下载MySQL

安装教程，注意文件路径

2021MySql-8.0.26安装详细教程（保姆级）_2021mysql-8.0.26安装详细教程(保姆级)_mysql8.0.26_ylb呀的博客-cs-CSDN博客

下载MySQL Workbench

MySQL Workbench 安装及使用-CSDN博客

1.SQLAlchemy介绍

1.1 mysql驱动包

Flask 想要操作数据库，必须要先安装 Python 操作 MySQL 的驱动。在 Python 中，目前有以下 MySQL 驱动包。↵

（1）MySQL-python：也就是 MySQLdb。是对 C 语言操作 MySQL 数据库的一个简单封装。遵循了 Python DB API v2。但是只支持 Python2。↵

（2）mysqlclient：是 MySQL-python 的另外一个分支。支持 Python3 并且修复了一些 bug。是目前为止执行效率最高的驱动，但是安装的时候容易因为环境问题出错。↵

（3）pymysql：纯 Python 实现的一个驱动。因为是纯 Python 编写的，因此执行效率不如 mysqlclient。也正因为是纯 Python 写的，因此可以和 Python 代码无缝衔接。↵

（4）mysql-connector-python：MySQL 官方推出的纯 Python 连接 MySQL 的驱动，执行效率 pymysql 还慢。↵

用下列命令安装：

```
pip install pymysql
```

1.2 SQLAlchemy

SQLAlchemy是一个数据库的ORM框架，让我们操作数据库的时候不要再用SQL语句了，跟直接操作模型一样。安装命令为：

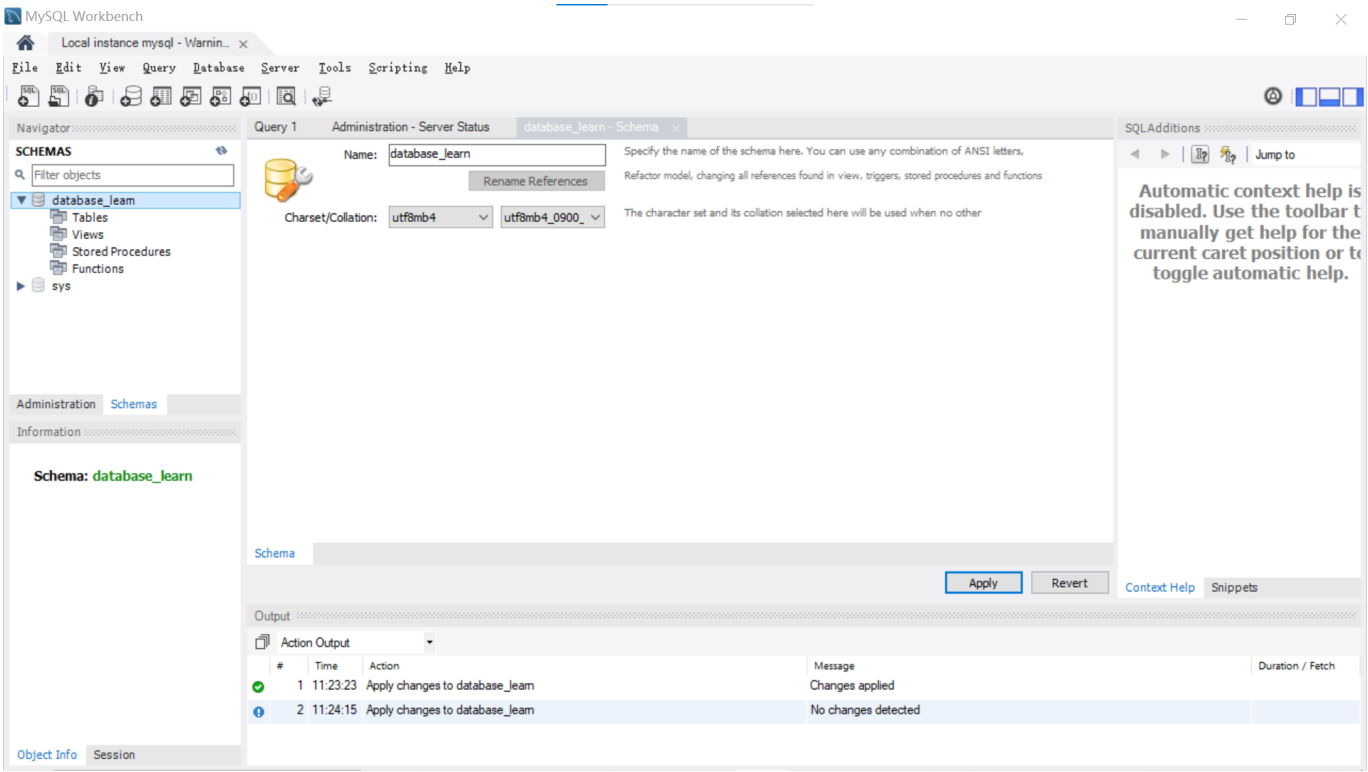
```
pip install flask-sqlalchemy
```

SQLAlchemy官方文档:https://www.sqlalchemy.org/

1.3 flask连接数据库

```python
from flask import Flask, render_template
from flask_sqlalchemy import SQLAlchemy
from sqlalchemy import  text

app = Flask(__name__)

#MySQL所在的主机名，由于是在本主机上运行，因此HOSTNAME为127.0.0.1;
#若在虚拟机或者云服务器上运行，需要把ip地址改为对应虚拟机或云服务器的ip地址
HOSTNAME = '127.0.0.1'

#MySQL监听的端口号，默认3306
PORT     = '3306'

#连接MySQL的用户名，读者自己设置
USERNAME = 'root'

#连接MySQL的密码,读者自己的密码
PASSWORD = '******'

#MySQL上创建的数据库名称
DATABASE = 'database_learn'

#我们用的数据库驱动程序为pymysql
app.config['SQLALCHEMY_DATABASE_URI'] = f'mysql+pymysql://{USERNAME}:
{PASSWORD}@{HOSTNAME}:{PORT}/{DATABASE}?charset=utf8mb4'

#在app.config中设置好连接数据库的信息
#然后使用SQLAlchemy(app)创建一个db对象
#在SQLAlchemy会自动读取app.config中连接数据库的信息
db=SQLAlchemy(app)

with app.app_context():
    with db.engine.connect() as conn:
        rs=conn.execute(text("select 1"))
        print(rs.fetchone())#(1,)

@app.route("/")
def index():
    return render_template("index.html")

if __name__ == '__main__':
    app.run(debug = True)
```

