Django

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

引言

web框架的实现的功能：

1. socket收发消息
2. 根据不同的路径返回不同的内容
3. 返回HTML页面
4. 返回动态的页面 （模板的渲染 字符串的替换）

HTTP协议，通用版本：1.1

（https://www.cnblogs.com/maple-shaw/articles/9060408.html）

1. HTTP是一个客户端终端（用户）和服务器端（网站）请求和应答的标准（TCP）。
2. HTTP/1.1协议中共定义了八种方法（也叫"动作"）来以不同方式操作指定的资源：
   1. **GET**：向指定的资源发出显示请求（常用）
   2. **HEAD**：与get方法想同，只不过服务器不传回资源的文本部分
   3. **POST**：向指定资源提交数据，请求服务器进行处理（常用）
   4. **PUT**：向指定资源上传其最新内容
   5. **DELETE**：请求服务器删除Request-URI所标识的资源
   6. **TRACE**： 回收服务器收到的请求，主要用于测试或诊断。

7. **OPTIONS：**这个方法可使服务器传回该资源所支持的所有HTTP请求方法。用'*'来代替资源名称，向Web服务器发送OPTIONS请求，可以测试服务器功能是否正常运作。
8. **CONECT：** HTTP/1.1协议中预留给能够将连接改为管道方式的代理服务器。通常用于SSL加密服务器的链接（经由非加密的HTTP代理服务器）。

HTTP状态码：

状态代码的第一个数字代表当前响应的类型：

- 1xx消息——请求已被服务器接收，继续处理
- 2xx成功——请求已成功被服务器接收、理解、并接受
- 3xx重定向——需要后续操作才能完成这一请求
- 4xx请求错误——请求含有词法错误或者无法被执行
- 5xx服务器错误——服务器在处理某个正确请求时发生错误

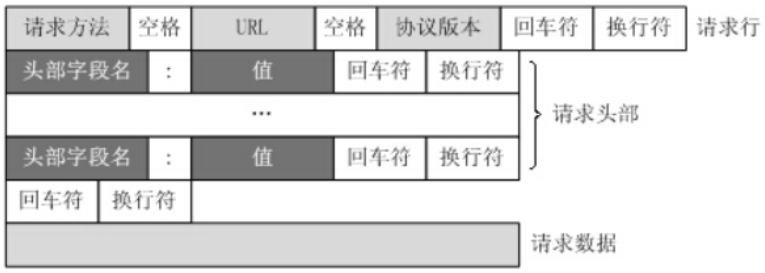
URL：

- 传送协议。
- 层级URL标记符号(为[//],固定不变)
- 访问资源需要的凭证信息（可省略）
- 服务器。（通常为域名，有时为IP地址）
- 端口号。（以数字方式表示，若为HTTP的默认值“:80”可省略）
- 路径。（以“/”字符区别路径中的每一个目录名称）
- 查询。（GET模式的窗体参数，以“?”字符为起点，每个参数以“&”隔开，再以“=”分开参数名称与数据，通常以UTF8的URL编码，避开字符冲突的问题）
- 片段。以“#”字符为起点

数据请求和响应的步骤

1. 在浏览器的地址栏中输入URL地址，回车就会发送一个GET请求
2. Django接收到请求后，根据URL的路径找到对应的函数，执行函数拿到结果
3. Django将结果封装成http响应的报文返回给浏览器
4. 浏览器接受响应，断开连接，解析数据

HTTP请求格式



回到顶部

HTTP响应格式

框架的功能：

1. socket收发消息
2. 根据不同的路径返回不同的结果（HTML页面）
3. 返回动态的页面（模版的渲染，字符串的替换）

根据路径返回具体的HTML文件

详细内容见：https://www.cnblogs.com/maple-shaw/p/8862330.html

让网页动起来

选择使用字符串替换来实现这个需求

而Python标准库提供的独立WSGI服务器叫**wsgiref**，Django开发环境用的就是这个模块来做服务器。

wsgiref：用这个可以替换 自己写的web框架的socket server部分：

jinja2：渲染数据

我完全可以从数据库中查询数据，然后去替换我html中的对应内容，然后再发送给浏览器完成渲染。这个过程就相当于HTML模板渲染数据。 本质上就是HTML内容中利用一些特殊的符号来替换要展示的数据。 我这里用的特殊符号是我定义的，其实模板渲染有个现成的工具：`jinja2`

下载：pip install jinja2

Django安装

下载网址：https://www.djangoproject.com/download/

下载命令行：CMD窗口 pip install django==1.11.25 -i https://pypi.tuna.tsinghua.edu.cn/simple

pycharm中也可以安装，安装1.11.25版本

创建项目：

1. pycharm中：file ——》 new project ——》 填写项目名称 ——》 选择解释器 ——》 create
2. cmd中：django-admin startproject项目名

启动项目：

1. python manage.py runserver # 127.0.0.1:8000
2. python manage.py runserver 80 # 127.0.0.1:80
3. python manage.py runserver 0.0.0.0:80 # 0.0.0.0:80
4. pycharm中启动

初识Django

配置过程：https://www.cnblogs.com/maple-shaw/p/8862330.html（主要查看创建时路径问题）

1. urls.py中写路径和函数的对应关系

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

2. 写函数：

```
from django.shortcuts import HttpResponse ,render
def index(request):
    # 逻辑
    # 查询数据库，插入
    # 返回信息给浏览器
    # return HttpResponse('欢迎进入day5项目')    # 返回字符串
    return render(request,'index.html')  # 返回一个HTML页面      页面写在
templates文件夹中
```

Django必备三件套

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

HttpResponse： 内部传入一个字符串参数，返回给浏览器。

```
def index(request):
    # 业务逻辑代码
    return HttpResponse("OK")
```

render：除request参数外还接受一个待渲染的模板文件和一个保存具体数据的字典参数。将数据填充进模板文件，最后把结果返回给浏览器。

```
def index(request):
    # 业务逻辑代码
    return render(request, "index.html", {"name": "alex", "hobby": ["烫头", "泡
吧"]})
```

redirect: 接受一个URL参数，表示跳转到指定的URL。

```
def index(request):
    # 业务逻辑代码
    return redirect("/home/")
```

静态文件的配置

settings.py文件

```
模板文件 TEMPLATES DIRS [ os.path.join(BASE_DIR,'templates') ]

STATIC_URL = '/static/'   # 静态文件的别名  以它做开头

STATICFILES_DIRS = [   #  静态文件存放的具体路径 名字是固定的，千万不可以改
    os.path.join(BASE_DIR,'static',)
]
```

强大的模版网站：需要的模版可以去c v：

http://www.jq22.com/

form表单提交数据

1. action=“ “ 向当前的地址进行提交
2. method= “post” 表示请求方式
3. input 需要有name属性和value值
4. button按钮或者input类型是submit
5. 注意只有注释后才可以提交post请求，在settings.Py文件中

```
MIDDLEWARE
'django.middleware.csrf.CsrfViewMiddleware' #注释这一行代码
```

```
request.method  请求方式 GET POST
request.POST    POST请求的数据  {}
redirect（‘要重定向的地址’）    重定向
```

APP

创建app：

1. python manage.py startapp app名称
2. pycharm启动Django项目时直接创建

注册app：settings

```
INSTALLED_APPS = [
    ...
    'app01',        #第一种方法
    'app01.apps.App01Config'    #第二种写法 推荐写法
]
```

app文件目录

```
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）
```

ORM数据库操作工具

对应关系：

1. 类 ---> 表
2. 对象 ---> 数据行（记录）
3. 属性 ---> 字段

Django使用数据库的流程

1. 创建一个mysql数据库，在cmd命令行中
2. 在settings文件中配置数据库相关信息

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',   #  引擎  数据库类型
        'NAME': 'day53',                # 数据库名称
        'HOST': '127.0.0.1',            # ip
        'PORT': 3306,                   # 端口
        'USER': 'root',                 # 用户名
        'PASSWORD': '123'               # 密码
    }
```

3. 告诉Django使用pymysql模块连接数据库， 写在与settings同级目录下的`__init__`中：

```
import pymysql
pymysql.install_as_MySQLdb()   # 能执行就可以，可以放在文件下的 __ init __ 文件下，也可以放在settings文件下
```

4. 在app文件夹下models.py文件中写入以下代码：

```
class User(models.Model):
    username = models.CharField(max_length=32)  # username   varchar(32)
    password = models.CharField(max_length=32)  # username   varchar(32)
```

5. 执行命令：让数据有对应的结构的变化，类似于数据库创建表

```
在python terminal 框下执行以下命令检测数据库变化，

1.python manage.py makemigrations # 检查所有已经注册的app下的models.py的变化 记录成一个文件 migrations文件，

2.python manage.py migrate   #迁移数据库
```

ORM操作

```
from app01.models import User
User.objects.all()  #  查询所有的数据   QuerySet 对象列表  【  对象 】
User.objects.get(username='alexs',password='alexdsb')
# 对象  get  只能获取数据库唯一存在的数据  有且唯一 （不存在或者多条数据就报错）

User.objects.filter(password='dsb')  # 获取满足条件的所有对象  对象列表
```

get和post的区别

get 获取页面

?k1=v1&k1=v2 request.GET.get() url 上携带的参数

get请求没有请求体

post 提交数据

提交的数据 在请求体中

Django中所有的命令

1. 下载django
   pip install django==1.11.25 -i 源
2. 创建django项目
   django-admin startproject 项目名
3. 启动项目
   切换到项目目录下
   python manage.py runserver # 127.0.0.1:8000
   python manage.py runserver 80 # 127.0.0.1:80
   python manage.py runserver 0.0.0.0:80 # 0.0.0.0:80
4. 创建app
   python manage.py startapp app名称
5. 数据库迁移的命令
   python manage.py makemigrations # 检测已经注册app下的models变更记录
   python manage.py migrate # 将变更记录同步到数据库中

django的配置

静态文件

STATIC_URL = '/static/' # 静态文件的别名

STATICFILES_DIRS = [# 静态文件存放的路径

    os.path.join(BASE_DIR ,'static')

]

数据库

ENGINE mysql

NAME “数据库名称”

HOST “ip”

PORT 端口号

USER “用户名”

PASSWORD “密码”

中间件

注释掉 一个 csrf (可以提交POST请求)

app

INSTALLED_APPS = [

'app名称'

‘app名称.apps.app名称Config’

]

5. 模板

TEMPLATES DIRS =[os.path.join(BASE_DIR ,'templates')]

示例：图书管理系统出版社管理界面设计

展示

设计URL地址

```
from app01 import views

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url(r'^publisher_list/', views.publisher_list),
]
```

写函数

```
def publisher_list(request):
    # 从数据库中获取所有的出版社的信息
    all_publisher = models.Publisher.objects.all()  # 对象列表

    # 将数据展示到页面中
    return render(request,'publisher_list.html',{'k1':all_publisher})
```

模板语法

```
{{ 变量  }}

for循环：
{% for i in 变量  %}
    循环体 {{ i }}
{% endfor  %}

if判断：
{% 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  %}
```

新增

```
方式一：
models.Publisher.objects.create(name=pub_name,addr=pub_addr)  # 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
```

删除

```
方法一：推荐使用
models.Publisher.objects.filter(pid=pid).delete()  # 对象列表 删除
方法二：考虑使用
models.Publisher.objects.get(pid=pid).delete()  # 对象 删除
```

编辑

```
 pub_obj.name = pub_name
 pub_obj.addr = pub_addr
 pub_obj.save()  # 将修改提交的数据库
```

书籍管理界面的实现

函数

```
def  func(request):
    # 逻辑
    return HttpResponse对象


request.GET    url上携带的参数   {}
request.POST   POST请求提交的数据    {}
request.method  请求方法  GET  POST

HttpResponse('字符串')      返回字符串
render(request,'模板文件的名字',{k1:v1})      返回一个HTML页面
redirect('重定向的地址')     重定向
```

外键

描述 一对多 的关系，通常写在多的那一边

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


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

展示

```
from app01 import models
all_books = models.Book.objects.all()

for book in all_books:
    print(book.id)  print(book.pk)
    print(book)   # 对象  __str__
    print(book.title)
    print(book.pub)      # 外键关联的对象  print(book.pub.pk)
print(book.pub.name)
    print(book.pub_id)   # 外键关联的对象id
```

新增

```
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()
```

删除

```
models.Book.objects.filter(id=id).delete()
models.Book.objects.filter(id=id).first().delete()
```

编辑

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

模板的语法：写在html界面

```
{{ 变量  }}

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

```
{%  if 条件 %}
    内容
{% endif %}

{%  if 条件 %}
    内容
{% else %}
    内容2
{% endif %}


{%  if 条件 %}
    内容
{%  elif 条件1 %}
    内容
{% else %}
    内容2
{% endif %}
```

作者的管理

作者 书籍 多对多的关系

```
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
```

多对多关系表的创建方式:

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
```

自己手动创建

```
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)
```

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=True)
    author = models.ForeignKey('Author',on_delete=models.CASCADE,related_name='x1',null=True)
    tuijianren = models.ForeignKey('Author',on_delete=models.CASCADE,related_name='x2',null=True)
    date = models.CharField(max_length=32)
```

