数据库基础

> 数据库的基本学习

::: tips
自我评价

- 关于数据库的学习，我并没有一直跟随着上课老师的脚步去走，因为正如PYTHON的学习一般，数据库的基础学习不需要花费大量时间，主要是注重语法的调用，然后注重听上课时老师讲的一些易错的点,(ex:NULL的不同场景的表现)，进而去完善自己的学习。
- 数据库更注重于实际应用，我在搭建博客的时候，文章归类问题便是应用数据库的相关知识，并且在做期末项目的时候，思考到了项目应该与我们所学专业相互融合，我们对基础的SQL终端进行了一些“嵌入式”的改良。

:::

- 数据库语法
- 数据库语法实操--已经通关

基本操作

数据类型

数字类型

数据类型	说明
INT/INTEGER	整数
SMALLINT	小范围整数（32位）
BIGINT	大范围整数（64位）

数据类型	说明
FLOAT	单精度浮点数
DOUBLE/REAL	双精度浮点数
DECIMAL(n, m) / NUMERIC(n, m)	精确小数，n位m位小数

字符串类型

数据类型	说明
CHAR(n)	固定长度字符串
VARCHAR(n)	可变长度字符串
TEXT/ CLOB	长文本

日期时间类型

数据类型	说明
DATE	日期
TIME	时间
DATETIME	日期+时间
TIMESTAMP	时间戳，记录时间点

布尔以及其他类型

数据类型	说明
ENUM	枚举类型（const）
JSON	JSON数据和事
UUID	通用唯一标识符

基本语法

创建表格（CREATE TABLE）

```
CREATE TABLE 表名称 (
   列名称1   数据类型,
   列名称2   数据类型,
   列名称3   数据类型,
   ....
);
```

- 操作实例

```
CREATE TABLE `user`(
    `id` int(100) unsigned NOT NULL AUTO_INCREMENT pri
    `password` varchar(32) NOT NULL DEFAULT '' COMMENT
    `mobile` varchar(32) NOT NULL DEFAULT '' COMMENT '
    `update` timestamp(6) NOT NULL DEFAULT CURRENT_TIM
    UNIQUE INDEX id_user_mobile(`mobile`)
)
```

- COMMENT 表示注释
- UNSIGNED 是非负数
- AUTO_INCREMENT 自动递增标签
- DEFAULT 表示默认
- NOT NULL 表示不为空
- CURRENT_TIMESTAMP(6) ON UPDATE CURRENT_TIMESTAMP(6)修改更新时间

删除表格

```
DROP TABLE 表名称; -- 删除表
TRUNCATE TABLE Shippers; -- 不删除表，删除表内数据
```

SELECT

```
SELECT 列名称 FROM 表名称 -- *表示所有
SELECT 列名称 FROM 表名称 WHERE 列数据满足条件
```

- 操作实例

```
-- SELECT S_ID FROM Student WHERE S_ID in (1,10) and S
-- SELECT S_ID FROM Student WHERE S_ID in(1, 10) limit
-- 结果数据自动去重
SELECT DISTINCT 列名称 FROM 表名称
-- 数据分组排序统计
COUNT(列) 计数 GROUP BY 进行分组 order by 排序
SELECT 列数据, COUNT(列) FROM 表 GROUP BY tag order by i
SELECT tag
```

UPDATE

```
UPDATE 表名称 SET 列名称1 = 值1, 列名称2 = 值2 WHERE 条件
```

- 操作实例

```
-- update语句设置字段值为另一个结果取出来的字段
    UPDATE user set name = (SELECT name from user1 WHE

-- 更新表 orders 中 id=1 的那一行数据更新它的 title 字段
UPDATE orders set title='这里是标题' WHERE id=1;
```

INSERT

```
INSERT INTO 表名称(列名称, 列名称) VALUES(值, 值)
INSERT INTO 表名称 SET 列名称 = 值
INSERT INTO 表名称 VALUES(值1, 值2) -- 按顺序插入
```

- 操作实例

```
INSERT INTO id SET a = 1, b = 2;
-- 等价于
INSERT INTO id (a, b) VALUES(1,2)
```

DELETE + WHERE

```
DELECT FROM 表 WHERE 条件;
SELECT 列 FROM 表 WHERE 条件
```

AND，OR，NOT,IN

- AND 相当于&运算，OR相当于 | 运算，NOT相当于~运算
  计算和 算法中一致，先处理括号中的BOOLEAN，不做过多介绍
- IN 表示在某某之中

```
SELECT 列名称, 列名称, ... FROM 表名称 WHERE 条件1 AND 条件
SELECT 列名称1, 列名称2, ... FROM 表名称 WHERE 条件1 OR 条
SELECT 列名称1, 列名称2, ... FROM 表名称 WHERE NOT 条件2;
```

