Django

相关知识点连接

Python 反射参考博客： https://www.cnblogs.com/jin-xin/articles/10325036.html

《Python正则表达式用法总结》、《Python数据结构与算法教程》、《Python函数使用技巧总结》、《Python字符串操作技巧汇总》、《Python入门与进阶经典教程》及《Python文件与目录操作技巧汇总》

内置过滤器参考文档：https://docs.djangoproject.com/en/1.11/ref/templates/builtins/#built-in-filter-reference

orm的更多操作参考博客： https://www.cnblogs.com/maple-shaw/articles/9323320.html

Django中ORM操作官方文档： https://docs.djangoproject.com/en/1.11/ref/models/querysets/

1. 简介目录:https://www.cnblogs.com/maple-shaw/p/9029086.html
2. 路由系统:https://www.cnblogs.com/maple-shaw/articles/9282718.html
3. 视图:https://www.cnblogs.com/maple-shaw/articles/9285269.html
4. 模板:https://www.cnblogs.com/maple-shaw/articles/9333821.html
5. ORM 字段和参数：https://www.cnblogs.com/maple-shaw/articles/9323320.html 查询操作:https://www.cnblogs.com/maple-shaw/articles/9403501.html 练习题：https://www.cnblogs.com/maple-shaw/articles/9414626.html
6. cookie和session：https://www.cnblogs.com/maple-shaw/articles/9502602.html
7. 中间件：https://www.cnblogs.com/maple-shaw/articles/9333824.html
8. ajax：https://www.cnblogs.com/maple-shaw/articles/9524153.html
9. form组件：https://www.cnblogs.com/maple-shaw/articles/9537309.html
10. auth模块：https://www.cnblogs.com/maple-shaw/articles/9537320.html

使用admin的步骤：

1. 创建一个超级用户

   python manage.py createsuperuser

   输入用户名 和 秘密

2. 在app下的admin.py中注册model

```
from django.contrib import admin
from app01 import models
# Register your models here.
admin.site.register(models.Person)
```

3. 地址栏输入/admin/

常用的装饰器

```
Python中常用的内置装饰器
@property    使调用类中的方法像引用类中的字段属性一样
@staticmethod    将类中的方法装饰为静态方法，即类不需要创建实例的情况下，可以通过类名直接引用。
@classmethod   类方法的第一个参数是一个类，是将类本身作为操作的方法。
显示原来的函数名
from functools import wraps
def a_new_decorator(a_func):
    @wraps(a_func)
    def wrapTheFunction():
        print("I am doing some boring work before executing a_func()")
        a_func()
        print("I am doing some boring work after executing a_func()")
     return wrapTheFunction

Django中常用装饰器
在自定义过滤器的
from  django import template
register = template.Library()   #  register名字不能错

# filter 最多一个参数
@register.filter
def str_upper(value,arg)
  return 'xxxxx'

#simple_tag
@register.simple_tag
def str_join(*args, **kwargs):
    return '*'.join(args) + '_'.join(kwargs.values())

#inclusion_tag
@register.inclusion_tag('page.html')需要传入一个HTML页面,将这个HTML页面在导入需要渲染模板中
def page(num):
    return {'num':range(1,num+1)}     必须是字典

Django视图中的装饰器
```

```
from django.utils.decorators import method_decorator

from django.views.decorators.csrf import csrf_protect,
csrf_exempt, ensure_csrf_cookie
# csrf_protect 需要csrf的校验
# csrf_exempt  不需要csrf的校验
# ensure_csrf_cookie    确保有csrf的cookie  csrftoken
```

CBV中加csrf_exempt的时候，只能加在dispatch上

```
 @method_decorator(wrapper, name='post') 加在当前方法上，只对
当前方法生效
 @method_decorator(wrapper, name='dispatch')加在dispatch方
法上，对所有的请求方式生效
class PublisherAdd(View):
     @method_decorator(wrapper)    加在dispatch方法上，对所有
的请求方式生效
     def dispatch(self, request, *args, **kwargs):
         ret = super().dispatch(request, *args, **kwargs)
         return ret

    @method_decorator(wrapper)     加在当前方法上，只对当前方法
生效
    def get(self, request):
        # 处理get请求的逻辑
        print(1, request.method)
        print(self.request is request)
        return render(request, 'publisher_add.html')
```

登录装饰器

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

Django里面的@login_required就是一个很好的例子。使用它只用一句代码就可以检查用户是否通过身份验证，并将未登录用户重定向到登录url

保护装饰器

有时需要保护一些视图，只允许某些用户组访问。这时就可以使用下面的装饰器来检查用户是否属于该用户组。

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

@group_required('admins','seller')
```

未登录用户

@anonymous_required

参考Django自带的 login_required 装饰器，但是功能是相反的情况，即用户必须是未登录的，否则用户将被重定向到 settings.py 中定义的地址。当我们想要已登录的用户不允许进入某些视图(比如登录)时，非常有用

只允许超级用户才能访问视图的装饰器

```
from django.core.exceptions import PermissionDenied
```

```
@superuser_only

ajax装饰器
这个装饰器用于检查请求是否是AJAX请求，在使用jQuery等Javascript框架时，这是一个非常有用的装饰器，也是一种保护应用程序的好方法

from django.http import HttpResponseBadRequest

@ajax_required

响应时间的装饰器
如果您需要改进某个视图的响应时间，或者只想知道运行需要多长时间，那么这个装饰器非常有用

@timeit

豁免csrf的装饰器
from django.views.decorators.csrf import csrf_exempt

@csrf_exempt

flask中常用的装饰器
（1）@app.before_request请求达到视图函数之前装饰器函数，正常状态务必return None
（2）@app.after_request响应到达客户端之前装饰器函数，正常状态被装饰函数必须定义一个形参来接收response,务必return response
（3）@app.errorhandler(错误状态码)错误捕获装饰器，装饰其中必须传入4xx或5xx的错误状态码，同时在被装饰函数中定义一个形参来接收错误信息error
（4）@app.template_global()和@app.template_filter()装饰器函数直接在模板中可以全局使用
（5）@app.route()路由视图装饰器，第一个参数为请求路径，其它关键字参数使用相见flask之route路由学习
```

命令部分

下载:

pip install django==1.11.25

pip install django==1.11.25 -i源(下载地址)

创建项目;

django-admin startproject 项目名

创建app

Python manage.py startapp app名称

启动项目

Python manage.py runsever

Python manage.py runsever 80 #127.0.0.1:80

数据库迁移命令

Python manage.py makemigrations

检查已注册app下的model.py 的变更记录

python manage.py migrate 将数据同步到数据库中

Django配置部分(settings文件)

根目录：

```
BASE_DIR =
os.path.dirname(os.path.dirname(os.path.abspath(__file__))
)
```

模板配置：

```
BASE_DIR =
os.path.dirname(os.path.dirname(os.path.abspath(__file__))
)
TEMPLATES=[
• 'DIRS': [os.path.join(BASE_DIR, 'templates')]
]
```

静态文件配置（包括**css**，**js**，图片等）

```
TIATIC_URL='/static/'        静态文件的别名，以它做开头
STATICFILES_DIRS = [
  os.path.join(BASE_DIR,'static')      css，js，图片等文件存放
的路径
]
```

注册**app**

```
INSTALLED_APPS=[
 'app01'
 'app01.apps.App01config'  #推荐写法                        注册
二选一
 ]
```

中间件

```
注释掉MIDDLEWARE中的csrf中间件或者
在form表单中加入{% csrf_token %}可提交POST请求
```

访问：

```
ALLOWED_HOSTS = ['*']都可以访问，可以改成特定的ip
```

Django终端打印SQL语句

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'handlers': {
        'console':{
            'level':'DEBUG',
            'class':'logging.StreamHandler',
        },
    },
    'loggers': {
        'django.db.backends': {
            'handlers': ['console'],
            'propagate': True,
            'level':'DEBUG',
        },
    }
}
```

文件说明：

```
app文件夹目录
    admin.py    admin管理后台    增删改查数据库
    apps.py     app相关
    models.py   数据库表相关
    views.py    写函数（逻辑）
项目文件夹下
  _init_.py       数据库替换代码
  settings        配置文件
  urls.py         路由文件       写路径和函数的对应关系

templates文件夹
  模板，需要渲染的网页
```

django处理请求的流程

1. 在地址栏中输入地址，回车发送一个请求；
2. wsgi服务器接收到http请求，封装request对象；
3. django根据请求的地址找到对应的函数，执行；
4. 函数执行的结果，django将结果封装成http响应的格式返回给浏览器。

