Django框架

django 基础命令行

```
django-admin startproject mysite
cd mysite
python manage.py runserver
python manage.py startapp app # 创建后必须要注册

python manage.py makemigrations # 数据库迁移
python manage.py migrate # 同步到数据库
```

settings设置

```
DEBUG = True
# 模板路径配置
TEMPLATES = [
    DIR:[os.path.join(BASE_DIR, 'templates')]
]

INSTALLED_APPS= [
'app',  # 注册APP 简写
'app.apps.AppConfig' # 全写
]
# 静态文件配置
STATICFILES_DIRS = [
    os.path.join(BASE_DIR, 'static'),
]
DATABASES = [
    'default':{
        'ENGINE': 'django.db.backends.mysql',
        'NAME': '数据库名',
        'USER': 'root',
        'PASSWORD':'root',
        'HOST': '127.0.0.1',
        'PORT': '3306',
        'CHARSET': 'utf8'
    }
]

APPEND_SLASH = False # 取消路由自动加斜杠 默认为True

# 查看sql语句
LOGGING = {
    'version':1,

    'disable_existing_loggers':False,
```

```
    'handlers':{
        'console':{
            'level':'DEBUG',
            'class':'logging.StreamHandler',
        },
    }
    'loggers':{
        'django.db.backends':{
            'handles':['console'],
            'propagate': True,
            'level':'DEBUG',
        },
    }
}
```

django 三部曲

```
url.py
urlpatterns = [
    url(r'^index/', views.index)
]

#############################################################################

views.py
from django.shurtcuts import render, HttpResponse
def index(request):
    pass
    # return HttpResponse('.........')
    return render(request, '123.html', locals())
    # return redirect('url') # 写自己的 只写 后缀

#############################################################################

models.py
import
class User(models.Model):
    CharFiled 必须指定max_length参数

'''
参数设置
max_length参数 verbose_name 指定字段名 null可以为空 default 默认值
primary_key 设置主键
'''
```

模板语法传值

- {{}} 变量相关
- {%%} 逻辑相关

基本数据类型都可传 函数可传，但是不支持传参 传类时自动调用（实例化）

django 模板语法取值 是固定格式 只能使用 .

- 过滤器

  {{ 变量名| 过滤器：参数}}

  {{ str|length }} {{ str|default: '前无值时使用' }} {{ file_size|filesizeformat }} 文件大小

  {{ currrent_time|date:'Y-m-d H:i:s }}日期格式化 {{ l|slice:'0:4:2' }} 切片 支持步长

  {{ info|truncatechars:9 }} 切取字符 包含三个点 {{ egl|truncatewords:9 }} 切去单词 不包含三个点

  {{ msg|cut:' ' }} 移除指定字符 {{ l|join:'$' }}拼接 {{ n|add:10}} {{ |safe }} 取消转义

- {% for i in list %}

  {{ forloop}}

  {% endfor %}

- {% if b %}

  {% elif b %}

  {% else %}

  {% endif %}

- {% with ... as ...%} 起别名

自定义过滤器 标签 inclusion_tag

```
# 第一步 应用下 创建一个 templatetags文件夹
# 第二步 文件夹下新建一个.py文件
# 第三步

from django import template
register = template.Library()

@register.inclusion_tag('xxx.html')
def func()
    pass
```

{% load mytag%} {{ |baby }} 过滤器 最多俩个参数

自定义标签

from django import template register = template.Library()

@register.simpe_tag(name='plus') def func() pass

{% plus 参数1 参数2 参数3 ... %} #标签 可接受多个参数

自定义inclusion_tag

''' 先定义一个方法 在页面上调用该方法 该方法会生成一些数据然后传递给一个html页面 之后将渲染好的结果放到调用的位置 ''' @register.inclusion_tags('index.html') def func() pass

{% func 参数 %}

模板的继承

```
# 在原模板中 表示可被修改的部分
{% block 名称 %}
    被修改的内容
{% endblock %}

{% extends 'index.html'%}
{% block 名称 %}
    修改后的内容
{% endblock %}

# 模板的导入（了解）
将页面的局部作为一个模块导入
{% include 'index.html' %}
```

--static --css --js --image

{% load static %} # 加载静态文件 {% static '文件路径' %}

```


### form表单

```html
action  不写 提交当当前路由
        全写
        只写后缀
method get post
```

request方法

