Django

from django.shortcuts import render,HttpResponse

from django.urls import path,reverse,include

## 一、HTTP请求协议与响应协议

### （一）请求协议

服务器通过\r\n来区分首行、请求头的内容。

通过\r\n\r\n来区分内容实体（请求体），即在\r\n\r\n后面是请求体。

#### 请求方式: get与post请求

* GET提交的数据会放在URL之后，以?分割URL和传输数据，参数之间以&相连，如EditBook?name=test1&id=123456. POST方法是把提交的数据放在HTTP包的Body中.
* GET提交的数据大小有限制（因为浏览器对URL的长度有限制），而POST方法提交的数据没有限制.
* GET与POST请求在服务端获取请求数据方式不同。
* GET方式提交数据，会带来安全问题，比如一个登录页面，通过GET方式提交数据时，用户名和密码将出现在URL上，如果页面可以被缓存或者其他人可以访问这台机器，就可以从历史记录获得该用户的账号和密码.

get请求主要用于查询操作，无请求体，请求内容会放在url后面，以？隔开，参数之间以&相连，因此不保密。

post请求主要用于数据库更改等操作。

只有post请求才有请求体。post请求无长度限制，而get有长度限制。

### （二）响应协议

### （三）响应状态码



URL：

协议://IP:端口（80）/路径？a=1&b=2

## 二、wsgiref模块

负责http协议来组装和封装数据。

### （一）make\_sever

make\_sever模块

封装socket，等待链接

def application（environ, start\_response）：

pass

*# 封装socket*httped=make\_server(**""**,8060,application)  
  
*# 等待用户连接: conn,addr=sock.accept()*httped.serve\_forever() *# application(environ, start\_response)*

application为自定义的回调函数。当有用户链接，发送请求时，会执行application函数。会自动向回调函数传入两个参数，

1.environ：按着http协议解析的数据，是个字典形式。其中”PATH\_INFO”为请求URL中含有的路径部分。

environ中含有的key：

ALLUSERSPROFILE

APPDATA

COMMONPROGRAMFILES

COMMONPROGRAMFILES(X86)

COMMONPROGRAMW6432

COMPUTERNAME

COMSPEC

FPS\_BROWSER\_APP\_PROFILE\_STRING

FPS\_BROWSER\_USER\_PROFILE\_STRING

HOMEDRIVE

HOMEPATH

LOCALAPPDATA

LOGONSERVER

NUMBER\_OF\_PROCESSORS

OS

PATH

PATHEXT

PROCESSOR\_ARCHITECTURE

PROCESSOR\_IDENTIFIER

PROCESSOR\_LEVEL

PROCESSOR\_REVISION

PROGRAMDATA

PROGRAMFILES

PROGRAMFILES(X86)

PROGRAMW6432

PSMODULEPATH

PUBLIC

PYCHARM\_HOSTED

PYCHARM\_MATPLOTLIB\_PORT

PYTHONIOENCODING

PYTHONPATH

PYTHONUNBUFFERED

SESSIONNAME

SYSTEMDRIVE

SYSTEMROOT

TEMP

TMP

USERDOMAIN

USERDOMAIN\_ROAMINGPROFILE

USERNAME

USERPROFILE

WINDIR

SERVER\_NAME

GATEWAY\_INTERFACE

SERVER\_PORT

REMOTE\_HOST

CONTENT\_LENGTH

SCRIPT\_NAME

SERVER\_PROTOCOL

SERVER\_SOFTWARE

REQUEST\_METHOD

PATH\_INFO

QUERY\_STRING

REMOTE\_ADDR

CONTENT\_TYPE

HTTP\_HOST

HTTP\_CONNECTION

HTTP\_CACHE\_CONTROL

HTTP\_USER\_AGENT

HTTP\_UPGRADE\_INSECURE\_REQUESTS

HTTP\_ACCEPT

HTTP\_ACCEPT\_ENCODING

HTTP\_ACCEPT\_LANGUAGE

wsgi.input

wsgi.errors

wsgi.version

wsgi.run\_once

wsgi.url\_scheme

wsgi.multithread

wsgi.multiprocess

wsgi.file\_wrapper

2.start\_response:按着http协议组装数据。

**def** application(environ, start\_response):  
 *# 按着http协议解析数据:environ  
 # 按者http协议组装数据:start\_response* print(environ)  
 print(type(environ))  
  
 *# 当前的请求路径* path=environ.get(**"PATH\_INFO"**)  
 start\_response(**'200 OK'**, [])  
  
 **if** path==**"/login"**:  
 **with** open(**"login.html"**,**"r"**) **as** f:  
  
 data=f.read()  
 **elif** path==**"/index"**:  
 **with** open(**"index.html"**,**"r"**) **as** f:  
 data=f.read()  
  