ORDER BY，GROUP BY

- ORDER BY是根据你选的值进行排序

```
SELECT 列名称1, 列名称2, ... FROM 表名称 ORDER BY 列名称1,
```

- GROUP BY 根据你选的值进行分组

```
SELECT 列名称(s)
FROM 表名称
WHERE 条件
GROUP BY 列名称(s)
ORDER BY 列名称(s);
```

UNION

合并多个SELECT语句的结果

```
SELECT 列名称 FROM 表1
UNION
SELECT 列名称 FROM 表2
```

- 操作实例

```
SELECT E_name FROM China UNION SELECT E_name FROM Engl
```

BETWEEN，AS

- BETWEEN表示在某个区间之中

```
SELECT 列 FROM 表 BETWEEN 值1 AND 值2
```

- AS为表或者列取别名（& -- C++中的引用）

```
SELECT 列 AS 别名 FROM 表 -- 为列设置别名
SELECT 列 FROM 表 AS 别名 -- 为表设置别名
```

JOIN

INNER JOIN

返回两个表中满足条件的列

```
SELECT 列
FROM 表1
INNER JOIN 表2
ON 表1.列 = 表2.列
```

LEFT JOIN

::: success
注意
即使右表中没有匹配，也会从左表返回所有满足该列的数据
:::

```
SELECT 列名称(s)
FROM 表1
LEFT JOIN 表2
ON 表1.列名称 = 表2.列名称;
```

RIGHT JOIN

::: success
注意
即使左表中没有匹配，也会从右表返回所有满足该列的数据
:::

```
SELECT 列名称(s)
FROM 表1
RIGHT JOIN 表2
ON 表1.列名称 = 表2.列名称;
```

FULL OUTER JOIN

没有匹配的也会返回该行，但是不是显示除了主键的数据

```
SELECT 列名称(s)
FROM 表1
FULL OUTER JOIN 表2
ON 表1.列名称 = 表2.列名称
WHERE 条件;
```

SQL函数

- COUNT
  用于计数

```
SELECT COUNT(列名称) FROM 表名称 WHERE 条件;
```

- AVG
  取平均值

```
SELECT AVG(列名称) FROM 表名称 WHERE 条件;
```

- SUM
  求和

```
SELECT SUM(列名称) FROM 表名称 WHERE 条件;
```

- MAX，MIN
  求最大值最小值

```
SELECT MIN(列名称) FROM 表名称 WHERE 条件;
```

触发器

```
create trigger <触发器名称>
{ before | after}          -- 之前或者之后出发
insert | update | delete   -- 指明了激活触发程序的语句的类型
on <表名>                   -- 操作哪张表
for each row               -- 触发器的执行间隔，for each r
<触发器SQL语句>
```

```
delimiter $
CREATE TRIGGER set_userdate BEFORE INSERT
on `message`
for EACH ROW
BEGIN
```

```
    set @statu = new.status; -- 声明复制变量 statu
    if @statu = 0 then         -- 判断 statu 是否等于 0
      UPDATE `user_accounts` SET status=1 WHERE openid=N
    end if;
  END
  $
  DELIMITER ; -- 恢复结束符号
```

OLD和NEW不区分大小写

- NEW 用NEW.col_name，没有旧行。在DELETE触发程序中，仅能使用OLD.col_name，没有新行。
- OLD 用OLD.col_name来引用更新前的某一行的列

索引

提高查询效率，要求索引唯一

普通索引

```
ALTER TABLE `表名字` ADD INDEX 索引名字 ( `字段名字` )
-- 对于字段名字（表中一列数据）提供 `索引名字`作为索引
```

- 操作实例

```
-- 给 user 表中的 id字段 添加主键索引(PRIMARY key)
ALTER TABLE `user` ADD PRIMARY key (id);
```

主键索引

```
> ALTER TABLE `表名字` ADD PRIMARY KEY ( `字段名字` )
```

- 操作实例

```
-- 给 user 表中的 id字段 添加主键索引(PRIMARY key)
ALTER TABLE `user` ADD PRIMARY key (id);
```

唯一索引

```
ALTER TABLE `表名字` ADD UNIQUE (`字段名字`)
```

- 操作实例

```
-- 给 user 表中的 id字段 添加主键索引(PRIMARY key)
ALTER TABLE `user` ADD PRIMARY key (id);
```

全文索引

```
ALTER TABLE `表名字` ADD FULLTEXT (`字段名字`)
```

- 操作实例

```
-- 给 user 表中的 description 字段添加全文索引(FULLTEXT)
ALTER TABLE `user` ADD FULLTEXT (description);
```

建立索引的时机

- 可以使用 `LIKE` 以通配符%和_作为开头查询时

```
SELECT * FROM mytable WHERE username like'admin%'; --
SELECT * FROM mytable WHERE Name like'%admin'; -- 因此,
```