```
request.method  # 获取请求方法
request.POST # 获取post提交的数据
request.POST.get() # 只获取列表最后一个元素
request.POST.getlist() # 获取列表所有元素
request.GET # 获取get提交的数据  （提交的数据有大小限制 4kb左右）
request.GET.get() # 只获取列表最后一个元素
request.GET.getlist() # 获取列表所有元素
request.FILES # 获取文件
request.path # 只获取路由
request.path_info # 只获取路由
request.get_full_path() # 获取路由 + 参数
request.body # 获取二进制文件的字节串
```

表单查询

- 单表查询
```

```
# 增
ctime=datetime.dataetime.now()
models.User.objects.create(name='name', age='age', register_time=ctime)
# user = models.User((name='name', age='age', register_time=ctime)
# user.save()
# 删
res = models.User.objects.filter(pk=2).delete() # pk可以自动查找当前表的主键
# user = models.User.objects.filter(pk=2).first()
# user.delete()

# 修改
models.User.objects.filter(pk=4).update(name='newname')

# 必知必会13条
all() # 查询所有数据
filter() # 直接拿数据对象 不存在返回空none
get() # 直接拿数据对象 不存在报错
first() # 获取Querylist中的第一条元素
values() # 可以指定获取的数据字段 返回结果 [{},{}]
values_list() # 可以指定获取的数据字段 返回结果 [(),()] queryset多项可通过 对象.query 查看sql语句
distinct() # 去重 包含主键 无法去重
order_by() # 排序 可以指定参数 默认升序 降序只需在参数前加个 -
reverse() # 反转前提是数据已经排序完毕
count() # 统计当前数据的个数
exclude() # 排除在外
exists() # 判断是否存在
查看sql语句的通用方式
在settings下配置

# 双下划线查询
__gt 大于
__lt 小于
__gte 大于等于
__lte 小于等于
__in 或 age__in = [10,20,30]
__range 区间  __range = [18,20]
__contains 模糊查询（包含）默认区分大小写 忽略大小写 __icontains
__startwith
__endwith
__month 按照月份获取
__year
__day
```

一对多外键增删改查

```
# 增
# 方式一 直接写实际字段
models.Book.objects.create(title='title', price=132.23, publish_id=1)
# 方式二 虚拟字段
publish_obj = models.Publish.objects.filter(pk=2).first()
modles.Book.objects.create(title='title', price=132.23, publish_id=publish_obj)

# 删
models.Publish.objects.filter(pk=2).delete() # 级联删除

# 修改
publish_obj = models.Publish.objects.filter(pk=2).first()
modles.Book.objects.filter(pk=1).update(publish_id=publish_obj)
```

多对多外键增删改查

```
# 增
# 方式一
book_obj = models.Book.objects.filter(pk=1).first()
book_obj.authors.add(1) # add()中可以添加多个参数
# 方式二
book_obj = models.Book.objects.filter(pk=1).first()
auth_obj = models.Publish.objects.filter(pk=2).first()
book_obj.authors.add(auth_obj) # add()中可以添加多个参数

# 删
# 方式一
book_obj = models.Book.objects.filter(pk=1).first()
book_obj.authors.remove(1) # 删除外键关系
# 方式二
author_obj = models.Author.objects.filter(pk=2).first()
book_obj.authors.remove(auth_obj) # remove()中可以添加多个参数

# 修改
book_obj = models.Book.objects.filter(pk=1).first()
# 方式一
book_obj.authors.set([1, 2])
book_obj.authors.set([1, 2])
# 方式二
author_obj = models.Author.objects.filter(pk=2).first()
author_obj2 = models.Author.objects.filter(pk=3).first()
book_obj.authors.set([author_obj, author_obj2])
'''
set()括号内必须是一个可迭代对象，对象内既可以时数字 也可以是对象 并支持多个参数
'''

# 清空 在第三张关系表中 清空表关系
book_obj = models.Book.objects.filter(pk=1).first()
book_obj.authors.clear()
```

