数据库操作

ORM

ORM 全拼`Object-Relation Mapping`，中文意为 **对象-关系映射**。主要实现模型对象到关系数据库数据的映射

优点 :

- 只需要面向对象编程, 不需要面向数据库编写代码.
  - 对数据库的操作都转化成对类属性和方法的操作.
  - 不用编写各种数据库的`sql语句`.
- 实现了数据模型与数据库的解耦, 屏蔽了不同数据库操作上的差异.
  - 不再需要关注当前项目使用的是哪种数据库。
  - 通过简单的配置就可以轻松更换数据库, 而不需要修改代码.

缺点 :

- 相比较直接使用SQL语句操作数据库,有性能损失.
- 根据对象的操作转换成SQL语句,根据查询的结果转化成对象, 在映射过程中有性能损失.

Flask-SQLAlchemy

flask默认提供模型操作，但是并没有提供ORM，所以一般开发的时候我们会采用flask-SQLAlchemy模块来实现ORM操作。

SQLAlchemy是一个关系型数据库框架，它提供了高层的 ORM 和底层的原生数据库的操作。flask-sqlalchemy 是一个简化了 SQLAlchemy 操作的flask扩展。

SQLAlchemy: https://www.sqlalchemy.org/

安装 flask-sqlalchemy

```
pip install flask-sqlalchemy
```

如果连接的是 mysql 数据库，需要安装 mysqldb **驱动**

```
pip install flask-mysqldb
```

数据库连接设置

- 在 Flask-SQLAlchemy 中，数据库使用URL指定，而且程序使用的数据库必须保存到Flask配置对象的 **SQLALCHEMY_DATABASE_URI** 键中

```
app.config['SQLALCHEMY_DATABASE_URI'] = 'mysql://root:mysql@127.0.0.1:3306/test'
```

- 其他设置：

```
# 动态追踪修改设置，如未设置只会提示警告
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = True
#查询时会显示原始SQL语句
app.config['SQLALCHEMY_ECHO'] = True
```

- 配置完成需要去 MySQL 中创建项目所使用的数据库

```
$ mysql -uroot -p123456
$ create database flaskdemo charset=utf8;
```

常用的SQLAlchemy字段类型

类型名	python中类型	说明
Integer	int	普通整数，一般是32位
SmallInteger	int	取值范围小的整数，一般是16位
BigInteger	int或long	不限制精度的整数
Float	float	浮点数
Numeric	decimal.Decimal	普通整数，一般是32位
String	str	变长字符串
Text	str	变长字符串，对较长或不限长度的字符串做了优化
Unicode	unicode	变长Unicode字符串
UnicodeText	unicode	变长Unicode字符串，对较长或不限长度的字符串做了优化
Boolean	bool	布尔值
Date	datetime.date	时间
Time	datetime.datetime	日期和时间
LargeBinary	str	二进制文件

常用的SQLAlchemy列选项

选项名	说明
primary_key	如果为True，代表表的主键
unique	如果为True，代表这列不允许出现重复的值
index	如果为True，为这列创建索引，提高查询效率
nullable	如果为True，允许有空值，如果为False，不允许有空值
default	为这列定义默认值

常用的SQLAlchemy关系选项

选项名	说明
backref	在关系的另一模型中添加反向引用,用于设置外键名称,在1查多的
primary join	明确指定两个模型之间使用的联结条件
uselist	如果为False，不使用列表，而使用标量值
order_by	指定关系中记录的排序方式
secondary	指定多对多关系中关系表的名字
secondary join	在SQLAlchemy中无法自行决定时，指定多对多关系中的二级联结条件

数据库基本操作

- 在Flask-SQLAlchemy中，插入、修改、删除操作，均由数据库会话管理。
  - 会话用 db.session 表示。在准备把数据写入数据库前，要先将数据添加到会话中然后调用 commit() 方法提交会话。
- 在 Flask-SQLAlchemy 中，查询操作是通过 query 对象操作数据。
  - 最基本的查询是返回表中所有数据，可以通过过滤器进行更精确的数据库查询。