发送请求的途径

1. 地址栏中输入地址和a标签 GET请求
2. form表单可以设置发送请求的方式

八种请求方法

- GET
- POST
- PUT
- DELETE
- HEAD
- OPTIONS
- TRACE
- CONNECT

  （get、post、put、delete、head、options、trace、connect）

request请求相关的内容

request.GET url上携带的参数 {} get()

request.POST POST请求提交的数据 {} get()只获取的请求的最后一个 getlist()获取请求的列表

request.method 请求方法 GET POST 获取请求方法

响应

HttpResponse(‘字符串’) 返回的字符串‘

render(request,'模板的文件名',{}) 返回的是 完整的页面

redirect（重定向的地址 ） 重定向

HTTP的简单理解

```
http请求由三部分组成，分别是：请求行、消息报头、请求正文
```

```
http是一种超文本传输协议，传输的数据都是未加密的，也就是显示在明面上的，是现在互联网上应用最为广泛的一种网络协议，相对来说不太安全，但是所需成本很小。http一般的端口号为80.

HTTP1.0是短连接、HTTP1.1是长连接。
  在HTTP1.1中，通讯流程可以概括为以下几步：
        1. 客户端与服务器建立TCP连接，一般使用80端口。
        2. 客户端向服务器发送请求。
        3. 服务器向客户端发送响应。
        4. 重复上述步骤。
        5. 通讯完毕，TCP连接断开。
http协议的无状态性质
http协议是无状态的，同一个客户端的这次请求和上次请求是没有对应关系，对http服务器来说，它并不知道这两个请求来自同一个客户端。 为了解决这个问题， Web程序引入了Cookie机制来维护状态.
Http消息结构
        请求行：GET www.baidu.com/  HTTP/1.1
        请求头：
        请求体：
```

Django使用数据库

orm的更多操作参考博客： https://www.cnblogs.com/maple-shaw/articles/9323320.html

Django中ORM操作官方文档： https://docs.djangoproject.com/en/1.11/ref/models/querysets/

通过orm(对象关系映射)使用数据库

类 ——》 表

对象 ——》 数据行（记录）

属性 ——》 字段

在数据库中创建一个数据库或者使用原来的库

```
create  databases  库名
```

配置数据库（settings文件）

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',     修改数据使用的引擎
        'NAME': 'a2',                  数据库的库名
        'HOST':'127.0.0.1',            ip
        'POST':3306,                   端口
        'USER':'root',                 用户
        'PASSWORD':'123'               密码
    }
}
```

一般在域settings同级的init文件中插入

```
import pymysql
pymysql.install_as_MySQLdb()    替换原来使用的数据库引擎
```

注：数据库迁移的时候出现一个警告

```
WARNINGS:
?: (mysql.W002) MySQL Strict Mode is not set for database connection 'default'
HINT: MySQL's Strict Mode fixes many data integrity problems in MySQL,
such as data truncation upon insertion, by escalating warnings into errors.
It is strongly recommended you activate it.
```

在配置中多加一个OPTIONS参数：Django官 网解释

```
'OPTIONS': {
    'init_command': "SET sql_mode='STRICT_TRANS_TABLES'"},
```

常用字段

```
models.AutoField(primary_key=True)      自增，primary_key=True设置主键
models.CharField(max_length) # 字符串 varchar(32)
models.TextField      文本类型
models.BooleanField  # 布尔值
models.DateTimeField(auto_now=True)  设置时间
models.IntegerField  # 10  -21亿  + 21亿
models.FloatField    #浮点型
models.DecimalField(max_digits=6,decimal_places=3)  # 10进制  999.999
models.ForeignKey     设置外键
models.ManyToManyField  生成第三张表
```

字段参数

```
null=True      数据库中该字段可以为空
blank = True    form表单输入时允许为空
default     默认值
unique      唯一
choices     可选的参数
verbose_name   可显示的名字
db_index=True   索引
db_column       列名
editabl   是否可编辑
```

ORM的查询操作

```
import os
os.environ.setdefault("DJANGO_SETTINGS_MODULE",
"about_orm.settings")
import django
django.setup()

from app01 import models

# all()    获取所有的数据   QuerySet   对象列表
ret = models.Person.objects.all()

# get()    获取一个对象   对象   不存在或者多个就报错
# ret = models.Person.objects.get(name='alexdsb')

# filter()    获取满足条件的所有对象   QuerySet   对象列表
ret = models.Person.objects.filter(name='alexdsb')

# exclude()   获取不满足条件的所有对象    QuerySet   对象列表
ret = models.Person.objects.exclude(name='alexdsb')

# values   QuerySet    [ {} ]     对象字典
# values()   不写参数   获取所有字段的字段名和值
# values('name','age')   指定字段  获取指定字段的字段名和值
ret = models.Person.objects.values('name', 'age')
# for i in ret:
#     print(i,type(i))

# values_list   QuerySet    [ () ]    对象元组
# values_list()   不写参数   获取所有字段的值
# values_list('name','age')   指定字段  获取指定字段的值

ret = models.Person.objects.values_list('age', 'name')

# for i in ret:
#     print(i, type(i))
```

```
# order_by    排序   默认升序    降序 字段名前加-    支持多个字段
ret = models.Person.objects.all().order_by('-age', 'pk')


# reverse   对已经排序的结果进行反转
ret = models.Person.objects.all().order_by('pk')
ret = models.Person.objects.all().order_by('pk').reverse()

# distinct mysql不支持按字段去重
ret = models.Person.objects.all().distinct()
ret =
models.Person.objects.values('name','age').distinct()

# count()   计数
ret = models.Person.objects.all().count()
ret = models.Person.objects.filter(name='alexdsb').count()

# first   取第一个元素   取不到是None
ret = models.Person.objects.filter(name='xxxx').first()

# last   取最后一个元素
ret = models.Person.objects.values().first()

# exists   是否存在   存在是True
ret = models.Person.objects.filter(name='xx').exists()
print(ret)


"""
返回对象列表   QuerySet
all()   获取所有
filter()   获取满足条件的
exclude()   获取不满足条件的
values()    [ {},{} ]   获取数据字段和值
values_list()    [ (),() ] 获取值
order_by()   排序 默认升序    降序 -     支持多个字段排序
reverse()    反转   对已经排序的对象列表反转
distinct()   去重


返回对象
get   获取一个满足条件的对象
first   获取第一个元素
last    获取最后一个元素

返回数字
count    统计次数

返回布尔值
exists()
"""
```

单表的双下划线

```
import os

os.environ.setdefault("DJANGO_SETTINGS_MODULE",
"about_orm.settings")
import django

django.setup()

from app01 import models

ret = models.Person.objects.filter(pk__gt=3)  # gt
greater than 大于
ret = models.Person.objects.filter(pk__lt=3)  # lt  less
than   小于

ret = models.Person.objects.filter(pk__gte=3)  # gt
greater than equal  大于等于
ret = models.Person.objects.filter(pk__lte=3)  # lt  less
than equal    小于等于

ret = models.Person.objects.filter(pk__range=[1, 4])  # 范
围(包括1,4)
ret = models.Person.objects.filter(pk__in=[1, 4])   # 成员
判断
ret = models.Person.objects.filter(name__in=
['alexdsb','xx'])

ret = models.Person.objects.filter(name__contains='x')   #
contains 包含  like
ret = models.Person.objects.filter(name__icontains='X')
# ignore contains 忽略大小写

ret = models.Person.objects.filter(name__startswith='X')
# 以什么开头
ret = models.Person.objects.filter(name__istartswith='X')
  # 以什么开头

ret = models.Person.objects.filter(name__endswith='dsb')
# 以什么开头
ret = models.Person.objects.filter(name__iendswith='DSB')
  # 以什么开头



ret = models.Person.objects.filter(phone__isnull=False)
# __isnull=False  不为null

ret = models.Person.objects.filter(birth__year='2020')
ret = models.Person.objects.filter(birth__contains='2020-
11-02')
print(ret)
```

```
'''
字段__条件
__gt    大于
__gte    大于等于
__lt   小于
__lte    小于等于
__in= []     成员判断
__range = [3,6]   范围   3-6

__contains =''     包含    like
__icontains =''    忽略大小写

__startswith =''    以什么开头
__istartswith =''    以什么开头


__endswith =''    以什么结尾
__endswith =''    以什么结尾


__year ='2019'

__isnull = True    字段值为null
'''
```

在**app**文件夹下的**models.py**中写类

```
 在models写类（继承models.Model）：

 class Publisher(models.Model):
    pid = models.AutoField(primary_key=True)    #指定主键
    name = models.CharField(max_length=32)  # varchar(32)


class Book(models.Model):
    title = models.CharField(max_length=32)  # varchar(32)
    pub =
models.ForeignKey('Publisher',on_delete=models.CASCADE)
#设置外键（一对多）

  on_delete在2.0版本中是必填的
  on_delete=models.CASCADE(级联删除)
      models.SET()
        models.SET_DEFAULT    default=值
      models.SET_NULL       null=True
          models.DO_NOTHING     什么都不做


class Author(models.Model):
    name = models.CharField(max_length=32)  # varchar(32)
```

