3 模型

一、模型

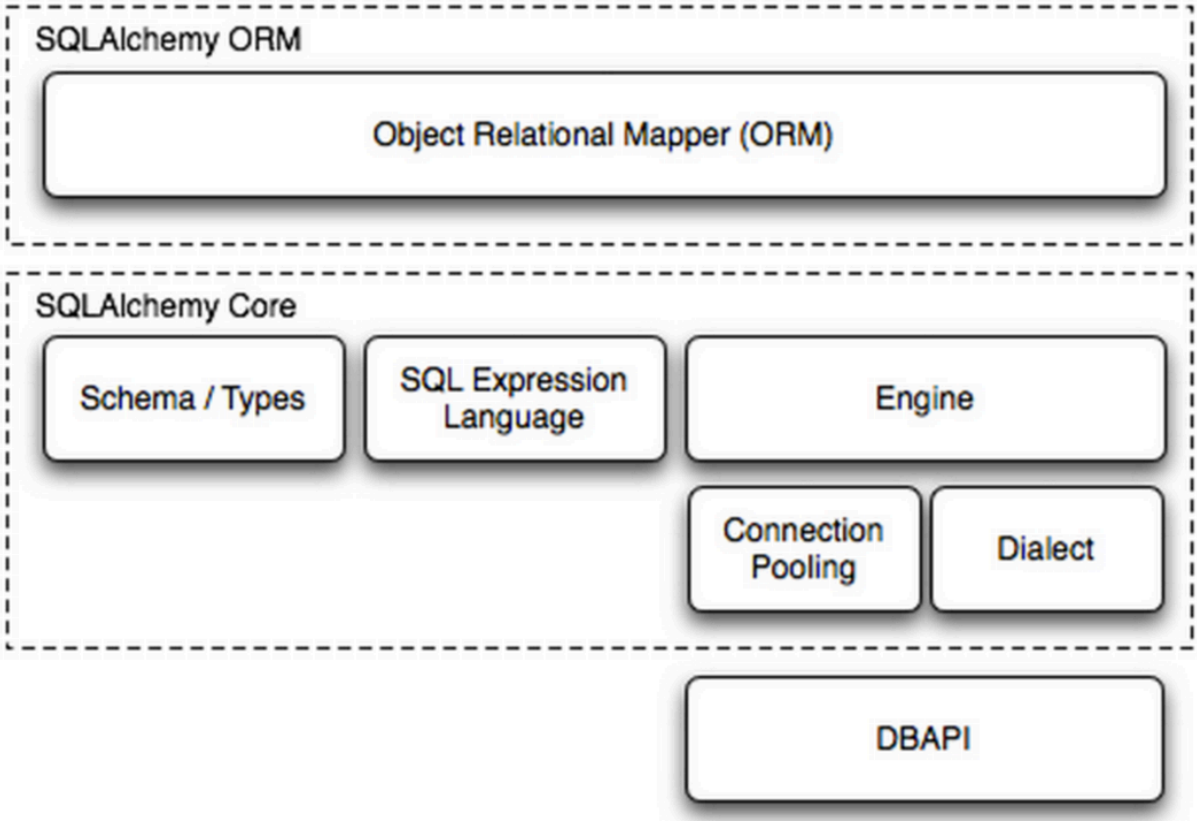
对象关系映射（Object Relational Mapping，简称ORM）模式是一种为了解决面向对象与关系数据库存在的互不匹配的现象的技术。简单的说，ORM是通过使用描述对象和数据库之间映射的元数据，自动生成sql语句，将程序中的对象自动保存到关系数据库中。优点：

- 隐藏了数据库访问的细节，简化了sql的使用，提高了开发效率
- 解耦业务逻辑层（view）和数据处理层（model)，简化了开发流程，提高了系统的可移植性
- 提高了安全性

缺点：

- 执行效率低
- 对复杂sql无能为力
- 增加了学习成本

SQLAlchemy是一个基于Python实现的ORM框架。该框架建立在 DB API之上，使用关系对象映射进行数据库操作，简言之便是：将类和对象转换成SQL，然后使用数据API执行SQL并获取执行结果。



组成部分：

- Engine，框架的引擎
- Connection Pooling ，数据库连接池
- Dialect，选择连接数据库的DB API种类
- Schema/Types，架构和类型
- SQL Exprression Language，SQL表达式语言