在视图函数中定义模型类

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy


app = Flask(__name__)

#设置连接数据库的URL
app.config['SQLALCHEMY_DATABASE_URI'] = 'mysql://root:mysql@127.0.0.1:3306/test'
```

```
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = True
#查询时会显示原始SQL语句
app.config['SQLALCHEMY_ECHO'] = True
db = SQLAlchemy(app)

class Role(db.Model):
    # 定义表名
    __tablename__ = 'roles'
    # 定义列对象
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(64), unique=True)
    us = db.relationship('User', backref='role')

    # repr()方法类似于django的__str__，用于打印模型对象时显示的字符串信息
    def __repr__(self):
        return 'Role:%s'% self.name

class User(db.Model):
    __tablename__ = 'users'
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(64), unique=True)
    email = db.Column(db.String(64),unique=True)
    password = db.Column(db.String(64))
    role_id = db.Column(db.Integer, db.ForeignKey('roles.id'))

    def __repr__(self):
        return 'User:%s'%self.name
if __name__ == '__main__':
    app.run(debug=True)
```

模型之前的关联

一对多

```
class Role(db.Model):
    ...
    #关键代码
    us = db.relationship('User', backref='role', lazy='dynamic')
    ...

class User(db.Model):
    ...
    role_id = db.Column(db.Integer, db.ForeignKey('roles.id'))
```

- 其中realtionship描述了Role和User的关系。在此文中，第一个参数为对应参照的类"User"
- 第二个参数backref为类User申明新属性的方法
- 第三个参数lazy决定了什么时候SQLALchemy从数据库中加载数据
    - 如果设置为子查询方式(subquery)，则会在加载完Role对象后，就立即加载与其关联的对象，这样会让总查询数量减少，但如果返回的条目数量很多，就会比较慢

        - 设置为 subquery 的话，role.users 返回所有数据列表
    - 另外,也可以设置为动态方式(dynamic)，这样关联对象会在被使用的时候再进行加载，并且在返回前进行过滤，如果返回的对象数很多，或者未来会变得很多，那最好采用这种方式
        - 设置为 dynamic 的话，role.users 返回查询对象，并没有做真正的查询，可以利用查询对象做其他逻辑，比如：先排序再返回结果

多对多

```
registrations = db.Table('registrations',
    db.Column('student_id', db.Integer, db.ForeignKey('students.id')),
    db.Column('course_id', db.Integer, db.ForeignKey('courses.id'))
)

class Course(db.Model):
    ...

class Student(db.Model):
    ...
    courses = db.relationship('Course',secondary=registrations,
                                    backref='students',
                                    lazy='dynamic')
```

常用的SQLAlchemy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

常用的SQLAlchemy查询结果的方法

方法	说明
all()	以列表形式返回查询的所有结果
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
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

创建表：

```
db.create_all()  # 注意，create_all()方法执行的时候，需要放在模型的后面
```

删除表

```
db.drop_all()
```

插入一条数据

```
ro1 = Role(name='admin')
db.session.add(ro1)
db.session.commit()
#再次插入一条数据
ro2 = Role(name='user')
db.session.add(ro2)
db.session.commit()
```

一次插入多条数据

```
us1 = User(name='wang',email='wang@163.com',password='123456',role_id=ro1.id)
us2 = User(name='zhang',email='zhang@189.com',password='201512',role_id=ro2.id)
us3 = User(name='chen',email='chen@126.com',password='987654',role_id=ro2.id)
us4 = User(name='zhou',email='zhou@163.com',password='456789',role_id=ro1.id)
us5 = User(name='tang',email='tang@163.com',password='158104',role_id=ro2.id)
us6 = User(name='wu',email='wu@gmail.com',password='5623514',role_id=ro2.id)
us7 = User(name='qian',email='qian@gmail.com',password='1543567',role_id=ro1.id)
us8 = User(name='liu',email='liu@163.com',password='867322',role_id=ro1.id)
us9 = User(name='li',email='li@163.com',password='4526342',role_id=ro2.id)
us10 = User(name='sun',email='sun@163.com',password='235523',role_id=ro2.id)
db.session.add_all([us1,us2,us3,us4,us5,us6,us7,us8,us9,us10])
db.session.commit()
```