 **return** [data.encode(**"utf8"**)]

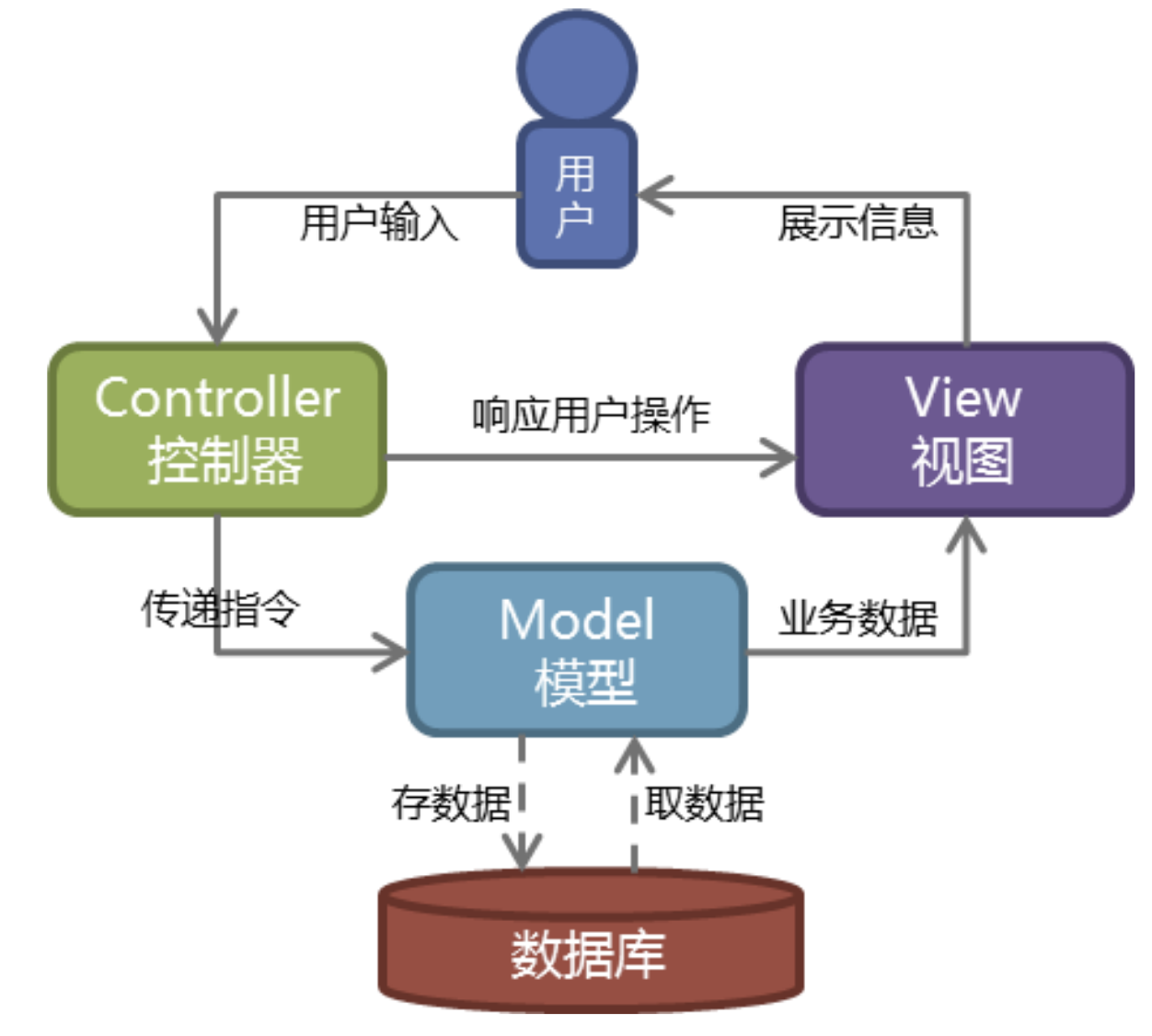
start\_response(**'200 OK'**, [(‘Content-Type’,’text/html’)])：两个参数，分别是消息响应状态代码、请求头，请求头以列表中嵌套元组形式。请求头可省略。

return [data.encode(‘utf8’)]为请求体。

## 三、Django

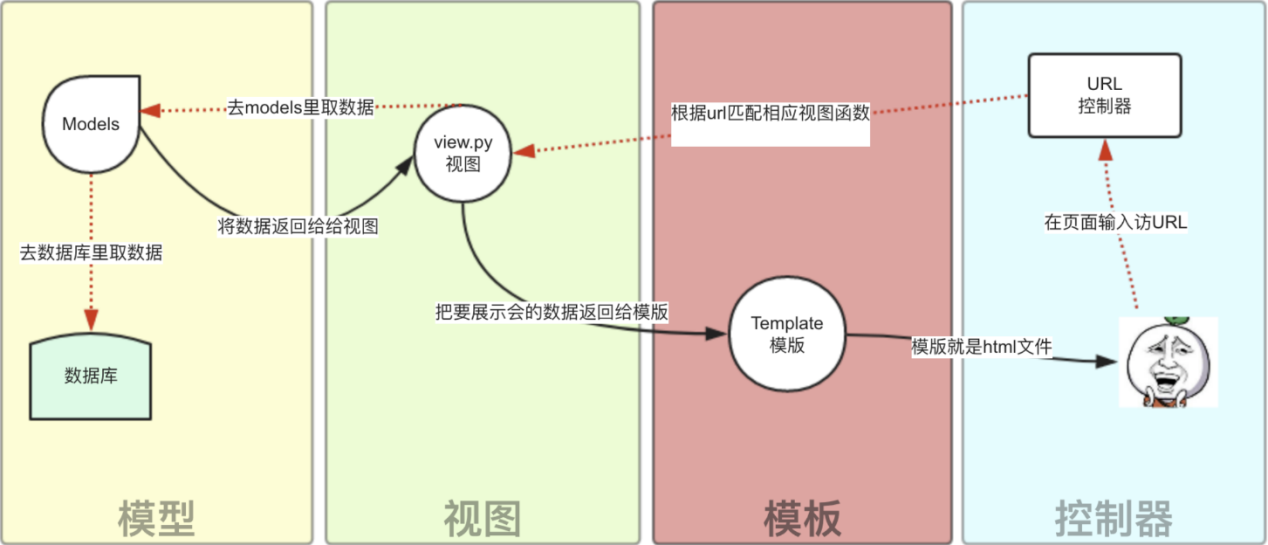
### （一）简介

3.1 MVC与MTV模型

MVC  
Web服务器开发领域里著名的MVC模式，所谓MVC就是把Web应用分为模型(M)，控制器(C)和视图(V)三层，他们之间以一种插件式的、松耦合的方式连接在一起，模型负责业务对象与数据库的映射(ORM)，视图负责与用户的交互(页面)，控制器接受用户的输入调用模型和视图完成用户的请求，其示意图如下所示：