多表查询

```
# 外键字段所在类 为正向类
# 正向 外键字段所在类 为正向类
# 反向
'''
正向查询按字段
反向查询按表名小写

在写ORM语句的时候和写sql语句是一样的 不一定一次写完
当结果可能有多个 需要加 .all() 遇到 app.表面.None 加all()
'''

# 子查询（基于对象的跨表查询）
正向
book_obj = modles.Book.objects.filter(pk=1).first()
book_obj.publish

publish_obj = models.Publish.objects.filter(name='东方出版社').first()
publish_obj.book_set

'''
当查询结果是多个的时候需要添加_set.all()
当查询结果只有一个的时候 不需要加
'''
# 联表查询(基于双下划线的跨表查询)
# 正向
res = models.Author.objects.filter(name='jason').values('author_detail__phone')
# 反向
res = models.AuthorDetail.objects.filter(author__name='jason').values('phone', 'author__name')

# 正向
res = models.Book.objects.filter(pk=1).values('publish__name', 'title')
# 反向
res = models.Publish.objects.filter(book__id=1).value('name', 'book__title')

# 正向
res = models.Book.objects.filter(pk=1).values('authors__name')
# 反向
res = models.Author.objects.filter(book__id=1).vlaues('name')

# book author authordetail 查询书籍主键为1的作者的手机号
res = models.Book.objects.filter(pk=1).values(authors__author_detail__phone)
```

数据库连接

```
# 在项目名下的__init__.py中
import pymysql
pymysql.install_as_MySQLdb()
```

数据的增删改查

```
# 增
models.book.objects.create()  创建一条记录
# 查
user = models.User(username=un, password=pd)
user.save()
models.book.objects.all() 查询所有的数据库记录 以对象的形式返回
models.book.objects.values('title', 'id') 返回查询集合 以字典的形式返回
models.book,objects.values_list() 返回查询集合 以元组的形式返回
models.book.objects.order_by('pub','id') 返回查询集合 以对象形式返回  位置越靠前 优先级越高
models.book.objects.filter(pub='', '') 返回查询集合 以对象的形式返回 不存在返回空
models.book.objects.get(pub='') 返回一条记录 不存在则报错
models.book.objects.exclude(条件) 返回不满足条件的记录
.update(username=username, password=pd) # 更新数据
.first() # 获取第一条数据
# 修改
abook = models.book.objects.get(id=1)
abook.title = "xxxx"
abook,save()
修改多个（利用查询结果集）
some_books = models.book.objects.all()/filter().updata(price=100)  # 可单量 可批量
# 删除数据库记录
abook = models.book.objects.get(id=1)
abook.delete()
修改多个（利用查询结果集）
some_books = models.book.objects.all()/filter()
some_books.delete()
```

choices参数（数据库字段设计常见）

```
'''
只要符合字段可能性可以列举完全的 一般情况下使用choices参数
'''
class User(models.Model):
    username = models.CharField('姓名', max_length=20)
    gender_choices = (
        (1, '男'),
        (2, '女'),
        (3, '其他'),
    )
    gender = models.IntegerField(choices=ender_choices)
  # 获取字段名 get_字段名_display()
```

多对多三种创建方式

```
# 全自动 利用orm帮助我们创建第三张表 第三张表拓展性差
# 纯手动 自己创建第三张表 指定利用外键指定表关系（不建议使用）
# 半自动 全自动的前提下 创建一张表
class Book2Author(models.Model):
    book = models.ForeignKey(to='Book')
    author = models.ForeignKey(to='Author')

# 除此以外 还需要再ManyToManyField方法中 添加 through='Book2Author', through_field=('book',
'author')/through_field=('author', 'book')
无法使用 set() add() remove() clear()
```

在filter、exclude、get中可以加入以下查询谓词

- pub_contains = '北大'

聚合查询

```
# 只要是和数据库相关的模块 基本都在 django.db.models 里 如果没有则应该在django.db中
'''
聚合查询通常情况下都是配合分组一起使用的
'''
from django.db.models import Max, MIn, Sum, Count, Avg
res = models.Books.objects.aggregate(Avg('price'))
res = models.Books.objects.aggregate(Avg('price'), Min('price'), Sum('price'), Count('price'))
```

分组查询

```
# annotate
res = models.Book.objects.annotate(author_num=Count('authors')).values('title', 'author_num') #
models后面 .什么就是按照什么分组
res = models.Publish.objects.annotate(min_price=Min('book__price')).values('name', 'min_price')
res =
models.Book.objects.annotate(author_num=Count('authors')).filter(author_num__gt=1).values('title
', 'author_num')
res = models.Author.objects.annotate(sum_price=Sum('book__price')).values('name', 'sum_price')

# 按照字段分组
models.Author.objects.values.annotate()

# 如果分组出错 需要修改 数据库严格模式
```