在MySQL Workbench中创建一个数据库database_learn




```
WARNING: This is a development server. Do not use it in a production deployment. Use
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with watchdog (windowsapi)
(1,)
 * Debugger is active!
```

1.4 ORM对象

ORM，即Object-Relational Mapping（对象关系映射），它的作用是在关系型数据库和业务实体对象之间作一个映射，这样，我们在具体的操作业务对象的时候，就不需要再去和复杂的SQL语句打交道，只需简单的操作对象的属性和方法。

ORM模型的简单性简化了数据库查询过程。使用ORM查询工具，用户可以访问期望数据，而不必理解数据库的底层结构。

一个ORM模型与数据库中一个表相对应，ORM模型中的每个类属性分别对应表的每个字段。ORM模型的每个实例对象对应表中每条记录，ORM技术提供了面向对象与SQL交互的桥梁，让开发者用面向对象的方式操作数据库，使用ORM模型具有以下优势：

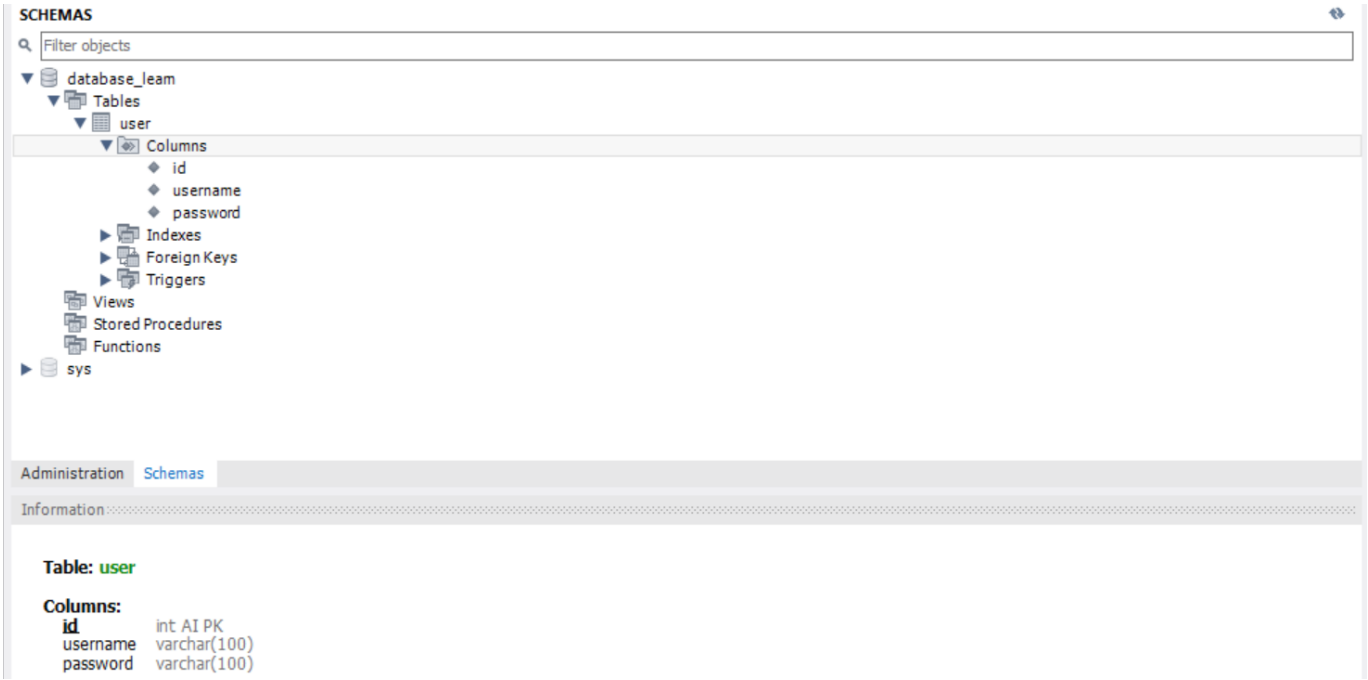
（1）开发效率高；

（2）安全性高；

（3）灵活性强；支持不同关系数据库，针对不同数据库，ORM模型代码几乎一模一样，只需要修改少量代码；

```python
from flask import Flask, render_template
from flask_sqlalchemy import SQLAlchemy
from sqlalchemy import  text

app = Flask(__name__)

#MySQL所在的主机名，由于是在本主机上运行，因此HOSTNAME为127.0.0.1;
#若在虚拟机或者云服务器上运行，需要把ip地址改为对应虚拟机或云服务器的ip地址
HOSTNAME = '127.0.0.1'

#MySQL监听的端口号，默认3306
PORT     = '3306'

#连接MySQL的用户名，读者自己设置
USERNAME = 'root'

#连接MySQL的密码
PASSWORD = '111111'

#MySQL上创建的数据库名称
DATABASE = 'database_learn'

#我们用的数据库驱动程序为pymysql
app.config['SQLALCHEMY_DATABASE_URI'] = f'mysql+pymysql://{USERNAME}:
{PASSWORD}@{HOSTNAME}:{PORT}/{DATABASE}?charset=utf8mb4'

#在app.config中设置好连接数据库的信息
#然后使用SQLAlchemy(app)创建一个db对象
#在SQLAlchemy会自动读取app.config中连接数据库的信息
db=SQLAlchemy(app)

#创建ORM对象
class User(db.Model):
    #表名
    __tablename__ = "user"
    #autoincrement表示增加一列，自动加1
    id = db.Column(db.Integer,primary_key=True,autoincrement=True)
    #db.String自动映射成varchar类型，指定最大长度
    #nullable表示该字段不能为空；也就是null=0;
    username=db.Column(db.String(100),nullable=False)
    password=db.Column(db.String(100),nullable=False)

#创建表
with app.app_context():#理解上下文关系
    db.create_all()
#username=User(username='张三',password='111111')#可以这样创建一个id
#sql: insert user(username,password) values("张三","111111")


@app.route("/")
```

```python
def index():
    return render_template("index.html")

if __name__ == '__main__':
    app.run(debug = True)
```



SQLAlchemy会自动的设置第一个Integer的主键并且没有被标记为外键的字段添加自增长的属性。

1.4.1 Column常用参数：

- `default`：默认值。
- `nullable`：是否可空。
- `primary_key`：是否为主键。
- `unique`：是否唯一。
- `autoincrement`：是否自动增长。
- `onupdate`：更新的时候执行的函数。
- `name`：该属性在数据库中的字段映射。