MTV  
Django的MTV模式本质上和MVC是一样的，也是为了各组件间保持松耦合关系，只是定义上有些许不同，Django的MTV分别是值：

1. M 代表模型（Model）： 负责业务对象和数据库的关系映射(ORM)。
2. T 代表模板 (Template)：负责如何把页面展示给用户(html)。
3. V 代表视图（View）： 负责业务逻辑，并在适当时候调用Model和Template。  
   除了以上三层之外，还需要一个URL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，MTV的响应模式如下所示：



一般是用户通过浏览器向我们的服务器发起一个请求(request)，这个请求回去访问视图函数，（如果不涉及到数据调用，那么这个时候视图函数返回一个模板也就是一个网页给用户），视图函数调用模型，模型去数据库查找数据，然后逐级返回，视图函数把返回的数据填充到模板中空格中，最后返回网页给用户。

Django的四块：

1.URL控制器：分发URL地址给View

2.Template模板：负责把页面展示给用户。html

3.视图View：业务逻辑，接收到分发的URL后，根据请求调用Template和Models

4.Models模型：负责业务对象和数据库关系的映射。

### （二）Django的下载和基本命令

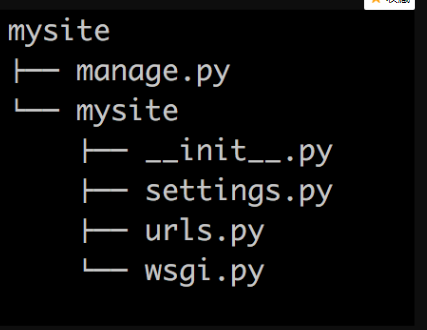
#### 1、下载Django：

pip3 install django

#### 2、创建一个django project

##### 1.通过命令行创建

django-admin.py startproject mysite

当前目录下会生成mysite的工程，目录结构如下：

manage.py ----- Django项目里面的工具，通过它可以调用django shell和数据库等，和

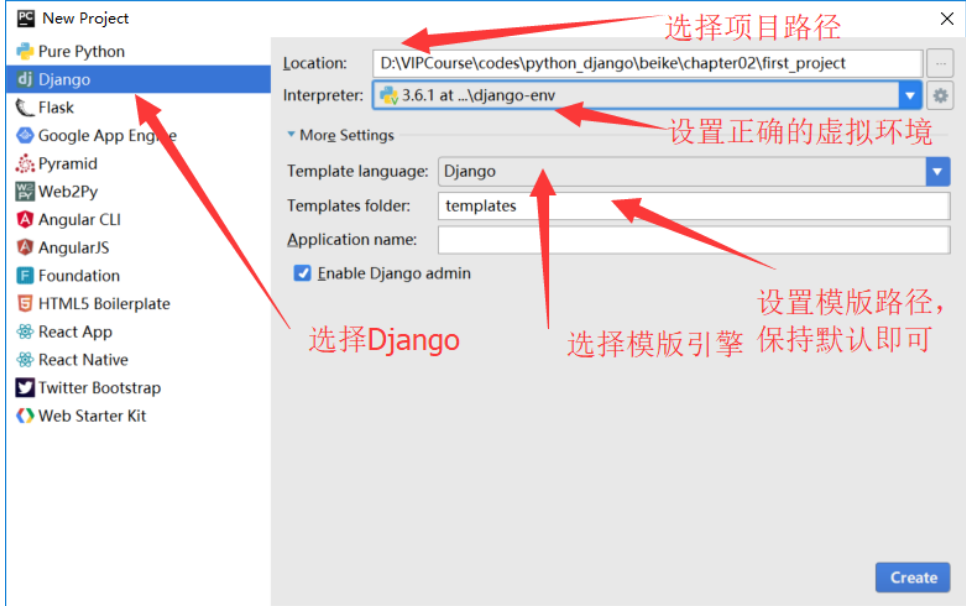
Django进行交互。

settings.py ---- 包含了项目的默认设置，包括数据库信息，调试标志以及其他一些工作变

量。  
urls.py ----- 负责把URL模式映射到应用程序。

##### 2.通过pycharm创建

用pycharm新建一个Django项目，新建项目的截图如下：



使用pycharm创建完项目后，还是需要重新进入到命令行单独创建app的。

#### 3、在mysite目录下创建应用

python manage.py startapp blog



#### 4.启动服务器

python manage.py runserver 8888

8888为端口。

### （三）Django简单实例

#### 1.地址分发URL

地址分发解析地址，并回调相应的函数，自动向该函数传入参数request，是地址解析内容对象，包含解析的所有内容。

#### 2.html文件（tenplates）：

{{date}}：是参数，需要从外部拿到的数据

#### 3.view文件

参数request是从地址分发模块中默认传入的，包含所有URL相关信息的对象。

##### render（）：

读取超文本内容，并返回，并可向超文本中传入参数。默认地址为template，故不用写html文件的地址。并将符合模板语法的变量进行替换。

模板语法：

1.{‘html中的变量’:py中的变量}:将py中的数据输入到html中。

2.{% url ‘地址的别名’ %} ：再urls中找到该地址及别名，用真正的地址代替其别名。

### （四）静态文件配置

静态文件如 jQuery、JavaScript、css等配置文件的存放地址，在根目录下建一个存放静态文件的文件夹，可以叫任何名字，如‘statics’。

在配置文件settings.py中配置静态文件地址。

注意存放静态文件的地址的变量名是固定的STATICFILES\_DIRS，是个列表，Django在调用静态地址时，即使用的这个名字。’/static/’是静态地址的别名，即在客户端访问时，就输入’ip:端口/static/文件名’。如果在此处将STATIC\_URL改变，则在客户端输入url时也要响应改变。

