数据库_SQL

数据库语言

- DDL：数据库定义语言

  用来创建数据库中的各种对象：表、视图、索引、同义词、集群等

  关键字有：CREATE TABLE（表）、VIEW（视图）、INDEX（索引）、SYN（同义词）、CLUSTER（集群）

- DQL：数据库查询语言

  由 select 子句、from 子句、where 子句 构成

  ```
  select <字段列表> from <表 / 视图> where <查询条件>
  ```

- DML：数据库操纵语言

  ```
  # 1 ：插入
  insert into <表>( <字段列表> ) values( <字段值列表> )

  # 2 ：更新
  update <表> set <字段>=<字段值> where <查询条件>

  # 3 ：删除
  delete from <表> where <查询条件>
  truncate table <表>  # 清空表
  # 注意：drop table table_name 用于删除表！是 DDL 指令！
  ```

- DCL：数据库控制语言

  用于授予 或 回收访问数据库的某种特权，并控制数据库操纵事务发生的时间和效果！

  ```
  # 1 ：授权
  grant

  # 2 ：事务回滚：是数据库状态会到事务提交之前的状态！
  rollback;

  # 3 ：事务提交：显示提交、隐式提交、自动提交
      # 1）：显示执行 commit 提交
      commit;

      # 2）：隐式提交：使用 sql 命令默认会提交事务
      alter、comment 、connect 、create 、drop 、exit 、grant 、quit 、remove 、
  rename

      # 3）：自动提交：将 autocommit 设置为 on ，则插入、修改、删除语句执行之后，系统自动
  提交！
      set autocommit on;
  ```

条件过滤

- WHERE 子句判断条件！

范围过滤 vs 精确过滤

- 操作符

操作符	说明
=	等于
<	小于
>	大于
<> 或 !=	不等于
<= 或 !>	小于等于
>= 或 !<	大于等于
BETWEEN AND	在两个值之间
IS NULL	为 NULL 值

- 注意：0、NULL、空字符串 三者各不相同！

AND 、OR

- 可使用 AND 和 OR 连接多个条件。

  AND 优先级比 OR 高，当一个过滤表达式中涉及多个 AND 和 OR 时，应该使用 () 决定优先级！

IN

- IN 操作符 用于 匹配一组值，其后也可以接一个 select 子句，表示 匹配 子查询的结果！

```
# 查询 id 为 2、3、4、5 的记录
select <字段列表> from table_1 where id in (2,3,4,5);

# 根据子查询，匹配table_1 的记录
select <字段列表> from table_1 where id in (select <某一列> from table_2);
```

NOT

- NOT 表示否定一个条件

模糊匹配

- like 搭配 通配符：

```
<!--
 · 通配符
    _ 匹配 1 个任意字符
    % 匹配 0个或者多个任意字符
-->
```

```
# 匹配：col_1 以 a开头的两位字符 并且 col_2 以 "ab" 为前缀
select <字段列表> from <表> where col_1 like "a_" and col_2 like "ab%";
```

- REGEXP 搭配 正则表达式

通配符 和 正则表达式：sql 查询

查询

distinct 查询

- 去重查询：作用于所 select 的所有字段，只有当所有字段的值都相同才会被认为相同！

```
select distinct <字段列表> from <表>;
```

limit 查询

- 限制返回的 行数

```
select <字段列表> from <表> limit line_total;
```

- 可以有两个参数：param1 为起始行（从 0 开始），param2 为返回的总行数！

  用作分页查询

```
select <字段列表> from <表> limit num_st, num_total;
```

排序

- ORDER BY
  - ASC：升序（默认）
  - DESC：降序

```
# 按照 col_1 升序排序，如果相同，则按照 col_2 降序排序
select <字段列表> from <表> order by col_1 ASC,col_2 DESC;
```

