MySQL

MySQL期末复习整理

TODO:

- [x] MySQL的安装配置（Window系统下）
- [x] 命令操作
  - [x] 操作MySQL数据库
    - [x] 创建数据库
    - [x] 查看和选择数据库
    - [x] 删除数据库
  - [x] 常见的字段类型
    - [x] 数值类型
    - [x] 日期和时间类型
    - [x] 字符串类型
  - [x] 操作表
    - [x] 创建数据表
    - [x] 查看表结构
    - [x] 修改字段名称
    - [x] 修改字段数据类型
    - [x] 删除字段
    - [x] 删除表
    - [x] 数据表添加字段(三种方式)
  - [x] 操作数据（增删改查）
    - [x] 插入数据记录
    - [x] 更新数据记录
    - [x] 删除数据记录
    - [x] 清空表记录
  - [x] 操作索引
  - [x] 操作视图
  - [x] 操作触发器
  - [x] 事务机制

MySQL的安装配置（Windows）

MySQL下载地址：https://downloads.mysql.com/archives/community/

1. 下载

目前MySQL分为两大版本，**MySQL 5.x** 和 **MySQL 8.x**，其中大多数使用的是5.7版本的MySQL。

5.x版本可以直接从官网进行下载，其下载的是一个ZIP压缩包，下载完成后，直接将其解压至想要安装的文件夹即可。

2. 配置

完成解压安装后，需要写入配置文件，建议写在安装的文件夹中，创建`my.ini`文件，编辑其内容如下（以5.7.31为例）：

```
[mysqld]

# 设置数据库端口
port=3306
# 设置 MySQL 的安装目录
basedir=C:\\Program Files\\mysql-5.7.31-winx64
# 设置 MySQL 数据库的数据的存放目录，MySQL 8.x 不需要以下配置，系统自己生成即可，否则有可能报错
datadir=C:\\Program Files\\mysql-5.7.31-winx64\\data
```

可以通过`"C:\Program Files\mysql-5.7.31-winx64\bin\mysqld.exe" --help --verbose`查看MySQL配置文件的优先级，在安装目录下创建MySQL配置文件可以避免安装了多个MySQL版本造成的冲突问题。

3. 初始化

使用`"C:\Program Files\mysql-5.7.31-winx64\bin\mysqld.exe" --initialize-insecure`命令可以进行初始化。

该过程会自动创建`data`目录，数据会存放在这个目录中。同时创建一些必备的数据，例如默认账户root(无密码)，用于登录MySQL并通过指令操作MySQL。

可能出现的问题：`msvcr120.dll`不存在

1. 安装 vcredist：https://www.microsoft.com/zh-cn/download/confirmation.aspx?id=40784
2. 安装 dirctx：https://wwwmicrosoftcom/zh-CN/download/detailsaspx?id=35

4. 启动

方式一：临时启动

`>>> "C:\Program Files\mysql-5.7.31-winx64\bin\mysqld.exe"`

方式二：制作服务

可以通过以下命令制作服务：

`>>> "C:\Program Files\mysql-5.7.31-winx64\bin\mysqld.exe" --install mysql57`

通过以下方式启动服务：

`>>> net start mysql57`

当想要卸载MySQL服务时，可使用下列命令：

`>>> "C:\Program Files\mysql-5.7.31-winx64\bin\mysqld.exe" --remove mysql57`

5. 测试连接MySQL

通过`mysql.exe`可以测试连接MySQL

`>>> "C:\Program Files\mysql-5.7.31-winx64\bin\mysql.exe" -h 127.0.0.1 -P 3306 -u root -p`

该命令中，`-h`表示*IP地址*，`-P`表示*端口*，`-u`表示*用户名*，`-p`表示密码。在本地环境时，即连接本地数据库时，参数`-h`和参数`-P`都是可以省略的。

6. 关于密码

1. 设置和修改密码

   登录账户后，输入命令修改密码：

   ```
   -- 输入以下命令即可修改账户的密码
   set password = password('Your Password')
   ```

2. 忘记密码

   如果忘记了MySQL账户的密码，需要进行以下的操作：

   - 修改配置文件，在`[mysqld]`节点下添加`skip-grant-tables=1`。

     ```
     [mysqld]
     skip-grant-tables=1
     ```

   - 重启MySQL服务，再次登录时，无需密码即可登录。

     ```
     net stop mysql57
     net start mysql57

     mysql -u root -p
     ```

   - 进入数据库后执行修改密码命令。

     ```
     use mysql;
     update user set authentication_string = password('New Password'),
     password_last_changed()=now() where user='root';
     ```

   - 退出并再次修改配置文件，注释掉`skip-grant-tables=1`。

     ```
     [mysqld]
     # skip-grant-tables=1
     ```

   - 再次重启，即可使用新密码登录。

命令操作

操作MySQL数据库

MySQL数据库其实就像是文件夹，MySQL数据库中的表其实就相当于数据库中的Excel文件。

1. 创建数据库

```
-- 创建数据库命令 create database [数据库名称]
CREATE DATABASE testdb;
-- 或者使用更多的约束来限制 create 命令
CREATE DATABASE testdb DEFAULT CHARSET utf8 COLLATE utf8_general_ci;
```

2. 查看当前所有的数据库：`SHOW DATABASES [LIKE '数据库名'];`

```
-- LIKE 为可选字段，其后可以加数据库的名称，数据库名称用单引号包裹
-- % 百分号表示通配符
SHOW DATABASES LIKE 'test%';
```

结果如下：

```
+------------------+
| Database (test%) |
+------------------+
| test             |
| testdb           |
+------------------+
2 rows in set (0.00 sec)
```

3. 选择数据库：`USE 数据库名;`

```
-- 选择数据库，相当于进入文件夹
USE testdb;
```

4. 删除数据库：`DROP DATABASE 数据库名;`

```
-- 删除数据库
DROP DATABASE testdb;
```

5. [拓展] Python如何使用MySQL

```
import pymysql

# 连接MySQL
conn = pymysql.connect(host="127.0.0.1", port=3306, user='root',
passwd='root', charset=utf8)
cursor = conn.cursor()  # 获取游标

# 一般来说，pymysql中，需要查看的命令，都使用 fetchall() 来获取返回结果
# 而创建(新增)、删除、修改等，需要使用 commit() 来提交命令
# 1. 查看数据库
cursor.execute("SHOW DATABASES")
result = cursor.fetchall()

# 2. 创建数据库
cursor.execute("CREATE DATABASE testdb DEFAULT CHARSET utf8 COLLATE
utf8_general_ci")
conn.commit()
# ...(其他代码不再赘述)
```

```
# 关闭连接
cursor.close()
conn.close()
```

常见的字段类型

1. 数值类型

类型	大小	用途
TINYINT	1 byte	小整数值
SMALLINT	2 bytes	大整数值
MEDIUMINT	3 bytes	大整数值
INT或INTEGER	4 bytes	大整数值
BIGINT	8 bytes	极大整数值
FLOAT	4 bytes	单精度浮点数值
DOUBLE	8 bytes	双精度浮点数值
DECIMAL	对DECIMAL(M, D) 如果M>D，为M+2；否则为D+2	小数值