*# Static files (CSS, JavaScript, Images)  
# https://docs.djangoproject.com/en/2.0/howto/static-files/*STATIC\_URL = **'/static/'**

#对静态文件进行配置,先将静态文件的地址。该地址STATICFILES\_DIRS为固定的，不能改变。这样'/static/'通过Django转换，可以指代静态地址，是静态地址的别名。

STATICFILES\_DIRS=[os.path.join(BASE\_DIR,**'statics'**)]

在html中加载时，要输入静态文件的网络地址

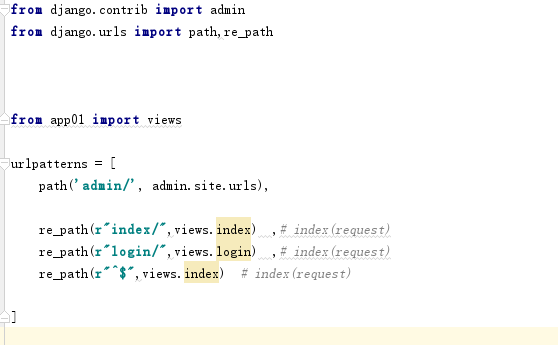


src=’/static/jquery-3.1.1.js’

包括外链的js、css、images文件。

### （五）路由控制urls.py

#### 1.简单配置include



path模块是Django 2.0的新模块。re\_path模块是1.0就有，2.0兼容，相当于正则匹配。

re\_path（）参数中的字符串和正则表达式的用法一样。

re\_path(r‘^sdfsf$’,回调函数)：以^开头，$结尾，确保唯一匹配。

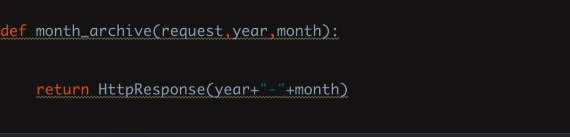


和正则表达式一样。

但是有分组匹配的时候（即有括号的时候），会自动将匹配到的字符串也当做参数传入回调函数，这样就多传入一个参数。一个分组多一个参数，多个分组多多个参数。

即views.year\_archive(request,year….) request为本身就需要带有的参数，后面参数与分组个数有关。

在view.py中创建函数



HttpResponse()为模块，里面参数为请求体。会被浏览器解析。请求行、请求头会在Django自动写。

#### 2.有名分组

(P<name>规则)：name为该分组的名字。

即正则表达中的有名分组。给分组起名字。



尖括号中为各分组的名字。

则在view.py中响应的回调函数，其传入参数也必须与组名相同，但是可以不区分顺序。相当于关键字传参。

#### 3.路由分发

路由分发按照总-分的顺序。

总项目有个urls.py，每个应用中自己建一个urls.py（将总urls复制过去）。

总urls.py将 请求分发至各app，各app自己内部再完成分发。

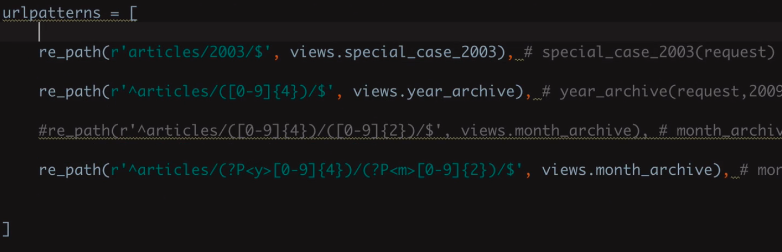
总urls：要导入include模块。

from django.urls import include



将含有app01的url路径分发至app01.urls。

然后在app01的urls中



对剩余的url进行匹配。

总的url地址为app01/articles/2008

app01/由总url来分配，剩下的由app的url来分配。

#### 4别名、反向解析

request.method：返回本次请求类型，返回字符串，’GET’或者’POST’

resquest.GET:返回一个字典，包含本次所有get请求的数据。

request.POST: 返回一个字典，包含本次所有post请求的数据。



返回的post请求的数据，为自定形式，内容为列表。

**反向解析是通过url别名来反向找出url。**

path（re\_path）和render两个结合实现的。

目的是对于一个界面，如果将其url改变时，所有从其他如表单action也跟着改变。

##### 方法一：在模板中进行反向解析

###### 步骤一

path（'login/’,views.login,name=’LOG’）：为’login/’这个url起一个别名’LOG’，不管login怎么变，’LOG’都指向他。

###### 步骤二

在需要用到该url的地方，一律由’LOG’来代替，该大括号为母版语法。

如果区分名称空间，则别名前需加上名称空间。’app01:Log’

###### 步骤三

view模块，写函数。

def login(request):  
  
  
 print(request.method)  
  
  **if** request.method=="GET":  
 return render(request,"login.html")#这个地方用render方法，渲染时，在碰到模板语法大括号时，会自动将login的名字替换’LOG’。

**else**:  
 print(request.GET)  
 print(request.POST)  
 user=request.POST.get(**"user"**)  
 pwd=request.POST.get(**"pwd"**)  
  
 **if** user==**"yuan" and** pwd==**"123"**:  
 **return** HttpResponse(**"登录成功!"**)  
 **else**:  
 **return** HttpResponse(**"用户名或者密码错误!"**)

这个地方用render方法，渲染时，在碰到模板语法大括号时，会自动将login的名字替换’LOG’。render其实内置了reverse方法。所以如果是不同名称空间下的相同别名时，在html的别名中要加入名称空间。

##### 方法二：在python脚本中进行反向解析

reverse的作用：在视图函数中使用reverse，就是用来通过别名找原文url。

url=reverse（‘LOG’）:即默认LOG为别名，即re\_path()中name=‘LOG’，首先会在全局中找LOG别名，如果没有，会在自己的项目中找LOG别名，然后解析出原文url。

