

##3.5数据库迁移

1.准备工作

在虚拟环境中安装扩展Flask-Migrate（实现数据库的迁移）

导入所需要用到的包

```
from flask import Flask
```

```
from flask_sqlalchemy import SQLAlchemy
```

```
from flask_script import Manager
```

```
from flask_migrate import Migrate,MigrateCommand
```

迁移准备工作

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

1. 导入所需要的包

```
app = Flask(__name__)
```

```
# 先进行数据库配置
```

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

```
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
```

```
# 再创建数据库对象
```

```
app.debug = True
```

```
db = SQLAlchemy(app)
```

3. 创建数据库对象

```
# 命令行
```

```
manager = Manager(app)
```

4. 创建命令行

```
Migrate(app,db)
```

5. 设置迁移的对象

```
#给manager添加数据库迁移的命令
```

```
# 第一个参数是将来命令行要使用的命令，
```

```
manager.add_command('db',MigrateCommand)
```

6. 给命令行manager添加数据库迁移的命令

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

```
def hello_world():
```

```
    return 'Hello World!'
```

迁移的命令

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

2. 创建表的模型类

```
#给manager添加数据库迁移的命令
# 第一个参数是将来命令行要使用的命令，
manager.add_command('db',MigrateCommand)
```

```
# 创建表模型类
class Role(db.Model):
    __tablename__ = 'table_role'
    id = db.Column(db.Integer,primary_key=True)
    name = db.Column(db.String(32),unique=True)
    users = db.relationship('User',backref='_role')

class User(db.Model):
    __tablename__ = 'table_user'
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(32),unique=True)
    role_id = db.Column(db.INTEGER,db.ForeignKey('table_role.id'))

    '多方' 设置外键
```

‘一方’ 设置
关系引用

创建表的模型类

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

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

代码

```
1  # -*- coding:utf-8 -*-
2  from flask import Flask
3  from flask_sqlalchemy import SQLAlchemy
4  from flask_script import Manager
5  from flask_migrate import Migrate,MigrateCommand
6
7  app = Flask(__name__)
8  # 先进行数据库配置
9  app.config['SQLALCHEMY_DATABASE_URI'] = 'mysql://root:mysql@127.0.0.1/migrate'
10 app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
11 # 再创建数据库对象
12 app.debug = True
13 db = SQLAlchemy(app)
14 #命令行
15 manager = Manager(app)
16 Migrate(app,db)
17
18 #给manager添加数据库迁移的命令
19 # 第一个参数是将来命令行要使用的命令，
20 manager.add_command('db',MigrateCommand)
21
22 # 创建表模型类
23 class Role(db.Model):
24     __tablename__ = 'table_role'
25     id = db.Column(db.Integer,primary_key=True)
26     name = db.Column(db.String(32),unique=True)
27     users = db.relationship('User',backref = 'role')
```

```

28
29 class User(db.Model):
30     __tablename__ = 'table_user'
31     id = db.Column(db.Integer, primary_key=True)
32     name = db.Column(db.String(32), unique=True)
33     role_id = db.Column(db.INTEGER, db.ForeignKey('table_role.id'))
34
35 @app.route('/')
36 def hello_world():
37     return 'Hello World!'
38
39 if __name__ == '__main__':
40     app.run()
41

```

3.执行迁移命令操作

第一步执行init命令-初始化迁移(只需要执行一次)

```
1 python sql_migrate.py db init
```

会在当前目录下生成一个migrations的文件

第二步执行migrate命令-生成迁移文件

```
1 python sql_migrate.py db migrate
```

第三步，执行update命令-执行迁移

```
1 python sql_migrate.py db upgrade
```

其他命令操作

查看数据库迁移记录

```
python app.py db history
```

1.版本回滚

downgrade后面不加版本号，默认回归到前一个版本

```
1 python app.py db downgrade
```

2.回滚到指定版本

```
1 python app.py db downgrade 版本号
```

3.更新到指定版本

```
1 python sql_migrate.py db upgrade 版本号
```

操作顺序总结：

- 1.python 文件 db init
- 2.python 文件 db migrate -m"版本名(注释)"
- 3.python 文件 db upgrade 然后观察表结构
- 4.根据需求修改模型
- 5.python 文件 db migrate -m"新版本名(注释)"
- 6.python 文件 db upgrade 然后观察表结构
- 7.若返回版本,则利用 python 文件 db history查看版本号
- 8.python 文件 db downgrade(upgrade) 版本号

