小回顾

单元测试

Django测试组成

Django 的单元测试采用 Python 的标准模块： unittest。该模块以类的形式定义测试。Django继承了该模块并对其做了一定修改，使其能配合Django的特性做一些测试。

主要测试对象：数据模型

运行方法: python manage.py test

测试文件：app目录下的test.py

Django测试注意点

使用Django提供的TestCase类作为被继承类。注意这里不是unittest的TestCase，否则无法使用django的测试功能。

创建数据时采用的是测试数据库，并且会在测试结束后销毁

测试行为不会影响到真实的数据库。

Django测试实现

案例：

```
from django.test import TestCase
from .models import Event,Guest
from datetime import datetime
# Create your tests here.

#发布会测试
class EventTestCase(TestCase):
    def setUp(self):
        Event.objects.create(name='测试训练营1',address='软件大
道',start_time=datetime.now(),limits=1000)
        Event.objects.create(name='测试训练营2',address='软件大
道',start_time=datetime.now(),limits=500)

    def test_event_address(self):
        event1 = Event.objects.get(name='测试训练营1')
        event2 = Event.objects.get(name='测试训练营2')
        self.assertEqual(event1.address,'软件大道')
        self.assertEqual(event2.address,'软件大道')

    def test_event_limits(self):
        event1 = Event.objects.get(name='测试训练营1')
        event2 = Event.objects.get(name='测试训练营2')
        self.assertEqual(event1.limits,1000)
```

```
        self.assertEqual(event2.limits,500)

    def test_address_update(self):
        event1 = Event.objects.get(name='测试训练营1')
        event1.address = '花神大道'
        event1.save()
        self.assertEqual(event1.address,'花神大道')

    def test_delete(self):
        #删前列出所有
        event_list1 = Event.objects.all()
        event1 = Event.objects.get(name='测试训练营1')
        self.assertIn(event1,event_list1) #检查
        #删除
        event1.delete()
        #删后列出所有
        event_list2 = Event.objects.all()
        self.assertNotIn(event1,event_list2)  #校验结果


#嘉宾测试
class GuestTestCase(TestCase):
    def setUp(self) -> None:  # -> None 新语法 表示返回空
        #嘉宾需要关联发布会，所以创建发布会
        self.event = Event.objects.create(name='测试训练营1',address='软件大
道',start_time=datetime.now(),limits=1000)
        Guest.objects.create(name='小
张',phone='13912345678',email='hello@test.com', event=self.event)

    def test_query(self):
        guest = Guest.objects.filter(name='小张')[0]
        self.assertEqual(guest.phone,'13912345678')


    def test_update(self):
        guest = Guest.objects.filter(name='小张')[0]
        guest.name='小明'
        guest.save()
        guest2 = Guest.objects.filter(name='小明')[0]
        self.assertEqual(guest.phone , guest2.phone)

    def test_delete(self):
        guest = Guest.objects.filter(name='小张')[0]
        guest.delete()
        guest_list = Guest.objects.all()
        self.assertNotIn(guest,guest_list)


    def test_add(self):
        payload = {
            'name':'小强',
            'phone':'13712345678',
            'email':'xiaoq@test.com',
            'event_id':self.event.id
        }
        # 模拟客户端发起请求
        self.client.post('/sgin/add_guest/',data=payload)
        guest = Guest.objects.filter(name='小强')[0]
```

```
        self.assertEqual(guest.phone,'13712345678')

"""
运行所有用例：
python3 manage.py test

运行sign应用下的所有用例：
python3 manage.py test sign

运行sign应用下的tests.py文件用例：
python3 manage.py test sign.tests

运行sign应用下的tests.py文件中的 GuestTestCase 测试类：
python3 manage.py test sign.tests.GuestTestCase
......

"""
```