###### 反向解析

在使用Django 项目时，一个常见的需求是获得URL 的最终形式，以用于嵌入到生成的内容中（视图中和显示给用户的URL等）或者用于处理服务器端的导航（重定向等）。人们强烈希望不要硬编码这些URL（费力、不可扩展且容易产生错误）或者设计一种与URLconf 毫不相关的专门的URL 生成机制，因为这样容易导致一定程度上产生过期的URL。

在需要URL 的地方，对于不同层级，Django 提供不同的工具用于URL 反查：

* 在模板中：使用url 模板标签。
* 在Python 代码中：使用from django.urls import reverse()函数

urls.py:

from django.conf.urls import url

from . import views

urlpatterns = [

#...

re\_path(r'^articles/([0-9]{4})/$',views.year\_archive,name='news-year-archive'),

#...

]

在模板中：

<a href="{% url 'news-year-archive' 2012 %}">2012 Archive</a>

<ul>

{% for yearvar in year\_list %}

<li><a href="{% url 'news-year-archive' yearvar %}">{{ yearvar }} Archive</a></li>

{% endfor %}

</ul>

在python中：

view.py

from django.urls import reverse

from django.http import HttpResponseRedirect

def redirect\_to\_year(request):

# ...

year = 2006

# ...

return HttpResponseRedirect(reverse('news-year-archive', args=(year,))) # 同redirect("/path/")

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。

对于含有正则表达式的别名，需要匹配该正则的参数。

导入函数

from django.urls import reverse

在view.py中获取url

url=reverse(“s\_c\_2003”)

然后重定向。

“s\_c\_2003”为url的别名，通过reverse，来找到真正的url，并返回给url变量。

##### 正则表达式的反向解析



此url中含有一个正则表达式分组。注意必须是分组

则在解析时，要填入一个相应的参数，位数和分组要求的位数一致，大小随意。



即给一个默认值，如果找不到相应的数字时，就用4009来代替正则。如果找到了，就填入真实的数字。

如果在模板层设置别名时指定了名称空间，则在反向解析时也要指定名称空间。

#### 5.名称空间

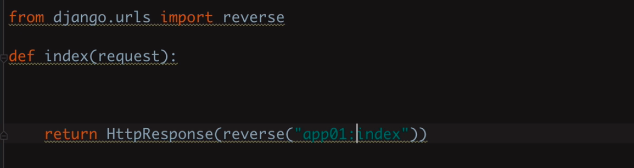
对于不同的url可能会起到相同的别名，则会被覆盖掉，只保留一个url。

解决方法：在分配地址url时指定名称空间，如在哪个app的范围内，那么只会在这个app范围内找，不会覆盖。

re\_path（r'app01/’,include((“app01.urls”,”app01”))）

此处为一级分发，即总的地址分发，分发至各app。include中是个元组，前面指向二级地址分发，后面指定名称空间。

使用时：



reverse(“app01:别名”)

#### 6.path新功能

比re\_path多两个功能

思考情况如下：

urlpatterns = [

re\_path('articles/(?P<year>[0-9]{4})/', year\_archive),

re\_path('article/(?P<article\_id>[a-zA-Z0-9]+)/detail/', detail\_view),

re\_path('articles/(?P<article\_id>[a-zA-Z0-9]+)/edit/', edit\_view),

re\_path('articles/(?P<article\_id>[a-zA-Z0-9]+)/delete/', delete\_view), ]

考虑下这样的两个问题：

第一个问题，函数 year\_archive 中year参数是字符串类型的，因此需要先转化为整数类型的变量值，当然year=int(year) 不会有诸如如TypeError或者ValueError的异常。那么有没有一种方法，在url中，使得这一转化步骤可以由Django自动完成？

第二个问题，三个路由中article\_id都是同样的正则表达式，但是你需要写三遍，当之后article\_id规则改变后，需要同时修改三处代码，那么有没有一种方法，只需修改一处即可？

在Django2.0中，可以使用 path 解决以上的两个问题。

##### 1.基本示例

这是一个简单的例子：

from django.urls import path

from . import views

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>/', views.article\_detail),

]

基本规则：

* 使用尖括号(<>)从url中捕获值。
* 捕获值中可以包含一个转化器类型（converter type），比如使用 <int:name> 捕获一个整数变量。若果没有转化器，将匹配任何字符串，当然也包括了 / 字符。
* 无需添加前导斜杠。

以下是根据 [2.0官方文档](https://docs.djangoproject.com/en/2.0/topics/http/urls/#example) 而整理的示例分析表：



##### 2.path转化器

文档原文是Path converters，暂且翻译为转化器。

Django默认支持以下5个转化器：

str,匹配除了路径分隔符（/）之外的非空字符串，这是默认的形式

int,匹配正整数，包含0。

slug,匹配字母、数字以及横杠、下划线组成的字符串。

uuid,匹配格式化的uuid，如 075194d3-6885-417e-a8a8-6c931e272f00。

path,匹配任何非空字符串，包含了路径分隔符

##### 3.自定义转化器

对于一些复杂或者复用的需要，可以定义自己的转化器。转化器是一个类或接口，它的要求有三点：

regex 类属性，字符串类型

to\_python(self, value) 方法，value是由类属性 regex 所匹配到的字符串，返回具体的Python变量值，以供Django传递到对应的视图函数中。

to\_url(self, value) 方法，和 to\_python 相反，value是一个具体的Python变量值，返回其字符串，通常用于url反向引用。

例子：

class FourDigitYearConverter: #定义一个转换类

regex = '[0-9]{4}'

def to\_python(self, value):

return int(value)

def to\_url(self, value):

return '%04d' % value

使用register\_converter 将其注册到URL配置中：

from django.urls import register\_converter, path

from . import converters, views

#注册自定义转换器，并为其起名字。

register\_converter(converters.FourDigitYearConverter, 'yyyy') #‘yyyy’

为转换器的名字

urlpatterns = [

path('articles/2003/', views.special\_case\_2003),

path('articles/<yyyy:year>/', views.year\_archive),

...