1.4.2 sqlalchemy常用数据类型：

- `Integer`：整形。
- `Float`：浮点类型。
- `Boolean`：传递`True/False`进去。
- `DECIMAL`：定点类型。
- `enum`：枚举类型。
- `Date`：传递`datetime.date()`进去。
- `DateTime`：传递`datetime.datetime()`进去。
- `Time`：传递`datetime.time()`进去。
- `String`：字符类型，使用时需要指定长度，区别于`Text`类型。
- `Text`：文本类型。
- `LONGTEXT`：长文本类型。

1.5 CRUD操作

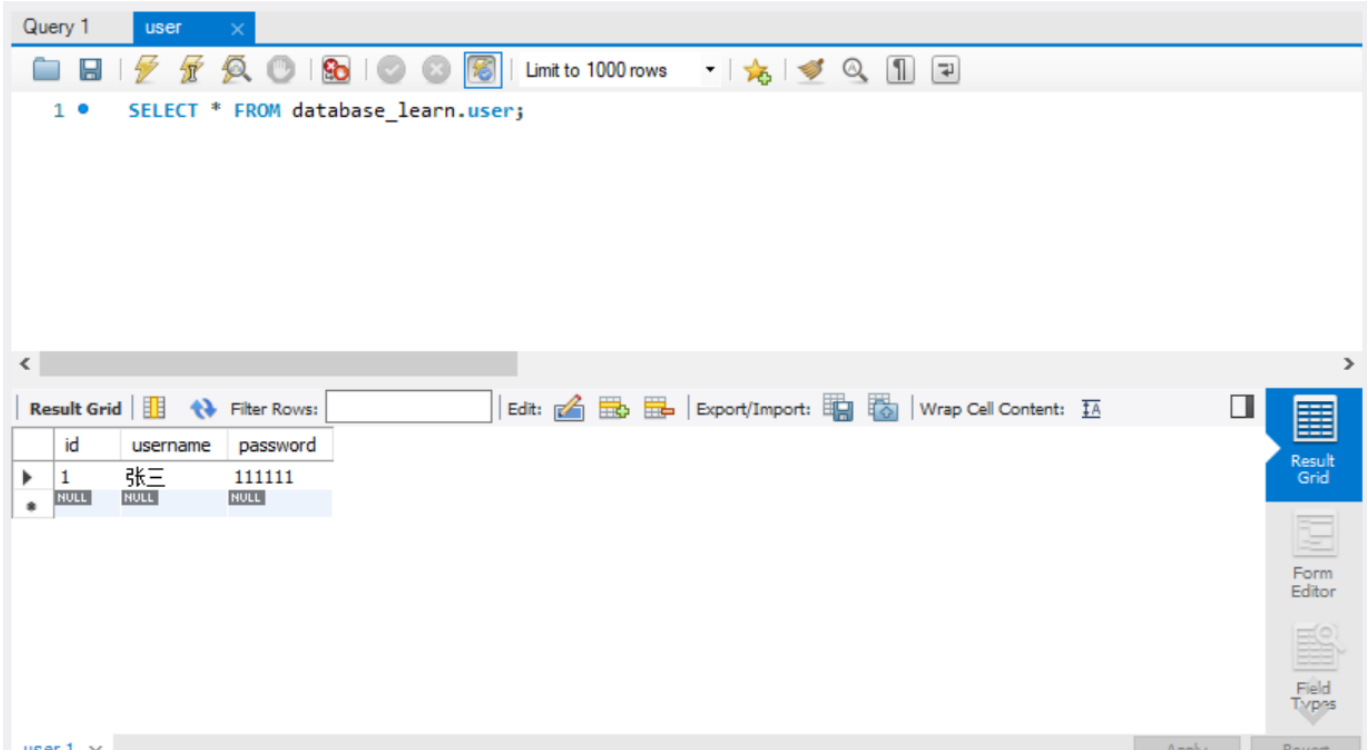
CRUD是指在做计算处理时的增加(Create)、读取查询(Retrieve)、更新(Update)和删除(Delete)几个单词的首字母简写。主要被用在描述软件系统中DataBase或者持久层的基本操作功能。

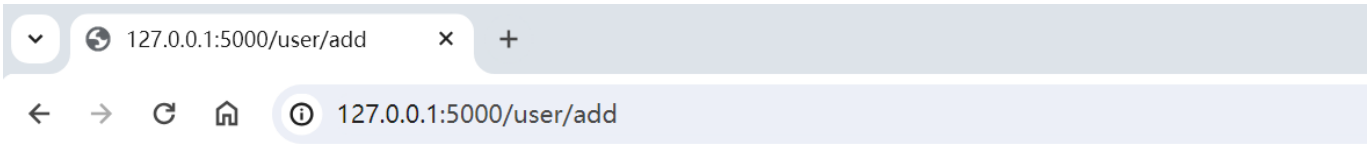
使用ORM进行CRUD(Create,Read,Update,Delete)操作，需要先把操作添加到会话中，通过db.session可以获取到会话对象，会话对象只是在内存中，如果想要把会话中的操作回滚，则可以通过db.session.rollback()实现；

1.5.1 Create操作

```
@app.route("/user/add")
def add_user():
    #1.创建ORM对象
    user=User(username='张三',password='111111')
    #2.得到ORM对象添加到db.session中
    db.session.add(user)
    #3.将db.session中的改变同步到数据库中
    db.session.commit()
    return "用户创建成功"
```

运行结果如下：





用户创建成功

1.5.2 数据查找

1.5.2.1.通过get查找

```
@app.route("/user/query")
def query_user():
    #1.get查找：根据主键查找
    user=User.query.get(1)
    print(f"{user.id}:{user.username}-{user.password}")
    return "数据查找成功"
```

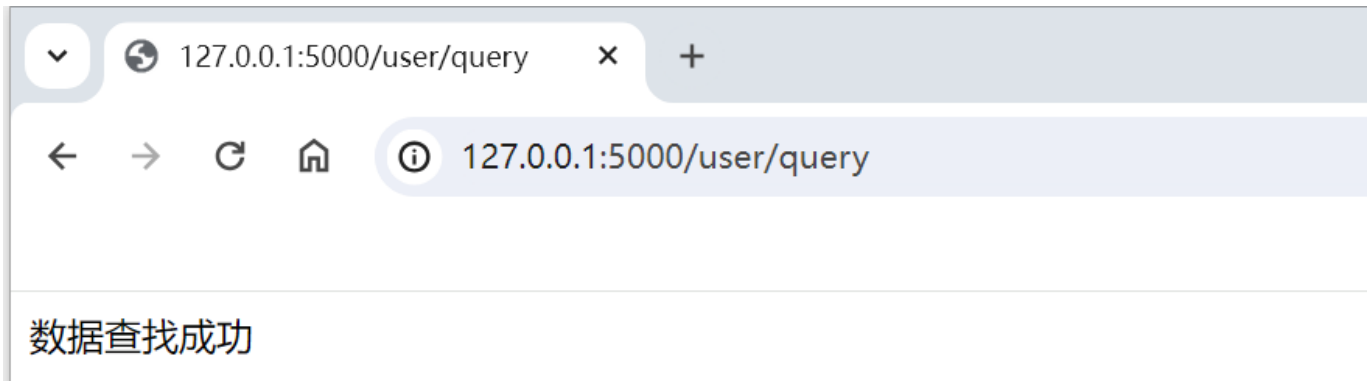
运行结果如下：

数据查找成功

```
emy 2.0 at: https://sqlalche.me/e/b8d9)
  user=User.query.get(1)
1:张三-111111
127.0.0.1 - - [02/Mar/2024 09:33:11] "GET /user/query HTTP/1.1" 200 -
```