#4发送邮件

步骤：

第一步导入Mail和Message

```
1 from flask_mail import Mail,Message
```

第二步，配置参数

```
1 app.config['MAIL_SERVER'] = "smtp.126.com"
2 app.config['MAIL_PORT'] = 465
3 app.config['MAIL_USE_SSL'] = True
4 app.config['MAIL_USERNAME'] = "huidongpeng@126.com"
5 app.config['MAIL_PASSWORD'] = "heima666"
6 app.config['MAIL_DEFAULT_SENDER'] = 'FlaskAdmin<huidongpeng@126.com>'
```

第三步，创建邮件对象

```
1 mail = Mail(app)
```

第四步，创建发送邮件的试图函数

```
1 @app.route('/send_mail')
2 def send_mail():
3     message = Message('邮件发送演示',recipients=['yrt333333@163.com'])
4     # message.body = '欢迎使用flask发邮件'
5     message.html = '<h1>这是用html格式发送的邮件</h1>'
6     mail.send(message)
7     return "发送成功!"
```

```
# -*- coding:utf-8 -*-
```

```
from flask import Flask
from flask_mail import Mail,Message
app = Flask(__name__)
```

1. 导入Mail,Message包

```
#配置发邮件参数：服务器/端口/安全套接字层/邮箱名/授权码/默认发送人
```

```
app.config['MAIL_SERVER'] = "smtp.163.com"
app.config['MAIL_PORT'] = 465
app.config['MAIL_USE_SSL'] = True
app.config['MAIL_USERNAME'] = "yrt333333@163.com"
app.config['MAIL_PASSWORD'] = "yrt333333"
app.config['MAIL_DEFAULT_SENDER'] = 'FlaskAdmin<yrt333333@163.com>'
app.debug = True
```

2. 配置参数

```
#创建邮件
```

```
mail = Mail(app)
```

3. 创建邮件对象

```
@app.route('/')
def mail_link():
```

```
    return '<a href="/send_mail">点击发送邮件</a>'
```

发送对象，可以是多个，用列表存放

```
@app.route('/send_mail')
```

```
def send_mail():
    message = Message('邮件发送演示',recipients=['yrt333333@163.com'])
    # message.body = '欢迎使用flask发邮件'
    message.html = '<h1>这是用html格式发送的邮件</h1>'
    mail.send(message)
    return "发送成功!"
```

4. 编辑邮件内容，执行发送邮件命令

邮件主题内容

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

邮件发送命令

代码实现

```
1 # -*- coding:utf-8 -*-
2 from flask import Flask
3 from flask_mail import Mail,Message
4 app = Flask(__name__)
5
6 #配置发邮件参数：服务器/端口/安全套接字层/邮箱名/授权码/默认发送人
7 app.config['MAIL_SERVER'] = "smtp.163.com"
8 app.config['MAIL_PORT'] = 465
9 app.config['MAIL_USE_SSL'] = True
```

```

10 app.config['MAIL_USERNAME'] = "yrt333333@163.com"
11 app.config['MAIL_PASSWORD'] = "yrt333333"
12 app.config['MAIL_DEFAULT_SENDER'] = 'FlaskAdmin<yrt333333@163.com>'
13 app.debug = True
14 #创建邮件
15 mail = Mail(app)
16
17 @app.route('/')
18 def mail_link():
19     return '<a href="/send_mail">点击发送邮件</a>'
20
21 @app.route('/send_mail')
22 def send_mail():
23     message = Message('邮件发送演示',recipients=['yrt333333@163.com'])
24     # message.body = '欢迎使用flask发邮件'
25     message.html = '<h1>这是用html格式发送的邮件</h1>'
26     mail.send(message)
27     return "发送成功！"
28
29
30 if __name__ == '__main__':
31     app.run()

