django_study Documentation

发布 *v1.0*

yuanjh

2021 年 01 月 13 日

Contents:

公司业务需要，需要将原 flask 框架项目迁移到 django 上，django 本人只是在学校时简单使用过，况且也是好久之前的事情了，顺便整理下相关知识点

CHAPTER 1

django 入门进阶 01 学习笔记 01

学习笔记, 第一个 Django 应用

1.1 Part 1：请求与响应

django-admin startproject mysite

一个新建立的项目结构大概如下：

```
mysite/
    manage.py
    mysite/
        __init__.py
        settings.py
        urls.py
        asgi.py
        wsgi.py
```

python manage.py startapp polls

系统会自动生成 polls 应用的目录，其结构如下：

```
polls/
    __init__.py
    admin.py
```

(continues on next page)

(续上页)

```
    apps.py
    migrations/
        __init__.py
    models.py
    tests.py
    views.py
```

polls/views.py 文件中，编写代码：

```
from django.http import HttpResponse

def index(request):
    return HttpResponse("这里是 liujiangblog.com 的投票站点")
```

urls.py（不要换成别的名字），在其中输入代码如下：

```
from django.urls import path

from . import views

urlpatterns = [
    path('', views.index, name='index'),
]
```

此时，目录的文件结构是这样的：

```
polls/
    __init__.py
    admin.py
    apps.py
    migrations/
        __init__.py
    models.py
    tests.py
    urls.py
    views.py
```

mysite/urls.py 文件，代码如下：

```
from django.contrib import admin
from django.urls import include, path

urlpatterns = [
    path('polls/', include('polls.urls')),
```

(continues on next page)

(续上页)

```
    path('admin/', admin.site.urls),
]
```

1.2 Part 2：模型与后台

polls/models.py

```
from django.db import models


class Question(models.Model):
    question_text = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')


class Choice(models.Model):
    question = models.ForeignKey(Question, on_delete=models.CASCADE)
    choice_text = models.CharField(max_length=200)
    votes = models.IntegerField(default=0)
```

INSTALLED_APPS 中，将该路径添加进去（字符串格式）

python manage.py makemigrations polls

python manage.py sqlmigrate polls 0001# 展示 SQL 语句

python manage.py check# 检查项目中的错误，并不实际进行迁移或者链接数据库的操作。

python manage.py migrate# 运行 migrate 命令，在数据库中进行真正的表操作了。

进入 Python 的 shell，请输入命令：

python manage.py shell

相比较直接输入 “python” 命令的方式进入 Python 环境，调用 manage.py 参数能将 DJANGO_SETTINGS_MODULE 环境变量导入，它将自动按照 mysite/settings.py 中的设置，配置好你的 python shell 环境，这样，你就可以导入和调用任何你项目内的模块了。

或者你也可以这样，先进入一个纯净的 python 环境，然后启动 Django，具体如下：

```
>>> import django
>>> django.setup()
```

尝试下面的 API 吧

```
>>> from polls.models import Question, Choice # 导入我们写的模型类

# 现在系统内还没有 questions 对象
>>> Question.objects.all()
```

1.3 Part 3：视图和模板

get_object_or_404() 方法将一个 Django 模型作为第一个位置参数，后面可以跟上任意数量的关键字参数，如果对象不存在则弹出 Http404 错误。

get_list_or_404() 方法，和上面的 get_object_or_404() 类似，只不过是用来替代 filter() 函数

删除模板中硬编码的 URLs

```
<li><a href="{% url 'detail' question.id %}">{{ question.question_text }}</a></li>
```

Django 会在 polls.urls 文件中查找 name=’detail’ 的路由，具体的就是下面这行：

```
path('<int:question_id>/', views.detail, name='detail'),
```

让我们将代码修改得更严谨一点，将 polls/templates/polls/index.html 中的

```
<li><a href="{% url 'detail' question.id %}">{{ question.question_text }}</a></li>
```

修改为：

```
<li><a href="{% url 'polls:detail' question.id %}">{{ question.question_text }}</a></
↪li>
```

注意引用方法是冒号，不是圆点也不是斜杠！

1.4 Part 4：表单和类视图（略）

1.5 Part 5：测试

polls/tests.py 文件中：

```
import datetime
from django.utils import timezone
from django.test import TestCase
from .models import Question
```

(continues on next page)

(续上页)

```
class QuestionMethodTests(TestCase):
    def test_was_published_recently_with_future_question(self):
        """
        在将来发布的问卷应该返回 False
        """
        time = timezone.now() + datetime.timedelta(days=30)
        future_question = Question(pub_date=time)
        self.assertIs(future_question.was_published_recently(), False)
```

python manage.py test polls

其中都发生了些什么?:

python manage.py test polls 命令会查找投票应用中所有的测试程序

发现一个 django.test.TestCase 的子类

为测试创建一个专用的数据库

查找名字以 test 开头的测试方法

在 test_was_published_recently_with_future_question 方法中，创建一个 Question 实例，该实例的 pub_data 字段的值是 30 天后的未来日期。

然后利用 assertIs() 方法，它发现 was_published_recently() 返回了 True，而不是我们希望的 False。

最后，测试程序会通知我们哪个测试失败了，错误出现在哪一行。

1.6 Part 6：静态文件

每个 templates 包含一个与应用同名的子目录，每个 static 也包含一个与应用同名的子目录。

1.7 Part 7：自定义 admin

当表单含有大量字段的时候，你更多的是想将表单划分为一些字段的集合。

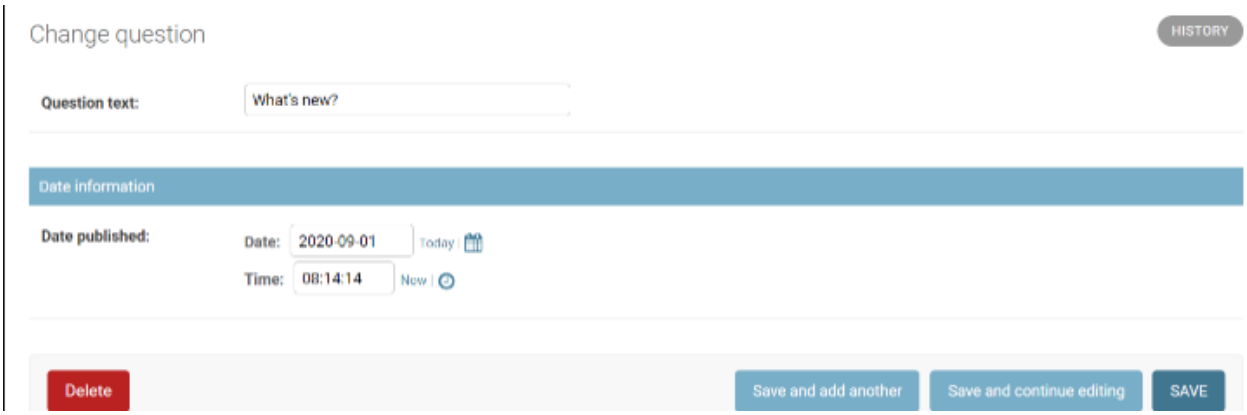
再次修改 polls/admin.py:

```
from django.contrib import admin

from .models import Question


class QuestionAdmin(admin.ModelAdmin):
    fieldsets = [
```

(continues on next page)

(续上页)

```
        (None,                {'fields': ['question_text']}),
        ('Date information', {'fields': ['pub_date']}),
    ]

admin.site.register(Question, QuestionAdmin)
```



1.8 其他，数据初始化方案（填充初始化数据）

低频操作。

方案 1，导出 sql

方案 2，manage.py 的 dumpdata 和 loaddata

CHAPTER 2

django 入门进阶 02 学习笔记 02

基于教程，刘江的博客教程 Django 教程:https://www.liujiangblog.com/course/django/87

第一章: 模型层

2.1 1.1 模型和字段

FileField

ImageField

FilePathField

UUIDField

2.2 1.2 关系类型字段

一，一对多

外键要定义在‘多’的一方！

```
parent_comment = models.ForeignKey('self', on_delete=models.CASCADE)
on_delete
limit_choices_to
related_name
```

(continues on next page)

(续上页)

```
related_query_name
to_field
db_constraint
swappable
```

二、多对多（ManyToManyField）

symmetrical

默认情况下，Django 中的多对多关系是对称的。

Django 认为，如果我是你的朋友，那么你也是我的朋友，这是一种对称关系，Django 不会为 Person 模型添加 person_set 属性用于反向关联。如果你不想使用这种对称关系，可以将 symmetrical 设置为 False，这将强制 Django 为反向关联添加描述符。

through_fields

Membership 模型中包含两个关联 Person 的外键，Django 无法确定到底使用哪个作为和 Group 关联的对象。所以，在这个例子中，必须显式的指定 through_fields 参数，用于定义关系。

through_fields 参数指定从中间表模型 Membership 中选择哪两个字段，作为关系连接字段。

db_table

ManyToManyField 多对多字段不支持 Django 内置的 validators 验证功能。

null 参数对 ManyToManyField 多对多字段无效！设置 null=True 毫无意义

三、一对一（OneToOneField）

2.3 1.3 字段的参数

null

该值为 True 时，Django 在数据库用 NULL 保存空值。默认值为 False。对于保存字符串类型数据的字段，请尽量避免将此参数设为 True，那样会导致两种‘没有数据’的情况，一种是 NULL，另一种是‘空字符串’。

blank

True 时，字段可以为空。默认 False。和 null 参数不同的是，null 是纯数据库层面的，而 blank 是验证相关的，它与表单验证是否允许输入框内为空有关，与数据库无关。所以要小心一个 null 为 False，blank 为 True 的字段接收到一个空值可能会出 bug 或异常。

```
choices
class Student(models.Model):
    FRESHMAN = 'FR'
    SOPHOMORE = 'SO'
    JUNIOR = 'JR'
```

(continues on next page)

(续上页)

```
    SENIOR = 'SR'
    YEAR_IN_SCHOOL_CHOICES = (
        (FRESHMAN, 'Freshman'),
        (SOPHOMORE, 'Sophomore'),
        (JUNIOR, 'Junior'),
        (SENIOR, 'Senior'),
    )
    year_in_school = models.CharField(
        max_length=2,
        choices=YEAR_IN_SCHOOL_CHOICES,
        default=FRESHMAN,
    )
class Person(models.Model):
    SHIRT_SIZES = (
    ('S', 'Small'),
    ('M', 'Medium'),
    ('L', 'Large'),
    )
    name = models.CharField(max_length=60)
    shirt_size = models.CharField(max_length=1, choices=SHIRT_SIZES)
p = Person(name="Fred Flintstone", shirt_size="L")
p.shirt_size
p.get_shirt_size_display()
db_column
db_index
```

该参数接收布尔值。如果为 True，数据库将为该字段创建索引。

```
db_tablespace
default
```

editable

如果设为 False，那么当前字段将不会在 admin 后台或者其它的 ModelForm 表单中显示，同时还会被模型验证功能跳过。参数默认值为 True。

error_messages

用于自定义错误信息。参数接收字典类型的值。字典的键可以是 null、blank、invalid、invalid_choice、unique 和 unique_for_date 其中的一个。

help_text

额外显示在表单部件上的帮助文本。使用时请注意转义为纯文本，防止脚本攻击。

primary_key

如果你没有给模型的任何字段设置这个参数为 True，Django 将自动创建一个 AutoField 自增字段，名为‘id’，并设置为主键。也就是 id = models.AutoField(primary_key=True)。

如果你为某个字段设置了 primary_key=True，则当前字段变为主键，并关闭 Django 自动生成 id 主键的功能。

另外，主键字段不可修改，如果你给某个对象的主键赋个新值实际上是创建一个新对象，并不会修改原来的对象。

unique

设为 True 时，在整个数据表内该字段的数据不可重复。

注意：对于 ManyToManyField 和 OneToOneField 关系类型，该参数无效。

unique_for_date

日期唯一。可能不太好理解。举个栗子，如果你有一个名叫 title 的字段，并设置了参数 unique_for_date=”pub_date”，那么 Django 将不允许有两个模型对象具备同样的 title 和 pub_date。有点类似联合约束。

unique_for_month

同上，只是月份唯一。

unique_for_year

同上，只是年份唯一。

verbose_name

为字段设置一个人类可读，更加直观的别名。

对于每一个字段类型，除了 ForeignKey、ManyToManyField 和 OneToOneField 这三个特殊的关系类型，其第一可选位置参数都是 verbose_name。如果没指定这个参数，Django 会利用字段的属性名自动创建它，并将下划线转换为空格。

下面这个例子的 verbose name 是”person’s first name”：

first_name = models.CharField(“person’s first name”, max_length=30)

下面这个例子的 verbose name 是”first name”：

first_name = models.CharField(max_length=30)

对于外键、多对多和一对一字字段，由于第一个参数需要用来指定关联的模型，因此必须用关键字参数 verbose_name 来明确指定。如下：

2.4 1.4 多对多中间表详解

2.5 1.5 模型的元数据 Meta

abstract

app_label

base_manager_name

db_table

db_tablespace

default_manager_name

default_related_name

get_latest_by

managed

order_with_respect_to

ordering

permissions

default_permissions

proxy

required_db_features

required_db_vendor

select_on_save

indexes

unique_together

verbose_name

verbose_name_plural

label

label_lower

2.6 1.6 模型的继承

抽象基类, 多表继承, 代理模型

抽象基类中的 abstract=True 这个元数据不会被继承。也就是说如果想让一个抽象基类的子模型，同样成为一个抽象基类，那你必须显式的在该子模型的 Meta 中同样声明一个 abstract = True；

有一些元数据对抽象基类无效，比如 db_table，首先是抽象基类本身不会创建数据表，其次它的所有子类也不会按照这个元数据来设置表名。

警惕 related_name 和 related_query_name 参数

```
class Base(models.Model):
    m2m = models.ManyToManyField(
    OtherModel,
    related_name="%(app_label)s_%(class)s_related",
    related_query_name="%(app_label)s_%(class)ss",
    )

    class Meta:
        abstract = True
```

如果一个 Place 对象同时也是一个 Restaurant 对象，你可以使用小写的子类名，在父类中访问它，

Meta 和多表继承

由于父类和子类都在数据库内有物理存在的表，父类的 Meta 类会对子类造成不确定的影响，因此，Django 在这种情况下关闭了子类继承父类的 Meta 功能。这一点和抽象基类的继承方式有所不同。