```
    books = models.ManyToManyField(Book)  # 生成第三张表  不会在Author表中生成字段（多对多）
```

自定义字段和查询外键及多对多

```
from django.db import models


class MyCharField(models.Field):
    """
    自定义的char类型的字段类
    """

    def __init__(self, max_length, *args, **kwargs):
        self.max_length = max_length
        super(MyCharField, self).__init__(max_length=max_length, *args, **kwargs)

    def db_type(self, connection):
        """
        限定生成数据库表的字段类型为char，长度为max_length指定的值
        """
        return 'char(%s)' % self.max_length


# Create your models here.
class Person(models.Model):
    pid = models.AutoField(primary_key=True)  # 主键  pid pk
    name = models.CharField('用户名', max_length=32, db_column='username', help_text='不能是纯数字')  # varcher(32)
    age = models.IntegerField(default=18, )  # -21亿  -21亿
    birth = models.DateTimeField(auto_now=True)
    phone = MyCharField(max_length=11, null=True, blank=True)
    gender = models.BooleanField(choices=((True, '男'), (False, '女')))  # 1 男

    # price = models.DecimalField(max_digits=5,decimal_places=2)  # 999.99

    def __str__(self):
        return "{} {} {}".format(self.pk, self.name, self.age)

    class Meta:
        # 数据库中生成的表名称 默认 app名称 + 下划线 + 类名
```

```
        db_table = "person"

        # admin中显示的表名称
        verbose_name = '个人信息'

        # verbose_name加s
        verbose_name_plural = '所有用户信息'

        # # 联合索引
        # index_together = [
        #     ("name", "age"),  # 应为两个存在的字段
        # ]
        #
        # # 联合唯一索引
        # unique_together = (("name", "age"),)  # 应为两个
存在的字段


class Publisher(models.Model):
    name = models.CharField(max_length=32)

    def __str__(self):
        return "< Publisher object :{}
>".format(self.name)


class Book(models.Model):
    name = models.CharField(max_length=32)
    price =
models.DecimalField(max_digits=5,decimal_places=2)  #
999.99
    kucun = models.IntegerField(default=100)
    sale = models.IntegerField(default=0)
    publisher = models.ForeignKey('Publisher',
on_delete=models.CASCADE,
related_name='books',related_query_name='book',null=True,b
lank=True)

    def __str__(self):
        return "< Book object :{} >".format(self.name)


class Author(models.Model):
    name = models.CharField(max_length=32)
    books = models.ManyToManyField('Book',)

    def __str__(self):
        return "< Author object :{} >".format(self.name)
```

外键

```
```

```
# 基于对象的查询
# 正向查询
book_obj = models.Book.objects.get(pk=4)
print(book_obj.name)
print(book_obj.publisher)
print(book_obj.publisher_id)

# 反向查询
pub_obj = models.Publisher.objects.get(pk=1)
# 不指定related_name  表名小写_set
print(pub_obj)
print(pub_obj.book_set)   # 表名小写_set   关系管理对象
print(pub_obj.book_set.all())

# 指定related_name='books'  表名小写_set
print(pub_obj.books)
print(pub_obj.books.all())



# 基于字段的查询
# 查询“李小璐出版社”出版社的书
ret = models.Book.objects.filter(publisher__name='李小璐出
版社')

# 查询“绿光”书的出版社
# 没有指定related_name  表名小写
ret = models.Publisher.objects.filter(book__name='绿光')

# 指定related_name='books'
ret = models.Publisher.objects.filter(books__name='绿光')

# 指定related_query_name='book'
ret = models.Publisher.objects.filter(book__name='绿光')

print(ret)
```

多对多

```
# 基于对象的
author_obj = models.Author.objects.get(name='yinmo')
# print(author_obj.books)      # 关系管理对象
# print(author_obj.books.all())  # 所关联的书

book_obj = models.Book.objects.get(name='原谅你')
# print(book_obj.author_set)
# print(book_obj.author_set.all())

# print(book_obj.authors)
# print(book_obj.authors.all())

# 基于字段的查询
```

```
# print(models.Author.objects.filter(books__name='原谅你'))
# print(models.Book.objects.filter(author__name='yinmo'))

# all 查询所有的对象

# set（） 设置多对多的关系 [id，id] [对象，对象]
# author_obj.books.set([1,3])
# author_obj.books.set(models.Book.objects.filter(pk__in=
[2,4]))

# add 新增多对多的关系 id，id 对象，对象
# author_obj.books.add(1,3)
# author_obj.books.add(*models.Book.objects.filter(pk__in=
[1,3]))

# remove 删除多对多的关系 id，id 对象，对象
# author_obj.books.remove(1,3)
#
author_obj.books.remove(*models.Book.objects.filter(pk__in
=[2,4]))

# clear 删除所有多对多的关系
# author_obj.books.clear()
# author_obj.books.set([])

# create 创建一个所关联的对象并且和当前对象绑定关系
# ret = author_obj.books.create(name='yinmo的春
天',publisher_id=1)
# print(ret)

pub_obj = models.Publisher.objects.get(pk=1)
print(pub_obj.books.all())
#
print(pub_obj.books.set(models.Book.objects.filter(pk__in=
[3,4])))
#
print(pub_obj.books.add(*models.Book.objects.filter(pk__in
=[1])))

# 外键的关系管理对象需要有remove和clear 外键字段必须可以为空
#
print(pub_obj.books.remove(*models.Book.objects.filter(pk_
_in=[1])))
# print(pub_obj.books.clear())

# print(pub_obj.books.create(name="你喜欢的绿色"))
```

聚合和分组

```
# 所有的价格
from django.db.models import Max, Min, Avg, Count, Sum
```

```
分组 关键字:aggregate
ret = models.Book.objects.all().aggregate(Sum('price'))
ret =
models.Book.objects.all().aggregate(sum=Sum('price'),avg=A
vg('price'),count=Count('pk'))

ret = models.Book.objects.filter(pk__in=
[1,2]).aggregate(sum=Sum('price'),avg=Avg('price'),count=C
ount('pk'))

# print(ret,type(ret))

聚合  关键字:annotate（注释）
# 每一本书的作者个数
ret = models.Book.objects.annotate(count=Count('author'))

# 统计出每个出版社买的最便宜的书的价格
# 方式一
ret =
models.Publisher.objects.annotate(Min('book__price')).valu
es()
# for i in ret:
#     print(i)
# 方拾二
# ret =
models.Book.objects.values('publisher__name').annotate(min
=Min('price'))

# 统计不止一个作者的图书
ret =
models.Book.objects.annotate(count=Count('author')).filter
(count__gt=1)

# 查询各个作者出的书的总价格
ret =
models.Author.objects.annotate(Sum('books__price')).values
()

print(ret)
```

F和Q

```
from django.db.models import F, Q
F:做计算
ret = models.Book.objects.filter(sale__gt=F('kucun'))
ret = models.Book.objects.all().update(publisher_id=3)
ret =
models.Book.objects.filter(pk=1).update(sale=F('sale')*2+4
3)

Q:查询条件
# & 与  and
# | 或  or
# ~ 非  not
# Q(pk__gt=5)
ret = models.Book.objects.filter(Q(~Q(pk__gt=5) |
Q(pk__lt=3)) & Q(publisher_id__in=[1, 3]))
print(ret)
```

事务

```
from django.db import transaction
将try写在with外面，内部执行出错会回滚
try:
    with transaction.atomic():
        # 一系列的操作

 models.Book.objects.update(publisher_id=4).select_for_upd
ate()#加行级锁
        models.Book.objects.update(publisher_id=3)
        int('ss')
        models.Book.objects.update(publisher_id=2)
        models.Book.objects.update(publisher_id=5)

except Exception as e:
    print(e)
```

迁移数据库

```
python  manage.py  makemigrations   #  检查已经注册的APP下的
models.py的变更记录
python  manage.py  migrate   # 将变更记同步到数据库中
```

数据库的展示(查看)

```
from app01 import  models
models.Publisher.objects.all()   #  查询所有的数据
QuerySet 【对象】     对象列表
models.Publisher.objects.get(name='xxx')  #  查询有且唯一的
对象  没有或者是多个就报错
models.Publisher.objects.filter(name='xxx')   # 查询所有符合
条件的对象   QuerySet 【对象】     对象列表
```

```
for  i in models.Publisher.objects.all() :
  print(i)    #  Publisher object   __str__()
    print(i.name)