]

### （六）视图层view

view.py负责业务逻辑。

#### 1.request参数

##### （1）request.method：

返回本次浏览器请求类型，返回值为字符串，‘GET’‘POST’

地址栏发请求默认都是GET请求。

##### （2）request.POST、request.GET

存储了该请求的内容，以字典性形式存储。key为数据名，内容用列表存放。

url发送get请求，如果有请求内容时，是以?标注，以&来分割数据。

127.0.0.1:8800/login/?name=hehe&pwd=123

问号后面为发送请求的数据，&为不同数据分开。

则request.GET里面存储的是：

{‘name’:[’hehe’],’pwd’:[‘123’]}

对于select、checkbox标签返回的多选值，用request.POST.getlist(‘键值’)一次性以列表形式取出。

##### （3）request.path

里面存储了本次请求的路径。

路径就是url中端口号之后，？之前的内容。

如果在端口号之后什么都没有，也可以匹配。

re\_path(r’^$’,view.index)

则request.path中存放的只有一个 \，即根目录。

##### 前端使用request

对于request中的内容，前端可以直接通过模板语法拿来用，而不用后端进行特定传值。



##### 3.Request.body

　　一个字符串，代表请求报文的主体。在处理非 HTTP 形式的报文时非常有用，例如：二进制图片、XML,Json等。

　　但是，如果要处理表单数据的时候，推荐还是使用 Request.POST 。

##### 4.Request.path

　　一个字符串，表示请求的路径组件（不含域名）。

　　例如："/music/bands/the\_beatles/"

##### 6.Request.encoding

　　一个字符串，表示提交的数据的编码方式（如果为 None 则表示使用 DEFAULT\_CHARSET 的设置，默认为 'utf-8'）。

这个属性是可写的，你可以修改它来修改访问表单数据使用的编码。

接下来对属性的任何访问（例如从 GET 或 POST 中读取数据）将使用新的 encoding 值。

如果你知道表单数据的编码不是 DEFAULT\_CHARSET ，则使用它。

##### 7.Request.META

　　一个标准的Python 字典，包含所有的HTTP 首部。具体的头部信息取决于客户端和服务器，下面是一些示例：

CONTENT\_LENGTH —— 请求的正文的长度（是一个字符串）。

CONTENT\_TYPE —— 请求的正文的MIME 类型。

HTTP\_ACCEPT —— 响应可接收的Content-Type。

HTTP\_ACCEPT\_ENCODING —— 响应可接收的编码。

HTTP\_ACCEPT\_LANGUAGE —— 响应可接收的语言。

HTTP\_HOST —— 客服端发送的HTTP Host 头部。

HTTP\_REFERER —— Referring 页面。

HTTP\_USER\_AGENT —— 客户端的user-agent 字符串。

QUERY\_STRING —— 单个字符串形式的查询字符串（未解析过的形式）。

REMOTE\_ADDR —— 客户端的IP 地址。

REMOTE\_HOST —— 客户端的主机名。

REMOTE\_USER —— 服务器认证后的用户。

REQUEST\_METHOD —— 一个字符串，例如"GET" 或"POST"。

SERVER\_NAME —— 服务器的主机名。

SERVER\_PORT —— 服务器的端口（是一个字符串）。

　　从上面可以看到，除 CONTENT\_LENGTH 和 CONTENT\_TYPE 之外，请求中的任何 HTTP 首部转换为 META 的键时，

都会将所有字母大写并将连接符替换为下划线最后加上 HTTP\_ 前缀。

所以，一个叫做 X-Bender 的头部将转换成 META 中的 HTTP\_X\_BENDER 键。

##### 8.Request.FILES

　　一个类似于字典的对象，包含所有的上传文件信息。

FILES 中的每个键为<input type="file" name="" /> 中的name，值则为对应的数据。

　　注意，FILES 只有在请求的方法为POST 且提交的<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。

 返回一个包含了所有的上传文件的  querydict  对象。通过表单所上传的所有  文件  都会保存在该属性中。

　　                                             key的值是input标签中name属性的值，value的值是一个UploadedFile对象

##### 9.Request.COOKIES

　　一个标准的Python 字典，包含所有的cookie。键和值都为字符串。

##### 10.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。

完整的细节参见会话的文档。

##### 11.Request.user(用户认证组件下使用)

　　一个 AUTH\_USER\_MODEL 类型的对象，表示当前登录的用户。

　　如果用户当前没有登录，user 将设置为 django.contrib.auth.models.AnonymousUser 的一个实例。你可以通过 is\_authenticated() 区分它们。

例如：

if request.user.is\_authenticated():

# Do something for logged-in users.

else:

# Do something for anonymous users.

user 只有当Django 启用 AuthenticationMiddleware 中间件时才可用。

##### （4）常用方法

###### 1.Request.get\_full\_path()

　　返回 path，如果可以将加上查询字符串。

　　例如："/music/bands/the\_beatles/?print=true"

###### 2.Request.is\_ajax()

　　如果请求是通过XMLHttpRequest 发起的，则返回True，方法是检查 HTTP\_X\_REQUESTED\_WITH 相应的首部是否是字符串'XMLHttpRequest'。

　　大部分现代的 JavaScript 库都会发送这个头部。如果你编写自己的 XMLHttpRequest 调用（在浏览器端），你必须手工设置这个值来让 is\_ajax() 可以工作。

