Flask-SQLAlchemy 查询中的查询是通过query对象操作数据。通过过滤器进行条件限定精确查询。

1.基本查询

查询所有数据

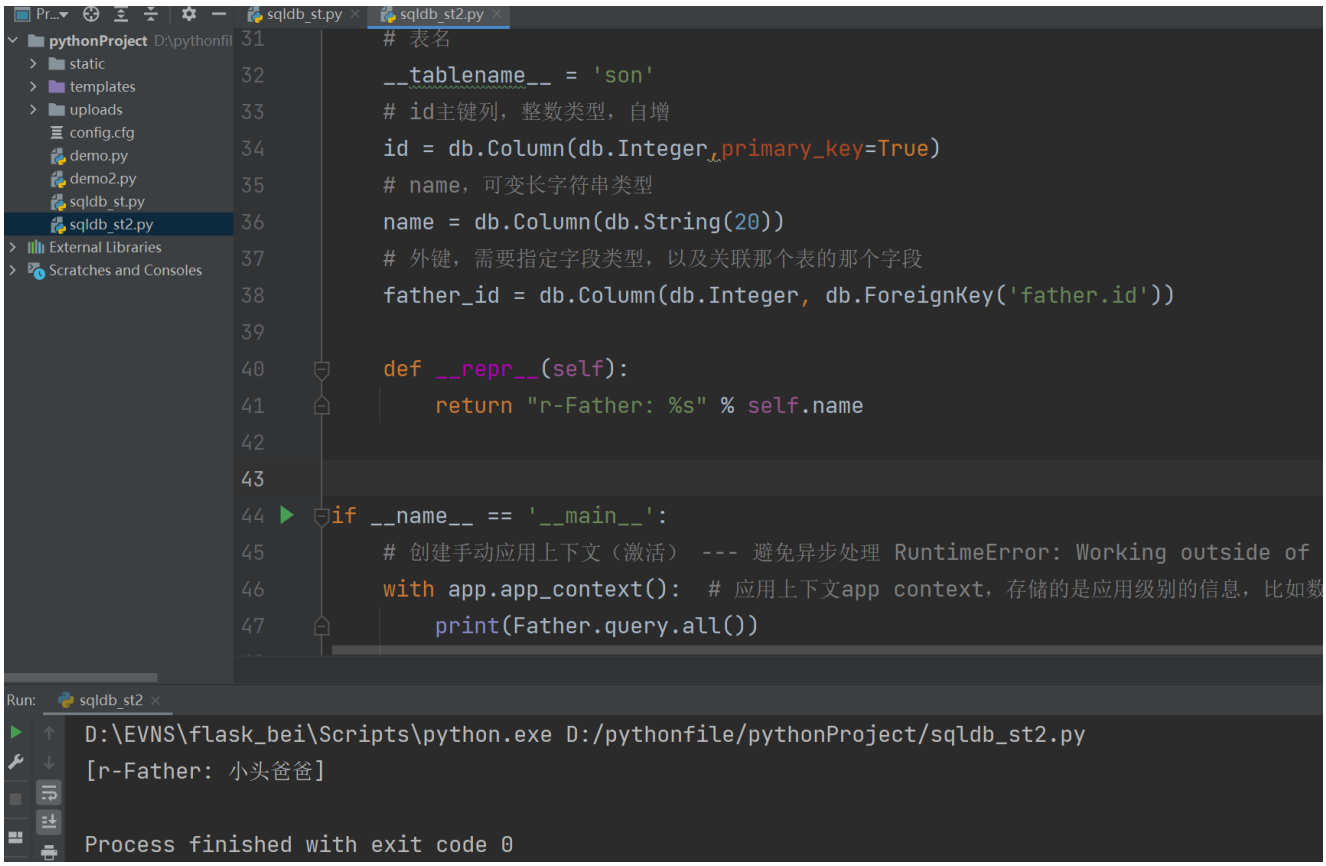
模型类.query.all()

Repr_ 方法：

查询默认返回的是id,如果要直观的看到查询出来的是什么数据，可以在模型类中添加一个 repr 方法，类似于django模型类中的 str 方法。

```
    def __repr__(self):
        return "r-Father: %s" % self.name
```