1.5.2.2.filter_by查找

```python
@app.route("/user/query")
def query_user():
    #2.filter_by查找
    #users为Query:类似于数组,可以使用数组的切片等操作
    users=User.query.filter_by(username="张三")
    for user in users:
        print(user.username)
    return "数据查找成功"
```




```
 * Restarting with watchdog (windowsapi)
 * Debugger is active!
 * Debugger PIN: 144-203-869
张三
127.0.0.1 - - [02/Mar/2024 09:39:41] "GET /user/query HTTP/1.1" 200 -
```

1.5.2.3 其他

表 5-3 query 常用提取方法

方法名	描述
query.all()	获取查询结果集中的所有对象，是列表类型。
query.first()	获取结果集中的第一个对象。
query.one()	获取结果集中的第一个对象，如果结果集中对象数量不等于 1，则会抛出异常。
query.one_or_none()	与 one 类似，结果不为 1 的时候，不是抛出异常，而是返回 None。
query.get(pk)	根据主键获取当前 ORM 模型的第一条数据。
query.exists()	判断数据是否存在。
query.count()	获取结果集的个数。

```
@app.route("/user/query")
def query_user():
    #3.filter传递查询条件
    #users为Query:类似于数组,可以使用数组的切片等操作
    users=User.query.filter(User.username=="张三")
    for user in users:
        print(user.username)
    return "数据查找成功"
```

有条件的查询：

1）模型类.query.filter_by(字段名 =值) 等价于 select * from user where 字段名=值;

2）模型类.query.filter_by(字段名 =值).first() 等价于 select * from user where 字段名=值 limit 1;

3）模型类.query.filter(模型类.字段名.endswith('z')).all() 等价于 select * from user where 字段名 like '%z';

4）模型类.query.filter(模型类.字段名.startswith('z')).all() 等价于 select * from user where 字段名 like 'z%';

5）模型类.query.filter(模型类.字段名.contains('z')).all() 等价于 select * from user where 字段名 like '%z%';

6）模型类.query.filter(模型类.字段名.like('%z%')).all() 等价于 select * from user where 字段名 like '%z%';

7）模型类.query.filter(模型类.字段名.in_(['a','b','c'])).all() 等价于 select * from user where 字段名 in ('a','b','c');

8）模型类.query.filter(模型类.字段名.between(开始,结束)).all() 等价于 select * from user where 字段名 between 开始 and 结束;

表 3-4 query 常用过滤方法

方法名	描述
query.filter()	根据查询条件过滤。
query.filter_by()	根据关键字参数过滤。
query.slice(start,stop)	对结果进行切片操作。
query.limit(limit)	对结果数量进行限制。
query.offset(offset)	在查询的时候跳过前面 offset 条数据。
query.order_by()	根据给定字段进行排序。
query.group_by()	根据给定字段进行分组。

(1)order_by()函数

```
@app.route("/user/order")
def order_user():
    #正序排序
    users=User.query.order_by("id")
    users=User.query.order_by(User.id)

    #倒序排序
    users=User.query.order_by(db.text("-id"))
    users=User.query.order_by(User.id.desc())
    #from sqlalchemy import desc
    #users=User.query.order_by(desc("id"))
    for user in users:
        print(user.id)

    return "数据排序成功"
```

限制（获取指定数量数据） limit+offset 1.1、limit 模型类.query.order_by (模型类.字段名).limit(2).all() 等价于 select * from user where 字段名=值 order by 字段名 limit(2)；

1.2、limit+offset 跳过前2位再取值前两位 就是3、4

模型类.query.order_by (模型类.字段名).offset(2).limit(2).all()

排序

1、无条件的排序 模型类.query.order_by (模型类.字段名.desc()).all() 等价于 select * from user order by 字段名 desc；

2、有条件的排序 模型类.query.filter(模型类.字段名==值).order_by (模型类.字段名.desc()).all() 等价于 select * from user where 字段名=值 order by 字段名 desc；

也可以这样写

模型类.query.filter(模型类.字段名==值).order_by (-模型类.字段名).all() 等价于 select * from user where 字段名=值 order by 字段名 desc；

组合查询

```
from sqlalchemy import or_, and_, not_
```

1）模型类.query.filter(or_(模型类.字段名.like('z%'),模型类.字段名.contains('a'))).all() 等价于 select * from user where 字段名 like 'z%' or 字段名 like '%a%'；

2）模型类.query.filter(and_(模型类.字段名.like('z%'),模型类.字段名._ *lt* _('2021-12-12 00:00:00'))).all() 等价于 select * from user where 字段名 like 'z%' and 字段名 < '2021-12-12 00:00:00'；

扩展

> __gt__

>= __ge__(gt equal)

<= __le__(lt euqal)

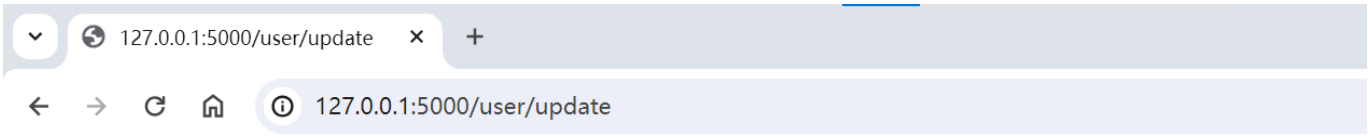
!= not_

3）模型类.query.filter(not_(模型类.字段名.contains('a'))).all() 等价于 select * from user where 字段名 not like '%a%' ；