　　如果一个响应需要根据请求是否是通过AJAX 发起的，并且你正在使用某种形式的缓存例如Django 的 cache middleware，

你应该使用 vary\_on\_headers('HTTP\_X\_REQUESTED\_WITH') 装饰你的视图以让响应能够正确地缓存。

#### 2.视图响应对象

##### （1）Httpresponse

return Httpresponse(‘字符串’)

返回给浏览器，并在浏览器中渲染显示。

##### （2）render

render最终返回的也是Httpresponse，只是对httpresponse进行了封装，丰富了功能。

return render(response,’html文件名’ [,超文本文件中所需参数])

render（）方法封装了httpresponse等

###### render向html传入参数

render（request,'超文本文件’[,超文本文件中所需参数]）

html中加入参数，就不叫html文件，而是叫做模板文件。

如:<p>{{ time }}</p>

则在render中可以通过字典传入该参数。

render（request,’time.html’,{‘time’:python中的数据}）

这样就可以传入参数。

##### (3)redirect

重定向，向服务器发送请求，请求该页面。

在视图函数中return：

return redirect(‘/index’)

相当于由浏览器发送index的url请求。

##### （4）Httpresponseredirect()

### （七）模板层Templates

#### 1.向模板文件传入参数

**步骤一：模板文件需传入参数格式**

html中加入参数，就不叫html文件，而是叫做模板文件。

如:<p>{{ time }}</p>

格式：{{ 变量名 }}

**步骤二：view视图中的操作**

render（request,'超文本文件’[,超文本文件中所需参数]）

则在render中可以通过字典传入该参数。

render（request,’time.html’,{‘time’:python中的数据}）

这样就可以传入参数。

如果变量过多，也可以使用locals（）代替该字典，但是变量名需要和字典的key一致。

即key为’time’，模板中变量名也必须为time

#### 模板位置

模板文件夹templates可以建在根目录下，也可以建在App中，django在找模板时，会先在根目录中找，然后按照setting中App的注册顺序来找。

#### 2.模板语法

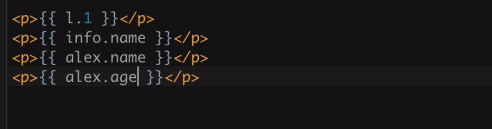
模板语法:

##### 1.变量: {{}}

1 深度查询 句点符  
 2 过滤器 {{val|filter\_name:参数}}

###### 深度查询

即查询列表或者字典中某一项，在html中通过句点来查询。



如上图中，L为一个列表，info为一个字典，alex为一个实例对象。

###### 过滤器

**语法：**

1

{{obj|filter\_\_name:param}}

###### 常用过滤器：

default 设置默认值

如果一个变量是false或者为空，使用给定的默认值。否则，使用变量的值。例如：

|  |  |
| --- | --- |
| 1 | {{ value|default:"nothing" }} |

length 返回长度

返回值的长度。它对字符串和列表都起作用。例如：

{ value|length }}

如果 value 是 ['a', 'b', 'c', 'd']，那么输出是 4。

filesizeformat 将数值转换为文件尺寸

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）。例如：

|  |  |
| --- | --- |
| 1 | {{ value|filesizeformat }} |

如果 value 是 123456789，输出将会是 117.7 MB。

date 日期格式

如果 value=datetime.datetime.now()

{{ value|date:"Y-m-d" }}

slice 切片

如果 value="hello world"

{{ value|slice:"2:-1" }}

truncatechars 按长度截断

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：**要截断的字符数

例如：

{{ value|truncatechars:9 }}

safe

Django的模板中会对HTML标签和JS等语法标签进行自动转义，原因显而易见，这样是为了安全。但是有的时候我们可能不希望这些HTML元素被转义，比如我们做一个内容管理系统，后台添加的文章中是经过修饰的，这些修饰可能是通过一个类似于FCKeditor编辑加注了HTML修饰符的文本，如果自动转义的话显示的就是保护HTML标签的源文件。为了在Django中关闭HTML的自动转义有两种方式，如果是一个单独的变量我们可以通过过滤器“|safe”的方式告诉Django这段代码是安全的不必转义。比如：

value="<a href="">点击</a>"

|  |
| --- |
| 1 |

|  |  |
| --- | --- |
| 1 | {{ value|safe}}如果不加safe，会自动将value中的<等符号进行转义成 &gt; 等，value会被渲染成  纯文本，如果加了safe，则会将标签之间传入，会被渲染成a标签。 |

这里简单介绍一些常用的模板的过滤器，[http://python.usyiyi.cn/translate/django\_182/ref/templates/builtins.html#ref-templates-builtins-tags](http://python.usyiyi.cn/translate/django_182/ref/templates/builtins.html%23ref-templates-builtins-tags)

|  |  |  |
| --- | --- | --- |
| 过滤器 | 描述 | 示例 |

upper 以大写方式输出 {{ user.name | upper }}

|  |  |  |
| --- | --- | --- |
| add | 给value加上一个数值 | {{ user.age | add:”5” }} |

addslashes 单引号加上转义号

capfirst 第一个字母大写 {{ ‘good’| capfirst }} 返回”Good”

center 输出指定长度的字符串，把变量居中 {{ “abcd”| center:”50” }}

cut 删除指定字符串 {{ “You are not a Englishman” | cut:”not” }}

|  |  |  |
| --- | --- | --- |
| date | 格式化日期 |  |
| default | 如果值不存在，则使用默认值代替 | {{ value | default:”(N/A)” }} |