展示

```
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
```

新增

```
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
```

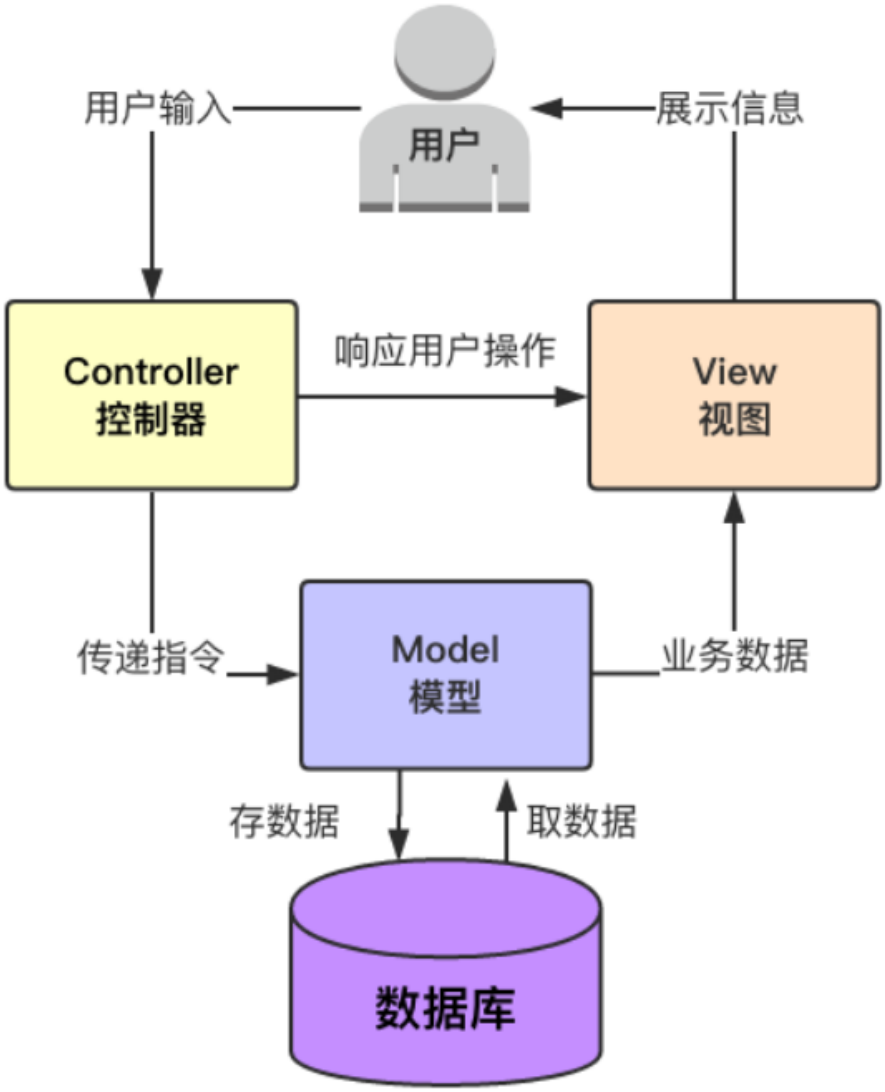
MVC和MTV的区别：

MVC： 全名是Model View Controller，是软件工程中的一种软件架构模式

M: model 模型 数据库交互

V：view 视图 展示给用户看的 HTML

C ： controller 控制器 业务逻辑 传递指令



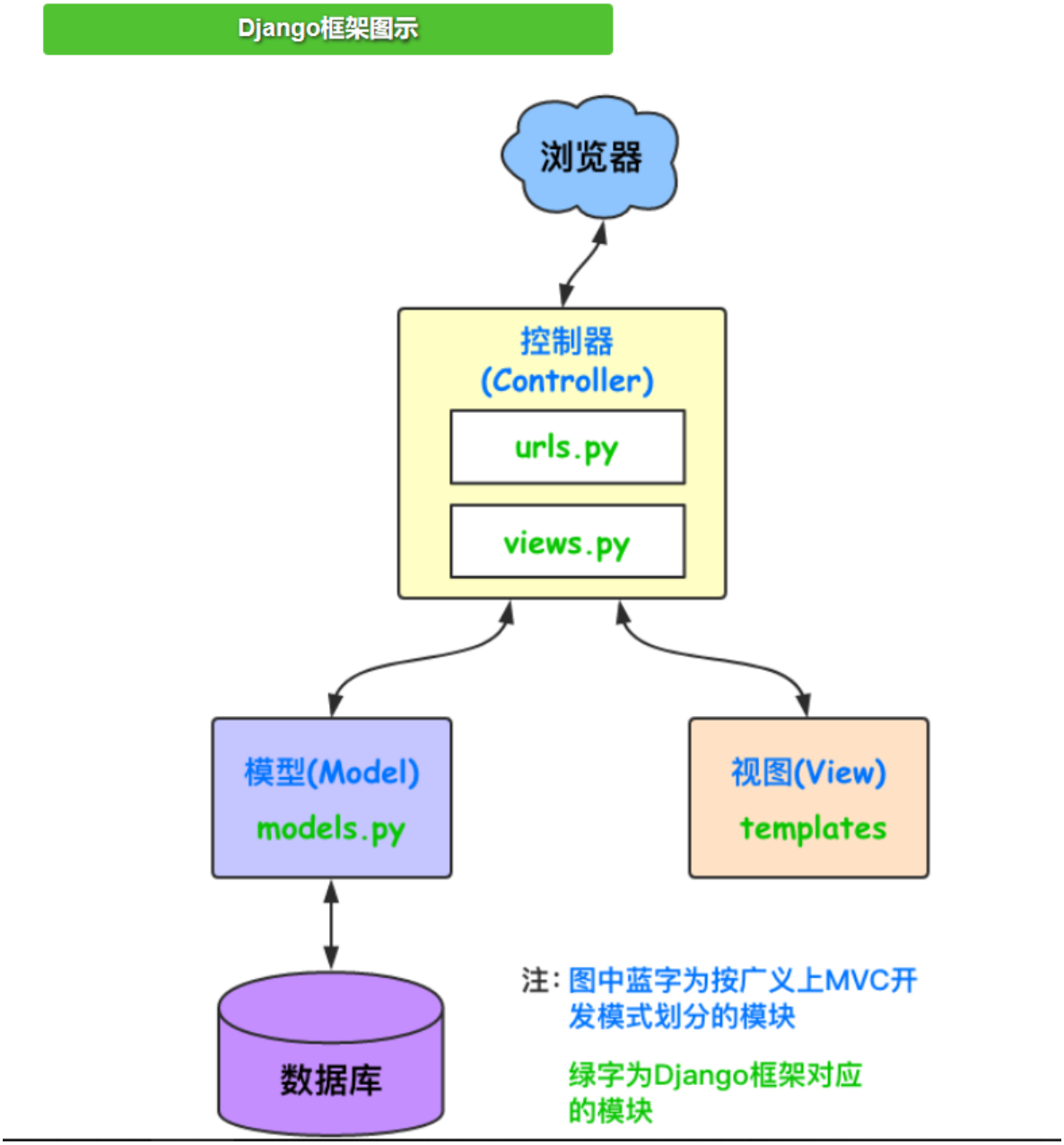
MTV：Django的MTV模式

Model(模型)：负责业务对象与数据库的对象(ORM)

Template(模版)：负责如何把页面展示给用户

View(视图)：负责业务逻辑，并在适当的时候调用Model和Template

Django还有一个urls分发器，它的作用是将一个个URL的页面请求分发给不同的view处理，view再调用相应的Model和Template



Django模版（Template）系统：

常用语法：

{{ }}和 {% %}

{{ }}表示变量，在模板渲染的时候替换成值，{% %}表示逻辑相关的操作。

变量：{{变量名}}：变量名由数字，字母和下划线组成

点（.）在模板语言中有特殊的含义，用来获取对象的相应属性值。

```
{# 取l中的第一个参数 #}
{{ l.0 }}
{# 取字典中key的值 #}
{{ d.name }}
{# 取对象的name属性 #}
{{ person_list.0.name }}
{# .操作只能调用不带参数的方法 #}
{{ person_list.0.dream }}
```

```
当模板系统遇到一个（.）时，会按照如下的顺序去查询：

    在字典中查询
    属性或者方法
    数字索引
```

filter：过滤器，用来修改变量的显示结果

语法： **{{ value|filter_name:参数 }}**

':'左右没有空格没有空格没有空格

default：

```
{{ value|default:"nothing"}}
```

冒号后如果value值没传的话就显示nothing

注：TEMPLATES的OPTIONS可以增加一个选项：string_if_invalid：'找不到'，可以替代default的的作用。

filesizeformat

将值格式化为一个 “人类可读的” 文件尺寸 （例如 '13 KB', '4.1 MB', '102 bytes', 等等）。如：

```
{{ value|filesizeformat }}
```

如果 value 是 123456789，输出将会是 117.7 MB。

add ： 给变量加参数

```
{{ value|add:"2" }}    value是数字4，则输出结果为6。
{{ 'a'|add:'b'  }}    字符串的拼接，返回ab
{{ [1,2]|add:[3,4]  }}    列表的拼接，返回【1，2，3，4】
```

```
{{ first|add:second }}
```

如果first是 [1,.2,3] ， second是 [4,5,6] ， 那输出结果是 [1,2,3,4,5,6]

lower：小写

```
{{ value|lower }}
```

upper：大写

```
{{ value|upper}}
```

title：标题

```
{{ value|title }}
```

ljust：左对齐

```
"{{ value|ljust:"10" }}"
```

rjust：右对齐

```
"{{ value|rjust:"10" }}"
```

center：居中

```
"{{ value|center:"15" }}"
```

length：{{ value|length }}

返回value的长度，如 value=['a', 'b', 'c', 'd']的话，就显示4.

slice　　：切片

```
{{value|slice:"2:-1"}}
{{ string|slice:'-1::-1' }}
```

first：取第一个元素

```
{{ value|first }}
```

last：取最后一个元素

```
{{ value|last }}
```

join：使用字符串拼接列表。同python的str.join(list)。

```
{{ value|join:" // " }}
```

truncatechars

如果字符串字符多于指定的字符数量，那么会被截断。截断的字符串将以可翻译的省略号序列（“...”）结尾。

参数：截断的字符数

```
{{ value|truncatechars:9}}
```

date：日期格式化

```
{{ now|date:'Y-m-d H:i:s' }}


date 日期时间格式化 {{ 日期时间类型|date:'Y-m-d H:i:s' }}

• 在settings配置文件中修改以下内容:

• USE_L10N = False

• DATETIME_FORMAT = 'Y-m-d H:i:s'

• DATE_FORMAT = 'Y-m-d'
```

safe:

```
{{ 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

创建步骤:

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，L是大写
```

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()
```

使用方法：在模版中

```
{% load my_tags  %}
{{ 变量|str_upper:'参数'  }}
```

标签： {% %}

for:

```
格式:
{%  for i in 可迭代对象 %}
    {{ 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（表示占格数）="4" >没有数据</td> </tr>
{% endfor %}
```

if：不支持连续判断

```
{% if user_list %}
  用户人数：{{ user_list|length }}
{% elif black_list %}
  黑名单数：{{ black_list|length }}
{% else %}
  没有用户
{% endif %}
```

if语句支持 and 、or、==、>、<、!=、<=、>=、in、not in、is、is not判断。

with： 定义一个中间变量

格式:

```
{% with 对象 as 别名 %}


{% endwith %}
```

```
{% with p_list.0.name as alex_name %}
    {{ alex_name }}
    {{ alex_name }}
    {{ alex_name }}
    {{ alex_name }}
{% endwith %}

等同于

{% with alex_name=p_list.0.name   %}
    {{ alex_name }}
    {{ alex_name }}
    {{ alex_name }}
    {{ alex_name }}
```

```
{% endwith %}
```

csrf_token： 这个标签用于跨站请求伪造保护

用法： 在页面的form表单里面写上{% csrf_token %}

```
{% 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，可以不用注释之前的那个
```

注释事项：

1. Django不支持连续判断
2. Django的模版语言中属性的优先级高于方法

```
函数中：
def xx(request):
    d = {"a": 1, "b": 2, "c": 3, "items": "100"}
    return render(request, "xx.html", {"data": d})


 模版中体现：
 {{ data.items }}：默认会取items key的值
```

母版和继承：

母版：

1. 母版就是一个HTML页面，提取到的多个页面的公共部分
2. 在母版的HTML页面内需要填充的地方事先定义一个block块，由子页面进行填充

```
<head>
  <meta charset="UTF-8">
  <meta http-equiv="x-ua-compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Title</title>
  {% block page-css %}

  {% endblock %}
</head>
<body>

<h1>这是母板的标题</h1>

{% block page-main %}   #这个名字和继承中block中写的名字是一样的

{% endblock %}
<h1>母板底部内容</h1>
{% block page-js %}

{% endblock %}
</body>
</html>
```

继承：

1. 在子页面导入：页面最上方{% extends ‘母版的名字’ %}
2. 在block块中写入子页面内容： 子页面中通过定义母板中的block名来对应替换母板中相应的内容。

```
{% extends 'xxx.html'%}     #继承母版
{% block 想定义的内容 %}

{% endblock %}
```

注意事项：

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' %} 定义她的上面不要写内容，想显示的内容要写在block块中
3. 多定义点block块，css，js都可以在母版中定义，子页面中调用

组件： 可以将常用的页面内容如导航条，页尾信息等组件保存在单独的文件中，然后在需要使用的地方按如下语法导入即可。

1. 先写一个html文件 存放公共组件代码，一般存放的都是导航栏等不变的文件： **nav.html**
2. 在需要用的页面导入：**{% include 'nav.html' %}**

静态文件：