添加 repr 方法后再查询，直接返回对象显示的是repr方法的返回值。



2.常用查询方法

方法	作用
all()	返回数据库中所有数据，列表形式返回
first()	返回查询的第一个结果，如果未查到，返回None
first_or_404()	返回查询的第一个结果，如果未查到，返回404
get()	返回指定主键对应的行，如不存在，返回None

方法	作用
get_or_404()	返回指定主键对应的行，如不存在，返回404
count()	返回查询结果的数量
paginate()	返回一个Paginate对象，它包含指定范围内的结果

查询过滤器

过滤器	说明
filter()	把过滤器添加到原查询上，返回一个新查询
filter_by()	把等值过滤器添加到原查询上，返回一个新查询
limit	使用指定的值限定原查询返回的结果
offset()	偏移原查询返回的结果，返回一个新查询
order_by()	根据指定条件对原查询结果进行排序，返回一个新查询
group_by()	根据指定条件对原查询结果进行分组，返回一个新查询

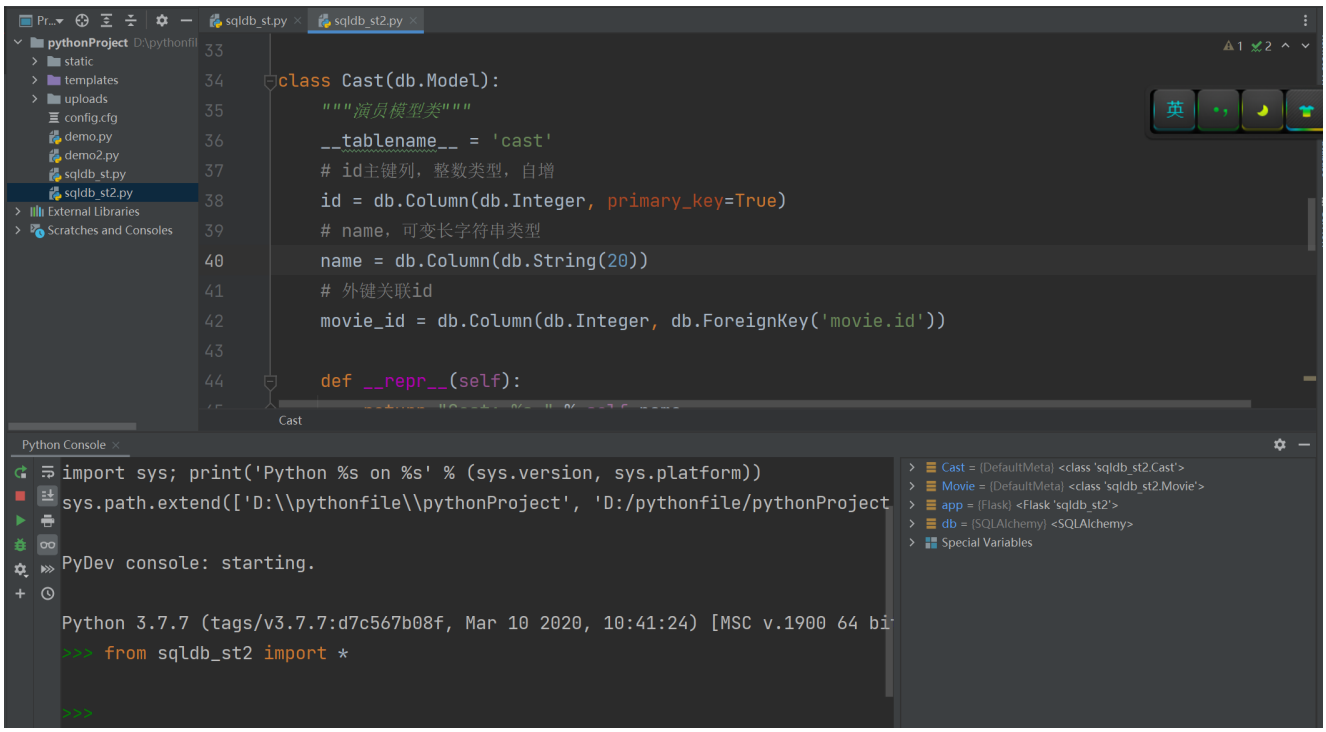
模型数据构建

```
# -*- encoding: utf-8 -*-

# -*- encoding: utf-8 -*-
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
```

```
app = Flask(__name__)  # type:Flask

# 设置连接的数据库 注意采用pymysql连接 mysql+pymysql
app.config['SQLALCHEMY_DATABASE_URI']
= 'mysql+pymysql://root:qwe123@127.0.0.1:3306/py'

# 追踪对象的修改并且发送信号 --- 目前部分版本
SQLALCHEMY_TRACK_MODIFICATIONS存在逻辑问题，因此避免需主动配置一
下
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = True

# 创建数据库对象
db = SQLAlchemy(app)


class Movie(db.Model):
    """创建一个电影模型类"""
    # 表名
    __tablename__ = 'movie'
    # id主键列，整数类型，自增
    id = db.Column(db.Integer, primary_key=True)
    # name，可变长字符串类型
    name = db.Column(db.String(20))
    # 关系字段，不是数据库中真实存在的字段，而是为了方便查询添加的
属性