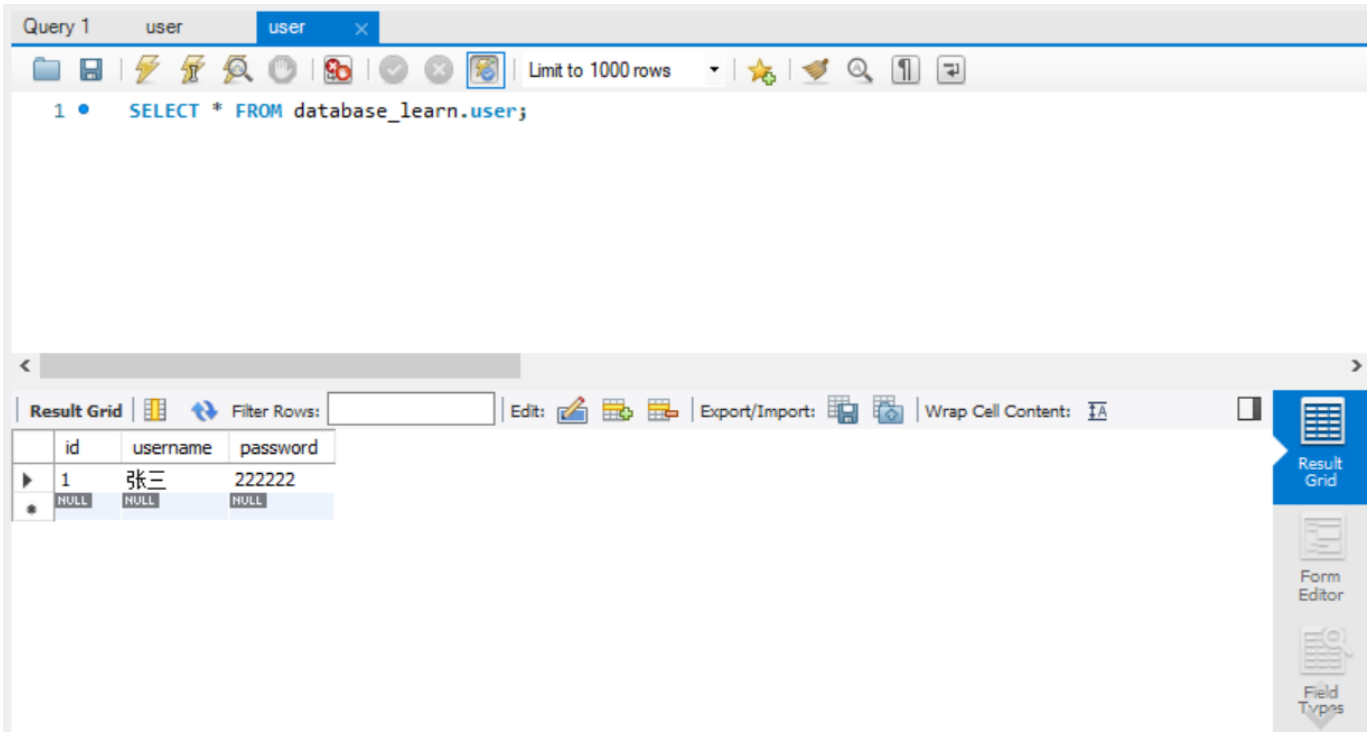
1.5.3 Update操作

```
@app.route("/user/update")
def update_user():
    #如果数组为空，[0]会报错
    #uses=User.query.filter_by(username="张三")[0]

    #如果数组为空，返回null，安全性高
    user=User.query.filter_by(username="张三").first()
    user.password="222222"
    db.session.commit()
    return "数据修改成功"
```

运行结果如下：



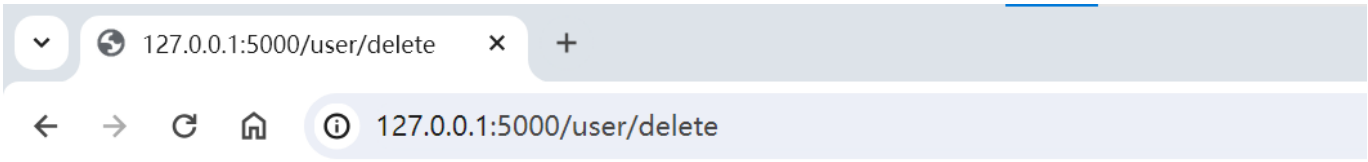
数据修改成功



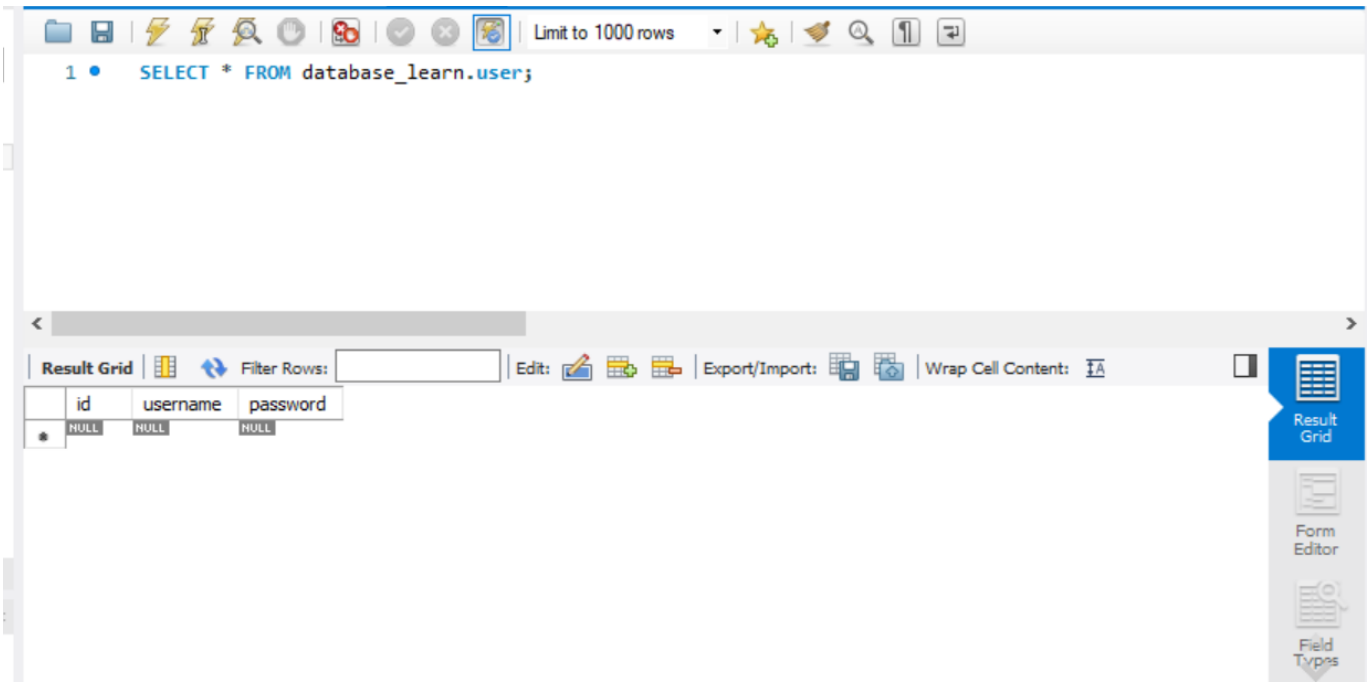
1.5.4 Delete操作

```
@app.route("/user/delete")
def delete_user():
    #1.查找
    user=User.query.get(1)
    #2.从db.session中删除
    db.session.delete(user)
    #3.将db.session中的修改，同步到数据库中
    db.session.commit()
    return "数据删除成功！"
```