F对象

一个F对象代表数据库中某个字段的信息

```
from django.db.models import F
books = models.Book.objects.all()
```

```
# for book in books:
#     p = float(book.price)
#     p += up
#     book.price = p
#     book.save()
books.update(price=F('price') - float(up))
models.book.objects.filter('market_price__gt=F('price')
# 在操作字符类型的数据时 F不能直接做到数据拼接
# 解决方法
from django.db.models.functions import Concat
from django.db.models import values
molels.Book.objcts.update(title=Concat(F('title'), value('xxx')))
```

Q对象

实现与或非等操作

- 与操作 &
- 或操作 |
- 非操作 ~
- 异或操作 ^

```
book = models.Book.objects.filter(Q(price__lt=20) | Q(pub='清华大学出版社'))

# Q的高阶用法
q = Q()
q.connector = 'or'
q.children.append(('price__lt', 600))
```

django中如何开启事务

```
ACID
    原子性  不可分割的最小单位
    一致性  与原子性相辅相成
    隔离性  事物之间互不干扰
    持久性  事务一旦确认永久生效
    事务回滚 rollback
    事务确认 commit
# 事务
from django.db import transaction
with transaction.atoomic():
    sql1
    sql2
# 其他操作
```

ORM中常用字段及参数

```
AutoField
```

```
CharField
    verbose_name  # 注释
    max_length   # 长度
IntegrField
BigIntegrField
DecimalField
    max_digits
    decimal_plces
EmailField
    varchar
DateField
DetaTimeField
    auto_now  # 每次修改数据都自动更新时间
    auto_now_add    # 只在创建记录时创建时间 后续不更新
BooleanField(Field)
TextField(Field) # 无字数限制
FileField(Field)
    upload_to = "/data" # 给该字段传一个文件对象 自动将文件保存到/data目录下然后将文件路径保存到数据中/data/a.txt

# 自定义字段
class MyCharField(models.Field):
    def __init__(self, max_length, *args, **kwargs):
        self.max_length = max_length
        super().__init__(max_length=max_length, *args, **kwargs)

# 外键字段
unique=True
ForeignKey(unique=True) === OneToOneField()
db_index 创建索引
to_field 这是关联字段
on_delete # django2.0及以上版本 需要自己指定外键字段的级联更新级联删除
```

数据库查询优化

```
only 和 defer
defer()  括号内放的字段不在查询出的对象中
only() 与defer() 相反

select_related 和 prefetch—related------与跨表操作有关
select_related()
# 内部将俩张表连起来 然后一次性将大表封装给查询出来的对象（括号内只能放外键字段 一对一 一对多）
prefetch—related()
# 该方法内部就是子查询 将子查询的查询结果 封装到对象中
'''
ORM语句特点
    惰性查询：用则查 不用则不查
'''
```

原生数据库操作方法

```
Entry.objects.raw(sql语句)
```

使用django中的游标cursor对数据库进行增删改

1. 导入游标所在的包

```
from django.db import connection
with connection.cursor() as cur:
    cur.execute('sql语句')
```

django的后台管理

```
python manage.py createsuperuser
# 访问地址 http://127.0.0.1:8000
```

- 自定义后台管理数据表

```
# 在admin.py中
from . import models
admin.site.register(models.类)
```

- 模型管理器类的使用方法

```
from django.contrib import  admin
class BookManager(admin.ModelAdmin):
    list_display = ['id', 'title']
    list_display_links = ['id', 'title']  # 是否链接到对象的更改页面
    list_filter =['pub']  # 是否添加到过滤器
    search_fields = [字段] # 是否添加搜索框

admin.site.register(models.Book, BookManager)
admin.site.register(models.Author)
```

Meta 内嵌类 定义模型类及展示形式

```
class Author(models.Model):
    name = models.CharField('姓名', unique=True, ad_index=True, max_length=20)
    ...
    class Meta:
        db_table = 'myauthor'  # 表名
        verbose_name = 'AaAaAaAaAa'  # 单数名
        verbose_name_purial = 'auTTThor'  # 复数名
```

数据表关联关系映射(ORM)