但是，还有两个 Meta 元数据特殊一点，那就是 ordering 和 get_latest_by，这两个参数是会被继承的。因此，如果在多表继承中，你不想让你的子类继承父类的上面两种参数，就必须在子类中显示的指出或重写

多表继承和反向关联

三、代理模型

声明一个代理模型只需要将 Meta 中 proxy 的值设为 True。

四、多重继承

Django 的模型体系支持多重继承，就像 Python 一样。如果多个父类都含有 Meta 类，则只有第一个父类的会被使用，剩下的会忽略掉。一般情况，能不要多重继承就不要，尽量让继承关系简单和直接，避免不必要的混乱和复杂。

请注意，继承同时含有相同 id 主键字段的类将抛出异常。为了解决这个问题，你可以在基类模型中显式的使用 AutoField 字段。或者使用一个共同的祖先来持有 AutoField 字段，并在直接的父类里通过一个 OneToOne 字段保持与祖先的关系，如下所示：

```
class Piece(models.Model):
    pass

class Article(Piece):
    article_piece = models.OneToOneField(Piece, on_delete=models.CASCADE, parent_
↪link=True)
    ...

class Book(Piece):
    book_piece = models.OneToOneField(Piece, on_delete=models.CASCADE, parent_
↪link=True)
    ...
```

(continues on next page)

(续上页)

```
class BookReview(Book, Article):
    pass
```

警告在 Python 语言层面，子类可以拥有和父类相同的属性名，这样会造成覆盖现象。但是对于 Django，如果继承的是一个非抽象基类，那么子类与父类之间不可以有相同的字段名！

比如下面是不行的！

```
class A(models.Model):
    name = models.CharField(max_length=30)

class B(A):
    name = models.CharField(max_length=30)
```

2.7 1.7 用包来组织模型

CHAPTER 3

django 入门进阶 03 学习笔记 03

第一章: 模型层

3.1 1.8 查询操作

一、创建对象

```
>>> from blog.models import Blog
>>> b = Blog(name='Beatles Blog', tagline='All the latest Beatles news.')
>>> b.save()

b = Blog.objects.create(name='Beatles Blog', tagline='All the latest Beatles news.')
```

二、保存对象

1. 保存外键和多对多字段

```
>>> entry = Entry.objects.get(pk=1)
>>> cheese_blog = Blog.objects.get(name="Cheddar Talk")
>>> entry.blog = cheese_blog
>>> entry.save()
```

多对多字段的保存稍微有点区别，需要调用一个 add() 方法，而不是直接给属性赋值，但它不需要调用 save 方法。如下例所示：

```
>>> from blog.models import Author
>>> joe = Author.objects.create(name="Joe")
>>> entry.authors.add(joe)
```

同时添加多个对象到多对多的字段

```
>>> entry.authors.add(john, paul, george, ringo)
```

三、检索对象

通过模型的 Manager 获得 QuerySet，每个模型至少具有一个 Manager，默认情况下，它被称作 objects，可以通过模型类直接调用它，但不能通过模型类的实例调用它，以此实现“表级别”操作和“记录级别”操作的强制分离。

1. 检索所有对象
2. 过滤对象

```
filter(**kwargs)：返回一个根据指定参数查询出来的 QuerySet
exclude(**kwargs)：返回除了根据指定参数查询出来结果的 QuerySet
```

链式过滤

被过滤的 QuerySets 都是唯一的

```
>>> q1 = Entry.objects.filter(headline__startswith="What")
>>> q2 = q1.exclude(pub_date__gte=datetime.date.today())
>>> q3 = q1.filter(pub_date__gte=datetime.date.today())
```

例子中的 q2 和 q3 虽然由 q1 得来，是 q1 的子集，但是都是独立自主存在的。同样 q1 也不会受到 q2 和 q3 的影响。

1. 检索单一对象

注意：使用 get() 方法和使用 filter() 方法然后通过 [0] 的方式分片，有着不同的地方。看似两者都是获取单一对象。但是，如果在查询时没有匹配到对象，那么 get() 方法将抛出 DoesNotExist 异常。

在使用 get() 方法查询时，如果结果超过 1 个，则会抛出 MultipleObjectsReturned 异常

1. 其它 QuerySet 方法
2. QuerySet 使用限制

注意：不支持负索引！例如 Entry.objects.all()[-1] 是不允许的

通常情况，切片操作会返回一个新的 QuerySet，并且不会被立刻执行。但是有一个例外，那就是指定步长的时候，查询操作会立刻在数据库内执行，如下：

```
>>> Entry.objects.all()[:10:2]
```

若要获取单一的对象而不是一个列表（例如，SELECT foo FROM bar LIMIT 1），可以简单地使用索引而不是切片。例如，下面的语句返回数据库中根据标题排序后的第一条 Entry：

```
>>> Entry.objects.order_by('headline')[0]
```

它相当于：

```
>>> Entry.objects.order_by('headline')[0:1].get()
```

注意：如果没有匹配到对象，那么第一种方法会抛出 IndexError 异常，而第二种方式会抛出 DoesNotExist 异常。

也就是说在使用 get 和切片的时候，要注意查询结果的元素个数。

1. 字段查询

字段查询其实就是 filter()、exclude() 和 get() 等方法的关键字参数。其基本格式是：field__lookuptype=value，注意其中是双下划线。例如：

```
>>> Entry.objects.filter(pub_date__lte='2006-01-01')
#   相当于：
SELECT * FROM blog_entry WHERE pub_date <= '2006-01-01';
```

有一个例外，那就是 ForeignKey 字段，你可以为其添加一个“_id”后缀（单下划线）。这种情况下键值是外键模型的主键原生值。例如：

```
>>> Entry.objects.filter(blog_id=4)
```

exact：默认类型。

iexact：不区分大小写。

contains：表示包含的意思！大小写敏感！

icontains：contains 的大小写不敏感模式。

startswith 和 endswith : 以什么开头和以什么结尾。大小写敏感！

istartswith 和 iendswith 是不区分大小写的模式。

1. 跨越关系查询

Django 提供了强大并且直观的方式解决跨越关联的查询，它在后台自动执行包含 JOIN 的 SQL 语句。要跨越某个关联，只需使用关联的模型字段名称，并使用双下划线分隔，直至你想要的字段（可以链式跨越，无限跨度）。

跨越多值的关系查询

最基本的 filter 和 exclude 的关键字参数只有一个，这种情况很好理解。但是当关键字参数有多个，且是跨越外键或者多对多的情况下，那么就比较复杂，让人迷惑了。我们看下面的例子：

```
Blog.objects.filter(entry__headline__contains='Lennon', entry__pub_date__year=2008)
```

这是一个跨外键、两个过滤参数的查询。此时我们理解两个参数之间属于-与“and”的关系，也就是说，过滤出来的 BLog 对象对应的 entry 对象必须同时满足上面两个条件。这点很好理解。也就是说上面要求至少有一个 entry 同时满足两个条件。

但是，看下面的用法：

```
Blog.objects.filter(entry__headline__contains='Lennon').filter(entry__pub_date__
↪year=2008)
```

把两个参数拆开，放在两个 filter 调用里面，按照我们前面说过的链式过滤，这个结果应该和上面的例子一样。可实际上，它不一样，Django 在这种情况下，将两个 filter 之间的关系设计为-或“or”，这真是让人头疼。

多对多关系下的多值查询和外键 foreignkey 的情况一样。

但是，更头疼的来了，exclude 的策略设计的又和 filter 不一样！

```
Blog.objects.exclude(entry__headline__contains='Lennon',entry__pub_date__year=2008,)
```

这会排除 headline 中包含“Lennon”的 Entry 和在 2008 年发布的 Entry，中间是一个-和“or”的关系！

那么要排除同时满足上面两个条件的对象，该怎么办呢？看下面：

```
Blog.objects.exclude(
entry=Entry.objects.filter(
    headline__contains='Lennon',
    pub_date__year=2008,
),
)
```

（有没有很坑爹的感觉？所以，建议在碰到跨关系的多值查询时，尽量使用 Q 查询）

1. 使用 F 表达式引用模型的字段
2. 主键的快捷查询方式：pk

pk 就是 primary key 的缩写。通常情况下，一个模型的主键为“id”，所以下面三个语句的效果一样：

```
>>> Blog.objects.get(id__exact=14) # Explicit form
>>> Blog.objects.get(id=14) # __exact is implied
>>> Blog.objects.get(pk=14) # pk implies id__exact
```

1. 在 LIKE 语句中转义百分符号和下划线

在原生 SQL 语句中% 符号有特殊的作用。Django 帮你自动转义了百分符号和下划线，你可以和普通字符一样使用它们，如下所示：

```
>>> Entry.objects.filter(headline__contains='%')
# 它和下面的一样
# SELECT ... WHERE headline LIKE '%\%%';
```

1. 缓存与查询集

要想高效的利用查询结果，降低数据库负载，你必须善于利用缓存。看下面的例子，这会造成 2 次实际的数据库操作，加倍数据库的负载，同时由于时间差的问题，可能在两次操作之间数据被删除或修改或添加，导致脏数据的问题：

```
>>> print([e.headline for e in Entry.objects.all()])
>>> print([e.pub_date for e in Entry.objects.all()])
```

为了避免上面的问题，好的使用方式如下，这只产生一次实际的查询操作，并且保持了数据的一致性：

```
>>> queryset = Entry.objects.all()
>>> print([p.headline for p in queryset]) # 提交查询
>>> print([p.pub_date for p in queryset]) # 重用查询缓存
```

何时不会被缓存

有一些操作不会缓存 QuerySet，例如切片和索引。这就导致这些操作没有缓存可用，每次都会执行实际的数据库查询操作。例如：

```
>>> queryset = Entry.objects.all()
>>> print(queryset[5]) # 查询数据库
>>> print(queryset[5]) # 再次查询数据库
```

但是，如果已经遍历过整个 QuerySet，那么就相当于缓存过，后续的操作则会使用缓存，例如：

```
>>> queryset = Entry.objects.all()
>>> [entry for entry in queryset] # 查询数据库
>>> print(queryset[5]) # 使用缓存
>>> print(queryset[5]) # 使用缓存
```

下面的这些操作都将遍历 QuerySet 并建立缓存：

```
>>> [entry for entry in queryset]
>>> bool(queryset)
>>> entry in queryset
>>> list(queryset)
```

注意：简单的打印 QuerySet 并不会建立缓存，因为 __repr__() 调用只返回全部查询集的一个切片。

```
Poll.objects.get(
Q(question__startswith='Who'),
```

(continues on next page)

(续上页)

```
Q(pub_date=date(2005, 5, 2)) | Q(pub_date=date(2005, 5, 6))
)
# 它相当于
# SELECT * from polls WHERE question LIKE 'Who%'
AND (pub_date = '2005-05-02' OR pub_date = '2005-05-06')
```

当关键字参数和 Q 对象组合使用时，Q 对象必须放在前面，如下例子：

```
Poll.objects.get(
Q(pub_date=date(2005, 5, 2)) | Q(pub_date=date(2005, 5, 6)),question__startswith='Who
↪',)
```

如果关键字参数放在 Q 对象的前面，则会报错。

五、比较对象

要比较两个模型实例，只需要使用 python 提供的双等号比较符就可以了。在后台，其实比较的是两个实例的主键的值。下面两种方法是等同的：

```
>>> some_entry == other_entry
>>> some_entry.id == other_entry.id
```

如果模型的主键不叫做“id”也没关系，后台总是会使用正确的主键名字进行比较，例如，如果一个模型的主键的名字是“name”，那么下面是相等的：

```
>>> some_obj == other_obj
>>> some_obj.name == other_obj.name
```

六、删除对象

删除对象使用的是对象的 delete() 方法。该方法将返回被删除对象的总数量和一个字典，字典包含了每种被删除对象的类型和该类型的数量。如下所示：

```
>>> e.delete()
(1, {'weblog.Entry': 1})
```

也可以批量删除。每个 QuerySet 都有一个 delete() 方法，它能删除该 QuerySet 的所有成员。例如：

```
>>> Entry.objects.filter(pub_date__year=2005).delete()
(5, {'webapp.Entry': 5})
```

当 Django 删除一个对象时，它默认使用 SQL 的 ON DELETE CASCADE 约束，也就是说，任何有外键指向要删除对象的对象将一起被删除。例如：

```
b = Blog.objects.get(pk=1)
# 下面的动作将删除该条 Blog 和所有的它关联的 Entry 对象
b.delete()
```

这种级联的行为可以通过的 ForeignKey 的 on_delete 参数自定义。

注意，delete() 是唯一没有在管理器上暴露出来的方法。这是刻意设计的一个安全机制，用来防止你意外地请求类似 Entry.objects.delete() 的动作，而不慎删除了所有的条目。如果你确实想删除所有的对象，你必须明确地请求一个完全的查询集，像下面这样：

```
Entry.objects.all().delete()
```

七、复制模型实例

```
blog = Blog(name='My blog', tagline='Blogging is easy')
blog.save() # blog.pk == 1
#
blog.pk = None
blog.save() # blog.pk == 2
```

八、批量更新对象

```
# 更新所有 2007 年发布的 entry 的 headline
Entry.objects.filter(pub_date__year=2007).update(headline='Everything is the same')
```

九、关系的对象

1. 一对多（外键）

正向查询:

反向查询:

使用自定义的反向管理器:

处理关联对象的其它方法:

1. 多对多
2. 一对一
3. 反向关联是如何实现的?
4. 通过关联对象进行查询

十、使用原生 SQL 语句

3.2 查询集 API

一、QuerySet 何时被提交

迭代

QuerySet 是可迭代的，在首次迭代查询集时执行实际的数据库查询

切片：如果使用切片的”step“参数，Django 将执行数据库查询并返回一个列表。

Pickling/缓存

repr()

len()：当你对 QuerySet 调用 len() 时，将提交数据库操作。

list()：对 QuerySet 调用 list() 将强制提交操作 entry_list = list(Entry.objects.all())

bool()