较为常用的如下：

- **INT型**：`INT [(m)][UNSIGNED][ZEROFILL]`

```
INT                 表示有符号，取值范围：-2147483648~2147483647
INT UNSIGNED        表示无符号，取值范围：0~4294967295
INT(5) ZEROFILL     仅用于显示，当不满足5位时，按照左边补0，例如：00002；满足时，正常显示
```

```
-- 创建表
CREATE TABLE L1(
    id INT,
    uid INT UNSIGNED,
    zid INT(5) ZEROFILL
);

-- 向表中插入数据
INSERT INTO L1(id, uid, zid) VALUES(1, 2, 3);
INSERT INTO L1(id, uid, zid) VALUES(10000, 20000, 30000);

-- 查看表
SELECT * FROM L1;
```

结果如下：

```
+-------+-------+-------+
| id    | uid   | zid   |
+-------+-------+-------+
|     1 |     2 | 00003 |
| 10000 | 20000 | 30000 |
+-------+-------+-------+
2 rows in set (0.00 sec)
```

- **TINYINT** `TINYINT [(m)][UNSIGNED][ZEROFILL]`
- **BIGINT** `BIGINT [(m)][UNSIGNED][ZEROFILL]`
- **DECIMAL(M, D)** `DECIMAL[(m[,d])][UNSIGNED][ZEROFILL]`

  准确的小数值，M是数字总个数(包括整数部分和小数部分，负号不算)，D是小数点后个数。M最大值为65，D最大值为30。

```sql
-- 创建表
CREATE TABLE L2(
    id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
    salary DECIMAL(8, 2)
)DEFAULT CHARSET=utf8;

-- 插入数据
INSERT INTO L2(salary) VALUES(5.28);
INSERT INTO L2(salary) VALUES(5.282);
INSERT INTO L2(salary) VALUES(512132.28);
INSERT INTO L2(salary) VALUES(512132.282);

-- 查看表
SELECT * FROM L2;
```

  结果如下：

```
Query OK, 1 row affected (0.01 sec)
Query OK, 1 row affected, 1 warning (0.00 sec)  # 出现警告，小数位超出
Query OK, 1 row affected (0.00 sec)
Query OK, 1 row affected, 1 warning (0.00 sec)  # 出现警告，小数位超出
+----+-----------+
| id | salary    |
+----+-----------+
|  1 |      5.28 |
|  2 |      5.28 |
|  3 | 512132.28 |
|  4 | 512132.28 |
+----+-----------+
```

- **FLOAT(M, D)** `FLOAT[(m[,d])][UNSIGNED][ZEROFILL]`
- **DOUBLE(M, D)** `DOUBLE[(m[,d])][UNSIGNED][ZEROFILL]`

2. 日期时间类型

类型	大小	格式	用途

类型	大小	格式	用途
DATE	3	YYYY-MM-DD	日期值
TIME	3	HH:MM:SS	时间或持续时间
YEAR	1	YYYY	年份值
DATETIME	8	YYYY-MM-DD HH:MM:SS	混合日期和时间值
TIMESTAMP	4	YYYYMMDD HHMMSS	混合日期和时间值，时间戳

- `TIMESTAMP`

  对于**TIMESTAMP**，它把客户端插入的时间从当前时区转化为UTC（世界标准时间）进行存储。查询时，将其又转化为客户端当前时区进行返回。
  对于**DATETIME**，不做任何改变，原样输入和输出。

```
-- 创建表
CREATE TABLE L3(
    id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
    dt DATETIME,
    tt TIMESTAMP
)DEFAULT CHARSET=utf8;

-- 插入数据
INSERT INTO L3(dt, tt) VALUES ("2025-11-11 11:11:44", "2025-11-11
11:11:44");

-- 查看表
SELECT * FROM L3;
```

  结果如下：

```
+----+---------------------+---------------------+
| id | dt                  | tt                  |
+----+---------------------+---------------------+
|  1 | 2025-11-11 11:11:44 | 2025-11-11 11:11:44 |
+----+---------------------+---------------------+
1 row in set (0.00 sec)
```

3. 字符串类型

类型	大小	用途
CHAR	0-255 bytes	定长字符串
VARCHAR	0-65535 bytes	变长字符串
TINYBLOB	0-255 bytes	不超过 255 个字符的二进制字符串
TINYTEXT	0-255 bytes	短文本字符串
BLOB	0-65535 bytes	二进制形式的长文本数据
TEXT	0-65535 bytes	长文本数据
MEDIUMBLOB	0-16777215 bytes	二进制形式的中等长度文本数据
MEDIUMTEXT	0-16777215 bytes	中等长度文本数据
LONGBLOB	0-4294967295 bytes	二进制形式的极大文本数据
LONGTEXT	0-4294967295 bytes	极大文本数据

- **CHAR** `CHAR(m)`

  定长字符串，m代表字符串的长度，最多可容纳255个字符。
  定长的体现：即使内容长度小于m，也会占用m长度。例如: char(5)，数据是：yes，底层也会占用5个字符；如果超出m长度限制（默认MySQL是严格模式，所以会报错）。

  如果在配置文件中加入如下配置，`sql-mode="NO_AUTO_CREATE_USER, NO_ENGINE_SUBSTITUTION"`保存并重启，此时MySQL则是非严格模式，此时超过长度则自动截断（不报错）。
  注意：默认底层存储是固定的长度 (不够则用空格补齐)，但是查询数据时，会自动将空白去除。如果想要保留空白，在`sql-mode`中加入`PAD_CHAR_TO_FULL_LENGTH`即可查看模式`sql-mode`，执行命令: `SHOW VARIABLES LIKE 'sql-mode'`;
  一般适用于：固定长度的内容

- **VARCHAR** `VARCHAR(m)`

  变长字符串，m代表字符串的长度，最多可容纳65535个字节
  变长的体现：内容小于m时，会按照真实数据长度存储；如果超出m长度限制（默认MySQL是严格模式，所以会报错）。
  如果在配置文件中加入如下配置，`sql-mode="NO_AUTO_CREATE_USER, NO_ENGINE_SUBSTITUTION"`保存并重启，此时MySQL则是非严格模式，此时超过长度则自动截断（不报错）

操作表

MySQL数据表中的列叫做**字段（Field）**，行叫做**记录（Record）**

1. 创建表：`CREATE TABLE 表名(列名 列类型);`

```
-- 选择 testdb 数据库
USE testdb;

-- 创建表
CREATE TABLE student(
    -- s_id INT AUTO_INCREMENT PRIMARY KEY 也表示
```

```
    s_id INT AUTO_INCREMENT NOT NULL,    -- AUTO_INCREMENT 表示自增
                                         -- NOT NULL 表示非空
    s_name VARCHAR(8) NOT NULL,
    s_age INT NOT NULL,
    s_class INT DEFAULT 3,               -- DEFAULT 3 表示默认值为 3
    s_email VARCHAR(32) NULL,            -- NULL 表示允许为空(默认)
    s_id2 INT,
    PRIMARY KEY(s_id, s_id2)             -- PRIMARY KEY 设置主键, PRIMARY(s_id,
s_id2)表示设置多列为主键
)DEFAULT CHARSET=utf8;
```

