数据库操作

Flask数据库
D：使用扩展包flask-sqlalchemy来操作数据库（增删改查）
E：通过 Python 对象来操作数据库，在舍弃一些性能开销的同时，换来的是开发效率的较大提升
U：**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
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	指定多对多中记录的排序方式
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

①安装扩展包及导包
安装flask-sqlalchemy：pip install flask-sqlalchemy
如果连接的是mysql数据，需安装mysqldb：pip install flask-sqlalchemy
导包：from flask_sqlalchemy import SQLAlchemy
②配置相关数据库的设置
#数据库信息设置
app.config['SQLALCHEMY_DATABASE_URI'] = 'mysql://root:mysql@127.0.0.1:3306/数据库名'
动态追踪修改设置，如未设置只会提示警告，极大影响mysql性能
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
③创建连接数据库的对象
db = SQLAlchemy(app)
④定义模型类，继承db.Model
定义数据库的表名： tablename
设置字段： 字段=db.Column(db.字段类型，字段选项)
例如:id = db.Column(db.Integer, primary_key=True)
name = db.Column(db.String(64), unique=True)
设置关联属性： books = db.relationship('Book', backref='author', lazy='dynamic')
给该模型类添加一个属性，第一个参数为多类类名，通过这个属性可以查询一对多所有对象
第二个参数backref ='该类类名小写'，是反向给多类申明一个新属性
第三个参数指定是lazy属性，即何时加载数据，dynamic指的是在访问属性的时候，并没有在内存中加载数据，而是返回一个query对象, 需要执行相应方法才可以获取对象，比如.all()
⑤无迁移式的建表和删表
db.drop_all() 删除该数据库所有的表　　db.create_all() 在该数据库下创建所有模型类映射的表

增删改查
增：①创建对象： b =Book（name='图书'）
②把数据添加到用户会话：db.session.add(b)
如果多个对象，使用db.session.add_all([b1,b2,b3])
③提交用户会话到数据库： db.session.commit()
删：第一种方式：
①查出该对象： b=Book.query.get(1)
②从用户会话删除该对对象： db.session.delete(b)
③提交用户会话： db.session.commit()
第二种方式：
①查出对象直接删除： Book.query.get(1).delete()
②提交用户会话： db.session.commit()
改：①查出该对象： b=Book.query.get(1)
②修改对象属性： b.name='小说'
③提交用户会话： db.session.commit()

查：①无条件查询：Book.query.查询执行器
②条件查询：Book.query.过滤器.查询执行器
常用过滤器如下：

过滤器	说明	示例
filter(条件)	返回符合该条件的查询集，BaseQuery对象	Book.query.filter(Book.id==1)
filter_by()	返回符合该等值条件的查询集	Book.query.filter_by(id=1)
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，返回一个新查询集	
order_by()	根据字段进行排序，默认是正序，返回新查询集,desc可以反序	Book.query.order_by(Book.id) Book.query.order_by(Book.id.desc)
group_by()	根据字段进行分组，返回新查询集合	

常用查询执行器如下：

方法	说明	示例
all()	以列表形式返回查询的所有结果	Book.query.filter(Book.id==1).all()
first()	返回查询的第一个结果，如果未查到，返回None	Book.query.filter(Book.id==1).first()
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	Book.query.get(1)
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	Book.query.count()
paginate()	返回一个Paginate对象，它包含指定范围内的结果,参数一：第几页，参数二：每页个数，参数3：如果没有该页数返回False	Book.query.paginate(2,3,False)

逻辑非，逻辑与，逻辑或
from sqlalchemy import not*,and*,or_
示例：User.query.filter(not*(User.name=='chen')).all()*
User.query.filter(and(User.name!='wang',User.email.endswith('163.com'))).all()

一对多，多对一关联查询
一对多：①先查询出一类对象，例如author=Author.query.get(1)
②根据我们设置的relationship属性获取这一类对象下的全部多类对象：
books=author.books (即该作者下全部书籍)
多对一：①查询出多类对象，例如book=Book.query.get(2)
②根据我们设置的backref反向设置的属性获取该多类对象对应的一类对象：
author =book.author (即这本书所属的作者)
flask数据库迁移
D：在数据库中建立模型类映射的数据库表，如果需要修改数据库模型，还要在修改之后更新数据库，最好的解决的方法使用数据库迁移框架Flask-Migrate
E：建立相关数据库表，而且追踪数据库模式的变化，然后把变动应用到数据库中，还可以回退版本。
U：**迁移步骤：**
①安装扩展包：迁移扩展包：pip install flask-migrate，脚本管理器包：pip install flask-script
②导包：from flask_migrate import Migrate,MigrateCommand from flask_script import Shell,Manager
③创建脚本管理器：manager= Manager（app）
④迁移关联应用和数据库：Migrate（app，db）
⑤添加迁移命令道脚本管理器：manager.add_command('db',MigrateCommand)
⑥使用脚本命令在命令行进行迁移操作：