二、QuerySet

QuerySet 类具有两个公有属性用于内省：

ordered：如果 QuerySet 是排好序的则为 True，否则为 False。

db：如果现在执行，则返回使用的数据库。

三、返回新 QuerySets 的 API

方法名解释

filter() 过滤查询对象。

exclude() 排除满足条件的对象

annotate() 使用聚合函数

order_by() 对查询集进行排序

reverse() 反向排序

distinct() 对查询集去重

values() 返回包含对象具体值的字典的 QuerySet

values_list() 与 values() 类似，只是返回的是元组而不是字典。

none() 创建空的查询集

all() 获取所有的对象

select_related() 附带查询关联对象

3.3 不返回 QuerySets 的 API

get() 获取单个对象

create() 创建对象，无需 save()

get_or_create() 查询对象，如果没有找到就新建对象

update_or_create() 更新对象，如果没有找到就创建对象

count() 统计对象的个数

latest() 获取最近的对象

earliest() 获取最早的对象

first() 获取第一个对象

last() 获取最后一个对象

aggregate() 聚合操作

exists() 判断 queryset 中是否有对象

update() 批量更新对象

delete() 批量删除对象

CHAPTER 4

django 入门进阶 04 学习笔记 04

第二章　视图层

4.1 URL 路由基础

四、path 转换器

默认情况下，Django 内置下面的路径转换器：

str：匹配任何非空字符串，但不含斜杠/，如果你没有专门指定转换器，那么这个是默认使用的；

int：匹配 0 和正整数，返回一个 int 类型

slug：可理解为注释、后缀、附属等概念，是 url 拖在最后的一部分解释性字符。该转换器匹配任何 ASCII 字符以及连接符和下划线，比如 building-your-1st-django-site；

uuid：匹配一个 uuid 格式的对象。为了防止冲突，规定必须使用破折号，所有字母必须小写，例如 075194d3-6885-417e-a8a8-6c931e272f00。返回一个 UUID 对象；

path：匹配任何非空字符串，重点是可以包含路径分隔符’/‘。这个转换器可以帮助你匹配整个 url 而不是一段一段的 url 字符串。要区分 path 转换器和 path() 方法。

五、自定义 path 转换器

其实就是写一个类，并包含下面的成员和属性：

类属性 regex：一个字符串形式的正则表达式属性；

to_python(self, value) 方法：一个用来将匹配到的字符串转换为你想要的那个数据类型，并传递给视图函数。如果转换失败，它必须弹出 ValueError 异常；

to_url(self, value) 方法：将 Python 数据类型转换为一段 url 的方法，上面方法的反向操作。

例如，新建一个 converters.py 文件，与 urlconf 同目录，写个下面的类：

```
class FourDigitYearConverter:
    regex = '[0-9]{4}'

    def to_python(self, value):
        return int(value)

    def to_url(self, value):
        return '%04d' % value
```

写完类后，在 URLconf 中注册，并使用它，如下所示，注册了一个 yyyy：

```
from django.urls import register_converter, path
from . import converters, views
register_converter(converters.FourDigitYearConverter, 'yyyy')

urlpatterns = [
    path('articles/2003/', views.special_case_2003),
    path('articles/<yyyy:year>/', views.year_archive),
    ...
]
```

八、指定视图参数的默认值

```
urlpatterns = [
    path('blog/', views.page),
    path('blog/page<int:num>/', views.page),
]

# View (in blog/views.py)

def page(request, num=1):
    # Output the appropriate page of blog entries, according to num.
    ...
```

九、自定义错误页面

```
from app import views

urlpatterns = [
```

(continues on next page)

(续上页)

```
    path('admin/', admin.site.urls),
]

# 增加的条目
handler400 = views.bad_request
handler403 = views.permission_denied
handler404 = views.page_not_found
handler500 = views.error
```

app/views.py 文件中增加四个处理视图：

```
def bad_request(request):
    return render(request, '400.html')


def permission_denied(request):
    return render(request, '403.html')


def page_not_found(request):
    return render(request, '404.html')


def error(request):
    return render(request, '500.html')
```

4.2 路由转发

一、路由转发

```
urlpatterns = [
    path('<page_slug>-<page_id>/', include([
        path('history/', views.history),
        path('edit/', views.edit),
        path('discuss/', views.discuss),
        path('permissions/', views.permissions),
    ])),
]
```

三、向视图传递额外的参数

```
urlpatterns = [
    path('blog/<int:year>/', views.year_archive, {'foo': 'bar'}),
]
```

四、传递额外的参数给 include()

```
urlpatterns = [
    path('blog/', include('inner'), {'blog_id': 3}),
]
```

4.3 反向解析和命名空间

一、反向解析 URL

```
<a href="{% url 'news-year-archive' 2012 %}">2012 Archive</a>
return HttpResponseRedirect(reverse('news-year-archive', args=(year,)))
```

二、URL 命名空间

```
urlpatterns = [
    path('author-polls/', include('polls.urls', namespace='author-polls')),
    path('publisher-polls/', include('polls.urls', namespace='publisher-polls')),
]
```

三、URL 命名空间和 include 的 URLconf

```
app_name = 'polls'
urlpatterns = [
urlpatterns = [
    path('polls/', include('polls.urls')),
]

polls_patterns = ([
    path('', views.IndexView.as_view(), name='index'),
    path('<int:pk>/', views.DetailView.as_view(), name='detail'),
], 'polls')
urlpatterns = [
    path('polls/', include(polls_patterns)),
]
```

4.4 视图函数及快捷方式

二、返回错误

```
my_object = get_object_or_404(MyModel, pk=1)
5. get_list_or_404()
```

CHAPTER 5

django 入门进阶 05 快捷复习手册

5.1 建立项目

django startproject projectname

5.2 启动服务

python manage.py runserver 9999# 自己指定端口

python manage.py # 使用默认 8000 端口

5.3 新增应用

python manage.py startapp blog#blog 为应用名称

添加应用名到 settings.py 中的 INSTALLED_APPS 里

```
INSTALLED_APPS = [
    ...
    'blog',
]
```

5.4 初始化数据表

执行 python manage.py makemigrations app 名（可选）

再执行 python manage.py migrate

查看 Django 会自动在 app/migreations/目录下生成移植文件

执行 python manage.py sqlmigrate appname 文件 id 查看 sql 语句, 比如 python manage.py sqlmigrate blog 0001

5.5 admin 后台控制台生成

python manage.py createsuperuser

5.6 数据查询

```
article = models.Article.objects.get(pk=1)
return render(request, 'blog/index.html', {'article': article})

编辑 admin.py:
admin.site.register(models.Article)
def __str__(self):(python 3.X)
        return self.title
```

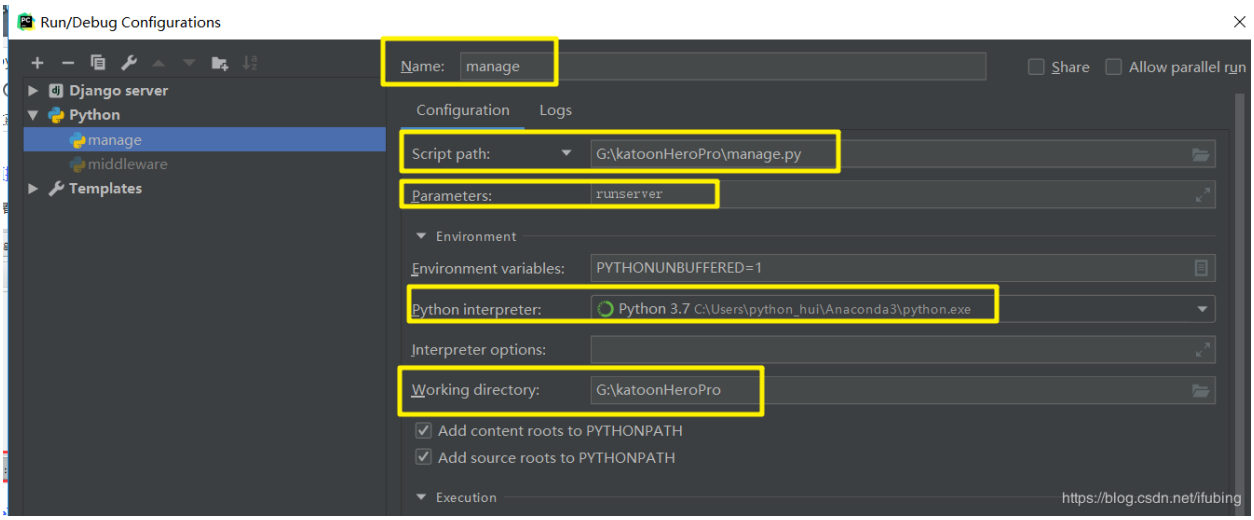
5.7 模板语言

```
<body>
<h1>{{article.title}}</h1>
<h3>{{article.content}}</h3>
</body>
  <a href="{% url 'blog:article_page' article.id%}">{{article.title}}</a>
```

5.8 url 适配

```
url(r'^article/(?P<article_id>[0-9]+)$', views.article_page),
```

5.9 debug 调试




```
{% for item in List %}
    {{ item }},
{% endfor %}
{% for item in List %}
    {{ item }}{% if not forloop.last %},{% endif %}
{% endfor %}
```

在 for 循环中还有很多有用的东西，如下：

```
变量          描述
forloop.counter          索引从 1 开始算
forloop.counter0          索引从 0 开始算
forloop.revcounter          索引从最大长度到 1
forloop.revcounter0          索引从最大长度到 0
forloop.first          当遍历的元素为第一项时为真
forloop.last          当遍历的元素为最后一项时为真
forloop.parentloop
用在嵌套的 for 循环中，
```

获取上一层 for 循环的 forloop

```
<ul>
{% for athlete in athlete_list %}
    <li>{{ athlete.name }}</li>
{% empty %}
    <li> 抱歉，列表为空</li>
{% endfor %}
</ul>
```

(continues on next page)

(续上页)

```
{% if var >= 90 %}
成绩优秀，自强学堂你没少去吧！学得不错
{% elif var >= 80 %}
成绩良好
{% elif var >= 70 %}
成绩一般
{% elif var >= 60 %}
需要努力
{% else %}
不及格啊，大哥！多去自强学堂学习啊！
{% endif %}
```

查看 Django queryset 执行的 SQL

```
print str(Author.objects.all().query)
```

比如我们要获取作者的 name 和 qq

```
authors = Author.objects.values_list('name', 'qq')
```

比如我们要获取作者的 name 和 qq

```
Author.objects.values('name', 'qq')
Out[13]: <QuerySet [{'qq': u'336643078', 'name': u'WeizhongTu'}, {'qq': u'915792575',
↪'name': u'twz915'}, {'qq': u'353506297', 'name': u'wangdachui'}, {'qq': u'004466315
↪', 'name': u'xiaoming'}]>
```

5.10 参考文献

https://blog.csdn.net/sinat_29214327/article/details/85226171

https://code.ziqiangxuetang.com/django/django-send-email.html

https://www.jianshu.com/p/116057746c64

CHAPTER 6

django 入门进阶 06 静态文件和模板

6.1 静态文件和模板

静态文件:css,js,image, 如果作为纯粹的 web 应用来看，静态文件的响应并不属于 web 应用范畴，**因为静态文件不涉及业务逻辑，也不需开发业务代码**。但几乎 100% 的 web 应用都支持对静态文件的直接访问。为何? 主要是 web 应用基本上必然依赖 css,js,img 等静态资源，我们不可能固执的开发一个纯粹的 web 应用，**只支持从 url 里进入视图函数，也只能从视图函数返回内容**（简单来说所有请求路径都必须体现在 url_route 和 view 视图中）。而要求用户独立开启静态资源的请求处理服务。所以，先把静态资源服务和包含业务逻辑的 web 应用独立认识，二者并不相同，但强相关。**其本身是独立于应用的**

模板: 在代码中 response 渲染中使用的，可以看做 view 视图的组成部分. 所以没有独立 url 配置，**依赖 view 视图存在**，属于应用一部分，包含了业务逻辑（需要渲染），这一点也可以从配置上看出来。

6.2 django 配置静态文件

STATIC_ROOT

所有的静态文静聚合的目录,

STATIC_ROOT 要写成绝对地址，在这里，比如我的项目 mysite 是/home/mysite/ 那么 STATIC_ROOT 为 /home/mysite/collect_static/

```
python manage.py collectstatic # 把所有的 static 文件都复制到 STATIC_ROOT 文件夹下
```

可见,STATIC_ROOT 是在部署的时候才发挥作用。简单来说，开发时可能按应用分散开发，**所以 css,js 等都是按应用分散到各自应用目录下**，但是在部署时，**不大可能在前置的 web 服务器中配置多个分散的应用目录**（而且也会带来高耦合问题）。所以需要“整合”到一个目录，然后 nginx 的服务器指向此目录即可。

STATICFILES_DIRS

开发时静态资源路径，前面说过部署时会使用 STATIC_ROOT 作为静态资源路径。但开发时一般不会有独立的静态资源服务器，所以允许分散性配置，同时，这个也是生成 STATIC_ROOT 里文件的“源路径”。

STATICFILES_DIRS 里文件夹又份两种，app 应用独立占用：一种就是在每个 app 里面新建一个 static 文件夹, 将静态文件放到里面, 在加载静态文件时, 比如要在模板中用到静态文件,django 会自动在每个 app 里面搜索 static 文件夹 (所以, 不要把文件夹的名字写错哦, 否则 django 就找不到你的文件夹了)

项目（多个或所有应用）共享：就是在所有的 app 文件外面, 建立一个公共的文件夹, 因为有些静态文件不是某个 app 独有的, 那么就可以把它放到一个公共文件夹里面, 方便管理 (注意, 建立一个公共的静态文件的文件夹只是一种易于管理的做法, 但是不是必须的,app 是可以跨 app 应用静态文件的, 因为最后所有的静态文件都会在 STATIC_ROOT 里面存在) 那现在的问题是如何让 django 知道你把一些静态文件放到 app 以外的公共文件夹中呢, 那就需要配置 STATICFILES_DIRS 了

```
STATICFILES_DIRS = ( os.path.join(BASE_DIR, 'common_static'), )
```

STATICFILES_DIRS 告诉 django, 首先到 STATICFILES_DIRS 里面寻找静态文件, 其次再到各个 app 的 static 文件夹里面找 (注意, django 查找静态文件是惰性查找, 查找到第一个, 就停止查找了)

STATIC_URL

那么到此为止, 静态文件的机制就可以运作了, 但是有一个问题, 我能不能通过 url 直接访问我在项目中的静态文件呢, 答案肯定是啦, 但是, 注意, 你是在浏览器是访问, 你不可能输入你的静态文件的本地绝对地址吧, 比如我的一种图片的本地地址为 /home/mysite/common_static/myapp/photo.png 那么别人不可能在浏览器上直接输入: http://192.168.1.2:8000/home/mysite/common_static/myapp/photo.png 这样子, 浏览器会报错, 没有该页面那么 django 是如何让浏览器也可以访问服务器上的静态文件呢, 前面已经说了, 直接访问服务器本地的地址是不行的, 那就需要一个映射,django 利用 STATIC_URL 来让浏览器可以直接访问静态文件, 比如:

```
STATIC_URL = '/static/'
```