单元测试主要测试模型的基本功能，测试场景可能不够全面，所以大家了解这个功能即可。在项目中通常使系统测试，集成测试，接口测试等方法对产品功能进行覆盖测试。

数据库表关联多对多关系

多对多原理以及字段定义方法

回顾下之前嘉宾与发布会：多个个嘉宾对应一个，这多对1，那么这个情况和真实的不太相符，一个嘉宾也可以参与多个发布会，因此，改成多对多更合理

多对多与多对1的定义方式不同，注意对比

u外键的定义方式（多对1）

models.ForeignKey(目标模型类, on_delete=models.CASCADE)

此外键定义在多的1方

u外键的定义方式（多对多）相对复杂，

models.ManyToManyField(目标模型类)

只可以在1方定义，不可以两方同时定义

多对多应用

修改发布会嘉宾的关系，变为多对多，外键定义在嘉宾这里，因为从业务来看，是嘉宾选择发布会，所以定义在嘉宾这里更合理。

```
# 定义发布会关联嘉宾
class Guest(models.Model):
    # django会自动帮你创建一个id字段作为主键
    # 关联发布会
    # event = models.ForeignKey(Event,on_delete=models.CASCADE)  #CASCADE 如果删除了关联的发布会，该嘉宾也会被删除
    events = models.ManyToManyField(Event)
```

执行 python manage.py makemigrations 和python manage.py migrate

完成后，数据库会自动新增一个关系表，发布会和嘉宾之间的关系，就存储在这里

主张表的名字默认是`应用名_模型1_模型2` 如：`sgin_guest_events`

关系发生变动后，视图和模板也需要进行相应的修改

视图修改

首先视图部分：add_guest

原来是关联1个发布会就够了，现在需要改成关联多个发布会，所以参数应该传递发布会的ID列表event_ids

更改前：

```
#关联发布会
event_id = request.POST['event_id']
```

更改后：

```
#关联发布会
#event_ids = request.POST['event_ids']  #这种方式只会获取最后一个值--错误方式
event_ids = request.POST.getlist('event_ids')  #获取值列表  --正确方式
```

此时，新增嘉宾，多对多需要另外进行处理，先创建嘉宾数据，再将其进行关联

```
guest = Guest.objects.create(name=name,phone=phone,email=email)
```

创建嘉宾数据时，不要传入event_ids,应该用数据对象进行关联，方法是：

方法1：

数据对象.多方.add(多方数据对象1,多方数据对象2,多方数据对象3,.....)

可以简化成数据对象.多方.add(*多方数据对象列表) # 利用自动解包可以省略手动传参

方法2：

数据对象.多方.set(多方数据对象列表)

```
#根据event_ids查找发布会数据列表
events = [Event.objects.get(pk=event_id) for event_id in event_ids ]
#将发布会数据列表关联到当前嘉宾
guest.events.add(*events)
# 或者 guest.events.set(events)
```

签到视图修改 do_sgin

由于guest对应了多个发布会，所以不能通过guest.event来查看关联的发布会，应该通过guest.events.all()

```
def do_sgin(request,event_id):
    # 从post请求获取参数，首先判断请求方法
    if request.method == 'POST':
```

```
        phone = request.POST.get('phone')
        current_event = Event.objects.get(pk=event_id)
        # 判断手机号是否正确
        res = Guest.objects.filter(phone=phone)
        if not res:
            return render(request,'event_detail.html',
{'event':current_event,'error':'手机号错误'})
        guest = res[0]
        # 是否属于当前发布会
        event_ids = [d[0] for d in guest.events.values_list('id')] # 取出所有关联
的发布会ID
        if event_id not in event_ids:
            return render(request,'event_detail.html',
{'event':current_event,'error':'非当前发布会嘉宾'})
        # 是否已经签到
        if guest.is_sgin:
            return render(request,'event_detail.html',
{'event':current_event,'error':'已签到，不要重复签到'})

        #进入签到
        guest.is_sgin=True
        guest.save()
        return redirect(f'/sgin/sgin_success/{phone}')
```

模板修改

现在的添加嘉宾这里selected下拉框只能选择1个发布会，改成多选框

来看修改前：

```
<select class="form-control" name="event_id">
  {% for event in events %}
    <option value={{ event.id }}>{{ event.name }}</option>
  {% endfor %}
</select>
```