一对多:
        all_books = models.Book.objects.all()  # 对象列表
            for book in all_books:
                print(book)
                print(book.title,type(book.title))
                print(book.pub)      # 外键关联的对象
                print(book.pub_id)  # 关联对象的id  直接从数
据库查询


多对多:
    all_authors = models.Author.objects.all()
        for author in all_authors:
                print(author)
                print(author.name)
                print(author.books,type(author.books))  #
多对多的关系管理对象

 print(author.books.all(),type(author.books.all()))  # 关
系对象列表



        all_books = models.Book.objects.all()
        for book in all_books:
            book  # 书籍对象
            book.title
            book.author_set          # 多对多的关系管理对象
            book.author_set.all()    # 书籍关联的所有的作者对象
对象列表
```

数据库的新增

```
方式一:

models.Publisher.objects.create(name=pub_name,addr=pub_add
r)  # 对象
方式二:
  pub_obj = models.Publisher(name=pub_name,addr=pub_addr)
 # 内存中的对象  和数据库没关系
  pub_obj.save()  # 插入到数据库中    #Publisher:models中的类
名

一对多:
    models.Book.objects.create(title='xxxx',pub=关联的对象)
    models.Book.objects.create(title='xxxx',pub_id=id)
```

```
    obj =models.Book(title='xxxx',pub=关联的对象)
    obj.save()

多对多
    author_obj =
models.Author.objects.create(name=author_name)  # 插入作者
信息
    author_obj.books.set(book_id) # 设置作者和书籍的多对多的关
系
```

数据库删除

```
models.Publisher.objects.filter(pid=pid).delete()  # 对象列
表 删除
models.Publisher.objects.get(pid=pid).delete()  # 对象 删除
```

数据库的修改

```
方式一：
  models.Publisher.objects.create(name='xxxx',addr='xxxx')
方式二：
  obj = models.Publisher(name='xxxx',addr='xxxx')
  obj.save()  # 将修改提交的数据库

一对多
    obj = models.Book.objects.filter(id=id).first()
    obj.title = 'xxxxx'
    # obj.pub = pub_obj
    obj.pub_id = pub_obj.id
    obj.save()
```

多对多关系表的创建方式

1. Django自己创建

```
class Book(models.Model):
    title = models.CharField(max_length=32,
unique=True)
    pub = models.ForeignKey('Publisher',
on_delete=models.CASCADE)


class Author(models.Model):
    name = models.CharField(max_length=32)
    books = models.ManyToManyField('Book')  # 不会
生成字段生成第三张表
```

2. 半自动创建(自己手动创建表 + ManyToManyField 可以利用查询方法不能用set)

```
class Book(models.Model):
    title = models.CharField(max_length=32, unique=True)

class Author(models.Model):
    name = models.CharField(max_length=32)
    books =
models.ManyToManyField('Book',through='AuthorBook')  # 只
用django提供的查询方法  不用创建表 用用户创建的表AuthorBook

class AuthorBook(models.Model):
    book =
models.ForeignKey('Book',on_delete=models.CASCADE)
    author =
models.ForeignKey('Author',on_delete=models.CASCADE)
    date = models.CharField(max_length=32)
```

```
class Book(models.Model):
    title = models.CharField(max_length=32, unique=True)


class Author(models.Model):
    name = models.CharField(max_length=32)
    books =
models.ManyToManyField('Book',through='AuthorBook',through
_fields=['author','book'])  # 只用django提供的查询方法  不用创
建表 用用户创建的表AuthorBook


class AuthorBook(models.Model):
    book =
models.ForeignKey('Book',on_delete=models.CASCADE,null=Tru
e)
    author =
models.ForeignKey('Author',on_delete=models.CASCADE,relate
d_name='x1',null=True)
    tuiianren =
models.ForeignKey('Author',on_delete=models.CASCADE,relate
d_name='x2',null=True)
    date = models.CharField(ma_length=32)
```

3. 自己动手创建

```
class Book(models.Model):
    title = models.CharField(max_length=32, unique=True)


class Author(models.Model):
    name = models.CharField(max_length=32)


class AuthorBook(models.Model):
    book =
models.ForeignKey('Book',on_delete=models.CASCADE)
    author =
models.ForeignKey('Author',on_delete=models.CASCADE)
    date = models.CharField(max_length=32)
```

MVC和MTV的区别

- MVC:

  M: model 模型 和数据库交互

  V: view 视图 展示数据 HTML

  C: controller 控制器 业务流程 传递指令

- MTV：

  M: model 模型 ORM

  T：template 模板 HTML

  V : view 视图 业务逻辑

模板

```
render(request,'模板的文件名'，{k1:v1})

变量
{{  k1 }}

for循环
{% for i in list %}
  {{ forloop.counter }}
  {{ i }}
{% endfor %}

if判读

{% if 条件 %}
  满足条件后的结果
{% endif  %}


{% if 条件 %}
  满足条件后的结果
{% else %}
```

```
  不满足条件后的结果
{% endif  %}


{% if 条件 %}
  满足条件后的结果
{% elif 条件1 %}
  满足条件后的结果
{% else %}
  不满足条件后的结果
{% endif  %}


render（request，‘模板的文件名’，{ k1:v1 }）
{{ 变量 }}
{{ 变量.0 }}
{{ 变量.key }}
{{ 变量.keys }}     {{ 变量.values }}     {{ 变量.items }}

{{  变量.属性  }}  {{  变量.方法  }}

优先级：
  字典的key   >   方法或者属性   >   数字索引
```

过滤器

{{ value|过滤器名 }}

{{ value|过滤器名:参数 }}：两端不能有空格

内置过滤器

参考文档：https://docs.djangoproject.com/en/1.11/ref/templates/builtins/#built-in-filter-reference

```
  {{  变量|default:'默认值'  }}   变量不存在或者为空 显示默认值

  filesizeformat   显示文件大小  byte到PB

  {{ 2|add:'2'  }}   数字加法
  {{ 'a'|add:'b'  }}   字符串的拼接
  {{ [1,2]|add:[3,4]  }}   列表的拼接

  length  返回变量的长度

  slice切片
  {{ string|slice:'-1::-1' }}

  date  日期格式化
  {{ now|date:'Y-m-d H:i:s' }}

  可在settings设置:
```

```
•   USE_L10N =  False
•   DATETIME_FORMAT = 'Y-m-d H:i:s'
•   DATE_FORMAT = 'Y-m-d'

  safe    前面的内容不用转义   在模板中使用
  {{ a|safe }}  告诉django不需要进行转义
  from django.utils.safestring import mark_safe  # py文件中用
```

自定义过滤器

1. 在app下创建一个名叫templatetags的python包 （templatetags名字不能改）
2. 在包内创建py文件 （文件名可自定义 my_tags.py）
3. 在python文件中写代码：

```
from django import template
register = template.Library()  # register名字不能错
```

4. 写上一个函数 + 加装饰器

```
@register.filter
def str_upper(value,arg):  # 最多两个参数
    ret = value+arg
    return ret.upper()
```

使用：

在模板中

```
{% load my_tags  %}
{{ 'alex'|str_upper:'dsb'  }}
```

for循环

```
for循环支持嵌套
    {%  for i in list %}
        {{ forloop.counter }}
        {{ i }}
    {% endfor  %}

    {{ forloop.counter }}       循环的索引 从1开始
    {{ forloop.counter0 }}      循环的索引 从0开始
    {{ forloop.revcounter }}    循环的索引(倒叙) 到1结束
    {{ forloop.revcounter0 }}    循环的索引(倒叙) 到0结束
    {{ forloop.first }}         判断是否是第一次循环   是TRUE
    {{ forloop.last }}          判断是否是最后一次循环   是TRUE
    {{ forloop.parentloop }}    当前循环的外层循环的相关参数
```

```
    for...empty
     {% for tr in table %}
            <tr>
                {% for td in tr %}
                    <td > {{ td }} </td>
                {% endfor %}
            </tr>
        {% empty %}
            <tr> <td colspan="4" >没有数据</td> </tr>
  {% endfor %}


    for...empty
  {% for i  in list %}
    {{ i }} {{forloop }}
  {% empty %}
    参数
  {% endfor %}
```

if条件(可以和过滤器结合使用)

```
{% if alex.age == 84 %}
    当前已经到了第二个坎了
{% elif alex.age == 73 %}
    当前已经到了第一个坎了
{% else %}
    不在坎上，就在去坎上的路上
{% endif %}
```

注意：

```
1. 不支持算数运算   + - * / %
```

2. 不支持连续判断 10 > 5 > 1 false
3. if语句支持 and 、or、==、>、<、!=、<=、>=、in、not in、is、is not判断。

with起别名