那么可以在浏览器上输入: http://192.168.1.2:8000/static/common_static/myapp/photo.png 那么就相当与访问/home/mysite/common_static/myap/photo.png

开发环境下静态文件都是通过 Django 自带的 web 服务器来处理的（这样会更方面）。如果把 DEBUG 设置成 False，那么 Django 自带的 web 服务器自然不处理静态文件了，静态文件都交给 nginx，apache 来处理吧（这样会更高效）。

另外，Django 提供了一个 findstatic 命令来查找指定的静态文件所在的目录，例如：

```
D:\TestDjango>python manage.py findstatic Chrome.jpg ('D:/TestDjango/TestDjango/
↪templates',)
```

6.3 django 配置模板

模板相关配置

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR, 'templates')], # 公共共享文件夹
        'APP_DIRS': True,# 检索 app 里面的 templates 文件夹
    }
]
```

模板相关配置只有 2 项，实际这 2 项目，如果**和静态文件比对起来，只对应了 1 项**，就是 STATICFILES_DIRS

所以:TEMPLATES（原始）并不支持从 url 直接 respones，而是需要经过 url_route,view 处理，view_response(html)，

所以**模板不需要独立路由配置**，其路由有具体调用其的 view 决定，也就没有 STATIC_URL 类似的配置。当然也不存在静态资源统一保存的需求，也就不需要 STATIC_ROOT 类似的配置。

Django 模板查找机制

Django 查找模板的过程是在每个 app 的 templates 文件夹中找（而不只是当前 app 中的代码只在当前的 app 的 templates 文件夹中找）。

各个 app 的 templates 形成一个文件夹列表，Django 遍历这个列表，一个个文件夹进行查找，当在某一个文件夹找到的时候就停止，所有的都遍历完了还找不到指定的模板的时候就是 Template Not Found （过程类似于 Python 找包）。这样设计有利当然也有弊，有利是的地方是一个 app 可以用另一个 app 的模板文件，弊是有可能会找错了。所以我们使用的时候在 templates 中建立一个 app 同名的文件夹，这样就好了。

这就需要把每个 **app 中的 templates 文件夹中再建一个 app 的名称**，仅和该 app 相关的模板放在 app/templates/app/ 目录下面

不使用 view 直接跳转

```
urlpatterns = [ path('about/', TemplateView.as_view(template_name="about.html")), ]
```

6.4 参考

Django 模板:https://code.ziqiangxuetang.com/django/django-template.html

django—不使用 view，直接从 Url 转到 html：https://blog.csdn.net/weixin_30325487/article/details/97544472

Django 关于访问静态文件总结：https://blog.csdn.net/WaitForFree/article/details/39815507

django 静态文件之配置说明 (Django 中 STATIC_URL、STATIC_ROOT、STATICFILES_DIRS 区别关系)：https://blog.csdn.net/alxandral_brother/article/details/52202270

CHAPTER 7

django 入门进阶 07 用户模块与权限系统

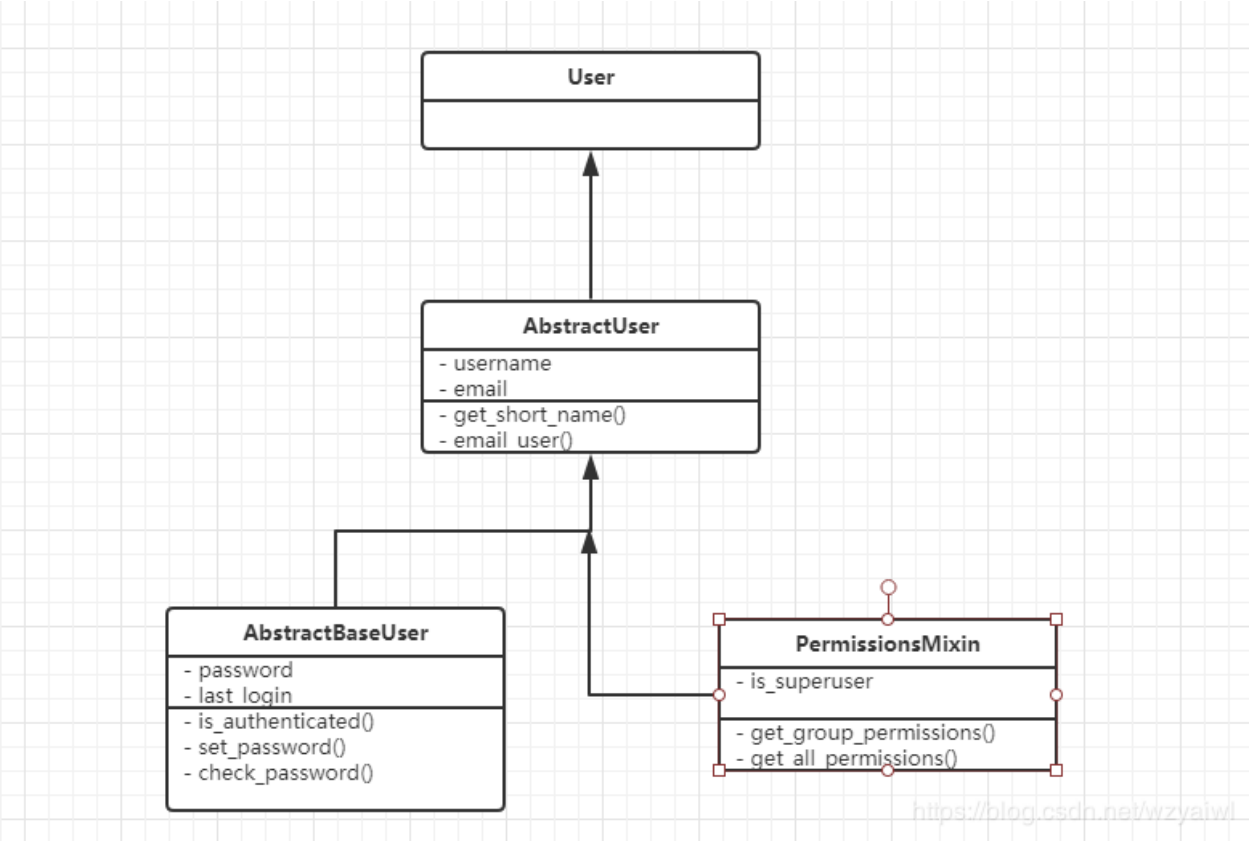
Django 默认提供了用户权限管理模块 auth

```
user 表，User 是 auth 模块中维护用户信息的表，在数据库中该表被命名为 auth_user. 该表继承自
Abstractuser.
group 表，定义用户组模型，该表只包含一个 name 字段和一个 permissions(权限) 多对多关系字段，在数据库
中被命名为 auth_group.
Permission， 权限表，提供表级别的权限控制，可以检查用户是否对某个表拥有增 (add)，改 (change)，删
(delete) 权限。
```

从数据库生成的表来看，这三张表实现了两两多对多的关联，其中用户和组生成的第三张表是 auth_user_groups,group 和 permission 生成的第三张表是 auth_group_permissions, 用户和权限生成的第三张表是 auth_user_user_permissions。这里我们主要介绍下 user 表。

7.1 User 表

在 auth 中 user 继承自 AbstractUser，其中 AbstractUser 又继承自 AbstractBaseUser 和 PermissionsMixin，其中 AbstractBaseUser 只保存了密码和登陆时间，PermissionsMixin 提供了权限先关的字段，比如：is_superuser 和组合权限表之间的关联。



User 对象的字段

```
password: Django 默认保存是加密后的密码，无法直接看到明文密码
last_login: 上一次登陆时间
is_superuser: 是否是超级管理员，是为 1，否为 0
username: 用户名
first_name
last_name
email: 邮箱
is_staff: 用户是否拥有网站的管理权限
is_active: 是否允许用户登录，设置为 False，可以在不删除用户的前提下禁止用户登录。
data_joined: 账户创建日期
groups:  与组多对多关联的字段
user_permissions:  与权限关联的多对多字段，也就是是说明了为什么第三张表表为 auth_user_user_
↪permissions. 表名 (user) + 字段名 (user_permissions)
```

类属性

```
is_authenticated: 判断是否被认证，即是否登陆
is_anonymous:  是否为匿名用户
username_validator: 指向用于验证用户名的验证实例，默认是 validators.UnicodeUsernameValidator
```

类方法

```
get_username():  获取用户名
get_full_name():  获取全名，即 first_name+ 空格 +last_name
get_short_name()：获取 first_name
set_password(raw_password)：  设置密码，如果 raw_password 是 None，则密码将被设置为不可用的密码，
就像使用了 set_unusable_password() 一样。
check_password(raw_password)：  检查密码是否正确。
set_unusable_password()：将用户标记为未设置密码，即密码为 None
has_usable_password()：返回该用户是否未设置密码
get_group_permissions()：  获取这个用户所在组中所具有的的全部权限。
has_perm()：  判断一个用户是否具有某个权限。
has_perms(perm_list)：  判断用户对一个权限列表是否具有权限。
has_module_perms(package_name)：  判断对 app 是否有权限。
```

7.2 常用方法

1.authenticate()：

提供了用户认证功能，即验证用户名以及密码是否正确，一般需要 username 、password 两个关键字参数。如果认证成功（用户名和密码正确有效），便会返回一个 User 对象。authenticate() 会在该 User 对象上设置一个属性来标识后端已经认证了该用户，且该信息在后续的登录过程中是需要的，认证失败返回 None。

user = auth.authenticate(username=’theuser’,password=’thepassword’)

2.login(HttpRequest, user):

该函数接受一个 HttpRequest 对象，以及一个经过认证的 User 对象。该函数实现一个用户登录的功能。它本质上会在后端为该用户生成相关 session 数据。

```
def my_view(request):
    username = request.POST['username']
    password = request.POST['password']
    user = auth.authenticate(username=username, password=password)
    if user is not None:
        auth.login(request, user)
        # Redirect to a success page.
        ...
    else:
    # Return an 'invalid login' error message.
        ...
```

3.logout(request)：

该函数接受一个 HttpRequest 对象，无返回值。当调用该函数时，当前请求的 session 信息会全部清除。该用户即使没有登录，使用该函数也不会报错。

```
def logout_view(request):
    auth.logout(request)
    # Redirect to a success page.
```

4.is_authenticated():

用来判断当前请求是否通过了认证。

```
def my_view(request):
    if not request.user.is_authenticated():
    return redirect('%s?next=%s' % (settings.LOGIN_URL, request.path))
```

5.login_requierd()：

auth 给我们提供的一个装饰器工具，用来快捷的给某个视图函数添加登录校验。

```
from django.contrib.auth.decorators import login_required

@login_required
def my_view(request):
    ...
```

若用户没有登录，则会跳转到 django 默认的登录 URL ‘/accounts/login/ ‘并传递当前访问 url 的绝对路径 (登陆成功后，会重定向到该路径)。如果需要自定义登录的 URL，则需要在 settings.py 文件中通过 LOGIN_URL 进行修改。LOGIN_URL = ‘/login/’ # 这里配置成你项目登录页面的路由

6.method_decorator()：

auth 给我们提供的一个装饰器工具，用来快捷的给某个类视图添加登录校验。

```
from django.contrib.auth.decorators import login_required
from django.utils.decorators import method_decorator

class Home(views.View):

    @method_decorator(login_required)
    def get(self, request):
        return render(request, 'home.html')
```

7.create_user()：

auth 提供的一个创建新用户的方法，需要提供必要参数（username、password）等。

```
from django.contrib.auth.models import User

user = User.objects.create_user(username = '用户名', password = '密码', email = '邮箱', .
↪..)
```

(continues on next page)

(续上页)

```
check_password(password)
```

8.create_superuser()：

auth 提供的一个创建新的超级用户的方法，需要提供必要参数（username、password）等。

```
from django.contrib.auth.models import User

user = User.objects.create_superuser(username = '用户名', password = '密码', email = '邮箱', ...)
check_password(password)
```

9.permission_requires(权限,raise_exception=True)：

对视图函数指定权限验证。

7.3 Group

django.contrib.auth.models.Group 定义了用户组的模型，每个用户组拥有 id 和 name 两个字段，该模型在数据库被映射为 auth_group 数据表。

User 对象中有一个名为 groups 的多对多字段，多对多关系由 auth_user_groups 数据表维护。Group 对象可以通过 user_set 反向查询用户组中的用户。

我们可以通过创建删除 Group 对象来添加或删除用户组:

```
# add
group = Group.objects.create(name=group_name)
group.save()
# del
group.delete()
```

我们可以通过标准的多对多字段操作管理用户与用户组的关系:

```
# 用户加入用户组
user.groups.add(group)
# 或者
group.user_set.add(user)

# 用户退出用户组
user.groups.remove(group)
# 或者
group.user_set.remove(user)
```

(continues on next page)

(续上页)

```
# 用户退出所有用户组
user.groups.clear()

# 用户组中所有用户退出组
group.user_set.clear()
```

7.4 Permission

Django 的 auth 系统提供了模型级的权限控制，即可以检查用户是否对某个数据表拥有增 (add), 改 (change), 删 (delete) 权限。

auth 系统无法提供对象级的权限控制，即检查用户是否对数据表中某条记录拥有增改删的权限。如果需要对象级权限控制可以使用 django-guardian。

假设在博客系统中有一张 article 数据表管理博文，auth 可以检查某个用户是否拥有对所有博文的管理权限，但无法检查用户对某一篇博文是否拥有管理权限。

检查用户权限

```
user.has_perm 方法用于检查用户是否拥有操作某个模型的权限:
user.has_perm('blog.add_article')
user.has_perm('blog.change_article')
user.has_perm('blog.delete_article')
```

上述语句检查用户是否拥有 blog 这个 app 中 article 模型的添加权限，若拥有权限则返回 True。

has_perm 仅是进行权限检查, 即是用户没有权限它也不会阻止程序员执行相关操作。

```
@permission_required 装饰器可以代替 has_perm 并在用户没有相应权限时重定向到登录页或者抛出异常。
# permission_required(perm[, login_url=None, raise_exception=False])