```

#5.蓝图-可以让应用实现模块化

1.原理：蓝图相当于一个储存操作（注册路由）方法的容器，当这个蓝图在其他应用中之后就能够调用这些操作；

创建蓝图的步骤

第一步：创建蓝图对象

第二步：使用蓝图对象实现相关路由

第三步：在app创建的地方注册蓝图

2.蓝图前缀-url_prefix

可以写在蓝图中

```

1 app_user = Blueprint('user',__name__,url_prefix='/api')

```

也可以写在应用中

```

1 app.register_blueprint(app_user,url_prefix='/api')

```

3.实现目录形式的蓝图模块

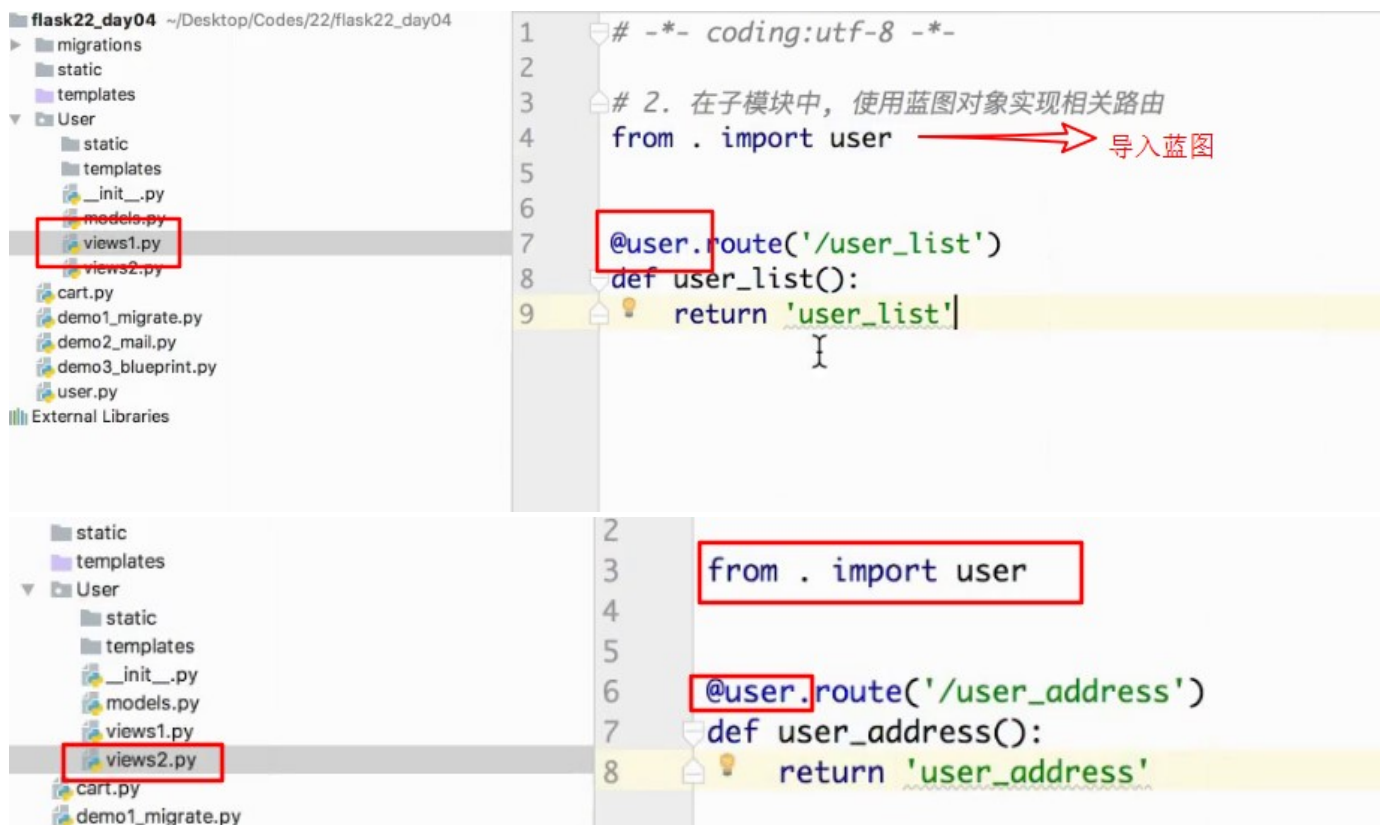
第一步：在应用的init文件中创建蓝图对象，并且导入其他子模块（蓝图对象默认不会自动关联相关路由，需要导入子模块让其发生关联）；



