的数据 是否符合where范围

cascaded

> 当使用with cascaded check option 时，他是要满足自身和父类的视图要求。(即使父类没有with check option)
> 而如果时不加入cascaded，只需要满足自身的要求。

local

> 和上面cascaded唯一的不同地方在于
> 他需要满足自身和父类的要求，但是如果父类没有with check option，那么这个父类的要求就不考察。

视图的更新

要使得视图可以更新，则视图中的行与基础表中的行之间必须存在一对一的关系，视图中有如下情况则不可以更新。

换句话来说，是指没办法去修改视图中的值，你想要修改的话必须修改原来的表这样才可以修改视图中值

- 聚合函数或窗口函数
- distinct
- group by
- having
- union 或者union all

视图的作用

- 简单
  简化了用户的操作，那些经常使用的查询可以被定义为视图，从而使得用户不必为以后的操作每次指定全部的条件。
- 安全
  数据库可以授权，但是不能授权到特定表中的特定行和列，通过视图用户只能查询和修改他们所能见到的数据。
- 数据独立

感觉就类似一个自己写好的封装show函数

视图的存储过程

存储过程的概念等同于函数

- 概念：存储过程就是数据库SQL语言层面的代码封装与重用
- 特点：
  1. 封装，复用
  2. 可以接受参数，也可以返回数据
  3. 减少网络交互，效率提升
- 创建

```
create procedure 储存过程名称([参数列表])
begin
  -- SQL语句
END;
```

在命令行中分号就是结束符，所以创建的语法在命令行需要其他方式来设置为结束符，关键字delimiter来指定SQL语句的结束符

下面这个使用$$作为结束符的例子，记得在设置存储过程之后，再次使用该语法将结束符设置回;

```
delimiter $$
```

- 调用

```
call 名称 ([参数]);
```

- 查看
  查询指定数据库的储存过程及状态信息

```
select * from information_schema.ROUTINES  where roution_schema = 'xxx';  --
```

查询某个存储过程的定义

```
show create procedure 存储过程名称;
```

- 删除

```
drop procedure [if exists] 存储过程名称;
```

系统变量

- **系统变量** 是mysql服务器提供， 不是用户定义的， 属于服务器层面。 分为全局变量(global), 会话变量(session)
  每创建一个会话(操作台), 里面的会话变量互相不干涉，全局变量对所有会话都有效

查看

- 查看所有系统变量

```
show [session | global] variables;
```

- 通过like模糊匹配方式查找变量

```
show [session | global] variables like '...' ;
```

- 查看指定变量的值

```
select @@[session. | global.] 系统变量名;
```

以上语句默认值为session

- 设置系统变量

```
1. set [session | global] 系统变量名 = 值;

2. set @@[session | global] 系统变量名 = 值;
```

服务器重新启动之后会将全部变量恢复成默认值，如果想要永久更改，需要到配置文件中修改

用户变量

- **用户定义变量** 是用户根据需要自己定义的变量，用户变量不用提前声明，再用的时候直接用“@变量名”使用就可以。其作用域为当前会话。
- 赋值 建议使用 := 来赋值 也就是方法2

```
1. set @var_name = expr [,@var_name = expr]...;

2. select @var_name := expr [,@var_name := expr]...;

3. select 字段名 into @var_name from 表名;
```

- 使用

```
select @var_name;
```

用户变量不需要提前声明，如果使用的时候没有对应的值返回的是null

局部变量

- **局部变量** 是根据需要定义在局部生效的变量，访问之前，需要declare声明。可用作存储过程内的局部变量和输入参数，局部变量的范围实在其内声明的begin...end块。
- 声明

```
declare 变量名 变量类型 [default ...];
```

变量类型就是数据库的字段类型

- 赋值

```
1. set 变量名 := 值;

2. select 字段名 into 变量名 from 表名...;
```

条件判断 if

- 语法

```
if 条件1 then
  ...
elseif 条件2 then  -- 可选
  ...
else   -- 可选
  ...
end if;
```

参数

类型	含义	备注
int	作为输入参数 == 传入值	默认
out	作为输出参数 == 返回值	
inout	上述两种的结合	

- 用法

```
create procedure 存储过程名 ([in/out/inout] 参数名 参数类型)
begin
  -- sql 语句
end;
```

举例

```
create procedure p3(in score int , out result varchar(10))
begin

    if score >= 82 then
        set result := '优秀';
    else
        set result := '普通';
```

```
    end if;

end;

call p3(90, @myp3result);

select @myp3result;
```

case

- 语法一

  ```
  case value1
    when value_n then 满足条件之后执行的语句
    ...
    else 都不满足的情况下所执行的语句
  end case;
  ```