@permission_required('blog.add_article')
def post_article(request):
    pass
```

每个模型默认拥有增 (add), 改 (change), 删 (delete) 权限。在 django.contrib.auth.models.Permission 模型中保存了项目中所有权限。

该模型在数据库中被保存为 auth_permission 数据表。每条权限拥有 id ,name , content_type_id, codename 四个字段。

管理用户权限

User 和 Permission 通过多对多字段 user.user_permissions 关联，在数据库中由 auth_user_user_permissions 数据表维护。

```
# 添加权限
user.user_permissions.add(permission)

# 删除权限:
user.user_permissions.delete(permission)

# 清空权限:
user.user_permissions.clear()
```

用户拥有他所在用户组的权限，使用用户组管理权限是一个更方便的方法。Group 中包含多对多字段 permissions，在数据库中由 auth_group_permissions 数据表维护。

```
# 添加权限:
group.permissions.add(permission)

# 删除权限:
group.permissions.delete(permission)

# 清空权限:
group.permissions.clear()
```

自定义权限

在定义 Model 时可以使用 Meta 自定义权限:

```
class Discussion(models.Model):
  ...
  class Meta:
      permissions = (
          ("create_discussion", "Can create a discussion"),
          ("reply_discussion", "Can reply discussion"),
      )
```

判断用户是否拥有自定义权限:

```
user.has_perm('blog.create_discussion')
```

7.5 视图的用户权限

无法添加或更改某个模型的用户将无法在管理员中看到。

如果我们在谈论自定义创建的视图，那么您可以创建一些检查用户权限的内容，如果没有该权限，则返回 404。权限与模型相关联，组可分配各种权限。

您可以向模型添加权限:

```
# myproject / myapp / models.py

class MyModel (models.Model):
class Meta:
permission = (
('permission_code', ' 友好许可描述'),
)
```

然后你可以检查一个用户是否有这样的权限：

```
@user_passes_test (lambda u: u.has_perm (' myapp.permission_code'))
def some_view (request):
#...
```

使用权限，您可以使用管理界面轻松地添加或删除用户和组。

7.6 用法实例

7.6.1 简单的认证

Login_required 装饰器

判断是否登陆，没有就自动重定向某个地方

用法：（局限于装饰函数）（如果要装饰一个类，则不能，要使用 method_decorator）

```
@login_required(login_url='users:login')
def index_fn(reuqest):
    return HttpResponse('大家都是大牛')
```

自动跳转到 login 页面

登陆后：

method_decorator 装饰器

```
@method_decorator(login_required(login_url='users:login'),name='dispatch')
class Index(View):
    def get(self, request):
        return render(request, 'admin/index.html', context={'request': request})
```

用法基本一样也就是一个是类视图一个是函数视图。

但是是否每一个都需要添加呢？不符合优秀程序员的做法（手动滑稽）因为实在是太丑了。

7.6.2 Minxin 认证

LoginRequired Mixin

需要设定重定向的 URL（有一点 restful 的风格）

```
class Index(LoginRequiredMixin,View):
    login_url = 'users:login'
    def get(self, request):
        return render(request, 'admin/index.html', context={'request': request})
```

聪明的看得出来了，是否每一次都添加 login_url 呢，这里显然不是的，可以将 login_url 设置在 Settings 里面，我的 Mixin 会自动去 settings 里面寻找 LOGIN_URL 的参数。

```
LOGIN_URL = 'users:login'
```

```
?next=/admin/         可以自定义 next
@login_required(redirect_field_name='my_redirect_field')      这样就可以代替 next
```

自动帮我们添加一个查询的参数在 URL 上面。

user_passes_test

这个比较通用于函数视图

以下为官方文档。描述为判断登陆的用户的邮箱以‘@example.com’结尾，就为 True，允许访问，如果相反，那么就禁止访问。可自定义

```
from django.contrib.auth.decorators import user_passes_test

def email_check(user):
    return user.email.endswith('@example.com')

@user_passes_test(email_check)
def my_view(request):
    ...
```

userPassesTestMixin

这个通用于类视图。

```
class Index(UserPassesTestMixin,View):
    def test_func(self):
        return self.request.user.username.endswith('son')
```

(continues on next page)

(续上页)

```
    def get(self, request):
        return render(request, 'admin/index.html', context={'request': request})
```

test_func 为 Mixin 所定的。测试是否通过，为 TRUE 则可以进入，FALSE 为相反。

7.6.3 权限

permission_required decorator

首先这里要注意权限和认证的区别，认证给你登陆了，但不一定给你看某些好看的东西。除非微信转钱。

permission_required(perm.login_url=None,raise_exception=False)

perm 为权限多个权限就可以用一个列表或者元组都行。放在函数视图里面。

```
from django.contrib.auth.decorators import permission_required

@permission_required('polls.can_vote')
def my_view(request):
    ...
```

perm 第一部分模型名字第二部分为 codename

permissionRequiredMixin mixin

使用于类视图里面。

```
lass TagManage(PermissionRequiredMixin,View):
    """
    create tags manage view
    route: admin/tags/int
    """
    permission_required = ('news.add_tag', 'news.view_tag')
    raise_exception = True # 403
    def get(self, request):
```

没有权限直接 403

7.7 参考

Django 权限系统 auth 模块详解：https://www.cnblogs.com/ccorz/p/6358074.html

Django2.0——权限系统 Auth:https://blog.csdn.net/wzyaiwl/article/details/88597166

Django - 某些视图的用户权限？(Django - user permissions to certain views?)：https://www.it1352.com/635496.html

django 权限功能 (根据不同的用户，设置不同的显示和访问权限):https://www.cnblogs.com/jackson669/p/12787676.html

CHAPTER 8

django 入门进阶 08 数据库事务

8.1 锁

1.1：乐观锁：

概念：同一条数据很少会因为并发修改而产生冲突，适用于读多写少的场景。

实现方式：读取一个字段，执行处理逻辑，当需要更新数据时，再次检查该字段是否和第一次读取一致。如果一致，更新数据，否则不更新，重新读取后再提交。

1.2：悲观锁：

概念：当一条数据正在被修改时，不允许其他任何关于这条数据的操作。

实现方式：读取一个字段之后，加锁，不允许其他任何读、写操作。执行处理逻辑，更新数据完毕后，释放锁。

1.3：二者比较：

乐观锁的开销远低于悲观锁

原因：当 A 锁定了 a 资源，需要 b 资源。而 b 被 B 锁定，正在等待 a 资源。此时，导致出现死锁。也可以通过设置超时来处理这个问题。

悲观锁可以有效降低冲突后，重试的次数

乐观锁可以提高响应速度

在并发比较少时，建议使用乐观锁，减少加锁、释放锁的开销，在并发比较高的时候，建议使用悲观锁。

8.2 事务

Django 默认每条数据库操作都会被立即提交到数据库。

会导致一个问题，若有一系列的数据库操作构成，要么全部执行，要么全部都不执行。

这时需要事务，将一系列数据库操作设置为一个事务，提交给数据库执行。

Django 提供 atomic 装饰器以开启事务，acomic 使用一个参数来指定数据库的名字，若不设置，django 会使用系统默认的数据库。

2.1：整个 View 函数开启事务：

```
from django.db import transaction

@transaction.atomic
def view(request):
        """ 整个 view 都开启了事务"""
    func()
```

2.2：部分函数开启事务：

```
from django.db import transaction

def view(request):
    func()

        # 下面开始执行事务操作，若出现异常操作，会回退到这里
    with transaction.atomic():
        # 开启了事务
        atmice_func()
```

2.3：不要再事务中处理异常：

```
from django.db import transaction

def view(request):
    func()
    try:
        with transaction.commit_on_success():
            do_more_stuff_1() # in transaction
            try:
                do_more_stuff_2() # not in transaction
            except:
                pass
            do_more_stuff_3() # in transaction
```

(continues on next page)

(续上页)

```
    except:
        pass
```

当退出原子块时，Django 会查看它是否正常退出或者是否有异常来确定是否提交或者回滚。

如果你捕获并处理原子块中的异常，可能会隐藏 Django 中发生问题的事实。

8.3 F 函数更新运算

通常更新数据库的操作，需要将对象读取到内存。在内存中进行修改之后，再写回数据库。

在内存中的操作，如果存在同时操作的情况，会导致运算逻辑错误。

F() 函数的作用就是直接生成 SQL 语句，不必将需要更新的对象读取到内存。避免了并发导致的数据不一致问题。

```
from django.db.models import F

course_obj = Course.objects.get(pk=1)
course_obj.purchased_quantity = F('purchased_quantity') + 1
course_obj.save()
```

使用 F() 函数保存值后, **再次使用实例调用并不能拿到新的值**. 这是因为 F() 函数是数据库操作, 并不是在内存中 python 进行的, 所以之前拿到的实例存储的还是之前的值. 所以**需要重新载入实例 (即重新获取实例)**。

优点 01:**F() 函数配合 update 可以优化效率**, 不再需要使用 get() 和 save() 方法

更新单个实例

```
reporter = Reporters.objects.filter(name='Tintin')
reporter.update(stories_filed=F('stories_filed') + 1)
```

更新多个实例

```
Reporter.objects.all().update(stories_filed=F('stories_filed') + 1)
```

优点 02:**F() 函数避免竞争**

8.4 Q 对象

多个过滤器逐个调用表示逻辑与关系，同 sql 语句中 where 部分的 and 关键字。

例：查询阅读量大于 20，并且编号小于 3 的图书。

list=BookInfo.objects.filter(bread__gt=20,id__lt=3) 或 list=BookInfo.objects.filter(bread__gt=20).filter(id__lt=3) 如果需要实现逻辑或 or 的查询，需要使用 Q() 对象结合 | 运算符，Q 对象被义在 django.db.models 中。

语法如下：Q(属性名 __ 运算符 = 值) 例：查询阅读量大于 20 的图书，改写为 Q 对象如下。

```
from django.db.models import Q
list = BookInfo.objects.filter(Q(bread__gt=20))
```

Q 对象可以使用 &、| 连接，& 表示并且，| 表示或。

例：查询阅读量大于 20，或编号小于 3 的图书，只能使用 Q 对象实现

list = BookInfo.objects.filter(Q(bread__gt=20) | Q(pk__lt=3))

Q 对象前可以使用 ~ 操作符，表示非 not。例：查询编号不等于 3 的图书。

```
list = BookInfo.objects.filter(~Q(pk=3))exclude() 也是取反
list = BookInfo.objects.exclude(id=3)
list = BookInfo.objects.exclude(id__exact=3)
```

聚合函数

使用 aggregate() 过滤器调用聚合函数。聚合函数包括：Avg，Count，Max，Min，Sum，被定义在 django.db.models 中。

例：查询图书的总阅读量。

```
from django.db.models import Sum
list = BookInfo.objects.aggregate(Sum('bread'))
```

注意 aggregate 的返回值是一个字典类型，格式如下：

{‘属性名 __ 聚合类小写’：值}

如:{‘bread__sum’:3}

使用 count 时一般不使用 aggregate() 过滤器。

例：查询图书总数。

list = BookInfo.objects.count()

注意 count 函数的返回值是一个数字。

8.5 利用 select_for_update 函数

```
select_for_update 使用的是悲观锁，这是数据库层面的，解决并发取数据后再修改的问题方法。

from django.db import transaction
```

(continues on next page)

(续上页)

```
from django.http import HttpResponse

# 类视图 (并发, 悲观锁)
class MyView(View):

    @transaction.atomic
    def post(self, request):
        # select * from 表名 where id=1 for update;
        # for update 就表示锁, 只有获取到锁才会执行查询, 否则阻塞等待。
        obj = 模型类名.objects.select_for_update().get(id=1)

        # 等事务提交后, 会自动释放锁。

        return HttpResponse('ok')
```

8.6 其他

django 中操作数据库均是对模型类进行 CRUD 操作

8.7 参考

Django 中事务的处理:https://www.jianshu.com/p/23ccd5f254bf

Django-框架保证并发时数据一致性:https://blog.csdn.net/Fe_cow/article/details/90267103

django 查询之 F 函数:https://www.jianshu.com/p/7562f5ed983e

Django models 模型-条件查询：https://www.cnblogs.com/daidechong/p/10820175.html

CHAPTER 9

django 入门进阶 09 中间件

9.1 什么是中间件

django 的中间件（middleware）是一个轻量级的插件系统，在 django 中的请求和响应中，可以利用中间件干预视图的请求和响应。

9.2 如何启用中间件

若要启用中间件组件，请将其添加到 Django 配置文件 settings.py 的 MIDDLEWARE 配置项列表中。

```
settings.py
34        'django.contrib.contenttypes',
35        'django.contrib.sessions',
36        'django.contrib.messages',
37        'django.contrib.staticfiles',
38        'Buyer',
39        'Saller',
40        'djcelery',
41    ]
42
43    MIDDLEWARE = [
44        'django.middleware.cache.UpdateCacheMiddleware',
45        'django.middleware.security.SecurityMiddleware',
46        'django.contrib.sessions.middleware.SessionMiddleware',
47        'django.middleware.common.CommonMiddleware',
48        'django.middleware.csrf.CsrfViewMiddleware',
49        'django.contrib.auth.middleware.AuthenticationMiddleware',
50        'django.contrib.messages.middleware.MessageMiddleware',
51        'django.middleware.clickjacking.XFrameOptionsMiddleware',
52        'django.middleware.cache.FetchFromCacheMiddleware',
53    ]
```

9.3 五大钩子函数

传统方式自定义中间件其实就是在编写五大钩子函数：