创建迁移文件夹	**python xxx.py db init**
生成当前版本迁移文件	python xxx.py db migrate -m '版本说明'
执行当前本迁移文件	python xxx.py db upgrade
回退一个迁移版本	python xxx.py db downgrade
查看迁移历史记录	python xxx.py db history
向上迁移到指定版本	python xxx.py db upgrade 版本号
向下迁移到指定版本	python xxx.py db downgrade 版本号
查看当前迁移版本	python xxx.py db current

1.flask连接数据库的四步：

1. 倒入第三方数据库扩展包：from flask_sqlalchemy import SQLAlchemy
2. 配置config属性，连接数据库：

   app.config["SQLALCHEMY_DATABASE_URI"] = "mysql://root:mysql@localhost/first_flask"
   app.config["SQLALCHEMY_TRACK_MODIFICATIONS"] = False
3. 创建数据库first_flask
4. 创建操作数据库对象：db = SQLAlchemy(app)

下面直接上代码解释：

```
# -*- coding:utf-8 -*-
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
app = Flask(__name__)
# url的格式为：数据库的协议：//用户名：密码@ip地址：端口号（默认可以不写）/数据库名
app.config["SQLALCHEMY_DATABASE_URI"] =
"mysql://root:mysql@localhost/first_flask"
# 动态追踪数据库的修改. 性能不好. 且未来版本中会移除. 目前只是为了解决控制台的提示才写的
app.config["SQLALCHEMY_TRACK_MODIFICATIONS"] = False
# 创建数据库的操作对象
db = SQLAlchemy(app)
class Role(db.Model):
    __tablename__ = "roles"
    id = db.Column(db.Integer,primary_key=True)
    name = db.Column(db.String(16),unique=True)
    # 给Role类创建一个uses属性，关联users表。
    # backref是反向的给User类创建一个role属性，关联roles表。这是flask特殊的属性。
    users = db.relationship('User',backref="role")
    # 相当于__str__方法。
    def __repr__(self):
        return "Role: %s %s" % (self.id,self.name)
class User(db.Model):
    # 给表重新定义一个名称，默认名称是类名的小写，比如该类默认的表名是user。
    __tablename__ = "users"
    id = db.Column(db.Integer,primary_key=True)
    name = db.Column(db.String(16),unique=True)
    email = db.Column(db.String(32),unique=True)
    password = db.Column(db.String(16))
    # 创建一个外键，和django不一样。flask需要指定具体的字段创建外键，不能根据类名创建外键
    role_id = db.Column(db.Integer,db.ForeignKey("roles.id"))

    def __repr__(self):
        return "User: %s %s %s %s" %
(self.id,self.name,self.password,self.role_id)

@app.route('/')
def hello_world():
    return 'Hello World!'

if __name__ == '__main__':
    # 删除所有的表
    db.drop_all()
    # 创建表
    db.create_all()

    ro1 = Role(name = "admin")
    # 先将ro1对象添加到会话中，可以回滚。
    db.session.add(ro1)

    ro2 = Role()
    ro2.name = 'user'
    db.session.add(ro2)
    # 最后插入完数据一定要提交
    db.session.commit()

    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@itheima.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@itheima.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()
    app.run(debug=True)
```

下面插播一条bug：

当把表格创建完成，注释这两句话：

```
 # 删除所有的表
    db.drop_all()
    # 创建表
    db.create_all()
```

然后向表格里面插入数据，此时会出现这样的错误：

```
sqlalchemy.exc.IntegrityError: (_mysql_exceptions.IntegrityError) (1062,
"Duplicate entry 'admin' for key 'name'") [SQL: u'INSERT INTO roles (name)
VALUES (%s)'] [parameters: ('admin',)]
```

查了网上的好多资料说把字段的约束unique=True去掉就好了，但是根本原因不在这。

原因就是因为app.run(debug=True)。开启debug模式之后，当我们修改代码的时候，比如将删除表和创建表这两句话注释，然后打开插入数据的注释。这个过程debug模式默认就已经把程序运行一遍了。此时数据库就已经有了数据，当我们再次手动执行的时候，又往数据库中插入了一条数据，这时候就会报错。因为字段的约束是唯一性的unique，所以解决的办法有两种：

第一种：就是不要将删除表和创建表这两句话注释，每次执行都要带着这两个句话。无论是debug模式自动执行还是我们手动执行程序，都会先删除表然后再创建表，所以执行多少次都不怕。

第二种：关闭debug模式。就是这样app.run()