```
查询所有用户数据
查询有多少个用户
查询第1个用户
查询id为4的用户[3种方式]
查询名字结尾字符为g的所有数据[开始/包含]
查询名字不等于wang的所有数据[2种方式]
查询名字和邮箱都以 li 开头的所有数据[2种方式]
查询password是 `123456` 或者 `email` 以 `163.com` 结尾的所有数据
查询id为 [1, 3, 5, 7, 9] 的用户列表
查询name为liu的角色数据
查询所有用户数据，并以邮箱排序
每页3个，查询第2页的数据
```

filter_by精确查询

返回名字等于wang的所有人

```
User.query.filter_by(name='wang').all()
```

first()返回查询到的第一个对象

```
User.query.first()
```

all()返回查询到的所有对象

```
User.query.all()
```

filter模糊查询，返回名字结尾字符为g的所有数据。

```
User.query.filter(User.name.endswith('g')).all()
```

get():参数为主键，如果主键不存在没有返回内容

```
User.query.get()
```

逻辑非，返回名字不等于wang的所有数据

```
User.query.filter(User.name!='wang').all()
```

not_ 相当于取反

```
from sqlalchemy import not_
User.query.filter(not_(User.name=='chen')).all()
```

逻辑与，需要导入and，返回and()条件满足的所有数据

```
from sqlalchemy import and_
User.query.filter(and_(User.name!='wang',User.email.endswith('163.com'))).all()
```

逻辑或，需要导入or_

```
from sqlalchemy import or_
User.query.filter(or_(User.name!='wang',User.email.endswith('163.com'))).all()
```

查询数据后删除

```
user = User.query.first()
db.session.delete(user)
db.session.commit()
User.query.all()
```

更新数据

```
user = User.query.first()
user.name = 'dong'
db.session.commit()
User.query.first()
```

关联查询

> 角色和用户的关系是一对多的关系，一个角色可以有多个用户，一个用户只能属于一个角色。

- 查询角色的所有用户

```
#查询roles表id为1的角色
ro1 = Role.query.get(1)
#查询该角色的所有用户
ro1.us.all()
```

- 查询用户所属角色

```
#查询users表id为3的用户
us1 = User.query.get(3)
#查询用户属于什么角色
us1.role
```

数据库迁移

- 在开发过程中，需要修改数据库模型，而且还要在修改之后更新数据库。最直接的方式就是删除旧表，但这样会丢失数据。
- 更好的解决办法是使用数据库迁移框架，它可以追踪数据库模式的变化，然后把变动应用到数据库中。
- 在Flask中可以使用Flask-Migrate扩展，来实现数据迁移。并且集成到Flask-Script中，所有操作通过命令就能完成。
- 为了导出数据库迁移命令，Flask-Migrate提供了一个MigrateCommand类，可以附加到flask-script的manager对象上。

首先要在虚拟环境中安装Flask-Migrate。

```
pip install flask-migrate
```

代码文件内容：

```
#coding=utf-8
from flask import Flask

from flask_sqlalchemy import SQLAlchemy
from flask_migrate import Migrate,MigrateCommand
from flask_script import Shell,Manager

app = Flask(__name__)
manager = Manager(app)

app.config['SQLALCHEMY_DATABASE_URI'] = 'mysql://root:mysql@127.0.0.1:3306/Flask_test'
app.config['SQLALCHEMY_COMMIT_ON_TEARDOWN'] = True
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = True
db = SQLAlchemy(app)

#第一个参数是Flask的实例，第二个参数是Sqlalchemy数据库实例
migrate = Migrate(app,db)
```

```
#manager是Flask-Script的实例，这条语句在flask-Script中添加一个db命令
manager.add_command('db',MigrateCommand)