运行结果如下：



数据删除成功！



1.6 表关系

关系型数据库一个强大的功能，就是多个表之间可以建立关系。

表之间的关系存在三种：一对一、一对多、多对多。而SQLAlchemy中的ORM也可以模拟这三种关系。因为一对一其实在SQLAlchemy中底层是通过一对多的方式模拟的，所以先来看下一对多的关系：

1.6.1 外键

外键是数据库层面的技术，在Flask-SQLAlchemy中支持创建ORM模型的时候就指定 外键，创建外键是通过db.ForeignKey实现的。

```python
class Article(db.Model):
    __tablename__="article"
    id = db.Column(db.Integer,primary_key=True,autoincrement=True)
    username=db.Column(db.String(200),nullable=False)
     #db.String最多只能存储255个字符
    content=db.Column(db.Text,nullable=False)

    #添加作者的外键
    author_id=db.Column(db.Integer,db.ForeignKey("user.id"))
```

外键约束有以下几项：

- RESTRICT：父表数据被删除，会阻止删除。默认就是这一项。
- NO ACTION：在MySQL中，同RESTRICT。
- CASCADE：级联删除。
- SET NULL：父表数据被删除，子表数据会设置为NULL。

1.6.2 一对多

1.6.2.1 建立关系

以上通过外键，实际上已经建立起一对多的关系；一篇文章只能引用一个作者，但是一个作者可以被多篇文章引用，但是以上只是建立了一个外键，通过Article的对象，还是无法直接获取author_id引用的那个User对象，为了达到操作ORM对象就跟操作普通python对象一样，Flask-SQLAlchemy提供了db.relationship来引用外键所指向的那个ORM模型，在以上的Article模型中，添加db.relationship

```python
from flask import Flask, render_template
from flask_sqlalchemy import SQLAlchemy
from sqlalchemy import  text

app = Flask(__name__)

#MySQL所在的主机名，由于是在本主机上运行，因此HOSTNAME为127.0.0.1;
#若在虚拟机或者云服务器上运行，需要把ip地址改为对应虚拟机或云服务器的ip地址
HOSTNAME = '127.0.0.1'

#MySQL监听的端口号，默认3306
PORT     = '3306'

#连接MySQL的用户名，读者自己设置
USERNAME = 'root'

#连接MySQL的密码
PASSWORD = '111111'

#MySQL上创建的数据库名称
DATABASE = 'database_learn'

#我们用的数据库驱动程序为pymysql
app.config['SQLALCHEMY_DATABASE_URI'] = f'mysql+pymysql://{USERNAME}:
{PASSWORD}@{HOSTNAME}:{PORT}/{DATABASE}?charset=utf8mb4'

#在app.config中设置好连接数据库的信息
#然后使用SQLAlchemy(app)创建一个db对象
#在SQLAlchemy会自动读取app.config中连接数据库的信息
db=SQLAlchemy(app)
'''
用于测试数据库是否连接成功
with app.app_context():
    with db.engine.connect() as conn:
        rs=conn.execute(text("select 1"))
        print(rs.fetchone())#(1,)'''
#创建ORM对象
class User(db.Model):
    #表名
    __tablename__ = "user"
    #autoincrement表示增加一列，自动加1
    id = db.Column(db.Integer,primary_key=True,autoincrement=True)
    #db.String自动映射成varchar类型，指定最大长度
    #nullable表示该字段不能为空；也就是null=0;
    username=db.Column(db.String(100),nullable=False)
    password=db.Column(db.String(100),nullable=False)

    #使用back_populates时，同时需要绑定两方
    #articles=db.relationship("Article",back_populates="author")#反向引用
```

```python
class Article(db.Model):
    __tablename__="article"
    id = db.Column(db.Integer,primary_key=True,autoincrement=True)
    title=db.Column(db.String(200),nullable=False)#db.String最多只能存储255个字符
    content=db.Column(db.Text,nullable=False)

    #添加作者的外键
    author_id=db.Column(db.Integer,db.ForeignKey("user.id"))

    #author=db.relationship("User",back_populates='articles')#反向引用
    #backref会自动的给User模型添加一个articles的属性，用来获取文章的列表
    author=db.relationship("User",backref='articles')




#创建表
with app.app_context():#理解上下文关系
    db.create_all()
#username=User(username='张三',password='111111')#可以这样创建一个id
#sql: insert user(username,password) values("张三","111111")


@app.route("/")
def index():
    return render_template("index.html")

@app.route("/user/add")
def add_user():
    #1.创建ORM对象
    user=User(username='张三',password='111111')
    #2.得到ORM对象添加到db.session中
    db.session.add(user)
    #3.将db.session中的改变同步到数据库中
    db.session.commit()
    return "用户创建成功"

@app.route("/user/query")
def query_user():
    #3.filter传递查询条件
    #users为Query:类似于数组,可以使用数组的切片等操作
    users=User.query.filter_by(User.username=="张三")
    for user in users:
        print(user.username)
    return "数据查找成功"

@app.route("/user/delete")
def delete_user():
    #1.查找
    user=User.query.get(1)
    #2.从db.session中删除
```

```
    db.session.delete(user)
    #3.将db.session中的修改，同步到数据库中
    db.session.commit()
    return "数据删除成功！"

@app.route("/article/add")
def add_article():
    article1=Article(title="flask",content="fxxxxxxx")
    article1.author=User.query.get(2)

    article2=Article(title="django",content="dxxxxxxx")
    article2.author=User.query.get(2)

    #添加到session中
    db.session.add_all([article1,article2])
    #3.将db.session中的改变同步到数据库中
    db.session.commit()
    return "数据创建成功"

@app.route("/article/query")
def query_article():
    user = User.query.get(2)
    for article in user.articles:
        print(article.title)
    return "文章查找成功"

if __name__ == '__main__':
    app.run(debug = True)
```