  每一步判断value1和value_n是否相等，相等执行后面语句

- 语法二

  ```
  case
    where 条件表达式一 then 满足条件一之后的执行语句
    where 条件表达式二 then 满足条件二之后的执行语句
    ...
    else 都不满足的情况下所执行的语句
  end case;
  ```

while

满足条件之后才执行循环体中的SQL语句

```
while 条件 do
  -- sql语句块
end while;
```

如果想要类似C语言中while(1)的效果 可以写while true do

repeat

repeat是先执行一次语句块，然后判定逻辑是否满足，如果满足，则退出，如果不满足则继续下一次循环。

```
repeat
  -- sql语句块
  until 条件
end repeat;
```

loop

loop是一种死循环，需要配合下面两个语句使用；

- leave：退出循环
- iterate：必须用在循环中，作用是跳过当前循环剩下的语句，直接进入下一次循环

```
[循环名称:] loop
  -- sql语句块
end loop [循环名称];
```

举例

```
-- loop
create procedure p4(in n int)
begin

    declare tot int default 0;

    sum:loop
        -- 实现从n加到0
        if n<=0 then
            leave sum;
        end if;

        -- 实现遇到奇数跳过
        if n%2 = 1 then
          set n := n - 1;
          iterate sum; -- 等同与continue;

        -- 赋值用 := 比较好
        -- 在mysql中 = 可以是条件判断符号
        set tot := tot + n;
        set n := n - 1;

    end loop sum;

    -- 展示结果
    select tot;

end;

call p4(100);   -- 调用函数展示结果
```

游标

- **游标** 是用来存储查询结果集的数据类型。在存储过程和函数中可以使用游标对结果集进行循环处理。游标的使用包括游标的声明、open、fetch、close，其语法分别如下。
- 声明游标

```
declare 游标名称 cursor for 查询语句;
```

- 打开游标

```
open 游标名称;
```

- 获取游标记录

```
fetch 游标名称 into 变量[,变量 ];
```

- 关闭游标

```
close 游标名称;
```

- 举例

```

-- 根据传入的参数 mut 来查询用户表 student 中，所有id编号小于等于mut的用户姓名和学号
-- 并将用户的姓名和学号插入到所创建的一张新表中。

-- 逻辑
-- 声明游标，存储查询结果集
-- 准备：  创建好表结构
-- 开启游标
-- 获取游标中的记录
-- 插入数据到新表中
-- 关闭游标

create procedure p10(in mut int)
begin
    -- 声明获取游标中的数据的变量，必须在声明游标之前声明
    declare uname varchar (20);
    declare unum varchar (20);
    -- 声明游标,寻找id编号小于mut的信息的name和num信息
    declare u_stu cursor for select name , num from student where id <= mut;

    -- 如果有这个表了先删除
    drop table if exists u_stu_pro;
    -- 创建表结构
    create table if not exists u_stu_pro(
        -- id是主键并且自增
```

```
        id int primary key auto_increment,
        name varchar(20),
        xuehao varchar(20)
    );

    -- 打开游标
    open u_stu;
    -- 实现插入表格中的逻辑
    while true do
        -- 获取信息
        fetch u_stu into uname,unum;
        -- 插入信息
        insert into u_stu_pro values (null, uname, unum);
    end while;
    -- 关闭游标
    close u_stu;

end;

call p10(4);
```

在上述的程序运行中有一个问题在于while的条件是true，那么这个程序会一直抓取数据直到无数据可以抓取，提示报错信息为[02000][1329] No data - zero rows fetched, selected, or processed，为了解决这个问题，我们引入条件处理程序的解决

条件处理程序

条件处理程序 可以用来定义在流程控制结构执行过程中遇到问题时相应的处理步骤。
在遇到这个提前考虑到的报错信息时，程序会直接执行程序员写好的对应处理方案

- 语法

```
declare HANDLER_ACTION handler for CONDITION_VALUE [,CONDITION_VALUE] ... 遇到该情况后你还想执行的语句

HANDLER_ACTION
1. continue  -- 继续执行当前程序
2. exit   --终止执行当前的程序

CONDITION_VALUE
1. sqlstate 报错编号  -- 状态码，比如02000对应没有数据可以继续抓取
2. sqlwarning  --所有以01开头的sqlstate报错编号的简写
3. not found -- 所有以02开头的sqlstate报错编号的简写
4. sqlexception  -- 除了上述2和3情况的其他所有报错编号的简写
```