主键一般用于表示当前这条数据的ID编号 (类似于人的身份证)，需要维护一个**不重复**的值，比较繁琐。所以，在数据库中一般会将主键`PRIMARY KEY`和自增`AUTO_INCREMENT`结合。

注意：一个表中只能有一个自增列，一般都是主键

2. 查看表结构：`DESCRIBE 表名`或简写为`DESC 表名`/`SHOW CREATE TABLE 表名 [\G][\g]`

使用DESCRIBE

```
-- 查看表结构
DESCRIBE student;
```

结果如下：

```
+---------+-------------+------+-----+---------+----------------+
| Field   | Type        | Null | Key | Default | Extra          |
+---------+-------------+------+-----+---------+----------------+
| s_id    | int         | NO   | PRI | NULL    | auto_increment |
| s_name  | varchar(8)  | NO   |     | NULL    |                |
| s_age   | int         | NO   |     | NULL    |                |
| s_class | int         | YES  |     | 3       |                |
| s_email | varchar(32) | YES  |     | NULL    |                |
| s_id2   | int         | NO   | PRI | NULL    |                |
+---------+-------------+------+-----+---------+----------------+
```

其中:

`Null`表示该列是否可以存储`NULL`值;

`Key`表示该列是否已编制索引，`PRI`表示该列是表主键的一部分，`UNI`表示该列是`UNIQUE`索引的一部分，`MUL`表示在列中某个给定值允许出现多次;

`Default`表示该列是否有默认值，如果有，值是多少;

`Extra`表示可以获取的与给定列有关的附加信息，如`AUTO_INCREMENT`等。

使用SHOW CREATE TABLE

```
-- 使用 SHOW CREATE TABLE 表名 [\G][\g]也可以查看表结构
-- SHOW CREATE TABLE 命令会以 SQL 语句的形式来展示表信息
-- 与 DESCRIBE 相比, SHOW CREATE TABLE 展示的内容更加丰富, 它可以查看表的存储引擎和字
符编码
SHOW CREATE TABLE student \G;    -- \G 和 \g 为可选参数
```

结果如下：

```
*************************** 1. row ***************************
       Table: student
Create Table: CREATE TABLE `student` (
  `s_id` int NOT NULL AUTO_INCREMENT,
  `s_name` varchar(8) NOT NULL,
  `s_age` int NOT NULL,
  `s_class` int DEFAULT '3',
  `s_email` varchar(32) DEFAULT NULL,
  `s_id2` int NOT NULL,
  PRIMARY KEY (`s_id`,`s_id2`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb3
1 row in set (0.00 sec)
```

3. 修改字段`ALTER TABLE`
   - 添加列`ADD`

```
-- 添加一个新的字段，默认在表的最后位置添加
ALTER TABLE 表名 ADD 列名 类型;
-- 在开头位置添加一个新的字段
ALTER TABLE 表名 ADD 列名 类型 FIRST;
-- 在已经存在的字段名后添加一个新的字段
ALTER TABLE 表名 ADD 列名 类型 AFTER 已经存在的字段名;
-- 其他的一些方式
ALTER TABLE 表名 ADD 列名 类型 DEFAULT 默认值;
ALTER TABLE 表名 ADD 列名 类型 NOT NULL DEFAULT 默认值;
ALTER TABLE 表名 ADD 列名 类型 NOT NULL PRIMARY KEY AUTO_INCREMENT;
```

```
-- 添加列
ALTER TABLE student ADD s_passwd VARCHAR(16) NOT NULL DEFAULT "123456";
```

   - 删除列`DROP`

```
ALTER TABLE 表名 DROP COLUMN 列名;
```

   - 修改列 类型`MODIFY`

```
ALTER TABLE 表名 MODIFY COLUMN 列名 类型;
```

   - 修改列 类型 + 名称`CHANGE`

```
ALTER TABLE 表名 CHANGE 原列名 新列名 新类型;
```

```
-- 例如
ALTER TABLE student CHANGE s_id s_id INT NOT NULL;
ALTER TABLE student CHANGE s_id s_id INT NOT NULL DEFAULT 5;
ALTER TABLE student CHANGE s_id s_id INT NOT NULL PRIMARY KEY
AUTO_INCREMENT;
ALTER TABLE student CHANGE s_id s_id INT NULL;  -- 允许为空，删除默认值，删
除自增
```

- 修改列默认值`ALTER`

```
ALTER TABLE 表名 ALTER 列名 SET DEFAULT 默认值;
```

- 删除列默认值

```
ALTER TABLE 表名 ALTER 列名 DROP DEFAULT;
```

- 添加主键

```
ALTER TABLE 表名 ADD PRIMARY KEY (列名);
```

- 删除主键

```
ALTER TABLE 表名 DROP PRIMARY KEY (列名);
```

4. 删除表：`DROP TABLE 表名`

```
-- 删除表
DROP TABLE student;
```

5. 清空表：`DELETE FROM 表名`或`TRUNCATE TABLE 表名`

```
-- DELETE FROM 方式
DELETE FROM student;          -- 不删除表，仅删除表中的数据

-- TRUNCATE TABLE 方式
TRUNCATE TABLE student;       -- 比 DELETE FROM 速度快，但是无法撤销和回滚
```

操作数据（增删改查）

1. 新增数据：`INSERT INTO`

```
-- 方式 1
-- 插入
INSERT INTO 表名 (列名1, 列名2, ..., 列名n) VALUES (对应列1的值, 对应列2的值, ..., 对应列n的值);
-- 快速地从一个或多个表中取出数据，并将这些数据作为行数据插入另一个表中
INSERT INTO 表名 (列名1, 列名2, ..., 列名n) SELECT 其他列名1, 其他列名2, ..., 其他列名n FROM 其他表;

-- 方式 2
INSERT INTO 表名 SET 列名1=值1, 列名2=值2, ..., 列名n=值n;
```

```
-- 演示插入数据
CREATE TABLE test_tb(
    t_id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    t_name VARCHAR(8) NOT NULL,
    t_phone CHAR(11) NOT NULL
);
```

```
INSERT INTO test_tb (t_name, t_phone) VALUES ("张三", "12345678910");
INSERT INTO test_tb (t_name, t_phone) VALUES ("李四", "12345678910");
INSERT INTO test_tb (t_name, t_phone) VALUES ("王五", "12345678910"), ("刘六",
"12345678910");
INSERT INTO test_tb VALUES (6, "赵启", "12345678910"), (7, "勾八",
"12345678910");  -- 只有三列

INSERT INTO test_tb SET t_name="牛牛", t_phone="12345678910";

-- 查看表
SELECT * FROM test_tb;
```

结果如下：

```
+------+--------+-------------+
| t_id | t_name | t_phone     |
+------+--------+-------------+
|    1 | 张三   | 12345678910 |
|    2 | 李四   | 12345678910 |
|    3 | 王五   | 12345678910 |
|    4 | 刘六   | 12345678910 |
|    6 | 赵启   | 12345678910 |
|    7 | 勾八   | 12345678910 |
|    8 | 牛牛   | 12345678910 |
+------+--------+-------------+
7 rows in set (0.00 sec)
```

2. 删除数据

```
-- 方式 1
DELETE FROM 表名 WHERE 子句 ORDER BY 子句 LIMIT 子句;

-- 方式 2
TRUNCATE TABLE 表名;
```