```
方式一：
{% with p_list.0.name as alex_name %}
    {{ alex_name }}
    {{ alex_name }}
    {{ alex_name }}
    {{ alex_name }}
{% endwith %}
方式二：
{% with alex_name=p_list.0.name %}     没有空格
    {{ alex_name }}
    {{ alex_name }}
    {{ alex_name }}
{% endwith %}
```

csrf跨站请求伪造

```
{% csrf_token %}  form表单中有一个隐藏的input标签
name='csrfmiddlewaretoken'
value 随机字符串可以提交post请求     #放在form内有效
```

母版和继承

母版：

1. 就是一个HTML页面，提取到多个页面的公共部分；
2. 定义block块，留下位置，让子页面填充

继承：

1. 写{% extends ‘母版的名字’ %}

2. 重写block块

注意点：

1. {% extends 'base.html' %} 母版的名字有引号的 不带引号会当做变量
2. {% extends 'base.html' %} 上不要写内容，想显示的内容要写在block块中
3. 多定义点block块，有css js

组件

一小段HTML代码

```
{% include 'nav.html' %}  在页面中插入
```

静态文件

```
{% load static %}
{% static '相对路径' %}

{% get_static_prefix %}     获取静态文件的别名

<link rel="stylesheet" href="{% static 'css/bootstrap.css'
%}">      相对路径
<link rel="stylesheet" href="{% get_static_prefix
%}css/dsb.css">
```

simple_tag 和 inclusion_tag

自定义simple_tag：

1. 在已注册app下创建一个名为templatetags的python包;
2. 在包内创建py文件 (任意指定，my_tags.py)
3. 在py文件中写代码

```
from  django import template
register = template.Library()   #  register名字不能
错
```

4. 写函数 + 加装饰器

```
# filter 最多一个参数
@register.filter
def str_upper(value,arg)
  return 'xxxxx'

#simple_tag
@register.simple_tag
def str_join(*args, **kwargs):
    return '*'.join(args) +
'_'.join(kwargs.values())

#inclusion_tag
@register.inclusion_tag('page.html')需要传入一个
HTML页面,将这个HTML页面在导入需要渲染模板中
def page(num):
    return {'num':range(1,num+1)}     必须是字典

# page.html
<ul>
  {% for i in num %}
      <li>{{ i }}</li>
    {% endfor %}
</ul>
# 在templates写page.html
```

使用：

模板中使用：

```
过滤器使用
  {% load my_tags  %}            文件名
  {{ 变量|str_upper：‘参数’ }}     函数名
simple_tag使用
   {% str_join '1' '2' '3' k1='4' k2='5'  %}

inclusion_tag使用
  {% load xxxx %}    xxxx:放过滤器的py文件
  {% page 10 %}      pege:过滤器的名称
```

FBV和CBV（视图的两种写法）

FBV function based view————————基于函数的视图

CBV class based view————————基于类的视图

定义CBV

```
from django.views import View
class PublisherAdd(View):

    def get(self,request):
        # 处理get请求的逻辑
        return response

    def post(self,request):
        # 处理post请求的逻辑
        return response
```

对应关系

```
url(r'^publisher_add/', views.PublisherAdd.as_view()),
```

s_view的流程

1. 程序加载时，执行View中as_view的方法 ， 返回一个view函数。
2. 请求到来的时候执行view函数：
    a. 实例化类 ——》 self
    b. self.request = request
    c. 执行self.dispatch(request, *args, **kwargs)
        i. 判断请求方式是否被允许：
            i. 允许
            通过反射获取当前对象中的请求方式所对应的方法

handler = getattr(self, request.method.lower(), self.http_method_not_allowed)

ii. 不允许

handler = self.http_method_not_allowed

iii. 执行handler 获取到结果，最终返回给django装饰器补充

```
from functools import wraps
def wrapper(func):
    @wraps(func)      将被装饰的函数重新赋值    inner=a|b
    def inner(*args, **kwargs):
        # 之前
        ret = func(*args, **kwargs)
        # 之后
        return ret
    return inner

@wrapper  # a = wrapper(a)  inner
def a():
    """
    这是a
    :return:
    """
    pass

@wrapper  # b = wrapper(b)  inner
def b():
    """
      这是b
      :return:
      """
    pass
print(a.__name__)
print(a.__doc__)
print(b.__name__)
print(b.__doc__)
```

django中给视图加上装饰器

FBV直接给函数添加装饰器

```
from functools import wraps
def wrapper(func):
    @wraps(func)
    def inner(request, *args, **kwargs)
        # 之前
        start_time = time.time()
```

```
        ret = func(request, *args, **kwargs)
        print("执行时间为{}".format(time.time() -
start_time))
        # 之后
        return ret
    return inner

@wrapper
def book_list(request):
    # 获取所有的数据
    all_books = models.Book.objects.all()  # 对象列表
    return render(request, 'book.html', {'all_books':
all_books, })
```

CBV添加装饰器

```
from django.views import View
from django.utils.decorators import method_decorator

from django.views.decorators.csrf import csrf_protect,
csrf_exempt, ensure_csrf_cookie
# csrf_protect 需要csrf的校验
# csrf_exempt  不需要csrf的校验
# ensure_csrf_cookie   确保有csrf的cookie  csrftoken

CBV中加csrf_exempt的时候，只能加在dispatch上

 @method_decorator(wrapper, name='post') 加在当前方法上，只对
当前方法生效
 @method_decorator(wrapper, name='dispatch')加在dispatch方
法上，对所有的请求方式生效
class PublisherAdd(View):
     @method_decorator(wrapper)    加在dispatch方法上，对所有
的请求方式生效
     def dispatch(self, request, *args, **kwargs):
         ret = super().dispatch(request, *args, **kwargs)
         return ret

    @method_decorator(wrapper)     加在当前方法上，只对当前方法
生效
    def get(self, request):
        # 处理get请求的逻辑
        print(1, request.method)
        print(self.request is request)
        return render(request, 'publisher_add.html')

    def post(self, request):
        print(2, request.method)
        print(request.body)
        pub_name = request.POST.get('pub_name')
```

```
        pub_addr = request.POST.get('pub_addr')
        if not pub_name:
            error = '出版社名称不能为空'
        elif
models.Publisher.objects.filter(name=pub_name):
            error = '名称已存在'
        else:
            ret =
models.Publisher.objects.create(name=pub_name,

 addr=pub_addr)
            return redirect('/publisher_list/')
        return render(request, 'publisher_add.html',
{'error': error})
```

request请求

```
request.method    # 请求中使用的HTTP方法的字符串表示，全大写表
示。
request.GET   #  URL上携带的参数
request.POST  # POST请求提交的数据    enctype="application/x-
www-form-urlencoded"
request.FILES # 上传的文件    enctype="multipart/form-data"
request.path_info    # 路径信息   不包含IP和端口   也不包含查询参数
request.body    # 请求体，byte类型 request.POST的数据就是从body
里面提取到的
request.COOKIES  # cookies一个标准的Python 字典，包含所有的
cookie。键和值都为字符串
request.session  # session  一个既可读又可写的类似于字典的对象，
表示当前的会话。
          #只有当Django 启用会话的支持时才可用。完整的细节参见会话
的文档。
request.META    # 请求头的信息

request.get_full_path() # 路径信息   不包含IP和端口 包含查询参数
request.path    #一个字符串，表示请求的路径组件（不含域名）
request.is_ajax()    # 是否是ajax请求 布尔值
request.is_secure  #如果请求时是安全的，则返回True；即请求通是过
HTTPS 发起的。
```

response对象

```
from django.http.response import JsonResponse
HttpResponse('字符串')     # 返回字符串
render(request,'模板的文件名', {})    # 返回一个页面
redirect(地址) # 重定向    本质：  响应头 Location：  地址
JsonResponse({})    # 返回json数据    返回非字典类型safe=False
Content-Type: application/json
    序列化：字典、数字、字符串、列表、布尔值、null
      不能序列化时 safe = False
```

路由

URLconf

```
    urlpatterns = [
        url(r'^admin/', admin.site.urls),
        url(r'^blog/$', views.blogs),  # /blog/
        url(r'^blog/[0-9]{4}/\d{2}/$', views.blog), #
/blog/2019/10/
    ]

    url(正则表达式的字符串，视图函数，)
```

动态路由：用正则表达式匹配

分组（正则加括号）

```
在视图函数的参数中，需要位置参数
urlpatterns = [
    url(r'^blog/([0-9]{4})/(\d{2})/$', views.blog),
]
```

命名分组

```
(?P<名> 正则式 ) 格式
urlpatterns = [
    url(r'^blog/(?P<year>[0-9]{4})/(?<month>\d{2})/$',
views.blog),
]
```

命名分组和分组的区别：命名分组是关键字传参，分组是位置位置传参

include路由分发