    cast = db.relationship('Cast', backref='Movie')

    def __repr__(self):
        return "Movie: %s " % self.name


class Cast(db.Model):
    """演员模型类"""
    __tablename__ = 'cast'
    # id主键列，整数类型，自增
    id = db.Column(db.Integer, primary_key=True)
    # name，可变长字符串类型
    name = db.Column(db.String(20))
```

```
    # 外键关联id
    movie_id = db.Column(db.Integer,
db.ForeignKey('movie.id'))

    def __repr__(self):
        return "Cast: %s " % self.name


if __name__ == '__main__':
    # 创建手动应用上下文（激活） --- 避免异步处理 RuntimeError:
Working outside of application context
    with app.app_context():  # 应用上下文app context，存储的是应
用级别的信息，比如数据库连接信息
        # 删除所有表，注意这条是危险命令，会将数据库中的表物理删
除。在实际生产环境下勿用。
        db.drop_all()
        # 创建所有表
        db.create_all()

        # 电影数据追加
        m1 = Movie(name='大话西游')
        m2 = Movie(name='功夫')
        db.session.add_all([m1, m2])
        db.session.commit()

        # 明星关联数据追加
        cast_list = []
        for i in ['周星驰', '朱茵', '吴孟达', '莫文蔚']:
            c = Cast(name=i, movie_id=m1.id)
            cast_list.append(c)

        for i in ['周星驰', '梁小龙', '元华']:
            c = Cast(name=i, movie_id=m2.id)
            cast_list.append(c)

        db.session.add_all(cast_list)
        db.session.commit()
```