在方式1中：

ORDER BY 子句：可选项。表示删除时，表中各行将按照子句中指定的顺序进行删除。

WHERE 子句：可选项。表示为删除操作限定删除条件，若省略该子句，则代表删除该表中的所有行。

LIMIT 子句：可选项。用于告知服务器在控制命令被返回到客户端前被删除行的最大值。

```
-- 删除数据
DELETE FROM test_tb WHERE t_id=1;
```

结果如下：

```
+------+--------+-------------+
| t_id | t_name | t_phone     |
+------+--------+-------------+
|    2 | 李四   | 12345678910 |
|    3 | 王五   | 12345678910 |
|    4 | 刘六   | 12345678910 |
|    6 | 赵启   | 12345678910 |
|    7 | 勾八   | 12345678910 |
|    8 | 牛牛   | 12345678910 |
+------+--------+-------------+
6 rows in set (0.00 sec)
```

3. 修改数据

```
UPDATE 表名 SET 列名=值;
UPDATE 表名 SET 列名=值 WHERE 子句;
UPDATE 表名 SET 列名1=值1, 列名2=值2, ..., 列名n=值n WHERE 子句 ORDER BY 子句
LIMIT 子句;
```

SET 子句：用于指定表中要修改的列名及其列值。其中，每个指定的列值可以是表达式，也可以是该列对应的默认值。如果指定的是默认值，可用关键字 DEFAULT 表示列值。

WHERE 子句：可选项。用于限定表中要修改的行。若不指定，则修改表中所有的行。

ORDER BY 子句：可选项。用于限定表中的行被修改的次序。

LIMIT 子句：可选项。用于限定被修改的行数。

4. 查询数据

```
SELECT * FROM 表名;
SELECT 列名1, 列名2, ..., 列名n FROM 表名;
SELECT 列名1 AS 别名1, 列名2 AS 别名2, ..., 列名n AS 别名n FROM 表名;
SELECT 列名1, 列名2, ..., 列名n FROM 表名 WHERE 条件;
```

```
SELECT t_id AS id, t_name AS name, t_phone AS phone FROM test_tb;
```

结果如下：

```
+----+------+-------------+
| id | name | phone       |
+----+------+-------------+
|  2 | 李四 | 12345678910 |
|  3 | 王五 | 12345678910 |
|  4 | 刘六 | 12345678910 |
|  6 | 赵启 | 12345678910 |
|  7 | 勾八 | 12345678910 |
|  8 | 牛牛 | 12345678910 |
+----+------+-------------+
6 rows in set (0.00 sec)
```

5. [拓展] 使用Python结合SQL完成用户登录系统

```sql
-- 创建数据库
CREATE DATABASE usersdb DEFAULT CHARSET utf8 COLLATE utf8_general_ci;

-- 使用数据库
USE userdb;

-- 创建表
CREATE TABLE users(
    id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(32).
    password VARCHAR(64)
)DEFAULT CHARSET=utf8;
```

```python
import pymysql

def register():
    print('用户注册')
    user = input('请输入用户名:')
    password = input('请输入密码:')
    # 连接指定数据库
    conn = pymysql.connect(
        host='127.0.0.1',
        port=3306,
        user='root',
        passwd='root',
        charset="utf8',
        db='usersdb')
    cursor = conn.cursor()

    # 执行SQL语句 (有SQL注入风险，稍后讲解)
    sql = 'insert into users(name,password) values("()","()")'.format(user,
password)
    cursor.execute(sql)
    conn.commit()

    # 关闭数据库连接
    cursor.close()
    conn .close()
    print("注册成功，用户名:{}, 密码:{}".format(user, password))

def login():
    print('用户登录')
    user = input('请输入用户名:')
    password = input('请输入密码:')
    # 连接指定数据库
    conn = pymysql.connect(
        host='127.0.0.1',
        port=3306,
        user='root',
        passwd='root',
        charset="utf8',
        db='usersdb')
    cursor = conn.cursor()
```

```
    # 执行SQL语句 (有SQL注入风险，稍后讲解)
    sql = "select * from users where name='{}' and
password='{}'".format(user, password)
    cursor.execute(sql)
    result = cursor.fetchone()

    # 关闭数据库连接
    cursor.close()
    conn.close()
    if result:
        print('登录成功', result)
    else:
        print('登录失败')
```

SQL 注入

如果用户在输入user时，输入了`' or 1=1 --`，这样即使用户输入的密码不存在，也会通过验证。

SQL语句会变成这样：`SELECT * FROM users WHERE name='' or 1=1 -- AND password={}`

可以看到`--`后的内容备注释掉了。

如何避免：`cursor.execute("SELECT * FROM users WHERE name=%s AND password=%s", [user, pwd])`使用这种方式可以避免SQL注入。

常用操作

```
-- 数据库准备
CREATE DATABASE empdb DEFAULT CHARSET utf8 COLLATE utf8_general_ci;

-- 进入数据库
USE empdb;

-- 数据表准备
-- 部门表
CREATE TABLE depart(
    id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
    title VARCHAR(16) NOT NULL
)DEFAULT CHARSET=utf8;

-- 员工信息表
CREATE TABLE info(
    id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(16) NOT NULL,
    email VARCHAR(32) NOT NULL,
    age INT,
    depart_id INT
)DEFAULT CHARSET=utf8;

-- 插入数据
INSERT INTO depart(title) VALUES ("开发"), ("运营"), ("销售");

INSERT INTO info(name, email, age, depart_id) VALUES ("武沛齐",
"wupeigilive,com", 19, 1);
INSERT INTO info(name, email, age, depart_id) VALUES ("于超", "pyyu@live.com",
49, 1);
```

```
INSERT INTO info(name, email, age, depart_id) VALUES ("alex", "alex@live,com",
9, 2);
INSERT INTO info(name, email, age, depart_id) VALUES ("tony", "tony@live.com",
29, 1);
INSERT INTO info(name, email, age, depart_id) VALUES ("kelly", "kelly@live.com",
99, 3);
INSERT INTO info(name, email, age, depart_id) VALUES ("james", "james@live,com",
49, 1);
INSERT INTO info(name, email, age, depart_id) VALUES ("李杰", "lijie@live.com",
49, 1);
```

结果如下：

```
+----------------+
| Tables_in_empdb |
+----------------+
| depart          |
| info            |
+----------------+
2 rows in set (0.00 sec)

+----+-------+
| id | title |
+----+-------+
|  1 | 开发  |
|  2 | 运营  |
|  3 | 销售  |
+----+-------+
3 rows in set (0.00 sec)

+----+--------+-----------------+------+-----------+
| id | name   | email           | age  | depart_id |
+----+--------+-----------------+------+-----------+
|  1 | 武沛齐 | wupeigilive,com |   19 |         1 |
|  2 | 于超   | pyyu@live.com   |   49 |         1 |
|  3 | alex   | alex@live,com   |    9 |         2 |
|  4 | tony   | tony@live.com   |   29 |         1 |
|  5 | kelly  | kelly@live.com  |   99 |         3 |
|  6 | james  | james@live,com  |   49 |         1 |
|  7 | 李杰   | lijie@live.com  |   49 |         1 |
+----+--------+-----------------+------+-----------+
7 rows in set (0.00 sec)
```