表关联

```
# 一对一
class A(models.Model):
    pass

class B(models.Model):
    mya = models.OneToOneField(to='表名') # mya = models.OneToOneField(A)

a = A()
b = B(mya=a)

# 一对多（外键字段在多的一方）
class A(models.Model):
    pass

class B(models.Model):
    mya = models.ForeignKey(to='表名')  # to_field参数可以省略 mya = models.ForeignKey(A)


a = A()
b1 = B(mya=a)
b2 = B(mya=a)
    # 多对一查找
    b1.mya
    # 一对多查找(两种方式)
    items = a.b_set.all()
    items = B.objects.filter(mya=a)
# 多对多（需要创建第三方表）
models.ManyToManyField(to='表名') # models.ManyToManyField(A)
```

WEB服务网关接口 wsgi

```
'''
wsgi 是协议
wsgiref 和 uwsgi是该协议的功能实现模块
'''
```

路由层

```
路由匹配
第一个参数是正则表达式
# 首页写法
url(r'^$', views.index)
```

无名分组

```
'''
给正则表达式加 ()
'''
url(r'^test/(\d+)/', views.index)
无名分组就是将括号内的数据当做位置参数传递给视图函数
```

有名分组

```
'''
有名分组就是将括号内的数据当做关键字参数传递给视图函数
'''
url(r'^test/(?P<year>\d+)/', views.index)
```

无名有名不能混用，但是单个分组可以多次使用

反向解析

```
# 通过一些方法 得到一个结果 该结果可以直接访问对应的url调用的试图
在url设置时 起别名
url(r'^test/', views.index,  name='sdf')
后端反向解析
import reverse
reverse('别名')
前端反向解析
{% url '别名' %}
有名分组反向解析
url(r'^test/(?P<year>\d+)/', views.index, name='xxx')
reverse('xxx', kwargs={})
reverse('xxx', args=(edit_id,)) # 后端  edit_id-->数据的主键值
{% url 'xxx' obj.edit_id%} # 前端
无名分组反向解析
url(r'^test/(\d+)/', views.index, name='xxx')
reverse('xxx', args=(edit_id,)) # 后端  edit_id-->数据的主键值
{% url 'xxx' obj.id%} # 前端
```

路由分发

```
# django的每一个应用都可以有自己的一套完整的文件夹 实现分组开发
总路由
from app1 import urls as app1_urls
import include
url(r'app1/', include(app1_urls))
---------------------------
url(r'app1/', include(app1.urls))  # 终极写法 不需要导包
子路由正常写

# 名称空间
正常情况下的反向解析无法识别app
```

```
url(r'app1/', include(app1_urls, namespace='app1'))
reverse('app1:reg')

# 伪静态
# 将一个动态网页伪装成静态网页
 # 目的 增大seo的查询力度
    增加搜索引擎的收录改率
```

jsonResponse

```
from django.http import JsonResponse
def ab_json(request):
user_dict = {'username':'json', 'password':'132'}
# json_str = json.dumps(user_dict)
return JsonResponse(user_dict, json_dumps_params={'ensure_ascii':False})
```

form 表单上传文件

```
form表单上传文件
method="post" enctype="multi/formdata"
```

```
request.FILES.get()
```

FBV CBV

```
from django import View
class Myclass(View):
    def get(self, request):
        return render(request, 'form.html')
    def post(self, request):
        return HttpResponse('post方法')

url(r'^myclass/$', views.Myclass.as_view())

    fbv和cbv各有千秋，其中cbv可以根据不同的请求方式匹配到对应的方法
    cbv执行流程

# CBV添加装饰器
from django import View
from django.utils.decorators import method_decorator

@method_decorator(login_auth, name='get') # 方式二
@method_decorator(login_auth, name='post') # 方式二
class Myclass(View):
    @method_decorator(login_auth) # 方式三 所有方法都被装饰
    def dispatch(self, request, *args, **kwargs):
        pass

   # @method_decorator(login_auth) # 方式一
```

```
    def get(self, request):
        return render(request, 'form.html')
    def post(self, request):
        return HttpResponse('post方法')

    url(r'^myclass/$', views.Myclass.as_view())
```