实操代码

```
1 # -*- coding:utf-8 -*-  
2 # 1. 创建蓝图对象，在__init__文件中执行，并导入其他子模块  
3 from flask import Blueprint  
4  
5 # 蓝图对象，默认没有指定static和templates文件夹。使用时必须先指定  
6 user = Blueprint('user', __name__, template_folder='templates')  
7  
8 # 蓝图对象默认不会自动关联相关的路由，需要导入子模块让其发生关联  
9 import views1, views2
```

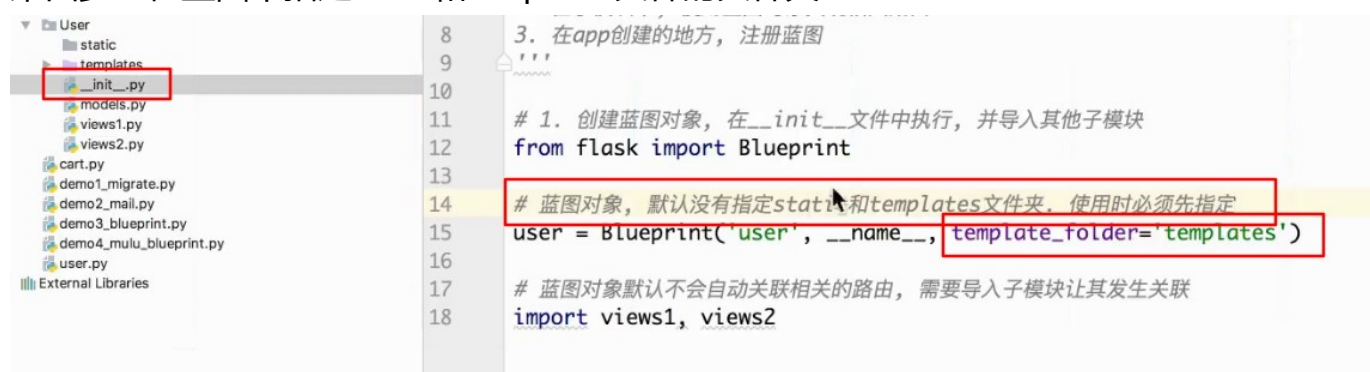
第二步：在子模块中，使用蓝图对象实现相关的路由



第三步：在应用中注册蓝图



第四步：在蓝图中指定static和templates文件的文件夹



#6.单元测试-编写代码去检测代码

1.断言-assert

用法：assert 判断条件 不满足条件，抛出错误提示信息（满足条件判断则继续往下执行）

常用的断言方法

断言方法	描述
assertEqual	如果两个值相等，则pass
assertNotEqual	如果两个值不相等，则pass
assertTrue	判断bool值为True，则pass
assertFalse	判断bool值为False，则pass
assertIsNone	不存在，则pass
assertIsNotNone	存在，则pass

2.单元测试的步骤

第一步：导入unittest，导入需要进行单元测试的类

```
1 import unittest
```



```
2 from demo6_login import app
```

第二步：定义一个登陆验证的测试类，继承unittest.TestCase类

```
1 class LoginTestCase(unittest.TestCase):
```

第三步：在测试类的内部，针对不同的情况定义不同的测试用例进行验证

测试类内部写法

```
class LoginTestCase(unittest.TestCase):  
    # 该方法会首先执行，方法名为固定写法  
    # 公用的代码 / 初始化的配置  
    def setUp(self):  
        # 开启测试模式  
        app.testing = True  
  
        # 绑定公用的client属性  
        self.client = app.test_client()  
  
    # 一、用户名和密码都没传的情况  
    # 测试用例：必须已test_开头，否则就无法找到该测试用例  
    def test_empty_username_password(self):...  
  
    # 测试 用户名和密码输入正确 --> 0  
    def test_correct_username_password(self):...  
  
    # TODO(zhubo) 还差3个用例  
  
    # 可以做一些测试之后的收尾工作  
    def tearDown(self):  
        # db.session.remove()  
        # db.drop_all()  
        pass  
  
# 在终端，通过python 文件名 运行测试  
if __name__ == '__main__':  
    unittest.main()
```