2.数据库的增删改查：

1.以下的方法都是返回一个新的查询，需要配合执行器使用。

filter(): 过滤，功能比较强大。
filter_by():过滤，用在一些比较简单的过滤场景。

order_by():排序。默认是升序，降序需要导包：from sqlalchemy import * 。然后引入desc方法。比如order_by(desc("email")).按照邮箱字母的降序排序。

group_by()：分组。

2.以下都是一些常用的执行器：配合上面的过滤器使用。

get():获得id等于几的函数。比如：查询id=1的对象。get(1)。切记：括号里没有“id=”，直接传入id的数值就ok。因为该函数的功能就是查询主键等于几的对象。

all():查询所有的数据。

first():查询第一个数据。

count():返回查询结果的数量。

paginate():分页查询，返回一个分页对象。paginate（参数1，参数2，参数3）

参数1：当前是第几页，参数2：每页显示几条记录，参数3：是否要返回错误。

返回的分页对象有三个属性：items：获得查询的结果，pages：获得一共有多少页，page：获得当前页。

3.常用的逻辑符：

需要倒入包才能用的有：from sqlalchemy import *

not_　　and_　　or_　还有上面说的排序desc。

常用的内置的有：in_　　表示某个字段在什么范围之中。

4.其他关系的一些数据库查询：

endswith（）：以什么结尾。

startswith（）：以什么开头。

contains（）：包含

5.下面体会一下上面的这些用法：

```
1. 查询所有用户数据
User.query.all()
2. 查询有多少个用户
User.query.count()
3. 查询第1个用户
User.query.first()

4. 查询id为4的用户[3种方式]
User.query.get(4)
User.query.filter_by(id=4).first()
User.query.filter(User.id==4).first()
filter:(类名.属性名==)
filter_by:(属性名=)

filter_by: 用于查询简单的列名，不支持比较运算符
filter比filter_by的功能更强大，支持比较运算符，支持or_、in_等语法。

5. 查询名字结尾字符为g的所有数据[开始/包含]
User.query.filter(User.name.endswith('g')).all()
User.query.filter(User.name.contains('g')).all()

6. 查询名字不等于wang的所有数据[2种方式]
 from sqlalchemy import not_注意了啊：逻辑查询的格式：逻辑符_(类属性其他的一些判断)
User.query.filter(not_(User.name=='wang')).all()
```

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

7. 查询名字和邮箱都以 li 开头的所有数据[2种方式]
from sqlalchemy import and_
User.query.filter(and_(User.name.startswith('li'),
User.email.startswith('li'))).all()
User.query.filter(User.name.startswith('li'), User.email.startswith('li')).all()

8. 查询password是 `123456` 或者 `email` 以 `itheima.com` 结尾的所有数据
from sqlalchemy import or_
User.query.filter(or_(User.password=='123456',
User.email.endswith('itheima.com'))).all()

9. 查询id为 [1, 3, 5, 7, 9] 的用户列表
User.query.filter(User.id.in_([1, 3, 5, 7, 9])).all()

10. 查询name为liu的角色数据
关系引用
User.query.filter_by(name='liu').first().role.name

11. 查询所有用户数据，并以邮箱排序
排序
User.query.order_by('email').all()  默认升序
User.query.order_by(desc('email')).all() 降序
12. 查询第2页的数据，每页只显示3条数据
help(User.query.paginate)
三个参数: 1. 当前要查询的页数 2. 每页的数量 3. 是否要返回错误

pages = User.query.paginate(2, 3, False)
pages.items # 获取查询的结果
pages.pages # 总页数
pages.page # 当前页数
```

3.使用第三方扩展框架迁移数据库文件。

使用框架需要配置的代码如下：

```
# -*- coding:utf-8 -*-
from flask import Flask
from flask_sqlalchemy import SQLAlchemy  # 操作数据库的扩展包
from flask_script import Manager  # 用命令操作的扩展包
from flask_migrate import Migrate,MigrateCommand  # 操作数据库迁移文件的扩展包

app = Flask(__name__)
app.debug = True
app.config["SQLALCHEMY_DATABASE_URI"] =
"mysql://root:mysql@localhost/second_flask"
app.config["SQLALCHEMY_TRACK_MODIFICATIONS"] = False

db = SQLAlchemy(app)
manager = Manager(app)
# 创建迁移对象
migrate = Migrate(app,db)
# 将迁移文件的命令添加到‘db’中
manager.add_command('db',MigrateCommand)

class Role(db.Model):
```

```
    __tablename__ = "table_roles"
    id = db.Column(db.Integer,primary_key=True)
    name = db.Column(db.String(16),unique=True)
    info = db.Column(db.String(100))
    Users = db.relationship("User",backref='role')

class User(db.Model):
    __tablename__ = "table_users"
    id = db.Column(db.Integer,primary_key=True)
    name = db.Column(db.String(16),unique=True)
    info = db.Column(db.String(200))
    role_id = db.Column(db.Integer,db.ForeignKey("table_roles.id"))


@app.route('/')
def hello_world():
    return 'Hello World!'

if __name__ == '__main__':

    manager.run()
```

使用迁移命令如下：

比如上面的代码所在的文件名称为database.py。

1.python database.py db init　　生成管理迁移文件的migrations目录

2.python database.py db migrate -m "注释"　　在migrations/versions中生成一个文件，该文件记录数据表的创建和更新的不同版本的代码。

3.python database.py db upgrade　　在数据库中生成对应的表格。

4.当需要改表格的时候，改完先执行第二步，然后再执行第三步。

5.需要修改数据表的版本号的时候需要做的操作如下：

python database.py db upgrade 版本号　　向上修改版本号

python database.py db downgrade 版本号　　向下修改版本号

可能用到的其他的语句：

python database.py db history　　查看历史版本号

python database.py db current　　查看当前版本号