- order by 永远是 最后的一项参数！

分组 查询

- 以指定的列为标准，将列值相同的所有记录 分为一组，也成为 聚集 查询！

  可以对同一组的数据使用汇总函数进行处理，例如：汇总每一组的数量，求每组的平均值 ... ...

```
# 按照 col 分组，相同 col 的记录会被划分到同一组中，count(*) 会计算 每组的记录总数（包括 NULL ）
select col,count(*) from table_1 group by col;

# count(col_2) 会按照每组内的 col_2 记录总行数，不包括 col_2 为 NULL 的行！
select col,count(col_2) from table_1 group by col;
```

- group by 分组，会自动以 按照分组字段 进行组间排序！

  也可以使用 order by 按照 组的大小 进行排序！

```
# 按照 col 分组，组间使用 total 升序排序
select col,count(*) total from table_1 group by col order by total;
# count(*) total 为 count(*) 列起别名，也可以使用 as：count(*) as total
```

- 分组过滤：having

where 用于过滤行，而 having 用于过滤 分组：group by 只能和 having 搭配，having 也只能和 group by 搭配！

如果在分组之前需要过滤某些行，则应该在 group by 之前使用 where 过滤！分组之后，只能使用 having 过滤！

所以：group by 总是在 where 之后，在 order by 之前！

```
# 按照 col 分组，分组之前过滤 id<=5 的行；
# 分组之后，按照 组的大小 过滤 total>=10 的组！
# 最后按照 组的大小 升序排序！
select col,count(*) AS total
from table_1 where id>5
group by col HAVING total<10
order by total;
```

- 分组规定：
  1. group by 总是在 where 之后，在 order by 之前！
  2. 分组 select 的字段，处了 分组字段 之外，其余字段都应该应用在 group by 子句中！
  3. NULL 会单独分为一组！
  4. 大多数 SQL 不支持 group by 列具有可变长度的数据类型！

子查询

- 子查询 是指 查询的语句中 嵌套了 select 子句！

  查询语句称为外层查询，被嵌套的select 子句称为内层查询！
- 可以将子查询的结果作为 where 子句的过滤条件！

```
# 从学校表中查询 老师id列表，再根据 老师id 在 教师表中查询教师的名字！
select name from teacher where teacher_id in (select teacher_id from
school);
```

- 子查询只能返回一 列！

```
# 如下，子查询中返回了所有列，这是不合法的，将会报错！
select name,(select * from teacher_A where teacher_A.id=teacher_B.id) from
teacher_B;

# 报错：子查询中只能返回 1 列！
ERROR 1241 (21000): Operand should contain 1 column(s)
```

- 实际应用示例：

```
# 获取用户的订单数量：cust表 与 order表
select name,(select count(*) from orders where orders.c_id=cust.id) AS total
from cust
order by total;
```

连接 查询

概述

- 连接用于连接多个表！
- 使用关键字 JOIN 关联两个表

使用 关键字ON 作为关联条件，如果作为关联条件的列名相同，则可以使用 using 代替！

```
# join + on
select * from table_1 join table_2 on table_1.id=table_2.id;
# join + using
select * from table_1 join table_2 using(id);

# using 和 on 区别在于：using 会将 同名关联列去重，而 on 则会都列出（无论是否同名）！
# 如上：使用 on 时，table_1.id 和 table_2.id 都会显示（两列结果相同），而使用 using，则只会显示一个 id 列！
```

- 表连接可以代替子查询，并且一般比子查询的效率更好（且不谈子查询的底层原理，子查询都会建立临时表，用完之后删除临时表，浪费时间！）
- 可以用 AS 关键字为 列、计算字段、表 取别名，这是为了简化 sql 语句 以及 连接相同表！
- 表连接可分为：内连接、自然连接、外连接（左外连接、右外连接、全外连接）！

内连接

- 内连接又称为 等值连接!
- 使用 inner join 关键字实现！

```
select t1.col,t2.col
from table_1 AS t1
inner join table_2 AS t2 on t1.id=t2.id;
```