cookie and session

```
#
resp = HttpResponse()
# resp = render()
# resp = redirect()
# 设置cookie
resp.set_cookie(key, 值, max_age=添加超时时间, expires=针对ie设置超时时间)
# 获取cookie
request.COOKIES.get(key)
# 删除cookies(注销功能)
obj = redirect('/login/')
obj.delete_cookie('username')

def login_auth(func):
    def inner(request,*args,**kwargs):
        is_login=request.COOKIES.get('is_login')
        if is_login:
            #代表登录了
            return func(request,*args,**kwargs)
        else:
            #表示没有登录，重定向到登录页面
            # 本次请求的全路径，包括参数
            url=request.get_full_path()
            return redirect('/login_cookie/?ReturnUrl=%s'%url)
    return inner
def login_cookie(request):
    if request.method=='GET':
        return render(request,'login.html')
    if request.method == 'POST':
        user = models.User.objects.filter(name=request.POST.get('name'),
pwd=request.POST.get('pwd')).first()
        if user:
            # 登录成功，写cookie
            # 登录成功，跳转到上次访问的页面
            last_url=request.GET.get('ReturnUrl')
            if last_url:
                obj= redirect(last_url)
            else:
                obj = redirect('/order/')
            obj.set_cookie('name',request.POST.get('name'))
            # obj.set_cookie('is_login',True,max_age=5)
            # 写的这个cookie只在order页面有效
            obj.set_cookie('is_login',True,path='/order/')

            return obj
```

```
        else:
            return redirect('/login_cookie/')
```

session 服务器端的存储方式

```
'''
django默认session过期时间是14天
'''
# 设置session
request.session['key'] = value
'''
1.  自动生成随机字符串
2.  自动将随机字符串和对应的数据存储到django_session表
    2.1  现在内存中产生数据的缓存
    2.2  在响应结果django中间件的时候才真正操作数据库
3.  将产生的随机字符串返回给客户端浏览器保存
'''
# 获取session
request.session.get('key')
'''
1. 自动从浏览器请求中获取session对应的随机字符串
2. 用随机字符串在django_session表中查找对应的数据
3. 对比 成果则封装为字典并返回 失败则返回None
'''
# django_session表中的数据条数 取决于计算机 同一个计算机和IP只对应一条数据
# 设置过期时间
request.session.set_expiry() # 整数 秒 / 日期 到期失效 / 0 窗口关闭立刻失效 / 不写 默认
# 清除session
request.session.delete()
request.session.flush()
```

Ajax

```
"""
异步提交 局部刷新
"""
```

django自带的序列化组件

```
from django.core import serializers
user_queryset = models.User.objects.all()
res = serializers.serializer('json', user_queryset)
return HttpResponse(res)
```

批量插入数据

```
book_list = []
for i in range(100000):
    book_obj = models.Book(title='xxx%d' % i)
    book_list.append(book_obj)
models.Book.objects.bulk_create(book_list)
```