```
{% load static %}

{% static '相对路径' %}

{% get_static_prefix %}     获取静态文件的别名
```

针对settings文件中static文件需要改名如何重新配置：

```
{% load static %}

<img src="{% static "images/hi.jpg" %}" />
```

```
引用JS文件时使用：：

{% load static %}
<script src="{% static "mytest.js" %}"></script>
```

```
某个文件多处被用到可以存为一个变量

{% load static %}
{% static "images/hi.jpg" as myphoto %}
<img src="{{ myphoto }}"></img>
```

get_static_prefix：

```
{% load static %}
<img src="{% get_static_prefix %}images/hi.jpg" alt="Hi!" />


或者

{% load static %}
{% get_static_prefix as STATIC_PREFIX %}

<img src="{{ STATIC_PREFIX }}images/hi.jpg" alt="Hi!" />
<img src="{{ STATIC_PREFIX }}images/hi2.jpg" alt="Hello!" />
```

自定义simple_tag：和自定义filter类似，只不过接收更灵活的参数

1. 在app文件夹下创建一个名为 templatetags的python包;
2. 包内创建py文件 (任意指定，my_tags.py)
3. 在py文件中写入以下函数：

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

4. 写装饰器+函数：

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

5. 在模版中如何使用：

```
{% load my_tags  %}
{{ 变量|str_upper：‘参数’ }}

{% str_join '1' '2' '3' k1='4' k2='5'  %}

{% page 10 %}
```

inclusion_tag： 多用于返回html代码片段

```
@register.inclusion_tag('page.html')
def page(num):
    return {'num':range(1,num+1)}

# 在templates写page.html
```

示例：

templatetags/my_inclusion.py

```
from django import template

register = template.Library()


@register.inclusion_tag('result.html')
def show_results(n):
    n = 1 if n < 1 else int(n)
    data = ["第{}项".format(i) for i in range(1, n+1)]
    return {"data": data}
```

templates/result.html

```
<ul>
  {% for choice in data %}
    <li>{{ choice }}</li>
  {% endfor %}
</ul>
```

templates/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="x-ua-compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>inclusion_tag test</title>
</head>
<body>

{% load my_inclusion %}

{% show_results 10 %}
</body>
</html>
```

Django的视图系统：View

一个视图函数（类），简称视图，是一个简单的Python 函数（类），它接受Web请求并且返回Web响应。

Django使用请求和响应对象来通过系统传递状态。

当浏览器向服务端请求一个页面时，Django创建一个HttpRequest对象，该对象包含关于请求的元数据。然后，Django加载相应的视图，将这个HttpRequest对象作为第一个参数传递给视图函数。

每个视图负责返回一个HttpResponse对象。

CBV和FBV：

FBV： 写过的都是基于函数的view，就叫FBV。

CBV：基于类的view，就叫做CBV

```
from django.views import View  #最后一个view种的v是大写
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


在urls.py文件中写入以下内容才可以正常使用：
url(r'^publisher_add/',views.PublisherAdd.as_view()),
```

as_view的使用流程：

1. 程序加载时，执行View中的as_view的方法，返回的是一个view函数。
2. 请求到来的时候执行view函数：
    1. 先实例化一个类成对象赋值给self
    2. 执行self.request=request的函数
    3. 执行self.dispathchde(request, *args, **kwargs) #注意参数问题
        1. 判断请求方是否被允许：
            1. 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)
            2. 不允许： handler = self.http_method_not_allowed
            3. 执行handle获取到结果，最终返回给Django

装饰器补充知识：

```
from functools import wraps


def wrapper(func):
    @wraps(func)        # 添加这一步，保证return返回的内容，装饰器的修复，保证传过来的东西不重复
    def inner(*args, **kwargs):
        # 之前
        ret = func(*args, **kwargs)
        # 之后
        return ret

    return inner
```

Django中给视图添加装饰器

FBV中直接给函数添加装饰器就可以

CBV中导入： from django.utils.decorators import method_decorator

1. 加在方法上：

```
@method_decorator(装饰器名)
def 函数名(self,request):
```

2.加在dispatch方法上：

```
@method_decorator(wrapper)
    def dispatch(self, request, *args, **kwargs):
        ret = super().dispatch(request, *args, **kwargs)
        return ret


@method_decorator(wrapper, name='dispatch')
class PublisherAdd(View):

# 使用CBV时要注意，请求过来后会先执行dispatch()这个方法，如果需要批量对具体的请求处理方法，如get，post等做一些操作的时候，这里我们可以手动改写dispatch方法，这个dispatch方法就和在FBV上加装饰器的效果一样
```

3.加在类的上方：

```
需要多少种方法，就在name定义多少种方法
@method_decorator(wrapper, name='post')
@method_decorator(wrapper, name='get')
class PublisherAdd(View):


或者直接定义一个，全部都可以使用：
@method_decorator(wrapper, name='dispatch')
class PublisherAdd(View):
```

Request对象：

当一个页面被请求时，Django就会创建一个包含本次请求原信息的HttpRequest对象。
Django会将这个对象自动传递给响应的视图函数，一般视图函数约定俗成地使用 request 参数承接这个对象

```
request.method    # 请求使用的HTTP 方式，全大写表示，如："GET"、"POST"
request.GET     #  得到URL上携带的字典类参数，
request.POST    # POST请求提交的数据    enctype="application/x-www-form-urlencoded"，检查使用的是否是POST 方法；应该使用 if request.method == "POST"
request.FILES   # 上传的文件，提交的<form> 带有enctype="multipart/form-data" 的情况下才会
   包含数据。否则，FILES 将为一个空的类似于字典的对象。，FILES 中的每个键为<input type="file" name="" /> 中的name，值则为对应的数据。
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 原始数据,byte数据类型
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.get_signed_cookie(key,salt='',default='')加盐的cookie
request.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。
request.META   # 请求头的信息，包含所有的HTTP 首部
```

```
HttpRequest.encoding  # 表示请求的数据的编码方式，（如果为 None 则表示使用
DEFAULT_CHARSET 的设置，默认为 'utf-8'）
request.get_full_path() # 路径信息  不包含IP和端口 包含查询参数
request.is_ajax()   # 是否是ajax请求 布尔值
HttpRequest.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

与由Django自动创建的HttpRequest对象相比，HttpResponse对象是我们的职责范围了。我们写的每个视图都需要实例化，填充和返回一个HttpResponse。

HttpResponse类位于django.http模块中。

```
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    响应头 Location:  地址
JsonResponse({})   # 默认使用的字典类型，返回json数据    如果要传递非字典类型必须设置
safe=False  Content-Type: application/json
```

设置或删除响应头信息

```
response = HttpResponse()
response['Content-Type'] = 'text/html; charset=UTF-8'
del response['Content-Type']
```

路由系统：URL

URL配置(URLconf)就像Django所支撑网站的目录。它的本质是URL与要为该URL调用的视图函数之间的映射表。

1.基本格式：

```
from django.conf.urls import url

urlpatterns = [
     url(正则表达式, views视图，参数，别名),
]
从上到下进行匹配，匹配到一个就不再往下匹配
```

2.参数说明：

- 正则表达式：一个正则表达式字符串

  r'' ^ 开头 $ 结尾 [0-9a-zA-Z]{4} \d \w . ? 0个或1个 *0个或无数个 + 至少一个
- views视图：一个可调用对象，通常为一个视图函数
- 参数：可选的要传递给视图函数的默认参数（字典形式）
- 别名：一个可选的name参数

3.注意事项：

Django2.0版本之后路由系统是下面的写法：

```
from django.urls import path, re_path

urlpatterns = [
    path('articles/2003/', views.special_case_2003),
    path('articles/<int:year>/', views.year_archive),
    path('articles/<int:year>/<int:month>/', views.month_archive),
    path('articles/<int:year>/<int:month>/<slug:slug>/', views.article_detail),
]
```

2.0版本中re_path和1.11版本的url是一样的用法。

4.分组：

```
urlpatterns = [

    url(r'^blog/([0-9]{4})/(\d{2})/$', views.blog),  #用括号把需要分组的东西包起来，它周围放置一对圆括号（分组匹配）

]
```

从URL上捕获参数，将参数按照位置传参传递给视图函数

1.urlpatterns中的元素按照书写顺序从上往下逐一匹配正则表达式，一旦匹配成功则不再继续。
2.若要从URL中捕获一个值，只需要在它周围放置一对圆括号（分组匹配）。
3.不需要添加一个前导的反斜杠，因为每个URL 都有。例如，应该是^articles 而不是 ^/articles。
4.每个正则表达式前面的'r' 是可选的但是建议加上。

5.分组命名匹配：

在Python的正则表达式中，分组命名正则表达式组的语法是`(?Ppattern)`，其中`name`是组的名称，`pattern`是要匹配的模式。

```
urlpatterns = [

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

]
```

从URL上捕获参数，将参数按照关键字传参传递给视图函数。这个实现与前面的示例完全相同，只有一个细微的差别：捕获的值作为关键字参数而不是位置参数传递给视图函数。

6.include：路由分发

```
from django.conf.urls import url, include
from django.contrib import admin

from app01 import views

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

    url(r'^app01/', include('app01.urls')),  #可以包含其他的URLconfs文件
    url(r'^app02/', include('app02.urls')),
]

包含app01 和app02 两个urls中的所有路径函数
```

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.PublisherAdd.as_view()),

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

    url(r'^publisher_edit/', views.publisher_edit),

]
```

7.传递额外的参数给视图函数（了解）

URLconfs 具有一个钩子，让你传递一个Python 字典作为额外的参数传递给视图函数。

`django.conf.urls.url()` 可以接收一个可选的第三个参数，它是一个字典，表示想要传递给视图函数的额外关键字参数。

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

urlpatterns = [
    url(r'^blog/(?P<year>[0-9]{4})/$', views.year_archive, {'foo': 'bar'}),
]
```

在这个例子中，对于/blog/2005/请求，Django 将调用views.year_archive(request, year='2005', foo='bar')。
当传递额外参数的字典中的参数和URL中捕获值的命名关键字参数同名时，函数调用时将使用的是字典中的参数，而不是URL中捕获的参数。

命名URL和URL反向解析：

简单来说就是可以给我们的URL匹配规则起个名字，一个URL匹配模式起一个名字。

这样我们以后就不需要写死URL代码了，只需要通过名字来调用当前的URL。

静态路由：

URL的命名

```
url(r'^home', views.home, name='home'),  # 给我的url匹配模式起名为 home
url(r'^index/(\d*)', views.index, name='index'),  # 给我的url匹配模式起名为index
```

URL的反向解析：

模版：

```
{% url 'home' %}    ——》  解析生成完整的URL路径
```

views函数中：

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

reverse('home')   ——》  '/app01/home/'
```

分组模式：

URL的命名：

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

URL反向解析

```
模版：
{% url 'publisher_del' 1  %}    ——》  解析生成完整的URL路径
'/app01/publisher_del/1/'
```

```
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/'
```

命名分组模式：

URL的命名

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

URL反向解析

模板

```
{% url 'publisher_del' 1  %}    ——》  解析生成完整的URL路径
 '/app01/publisher_del/1/'
{% url 'publisher_del' 对象.pk  %}    ——》  解析生成完整的URL路径
 '/app01/publisher_del/1/'
```

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/'
```

注意：为了完成上面例子中的URL 反查，你将需要使用命名的URL 模式。URL 的名称使用的字符串可以包含任何你喜欢的字符。不只限制在合法的Python 名称。

当命名你的URL 模式时，请确保使用的名称不会与其它应用中名称冲突。如果你的URL 模式叫做`comment`，而另外一个应用中也有一个同样的名称，当你在模板中使用这个名称的时候不能保证将插入哪个URL。

在URL 名称中加上一个前缀，比如应用的名称，将减少冲突的可能。我们建议使用`myapp-comment`而不是`comment`。

命名空间模式 namespace

即使不同的APP使用相同的URL名称，URL的命名空间模式也可以让你唯一反转命名的URL。

举个例子：

```
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：

模版中：

```
{% url 'namespace:name' % }
```

views函数中：

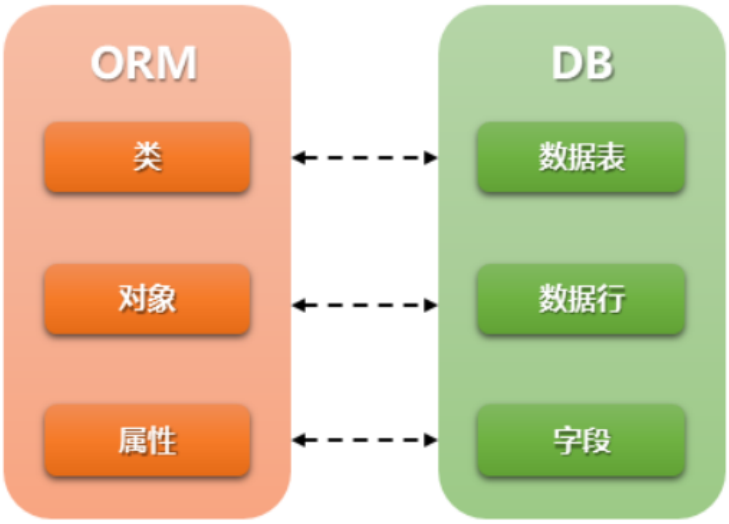
```
reverse( 'namespace:name' )
```

模版系统：Model

ORM：是一种为了解决面向对象与关系数据库存在的互不匹配的现象的技术

基本情况：

- 每个模型都是一个Python类，它是django.db.models.Model的子类。
- 模型的每个属性都代表一个数据库字段。
- 综上所述，Django为您提供了一个自动生成的数据库访问API，详询官方文档链接。