- 条件操作

```
SELECT * FROM info WHERE age >= 30;             -- 查找年龄大于 30 的
SELECT * FROM info WHERE id BETWEEN 2 AND 4;    -- 查找 ID 在 2 到 4 之间的
SELECT * FROM info WHERE id IN (1, 4, 6);       -- 查找 ID 在 1, 4 和 6 之间的
SELECT * FROM info WHERE id NOT IN (1, 4, 6);   -- 查找 ID 不在 1, 4 和 6 之间
的

SELECT id FROM depart;  -- +----+
                        -- | id |
                        -- +----+
```

```
                        -- |  1 |
                        -- |  2 |
                        -- |  3 |
                        -- +----+
SELECT * FROM info WHERE depart_id IN (SELECT id FROM depart);  -- 查找
depart_id 在 depart 表中的员工信息

SELECT * FROM info WHERE id > 5;                                    -- 下
面语句，先找 info 中 id>5 的表
SELECT * FROM (SELECT * FROM info WHERE id > 5) AS T WHERE T.age > 10;  -- 将
表作为 T
SELECT * FROM (SELECT * FROM info WHERE id > 5) AS T WHERE T.age = 10;
```

- 通配符

  一般用于模糊搜索，有`%`和`_`两种。

  `%`表示`n`个字符，`_`表示单个字符。

```
SELECT * FROM info WHERE email LIKE '%@%';
SELECT * FROM info WHERE name LIKE '___';
```

  适用于较小规模数据的搜索，不适用于较大规模数据的搜索（如：搜索引擎）。

- 指定列（映射）

```
SELECT * FROM info;             -- 普通查询
SELECT id, name FROM info;      -- 最好少写 *, 效率过低
```

```
SELECT id, name AS NM, 123 AS ADD_NUM FROM info;
```

```
+----+--------+---------+
| id | NM     | ADD_NUM |
+----+--------+---------+
|  1 | 武沛齐 |     123 |
|  2 | 于超   |     123 |
|  3 | alex   |     123 |
|  4 | tony   |     123 |
|  5 | kelly  |     123 |
|  6 | james  |     123 |
|  7 | 李杰   |     123 |
+----+--------+---------+
7 rows in set (0.00 sec)
```

```
SELECT id, name, 666 AS num,
    (SELECT max(id) FROM depart) AS MID,    -- 展示 depart 中 id 的最大值为 MID
列
    (SELECT min(id) FROM depart) AS NID,    -- 展示 depart 中 id 的最小值为 NID
列
    age
FROM info;
```

```
+----+--------+-----+------+------+------+
| id | name   | num | MID  | NID  | age  |
+----+--------+-----+------+------+------+
|  1 | 武沛齐 | 666 |    3 |    1 |   19 |
|  2 | 于超   | 666 |    3 |    1 |   49 |
|  3 | alex   | 666 |    3 |    1 |    9 |
|  4 | tony   | 666 |    3 |    1 |   29 |
|  5 | kelly  | 666 |    3 |    1 |   99 |
|  6 | james  | 666 |    3 |    1 |   49 |
|  7 | 李杰   | 666 |    3 |    1 |   49 |
+----+--------+-----+------+------+------+
7 rows in set (0.00 sec)
```

```
SELECT id, name,
    -- 查找 depart表的id = info表的depart_id 的列, 将这一列表头显示为 x1
    (SELECT title FROM depart WHERE depart.id=info.depart_id) AS x1,
    -- 查找 depart表的id = info表的id 的列, 将这一列表头显示为 x2
    (SELECT title FROM depart WHERE depart.id=info.id) AS x2
FROM info;  -- 效率很低
```

```
+----+--------+------+------+
| id | name   | x1   | x2   |
+----+--------+------+------+
|  1 | 武沛齐 | 开发 | 开发 |
|  2 | 于超   | 开发 | 运营 |
|  3 | alex   | 运营 | 销售 |
|  4 | tony   | 开发 | NULL |
|  5 | kelly  | 销售 | NULL |
|  6 | james  | 开发 | NULL |
|  7 | 李杰   | 开发 | NULL |
+----+--------+------+------+
7 rows in set (0.00 sec)
```

```
SELECT
    id,
    name,
    CASE depart_id WHEN 1 THEN "第一部门" END v1,    -- 取 depart_id
                                                     -- 当 depart_id=1 时, 显
示"第一部门"在 v1 列
                                                     -- 否则, 显示"NULL"在 v1 列
    CASE depart_id WHEN 1 THEN "第一部门" ELSE "其他" END v2,
    CASE depart_id WHEN 1 THEN "第一部门" WHEN 2 THEN "第二部门" ELSE "其他" END
v3,
    CASE WHEN age<18 THEN "少年" END v4,
    CASE WHEN age<18 THEN "少年" ELSE "油腻男" END v5,
    CASE WHEN age<18 THEN "少年" WHEN age<30 THEN "青年" ELSE "油腻男" END v6
FROM info;
```

- 排序

```
SELECT * FROM info ORDER BY age DESC;            -- 倒序排列，从大到小
SELECT * FROM info ORDER BY age ASC;             -- 倒序排列，从小到大
SELECT * FROM info ORDER BY age ASC, id DESC;    -- 先按照 age 从小到大排序，当
age 相同时，按照 id 从大到小排列
```

- `LIMIT`取部分数据

```
SELECT * FROM info LIMIT 5;                      -- 只获取前 5 行数据
SELECT * FROM info ORDER BY id DESC LIMIT 3;     -- 先排序，再获取前三条数据
SELECT * FROM info LIMIT 3 OFFSET 2;             -- 从位置 2开始，向后取 3条数据
```

- 分组：`GROUP BY`

```
SELECT name, age, MAX(id), MIN(id), SUM(id), AVG(id), COUNT(1) FROM info
GROUP BY age;
```

需要注意：单独使用`GROUP BY`关键字时，查询结果会**只显示每个分组的第一条记录**（因此不太推荐这种方式）。结果如下：

```
+--------+------+---------+---------+---------+---------+----------+
| name   | age  | MAX(id) | MIN(id) | SUM(id) | AVG(id) | COUNT(1) |
+--------+------+---------+---------+---------+---------+----------+
| 武沛齐 |   19 |       1 |       1 |       1 |  1.0000 |        1 |
| 于超   |   49 |       7 |       2 |      15 |  5.0000 |        3 |
| alex   |    9 |       3 |       3 |       3 |  3.0000 |        1 |
| tony   |   29 |       4 |       4 |       4 |  4.0000 |        1 |
| kelly  |   99 |       5 |       5 |       5 |  5.0000 |        1 |
+--------+------+---------+---------+---------+---------+----------+
5 rows in set (0.00 sec)
```

可以使用`GROUP_CONCAT()`函数把每个分组的字段值都显示出来。

```
SELECT GROUP_CONCAT(name), age, MAX(id), MIN(id), SUM(id), AVG(id), COUNT(1)
FROM info GROUP BY age;
```