```
from django.conf.urls import url, include
from django.contrib import admin
#可以将urls.py文件移到其他文件夹下，用include包起来就行
from app01 import views

urlpatterns = [
    url(r'^admin/', admin.site.urls),

    url(r'^app01/', include('app01.urls')),
    url(r'^app02/', include('app02.urls')),
```

```
    url(r'^file_upload/', views.file_upload),
    url(r'^get_data/', views.get_data),
]


app01 urls.py

from django.conf.urls import url
from app01 import views

urlpatterns = [

    url(r'^publisher_list/', views.publisher_list),
    # url(r'^publisher_add/', views.publisher_add),
    url(r'^publisher_add/', views.PublisherAdd.as_view()),
    # url(r'^publisher_add/', view),
    url(r'^publisher_del/(\d+)/', views.publisher_del),
    url(r'^publisher_edit/', views.publisher_edit),
]
```

```
可以传给视图函数关键字参数
urlpatterns = [
    url(r'^publisher_list/', views.publisher_list, {k1:
v1}),
```

URL的命名和反向解析

URL的命名

```
url(r'^publisher_list/', views.publisher_list,
name='publisher_list'),
```

URL的反向解析（加 引号）

模板

```
{% url 'publisher_list' "参数" %}     --》   解析生成完整的URL
路径   '/app01/publisher_list/参数'
```

views.py文件

```
from django.shortcuts import render, redirect,
HttpResponse, reverse
或者
from django.urls import reverse

reverse('pub_del',args=(1,))   --》
 '/app01/publisher_del/1/'
例:
return redirect(reverse('pub_del'))
```

分组和命名分组

URL的命名

```
 url(r'^publisher_del/(\d+)/',
views.publisher_del,name='publisher_del'),
```

URL的反向解析

```
url(r'^publisher_del/(\d+)/',
views.publisher_del,name='publisher_del'),
```

views.py文件

```
from django.shortcuts import render, redirect,
HttpResponse, reverse
from django.urls import reverse

reverse('pub_del',args=(1,))   --》
 '/app01/publisher_del/1/'
reverse('pub_del',kwargs={'pk':1})   --》
 '/app01/publisher_del/1/'
```

namespace名称空间

避免name重名

```
urlpatterns = [
    url(r'^app01/', include('app01.urls',
namespace='app01')),  # home   name=home
    url(r'^app02/', include('app02.urls',
namespace='app02')),  # home   name=home
]
```

反向解析生成URL

{% url 'namespace:name' % }

例：{% url '名字：URL的名字' %}

reverse('namespace:name') 例：reverse（名字：URL的名字）

cookie和session

cookie

定义：

保存在浏览器本地上的一组组键值对

为什么要有？

HTTP协议是无状态，每次请求之间都相互独立的，之间没有关系，没办法保存状态。

特性：

```
1. 服务器让浏览器 进行设置的,浏览器有权利不保存
```

2. 下次访问时自动携带相应的cookie 3. 保存在浏览器本地

在**Django**的操作

1. 获取cookie
    - request.COOKIES {}
    - request.COOKIES[] 或.get()
    - request.get_signed_cookie(key,salt='加密的盐',default='xxx') 获取加密的cookie
2. 设置set-cookie
    - response.set_cookie(key,value)
    - response.set_signed_cookie(key,value,salt="加密盐") 加密cookie
    - response.set_cookie(key,value,max_age='',path='/')
3. 删除cookie
    - response.delete_cookie(key)

session

定义

保存在服务器上的一组组键值对，必须依赖cookie

为什么要有？

1. cookie保存在浏览器上，不太安全
2. 浏览器对cookie的大小有一定的限制

session的流程：

1. 浏览器发送请求，没有cookie也没有session
2. 要设置session时，先根据浏览器生成一个唯一标识（session_key），存键值对，并且有超时时间
3. 返回cookie session_id = 唯一标识

在Django的使用

1. 设置

request.session[key] = value

2. 获取

   request.session[key]

   request.session.get(key)

3. 删除

```
request.session.pop(key)
request.session.delete()删除session保留cookie
request.session.flush()删除所有的session删除coolie
```

4. 其他

```
del request.session['k1']
# 会话session的key
request.session.session_key

# 将所有Session失效日期小于当前日期的数据删除
request.session.clear_expired()

# 删除当前会话的所有Session数据，不删除cookie
request.session.delete()

# 删除当前的会话数据并删除会话的Cookie。
request.session.flush()

# 设置会话Session和Cookie的超时时间
request.session.set_expiry(value)
    * 如果value是个整数，session会在些秒数后失效。
    * 如果value是个datatime或timedelta，session就会在这个时间后失效。
    * 如果value是0,用户关闭浏览器session就会失效。
    * 如果value是None,session会依赖全局session失效策略。
```

session的配置

```
from django.conf import global_settings

cookie名字
SESSION_COOKIE_NAME = 'sessionid'
超时时间
SESSION_COOKIE_AGE = 60 * 60 * 24 * 7 * 2
每次请求保存session
SESSION_SAVE_EVERY_REQUEST = True
浏览器关闭session数据失效
SESSION_EXPIRE_AT_BROWSER_CLOSE = True
存放session的位置,默认是数据库,
SESSION_ENGINE = 'django.contrib.sessions.backends.db'
```

中间件

定义：

中间件是一个用来处理Django的请求和响应的框架级别的钩子。它是一个轻量、低级别的插件系统，用于在全局范围内改变Django的输入和输出。每个中间件组件都负责做一些特定的功能。

本质：

中间件就是一个类，在全局范围内处理django的请求和响应。

自定义中间件

示例

```
from django.utils.deprecation import MiddlewareMixin  #需导入

class MD1(MiddlewareMixin):   #自己的类(中间件)

    def process_request(self, request):   #方法(函数名不能错)
        print("MD1里面的 process_request")

    def process_response(self, request, response):
        print("MD1里面的 process_response")
        return response
```

注册自定义的中间件

```
文件路径
 'app01.middlewares.my_middleware.MD2'
    文件路径.自己的类
```

process_request

```
from django.utils.deprecation import MiddlewareMixin

class MD1(MiddlewareMixin):
    def process_request(self, request):
        print("MD1里面的 process_request")
class MD2(MiddlewareMixin):
    def process_request(self, request):
        print("MD2里面的 process_request")
```

process_request(self,request)

执行时间：

在视图函数之前

执行顺序：

按照注册的顺序 顺序执行

参数：

request： 请求的对象，和视图函数是同一个

返回值：

None：正常流程

HttpResponse:之后中间件的process_request、路由、process_view、视图都不执行，执行执行当前中间件对应process_response方法，接着倒序执行之前的中间件中的process_response方法。

process_response

```
from django.utils.deprecation import MiddlewareMixin

class MD1(MiddlewareMixin):

    def process_request(self, request):
        print("MD1里面的 process_request")

    def process_response(self, request, response):
        print("MD1里面的 process_response")
        return response
class MD2(MiddlewareMixin):
    def process_request(self, request):
        print("MD2里面的 process_request")

    def process_response(self, request, response):
        print("MD2里面的 process_response")
        return response
```

process_response(self, request, response)

执行时间：

在视图函数之后

执行顺序：

按照注册的顺序 倒序执行

参数：

request： 请求的对象，和视图函数是同一个

response： 响应对象

返回值：

HttpResponse: 必须返回

process_view

```
from django.utils.deprecation import MiddlewareMixin

class MD1(MiddlewareMixin):
    def process_request(self, request):
        print("MD1里面的 process_request")
    def process_response(self, request, response):
        print("MD1里面的 process_response")
        return response
    def process_view(self, request, view_func, view_args,
view_kwargs):
        print("-" * 80)
        print("MD1 中的process_view")
        print(view_func, view_func.__name__)
class MD2(MiddlewareMixin):
    def process_request(self, request):
        print("MD2里面的 process_request")
    def process_response(self, request, response):
        print("MD2里面的 process_response")
        return response
    def process_view(self, request, view_func, view_args,
view_kwargs):
        print("-" * 80)
        print("MD2 中的process_view")
        print(view_func, view_func.__name__)
```

process_view(self, request, view_func, view_args, view_kwargs)

执行时间：

在路由匹配之后，在视图函数之前

执行顺序：

按照注册的顺序 顺序执行

参数：

request： 请求的对象，和视图函数是同一个

view_func：视图函数

view_args: 给视图传递的位置参数

view_kwargs： 给视图传递的关键字参数

返回值：

None： 正常流程

HttpResponse: 之后中间件的process_view、视图都不执行，直接执行最后一个中间件process_response，倒序执行之前中间件的process_response方法

process_exception