分页显示

```
user_queryset = models.User.objects.all()
```

中间件

django请求响应处理的钩子框架-- 插件系统

```
'''
中间件是django的门户
请求先经过中间件 才能到达django的后端 响应也需要先经过中间件才能到达浏览器
1. 全局用户身份校验
2. 全局用户权限校验
3. 全局访问频率校验
...

基于dango中间件的编程思想
1. 新建一个包 在包中创建py文件实现具体的功能，py文件中用类去包装功能
2. 在settings配置文件中 利用字符串的形式 添加功能 '包名.py文件名.类名'
3. 在包的__init__.py中写入如下代码段
import settings
import importlib

def send_all(content):
    for path_str in settings.BAOMING:
        module_path, class_name = path_str.rsplit('.', maxsplit=1)
        module = importlib.import_module(module_path)
        cls = getattr(module, class_name)
        obj = cls()
        obj.send(content)
4. 在执行文件中需要导入你创建的包
'''
# django 自带七个中间件
'django.middleware.security.SecurityMiddleware'
'django.contrib.sessions.middleware.SessionMiddleware',
'django.middleware.common.CommonMiddleware',
'django.middleware.csrf.CsrfViewMiddleware',
'django.contrib.auth.middleware.AuthenticationMiddleware',
'django.contrib.messages.middleware.MessageMiddleware',
'django.middleware.clickjacking.XFrameOptionsMiddleware',

# 自定义django中间件
'''

在项目名或者应用名下创建一个任意名称的文件夹
```

```
在文件夹内创建一个任意名称的py文件
在py文件内书写类 必须继承MiddlewareMixin
将类的路径以字符串的形式注册到配置文件中

process_request
    请求过来会经过每一个中间件的process_request方法，按照注册顺序重上往下依次执行
    如果中间件中没有定义process_request方法，则下一个执行
    如果该方法返回HttpResponse对象，则不再往后执行
    process_request 做全局功能相关的所有限制功能
process_response
    响应回去会经过每一个中间件的process_response方法，按照注册顺序重下往上依次执行
    该方法有俩个额外的参数request 和 response
    该方法必须返回HttpResponse对象，默认返回的就是形参response
如果在某一个process_request方法返回response，则经过当前中间件的process_response返回
process_view
    路由匹配成功之后 执行process_response之前触发（从上往下执行）
process_template_response
    返回的HttpResponse队友有render属性才会触发（从上往下执行）
process_exception
    当视图函数中出现异常才会触发(从上往下)
'''
from django.utils.deprecation import MiddlewareMixin
class MyMiddleware(MiddlewareMixin):
    def process_request(self, request):
        pass
    def process_response(self, request, response):
        pass
    def process_view(self, request, view_name, *args,**kwargs):
        pass
    def process_template_response(self, request, response):
        pass
# csrf跨站请求伪造
'''
网站在给用户返回一个具有提交功能的页面会指定一个唯一标识
接收到post请求网站会先进行标识校验，如果校验不成功则拒绝请求
'''
方式一：{% csrf_token %} # 在form表单中添加
方式二------针对ajax data:{'csrfmiddlewaretoken':'{{csrf_token}}'}
通用方式：
# 导入js代码  从官方文档获取

# csrf相关装饰器
'''
1. 网站整体不校验，个别函数校验
2. 网站整体校验，个别函数不校验
'''
from django.views.decorators.csrf import csrf_protect, csrf_exempt
FBV正常加
# CBV
@method_decorator(csrf_protect, name='post') # 第二种方式
from django.views import View
class Myclass(View)

    # def dispatch(self, request, *args, **kwargs) 第三种方式
```

```
        # return super(Myclass, self).dispath(request, *args, **kwargs)

    def get(self, request)
        return HttpResponse('get')

    @method_decorator(csrf_protect) # 第一种方式
    def post(self, request)
        return HttpResponse('post')
'''
针对csrf_exempt只有第三种方式方式可行
'''
```

分页

```
def book(request):
        bks = models.Book.objects.all()
        print('总页数是: ',  paginator.)
```

Django中的forms表单

- 作用：浏览器向服务器提交数据
- forms模块可以自动生成form内部的表单控件，在服务器端以对象的形式接受并处理数据

```
# forms 模块的使用步骤
'''
app下创建forms.py文件
'''
from django import forms
class classname(forms.Form):
    username = forms.CharField(min_length=3, max_length=10, lable='用户名')
    password = forms.CharField(min_length=3, max_length=10, lable='密码')
    email = forms.EmailField(lable='邮箱')
# 在模板中自动解析表单
{{ formname.as_p}} # 以p标签的形式渲染
{{ formname.as_table}} # 以table标签的形式渲染
{{ formname.as_ul}} # 以ul标签的新式渲染
# 在模板中手动解析表单
{% for f in myform %}
    {{f.label}} {{f}}
{% endfor %}

# 获取form表单提交的数据
form_obj = Myform(request.POST)
# 判断数据合法性
if form_obj.is_valid():
    pass
else:
```

- label 控件前的文本

- widget 指定小部件

  widget = forms.widgets.InputText(attrs={'class':'form-control',}) # 字段的多个属性直接用空格隔开
- initial 控件的初始值
- required 是否为必填值
- error_messages = {}
- validators 正则校验

form表单钩子函数（HOOK）

```
'''
在forms组件中 自定义校验规则
'''
# 局部钩子（单个字段）
'''校验用户名不能包含666'''
def clean_username(self):
    username = self.cleaned_data.get('username')
    if '666' in username:
        self.add_error('username', '用户名中出现666')
    return username

# 全局钩子（多个字段）
'''校验俩次密码是否一致'''
def clean(self):
    password = self.cleaned_data.get('password')
    confirm_password = self.cleaned_data.get('confirm_password')
    if not password == confirm_password:
        self.add_error('confirm_password', '俩次密码不一致')
    return self.cleaned_data
```