- 如果想要知道更多报错信息可以参考官方文档 https://dev.mysql.com/doc/mqsql-errors/8.0/en/server-error-reference.html
- 举例

```
-- 和上一个例子基本一致，唯一不同在于解决了while语句的报错信息

create procedure p11(in mut int)
begin
    -- 声明获取游标中的数据的变量，必须在声明游标之前声明
    declare uname varchar (20);
    declare unum varchar (20);
    -- 声明游标,寻找id编号小于mut的信息的name和num信息
    declare u_stu cursor for select name , num from student where id <= mut;

    -- 条件程序处理，遇到这个报错编号时直接执行这个语句
    declare exit handler for sqlstate '02000' close u_stu;

    -- 如果有这个表了先删除
    drop table if exists u_stu_pro;
    -- 创建表结构
    create table if not exists u_stu_pro(
        -- id是主键并且自增
        id int primary key auto_increment,
        name varchar(20),
        xuehao varchar(20)
    );

    -- 打开游标
    open u_stu;
    -- 实现插入表格中的逻辑
    while true do
        -- 获取信息
        fetch u_stu into uname,unum;
        -- 插入信息
        insert into u_stu_pro values (null, uname, unum);
    end while;
    -- 关闭游标
    close u_stu;

end;

call p11(3);
```

存储函数

- 存储函数是有返回值的存储过程，存储函数的参数只能是in类型的。
- 语法

```
create function 存储函数名称([参数列表])
returns type CHARACTERISTIC
begin
  -- sql语句
  return...;
end;

CHARACTERISTIC说明:
1. deterministic  -- 相同的输入参数总是产生相同的结果
2. no sql  -- 不包含sql语句
3. reads sql data  -- 包含读取数据的语句，但不包含写入数据的语句
```

- 举例

```
create function fun1(n int)
-- 存储函数必须要有返回值
returns int deterministic
begin
    declare tot int default 0;

    while n>0 do
        set tot := tot + n;
        set n := n -1;
    end while;

    return tot;
end;

select fun1(100); -- 储存函数有返回值所以这里直接展示
```

存储函数比较少用，因为存储过程都包含了存储函数的所有功能，而存储函数的限制要比存储过程更大（必须要有返回值等）

触发器

- 概念

> 触发器是与表有关的数据库对象，指 在insert/update/delete 之前或者之后，出发并执行触发器中定义的sql语句集合。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old和new来应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mysql中触发器还只支持行级触发，不支持语句级触发

行级触发指的是每一次调用insert等命令都会触发一次，语句级指的是一个总体的语句块不管insert等命令有几次，我的触发器只触发一次

触发器类型	new和old
insert型	new表示将要或者已经新增的数据
update型	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	old表示将要或者已经删除的数据

语法

- 创建

```
create trigger 触发器名称
before/after insert/update/delete
on tbl_name for each row -- 行级触发器
begin
  -- 触发器语句
end;
```

- 查看

```
show triggers;
```

- 删除

```
drop trigger [schema_name.]触发器名称;  -- 如果没有指定schema_name，默认当前数据库
```

- insert型触发器举例

```
-- insert触发器
-- 需求：通过触发器记录student表的数据变更日志(student_logs)，包含增加，修改，删除；

-- 准备工作：  日志表的创建
create table student_logs(
    id int (11) not null auto_increment,
    operation varchar(20) not null comment '操作类型，insert/update/dalete',
    operate_time datetime not null comment '操作时间',
    operate_id int(11) not null comment '操作id',
    operate_params varchar(500) comment '操作参数',
    primary key (`id`)
)engine=innodb default charset=utf8mb4;

-- 插入数据触发器
create trigger student_insert_trigger
    -- 在向表格student插入数据后触发
    after insert on student for each row
begin
```

```
    insert into student_logs(id, operation, operate_time, operate_id, operate_params)
values
        (null,'insert',now(),new.id,concat('插入的数据内容为: id=',new.id,'
,name=',new.name,', num=',new.num,', time=',new.time));

end;

-- 查看触发器
show triggers ;

-- 删除触发器
drop trigger student_insert_trigger;

-- 插入数据到student，并观察是否有更新日志
insert into student(id, name, num, time) VALUES (7,'胡七', '201804020288','2022-07-14');
```

- 更新型触发器举例

```
-- 修改数据的触发器

create trigger student_update_trigger
    -- 在向表格student更新数据后触发
    after update on student for each row
begin
    insert into student_logs(id, operation, operate_time, operate_id, operate_params)
        values (null,'update',now(),new.id,
            concat('更新之前的数据内容为: id=',old.id,' ,name=',old.name,',
num=',old.num,', time=',old.time,
                ' || 更新之后的数据内容为: id=',new.id,' ,name=',new.name,',
num=',new.num,', time=',new.time));
end;

-- 更新数据，并观察是否有更新日志
update student set time = '2023-07-03' where id = 6;

update student set num = '20190202011' where id = 1;
```

- 删除型触发器举例

```
-- 删除数据的触发器

create trigger student_delete_trigger
    after delete on student for each row
begin
    insert into student_logs(id, operation, operate_time, operate_id, operate_params)
values
        (null,'delete',now(),old.id,
```

```
            concat('删除之前的数据内容为: id=',old.id,' ,name=',old.name,',
num=',old.num,', time=',old.time));
end;

show triggers;

-- 删除语句，观察是否成功更新记录
delete from student where id = 8 ;
```