运行结果：

文章查找成功

```
e/e/b8d9)
  user = User.query.get(2)
flask
django
127.0.0.1 - - [02/Mar/2024 13:26:22] "GET /article/query HTTP/1.1" 200 -
```

2.ORM迁移

2.1Flask-Migrate插件

在实际的开发环境中，经常会发生数据库修改的行为。一般我们修改数据库不会直接手动的去修改，而是去修改ORM对应的模型，然后再把模型映射到数据库中。这时候如果有一个工具能专门做这种事情，就显得非常有用了，而flask-migrate就是做这个事情的。flask-migrate是基于Alembic进行的一个封装，并集成到Flask中，而所有的迁移操作其实都是Alembic做的，他能跟踪模型的变化，并将变化映射到数据库中。

```
pip install flask-migrate
```

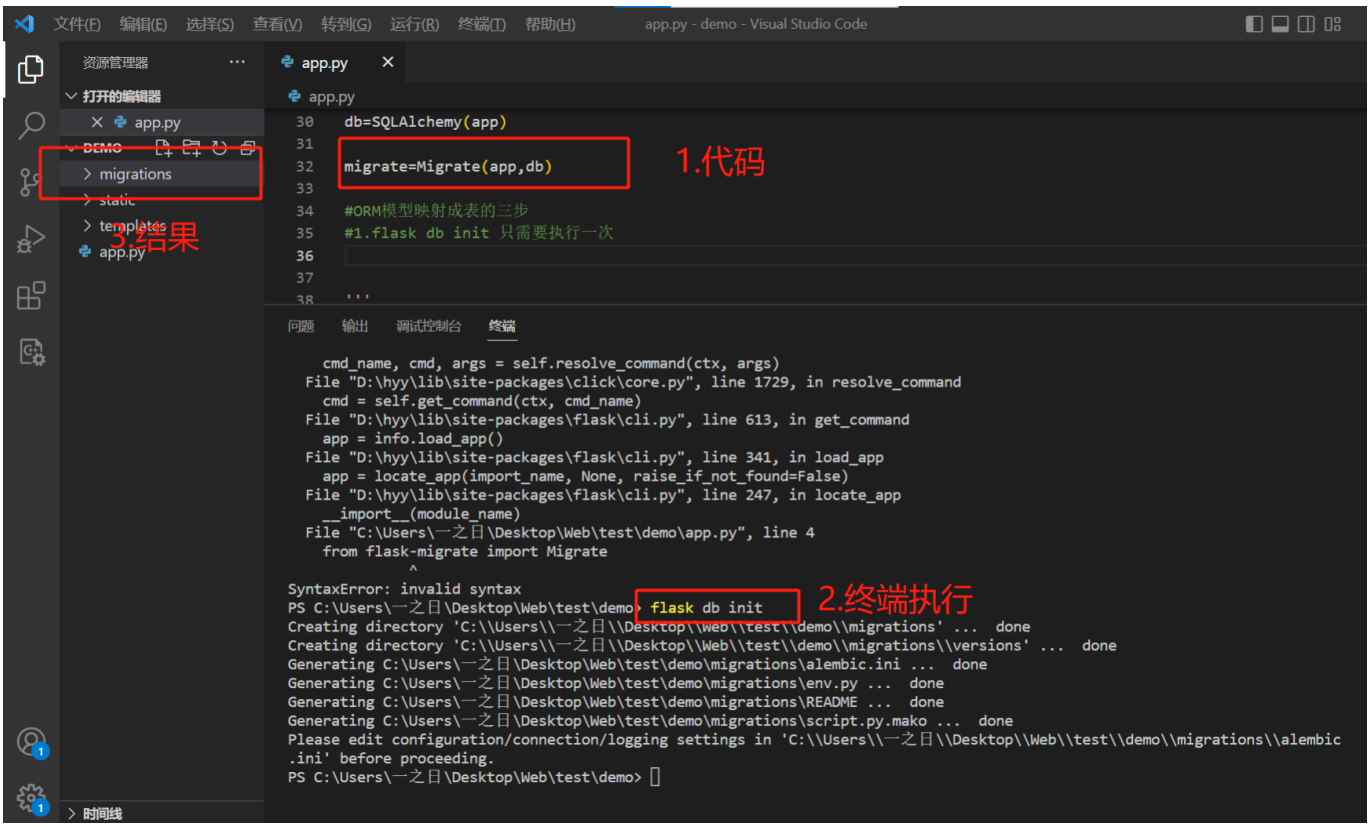
```
#创建表,方法只能识别新建的表，不能识别已有表内的改变
with app.app_context():#理解上下文关系
    db.create_all()
```