  SQLAlchemy本身无法操作数据库，其必须以来pymsql等第三方插件，Dialect用于和数据API进行交流，根据配置文件的不同调用不同的数据库API，从而实现对数据库的操作

1.1 安装

```
pip install pymysql
pip install flask-sqlalchemy
```

1.2 配置链接数据的url地址

- **mysql**

```
#'数据库类型+数据库驱动名称://用户名:口令@机器地址:端口号/数据库名'
app.config['SQLALCHEMY_DATABASE_URI'] =
'mysql+pymysql://root:123456@127.0.0.1:3306/python1809flask'
```

- **sqlite**

```
app.config['SQLALCHEMY_DATABASE_URI'] =
'sqlite:///'+os.path.join(os.getcwd(),'flask.sqlite')
```

- 开启自动提交（否则为事物的提交）

```
app.config['SQLALCHEMY_COMMIT_ON_TEARDOWN'] = True
```

二、设计模型

(1) 字段类型

类型	python中类型	说明
Integer	int	整型
SmallInteger	int	小整型
BigInteger	int	长整型
Float	float	浮点型
String	string	字符串
Text	string	长文本
Boolean	bool	整型
Date	datetime.date	日期
Time	datetime.time	时间
DateTime	datetime.datetime	日期和时间

(2) 约束条件

约束条件	说明
primary_key	主键
unique	唯一索引
index	常规索引
nullable	是否可以为null 默认True
default	默认值
name	表中的字段名，默认是模型字段名

注意：

1. flask默认为空
2. 设置外键不同 flask需要设置反向引用 多对多手动创建中间表
3. flask默认开启事物 需要提交或者回滚(可以设置自动提交)
4. 使用flask-sqlalchemy时要求每个模型都有一个主键，默认名字为id
5. 模型类名与数据表中的名字
    - 默认：将大驼峰格式的模型类名，转换为小写加下划线格式，如：`UserModel => user_model`
    - 指定：`__tablename__`，使用此类属性指定表名

三、创建模型

(1) 创建用户模型

实例

```
from flask import Flask
from flask_script import Manager
from flask_sqlalchemy import SQLAlchemy #导入ORM
from datetime import datetime
app = Flask(__name__)
#链接数据库地址
app.config['SQLALCHEMY_DATABASE_URI'] =
'mysql+pymysql://root:123456@127.0.0.1:3306/python1809flask'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
```

```
db = SQLAlchemy(app) #实例化模型
manager = Manager(app)


#创建用户模型
class User(db.Model):
    __tablename__ = 'user' #起表名 默认是类名
    id = db.Column(db.Integer,primary_key=True) #主键
    username = db.Column(db.String(20),index=True)
    password_hash = db.Column(db.String(128))
    age = db.Column(db.Integer,default=20)
    sex = db.Column(db.Boolean,default=True)
    role = db.Column(db.Boolean,default=False) #角色（前台普通用户 还是后台管理员）
    register = db.Column(db.DateTime,default=datetime.utcnow) #注册时间
    confirm = db.Column(db.Boolean,default=False)
```

(2) 创建表

```
#创建表
@app.route('/create_all/')
def create_all():
    db.create_all()
    return 'create_all'
```

(3) 删除表

```
#删除表
@app.route('/drop_all/')
def drop_all():
    db.drop_all()
    return 'drop_all'
```

四、实现增删改

(1) add 添加一条数据

```
#添加一条数据
@app.route('/add_one/')
def add_one():
    try:
        u = User(username='张
三',password_hash='xkjbaskdadhajkdhadksjh')
        db.session.add(u) #添加一条数据
        db.session.commit() #提交
    except:
        db.session.rollback() #回滚
    return 'add_one'
```

(2) add_all 添加多条数据

```
#添加多条数据
@app.route('/add_many/')
def add_many():
    u1 = User()
    u1.username = '李四'
    u2 = User()
    u2.username = '王无'
    db.session.add_all([u1,u2])
    db.session.commit()
    return 'add_many'
```

(3) 修改数据

```
#修改
@app.route('/update/')
def update():
    u = User.query.get(1)
    u.username = '兆六'
    db.session.add(u)
    db.session.commit()
    return '修改
```

(4) delete 删除数据

```
#删除
@app.route('/delete/')
def delete():
    u = User.query.get(1)
    db.session.delete(u)
    db.session.commit()
    return '删除'
```