字段：

常用字段：

1.AutoField：自增的整形字段，必填参数primary_key=True，则成为数据库的主键。无该字段时，django自动创建。一个model不能有两个AutoField字段。

```
BigAutoField(AutoField)- bigint自增列，必须填入参数 primary_key=True
```

2.IntegerField：一个整数类型。数值的范围是 -2147483648 ~ 2147483647。

```
SmallIntegerField(IntegerField):- 小整数 -32768 ～ 32767
PositiveSmallIntegerField(PositiveIntegerRelDbTypeMixin, IntegerField)- 正小整数
0 ～ 32767
IntegerField(Field)- 整数列(有符号的) -2147483648 ～ 2147483647
PositiveIntegerField(Po sitiveIntegerRelDbTypeMixin, IntegerField)- 正整数 0 ～
2147483647
BigIntegerField(IntegerField):- 长整型(有符号的) -9223372036854775808 ～
9223372036854775807

DurationField(Field):长整数，时间间隔，数据库中按照bigint存储，ORM中获取的值为
datetime.timedelta类型

FloatField(Field): 浮点型
```

3.CharField：字符类型，必须提供max_length参数。max_length表示字符的长度。

```
TextField(Field)： 文本类型
EmailField(CharField)： 字符串类型，Django Admin以及ModelForm中提供验证机制
IPAddressField(Field)： 字符串类型，Django Admin以及ModelForm中提供验证 IPV4 机制
```

```
GenericIPAddressField(Field)
        - 字符串类型，Django Admin以及ModelForm中提供验证 Ipv4和Ipv6
        - 参数：
            protocol，用于指定Ipv4或Ipv6， 'both',"ipv4","ipv6"
            unpack_ipv4， 如果指定为True，则输入::ffff:192.0.2.1时候，可解析为
192.0.2.1，开启此功能，需要protocol="both"

URLField(CharField)： 字符串类型，Django Admin以及ModelForm中提供验证 URL
SlugField(CharField): 字符串类型，Django Admin以及ModelForm中提供验证支持 字母、数字、
下划线、连接符（减号）
CommaSeparatedIntegerField(CharField):字符串类型，格式必须为逗号分割的数字
UUIDField(Field):字符串类型，Django Admin以及ModelForm中提供对UUID格式的验证
FilePathField(Field):字符串，Django Admin以及ModelForm中提供读取文件夹下文件的功能
        - 参数：
                path,                       文件夹路径
                match=None,                 正则匹配
                recursive=False,            递归下面的文件夹
                allow_files=True,           允许文件
                allow_folders=False,        允许文件夹

FileField(Field):字符串，路径保存在数据库，文件上传到指定目录
        - 参数：
            upload_to = ""      上传文件的保存路径
            storage = None      存储组件，默认
django.core.files.storage.FileSystemStorage
ImageField(FileField): 字符串，路径保存在数据库，文件上传到指定目录
        - 参数：
            upload_to = ""      上传文件的保存路径
            storage = None      存储组件，默认
django.core.files.storage.FileSystemStorage
            width_field=None,   上传图片的高度保存的数据库字段名（字符串）
            height_field=None   上传图片的宽度保存的数据库字段名（字符串）
```

4.DateField：日期类型，日期格式为YYYY-MM-DD，相当于Python中的datetime.date的实例。

```
参数：
    auto_now：每次修改时修改为当前日期时间。
    auto_now_add：新创建对象时自动添加当前日期时间。
auto_now和auto_now_add和default参数是互斥的，不能同时设置。
```

5.DatetimeField：日期时间字段，格式为YYYY-MM-DD HH:MM[:ss[.uuuuuu]][TZ]，相当于Python中的datetime.datetime的实例。

```
DateTimeField(DateField)
        - 日期+时间格式 YYYY-MM-DD HH:MM[:ss[.uuuuuu]][TZ]

    DateField(DateTimeCheckMixin, Field)
        - 日期格式       YYYY-MM-DD

    TimeField(DateTimeCheckMixin, Field)
        - 时间格式       HH:MM[:ss[.uuuuuu]]
```

6.BooleanField(Field)：布尔值类型

```
 NullBooleanField(Field):- 可以为空的布尔值
```

自定义字段

自定义二进制字段:

```
class UnsignedIntegerField(models.IntegerField):
    def db_type(self, connection):
        return 'integer UNSIGNED'
```

```
# PS: 返回值为字段在数据库中的属性。
# Django字段与数据库字段类型对应关系如下:
    'AutoField': 'integer AUTO_INCREMENT',
    'BigAutoField': 'bigint AUTO_INCREMENT',
    'BinaryField': 'longblob',
    'BooleanField': 'bool',
    'CharField': 'varchar(%(max_length)s)',
    'CommaSeparatedIntegerField': 'varchar(%(max_length)s)',
    'DateField': 'date',
    'DateTimeField': 'datetime',
    'DecimalField': 'numeric(%(max_digits)s, %(decimal_places)s)',
    'DurationField': 'bigint',
    'FileField': 'varchar(%(max_length)s)',
    'FilePathField': 'varchar(%(max_length)s)',
    'FloatField': 'double precision',
    'IntegerField': 'integer',
    'BigIntegerField': 'bigint',
    'IPAddressField': 'char(15)',
    'GenericIPAddressField': 'char(39)',
    'NullBooleanField': 'bool',
    'OneToOneField': 'integer',
    'PositiveIntegerField': 'integer UNSIGNED',
    'PositiveSmallIntegerField': 'smallint UNSIGNED',
    'SlugField': 'varchar(%(max_length)s)',
    'SmallIntegerField': 'smallint',
    'TextField': 'longtext',
    'TimeField': 'time',
    'UUIDField': 'char(32)',
```

自定义char类型字段:

```
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
```

字段参数：

```
null=True      数据库中该字段可以为空
blank=True     用户输入可以为空
default        默认值
db_index=True  索引
unique         唯一约束
verbose_name   显示的名称
choices        可选择的参数


db_column            数据库中字段的列名
primary_key          数据库中字段是否为主键
editable             Admin中是否可以编辑
help_text            Admin中该字段的提示信息
choices              Admin中显示选择框的内容，用不变动的数据放在内存中从而避免跨表操作，如：
gf = models.IntegerField(choices=[(0, '何穗'),(1, '大表姐'),],default=1)
```

使用admin的步骤：

1. 创建一个超级用户： python manage.py createsuperuser

   输入用户名和密码（自定义）

2. 在app文件夹下的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. 地址栏中输入ip+端口+/admin/

必知必会13条操作：

```
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   [ {} ]
# 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   [ () ]
# 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()
filter()
exclude()
values()    [ {},{} ]
values_list()    [ (),() ]
order_by()
reverse()
distinct()


返回对象
get
first
last

返回数字
count

返回布尔值
exists()
"""
```

```
<1> all():                  查询所有结果

<2> get(**kwargs):          返回与所给筛选条件相匹配的对象，返回结果有且只有一个，如果符合筛选条件的对象超过一个或者没有都会抛出错误。

<3> filter(**kwargs):       它包含了与所给筛选条件相匹配的对象

<4> exclude(**kwargs):      它包含了与所给筛选条件不匹配的对象

<5> values(*field):         返回一个ValueQuerySet——一个特殊的QuerySet，运行后得到的并不是一系列model的实例化对象，而是一个可迭代的字典序列

<6> values_list(*field):    它与values()非常相似，它返回的是一个元组序列，values返回的是一个字典序列

<7> order_by(*field):       对查询结果排序

<8> reverse():              对查询结果反向排序，请注意reverse()通常只能在具有已定义顺序的QuerySet上调用(在model类的Meta中指定ordering或调用order_by()方法)。

<9> distinct():             从返回结果中剔除重复纪录(如果你查询跨越多个表，可能在计算QuerySet时得到重复的结果。此时可以使用distinct()，注意只有在PostgreSQL中支持按字段去重。)

<10> count():               返回数据库中匹配查询(QuerySet)的对象数量。

<11> first():               返回第一条记录

<12> last():                返回最后一条记录

<13> exists():              如果QuerySet包含数据，就返回True，否则返回False
```

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
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)
```

Model Meta参数（不常用，了解格式）：

```
class UserInfo(models.Model):
    nid = models.AutoField(primary_key=True)
    username = models.CharField(max_length=32)

    class Meta:
        # 数据库中生成的表名称 默认 app名称 + 下划线 + 类名
        db_table = "table_name"

        # 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 = [
```

```
            ("pub_date", "deadline"),    # 应为两个存在的字段
        ]

        # 联合唯一索引
        unique_together = (("driver", "restaurant"),)    # 应为两个存在的字段
```

ORM操作：

基本操作

```
# 增
models.Tb1.objects.create(c1='xx', c2='oo')    # 增加一条数据，可以接受字典类型数据
**kwargs
obj = models.Tb1(c1='xx', c2='oo')
obj.save()


# 查
models.Tb1.objects.get(id=123)  # 获取单条数据，不存在则报错（不建议）
models.Tb1.objects.all()  # 获取全部
models.Tb1.objects.filter(name='seven')  # 获取指定条件的数据
models.Tb1.objects.exclude(name='seven')  # 去除指定条件的数据


# 删
# models.Tb1.objects.filter(name='seven').delete()  # 删除指定条件的数据


# 改
models.Tb1.objects.filter(name='seven').update(gender='0')    # 将指定条件的数据更
新，均支持 **kwargs
obj = models.Tb1.objects.get(id=1)
obj.c1 = '111'
obj.save()    # 修改单条数据
```

进阶操作：

```
# 获取个数
# models.Tb1.objects.filter(name='seven').count()

# 大于，小于
# models.Tb1.objects.filter(id__gt=1)              # 获取id大于1的值
# models.Tb1.objects.filter(id__gte=1)              # 获取id大于等于1的值
# models.Tb1.objects.filter(id__lt=10)             # 获取id小于10的值
# models.Tb1.objects.filter(id__lte=10)             # 获取id小于10的值
# models.Tb1.objects.filter(id__lt=10, id__gt=1)   # 获取id大于1 且 小于10的值

# 成员判断in
# models.Tb1.objects.filter(id__in=[11, 22, 33])   # 获取id等于11、22、33的数据
# models.Tb1.objects.exclude(id__in=[11, 22, 33])  # not in

# 是否为空 isnull
# Entry.objects.filter(pub_date__isnull=True)

# 包括contains
# models.Tb1.objects.filter(name__contains="ven")
# models.Tb1.objects.filter(name__icontains="ven") # icontains大小写不敏感
```

```
# models.Tb1.objects.exclude(name__icontains="ven")

# 范围range
# models.Tb1.objects.filter(id__range=[1, 2])   # 范围bettwen and

# 其他类似
# startswith，istartswith, endswith, iendswith,

# 排序order by
# models.Tb1.objects.filter(name='seven').order_by('id')    # asc
# models.Tb1.objects.filter(name='seven').order_by('-id')   # desc


# 分组group by
# from django.db.models import Count, Min, Max, Sum
# models.类名.objects.filter(c1=1).values('id').annotate(c=Count('num'))
对应的SQL语句
# SELECT "app01_tb1"."id", COUNT("app01_tb1"."num") AS "c" FROM "app01_tb1"
WHERE "app01_tb1"."c1" = 1 GROUP BY "app01_tb1"."id"

# limit 、offset
# models.类名.objects.all()[10:20]

# regex正则匹配，iregex 不区分大小写
# Entry.objects.get(title__regex=r'^(An?|The) +')
# Entry.objects.get(title__iregex=r'^(an?|the) +')

# date
# Entry.objects.filter(pub_date__date=datetime.date(2005, 1, 1))
# Entry.objects.filter(pub_date__date__gt=datetime.date(2005, 1, 1))

# year
# Entry.objects.filter(pub_date__year=2005)
# Entry.objects.filter(pub_date__year__gte=2005)

# month
# Entry.objects.filter(pub_date__month=12)
# Entry.objects.filter(pub_date__month__gte=6)

# day
# Entry.objects.filter(pub_date__day=3)
# Entry.objects.filter(pub_date__day__gte=3)

# week_day
# Entry.objects.filter(pub_date__week_day=2)
# Entry.objects.filter(pub_date__week_day__gte=2)

# hour
# Event.objects.filter(timestamp__hour=23)
# Event.objects.filter(time__hour=5)
# Event.objects.filter(timestamp__hour__gte=12)

# minute
# Event.objects.filter(timestamp__minute=29)
# Event.objects.filter(time__minute=46)
# Event.objects.filter(timestamp__minute__gte=29)

# second
```

```
# Event.objects.filter(timestamp__second=31)
# Event.objects.filter(time__second=2)
# Event.objects.filter(timestamp__second__gte=31)
```

QuerySet相关方法：