```
from django.utils.deprecation import MiddlewareMixin
class MD1(MiddlewareMixin):
    def process_request(self, request):
        print("MD1里面的 process_request")
    def process_response(self, request, response):
        print("MD1里面的 process_response")
        return response
    def process_view(self, request, view_func, view_args,
view_kwargs):
        print("-" * 80)
        print("MD1 中的process_view")
        print(view_func, view_func.__name__)
    def process_exception(self, request, exception):
        print(exception)
        print("MD1 中的process_exception")
class MD2(MiddlewareMixin):
    def process_request(self, request):
        print("MD2里面的 process_request")
    def process_response(self, request, response):
        print("MD2里面的 process_response")
        return response
    def process_view(self, request, view_func, view_args,
view_kwargs):
        print("-" * 80)
        print("MD2 中的process_view")
        print(view_func, view_func.__name__)
    def process_exception(self, request, exception):
```

```
        print(exception)
        print("MD2 中的process_exception")
```

process_exception(self, request, exception)

如果视图函数中无异常，process_exception方法不执行

执行时间：

在视图函数出错之后执行

执行顺序：

按照注册的顺序 倒序执行

参数：

request： 请求的对象，和视图函数是同一个

exception：报错的对象

返回值：

None： 自己没有处理，交给下一个中间件处理，所有的中间件都没有处理，django处理错误。

HttpResponse: 之后中间件的process_exception，直接执行最后一个中间件process_response，倒序执行之前中间件的process_response方法

process_template_response

```
class MD1(MiddlewareMixin):
    def process_request(self, request):
        print("MD1里面的 process_request")
    def process_response(self, request, response):
        print("MD1里面的 process_response")
        return response
    def process_view(self, request, view_func, view_args,
view_kwargs):
        print("-" * 80)
        print("MD1 中的process_view")
        print(view_func, view_func.__name__)
    def process_exception(self, request, exception):
        print(exception)
        print("MD1 中的process_exception")
        return HttpResponse(str(exception))
    def process_template_response(self, request,
response):
        print("MD1 中的process_template_response")
        return response
```

```
class MD2(MiddlewareMixin):
    def process_request(self, request):
        print("MD2里面的 process_request")
    def process_response(self, request, response):
        print("MD2里面的 process_response")
        return response
    def process_view(self, request, view_func, view_args,
view_kwargs):
        print("-" * 80)
        print("MD2 中的process_view")
        print(view_func, view_func.__name__)
    def process_exception(self, request, exception):
        print(exception)
        print("MD2 中的process_exception")
    def process_template_response(self, request,
response):
        print("MD2 中的process_template_response")
        return response
```

process_template_response(self,request,response)

它的参数，一个HttpRequest对象，response是TemplateResponse对象（由视图函数或者中间件产生）。

process_template_response是在视图函数执行完成后立即执行，但是它有一个前提条件，那就是视图函数返回的对象有一个render()方法（或者表明该对象是一个TemplateResponse对象或等价方法）。

执行时间：

当视图函数返回一个TemplateResponse对象

执行顺序：

按照注册的顺序 倒序执行

参数：

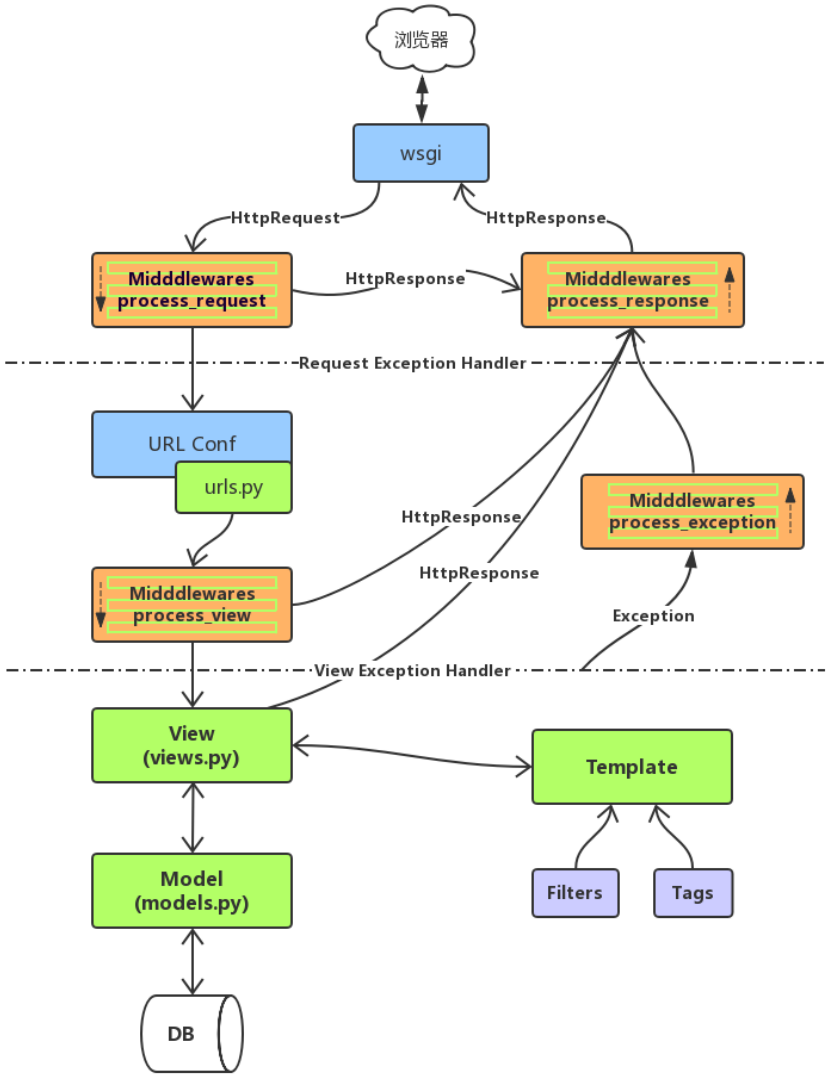
request： 请求的对象，和视图函数是同一个

response：响应的对象

返回值：

HttpResponse: 必须返回

Django请求流程图



AJAX

js技术，发请求

特点

1. 异步
2. 局部刷新
3. 数据量小

简单使用

使用jquery发送ajax请求

```
先引入jQuery
$.ajax({
    url:'' ,  //  请求发送的地址
    type: 'post',   //  请求方式
    data: {} ,  //  数据
    success:function (res){}      //  响应成功之后执行的回调函数  res返回的响应体

})
```

上传文件

```
模板:
<script
src="https://cdn.bootcss.com/jquery/3.4.1/jquery.js">
</script>
<script>

    $('button').click(function () {

        var form_data = new FormData();
        form_data.append('k1','v1');
        form_data.append('f1',$('#f1')[0].files[0]);

        $.ajax({
            url :'/file_upload/',
            type:'post',
            data:form_data,  // { k1:v1 }
            processData: false,  //   不需要处理数据
            contentType: false,  //   不需要contentType请求头
            success:function (ret) {
                console.log(ret)
            }

        })

    })

</script>


view视图函数
def file_upload(request):
    if request.method == 'POST':
        f1 = request.FILES.get('f1')
        with open(f1.name,'wb') as f:
            for i in f1.chunks():
                f.write(i)

        return HttpResponse('ok')
```

```
    return render(request, 'file_upload.html')
```

ajax通过Django的csrf的校验

前提：必须要有csrftoken的cookie

方式一：

给data添加csrfmiddlewaretoken键值对

```
 $.ajax({
            'url': '/calc/',
            'type': 'post',
      'data': {
                'csrfmiddlewaretoken':
$('[name="csrfmiddlewaretoken"]').val(),
                }
```

方式二：

给headers添加x-csrftoken的键值对

```
$('#b2').click(function () {
        // 点击事件触发后的逻辑
        $.ajax({
            'url': '/calc2/',
            'type': 'post',
            headers: {'x-csrftoken':
$('[name="csrfmiddlewaretoken"]').val()},
            'data': {
                'k1': $('[name="ii1"]').val(),
                'k2': $('[name="ii2"]').val(),
            },
            success: function (ret) {
                $('[name="ii3"]').val(ret)
            }
        })
    })
```

方式三

导入文件，文件内容

```
function getCookie(name) {
    var cookieValue = null;
    if (document.cookie && document.cookie !== '') {
        var cookies = document.cookie.split(';');
        for (var i = 0; i < cookies.length; i++) {
            var cookie = jQuery.trim(cookies[i]);
```

```
            // Does this cookie string begin with the name
we want?
            if (cookie.substring(0, name.length + 1) ===
(name + '=')) {
                cookieValue =
decodeURIComponent(cookie.substring(name.length + 1));
                break;
            }
        }
    }
    return cookieValue;
}

var csrftoken = getCookie('csrftoken');


function csrfSafeMethod(method) {
  // these HTTP methods do not require CSRF protection
  return (/^(GET|HEAD|OPTIONS|TRACE)$/.test(method));
}