(5) 自定义一个base类 实现增删改

```
class DbBase:
    #添加一条数据
    def save(self):
        try:
            db.session.add(self)  # 添加一条数据
            db.session.commit()  # 提交
        except:
            db.session.rollback()
    #添加多条数据
    @staticmethod
    def save_all(*args):
        try:
            db.session.add_all(args)
            db.session.commit()
        except:
            db.session.rollback()

    # 删除一条数据
    def delete(self):
        try:
            db.session.delete(self)  # 添加一条数据
            db.session.commit()  # 提交
        except:
            db.session.rollback()

#创建用户模型
class User(db.Model,DbBase):
    ...
```

使用

```python
#添加一条数据
@test.route('/add_one/')
def add_one():
    u = User(username='李四', password_hash='xkjbaskdadhajkdhadksjh')
    u.save()
    return 'add_one'

#添加多条数据
@test.route('/add_many/')
def add_many():
    u1 = User()
    u1.username = '兆六'
    u2 = User()
    u2.username = '王无'
    User.save_all(u1,u2)
    return 'add_many'

#修改
@test.route('/update/')
def update():
    u = User.query.get(1)
    u.username = '张三三'
    u.save()
    return '修改'

#删除
@test.route('/delete/')
def delete():
    u = User.query.get(1)
    u.delete()
    return '删除'
```

五、数据库查询

1.常见查询

方法	说明
get	根据主键进行查询，查到返回对象，没查到返回None
get_or_404	功能同上，查不到时，直接报404错
all	查询所有数据，返回一个***列表***(元素全是对象)
first	返回第一条数据，没有时返回None
first_or_404	功能同上，查不到时报404错
limit	限制结果集数量，返回查询对象，可以继续进行链式查询操作
offset	设置偏移量，返回查询对象，可以继续进行链式查询操作
order_by	结果集排序，可以指定多个字段，asc升序(默认)，desc降序
count	统计总数

(1) all() 返回一个列表 包含所有对象

```
用法：类名.query.all()
      db.session.query(模型名).all()
u = User.query.all()[0:2]
u = User.query.all()
u = db.session.query(User.username,User.password).all()  #指定字段
u = db.session.query(User).all()
```

注意：不能进行链式调用 返回的不是查询集 而是列表（你只能再次使用列表的属性个方法了）

(2)first() 取出第一条数据

```
data = User.query.first()
print(data,type(data))
```

注意：取出的值为第一条数据是模型对象， 结果不可迭代

(3) get() 只能获取主键对应的数据 查询不到返回None

```
User.query.get(id值)
```

(4)字段列表

```
user = User.query.with_entities(User.username,User.sex).all()
user = db.session.query(User.username,User.sex).all()
```

(5)别名

```
 # 表别名
 ub = aliased(UserHobby,name='ub')
 data = db.session.query(ub).all()
 #字段别名
 data = User.query.with_entities(User.username.label('name'),
User.sex).all()
```

(6)distinct

```
    from sqlalchemy import distinct
    data = db.session.query(User.sex).distinct().all()
    data = db.session.query(distinct(User.sex)).all()
    data = User.query.with_entities(distinct(User.sex)).all()
```

2.排序

- 默认升序/asc()
- -降序/desc()

```
data = User.query.order_by(User.age).limit(1) #取出年龄最小
data = User.query.order_by(-User.age).limit(1) #取出年龄最大
data = User.query.filter().order_by(-User.username,User.sex).all()
data =
User.query.filter().order_by(User.username.desc(),User.sex.asc()).all
()
```

3.聚合函数

- 说明：`max、min、sum、avg、count`
- 示例：

```
from sqlalchemy import func

# 求最大值
max_age = db.session.query(func.max(User.age)).scalar()
num = db.session.query(User).count()
```

4.分组

- group_by()
- having()
- 导入：from sqlalchemy import func

  `func`方法主要用来做统计，映射到sql语句中具体的统计方法，如：

  - `func.count(..)`
  - `func.sum(..)`

```
num = func.count('*').label('c')
data =
db.session.query(User.sex,num).group_by(User.sex).having(num>3).a
ll()
data =
db.session.query(User.sex,func.max(User.id).label('d')).group_by(
User.sex).all()
data =
User.query.with_entities(User.sex,num).group_by(User.sex).having(
num>3).all()
```