- 可以不明确使用 inner join，而使用 普通查询 并在 WHERE子句中 通过 指定列 将两个表 等值关联！

```
select t1.col,t2.col
from table_1 AS t1,table_2 AS t2
where ti.id=t2.id;
```

自然连接

- 以两个表中的 等值同名列 为枢纽，将两个表关联起来！

  同名列可以有多个，此时，只有两个表之间对应的多个同名列 完全等值时，才会关联起来！
- 使用关键字 NATURAL JOIN 完成自然连接！

```
select t1.col,t2.col
from table_1 AS t1
natural join table_2 AS t2;
```

- 内连接 vs 自然连接

  1. 内连接需要认为提供关联列，而 自然连接 自动通过 所有同名列 等值关联！

外连接

- 内连接 是等值连接，会自动去掉 两个表中 关联列不等值 的行记录！

  外连接，则是内连接的超集，他包含不等值的行，在不等值的行中，使用 NULL 填充所有列！
- 左（外）连接

  以左边的表为基准，按照关联列，匹配等值的行：

若右表关联列中 不存在 左表关联列的值，则保留左表相关行，右表使用 NULL 填充其所有列作为其关联的行！

若左表关联列中 不存在 右表关联列的值，则去掉该行！

即：以左表为标准，保留左表所有行，右表中等值关联，不存在的则用 NULL 填充所有列作为关联，且去掉右表不等值的行！

- 右（外）连接

  与左外连接 正好相反！

- 全（外）连接

  保留左右表中的所有行，相互之间不存在的 行 使用 NULL 填充

  即：左外连接 与 右外连接 结果的并集！

组合查询

- 组合查询 使用 Union 操作符，将多个 SELECT 查询的结果组合起来，一同返回，因此也成为 并查询 或者 复合查询！
- 使用场景：
  1. 从多个表中查询出相似结构的数据，并能组合成一个结果集！
  2. 从单个表中多次 select 查询，并将结果合并成一个结果集！
- 规定：
  1. Union 必须由两条或两条以上的 select 语句组成，select 之间使用 Union 连接。
  2. Union 中每个查询必须包含相同的列、表达式、聚合函数，他们出现的顺序可以不一致

     （这里是指列名相同，而表不一定！如果列名不同，则会以第一个 select 查询的字段为标准，或者使用 别名）
  3. 列的数据类型必须兼容，即：可以被数据库 隐式的进行 数据转换！
- 组合查询只能使用一个 order by 子句对结果集排序，并且只能位于最后！

  因为 Union 不允许对部分结果集排序！
- Union 会自动删除重复的行，如果想要保留 重复行，则需要使用 Union ALL
- 示例：

```
select col,id from table_1 where id<100
union
select col,id from table_2 where id<100
order by id;
```

[组合查询：详解]

视图

- Mysql 的 视图是一个 基于 基表 的虚拟表，不存放任何数据。它和表处于同一个命名空间，具有与普通表相同的 DML 操作！
- 视图定义

  视图由 SQL 查询 语句定义，在更新视图时，需要满足 定义视图 时的约定！

```
create [or replace]
[algorithm ={ UNDEFINED | MERGE | TEMPTABLE }]
[DEFINER = { user | CURRENT_USER }]
[SQL SECURITY { DEFINER | INVOKER }]
VIEW view_name [ (column_list) ]
AS select_statement
[WITH [ CASCADED | LOCAL ] CHECK OPTION]

# 中括号 [] 中的内容可选
# algorithm 表示 mysql 生成视图的算法：TEMPTABLE 表示生成临时表
# DEFINER 定义 视图的人
# SQL SECURITY 限定视图的使用权限
# with check option ：对于可更新视图，更新视图内容时，检查更新是否合法！不合法时报错！
```

- 定义示例：

```
create
view vitural_table
AS select * from table_1 where table_1.id between 10 and 100;
with check option;

# 使用视图
select * from vitural_table;
```