```
process_request(self,request)
process_response(self, request, response)
process_view(self, request, view_func, view_args, view_kwargs)
process_exception(self, request, exception)
process_template_response(self,request,response)
```

可以实现其中的任意一个或多个！

9.4 中间件的顺序问题

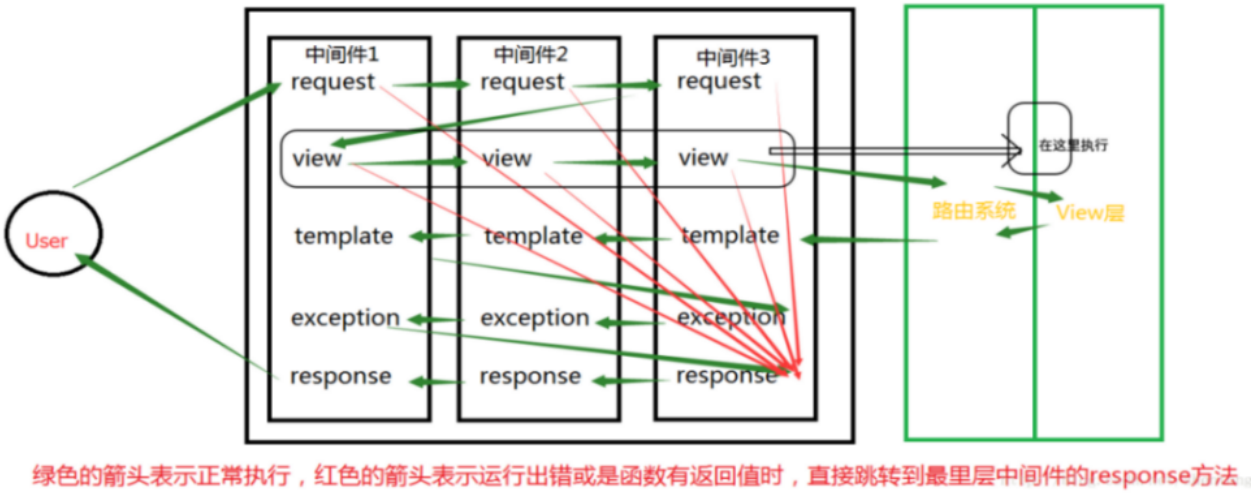
从上一部分可以看出，中间件是有多个的，既然有多个必然涉及到优先级或顺序问题，顺序大体上符合先来后到（不存在插队类型的优先级，要么正序，要么逆序，不能插队跳跃执行）

MIDDLEWARE 的顺序很重要，具有先后关系，因为有些中间件会依赖其他中间件。例如：Authentication-Middleware 需要在会话中间件中存储的经过身份验证的用户信息，因此它必须在 SessionMiddleware 后面运行。

在请求阶段，调用视图之前，Django 按照定义的顺序执行中间件 MIDDLEWARE，自顶向下。

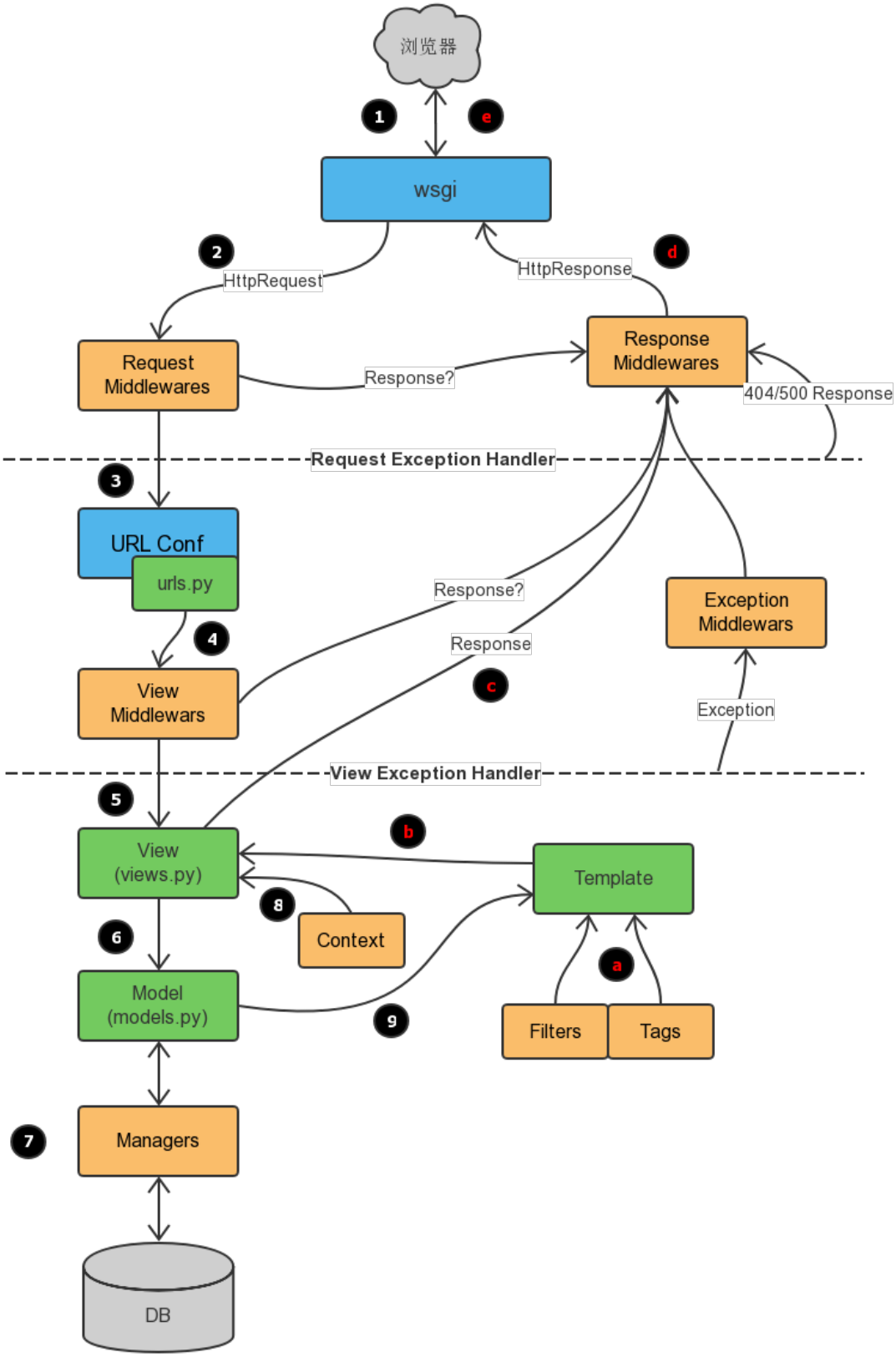
你可以把它想象成一个**洋葱：每个中间件类都是一个“皮层”**，它包裹起了洋葱的核心–实际业务视图。如果请求**通过了洋葱的所有中间件层，一直到内核的视图**，那么响应将在**返回的过程中以相反的顺序再通过每个中间件层**，最终返回给用户。

如果某个层的执行过程认为当前的请求应该被拒绝，或者发生了某些错误，导致**短路，直接返回了一个响应**，那么**剩下的中间件以及核心的视图函数都不会被执行**。



图上一部分标识错误了，已经标注出来 (view 的执行在路由后，view 视图函数前)

更为详细的执行流


浏览器
1
e
wsgi
2
HttpRequest
HttpResponse
d
Request Middlewares
Response?
Response Middlewares
404/500 Response
Request Exception Handler
3
URL Conf
urls.py
Response?
Exception Middlewars
4
Response
View Middlewars
c
Exception
View Exception Handler
5
View (views.py)
b
Template
8
Context
6
a
Model (models.py)
9
Filters
Tags
7
Managers
DB

9.5 可以做什么

Cache: 缓存中间件

如果启用了该中间件，Django 会以 CACHE_MIDDLEWARE_SECONDS 配置的参数进行全站级别的缓存。

Common: 通用中间件

禁止 DISALLOWED_USER_AGENTS 中的用户代理访问服务器

自动为 URL 添加斜杠后缀和 www 前缀功能。如果配置项 APPEND_SLASH 为 True ，并且访问的 URL 没有斜杠后缀，在 URLconf 中没有匹配成功，将自动添加斜杠，然后再次匹配，如果匹配成功，就跳转到对应的 url。PREPEND_WWW 的功能类似。

为非流式响应设置 Content-Length 头部信息。

GZip: 内容压缩中间件

用于减小响应体积，降低带宽压力，提高传输速度。

该中间件必须位于其它所有需要读写响应体内容的中间件之前。

Locale: 本地化中间件

用于处理国际化和本地化，语言翻译。

Message: 消息中间件

基于 cookie 或者会话的消息功能，比较常用。

Security: 安全中间件

Site: 站点框架。

这是一个很有用，但又被忽视的功能。

它可以让你的 Django 具备多站点支持的功能。

通过增加一个 site 属性，区分当前 request 请求访问的对应站点。

无需多个 IP 或域名，无需开启多个服务器，只需要一个 site 属性，就能搞定多站点服务。‘

Authentication: 认证框架

Django 最主要的中间件之一，提供用户认证服务。

CSRF protection: 提供 CSRF 防御机制的中间件

X-Frame-Options: 点击劫持防御中间件

9.6 可能遇到问题

全局异常 object() takes no parameters

错误信息如下

```
#  File "C:\python\lib\site-packages\django\core\handlers\wsgi.py", line 151, in __
↪init__
#     self.load_middleware()
#   File "C:\python\lib\site-packages\django\core\handlers\base.py", line 82, in load_
↪middleware
#     mw_instance = middleware(handler)
# TypeError: object() takes no parameters
```

出错原始代码:

```
# from django.http import HttpResponse
# class MyException(object):
#     def process_exception(request,response, exception):
#         return HttpResponse(exception.message)
```

正确代码 1:

```
# class MyException(object):
#     def __init__(self, get_response):
#         self.get_response = get_response
#
#     def __call__(self, request):
#         return self.get_response(request)
#
#     def process_exception(self, request, exception):
#         return HttpResponse(exception)
```

正确代码 2

```
from django.http import HttpResponse
from django.utils.deprecation import MiddlewareMixin

class MyException(MiddlewareMixin):
    def process_exception(self, request, exception):
        return HttpResponse(exception)
```

兼容 Django 新版本和旧版本

```
try:
    from django.utils.deprecation import MiddlewareMixin  # Django 1.10.x
except ImportError:
    MiddlewareMixin = object  # Django 1.4.x - Django 1.9.x
```

(continues on next page)

(续上页)

```
class SimpleMiddleware(MiddlewareMixin):
    def process_request(self, request):
        pass

    def process_response(request, response):
        pass
```

9.7 参考

Django(十二)：Django 框架中的 middleware 中间件:https://blog.csdn.net/qq_40558166/article/details/102467833

Django 框架全面讲解 – 中间件（MiddleWare）:https://blog.csdn.net/shentong1/article/details/78829599

Django 中间件:https://www.runoob.com/django/django-middleware.html

Django 之 Middleware(中间件):https://www.cnblogs.com/Alexephor/p/11272839.html

Django 中间件:https://www.liujiangblog.com/blog/45/

Django 中间件:https://code.ziqiangxuetang.com/django/django-middleware.html

Python Django，中间件，中间件函数，全局异常处理：https://blog.csdn.net/houyanhua1/article/details/85028983

django 全局异常捕获保存和输出，logger 配置:https://hhyo.github.io/2018/04/16/django-traceback/

django 中间件修改，以及 TypeError: object() takes no parameters 的异常处理:https://my.oschina.net/u/4374968/blog/3899317

CHAPTER 10

django 入门进阶 10 部署上线 (nginx,uwsgi,supervisor)

10.1 django 自身服务 ok

python manage.py runserver，验证可正常访问

10.2 uwsgi 安装和服务验证

安装:pip Install uwsgi

测试代码

```
def application(env, start_response):

    start_response('200 OK', [('Content-Type','text/html')])

    return [b"Hello World"]
```

执行下面命令行:

```
uwsgi --plugin python --http :8001 --wsgi-file test.py
```

重新访问 localhost:8001

就可以看到成功的显示了‘Hello world’

如果报错:error while loading shared libraries: libpcre.so.1: cannot open shared object file: No such file or directory

解决方式:

```
sudo apt-get install libpcre3 libpcre3-dev # 安装需要的包

find / -name libpcre.so.3 # 找到 libpcre.so.3 (一般在根目录/lib/x86_64-linux-gnu 下)

找到 /lib/x86_64-linux-gnu/libpcre.so.3

sudo ln -s /lib/x86_64-linux-gnu/libpcre.so.3 /usr/lib/libpcre.so.1 # 做软链接即可
```

10.3 uwsgi 对接 django 配置 ini

```
[uwsgi]

# 使用 nginx 连接时使用

#socket=127.0.0.1:8080  # 正式上线后使用此模式，速度稍有优势
# 直接做 web 服务器使用

http=127.0.0.1:8080  # 联调阶段优先使用 http 模式，方便定点测试
# 项目目录

chdir=/home/shuan/dailiyfresh # 这里可能需要多次尝试，如果报错 can't find module **，大概率这
里问题

# 项目中 wsgi.py 文件的目录，相对于项目目录

wsgi-file=dailiyfresh/wsgi.py # 这里尽可能使用绝对路径避免踩坑
# 指定启动的工作进程数

processes=4

# 指定工作进程中的线程数

threads=2

master=True

# 保存启动之后主进程的 pid
```

(continues on next page)

(续上页)

```
pidfile=uwsgi.pid

# 设置 uwsgi 后台运行，用 uwsgi.log 保存日志信息

daemonize=uwsgi.log

# 设置虚拟环境的路径

virtualenv=/home/shuan/.virtualenvs/bj18_py3# conda 环境路径
```

启动：uwsgi –ini uwsgi.ini

停止：uwsgi –stop uwsgi.pid

启动后访问 8080 查看是否启动成功。

10.4 uwsgi 对接 supervisor

安装 supervisor：pip install supervisor

生成初始配置文件：echo_supervisord_conf > /etc/supervisord.conf

```
[program:zqxt]

command=/usr/bin/uwsgi(视环境情况 which uwsgi，本例用 pip 安装应该是 conda 环境里的 uwsgi 地
址) --ini uwsgi_conf.ini  # 可终端单独执行此命令，确保正确
directory=/path/to/zqxt# 同 uwsgi_conf 的 chdir，项目目录

startsecs=0

stopwaitsecs=0

autostart=true # 测试阶段改为 false，让错误暴露出来

autorestart=true # 测试阶段改为 false，让错误暴露出来
```

修改:supervisor_conf

```
redirect_stderr = true  ; 把 stderr 重定向到 stdout，默认 false
stdout_logfile_maxbytes = 20MB  ; stdout 日志文件大小，默认 50MB
stdout_logfile = /data/log/plantool_stdout.log
stderr_logfile = /data/log/plantool_err.log
```

修改 wsgi_conf.ini

```
#daemonize=/var/log/uwsgi8011.log   #  守护进程一定要注释掉 (关键)
```

功能测试

```
启用 config 配置: supervisord -c /etc/supervisord.conf
supervisorctl status        //查看所有进程的状态
supervisorctl stop es       //停止 es
supervisorctl start es      //启动 es
supervisorctl restart es     //重启 es
supervisorctl update        //配置文件修改后使用该命令加载新的配置
supervisorctl reload        //重新启动配置中的所有程序
```

启动只是启动 conf 配置，需要 supervisorctl start 才是真正启动（也就是说，使用 supervisorctl 前必须先执行 supervisord）

下面这一段描述问题在于 supervisord 当做开启服务的了，其实不是，supervisord 之后使用 supervisorctl 才是真正开启服务.

再次启动，查看 supervisorctl status，状态为退出 (exited)，可见出了问题，查看错误日志 plantool_err.log