结果如下：

```
+--------------------+------+---------+---------+---------+---------+-------
---+
| GROUP_CONCAT(name) | age  | MAX(id) | MIN(id) | SUM(id) | AVG(id) |
COUNT(1) |
+--------------------+------+---------+---------+---------+---------+-------
---+
| alex               |    9 |       3 |       3 |       3 |  3.0000 |
 1 |
| 武沛齐             |   19 |       1 |       1 |       1 |  1.0000 |
 1 |
| tony               |   29 |       4 |       4 |       4 |  4.0000 |
 1 |
| 于超,james,李杰    |   49 |       7 |       2 |      15 |  5.0000 |         3
|
| kelly              |   99 |       5 |       5 |       5 |  5.0000 |
 1 |
+--------------------+------+---------+---------+---------+---------+-------
---+
5 rows in set (0.00 sec)
```

如果想对分完组后的数据再进行筛选（搜索），此时不能使用`WHERE`，而需要使用`HAVING`。

```
SELECT age, count(id) FROM info WHERE id>4 GROUP BY age HAVING COUNT(id)>=2;
```

结果如下：

```
+------+-----------+
| age  | count(id) |
+------+-----------+
|   49 |         2 |
+------+-----------+
1 row in set (0.00 sec)
```

SQL语句执行的优先级：

```
WHERE > GROUP BY > HAVING > ORDER BY > LIMIT
```

```
-- 要查询的表: info
-- 条件 id>2
-- 按照 age 进行分组
-- 对于分组后的数据再根据聚合条件过滤
-- 根据 age 从大到小排下序
-- 获取第一条数据
SELECT age, COUNT(id) FROM info WHERE id>2 GROUP BY age HAVING COUNT(id)>1
ORDER BY age DESC LIMIT 1;
```

- 左右连表：需要注意主表和从表之间的关系。

```
主表 LEFT OUTER JOIN 从表 ON 主表.xx = 从表.xx;      -- LEFT 表示左边为主表，外连接
从表 RIGHT OUTER JOIN 主表 ON 主表.xx = 从表.xx;     -- RIGHT 表示右边为主表
```

```
SELECT * FROM info LEFT OUTER JOIN depart ON info.depart_id = depart.id;
SELECT info.id, info.name, info.email, depart.title FROM info LEFT OUTER
JOIN depart ON info.depart_id = depart.id;
```

`OUTER JOIN`为外连接，`INNER JOIN`为内连接。

内连接会忽略那些没有对应关系的列。

SQL语句执行的优先级：

```
FROM > JOIN > ON > WHERE > GROUP BY > HAVING > SELECT > ORDER BY > LIMIT
```

- 上下连表：`UNION`很少使用

```
-- 列数相同即可，类型无需相同
SELECT id, title FROM depart
UNION
SELECT id, name FROM info;
```

```
-- 自动去重
SELECT id FROM depart
UNION
SELECT id FROM info;
```

```
-- 保留全部，不希望去重
SELECT id FROM depart
UNION ALL
SELECT id FROM info;
```

表关系

- 单表：单独的一张表就可以将信息保存
- 一对多关系：需要两张表来存储信息，且两张表存在`一对多`或`多对一`关系

  例如：有两张表，`info`和`depart`如下：

  info表

id	name	email	age	depart_id
...	...	...	...	...

  depart表

id	title
...	...

  `info`表中的`depart_id`字段对应的是`depart`表中的`id`字段。

  在开发中往往还会为他们添加一个**外键约束**，保证某一个列的值必须是其他表中的特定列已存在的值，例如:`info.depart_id`
  的值必须是`depart.id`中已存在的值。

- 创建时添加外键

```
-- 在创建时添加外键
CREATE TABLE depart(
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(16) NOT NULL
)DEFAULT CHARSET=utf8;

CREATE TABLE info(
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name varchar(16) NOT NULL,
    email varchar(32) NOT NULL,
    age INT,
    depart_id INT NOT NULL,
    -- 添加外键，外键一般命名为 fk_源表名_对应表名
    CONSTRAINT fk_info_depart FOREIGN KEY (depart_id) REFERENCES depart(id)
)DEFAULT CHARSET=utf8;
```

- 表结构已经创建完毕，额外想添加外键

```
-- 如果表已经创建好了，额外想添加外键
ALTER TABLE info ADD CONSTRAINT fk_info_depart FOREIGN KEY info(depart_id)
REFERENCES depart(id);
```

- 删除外键

```
-- 删除外键
ALTER TABLE info DROP FOREIGN KEY fk_info_depart;
```

- 多对多关系：需要三张表来存储信息，两张`单表`+`关系表`，构造出两个单表之间的多对多关系

```
CREATE TABLE boy(
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(16) NOT NULL
)DEFAULT CHARSET=utf8;

CREATE TABLE girl(
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(16) NOT NULL
)DEFAULT CHARSET=utf8;

-- 创建时添加外键
CREATE TABLE boy_girl(
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    boy_id INT NOT NULL,
    girl_id INT NOT NULL,
    CONSTRAINT fk_boy_girl_boy FOREIGN KEY boy_girl(boy_id) REFERENCES
boy(id),
    CONSTRAINT fk_boy_girl_girl FOREIGN KEY boy_girl(girl_id) REFERENCES
girl(id)
)DEFAULT CHARSET=utf8;
```

```
-- 如果表已经创建好了，额外想添加外键
ALTER TABLE boy_girl ADD CONSTRAINT fk_boy_girl_boy FOREIGN KEY
boy_girl(boy_id) REFERENCES boy(id);
ALTER TABLE boy_girl ADD CONSTRAINT fk_boy_girl_girl FOREIGN KEY
boy_girl(girl_id) REFERENCES girl(id);
```

用户授权

在MySQL的默认数据库`mysql`中的`user`表中，存储所有的账户信息（账户、权限等）。可以使用`SELECT`语句进行相关的查询。

```
USE mysql;

SELECT user, authentication_string, host FROM user;
```

```
+------------------+-------------------------------------------------------
-----------+-----------+
| user             | authentication_string
          | host      |
+------------------+-------------------------------------------------------
-----------+-----------+
| mysql.infoschema |
*************************************************************************** |
localhost |
| mysql.session    |
*************************************************************************** |
localhost |
| mysql.sys        |
*************************************************************************** |
localhost |
| root             |
*************************************************************************** |
localhost |
+------------------+-------------------------------------------------------
-----------+-----------+
```

- 创建和删除用户

  ```
  CREATE USER '用户名'@'连接者的IP地址' IDENTIFIED BY '密码';
  ```

  ```
  CREATE USER 'NILERA'@'127.0.0.%' IDENTIFIED BY '******';
  ```

```
+------------------+-----------------------------------------------------
---------------+-----------+
| user             | authentication_string
               | host      |
+------------------+-----------------------------------------------------
---------------+-----------+
| NILERA           |
*********************************************************************** |
127.0.0.% |
| mysql.infoschema |
*********************************************************************** |
localhost |
| mysql.session    |
*********************************************************************** |
localhost |
| mysql.sys        |
*********************************************************************** |
localhost |
| root             |
*********************************************************************** |
localhost |
+------------------+-----------------------------------------------------
---------------+-----------+
```