创建表后的修改

添加列

```
alter table 表名 add 列名 列数据类型 [after 插入位置];
```

- 操作实例

```
-- 在表students的最后追加列 address:
alter table students add address char(60);
-- 在名为 age 的列后插入列 birthday:
alter table students add birthday date after age;
-- 在名为 number_people 的列后插入列 weeks:
alter table students add column `weeks` varchar(5) not
```

修改列

```
alter table 表名 change 列名称 列新名称 新数据类型;
```

- 操作实例

```
-- 将表 tel 列改名为 telphone:
alter table students change tel telphone char(13) defa
-- 将 name 列的数据类型改为 char(16):
alter table students change name name char(16) not nul
-- 修改 COMMENT 前面必须得有类型属性
alter table students change name name char(16) COMMENT
```

删除列

```
alter table 表名 drop 列名称;
```

- 操作实例

```
-- 删除表students中的 birthday 列:
alter table students drop birthday;
```

重命名表

```
alter table 表名 rename 新表名;
```

- 操作实例

```
-- 重命名 students 表为 workmates:
alter table students rename workmates;
```

清空表数据

- `DELETE:` 1. DML语言;2. 可以回退;3. 可以有条件的删除;
- `TRUNCATE:` 1. DDL语言;2. 无法回退;3. 默认所有的表内容都删除;4. 删除速度比delete快。

```
delete from 表名;
truncate table "表名";
```

- 操作实例

```
-- 清空表为 workmates 里面的数据，不删除表。
delete from workmates;
-- 删除workmates表中的所有数据，且无法恢复
truncate table workmates;
```

删除整张表

```
drop table 表名;
```

- 操作实例

```
-- 删除 workmates 表:
drop table workmates;
```

删除整个数据库

```
drop database 数据库名;
```

- 操作实例

```sql
-- 删除 samp_db 数据库:
drop database samp_db;
```

相关数据库的扩展内容

事务与索引优化

事务是一组数据库操作（例如插入、更新、删除），这些操作要么全部成功执行，要么全部失败回滚，确保数据一致性。事务具有 ACID 特性：

- A（Atomicity，原子性）：事务操作不可分割，要么全做，要么全不做
- C（Consistency，一致性）：事务完成后，数据库保持一致状态
- I（Isolation，隔离性）：多个事务并发执行时，互不干扰
- D（Durability，持久性）：事务一旦提交，数据永久保存

索引优化

索引是数据库中用于加速查询的特殊结构，类似于书的目录。常见的索引类型包括：

- B+树索引：适合范围查询和排序 （二分）
- 哈希索引：适合精确匹配查询
- 全文索引：用于文本搜索

操作实例

```sql
-- 开启事务
START TRANSACTION;

-- 执行多条 SQL 操作
UPDATE account SET balance = balance - 100 WHERE user_
UPDATE account SET balance = balance + 100 WHERE user_

-- 提交事务（成功时）
COMMIT;
```

```sql
-- 或者回滚事务（失败时）
ROLLBACK;

-- 为 user 表的 email 字段添加 B+树索引
CREATE INDEX idx_email ON user(email);

-- 查看查询计划，优化索引使用
EXPLAIN SELECT * FROM user WHERE email = 'test@example
```

关系数据库范式

关系数据库范式（Normalization）是设计数据库表结构的一套规则，旨在减少数据冗余、确保数据一致性。常见的范式包括：

- 第一范式（1NF）：确保表中每个字段值不可再分（原子性），且每行有唯一标识（主键）。
- 第二范式（2NF）：在 1NF 基础上，确保非主键字段完全依赖于主键（消除部分依赖）。
- 第三范式（3NF）：在 2NF 基础上，确保非主键字段之间没有传递依赖。
- 反范式化：在某些场景（如高性能查询需求）下，可能有意违反范式，增加冗余以提升查询效率。
- 操作实例

```sql
-- 未规范化表（违反 1NF，address 字段可再分）
CREATE TABLE bad_order (
    order_id INT PRIMARY KEY,
    customer_name VARCHAR(50),
    address VARCHAR(200) -- 包含城市、街道等
);

-- 规范化到 1NF（拆分 address）
CREATE TABLE good_order (
    order_id INT PRIMARY KEY,
```

```
    customer_name VARCHAR(50),
    city VARCHAR(50),
    street VARCHAR(100)
);

-- 规范化到 2NF（将客户信息单独拆出，消除部分依赖）
CREATE TABLE customer (
    customer_id INT PRIMARY KEY,
    customer_name VARCHAR(50),
    city VARCHAR(50),
    street VARCHAR(100)
);

CREATE TABLE order (
    order_id INT PRIMARY KEY,
    customer_id INT,
    FOREIGN KEY (customer_id) REFERENCES customer(cust
);
```

::: warning

警告

后续是想在我们的国创中设计一个应用可以需要一个前后端开发的软件，到时候会更新数据库做项目的具体细节

:::