```
your memory page size is 4096 bytes
detected max file descriptor number: 1024
lock engine: pthread robust mutexes
thunder lock: disabled (you can enable it with --thunder-lock)
The -s/--socket option is missing and stdin is not a socket.
^C
```

thunder lock：search ，没有效信息，所以这个可能是正常状态的日志。

猜–socket 导致问题，搜索也没发现有效信息，可能也是对的。

10.5 nginx 对接 uwsgi

先确保 nginx 安装成功

nginx 安装后：http://localhost, 如果显示 nginx 欢迎页，说明 nginx 默认配置 ok

修改 nginx 配置 (大概意思)

```
server {

    listen       80 ;

    charset      utf-8;
```

(continues on next page)

(续上页)

```
    client_max_body_size 75M;


    location /api {  # 后台 api 接口
        proxy_pass 127.0.0.1:8080;
    }

    location /media  {

        alias /path/to/project/media;

    }



    location /static {

        alias /path/to/project/static;

    }



    location / {  # html 资源文件
        root /path/to/template;
        index index.html;
    }

}
```

先测试后端接口:127.0.0.1:8080/api/,ok

再测试转发接口:127.0.0.1:80/api/, 是否正确转发.

最后再测试 html, 静态文件 (static), 媒体文件 (media）等.

可能问题：报错 nginx: [emerg] getgrnam(“nginx”) failed：https://blog.csdn.net/qq_39556759/article/details/78406813

解决方法：

vim /aplication/nginx/conf/nginx.conf

去掉 user nobody 之前的 # 号（也就是说启用 user 的配置项）

可能问题：访问 Html,css,js 资源文件，报错，13: Permission denied

前端，templates/index.html，无权限

```
/templates/favicon.ico" failed (13: Permission denied), client: 127.0.0.1, server: , request: "GET /favicon.ico HTTP/1.1", host: "127.0.0.1:8017", ref
/templates/index.html" failed (13: Permission denied), client: 127.0.0.1, server: , request: "GET /index.html HTTP/1.1", host: "127.0.0.1:8017"
/templates/favicon.ico" failed (13: Permission denied), client: 127.0.0.1, server: , request: "GET /favicon.ico HTTP/1.1", host: "127.0.0.1:8017", ref
/templates/index.html" failed (13: Permission denied), client: 127.0.0.1, server: , request: "GET /index.html HTTP/1.1", host: "127.0.0.1:8017"
/templates/favicon.ico" failed (13: Permission denied), client: 127.0.0.1, server: , request: "GET /favicon.ico HTTP/1.1", host: "127.0.0.1:8017", ref
/templates/index.html" failed (13: Permission denied), client: 127.0.0.1, server: , request: "GET /index.html HTTP/1.1", host: "127.0.0.1:8017"
/templates/favicon.ico" failed (13: Permission denied), client: 127.0.0.1, server: , request: "GET /favicon.ico HTTP/1.1", host: "127.0.0.1:8017", ref
/templates/index.html" failed (13: Permission denied), client: 127.0.0.1, server: , request: "GET /index.html HTTP/1.1", host: "127.0.0.1:8017"
/templates/favicon.ico" failed (13: Permission denied), client: 127.0.0.1, server: , request: "GET /favicon.ico HTTP/1.1", host: "127.0.0.1:8017", ref
/templates/index.html" failed (13: Permission denied), client: 127.0.0.1, server: , request: "GET /index.html HTTP/1.1", host: "127.0.0.1:8017"
/templates/favicon.ico" failed (13: Permission denied), client: 127.0.0.1, server: , request: "GET /favicon.ico HTTP/1.1", host: "127.0.0.1:8017", ref
```

权限不足的解决方案:

一、由于启动用户和 nginx 工作用户不一致所致（user 配置项配置错误）

01，查看 nginx 的启动用户，发现是 nobody，而非 root 启动的

命令：ps aux | grep “nginx: worker process” | awk’{print $1}’

```
[root@app13 nginx]# ps aux | grep "nginx: worker process" | awk '{print $1}'
nobody
root
```

02，将 nginx.config 的 user 改为和启动用户一致，

命令：vi conf/nginx.conf

```
[root@app13 nginx]# vi conf/nginx.conf


user  root;
worker_processes  1;

#error_log  logs/error.log;
#error_log  logs/error.log  notice;
#error_log  logs/error.log  info;

#pid        logs/nginx.pid;
```

这一步，根据个人经验，**应该修改为” /html/or/templates/目录拥有者的用户” 和组信息**，比如

当前为用户 john(/templates/拥有者，负责启动 nginx 的人，是否有 root 权限无所谓), 则通过” id john” 查看所属组, 比如 work 组

则配置:” user john work”，**效果是启动后的 nginx 子进程显示的启动者是 john**（而实际 nginx 启动者可能是别人，有 root 权限的其他人）

这一步可能需要多做尝试，本人这一步卡了好久，尝试了很多组合，最后才发现正确配置 (主要是网上教程差异很大，拜版本不同所赐，踩坑颇多)

```
user www-data    #nginx -t 测试通过, 启动后访问权限不足
user nobody     # nginx -t 测试不通过,
user john    # nginx -t 测试不通过, (john 无 root 权限, 无法启动 nginx, 但是 templates 目录拥有者)
user yyyy    # nginx -t 测试不通过, (yyyy 有 root 权限, 且是 nginx 服务启动者)
删除:user 这一行    # nginx -t 测试通过, 启动后访问权限不足
```

(continues on next page)

(续上页)

```
user work john     # nginx -t 测试不通过,
user work yyyy     # nginx -t 测试不通过,
user john work     # nginx -t 测试通过, 启动后访问 ok
```

二、templates 权限问题，**如果 nginx 没有 web 目录的操作权限，也会出现 403 错误**。

解决办法：修改 web 目录的读写权限，或者是把 nginx 的启动用户改成目录的所属用户，重启 Nginx 即可解决

```
chmod -R 777 /data
chmod -R 777 /data/www/
```

10.6 验证无问题后的进一步改进

01，nginx 的转发（到 uwsgi），从 http 转发模式改为 socket 转发模式（nginx.conf,uwsgi.conf）（对接阶段使用 http，方便独立的正确性验证。正式环境改为 socket 模式，保证速度以及减少端口占用。）

02，supervisord.conf 配置的 autostart 和 autorestart 改为 true。确保进程死机后自动重启。

03，检查各路径配置，是否有私有路径（别人无法访问的路径），可能导致别人无法启动项目。

10.7 其他注意事项

10.7.1 启用 django 后台管理 admin 模块

启用 admin 后会发现无法进入登录界面（部分资源无法加载 static/admin/simple-ui/xxxx）

原因：问题 admin/下的静态资源无法访问。

大部分项目前后端完全分离，所以 templates 和 static 一般都是前端组提供，我们想当然就用了，而实际上 django_admin 模块内部也包含部分静态资源，当使用 django 内置服务器时可以检索到，但如果部署到线上则必须将 admin 内静态文件导出，整合到统一的 templates 目录中 (让 nginx 检索到)。解决：

```
python manage collectstatic # 自动收集静态文件到 django_setting 配置的 STATIC_ROOT 中
cp STATIC_ROOT templates/static
chmod -R 777 xxx
```

10.8 参考

linux 下部署 Django uwsgi: error while loading shared libraries: libpcre.so.1: cannot open shared object file: No such file or directory:https://www.cnblogs.com/erhangboke/p/11673156.html

初次使用 uwsgi:no python application found, check your startup logs for errors:https://www.cnblogs.com/loveyangaddddd/p/8119720.html

uWSGI 出现错误：no python application found, check your startup logs for errors:https://blog.csdn.net/weixin_40576010/article/details/89000128

supervisor 管理 uwsgi：https://www.cnblogs.com/supery007/p/9368242.html

Django 部署 (Nginx)：https://code.ziqiangxuetang.com/django/django-nginx-deploy.html

使用 supervisor 作为 uWSGI 的守护进程:luchanghong.lofter.com/post/f04c0_242345

Supervisor 使用详解:https://www.jianshu.com/p/0b9054b33db3

解决 Nginx 出现 403 forbidden (13: Permission denied) 报错的四种方法：https://www.cnblogs.com/williamjie/p/9604594.html

nginx 错误集锦：https://www.jianshu.com/p/3de849802a89

Nginx 之 proxy_pass 指令 url 反斜杠作用：https://blog.csdn.net/sleepIII/article/details/100787652

CHAPTER 11

django 入门进阶 11websocket

本文适合有一定 websocket 基础的，至少完整看过前后端 demo 的读者，一窍不通的小白建议先阅读“参考”部分的博文扫扫盲。

基于 django 的 dwebsocket 组件 (目前虽然不在维护，但正常使用没问题)

11.1 前端方法

```
<script type="text/javascript">
    var socket = new WebSocket("ws:" + window.location.host + "/drug/drug_connect/");
    socket.onopen = function () {
        console.log('WebSocket open');//成功连接上 Websocket
        socket.send('adasdasda。。。。');//发送数据到服务端
    };
    socket.onmessage = function (e) {
        console.log('message: ' + e.data);//打印服务端返回的数据
    };
    socket.onclose=function(e){
      console.log(e);
      socket.close(); //关闭 TCP 连接
    };
    if (socket.readyState == WebSocket.OPEN) socket.onopen();
</sctipt>
```

由于 js 的异步友好性，所以代码看起来非常清爽，也容易理解。socket 连接，成功后干什么（onopen），收到消息后干嘛（onmessage），如何关闭等。而且也会自动发送 Pingpong 包确保连接的保持。

11.2 后端方法

dwebsocket 的一些内置方法：

request.is_websocket（）：判断请求是否是 websocket 方式，是返回 true，否则返回 false

request.websocket：当请求为 websocket 的时候，会在 request 中增加一个 websocket 属性，

WebSocket.wait（）返回客户端发送的一条消息，没有收到消息则会导致阻塞

WebSocket.read（）和 wait 一样可以接受返回的消息，只是这种是非阻塞的，没有消息返回 None

WebSocket.count_messages（）返回消息的数量

WebSocket.has_messages（）返回是否有新的消息过来

WebSocket.send（message）像客户端发送消息，message 为 byte 类型

后端方法相对前端就没有那么友好了。如果面对一个特定需求如何实现呢?

11.3 场景 1，1v0 聊天

这个实际业务中没啥用，仅用来验证 websocket 接口性质，了解接口特性而已。

```
@accept_websocket
def start_server_script(request):
    if request.is_websocket():
        # " 这里实现 wbsocket 连接逻辑"
        for info in request.websocket:# 这里需要注意的是，这个 for 后面的对象，request.
↪websocket 是阻塞的，也就是说如果对方发送消息,info= 新消息，循环走一圈，如果对方不发消息，for 这里
会 “卡住”，这里容易忽略
            request.websocket.send("你刚才对我说:%s"%info)
         print('这里其实不会被执行')
    else:
        # " 这里实现 http 连接逻辑"
        pass
```

11.4 场景 2，多 v 多聊天

上面的例子很简单吧，自己和自己聊天，多 v 多聊天代码和上面差不多!

将 request.websocket 看做普通对象，将所所有连接的 websocket 保存全局变量中，依次 send(msg) 即可.

```
wobsocket_map=dict()
@accept_websocket
def start_server_script(request):
    if request.is_websocket():
        wobsocket_map[id(request.websocket)]=request.websocket
        # " 这里实现 wbsocket 连接逻辑"
        for info in request.websocket:# 这里需要注意的是，这个 for 后面的对象，request.
↪websocket 是阻塞的，也就是说如果对方发送消息,info= 新消息，循环走一圈，如果对方不发消息，for 这里
会“卡住”，这里容易忽略
            [websocket.send("你刚才对我说:%s"%info) for websocket in wobsocket_map.
↪values()]
         print('这里其实不会被执行')
    else:
        # " 这里实现 http 连接逻辑"
        pass
```

可以实现效果，将 msg 发送给所有连接到此 websocket 的对象！

11.5 场景 3，视频播放

视频播放是一种类似死循环的处理逻辑

```
# 伪代码，不保证能跑，意思差不多
# 其他函数通过 next(),or .send() 调用此生成器，不断生成各个视频帧的 base64 编码（看不懂的话，百度
↪"yield"）
url='rtmp://xxxx'
@accept_websocket
def show_video(request):
    video=video_img_base64(url)
    base64=next(video)
    while base64 is not None: # 这个就是类似死循环的东西（当然这个并非真正死循环，但很多情况退出只
能放到循环里 break，这里只能采用 while True）
        request.websocket.send({'img_base64':base64})
        base64=video.send(None)

def video_img_base64(url):
    cap=cv2.VideoCapture(url)
    ret,frame=cap.read()
    while ret:
        yield base64(frame.tobytes()).decode('utf8')
        ret,frame=cap.read()
    else:
        yield None  # 调用方一旦收到 None，避免调用 next()    or send() 否则抛出异常，需要做异
常处理
```

(continues on next page)

(续上页)

11.6 场景 4，视频播放及控制

相比前面的例子，多了控制逻辑，那么问题来了，控制逻辑放哪里?

如果后台也可以向 js 那样，onmessage(xxx), 这样就简单多了，onmessage()，根据 message 修改一个类似全局变量的东西就行。但是并没有。

从场景 1 的例子可以看出，websocket 非常擅长处理 request-response 的情况。例子 3, 看到其也可以处理持续 response 推送的情况，那么如何实现类似异步里面交互式响应呢?

这里提供一个简单模板

```
send_queue=Queue()
@accept_websocket
def websocket_ctrl(request):
    if request.is_websocket:
        while True:# 由于使用同步方式处理异步，所以这里必然死循环
            if request.websocket.count_messages()>0:
                message=request.websocket.read()
                while message:
                    onmessage(message)
                    message=request.websocket.read()
             if not send_queue.empty:# 需向 send_queue 放东西，send 函数中会放 message
                 messsage=send_queue.pop()
                 while message:
                    message=request.websocket.send(message)
                    messsage=send_queue.pop()

def onmessage(message):
    pass # 这里可以放你想在收到消息时做的事情，类似异步方法的 onmessage，如果需要反馈修改，比如改变上一级的执行逻辑，则可以通过返回值的方式，传递给上一级调用者，让上一级调用者通过返回值调整自身执行逻辑