#定义模型Role
class Role(db.Model):
    # 定义表名
    __tablename__ = 'roles'
    # 定义列对象
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(64), unique=True)
    user = db.relationship('User', backref='role')

    #repr()方法显示一个可读字符串，
    def __repr__(self):
        return 'Role:'.format(self.name)

#定义用户
class User(db.Model):
    __talbe__ = 'users'
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(64), unique=True, index=True)
    #设置外键
    role_id = db.Column(db.Integer, db.ForeignKey('roles.id'))

    def __repr__(self):
        return 'User:'.format(self.username)


if __name__ == '__main__':
    manager.run()
```

创建迁移仓库

```
#这个命令会创建migrations文件夹，所有迁移文件都放在里面。
python manage.py db init
```

创建迁移脚本

- 自动创建迁移脚本有两个函数
  - upgrade()：函数把迁移中的改动应用到数据库中。
  - downgrade()：函数则将改动删除。
- 自动创建的迁移脚本会根据模型定义和数据库当前状态的差异，生成upgrade()和downgrade()函数的内容。
- 对比不一定完全正确，有可能会遗漏一些细节，需要进行检查

```
python manage.py db migrate -m 'initial migration'
```

更新数据库

```
python manage.py db upgrade
```

返回以前的版本

可以根据history命令找到版本号,然后传给downgrade命令:

```
python manage.py db history

输出格式: <base> ->  版本号 (head), initial migration
```

回滚到指定版本

```
python manage.py db downgrade # 默认返回上一个版本
python manage.py db downgrade 版本号    # 返回到指定版本号对应的版本
```

数据迁移的步骤:

```
1. 初始化数据迁移的目录
python manage.py db init

2. 数据库的数据迁移版本初始化
python manage.py db migrate -m 'initial migration'

3. 升级版本[创建表]
python manage.py db upgrade

4. 降级版本[删除表]
python manage.py db downgrade
```

模块推荐

文档: https://faker.readthedocs.io/en/master/locales/zh_CN.html

github: https://github.com/joke2k/faker

flask-session

允许设置session到指定存储的空间中, 文档:

安装命令: https://pythonhosted.org/Flask-Session/

```
pip install Flask-Session
```

使用session之前,必须配置一下配置项:

```
SECRET_KEY = "luffycity" # session秘钥
```

redis基本配置

```
app.config['SESSION_TYPE'] = 'redis'  # session类型为redis
app.config['SESSION_PERMANENT'] = False  # 如果设置为True，则关闭浏览器session就失效。
app.config['SESSION_USE_SIGNER'] = False  # 是否对发送到浏览器上session的cookie值进行加密
app.config['SESSION_KEY_PREFIX'] = 'session:'  # 保存到session中的值的前缀
app.config['SESSION_REDIS'] = redis.Redis(host='127.0.0.1', port='6379',
password='123123')  # 用于连接redis的配置

Session(app)
```

SQLAlchemy基本配置

```
db = SQLAlchemy(app)

app.config['SESSION_TYPE'] = 'sqlalchemy'  # session类型为sqlalchemy
app.config['SESSION_SQLALCHEMY'] = db # SQLAlchemy对象
app.config['SESSION_SQLALCHEMY_TABLE'] = 'session' # session要保存的表名称
app.config['SESSION_PERMANENT'] = True  # 如果设置为True，则关闭浏览器session就失效。
app.config['SESSION_USE_SIGNER'] = False  # 是否对发送到浏览器上session的cookie值进行加密
app.config['SESSION_KEY_PREFIX'] = 'session:'  # 保存到session中的值的前缀