- 视图更新

  视图时基于 基表的虚拟表，对它的所有操作都会映射到实际的表中！所以，也能对视图进行更新，其本质就是 通过视图的定义 来更新实际的表！

  视图定义中的 with check option 属性会在更新视图时检查更新操作是否合法，不合法时会报错！

  由于视图是 sql 查询的结果集，所以，更新视图的操作需要满足 sql 查询的条件！

```
# 如上视图 vitural_table 的定义，它是 table id 为 10 到 100 之间的结果！
# 因此，更新视图时，table id 也应该要在 这个范围内！

# 示例：假设 视图中存在 id=5 的记录

# 将 id 更新为 101 时报错！（不在 1 ~ 100 之间）
mysql> update vitural_table set uid =101 where id=5;
ERROR 1369 (HY000): CHECK OPTION failed 'db.vitural_table'

# 将 id 更新为 100 时成功 —— 满足定义时的条件！
mysql> update vitural_table set uid =100 where id=5;
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

  如果定义时不加 with check option ，则不会报错，但是不合法的更新依旧不会成功！

- 作用：

  1. 视图被用作抽象装置，程序本身不需要关心基表的结构，只需要按照视图定义来获取或更新数据即可，一定程度上，起到安全层的作用！
  2. 可以用来简化复杂的 SQL 操作，比如：复杂的 表连接！
  3. 使用视图时，相当于只使用了实际表的部分数据！
  4. 通过只给用户访问视图的权限，能保证数据的安全性！

- 注意：

1. 视图是虚拟的表，不包含数据，也就不可能有索引！
2. mysql 不支持在 视图上建立 触发器！
3. 删除视图时不能使用 drop table 应该使用 drop view

存储过程

- 存储过程 是 封装了 特定功能的 SQL语句集 的代码块，存储在数据库中！

  它是一个 预编译的代码块，第一次调用编译之后，再次调用不再需要编译！

- 存储过程 定义

  在命令行中定义 存储过程 时，需要自定义分隔符，避免命令行分隔符 于 存储过程中的 分号(;) 冲突！

  - 存储过程 可以携带参数，有：in、out、inout 三种参数！

    in： 表示入参

    out： 表示 返回值

    inout： in 和 out 的结合

  - 存储过程通过 declare 声明变量，并使用 select into 为 1 个变量赋值！
  - 存储过程 不支持 集合操作！

- 示例：

```
# 自定义分隔符：暂时使用 // 作为 mysql 命令行的分隔符
delimiter //

# 定义存储过程 my_procedure，并携带 int 类型的 返回参数 ret
create procedure my_procedure(out ret int)
# 代码块开始
begin
    # 声明 int变量 var_1 var_2
    declare var_1 int;
    declare var_2 int;
    # 通过 select into 为 var_1 var_2 赋值
    select count(*) from teacher into var_1;
    select count(*) from student into var_2;
    # 通过 select into 为 ret 赋值
    select var_2 / var_1 into ret;
# 代码块结束
end //