```
def all(self)
    # 获取所有的数据对象

def filter(self, *args, **kwargs)
    # 条件查询
    # 条件可以是：参数，字典，Q

def exclude(self, *args, **kwargs)
    # 条件查询
    # 条件可以是：参数，字典，Q

def select_related(self, *fields)
    性能相关：表之间进行join连表操作，一次性获取关联的数据。

    总结：
    1. select_related主要针一对一和多对一关系进行优化。
    2. select_related使用SQL的JOIN语句进行优化，通过减少SQL查询的次数来进行优化、提高性能。

def prefetch_related(self, *lookups)
    性能相关：多表连表操作时速度会慢，使用其执行多次SQL查询在Python代码中实现连表操作。

    总结：
    1. 对于多对多字段（ManyToManyField）和一对多字段，可以使用prefetch_related()来进行优化。
    2. prefetch_related()的优化方式是分别查询每个表，然后用Python处理他们之间的关系。

def annotate(self, *args, **kwargs)
    # 用于实现聚合group by查询

    from django.db.models import Count, Avg, Max, Min, Sum

    v = models.UserInfo.objects.values('u_id').annotate(uid=Count('u_id'))
    # SELECT u_id, COUNT(ui) AS `uid` FROM UserInfo GROUP BY u_id


v=models.UserInfo.objects.values('u_id').annotate(uid=Count('u_id')).filter(uid__gt=1)
    # SELECT u_id, COUNT(ui_id) AS `uid` FROM UserInfo GROUP BY u_id having count(u_id) > 1

v=models.UserInfo.objects.values('u_id').annotate(uid=Count('u_id',distinct=True)).filter(uid__gt=1)
    # SELECT u_id, COUNT( DISTINCT ui_id) AS `uid` FROM UserInfo GROUP BY u_id having count(u_id) > 1

def distinct(self, *field_names)
    # 用于distinct去重
    models.UserInfo.objects.values('nid').distinct()
    # select distinct nid from userinfo
```

```
    注：只有在PostgreSQL中才能使用distinct进行去重

def order_by(self, *field_names)
    # 用于排序
    models.UserInfo.objects.all().order_by('-id','age')

def extra(self, select=None, where=None, params=None, tables=None,
order_by=None, select_params=None)
    # 构造额外的查询条件或者映射，如：子查询

    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'])

 def reverse(self):
    # 倒序
    models.UserInfo.objects.all().order_by('-nid').reverse()
    # 注：如果存在order_by，reverse则是倒序，如果多个排序则一一倒序


 def defer(self, *fields):
    models.UserInfo.objects.defer('username','id')
    或
    models.UserInfo.objects.filter(...).defer('username','id')
    #映射中排除某列数据

 def only(self, *fields):
    #仅取某个表中的数据
     models.UserInfo.objects.only('username','id')
     或
     models.UserInfo.objects.filter(...).only('username','id')

 def using(self, alias):
     指定使用的数据库，参数为别名（setting中的设置）


##################################################
# PUBLIC METHODS THAT RETURN A QUERYSET SUBCLASS #
##################################################

def raw(self, raw_query, params=None, translations=None, using=None):
    # 执行原生SQL
    models.UserInfo.objects.raw('select * from userinfo')

    # 如果SQL是其他表时，必须将名字设置为当前UserInfo对象的主键列名
    models.UserInfo.objects.raw('select id as nid from 其他表')

    # 为原生SQL设置参数
    models.UserInfo.objects.raw('select id as nid from userinfo where nid>%s',
params=[12,])

    # 将获取的到列名转换为指定列名
    name_map = {'first': 'first_name', 'last': 'last_name', 'bd': 'birth_date',
'pk': 'id'}
    Person.objects.raw('SELECT * FROM some_other_table', translations=name_map)
```

```
    # 指定数据库
    models.UserInfo.objects.raw('select * from userinfo', using="default")

    ################### 原生SQL ###################
    from django.db import connection, connections
    cursor = connection.cursor()  # cursor = connections['default'].cursor()
    cursor.execute("""SELECT * from auth_user where id = %s""", [1])
    row = cursor.fetchone() # fetchall()/fetchmany(..)


def values(self, *fields):
    # 获取每行数据为字典格式

def values_list(self, *fields, **kwargs):
    # 获取每行数据为元祖

def dates(self, field_name, kind, order='ASC'):
    # 根据时间进行某一部分进行去重查找并截取指定内容
    # kind只能是："year"（年）, "month"（年-月）, "day"（年-月-日）
    # order只能是："ASC"  "DESC"
    # 并获取转换后的时间
        - year : 年-01-01
        - month: 年-月-01
        - day  : 年-月-日

    models.DatePlus.objects.dates('ctime','day','DESC')

def none(self):
    # 空QuerySet对象


####################################
# METHODS THAT DO DATABASE QUERIES #
####################################

def aggregate(self, *args, **kwargs):
   # 聚合函数，获取字典类型聚合结果
   from django.db.models import Count, Avg, Max, Min, Sum
   result = models.UserInfo.objects.aggregate(k=Count('u_id', distinct=True),
n=Count('nid'))
   ===> {'k': 3, 'n': 4}

def count(self):
   # 获取个数

def get(self, *args, **kwargs):
   # 获取单个对象

def create(self, **kwargs):
   # 创建对象

def bulk_create(self, objs, batch_size=None):
    # 批量插入
    # batch_size表示一次插入的个数
    objs = [
        models.DDD(name='r11'),
        models.DDD(name='r22')
```

```
    ]
    models.DDD.objects.bulk_create(objs, 10)

def get_or_create(self, defaults=None, **kwargs):
    # 如果存在，则获取，否则，创建
    # defaults 指定创建时，其他字段的值
    obj, created = models.UserInfo.objects.get_or_create(username='root1',
defaults={'email': '1111111','u_id': 2, 't_id': 2})

def update_or_create(self, defaults=None, **kwargs):
    # 如果存在，则更新，否则，创建
    # defaults 指定创建时或更新时的其他字段
    obj, created = models.UserInfo.objects.update_or_create(username='root1',
defaults={'email': '1111111','u_id': 2, 't_id': 1})

def first(self):
   # 获取第一个

def last(self):
   # 获取最后一个

def in_bulk(self, id_list=None):
   # 根据主键ID进行查找
   id_list = [11,21,31]
   models.DDD.objects.in_bulk(id_list)

def delete(self):
   # 删除

def update(self, **kwargs):
    # 更新

def exists(self):
   # 是否有结果
```

ForeignKey操作：

基于对象的查询

正向查询：

```
book_obj=models.类名.objects.get(条件)
print(book_obj.name)  #查找条件对应的书名
print(book_obj.publisher)  #查找条件关联的出版社信息
print(book_obj.publisher.条件)  #查找条件关联的出版社条件信息
```

反向查询：不指定related_name，直接使用表名_set方法：

```
pub_obj = models.类名.objects.get(条件唯一)

print(pub_obj)
print(pub_obj.book_set)   # 表名小写_set   关系管理对象，显示出版社出版的所有图书
print(pub_obj.book_set.all())
```

指定了related_name='xxxx'，使用指定后的名字

```
print(pub_obj.xxxx)
print(pub_obj.xxxx.all())
```

基于字段查询：

```
# 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')
```

ManyToManyField（多对多操作）：

当点后面的对象 可能存在多个的时候就可以使用以下的方法。

create： 创建一个新的对象，保存对象，并将它添加到关联对象集之中，返回新创建的对象。

基于对象的：

get：

```
指定了related_name=‘books’
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())
```

```
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])))  #*打散
```

基于字段的：

filter：

```
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，格式为：[id，id] [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，格式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：移除多对多的关系

```
author_obj.books.remove(1,3)
author_obj.books.remove(*models.Book.objects.filter(pk__in=[2,4]))
对于ForeignKey对象，clear()和remove()方法仅在null=True时存在。
```

clear：清除所有的多对多的关系

```
author_obj.books.clear()
author_obj.books.set([])
对于ForeignKey对象，clear()和remove()方法仅在null=True时存在。
```

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)
```

注意事项：

```
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"))

对于所有类型的关联字段，add()、create()、remove()和clear(),set()都会马上更新数据库。换句话说，在关联的任何一端，都不需要再调用save()方法
```

聚合和分组：

导入：from django.db.models import Max, Min, Avg, Count, Sum

聚合：`aggregate()`是`QuerySet`的一个终止子句，意思是说，它返回一个包含一些键值对的字典。

```
示例1：
>>> from django.db.models import Avg, Sum, Max, Min, Count
>>> models.Book.objects.all().aggregate(Avg("price"))
{'price__avg': 13.233333}

如果你想要为聚合值指定一个名称，可以向聚合子句提供它。
>>> models.Book.objects.aggregate(average_price=Avg('price'))
{'average_price': 13.233333}

如果你希望生成不止一个聚合，你可以向aggregate()子句中添加另一个参数。所以，如果你也想知道所有图书价格的最大值和最小值，可以这样查询：
>>> models.Book.objects.all().aggregate(Avg("price"), Max("price"),
Min("price"))
{'price__avg': 13.233333, 'price__max': Decimal('19.90'), 'price__min':
Decimal('9.90')}
```

分组：annotate

```
原生SQL语句与ORM操作对比：
原生SQL语句：select dept,AVG(salary) from employee group by dept;
ORM查询:from django.db.models import Avg
Employee.objects.values("dept").annotate(avg=Avg("salary").values("dept", "avg")
```

```
SQL查询：select dept.name,AVG(salary) from employee inner join dept on
(employee.dept_id=dept.id) group by dept_id;
ORM查询：from django.db.models import Avg
models.Dept.objects.annotate(avg=Avg("employee__salary")).values("name", "avg")
```

F查询和Q查询：from django.db.models import F, Q

F查询：两个字段的值进行比较

```
ret = models.Book.objects.filter(sale__gt=F('kucun'))  #查询卖的大于库存的书

ret = models.Book.objects.all().update(publisher_id=3)   #更新所有的书出版社为id=3

ret = models.Book.objects.filter(pk=1).update(sale=F('sale')*2+43)  # 进行加减乘除运算
```

Q查询：`filter()`等方法中的关键字参数查询都是一起进行“AND” 的。 如果你需要执行更复杂的查询（例如`OR`语句），你可以使用`Q`对象。

你可以组合`&`和`|`操作符以及使用括号进行分组来编写任意复杂的`Q`对象。同时，`Q`对象可以使用`~`操作符取反，这允许组合正常的查询和取反(`NOT`)查询。

```
# & 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]))
```

查询函数可以混合使用`Q 对象`和关键字参数。所有提供给查询函数的参数（关键字参数或`Q`对象）都将"AND"在一起。但是，如果出现`Q`对象，它必须位于所有关键字参数的前面。

例如：查询出版年份是2017或2018，书名中带物语的所有书。

```
>>> models.Book.objects.filter(Q(publish_date__year=2018) |
Q(publish_date__year=2017), title__icontains="物语")
<QuerySet [<Book: 番茄物语>, <Book: 香蕉物语>, <Book: 橘子物语>]>
```

事务：

```
from django.db import transaction

try:   #try一定要写在with外面
    with transaction.atomic():
        # 一系列的操作
        models.Book.objects.update(publisher_id=4)
        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)
```

在Python脚本中调用Django环境：

```
写在settings文件中，打印日志文件

import os
if __name__ == '__main__':
    os.environ.setdefault("DJANGO_SETTINGS_MODULE", "BMS.settings")
    import django
    django.setup()

    from app01 import models
    books = models.Book.objects.all()
    print(books)
```

Cookie和Session：

Cookie：

1.什么是Cookie：保存在浏览器本地上的一组组键值对

HTTP协议是无状态的，每次请求之间都是相互独立的，之间没有关系，没有办法保存状态。 Cookie具体指的是一段小信息，它是服务器发送出来存储在浏览器上的一组组键值对，下次访问服务器时浏览器会自动携带这些键值对，以便服务器提取有用信息。

2.Cookie的原理：

由服务器产生内容，浏览器收到请求后保存在本地；当浏览器再次访问时，浏览器会自动带上Cookie，这样服务器就能通过Cookie的内容来判断这个是"谁"了。

3.特性：

1. 服务器让浏览器进行设置的
2. 保存在浏览器本地的
3. 下次访问时自动携带相应的Cookie

4.Django中如何操作Cookie：

4.1：获取

```
request.COOKIES['key']
request.COOKIES[].get
request.get_signed_cookie('key', default=RAISE_ERROR, salt='', max_age=None)#最大时间是安按秒来计算的，最好是设置default=‘’，不然就会弹出报错信息
```

get_signed_cookie方法的参数：

- default: 默认值
- salt: 加密盐
- max_age: 后台控制过期时间

4.2：设置Cookie

```
rep = HttpResponse(...)
rep = render(request, ...)

rep.set_cookie(key,value,...)
rep.set_signed_cookie(key,value,salt='加密盐',...)
```

参数：

- key, 键
- value='', 值
- max_age=None, 超时时间
- expires=None, 超时时间(IE requires expires, so set it if hasn't been already.)
- path='/', Cookie生效的路径，/ 表示根路径，特殊的：根路径的cookie可以被任何url的页面访问
- domain=None, Cookie生效的域名
- secure=False, https传输
- httponly=False 只能http协议传输，无法被JavaScript获取（不是绝对，底层抓包可以获取到也可以被覆盖）

4.3：删除Cookie