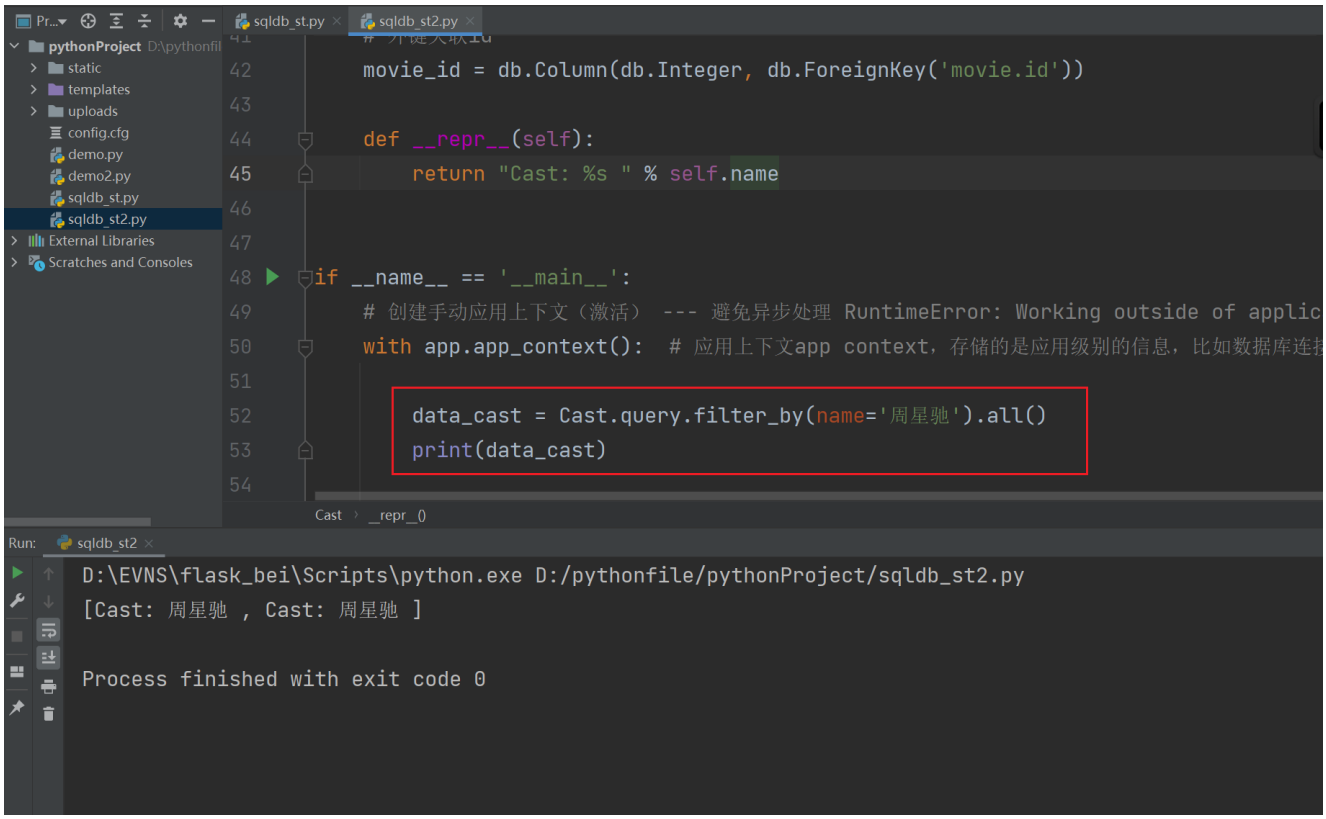
进入python shell ,导入模块



当然这样操作不太方便，对于flask而言，可直接操作执行

filter_by 精确匹配查询,条件只能是本表的字段

```
if __name__ == '__main__':
    # 创建手动应用上下文（激活） --- 避免异步处理 RuntimeError: Working outside of application context
    with app.app_context():  # 应用上下文app context，存储的是应用级别的信息，比如数据库连接信息

        data_cast = Cast.query.filter_by(name='周星驰').all()
        print(data_cast)
```



first()返回查询到的第一个对象

```
if __name__ == '__main__':
    with app.app_context():
        data_cast = Cast.query.first()
        print(data_cast)  # Cast: 周星驰
```

all()返回查询到的所有对象

```
if __name__ == '__main__':
    with app.app_context():
        data_cast = Cast.query.all()
        print(data_cast)  # [Cast: 周星驰 , Cast: 朱茵 , Cast: 吴孟达 , Cast: 莫文蔚 , Cast: 周星驰 , Cast: 梁小龙 , Cast: 元华 ]
```

filter 模糊查询

```
>>> Cast.query.filter(Cast.name.startswith('周')).all()
[Cast: 周星驰 , Cast: 周星驰 ]
>>> Cast.query.filter(Cast.name.endswith('龙')).all()
[Cast: 梁小龙 ]
>>> Cast.query.filter(Cast.name=='周星驰').all()
[Cast: 周星驰 , Cast: 周星驰 ]
```

like 正则 模糊匹配，跟mysql中的like一样。

```
>>> Cast.query.filter(Cast.name.like('周%%')).all()
[Cast: 周星驰 , Cast: 周星驰 ]

LIKE 子句中使用 ’ % ‘ 字符来表示0个或多个字符，实现模糊匹配。如果
没有使用 ‘ % ’ , LIKE 子句与’ = ’ 的效果是一样的。
```