$.ajaxSetup({
  beforeSend: function (xhr, settings) {
    if (!csrfSafeMethod(settings.type) &&
!this.crossDomain) {
      xhr.setRequestHeader("X-CSRFToken", csrftoken);
    }
  }
});
```

csrf相关装饰器

```
from django.views.decorators.csrf import
ensure_csrf_cookie,csrf_exempt,csrf_protect

# ensure_csrf_cookie  确保有cookie
# csrf_exempt   不需要进行csrf的校验
# csrf_protect  需要进行csrf的校验

注意:   csrf_exempt CBV的时候只能加载dispatch方法上
```

form组件

form组件

1. 提供input框
2. 能对数据做校验
3. 返回错误提示

定义：

```
from django import forms
class RegForm(forms.Form):
    username = forms.CharField(label='用户名')
    password = forms.CharField(label='密码')
```

使用：

```
函数
def register2(request):
    if request.method == 'POST':
        form_obj = RegForm(request.POST)

        if form_obj.is_valid():  # 校验数据
            # 插入数据库
            print(form_obj.cleaned_data)
            print(request.POST)

            return HttpResponse('注册成功')
    else:
        form_obj = RegForm()
    return render(request, 'register2.html', {'form_obj':
form_obj})


模板
{{ form_obj.as_p }}   # 展示所有的字段

{{ form_obj.username }}           #  生成input框
{{ form_obj.username.label }}     #  中文提示
{{ form_obj.username.id_for_label }}    # input框的id
{{ form_obj.username.errors }}     # 该字段的所有的错误
{{ form_obj.username.errors.0 }}    # 该字段的第一个的错误

{{ form_obj.errors }}    # 该form表单中所有的错误常用
```

常用字段

```
CharField      # 文本输入框
ChoiceField    # 单选框
MultipleChoiceField  # 多选框
```

字段参数

```
    required=True,                  是否必填
    widget=None,                    HTML插件
    label=None,                     用于生成Label标签或显示内容
    initial=None,                   初始值
    error_messages=None,            错误信息 {'required': '不能
为空', 'invalid': '格式错误'}
    disabled=False,                 是否可以编辑
    validators=[],                  自定义校验器
```

校验

自定义校验规则

1. 写函数

```
from django.core.exceptions import ValidationError
def checkusername(value):
  # 通过校验规则  什么事都不用干
  # bu通过校验规则  抛出异常ValidationError
    if models.User.objects.filter(username=value):
        raise ValidationError('用户名已存在')

username = forms.CharField(

        validators=[checkusername,]）
```

使用内置的校验器

```
from django.core.validators import RegexValidator

phone =
forms.CharField(min_length=11,max_length=11,validators=
[RegexValidator(r'^1[3-9]\d{9}$','手机号格式不正确')])
```

局部钩子和全局钩子

```
    def clean(self):
        # 全局钩子    校验多组数据44444
        # 通过校验    返回self.cleaned_data
        # 不通过校验    抛出异常
        password = self.cleaned_data.get('password')
        re_password = self.cleaned_data.get('re_password')
        if password == re_password:
            return self.cleaned_data
        else:
            self.add_error('password','两次密码不一致!!')
            raise ValidationError('两次密码不一致')

    def clean_username(self):
        # 局部钩子   校验单组数据
```

```
        # 通过校验    返回当前字段的值
        # 不通过校验    抛出异常
        value = self.cleaned_data.get('username')
        if models.User.objects.filter(username=value):
            raise ValidationError('用户名已存在')
        return value
```

django_debug_toolbar

博客：https://www.cnblogs.com/maple-shaw/articles/7808910.html

django缓存

参考博客：https://www.cnblogs.com/maple-shaw/articles/7563029.html

- 配置

```
CACHES = {
    'default': {
        'BACKEND':
'django.core.cache.backends.locmem.LocMemCache',
        'LOCATION': 'unique-snowflake',
        'TIMEOUT': 300,  # 缓存超时时间（默认300，
None表示永不过期，0表示立即过期）
        'OPTIONS': {
            'MAX_ENTRIES': 300,  # 最大缓存个数（默
认300）
            'CULL_FREQUENCY': 3,  # 缓存到达最大个数
之后，剔除缓存个数的比例，即：1/CULL_FREQUENCY（默认3）
        },
    }
}
```

- 使用

```
1. 应用到视图
from django.views.decorators.cache import
cache_page

@cache_page(10)
def index(request):
    # return HttpResponse('index')
    print('index')
    return render(request, 'index.html')

2.全站缓存
位置不能变
MIDDLEWARE = [
```

```
 'django.middleware.cache.UpdateCacheMiddleware',
    '其他的中间件',

 'django.middleware.cache.FetchFromCacheMiddlewar
e',
]

3.局部缓存，使用到模板
{% load cache %}

{% cache 5  'sdadlkjw'   %}
时间：{{ now }}
{% endcache %}

4。使用redis做缓存
pip install django-redis
参考博客：https://django-redis-
chs.readthedocs.io/zh_CN/latest/
 CACHES = {
    "default": {
        "BACKEND":
"django_redis.cache.RedisCache",
        "LOCATION": "redis://127.0.0.1:6379/1",
        "OPTIONS": {
            "CLIENT_CLASS":
"django_redis.client.DefaultClient",
        }
    }
}


5.作为 session backend 使用配置
Django 默认可以使用任何 cache backend 作为 session
backend，将 django-redis 作为 session 储存后端不用安
装任何额外的 backend
SESSION_ENGINE =
"django.contrib.sessions.backends.cache"
SESSION_CACHE_ALIAS = "default"
```

信号

参考博客：https://www.cnblogs.com/maple-shaw/articles/7563029.html

```
# __init__.py

from django.db.models.signals import pre_save, post_save


def logger(sender, **kwargs):
    print('xxxxxx ')
```

```
    print(sender)
    print(kwargs)


from django.db.models.signals import pre_migrate,
post_migrate

post_save.connect(logger)
pre_migrate.connect(logger)
```

django执行原生SQL

- 方式一 row()

```
解释：执行原始sql并返回模型
说明：依赖model多用于查询

raw()方法执行原生sql语句：
raw()方法执行原生sql(调用的类名不区分是谁，只要存在均可执行)

不会对传入的SQL语句进行检查.raw()。Django期望该语句将从数据库中返回一组行，但是不执行任何操作。如果查询不返回行，则将导致（未知）错误。
例：
  ret=models.Book.objects.raw('select * from
app01_book')
  # ret=models.Publish.objects.raw('select * from
app01_book')
  for book in ret:
      print(book.book_name)
      # print(book.__dict__)
```

- 方式二 .extra函数

```
解释：结果集修改器，一种提供额外查询参数的机制
说明：依赖model模型
用在where后：

Book.objects.filter(publisher_id="1").extra(where
=["title='python学习1'"])

用在select后

Book.objects.filter(publisher_id="1").extra(selec
t={"count":"select count(*) from hello_book"})


  Entry.objects.extra(select={'new_id': "select
col from sometable where othercol > %s"},
select_params=(1,))
```

```
    Entry.objects.extra(where=['headline=%s'],
params=['Lennon'])
    Entry.objects.extra(where=["foo='a' OR bar =
'a'", "baz = 'a'"])
    Entry.objects.extra(select={'new_id': "select
id from tb where id > %s"}, select_params=(1,),
order_by=['-nid'])
```

- 方式三 执行自定义SQL

```
解释：利用游标执行
导入：from django.db import connection
说明：不依赖model

用法：
from django.db import connection
       cursor = connection.cursor()
       #插入
       cursor.execute("insert into
hello_author(name) values('xiaol')")
       #更新
       cursor.execute("update hello_author set
name='xiaol' where id=1")
       #删除
       cursor.execute("delete from hello_author
where name='xiaol'")
       #查询
       cursor.execute("select * from
hello_author")
       #返回一行
       raw = cursor.fetchone()
       print(raw)
       # #返回所有
       # cursor.fetchall()
```

django分库分表

django读写分离

补充

1. 详细描述是jsonp实现机制？

```
参考博客
https://www.cnblogs.com/dowinning/archive/2012/04
/19/json-jsonp-jquery.html
```

2. django的orm如何通过数据自动化生成models类？

```
Python manage.py inspectdb

python manage.py inspectdb > appname/models.py
```

3. django中如何设置缓存？

```
参考博客：
https://www.cnblogs.com/wupeiqi/articles/5246483.
html
```

4. django中信号的作用？

```
同上
```

5. django中如何设置读写分离

```
参考博客：
https://www.cnblogs.com/fat39/p/10044712.html
```

6. django操作ORM

参考博客 https://www.cnblogs.com/sss4/p/7070942.html