```
response.delete_cookie(key)
```

4.4:Cookie版登陆校验

```
def check_login(func):
    @wraps(func)
    def inner(request, *args, **kwargs):
        next_url = request.get_full_path()
        if request.get_signed_cookie("login", salt="SSS", default=None) ==
"yes":
            # 已经登录的用户...
            return func(request, *args, **kwargs)
        else:
            # 没有登录的用户，跳转刚到登录页面
            return redirect("/login/?next={}".format(next_url))
```

```
    return inner


def login(request):
    if request.method == "POST":
        username = request.POST.get("username")
        passwd = request.POST.get("password")
        if username == "xxx" and passwd == "dashabi":
            next_url = request.GET.get("next")
            if next_url and next_url != "/logout/":
                response = redirect(next_url)
            else:
                response = redirect("/class_list/")
            response.set_signed_cookie("login", "yes", salt="SSS")
            return response
    return render(request, "login.html")
```

Session：保存在服务器上的一组组键值对，必须依赖Cookie

Session的由来：

Cookie虽然在一定程度上解决了“保持状态”的需求，但是由于Cookie本身最大支持4096字节，以及Cookie本身保存在客户端，可能被拦截或窃取，因此就需要有一种新的东西，它能支持更多的字节，并且他保存在服务器，有较高的安全性。这就是Session。

Cookie弥补了HTTP无状态的不足，让服务器知道来的人是“谁”；但是Cookie以文本的形式保存在本地，自身安全性较差；所以我们就通过Cookie识别不同的用户，对应的在Session里保存私密的信息以及超过4096字节的文本。

上述所说的Cookie和Session其实是共通性的东西，不限于语言和框架。

Django中Session的使用方法：

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，现根据浏览器生成一个唯一标识（session_key），并且有超时时间
3. 返回cookie session_id=唯一标识

获取，设置，删除：

```
request.session['k1']
request.session.get('k1',None)
request.session['k1'] = 123
request.session.setdefault('k1',123) # 存在则不设置
del request.session['k1']
request.session.pop('k1')
```

会话中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() #删除已经失效的session数据
```

检查会话Session的key在数据中是否存在：

```
request.session.exists('key')
```

删除当前会话的所有session数据

```
request.session.delete()
```

删除当前会话数据并删除会话的cookie

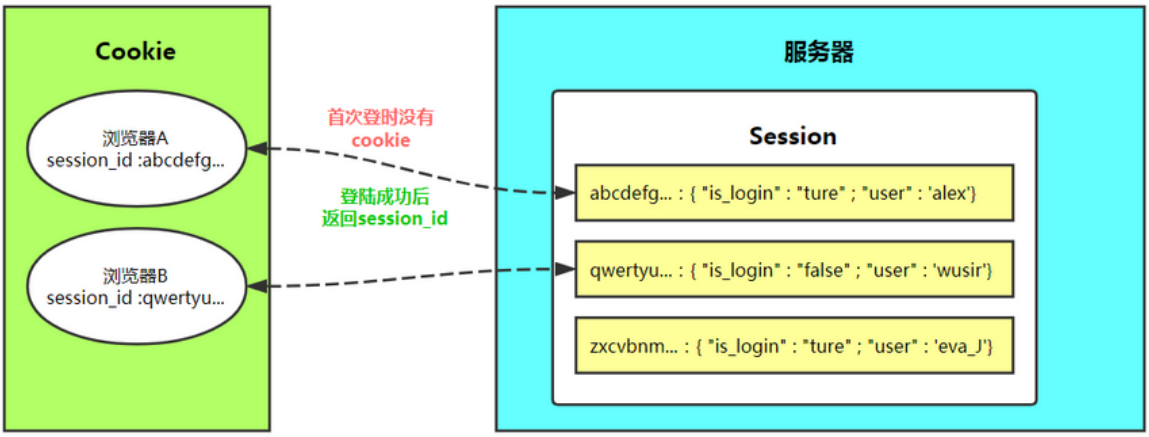
```
request.session.flush()
用于确保前面的会话数据不可以再次被用户的浏览器访问
```

设置会话Session和cookie的超时时间

```
request.session.set_expiry(value)

注意事项：
    如果value是个整数，session会在秒数后失效。
    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流程解析：



Session版登录验证：

```
from functools import wraps

def check_login(func):
    @wraps(func)
    def inner(request, *args, **kwargs):
        next_url = request.get_full_path()
        if request.session.get("user"):
            return func(request, *args, **kwargs)
        else:
            return redirect("/login/?next={}".format(next_url))
    return inner

def login(request):
    if request.method == "POST":
        user = request.POST.get("user")
```

```
        pwd = request.POST.get("pwd")

        if user == "alex" and pwd == "alex1234":
            # 设置session
            request.session["user"] = user
            # 获取跳到登陆页面之前的URL
            next_url = request.GET.get("next")
            # 如果有，就跳转回登陆之前的URL
            if next_url:
                return redirect(next_url)
            # 否则默认跳转到index页面
            else:
                return redirect("/index/")
    return render(request, "login.html")

@check_login
def logout(request):
    # 删除所有当前请求相关的session
    request.session.delete()
    return redirect("/login/")

@check_login
def index(request):
    current_user = request.session.get("user", None)
    return render(request, "index.html", {"user": current_user})
```

Django中的Session配置（了解）：

```
from django.conf import global_settings
1. 数据库Session
SESSION_ENGINE = 'django.contrib.sessions.backends.db'   # 引擎（默认）

2. 缓存Session
SESSION_ENGINE = 'django.contrib.sessions.backends.cache'  # 引擎
SESSION_CACHE_ALIAS = 'default'                            # 使用的缓存别名（默认内存缓存，也可以是memcache），此处别名依赖缓存的设置

3. 文件Session
SESSION_ENGINE = 'django.contrib.sessions.backends.file'    # 引擎
SESSION_FILE_PATH = None                                    # 缓存文件路径，如果为None，则使用tempfile模块获取一个临时地址tempfile.gettempdir()

4. 缓存+数据库
SESSION_ENGINE = 'django.contrib.sessions.backends.cached_db'        # 引擎

5. 加密Cookie Session
SESSION_ENGINE = 'django.contrib.sessions.backends.signed_cookies'   # 引擎

其他公用设置项：
SESSION_COOKIE_NAME ＝ "sessionid"                       # Session的cookie保存在浏览器上时的key，即：sessionid＝随机字符串（默认）
SESSION_COOKIE_PATH ＝ "/"                               # Session的cookie保存的路径（默认）
SESSION_COOKIE_DOMAIN = None                             # Session的cookie保存的域名（默认）
SESSION_COOKIE_SECURE = False                            # 是否Https传输cookie（默认）
```

```
SESSION_COOKIE_HTTPONLY = True                          # 是否Session的cookie只支持http传输（默认）
SESSION_COOKIE_AGE = 1209600                            # Session的cookie失效日期（2周）（默认）
SESSION_EXPIRE_AT_BROWSER_CLOSE = False                 # 是否关闭浏览器使得Session过期（默认）
SESSION_SAVE_EVERY_REQUEST = False                      # 是否每次请求都保存Session，默认修改之后才保存（默认）
```

中间件：from django.utils.deprecation import MiddlewareMixin

什么是中间件：就是一个类，在全局范围内处理Django的请求和响应

官方的说法：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。

但是由于其影响的是全局，所以需要谨慎使用，使用不当会影响性能。

说的直白一点中间件是帮助我们在视图函数执行之前和执行之后都可以做一些额外的操作，它本质上就是一个自定义类，类中定义了几个方法，Django框架会在处理请求的特定的时间去执行这些方法。

使用中间件：5个方法，四个特点

返回值可以是None或一个HttpResponse对象，如果是None，则继续按照django定义的规则向后继续执行，如果是HttpResponse对象，则直接将该对象返回给用户。

使用自定义中间件之前必须先去settings文件中的 MIDDLEWARE配置项中注册上述自定义中间件：