# 分割符换回 分号(;)
delimiter ;
```

  调用过程：

```
call my_procedure(@ret);
select @ret;
```

- 优点

  1. 封装了 sql 代码，存储数据库中，保证了一定的安全性！
  2. 实现代码复用
  3. 是预编译代码块，所以，执行效率高，不需要重复编译！

- 缺点

1. mysql 中并没有为存储过程提供方便的开发和调试工具，所以，编写 MySQL 的存储代码较其他数据库要困难些！
2. mysql 并没有控制存储过程的资源消耗，如果存储过程中出现错误，可能数据库会崩掉！
3. 业务逻辑放在数据库上，难以维护！
4. 每个数据库的存储过程 语法不一样，可移植性差！

触发器 trigger

- 触发器 是 与表有关的数据库对象，在满足定义条件时触发，并执行在触发器中定义的 SQL 代码块！

  在执行 INSERT、DELETE、UPDATE 命令之前或者之后 自动执行 触发中定义 代码块！

  触发器 trigger 详解

- 最多可以为一个表建立 6 个触发器，即分别为 INSERT、UPDATE、DELETE 者三个操作的 BEFORE、AFTER 版本！

  BEFORE 和 AFTER 表示触发器触发执行的时间：

  BEFORE 表示每行操作之前触发，常用于数据验证 和 净化

  AFTER 表示每行之后触发，常用于审计跟踪，将记录修改到另一张表中！

- 触发器会创建虚拟表
  1. INSERT 触发器包含一个 名为 NEW 的虚拟表
  2. DELETE 触发器包含一个名为 OLD的虚拟表，并且是只读的！
  3. UPDATE 触发器包含一个 NEW 和 OLD 的虚拟表，其中 NEW 可以被修改，而 OLD 是只读的！
- 触发器语法

```
CREATE TRIGGER trigger_name trigger_time trigger_event ON tb_name FOR EACH
ROW trigger_stmt
# trigger_name：触发器的名称
# tirgger_time：触发时机，为BEFORE或者AFTER
# trigger_event：触发事件，为INSERT、DELETE或者UPDATE
# tb_name：表示建立触发器的表明，就是在哪张表上建立触发器
# trigger_stmt：触发器的程序体，可以是 一条SQL语句 或者是用 BEGIN 和 END 包含的多条语句
# 所以可以说MySQL创建以下六种触发器：
  BEFORE INSERT, BEFORE DELETE, BEFORE UPDATE
  AFTER INSERT, AFTER DELETE, AFTER UPDATE
```

  示例

```
# 创建触发器
create trigger
mytrigger after insert on table_1
for each row
select new.col into @result;

# 当执行 insert 之后，将结果写入 @result 变量中，通过 select 获取结果！
SELECT @result;
```

- 作用
  1. 通过触发器，用户可以实现 MySQL 数据库本身不支持的一些操作，例如：对于传统 CHECK 约束的支持、物化视图、高级复制、审计等！（记录日志等）

2. 通过触发器保证数据库数据完整性，例如：参照完整性：不使用物理外键，而是自己在触发器中实现表之间逻辑外键关系！

- 删除触发器：drop trigger my_trigger;
- 注意：

1. 触发器并不能保证操作的原子性！

   在 MyISAM 中通过触发器更新另一个表，若触发器执行出错，则无法完成回滚！因此，在 MyISAM 中触发器并不能保证操作同时成功或者同时失败！

   在 InnoDB 储存引擎中，触发器和操作是在同一个事务中完成的，所以，此时触发器的执行能保证原子性！

2. 如果在 InnoDB 表上建立触发器检查数据的一致性需要特别小心 MVCC，稍不小心，可能会获得错误的结果！

   例如：如果打算编写一个 BEFORE INSERT 触发器检查写入的数据对应列在另一个表中是否存在，但若你在触发器中没有使用 SELECT FOR UPDATE，那么并发的更新语句可能会立即更新对应记录，导致数据不一致！

3. 对每一个表的每一个事件，最多定义一个触发器（例如：不能再 AFTER INSERT 上定义两个触发器）
4. Mysql 只支持 “基于行的触发” —— 触发器始终是针对一条记录的，而不是针对整个 SQL 语句，如果变更的数据集非常大，效率会很低！
5. 触发器掩盖了服务器背后的工作，一条简单的 SQL，因为触发器，背后可能包含了很多看不见的工作！

   因此触发器的问题很难排查，如果触发器引发 了性能问题，那将难以分析和定位！

6. 触发器可能导致死锁 和 锁等待！

   如果触发器失败，那么原来的 SQL 也会失败，如果没有意识到这其中是 触发器的问题，也很难排查问题！

7. 触发器 不允许触发器使用 CALL ，也就是不能调用 存储过程 ？？？ （不支持调用动态 SQL？）