auth模块

```
'''
使用auth模块，要用就用全套，要么别用
'''
# 创建超级用户
python manage.py createsuperuser

from django.contrib import auth

user_obj = auth.authenticate(request, username=username, password=password)  # auth自动校验用户名和密码 成功则返回一个用户对象 否则返回None
if user_obj:
    auth.login(request, user_obj)  # 在session中更新用户数据  只要login方法执行，你可在任何地方获取当前登陆的用户对象


 from django.contrib.auth.decorators import login_required # 此装饰器一经添加 用户需登录才能访问试图函数
# 局部配置
```

```
@login_required(login_url='/login/')
def home():
    pass
# 全局配置
# 在settings文件中
LOGIN_URL= /login/
在视图函数直接 @login_required
# 局部优先级大于全局优先级

request.user.check_password(old_password)  # 返回布尔值
request.user.set_password(new_password)
request.user.save()

# 注销用户
auth.logout(request)  # 类似于 request.sesson.flush()
# 注册
from django.contrib.auth.models import User
# 创建普通用户
User.objects.create_user(username=username, password=password)
# 创建超级用户
User.objects.create_superuser(username=username, password=password,  email=email) # 此处邮箱必填

# 扩展auth_user表
# 第一种： 一对一关系 （不推荐）
# 第二种：利用面向对象的继承
'''
如果继承了AbstractUser
那么在执行数据库迁移命令的时候auth_user表就不会再被创建
而UserInfo表中会出现auth_user所有的字段外加自己扩展的字段
这么做的好处在于你能够直接点击你自己的表更加快速的完成操作及拓展
前提 1. 在继承之前没有执行过数据库迁移
    2. 继承的类里面不要覆盖AbstractUser里面的字段名
    3. 需要在配置文件中告诉django
        AUTH_USER_MODEL = 'app.UserInfo'
'''
from django.contrib.auth.models import User, AbstractUser
class UserInfo(AbstractUser):
    phone = modles.BigIntegerField()
```

项目部署

1. 安装和配置相同版本的数据库
2. django项目迁移（python版本、依赖包）
    - 安装python

```
pip install python3
```

    - 部署django

```
'''pip install django==xxx.xxx.xxx'''
pip freeze > a.txt
pip install -r a.txt
```

- 复制源代码

```
scp -a 路径 root@123.456.45.789//root/mydjango
```

- 远程登录云主机

```
ssh root@123.456.45.789
```

3. 用uwsgi代替 python manage.py runserver 方法启动服务器

   WSGI是一种web服务器的网关接口

```
sudo pip install uwsgi
```

   在项目文件夹(与wsgi.py同级目录)中创建 uwsgi.ini

```
[uwsgi]
http-socket=:55555//配置uwsgi监听的socket(ip+端口)
callable=app//uwsgi调用的python应用实例名称,Flask里默认是app,根据具体项目代码实例命名来设置
wsgi-file=server.py//调用的主程序文件，绝对路径或相对于该ini文件位置的相对路径均可
master=true//以独立守护进程运行
processes=8//配置进程数量
threads=4//配置线程数量
enable-threads=true//允许在请求中开启新线程
stats=127.0.0.1:9191//返回一个json串，显示各进程和worker的状态
pidfile=uwsgi.pid//存放uwsgi进程的pid，便于重启和关闭操作
listen=1024//监听队列长度，默认100，设置大于100的值时，需要先调整系统参数
chdir = /project  //指定项目目录为主目录
daemonize=uwsgi.daemonize.log//以守护进程运行，日志文件路径
memory-report=true//启用内存报告，报告占用的内存
buffer-size=65535//设置请求的最大大小（排除request-body），这一般映射到请求头的大小。默认情况下，它是4k，大cookies的情况下需要加大该配置
```

1. 配置nginx反射代理服务器
2. 用nginx配置静态文件路径 解决静态路径问题

```
```

# #

pycharm 社区版针对Django的调试方法

1. 添加自己的调试配置 add configuration

2. 点击 + 添加一个自己的配置

    3. 选择运行项目的主模块位置 manage.py
    4. 添加参数 runserver 命令行参数

补充知识点

```
import importlib
'''
该方法最小只能到py文件名
'''
# a.py
res = 'myfile.b'
ret =  importlib.import_module(res)  # 以上两行相当于 from . import b
# b.py
```