固定的方法，在一开始就执行，可以将初始化或者公共的配置写在此方法中

中间部分用于编写测试用例

固定方法，做一些测试收尾的工作

第四步：用户名和密码为空的测试用例的实现

测试用户名，密码是否为空的用例实现

```
def test_empty_username_password(self):  
    # 1. 获取测试客户端  
    # client = app.test_client()  
  
    # 2. 测试客户端发送请求  
    # data以字典方式去填写  
    # client.post发送请求，并返回结果  
    response = self.client.post('/login', data={})  
  
    # 3. 获取响应数据  
    response_data = response.data  
  
    # 4. 将字符串数据转为字典  
    # json.dumps() 将字典数据转为字符串  
    # json.loads() 将字符串数据转为字典  
    response_dict = json.loads(response_data)  
  
    # 5. 先判断是否有errcode  
    self.assertIn('errcode', response_dict, 'no errcode')  
  
    # 6. 再判断errcode是否为-2  
    errcode = response_dict['errcode']  
    self.assertEqual(errcode, -2, 'errcode must is -2, but current is %s' % errcode)  
    print response  
    print errcode
```

先获取测试客户端模拟请求

获取到响应数据（响应数据为对象，调用data属性转为字符串）

将数据由字符串转为字典

此处断言先判断是否有errcode值，没有则会抛出异常；有，则会继续往下执行

此处断言先判断errcode值是否为-2，不是则会抛出异常；是，则会继续往下执行

操作代码

```

1  # -*- coding:utf-8 -*-
2  import unittest
3  import json
4  from demo6_login import app
5
6  # 测试时，如果光标定位到某个函数内执行，只会对单个函数进行测试
7  # 如果要测试全部的用例，在顶部或底部执行。即可对所有情况进行测试
8
9  # 针对一个接口(API)的测试，写一个类
10 # 类的内部，针对不同的测试用例(不同的测试情况)，写不同的函数
11
12 class LoginTestCase(unittest.TestCase):
13
14     # 该方法会首先执行，方法名为固定写法
15     # 公用的代码 / 初始化的配置
16     def setUp(self):
17         # 开启测试模式
18         app.testing = True
19
20         # 绑定公用的client属性
21         self.client = app.test_client()
22
23     # 一. 用户名和密码都没传的情况
24     # 测试用例：必须已test_开头。否则就无法找到该测试用例
25     def test_empty_username_password(self):
26
27         # 1. 获取测试客户端
28         # client = app.test_client()
29
30         # 2. 测试客户端发送请求
31         # data以字典方式去填写
32         # client.post发送请求，并返回结果
33         response = self.client.post('/login', data={})
34
35         # 3. 获取响应数据
36         response_data = response.data
37
38         # 4. 将字符串数据转为字典
39         # json.dumps() 将字典数据转为字符串
40         # json.loads() 将字符串数据转为字典
41         response_dict = json.loads(response_data)
42
43         # 5. 先判断是否有errcode
44         self.assertIn('errcode', response_dict, 'no errcode')
45

```

```

46         # 6. 再判断errcode是否为-2
47         errcode = response_dict['errcode']
48         self.assertEqual(errcode, -2, 'errcode must is -2, but current is %s'
% errcode)
49         print response
50         print errcode
51
52     # 测试 用户名和密码输入正确 --> 0
53     def test_correct_username_password(self):
54         # client = app.test_client()
55         response = self.client.post('/login', data={'username':'itheima',
'password':'python'})
56         response_data = response.data
57         response_dict = json.loads(response_data)
58         self.assertIn('errcode', response_dict, 'no errcode')
59         errcode = response_dict['errcode']
60         self.assertEqual(errcode, 0, 'errcode must is 0, but current is %s' %
errcode)
61         print errcode
62
63     # TODO(zhubo) 还差3个用例
64
65     # 可以做一些测试之后的收尾工作
66     def tearDown(self):
67         # db.session.remove()
68         # db.drop_all()
69         pass
70
71
72 # 在终端, 通过python 文件名 运行测试
73 if __name__ == '__main__':
74     unittest.main()
75

```