- 修改用户

```
RENAME USER "用户名"@'IP地址' TO '新用户名'@'IP地址';
```

- 修改密码

```
SET PASSWORD FOR '用户名'@'IP地址' = PASSWORD('新密码');
```

- 授权

```
GRANT 权限 ON 数据库.表 TO '用户'@'IP地址';
```

```
GRANT ALL PRIVILEGES ON *.* TO 'NILERA'@'localhost';     -- 给予用户NILERA所有数
据库中所有表的所有权限
GRANT SELECT *.* TO 'NILERA'@'localhost';                -- 给予用户NILERA所有数
据库中所有表的查询权限

FLUSH PRIVILEGES;    -- 修改的权限立即生效
```

- 查看权限

```
SHOW GRANTS FOR '用户'@'IP地址';
```

结果如下:

```
+-------------------------------------------+
| Grants for NILERA@127.0.0.%               |
+-------------------------------------------+
| GRANT USAGE ON *.* TO `NILERA`@`127.0.0.%` |
+-------------------------------------------+
1 row in set (0.00 sec)
```

- 取消授权

```
REVOKE 权限 ON 数据库.表 FROM '用户'@'IP地址';
```

索引

对于数据库中有大量数据的情况，例如有三百万条数据，不使用索引可能需要700秒的时间进行查询，使用索引可能只需要不到1秒时间。

索引的原理：为什么加上索引之后速度能有这么大的提升呢? 因为索引的底层是基于**B+Tree**的数据结构存储的。

关于**B+树：**一文彻底搞懂MySQL基础：B树和B+树的区别_mysql b树和b+树的区别-CSDN博客

彻底搞懂MySQL的主键索引、普通索引和联合索引 - 知乎 (zhihu.com)

数据库的索引是基于上述B+Tree的数据结构实现，但在创建数据库表时，如果指定不同的引擎，底层使用的B+Tree结构的原理有些不同。

- MYISAM引擎，非聚簇索引（数据和索引结构分开存储），简单理解就是，存在一张物理表，也存在一张索引表，即“有表有树”。
- INNODB引擎，聚簇索引（数据和主键索引结构存储在一起），简单理解就是，不存在一张物理表，数据和索引存在一起，即“无表有树”。

上述聚索引和非聚索引底层均利用了**B+Tree**结构结构，只不过内部数据存储有些不同。在企业开发中一般都会使用**INNODB**引擎（内部支持事务、行级锁、外键等特点），在MySQL5.5版本之后默认引擎也是**INNODB**。

常见索引

- 主键索引：加速查找、不能为空、不能重复

```
-- 方式1，直接指定主键
CREATE TABLE 表名(
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(32) NOT NULL
);

-- 方式2，单独指定主键
CREATE TABLE 表名(
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(32) NOT NULL,
    PRIMARY KEY(id)
);

-- 方式3，联合主键
CREATE TABLE 表名(
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
```

```
    name VARCHAR(32) NOT NULL,
    PRIMARY KEY(id, name)    -- 联合主键
                             -- 仅举例使用，该方法 name 不能重复
);
```

- 唯一索引：加速查找、不能重复

```
-- 方式1，创建表时直接创建索引
CREATE TEBLE 表名(
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(32) NOT NULL,
    email VARCHAR(32) NOT NULL,
    UNIQUE ix_name(name),        -- 为 name 列创建唯一索引
    UNIQUE ix_email(email)       -- 为 email 列创建唯一索引
);

-- 方式2，创建表时创建联合唯一索引
CREATE TEBLE 表名(
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(32) NOT NULL,
    email VARCHAR(32) NOT NULL,
    UNIQUE(name, email)          -- 联合唯一索引
);

-- 方式3，给已经创建好的表添加索引
CREATE UNIQUE INDEX 索引名 ON 表名(列名);
```

方式1和方式2的区别：方式1，采用分别创建唯一索引的方式，创建索引的每一列都不允许存在重复值；方式2，采用联合唯一索引的方式，表示联合起来的列中的记录不能有重复值，即`(列名1, 列名2, ..., 列名n)`中的每一列可以有重复值，但是联合起来的记录中，不能有重复值。

```
-- 删除索引
DROP UNIQUE INDEX 索引名 ON 表名;
```

- 普通索引：加速查找

```
-- 方式1，创建表时直接创建索引
CREATE TEBLE 表名(
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(32) NOT NULL,
    email VARCHAR(32) NOT NULL,
    INDEX ix_name(name),         -- 为 name 列创建索引
    INDEX ix_email(email)        -- 为 email 列创建索引
);

-- 方式2，创建表时创建联合唯一索引
CREATE TEBLE 表名(
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(32) NOT NULL,
    email VARCHAR(32) NOT NULL,
    INDEX ix_email(name, email) -- 联合唯一索引
);
```

```
-- 方式3，给已经创建好的表添加索引
CREATE INDEX 索引名 ON 表名(列名);
```

```
-- 删除索引
DROP INDEX 索引名 ON 表名;
```

- 需要注意：查询最好要命中索引，有些情况可能会导致无法命中索引。
  - 类型不匹配：建立索引的列，如`name`列，是`VARCHAR`类型，但是查询的时候给了整数类型，就会导致无法命中。
  - 使用不等于：仅针对非主键，主键使用不等于也会命中。
  - 使用`OR`：当or条件中有未建立索引的列才失效。
  - 使用排序：当根据索引排序时候，选择的映射如果不是索引，则不走索引。
  - 使用`LIKE`：模糊匹配时。
  - 使用函数：函数在前无法命中。
  - 最左前缀问题：如果是联合索引，要遵循最左前缀原则。

    ```
    -- 如果联合索引为: (name, password)
    name AND password;      -- 命中
    name;                   -- 命中
    password;               -- 未命中
    name OR password;       -- 未命中
    ```

- 为了评估或者判断SQL语句执行的速度，可以使用**执行计划**`explain SQL语句`

  ```
  EXPLAIN SELECT * FROM test_tb;
  ```

  结果如下：

id	select_type	table	partitions	types	possible_keys	key	key_llen	ref	rows	filtered	EXtra
1	SIMPLE	test_tb	(NULL)	ALL	(NULL)	(NULL)	(NULL)	(NULL)	6	100.00	(NULL)

  需要关注：`type`列，性能从低到高依次为：`ALL < INDEX < RANGE < INDEX_MERGE < REF_OR_NULL < REF < EQ_REF < SYSTEM/CONST`

函数

存储过程

存储过程，是一个存储在MySQL中的SQL语句集合，当主动去调用存储过程时，其中内部的SQL语句会按照逻辑执行。

```
-- 可以通过 delimiter 更改语句结束符
-- 默认情况下，MySQL 解释器每遇到一个分号，执行一段语句
-- 但有时候，不希望 MySQL 这么做，如在输入较多的语句，且语句中包含有分号
-- 将结束符 ; 更改为 $$
DELIMITER $$

-- 将结束符 $$ 更改为 ;
DELIMITER ;
```

```
-- 创建存储过程
DELIMITER $$
CREATE PROCEDURE p1()
BEGIN
    SELECT * FROM test_tb;
END $$
DELIMITER ;
```

```
-- 执行存储过程
CALL p1();
```

```
-- 删除存储过程
DROP PROCEDURE p1();
```