Session(app)
```

蓝图 Blueprint

模块化

随着flask程序越来越复杂,我们需要对程序进行模块化的处理,之前学习过python的模块化管理,于是针对一个简单的flask程序进行模块化处理

简单来说，Blueprint 是一个存储操作方法的容器，这些操作在这个Blueprint 被注册到一个应用之后就可以被调用，Flask 可以通过Blueprint来组织URL以及处理请求。

Flask使用Blueprint让应用实现模块化，在Flask中，Blueprint具有如下属性：

- 一个应用可以具有多个Blueprint
- 可以将一个Blueprint注册到任何一个未使用的URL下比如 “/”、“/sample”或者子域名
- 在一个应用中，一个模块可以注册多次
- Blueprint可以单独具有自己的模板、静态文件或者其它的通用操作方法，它并不是必须要实现应用的视图和函数的
- 在一个应用初始化时，就应该要注册需要使用的Blueprint

但是一个Blueprint并不是一个完整的应用，它不能独立于应用运行，而必须要注册到某一个应用中。

Blueprint对象用起来和一个应用/Flask对象差不多，最大的区别在于一个 蓝图对象没有办法独立运行，必须将它注册到一个应用对象上才能生效

使用蓝图可以分为四个步骤

1. 创建一个蓝图目录,例如**users**,并在`__init__.py`文件中创建蓝图对象

```
users=Blueprint('users',__name__)
```

2. 在这个蓝图目录下, 创建views.py文件,保存当前蓝图使用的视图函数

```
@admin.route('/')
def home():
    return 'user.home'
```

3. 在**users/init.py**中引入views.py中所有的视图函数

```
from flask import Blueprint
# 等同于原来在 manage.py里面的 app = Flask()
users=Blueprint('users',__name__)

from .views import *
```

4. 在主应用manage.py文件中的app对象上注册这个**users**蓝图对象

```
from users import users
app.register_blueprint(users,url_prefix='/users')
```

当这个应用启动后,通过/users/可以访问到蓝图中定义的视图函数

运行机制

- 蓝图是保存了一组将来可以在应用对象上执行的操作，注册路由就是一种操作
- 当在应用对象上调用 route 装饰器注册路由时,这个操作将修改对象的url_map路由表

- 然而，蓝图对象根本没有路由表，当我们在蓝图对象上调用route装饰器注册路由时,它只是在内部的一个延迟操作记录列表defered_functions中添加了一个项
- 当执行应用对象的 register_blueprint() 方法时，应用对象将从蓝图对象的 defered_functions 列表中取出每一项，并以自身作为参数执行该匿名函数，即调用应用对象的 add_url_rule() 方法，这将真正的修改应用对象的路由表

蓝图的url前缀

- 当我们在应用对象上注册一个蓝图时，可以指定一个url_prefix关键字参数（这个参数默认是/）
- 在应用最终的路由表 url_map中，在蓝图上注册的路由URL自动被加上了这个前缀，这个可以保证在多个蓝图中使用相同的URL规则而不会最终引起冲突，只要在注册蓝图时将不同的蓝图挂接到不同的自路径即可
- url_for

```
url_for('admin.index') # /admin/
```

注册静态路由

和应用对象不同，蓝图对象创建时不会默认注册静态目录的路由。需要我们在 创建时指定 static_folder 参数。

下面的示例将蓝图所在目录下的static_admin目录设置为静态目录

```
admin = Blueprint("admin",__name__,static_folder='static_admin')
app.register_blueprint(admin,url_prefix='/admin')
```

现在就可以使用/admin/static_admin/ 访问static_admin目录下的静态文件了 定制静态目录URL规则： 可以在创建蓝图对象时使用 static_url_path 来改变静态目录的路由。下面的示例将为 static_admin 文件夹的路由设置为 /lib

```
admin = Blueprint("admin",__name__,static_folder='static_admin',static_url_path='/lib')
app.register_blueprint(admin,url_prefix='/admin')
```

设置模版目录

蓝图对象默认的模板目录为系统的模版目录，可以在创建蓝图对象时使用 template_folder 关键字参数设置模板目录

```
admin = Blueprint('admin',__name__,template_folder='my_templates')
```

> 注:如果在 templates 中存在和 my_templates 同名文件,则系统会优先使用 templates 中的文件