修改后：

```
<select class="form-control" name="event_ids" multiple>
    {% for event in events %}
        <option value={{ event.id }}>{{ event.name }}</option>
    {% endfor %}
</select>
```

修改了name 增加了multiple属性

刷新页面创建一个嘉宾进行测试，发现成功。进入该嘉宾详情页，发现参与的发布会没有了

去对应的模板页面guest_detail.html查看：

```
<p>参与发布会：{{ guest.event }}</p> 这里还是用之前的属性
```

更改后：

```
<p>参与发布会：{% for event in guest.events.all %}<span>{{ event }}</span>{%
endfor %}</p>
```

扩展：关联发布会加上超链接：

```
<p>参与发布会：{% for event in guest.events.all %}<a href="/sgin/event_detail/{{
event.id }}"> {{ event }} |</a>{% endfor %}</p>
```

多对多中间表

接下来请大家仔细观察发布会关联嘉宾这里的模型定义。

```
    # 加入时间 --创建数据的时候就自动取当前时间 auto_now_add=True
    join_time = models.DateTimeField(auto_now_add=True)
    # 是否签到
    is_sgin = models.BooleanField(default=False)
```

注意这两个字段，有没有什么问题?

之前的嘉宾和发布会是多对 1的关系，那么1个嘉宾保存对应的加入时间和签到状态是没有问题的，但是现在改成了多对多，那么数据还这样存储的话就会出现嘉宾对关联的发布会1进行签到，但是再签到发布会2，3的时候发现也签到了。因为签到状态绑定在了嘉宾这边，修改了签到状态就相当于把所有发布会都签到了，这显然不合理。同样加入时间也是一样

那么这两个这个字段应该定义在哪里呢，仔细想想表示嘉宾和发布会两表之间的关系在哪?

答案是：中间表

我们可以在中间表这里再添加两个字段，重新定义该中间表。由于我们之前没有显示定义该表，这个表是自动创建的。现在我们显示定义该表

```
#sgin/models.py
class GuestEvent(models.Model):
    # 通过外键关联对应数据，on_delete设置为删除对应的数据即删除该条记录，如发布会1对应嘉宾2
    # 当发布会1或嘉宾2任意一个被删除，这个条对应关系也就不应该存在了
    event = models.ForeignKey(Event,verbose_name='发布
会',on_delete=models.CASCADE)
    guest = models.ForeignKey(Guest,verbose_name='嘉宾',on_delete=models.CASCADE)
    # 加入时间 --创建数据的时候就自动取当前时间 auto_now_add=True
    join_time = models.DateTimeField(auto_now_add=True)
    # 是否签到
    is_sgin = models.BooleanField(default=False)
```

修改发布会

```
# 定义发布会关联嘉宾
class Guest(models.Model):
    # django会自动帮你创建一个id字段作为主键
    # 关联发布会
    # event = models.ForeignKey(Event,on_delete=models.CASCADE)  #CASCADE 如果删除了关联的发布会，该嘉宾也会被删除
    events = models.ManyToManyField(Event,through='GuestEvent') # through=中间表明
    # 姓名  字符串  64  唯一
    name = models.CharField(max_length=64,unique=True)
    # 手机号  字符串  11  唯一
    phone = models.CharField(max_length=11,unique=True)
    # 邮箱  邮箱格式 xxx@yyy.zz
    email = models.EmailField()
```

运行migrate时提示错误

```
ValueError: Cannot alter field sgin.Guest.events into sgin.Guest.events - they are not compatible types (you cannot alter to or from M
2M fields, or add or remove through= on M2M fields)
```

修改表名--`应用名_模型_多对对关联字段` 和默认创建的表名相同

```
class Meta:  # 元类 作用是可以设置模型信息
   db_table = "sgin_guest_events"   #模型对应的表名
```

原迁移文件是要创建一个表，但是现在我的数据在刚刚默认创建的中间表中，因此只需要修改中间表字段就可以，当我们要修改数据库中已存在的表，就要修改迁移文件了。**这里属于高端操作，如果我们一开始就定义数据模型，那么这一步可以不需要。**