```
示例：
MIDDLEWARE = [
    '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',
    'middlewares.MD1',  # 自定义中间件MD1
    'middlewares.MD2'  # 自定义中间件MD2
]
```

process_request(self,request)：（ request和视图函数中的request是一样的。 ）

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。

注意事项：

1. 中间件的process_request方法是在执行视图函数之前执行的。
2. 当配置多个中间件时，会按照MIDDLEWARE中的注册顺序，也就是列表的索引值，从前到后依次执行的。
3. 不同中间件之间传递的request都是同一个对象
4. 如果要使用session中间件的话新注册的中间件必须在session中间件之后

process_response(self, request, response)：（response是视图函数返回的HttpResponse对象。该方法的返回值也必须是HttpResponse对象。）

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，可以返回response或者自定义的内容

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

四个参数

request是HttpRequest对象。

view_func是Django即将使用的视图函数。（它是实际的函数对象，而不是函数的名称作为字符串。）

view_args是将传递给视图的位置参数的列表.

view_kwargs是将传递给视图的关键字参数的字典。 view_args和view_kwargs都不包含第一个视图参数（request）。

Django会在调用视图函数之前调用process_view方法。

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(self, request, exception)

一个HttpRequest对象

一个exception是视图函数异常产生的Exception对象。

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： 交给下一个中间件的process_exception方法来处理异常，所有的中间件都没有处理，django处理错误。

HttpResponse：Django将调用模板和中间件中的process_response方法，并返回给浏览器，否则将默认处理异常，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(self,request,response)

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

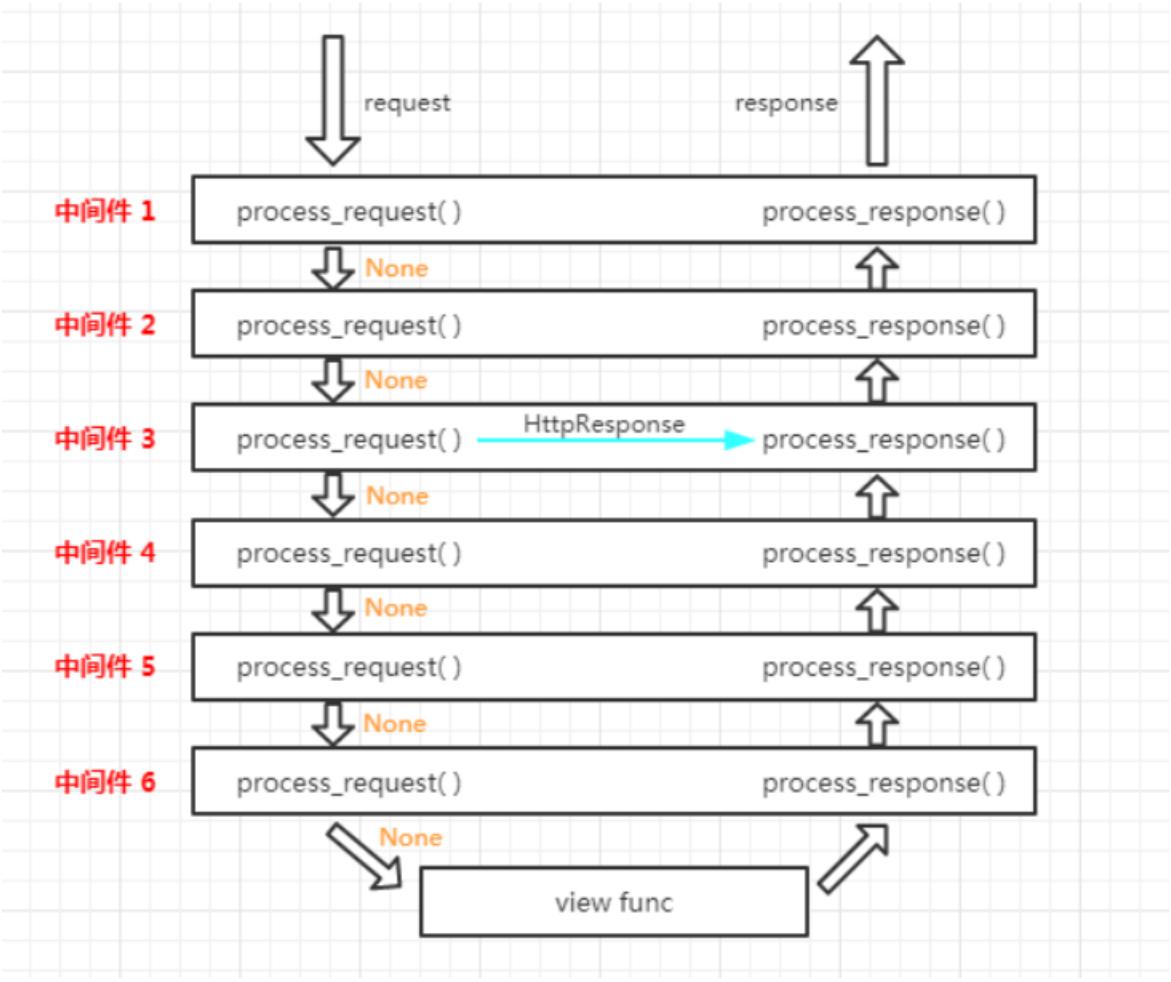
response：响应的对象

返回值：

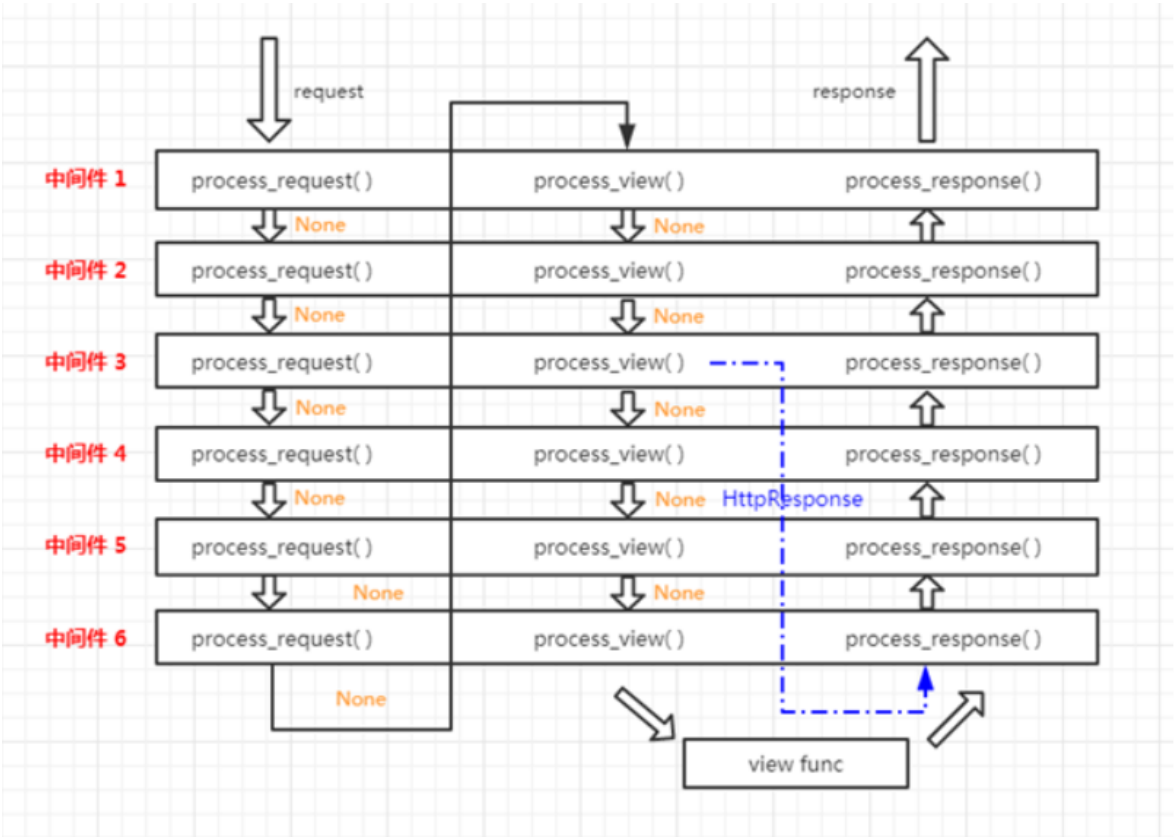
HttpResponse: 必须返回

执行流程：

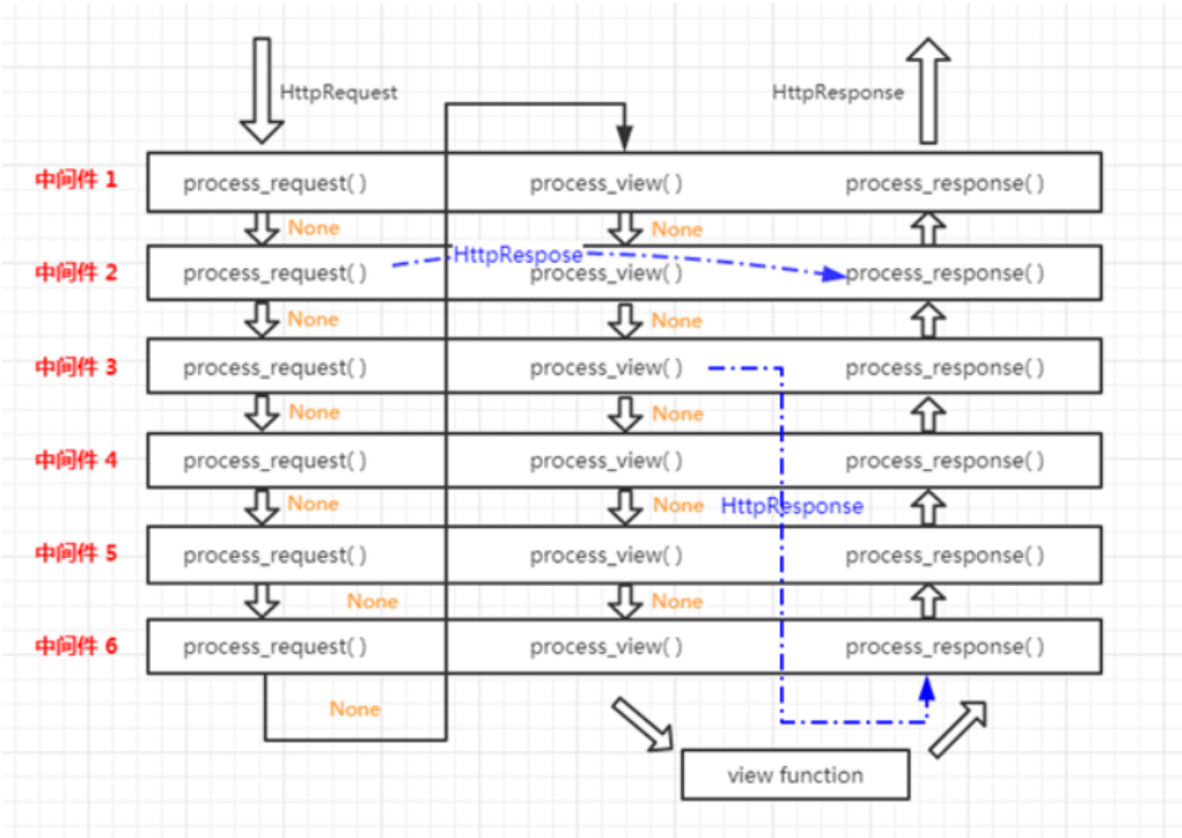
请求到达中间件之后，先按照正序执行每个注册中间件的process_reques方法，process_request方法返回的值是None，就依次执行，如果返回的值是HttpResponse对象，不再执行后面的process_request方法，而是执行当前对应中间件的process_response方法，将HttpResponse对象返回给浏览器。也就是说：如果MIDDLEWARE中注册了6个中间件，执行过程中，第3个中间件返回了一个HttpResponse对象，那么第4,5,6中间件的process_request和process_response方法都不执行，顺序执行3,2,1中间件的process_response方法。

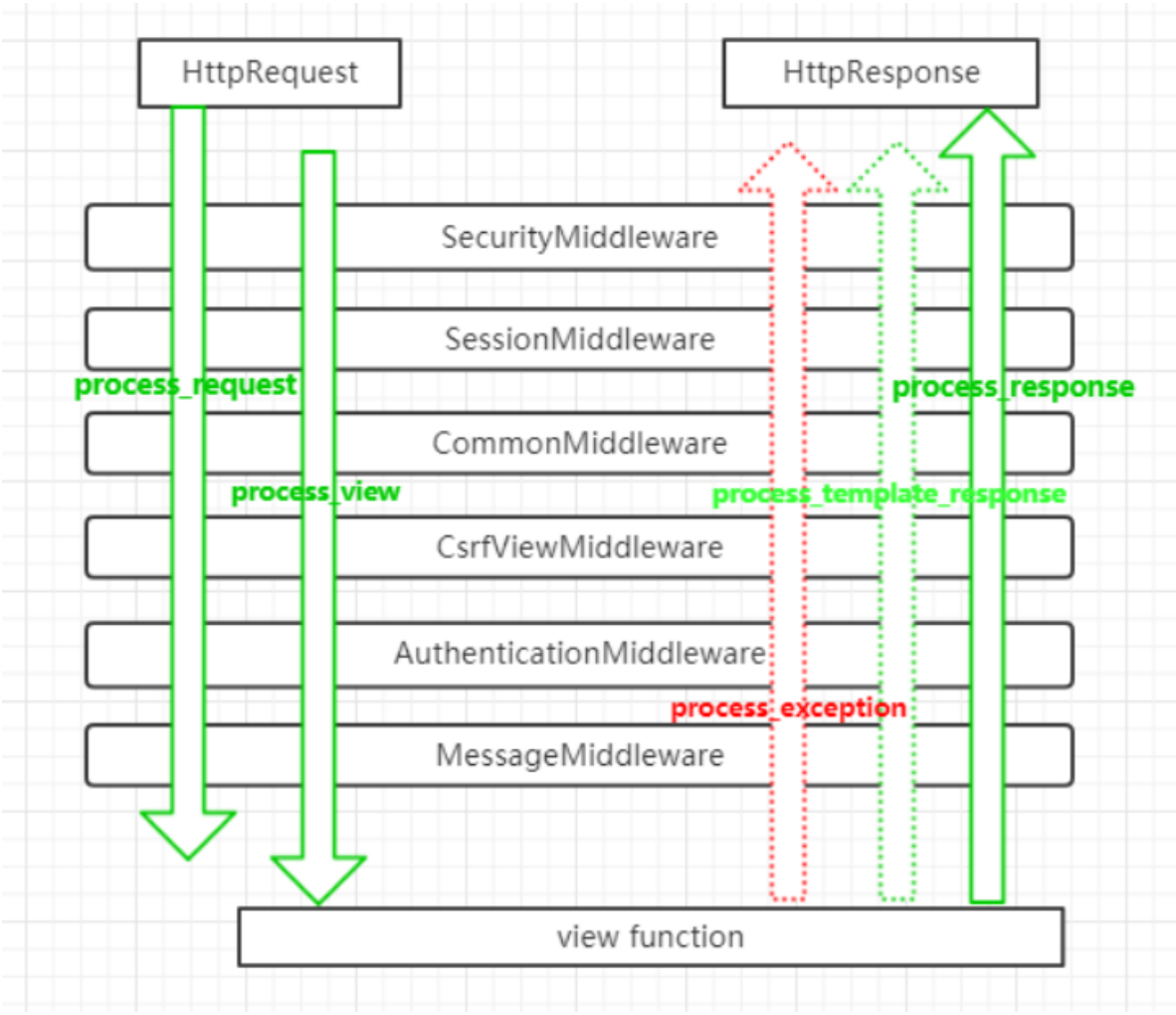


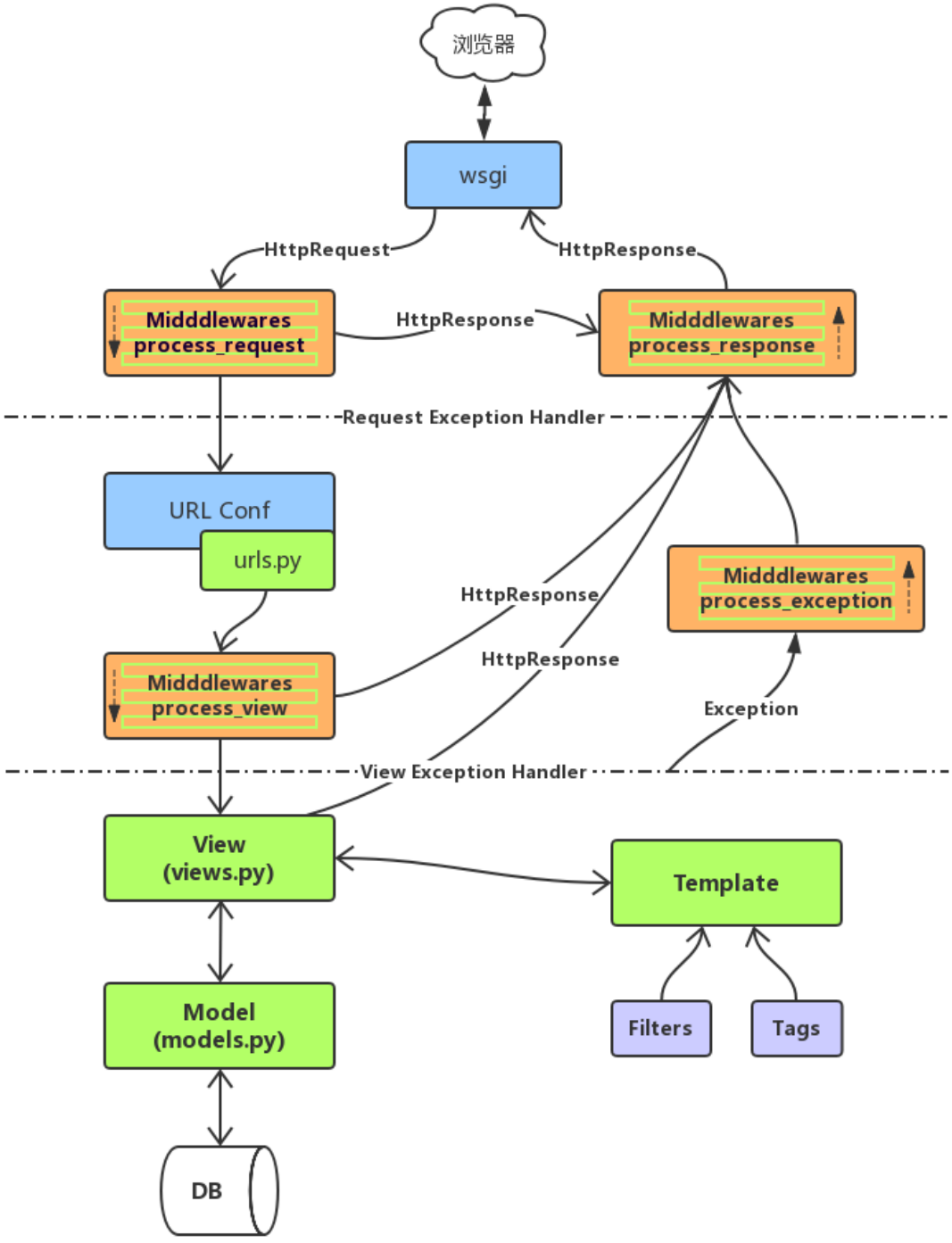
process_request方法都执行完后，匹配路由，找到要执行的视图函数，先不执行视图函数，先执行中间件中的process_view方法，process_view方法返回None，继续按顺序执行，所有process_view方法执行完后执行视图函数。假如中间件3 的process_view方法返回了HttpResponse对象，则4,5,6的process_view以及视图函数都不执行，直接从最后一个中间件，也就是中间件6的process_response方法开始倒序执行。



process_template_response和process_exception两个方法的触发是有条件的，执行顺序也是倒序。
总结所有的执行流程如下：




HttpRequest
HttpResponse
SecurityMiddleware
SessionMiddleware
process_request
process_response
CommonMiddleware
process_view
process_template_response
CsrfViewMiddleware
AuthenticationMiddleware
process_exception
MessageMiddleware
view function


浏览器
wsgi
HttpRequest
HttpResponse
Midddlewares
process_request
HttpResponse
Midddlewares
process_response
Request Exception Handler
URL Conf
urls.py
HttpResponse
Midddlewares
process_exception
Midddlewares
process_view
HttpResponse
Exception
View Exception Handler
View
(views.py)
Template
Model
(models.py)
Filters
Tags
DB

AJAX:

stringify与parse方法：json对象和字符串转换的两个方法

JSON.parse(): 用于将一个 JSON 字符串转换为 JavaScript 对象