因此我们用migrate做迁移

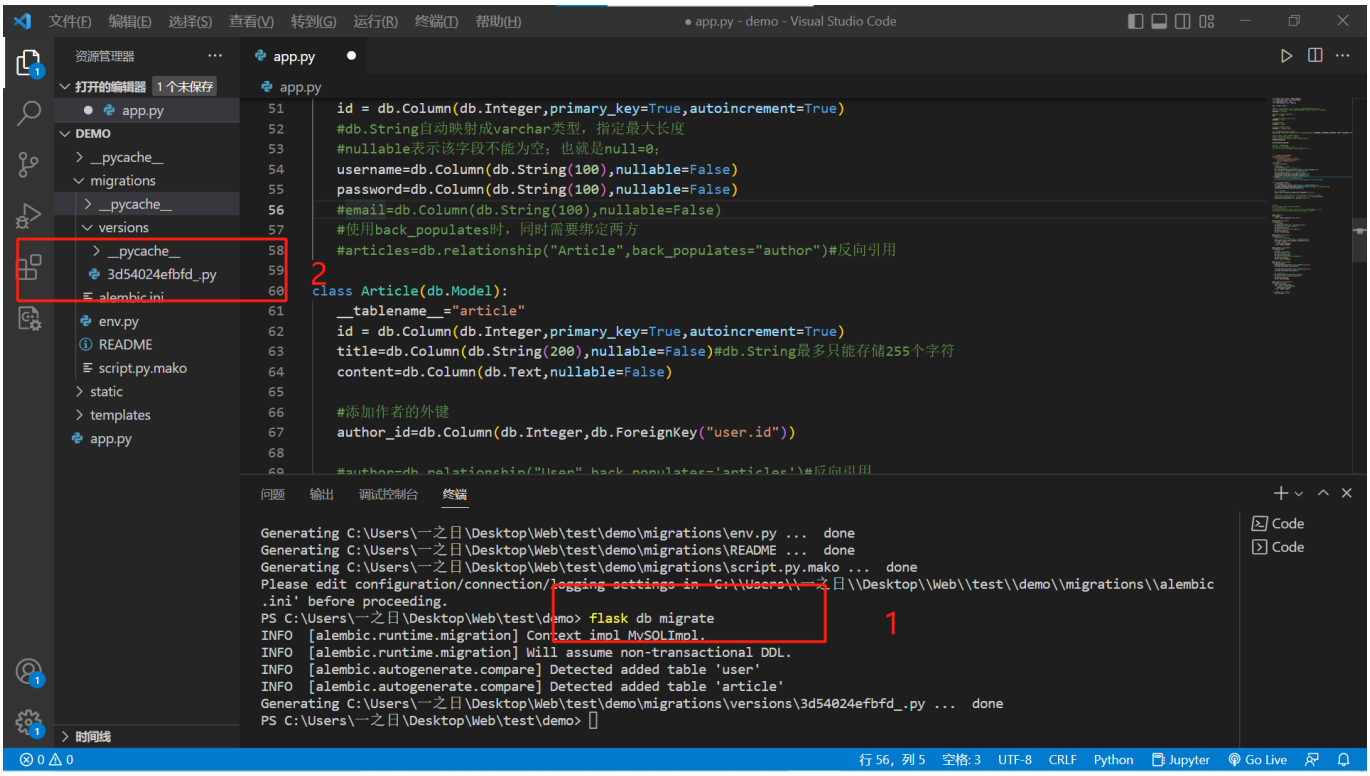
#ORM模型映射成表的三步

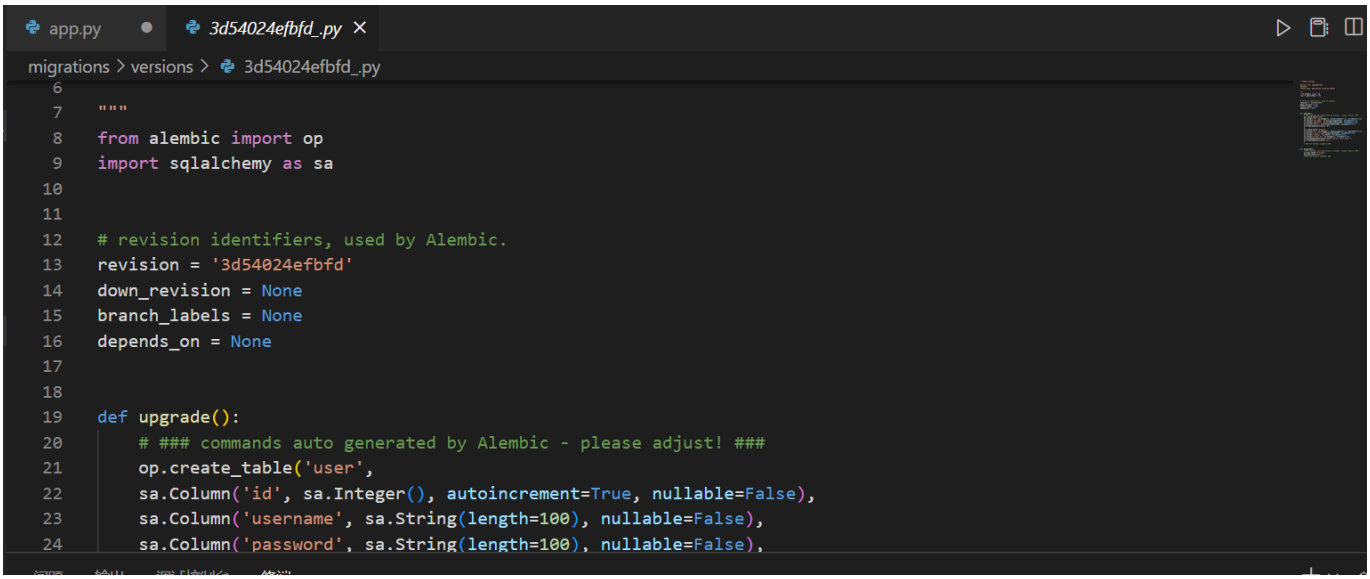
- 1.flask db init 只需要执行一次
- 2.flask db migrate:识别ORM模型的改变，生成迁移脚本
- 3.flask db upgrade:运行迁移脚本，同步到数据库中

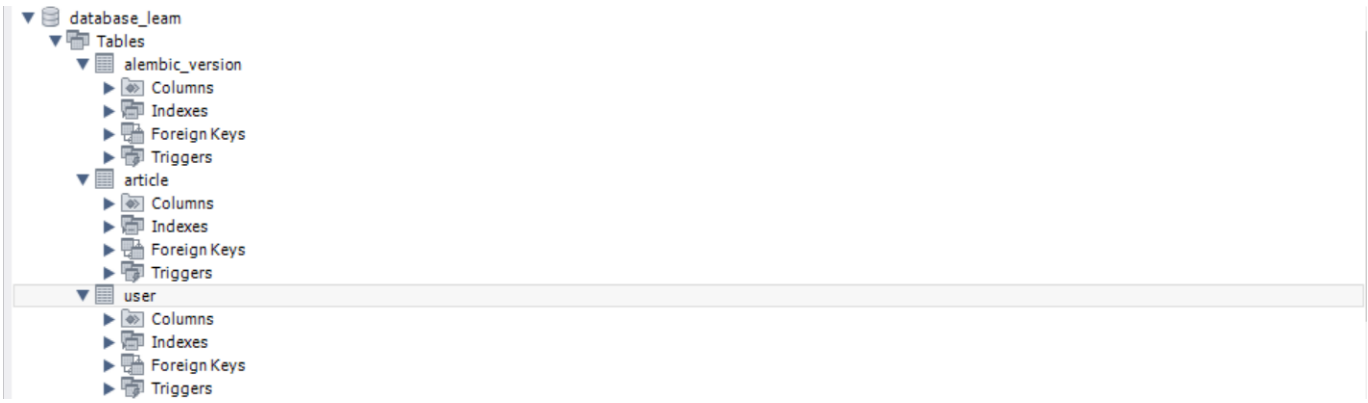
1.flask db init 只需要执行一次



2.flask db migrate:识别ORM模型的改变，生成迁移脚本







3.flask db upgrade:运行迁移脚本，同步到数据库中