修改最新的迁移文件（运行makemigrations命令生成的最新文件，在应用程序/migrations下面）

```
#修改
state_operations = [
        migrations.RemoveField(
            model_name='guest',
            name='is_sgin',
        ),
        migrations.RemoveField(
            model_name='guest',
            name='join_time',
        ),
        migrations.CreateModel(
            name='GuestEvent',
            fields=[
                ('id', models.AutoField(auto_created=True, primary_key=True, serialize=False, verbose_name='ID')),
                ('join_time', models.DateTimeField(auto_now_add=True)),
                ('is_sgin', models.BooleanField(default=False)),
                ('event', models.ForeignKey(on_delete=django.db.models.deletion.CASCADE, to='sgin.event', verbose_name='发布会')),
                ('guest', models.ForeignKey(on_delete=django.db.models.deletion.CASCADE, to='sgin.guest', verbose_name='嘉宾')),
```

```
            ],
            options={
                'db_table': 'sgin_guest_event',
            },
        ),
        migrations.AlterField(
            model_name='guest',
            name='events',
            field=models.ManyToManyField(through='sgin.GuestEvent',
to='sgin.Event'),
        ),
        #指定修改的数据库
        migrations.AlterModelTable(
            name='GuestEvent',  # 中间表的模型名
            table='sgin_guest_events'  # 当前数据库中的中间表名
        )
    ]
    # 重新定义数据库操作
    operations = [
        #分离数据库状态
        migrations.SeparateDatabaseAndState(state_operations=state_operations),
        # 新增join_time字段
        migrations.AddField(
            model_name='GuestEvent',
            name='join_time',
            field=models.DateTimeField(auto_now_add=True),
        ),
        # 新增is_sgin字段
        migrations.AddField(
            model_name='GuestEvent',
            name='is_sgin',
            field=models.BooleanField(default=False),
        ),
        migrations.AlterModelTable(
            name='GuestEvent',
            #不创建实际的表
            table=None,
        ),
    ]
```

视图修改

```
def do_sgin(request,event_id):
        #.....前置代码
# 是否已经签到
        ge=GuestEvent.objects.get(guest_id=guest.id,event_id=event_id)
        if ge.is_sgin:
            return render(request,'event_detail.html',
{'event':current_event,'error':'已签到，不要重复签到'})

        #进入签到--操作中间模型
        ge.is_sgin=True
        ge.save()
```

数据库事务

看下创建嘉宾的代码

```
guest = Guest.objects.create(name=name,phone=phone,email=email)
#根据event_ids查找发布会数据列表
events = [Event.objects.get(pk=event_id) for event_id in event_ids ]
#将发布会数据列表关联到当前嘉宾
# guest.events.add(*events)
guest.events.set(events)
```

#如果创建嘉宾的时候没有问题，而查找发布会或关联发布会的步骤出现了问题，就会出现嘉宾创建了，但是没有关联到发布会，这是一个bug

我们应该这样处理，要么一次性成功创建且关联到发布会，要么失败不创建。

在数据库中这种要成功就一起成功，要失败就一起失败的做法叫事务，如果这里嘉宾创建成功但是后面的步骤失败了，那么数据库会发起回滚操作，将创建的嘉宾撤销。

在django中，可以用with transaction.atomic(): 语句块实现

```
from django.db import transaction
with transaction.atomic():
    #事务操作。。。
```

创建嘉宾代码部分可以改成：

```
#创建嘉宾
        try:
            with transaction.atomic():
                guest = Guest.objects.create(name=name,phone=phone,email=email)
                #根据event_ids查找发布会数据列表
                events = [Event.objects.get(pk=event_id) for event_id in
event_ids ]
                #将发布会数据列表关联到当前嘉宾
                # guest.events.add(*events)
                guest.events.set(events)
        except Exception as e:
            return render(request,'guest_add.html',{'error': repr(e)}) #返回精简错
误信息
            #保存成功-跳转到嘉宾列表页
        return redirect('/sgin/guests/')
```

项目总结梳理

1.业务逻辑梳理

2.知识点梳理

3.后续的学习任务重点与准备