3.逻辑运算符

```
and_ 逻辑与
or_ 逻辑非
not_ 取反

查询中使用逻辑运算需要先导入
from sqlalchemy import and_,or_,not_
```

```
from sqlalchemy import and_,or_,not_

#查询名字为'周星驰'并且movie_id为2的演员数据。
In [11]:Cast.query.filter(and_(Cast.name=='周星
驰',Cast.movie_id==2)).all()
Out[11]:[Cast: 周星驰 ]
```

```
#查询名字为'周星驰'或者id为2的演员数据。
In [12]:Cast.query.filter(or_(Cast.name=='周星
驰',Cast.movie_id==2)).all()
Out[12]:[Cast: 周星驰 , Cast: 周星驰 , Cast: 梁小龙 , Cast: 元华
]

#查询名字不为'周星驰'的演员
In [13]: Cast.query.filter(not_(Cast.name=='周星驰')).all()
Out[13]: [Cast: 朱茵 , Cast: 吴孟达 , Cast: 莫文蔚 , Cast: 梁小
龙 , Cast: 元华 ]
```

4.关联查询

关联查询使用到 `relationship` 字段。这个字段不是真实存在于数据库中的，而是为了方便查询定义的。

查询出一个演员可以快速将他对应的电影查询出来，或者反过来，知道一部电影，可以直接将电影的演员全部查询出来。

查询名字为'朱茵'的演员，并且查出她出演的电影：

```
# 查询出演员'朱茵'
In [10]: actor = Cast.query.filter(Cast.name=='朱茵').all()

In [11]: actor
Out[11]: [Cast: 朱茵 ]

# 直接通过 relationship 中定义的 backref 属性可以直接查询出演员对
应的电影。
In [12]: actor[0].Movie
Out[12]: Movie: 大话西游
```

```
In [13]: movie = actor[0].Movie
In [14]: movie
Out[14]: Movie: 大话西游

# 通过电影直接查询出所有演员
In [15]: movie.cast
Out[15]: [Cast: 周星驰 , Cast: 朱茵 , Cast: 吴孟达 , Cast: 莫文蔚 ]
```

5.删除数据

先查询后删除

```
# 删除演员表中的'吴孟达'
In [2]: Cast.query.filter(Cast.name=='吴孟达').first()
Out[2]: Cast: 吴孟达

# 将数据查询出来
In [3]: actor = Cast.query.filter(Cast.name=='吴孟达').first()

# 删除对象
In [4]: db.session.delete(actor)

# 提交到数据库
In [5]: db.session.commit()

# 查询所有演员，'吴孟达' 已经被删除
In [6]: Cast.query.all()
Out[6]: [Cast: 周星驰 , Cast: 朱茵 , Cast: 莫文蔚 , Cast: 周星驰 , Cast: 梁小龙 , Cast: 元华 ]
```

6.更新数据

```
# 将'梁小龙'的名字改成'火云邪神'
# 第一个种方式

In [6]: Cast.query.all()
Out[6]: [Cast: 周星驰 , Cast: 朱茵 , Cast: 莫文蔚 , Cast: 周星驰
, Cast: 梁小龙 , Cast: 元华 ]

# 先将'梁小龙'查询出来
In [8]: actor = Cast.query.filter(Cast.name=='梁小龙').first()

# 将'梁小龙'的名字改成'火云邪神'
In [9]: actor.name = '火云邪神'
In [10]: db.session.add(actor)
In [11]: db.session.commit()

In [12]: Cast.query.all()
Out[12]: [Cast: 周星驰 , Cast: 朱茵 , Cast: 莫文蔚 , Cast: 周星
驰 , Cast: 火云邪神 , Cast: 元华 ]
```

```
# 第二种方式 update 方式
# 将'朱茵' 修改成'紫霞仙子'

In [19]:Cast.query.filter(Cast.name=='朱
茵').update({'name':'紫霞仙子'})
In [20]: Cast.query.all()
Out[20]: [Cast: 周星驰 , Cast: 紫霞仙子 , Cast: 莫文蔚 , Cast:
周星驰 , Cast: 火云邪神 , Cast: 元华
]
```