```
JSON.parse('{"name":"alex"}');
```

JSON.stringify(): 用于将 JavaScript 值转换为 JSON 字符串。

```
JSON.stringify({"name":"alex"})
```

ajax简介：js的技术，就是发送请求接收响应的技术，传输少量的数据

含义：

AJAX（Asynchronous Javascript And XML）翻译成中文就是“异步的Javascript和XML”。即使用Javascript语言与服务器进行异步交互，传输的数据为XML（当然，传输的数据不只是XML）。

异步交互：客户端发出一个请求后，无需等待服务器响应结束，就可以发出第二个请求。

优点：

1. 局部刷新
2. 传输的数据量少，响应内容不再是整个页面，所以ajax性能更高
3. 异步： 不重新加载整个页面的情况下，可以与服务器交换数据并更新部分网页内容

网页发送请求的途径：

1. 通过a标签获取到get请求
2. 地址栏输入信息得到get请求
3. form表单可以获取get/post请求

应用场景：

1. 根据用户输入的内容搜索引擎自动提示搜索关键字
2. 注册的时候的用户名查重（当用户输入完成之后可以不用提交，鼠标移动之后网页自动验证，整个过程中网页没有刷新，只刷新输入部分）

简单使用：

```
$.ajax({
            'url':'/calc/',
            'type':'post',
            'data':{
                'k1':$('[name="i1"]').val(),
                'k2':$('[name="i2"]').val(),
            },
            success:function (ret) {
               $('[name="i3"]').val(ret)
            }
        })
```

参数

```
$.ajax({
            url: '/test/',      #  提交的地址
            type: 'post',       #  请求方式
            data: {             #  提交的数据
                name: 'alex',
                age: 73,
                hobby: JSON.stringify(['装逼', '画饼', '上过北大'])
            },
            success: function (ret) {       # 响应成功的回调函数

            },
            error:function (ret) {          # 响应失败的回调函数
                console.log(ret)
            }
        })
```

上传文件

```
$('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)
            }

        })

    })
```

ajax能通过django的csrf的校验：

前提必须有cookie：

1. {% csrf_token %}
2. 给视图加装饰器

```
from django.views.decorators.csrf import ensure_csrf_cookie

@ensure_csrf_cookie
def index(request):
```

方式一：data中添加键值对 csrfmiddlewaretoken

```
$.ajax({
            'url': '/calc/',
            'type': 'post',
            'data': {
                'csrfmiddlewaretoken': $('[name="csrfmiddlewaretoken"]').val(),
                'k1': $('[name="i1"]').val(),
                'k2': $('[name="i2"]').val(),
            },
            success: function (ret) {
                $('[name="i3"]').val(ret)
            }
        })
```

方式二：加请求头 x-csrftoken

```
$('#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：

```
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
```

导入文件：写在静态文件中，直接导入html模版中

```
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);
    }
  }
});
```

form组件：

功能：

1. 生成页面所用的html标签
2. 对用户提交的数据进行校验
3. 保留上次输入的内容，返回错误提示

使用form组件示例注册功能校验

定义类：

```
views：

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


# 使用form组件实现注册方式
```

```
def register2(request):
    form_obj = RegForm()
    if request.method == "POST":
        # 实例化form对象的时候，把post提交过来的数据直接传进去
        form_obj = RegForm(request.POST)
        # 调用form_obj校验数据的方法
        if form_obj.is_valid():
            return HttpResponse("注册成功")
    return render(request, "register2.html", {"form_obj": form_obj})
```

html模版：

```
<body>
    <form action="/reg2/" method="post" novalidate autocomplete="off">
        {% csrf_token %}
        <div>
            <label for="{{ form_obj.name.id_for_label }}">{{ form_obj.name.label
}}</label>
            {{ form_obj.name }} {{ form_obj.name.errors.0 }}
        </div>
        <div>
            <label for="{{ form_obj.pwd.id_for_label }}">{{ form_obj.pwd.label
}}</label>
            {{ form_obj.pwd }} {{ form_obj.pwd.errors.0 }}
        </div>
        <div>
            <input type="submit" class="btn btn-success" value="注册">
        </div>
    </form>
</body>
```

```
{{ 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
```

常用字段： 字段用于对用户请求数据的验证，插件用于自动生成HTML;

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

initial：设置默认值

error_messages：重写错误信息

```
error_messages={
            "required": "不能为空",
            "invalid": "格式错误",
            "min_length": "用户名最短8位"
        }
```

password：设置密码输入为密文

```
 widget=forms.widgets.PasswordInput(attrs={'class': 'c1'}, render_value=True)
```

radioSelect：勾选框

```
gender = forms.fields.ChoiceField(
        choices=((1, "男"), (2, "女"), (3, "保密")),
        label="性别",
        initial=3,
        widget=forms.widgets.RadioSelect()
```

select：单选值

```
   hobby = forms.fields.ChoiceField(
        choices=((1, "篮球"), (2, "足球"), (3, "双色球"), ),
        label="爱好",
        initial=3,
        widget=forms.widgets.Select()
```

多选select：

```
hobby = forms.fields.MultipleChoiceField(
        choices=((1, "篮球"), (2, "足球"), (3, "双色球"), ),
        label="爱好",
        initial=[1, 3],
        widget=forms.widgets.SelectMultiple()
    )
```

checkbox：单选框

```
keep = forms.fields.ChoiceField(
        label="是否记住密码",
        initial="checked",
        widget=forms.widgets.CheckboxInput()
    )
```

多选checkbox：

```
hobby = forms.fields.MultipleChoiceField(
        choices=((1, "篮球"), (2, "足球"), (3, "双色球"),),
        label="爱好",
        initial=[1, 3],
        widget=forms.widgets.CheckboxSelectMultiple()
```

choice注意事项：

在使用选择标签时，需要注意choices的选项可以从数据库中获取，但是由于是静态字段 ***获取的值无法实时更新***，那么需要自定义构造方法从而达到此目的。

方式一：

```
from django.forms import Form
from django.forms import widgets
from django.forms import fields


class MyForm(Form):

    user = fields.ChoiceField(
        # choices=((1, '上海'), (2, '北京'),),
        initial=2,
        widget=widgets.Select
    )

    def __init__(self, *args, **kwargs):
        super(MyForm,self).__init__(*args, **kwargs)
        # self.fields['user'].choices = ((1, '上海'), (2, '北京'),)
        # 或
        self.fields['user'].choices =
models.Classes.objects.all().values_list('id','caption')
```

方式二：

```
from django import forms
from django.forms import fields
from django.forms import models as form_model


class FInfo(forms.Form):
    authors =
form_model.ModelMultipleChoiceField(queryset=models.NNewType.objects.all())  #
多选
    # authors =
form_model.ModelChoiceField(queryset=models.NNewType.objects.all())  # 单选
```

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=[],                自定义验证规则
```

```
Field
    required=True,                是否允许为空
    widget=None,                  HTML插件
    label=None,                   用于生成Label标签或显示内容
    initial=None,                 初始值
```

```
    help_text='',                   帮助信息(在标签旁边显示)
    error_messages=None,            错误信息 {'required': '不能为空', 'invalid': '格式
错误'}
    validators=[],                  自定义验证规则
    localize=False,                 是否支持本地化
    disabled=False,                 是否可以编辑
    label_suffix=None               Label内容后缀


CharField(Field)
    max_length=None,                最大长度
    min_length=None,                最小长度
    strip=True                      是否移除用户输入空白

IntegerField(Field)
    max_value=None,                 最大值
    min_value=None,                 最小值

FloatField(IntegerField)
    ...

DecimalField(IntegerField)
    max_value=None,                 最大值
    min_value=None,                 最小值
    max_digits=None,                总长度
    decimal_places=None,            小数位长度

BaseTemporalField(Field)
    input_formats=None              时间格式化

DateField(BaseTemporalField)    格式：2015-09-01
TimeField(BaseTemporalField)    格式：11:12
DateTimeField(BaseTemporalField)格式：2015-09-01 11:12

DurationField(Field)            时间间隔：%d %H:%M:%S.%f
    ...

RegexField(CharField)
    regex,                          自定制正则表达式
    max_length=None,                最大长度
    min_length=None,                最小长度
    error_message=None,             忽略，错误信息使用 error_messages={'invalid':
'...'}

EmailField(CharField)
    ...

FileField(Field)
    allow_empty_file=False      是否允许空文件

ImageField(FileField)
    ...
    注：需要PIL模块，pip3 install Pillow
    以上两个字典使用时，需要注意两点：
        - form表单中 enctype="multipart/form-data"
        - view函数中 obj = MyForm(request.POST, request.FILES)

URLField(Field)
```

```
    ...


BooleanField(Field)
    ...

NullBooleanField(BooleanField)
    ...

ChoiceField(Field)
    ...
    choices=(),                 选项，如：choices = ((0,'上海'),(1,'北京'),)
    required=True,              是否必填
    widget=None,                插件，默认select插件
    label=None,                 Label内容
    initial=None,               初始值
    help_text='',               帮助提示


ModelChoiceField(ChoiceField)
    ...                         django.forms.models.ModelChoiceField
    queryset,                   # 查询数据库中的数据
    empty_label="---------",    # 默认空显示内容
    to_field_name=None,         # HTML中value的值对应的字段
    limit_choices_to=None       # ModelForm中对queryset二次筛选

ModelMultipleChoiceField(ModelChoiceField)
    ...                         django.forms.models.ModelMultipleChoiceField



TypedChoiceField(ChoiceField)
    coerce = lambda val: val    对选中的值进行一次转换
    empty_value= ''             空值的默认值

MultipleChoiceField(ChoiceField)
    ...

TypedMultipleChoiceField(MultipleChoiceField)
    coerce = lambda val: val    对选中的每一个值进行一次转换
    empty_value= ''             空值的默认值

ComboField(Field)
    fields=()                   使用多个验证，如下：即验证最大长度20，又验证邮箱格式
                                fields.ComboField(fields=
[fields.CharField(max_length=20), fields.EmailField(),])

MultiValueField(Field)
    PS: 抽象类，子类中可以实现聚合多个字典去匹配一个值，要配合MultiWidget使用

SplitDateTimeField(MultiValueField)
    input_date_formats=None,    格式列表：['%Y--%m--%d', '%m%d/%Y', '%m/%d/%y']
    input_time_formats=None     格式列表：['%H:%M:%S', '%H:%M:%S.%f', '%H:%M']

FilePathField(ChoiceField)      文件选项，目录下文件显示在页面中
    path,                       文件夹路径
    match=None,                 正则匹配
    recursive=False,            递归下面的文件夹
```

```
    allow_files=True,           允许文件
    allow_folders=False,        允许文件夹
    required=True,
    widget=None,
    label=None,
    initial=None,
    help_text=''

GenericIPAddressField
    protocol='both',            both,ipv4,ipv6支持的IP格式
    unpack_ipv4=False           解析ipv4地址，如果是::ffff:192.0.2.1时候，可解析为
192.0.2.1，  PS：protocol必须为both才能启用

SlugField(CharField)            数字，字母，下划线，减号（连字符）
    ...

UUIDField(CharField)            uuid类型
```

校验

自定义校验：

方式一：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,])
```

方式二：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')])
```

钩子校验：

局部钩子

```
def clean_username(self):
        # 局部钩子
        # 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
```

全局钩子：

```
    def clean(self):
        # 全局钩子
        # 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')
```