5. 限制

- offset(num) 偏移num条数据
- limit(num) 取num条数据

```
u = User.query.offset(4)
u = User.query.limit(4)
u = User.query.offset(1).limit(5)
```

6.条件查询

可以是哟filter和filter_by,ilter_by只用于单字段查询

- 关系运算

```
>,  __gt__
>=, __ge__
<,  __lt__
<=, __le__
==, __eq__
!=, __ne__
如:
    users = User.query.filter(User.age > 20).all()
    # 与上面等价
    users = User.query.filter(User.age.__gt__(20)).all()
```

- contains() 包含

```
u = User.query.filter(User.username.contains('5'))
u = User.query.filter(User.username.contains('5'),User.age>70)
```

- like 模糊查询

```
u = User.query.filter(User.username.like('%5'))
u = User.query.filter(User.username.like('5%'))
u = User.query.filter(User.username.like('%5%'))
```

- startswith endswith以...开头 以...结尾

```
u = User.query.filter(User.username.startswith('李'))
u = User.query.filter(User.username.endswith('六'))
```

- in 和 not in

```
u = User.query.filter(User.age.in_([10,20,30,67,68]))
u = User.query.filter(~User.age.in_([10,20,30,67,68]))
u = User.query.filter(User.age.notin_([10,20,30,67,68]))
u = User.query.filter(~User.age.notin_([10,20,30,67,68]))
```

- 判空 is null

```
u = User.query.filter(User.username==None)
u = User.query.filter(User.username.is_(None))
u = User.query.filter(User.username!=None)
u = User.query.filter(User.username.isnot(None))
```

- filter_by() 只支持单条件查询

```
User.query.filter_by()
User.query.filter_by(id=1)
User.query.filter_by(sex=True)
```

- 逻辑运算

```
from sqlalchemy import and_, or_,not_

# 默认就是逻辑与
# users = User.query.filter(User.id > 2, User.age > 20).all()
# 与上式等价
# users = User.query.filter(and_(User.id > 2, User.age >
20)).all()
# 逻辑或
users = User.query.filter(or_(User.id > 2, User.age > 20)).all()
# 逻辑非
u = User.query.filter(not_(User.sex==True,))
```

7.联合查询

```
data = db.session.query(xuankebiao).join(Student).all()
#1 内链接
data = db.session.query(Category,Posts).filter(Category.cid ==
Posts.cid,Posts.pid<6).all()
data = db.session.query(Category).join(Posts,Category.cid ==
Posts.cid).filter(Posts.pid<6)
data =
db.session.query(Category,Posts,User).filter(Category.cid==Posts.cid,
Posts.uid==User.uid)

# 2.外链接
data = Category.query.outerjoin(Posts,Category.cid==Posts.cid).all()
```

```
data =
db.session.query(Category.categoryname,Category.pid).outerjoin(Posts,
Category.cid==Posts.cid).all()

# 并
data1 = Category.query.filter(Category.cid <= 3)
data2 = Category.query.filter(Category.cid > 1)
data = data1.union(data2).all()
```

8.原生sql查询

```
db.session.execute("select * from student").fetchall()
db.session.execute('insert into users(name) values(:value)',params=
{"value":'wupeiqi'})
 db.session.commit()
```

六、数据库迁移

安装

1. pip install flask-script
2. pip install flask-migrate

实例

```
from flask import Flask
from flask_sqlalchemy import SQLalchemy
from flask_script import Manager
from flask_migrate import Migrate,MigrateCommand

app = Flask(__name__)
db = SQLalchemy(app) #实例化ORM模型
migrate = Migrate(app,db) #实例化迁移对象
manager = Manger(app)
manager.add_command('db',MigrateCommand) #添加迁移命令 别名为db
```

迁移步骤（命令）

(1) 迁移初始化（创建迁移目录，只做一次）

python manage.py db init

(2) 创建迁移文件

python manage.py db migrate

(3) 执行迁移

python manage.py db upgrade

- 注意，模型必须被引用才能导入，否则不会发生任何变化

七. 反向生成

可以根据数据库表结构，反向生成模型。第三方库flask-sqlcodegen可以完成此功能。

1. 安装

```
pip install flask-sqlacodegen
```

2. 生成模型

```
flask-sqlacodegen mysql://root:@127.0.0.1/数据库名 --outfile 模型文件名
--flask
```