# 他人调用这个方法，将消息加入发送队列，在 websocket_ctrl 的循环中会取得，并发出
def send(message):
   send_queue.put(message)
```

注意：尽可能避免使用

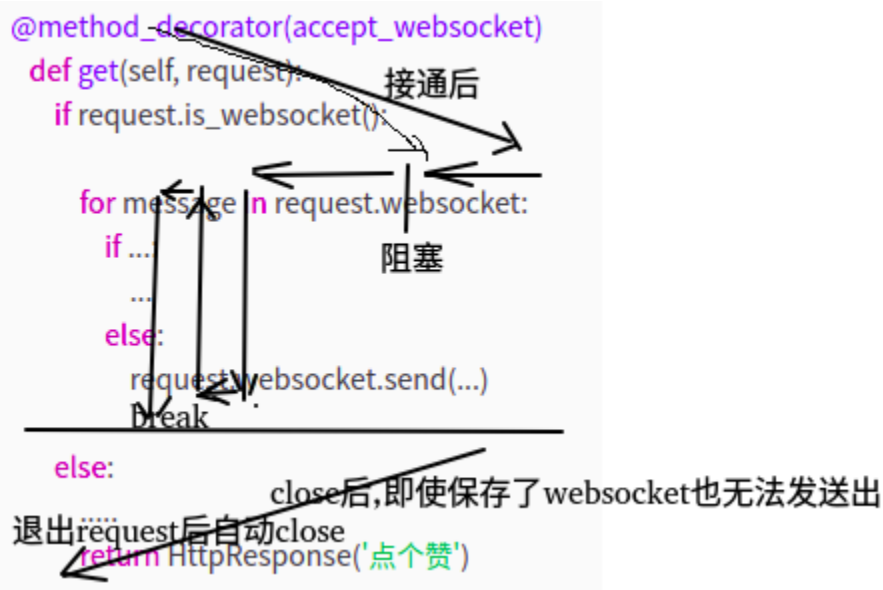
```
for message in request.websocket:
    print(message)
```

由于其会导致阻塞，特别强调这一点，因为大多数情况，我们看到 for 循环，会想当然以为”它很快会结束“,

其实未必。还有一点就是需要加异常处理，对方可能主动关闭连接，此时后台如果发送消息，会抛出异常。

11.7 总结

流程图



多重捕获会怎样？直观理解即可，只能有一个捕获到消息，不会重复捕获。

11.8 参考

https://www.520pf.cn/article/135.html

https://blog.csdn.net/xianailili/article/details/82180114

CHAPTER 12

django 入门进阶 12 信号

看起来简单，用起来简单。理解起来则未必容易。上学那会这一块就没整明白，这两天又查了下资料，算是基本弄懂了。

为何难以理解？个人感觉起名占了很大一部分，如果 signal 命名为“观察者”，“监控者”，“盯梢者”，就容易理解多了。其本质就是一种典型观察者模式。命名为信号，第一感觉是”信号量“类似的东西。

12.1 定义信号

```
import django.dispatch
pizza_done = django.dispatch.Signal(providing_args=["toppings", "size"])
```

12.2 发送信号

```
class PizzaStore(object):
'''

    def send_pizza(self, toppings, size):
        pizza_done.send(sender=self.__class__, toppings=toppings, size=size)
'''
```

12.3 断开信号

```
Signal.disconnect(receiver=None, sender=None, dispatch_uid=None)[source]
```

Signal.disconnect() 用来断开信号的接收器。和 Signal.connect() 中的参数相同。如果接收器成功断开，返回 True，否则返回 False。

参考文档：https://www.liujiangblog.com/course/django/170

https://www.cnblogs.com/qwj-sysu/p/4224805.html

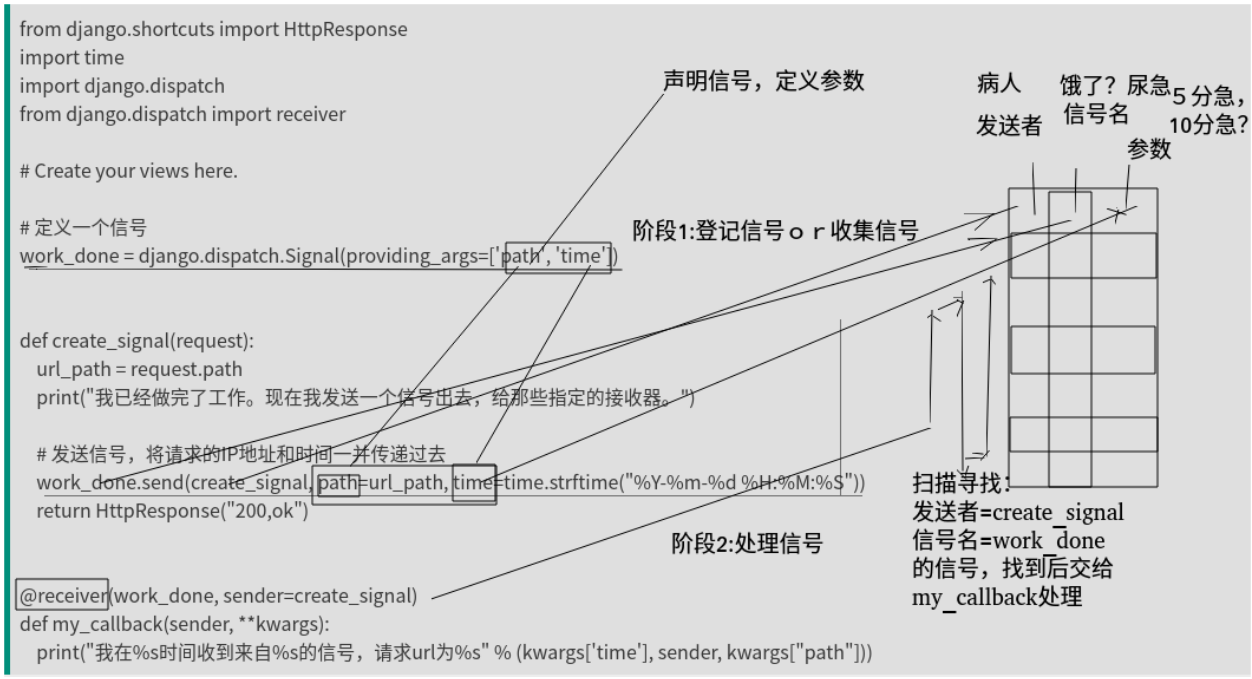
https://blog.csdn.net/qq_37049050/article/details/81299873

12.4 完整例子

```
from django.shortcuts import HttpResponse
import time
import django.dispatch
from django.dispatch import receiver

# Create your views here.

# 定义一个信号
work_done = django.dispatch.Signal(providing_args=['path', 'time'])


def create_signal(request):
    url_path = request.path
    print("我已经做完了工作。现在我发送一个信号出去，给那些指定的接收器。")

    # 发送信号，将请求的 IP 地址和时间一并传递过去
    work_done.send(create_signal, path=url_path, time=time.strftime("%Y-%m-%d %H:%M:%S
↪"))
    return HttpResponse("200,ok")


@receiver(work_done, sender=create_signal)
def my_callback(sender, **kwargs):
    print("我在%s时间收到来自%s的信号，请求 url 为%s" % (kwargs['time'], sender, kwargs[
↪"path"]))
```

12.5 如何理解

难以理解除了上面提到的命名问题外，还有一因素就是大家开发中可能较少使用观察者模式，就以上面的例子说下个人理解方式.



类比一下: **医院照顾病人模型**

定义信号:

尿急，饿了，渴了等等，各种不爽理由

发送信号 (阶段 1):

渴了 (work_done).send(张三（create_signal）, 十分渴 (path,time 等参数))，虽然感觉上有点小奇怪（如果是, 张三.send(渴了, 十分渴) 感觉会好点），但实际要素来讲，已经充分了，**谁 - 什么事 - 额外参数**。这些信息记录到”找茬登记册“上。

接收信号 (阶段 2):

专门处理张三渴了的护士 (receiver)，在”找茬登记册“反复寻找，**满足发送者＝张三，找事类型＝渴了，的事件**。然后根据参数，10 分渴 or5 分渴 ((path,time)) 决定带水杯还是水桶过去，让他**喝水 (my_callback)**。

如果有另一个 receive, 就是对应另一个护士, 另一个护士职责是”张三“＋”口渴“时就量他**体温 (my_callback)**。

这么做的好处是什么？解耦，非常明显，这里对操作序列的增加和减少非常轻松，只需改动注解就行，不用关注其他函数原有内部逻辑。

其实也就信号发送感觉上有点怪，**信号.send(发送者，参数)**。其实稍微思考下就能理解，其表达能力和 **发送者.send(信号，参数)** 是一样的。(如果采用后者，则需要对每个使用信号的对象都实现接口 send，侵入性较强)

其实 javaspring 有类似的工具，事件，可惜自己也未曾使用过，实际研发项目中也非常少见，了解了 django 的信号，回头看 spring 事件，好像 django 的信号反而更容易理解。

12.6 防止重复信号

为了防止重复信号，可以设置 dispatch_uid 参数来标识你的接收器，标识符通常是一个字符串，如下所示：

```
from django.core.signals import request_finished
request_finished.connect(my_callback, dispatch_uid="my_unique_identifier")
```

12.7 疑问

信号提供了断开接口，但为何需要要断开？何时断开，没找到答案，自己也没想明白，个人理解，如果断开是否造成同一个程序的二义性执行（断开了自然比没有断开少做事情了），这显然违反基本软件开发原则（可复现）

CHAPTER 13

django 入门进阶 13 异常之 makemigrations

makemigrations 是 django 中的常用操作，但是坑也比较多。

坑的主要原因，使用 django 的 manage.py makemigrations，django 会加载整个项目，而不仅仅是 models.py。而这会引发一系列问题。

13.1 01，初次初始化时使用了未（来得及）创建的表

比如：缓存类型对象查询到表，报错，进而导致无法执行 makemigrations（无法创建对象表）类似先有鸡先有蛋矛盾。

比如:

```
views.py
member_buffer=Member.objects.all()
```

本意是为了当做类似缓存使用（就这意思，代码对没对不重要），但是 makemigrations 之前是没有表 Member 的，

要创建表 Member 就好扫描到 views.py，然后就会报错。陷入悖论之中

解决：先注释掉 views.py 中的这一类查询，然后 makemigrations，最后再恢复回来

13.2 02，非守护线程维持进程导致无法正确退出

和第一个问题类似，不过藏的更深，表现为执行 makemigration 后，命令未正常结束，而是卡在那里。

```
views.py
class Handler():
    def __init__(self):
        xxxxx
        thread.Thread().start()
handler=Handler()
```

handler 此时是个单例，所以 Handler 里面的 __init__ 会被执行到。意味着如果 __init__ 存在 thread.Thread().start() 线程启动语句的话，那么也会被执行，创建出子线程。这会导致 makemigrations 结束后，无法自动退出，光标在最后一行输出后闪动（实际 makemigrations 以及成功完成），由于子线程有锁，会阻塞父进程的退出（父进程认为事情没干完，所以也阻塞在那里）。

这个问题迷惑性较强，一般第一反应都是 models.py 写的有问题。其实和 models.py 没关系。

本质原因是代码不规范导致的。__init__ 中是初始化部分，最好别包含线程启动等实际性执行操作。可以用 start 函数统一启动，这样也丰富编码规范。

13.3 03，django.db.utils.OperationalError: no such table.

```
python  manage.py  makemigrations
python   manage.py  migrate
```

数据库中有一个 django_migrations 数据表，找到你需要创建数据表的那个 name，然后 delete，再运行上面两个文件即可解决报错问题.

凡是 makemigrations 相关问题，有解决的套路

1，如果只修改了部分 app 则 makemigrations 和 migrate 都指定具体 app，避免全部 app 的扫描和更新。降低失败概率。

2，如果修改的 app 生成表失败，或者和预想不一致，则删除 django_migrations 数据表中有问题 app 相关记录，重新 makemigrations+migrate 生成（数据会丢失，线上别这么干就行了）

参考:https://blog.csdn.net/guyunzh/article/details/79487495

CHAPTER 14

django 入门进阶 14 其他异常报错等

django 常见报错的解决方案汇总

14.1 报错：djanog xxx doesn’t declare an explicit app_label and isn’t in an application

14.1.1 解决方案 01(INSTALLED_APPS)

报错举例：Runtime Error: Model class addresstest.models.address_info doesn’t declare an explicit app_label and isn’t in an application in INSTALLED_APPS.

解决: 需要在 settings.py 中的 INSTALLED_APPS 添加 app 包名称：addresstest

参考：django:doesn’t declare an explicit app_label and isn’t in an application in INSTALLED_APPS.:https://blog.csdn.net/wang603603/article/details/86932856

14.1.2 解决方案 02(django.contrib.sites)

在 settings.py 中增加

```
INSTALLED_APPS = [
        ...
        'django.contrib.sites',
]
```

参考:doesn’t declare an explicit app_label and isn’t in an application in INSTALLED_APPS.:https://www.cnblogs.com/crave/p/11043786.html

14.1.3 解决方案 03(import path)

补充 import 为完整路径

from models import User => from users.models import User

参考:Django 开发 BUG 记录—RuntimeError: Model class models.User doesn’t declare an explicit app_label and isn’ :https://www.pianshen.com/article/743343293/

14.1.4 解决方案 04(abs_path)

从绝对路径导包会报错, 改为相对路径

```
# from meiduo.apps.users import views

from . import views
urlpatterns=[]
```

参考:Django 项目一个小错误 doesn’t declare an explicit app_label:https://blog.csdn.net/u011519550/article/details/82354262

14.1.5 解决方案 05(我的方案)(pycharm auto import problem)

将 from xxx.yy.zz import 改为 from zz import 原因 python manage.py 启动时：python /xxx/yyy/zz.py。

会自动加入 sys.path：/xxx/yyy/

导致 pycharm 自动提示 import 路径不完全正确，从而 django 的 model 的 import 会尝试定位名为 yy.zz 的 app，实际 app_name 是 zz

简单来说就是 pycharm 自动提示 import 路径有误的问题。修改后 pycharm 提示代码有误，但实际可正确执行

CHAPTER 15

Indices and tables

- genindex
- modindex
- search