如何写出更高级的存储过程：

- 存储过程的参数

  存储过程的参数有三种：`in`仅作为传入参数、`out`仅作为返回值、`inout`既可以作为传入又可以当作返回值。

  ```
  DELIMITER $$
  CREATE PROCEDURE p2(
      IN i1 INT,
      IN i2 INT,
      INOUT i3 INT,
      OUT r1 INT
  )
  BEGIN
      DECLARE temp1 INT;              -- 声明 temp1
      DECLARE temp2 INT DEFAULT 0;    -- 声明 temp2

      SET temp1 = 1;                  -- 设置 temp1 的值为 1
      SET r1 = i1 + i2 + temp1 + temp2;
      SET i3 = i3 + 100;
  END $$
  DELIMITER ;
  ```

  ```
  SET @t1 = 4;        -- 定义一个用户级的变量 t1
  SET @t2 = 0;        -- 定义一个用户级的变量 t1
  CALL p2(1, 2, @t1, @t2);
  SELECT @t1, @t2;
  ```

- 返回值与结果集（Python编程需注意，这里可以不做了解）

```
DELIMITER $$
CREATE PROCEDURE p3(
    IN i1 INT,
    IN i2 INT,
    INOUT i3 INT,
    OUT r1 INT
)
BEGIN
    DECLARE temp1 INT;              -- 声明 temp1
    DECLARE temp2 INT DEFAULT 0;    -- 声明 temp2

    SET temp1 = 1;                  -- 设置 temp1 的值为 1
    SET r1 = i1 + i2 + temp1 + temp2;
    SET i3 = i3 + 100;
    SELECT * FROM test_tb;
END $$
DELIMITER ;
```

```
SET @t1 = 4;        -- 定义一个用户级的变量 t1
SET @t2 = 0;        -- 定义一个用户级的变量 t1
CALL p3(1, 2, @t1, @t2);
SELECT @t1, @t2;
-- SELECT @_p3_0, @_p3_1, @_p3_2;
```

- 事务与异常

```
DELIMITER $$
CREATE PROCEDURE p4(
    OUT p_return_code TINYINT
)
BEGIN
    -- 定义异常/错误，当出现异常/错误时执行这部分代码
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
    BEGIN
        -- ERROR
        SET p_return_code = 1;
        ROLLBACK;
    END;

    -- 定义警告，当出现警告时执行这部分代码
    DECLARE EXIT HANDLER FOR SQLWARNING
    BEGIN
        -- WARNING
        SET p_return_code = 2;
        ROLLBACK;
    END;

    -- 创建事务，在存储过程中执行这些代码
    START TRANSACTION;
        DELETE FROM d1;
        INSERT INTO tb(name) VALUES ("SEVEN");
    COMMIT;
```

```
    -- SUCCESS, 当成功时设置 p_return_code 为 0
    SET p_return_code = 0;
    END$$
DELIMITER ;
```

```
SET @ret = 100;
CALL p4(@ret);
SELECT @ret;
```

- 游标（效率很低，很少使用，了解即可）

```
DELIMITER $$
CREATE PROCEDURE p5()
BEGIN
    DECLARE sid INT;
    DECLARE sname VARCHAR(50);
    DECLARE done INT DEFAULT false;

    DECLARE my_cursor CURSOR FOR SELECT id, name from test_tb;
    -- 定义继续句柄, 当无法找到新值时(NOT FOUND), 设置 done 的值为TRUE
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;
    OPEN my_cursor;
        xxoo: LOOP
            FETCH my_cursor INTO sid, sname;
            IF done THEN
                LEAVE xxoo;
            END IF;
            INSERT INTO tl(name) values(sname);
        END LOOP xxoo;
    CLOSE my_cursor;
END $$
DELIMITER ;
```

视图

视图其实是一个虚拟表(非真实存在)，其本质是**根据SQL语句获取动态的数据集，并为其命名**，用户使用时只需使用**名称**即可获取结果集，并可以将其**当作**表来使用。

注意：基于视图**只能查询**，针对视图不能执行增加、修改、删除。如果源表发生变化，视图表也会发生变化。

```
SELECT *
FROM (SELECT nid, name FROM tbl WHERE nid>2) AS A   -- 每次写这个太麻烦，于是就有了视图
WHERE A.name > 'alex';
```

- 创建视图

```
CREATE VIEW v1 AS SELECT id, name FROM test_tb WHERE id>1;
```

- 使用视图

```
SELECT * FROM v1;
```

- 删除视图

```
DROP VIEW v1;
```

- 修改视图

```
ALTER VIEW v1 AS SQL语句
```

触发器

对某个表进行**增/删/改**操作的前后如果希望触发某个特定的行为时，可以使用触发器。

```
-- 插入前
CREATE TRIGGER tri_before_insert_tb1 BEFORE INSERT ON tb1 FOR EACH ROW
BEGIN
    ...
END

-- 插入后
CREATE TRIGGER tri_after_insert_tb1 AFTER INSERT ON tb1 FOR EACH ROW
BEGIN
    ...
END

-- 删除前
CREATE TRIGGER tri_before_delete_tb1 BEFORE DELETE ON tb1 FOR EACH ROW
BEGIN
    ...
END

-- 删除后
CREATE TRIGGER tri_after_delete_tb1 AFTER DELETE ON tb1 FOR EACH ROW
BEGIN
    ...
END

-- 更新前
CREATE TRIGGER tri_before_update_tb1 BEFORE UPDATE ON tb1 FOR EACH ROW
BEGIN
    ...
END

-- 更新后
CREATE TRIGGER tri_after_update_tb1 AFTER UPDATE ON tb1 FOR EACH ROW
BEGIN
    ...
END
```

```
DELIMITER $$
CREATE TRIGGER tri_before_insert_t1 BEFORE INSERT ON t1 FOR EACH ROW
BEGIN
IF NEW.name = 'alex' THEN
    INSERT INTO t2 (name) VALUES (NEW.id);
END IF;
END $$
DELIMITER ;
```

```
DELIMITER $$
CREATE TRIGGER tri_before_insert_t1 BEFORE INSERT ON t1 FOR EACH ROW
BEGIN
IF OLD.name = 'alex' THEN
    INSERT INTO t2 (name) VALUES (OLD.id);
END IF;
END $$
DELIMITER ;
```

注意： `NEW`表示新数据，`OLD`表示原来的数据。

事务机制

如果有一件事需要两个步骤，两个步骤必须同时完成才算完成，并且如果第一步完成、第二步失败，则回滚到初始状态。

事务具有四大特性（ACID）：

- 原子性(Atomicity)

  原子性是指事务包含的所有操作不可分割，要么全部成功，要么全部失败回滚。

- 一致性(Consistency)

  执行的前后数据的完整性保持一致。

- 隔离性(Isolation)

  一个事务执行的过程中，不应该受到其他事务的干扰。

- 持久性(Durability)

  事务一旦结束，数据就持久到数据库。

```
-- 事务演示
BEGIN;      -- 开始事务，或者使用 START TRANSACATION;
UPDATE users SET amount=amount-2 WHERE id=1;    -- 执行操作
UPDATE users SET amount=amount+2 WHERE id=2;    -- 执行操作
COMMIT;     -- 提交事务，如果不想提交，而是想要回滚，则替换 COMMIT 为 ROLLBACK;
```