default\_if\_none 如果值为None, 则使用默认值代替

dictsort 按某字段排序，变量必须是一个dictionary {% for moment in moments | dictsort:”id” %}

|  |  |  |
| --- | --- | --- |
| dictsortreversed | 按某字段倒序排序，变量必须是dictionary |  |
| divisibleby | 判断是否可以被数字整除 | {{ 224 | divisibleby:2 }} 返回 True |
| escape | 按HTML转义，比如将”<”转换为”&lt” |  |
| filesizeformat | 增加数字的可读性，转换结果为13KB,89MB,3Bytes等 | {{ 1024 | filesizeformat }} 返回 1.0KB |
| first | 返回列表的第1个元素，变量必须是一个列表 |  |

floatformat 转换为指定精度的小数，默认保留1位小数 {{ 3.1415926 | floatformat:3 }} 返回 3.142 四舍五入

get\_digit 从个位数开始截取指定位置的数字 {{ 123456 | get\_digit:’1’}}

join 用指定分隔符连接列表 {{ [‘abc’,’45’] | join:’\*’ }} 返回 abc\*45

length 返回列表中元素的个数或字符串长度

length\_is 检查列表，字符串长度是否符合指定的值 {{ ‘hello’| length\_is:’3’ }}

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| linebreaks | | | | 用<p>或<br>标签包裹变量 | {{ “Hi\n\nDavid”|linebreaks }} 返回<p>Hi</p><p>David</p> |
| linebreaksbr | | | | 用<br/>标签代替换行符 |  |
| linenumbers | | | | 为变量中的每一行加上行号 |  |
| ljust | | | | 输出指定长度的字符串，变量左对齐 | {{‘ab’|ljust:5}}返回 ‘ab   ’ |
| lower | | | | 字符串变小写 |  |
| make\_list | | | | 将字符串转换为列表 |  |
| pluralize | | | | 根据数字确定是否输出英文复数符号 |  |
|  |  |  |

random 返回列表的随机一项

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| removetags | | | | 删除字符串中指定的HTML标记 | {{value | removetags: “h1 h2”}} |
|  | |  |  |

rjust 输出指定长度的字符串，变量右对齐

slugify 在字符串中留下减号和下划线，其它符号删除，空格用减号替换 {{ '5-2=3and5 2=3' | slugify }} 返回 5-23and5-23

stringformat 字符串格式化，语法同python

time 返回日期的时间部分

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| slice | | | | 切片操作， 返回列表 | {{[3,9,1] | slice:’:2’}} 返回 [3,9]  {{ 'asdikfjhihgie' | slice:':5' }} 返回 ‘asdik’ |
| timesince | | | | 以“到现在为止过了多长时间”显示时间变量 | 结果可能为 45days, 3 hours |
|  | |  |  |

timeuntil 以“从现在开始到时间变量”还有多长时间显示时间变量

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| title | | | | 每个单词首字母大写 |  |
| truncatewords | | | | 将字符串转换为省略表达方式 | {{ 'This is a pen' | truncatewords:2 }}返回  This is ... |
| truncatewords\_html | | | | 同上，但保留其中的HTML标签 | {{ '<p>This is a pen</p>' | truncatewords:2 }}返回  <p>This is ...</p> |
| urlencode | | | | 将字符串中的特殊字符转换为url兼容表达方式 | {{ ‘http://www.aaa.com/foo?a=b&b=c’ | urlencode}} |
| urlize | | | | 将变量字符串中的url由纯文本变为链接 |  |
| wordcount | | | | 返回变量字符串中的单词数 |  |
|  |  |  |

yesno 将布尔变量转换为字符串yes, no 或maybe

{{ True | yesno }} 返回 yes

{{ False | yesno }} no

{{ None | yesno }} maybe

##### 2.标签: {% %}

###### for标签

语法格式：

{% for 变量名 in 范围%}

{% endfor %}

如：

{% for i in l %}

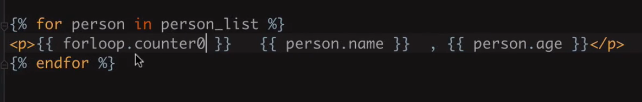
<p>{{ i }}</p>

{% endfor %}

l为一个列表，此处为遍历列表，并把列表中的每个值对应一个p标签。

{{forloop.counter}} 记录循环的次数，并显示。从1开始。每循环一次，加

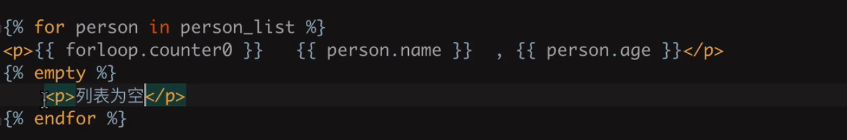
一。必须放在循环体内。



注意：forloop.counter0：计数从0开始。

forloop.first forloop.last 第一次循环、最后一次循环

empty



循环为空时，会运行。

###### if标签

{% if num > 100 or num < 0 %}

<p>无效</p>

{% elif num > 80 and num < 100 %}

<p>优秀</p>

{% else %}

<p>凑活吧</p>

{% endif %}

###### with标签

使用一个简单地名字缓存一个复杂的变量，当你需要使用一个“昂贵的”方法（比如访问数据库）很多次的时候是非常有用的  
例如：

{% with total=business.employees.count %}

{{ total }} employee{{ total|pluralize }}

{% endwith %}

类似于给变量起别名。

也可以是

{% with 变量名 as 别名 %}

…

….

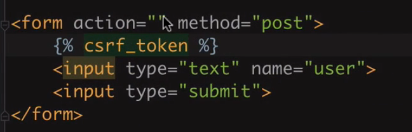
..

{%endwith%}

###### csrf\_token

这个标签用于跨站请求伪造保护

对于form表单等post请求，由于受到保护，所以不会显示。当使用csrf\_token标签时，在先发送get请求时，会在html加一个隐藏的input标签，获取一个码值，再发送post请求时，用于告诉服务端，这个post请求是安全的，使该页面显现。



post请求被隐藏的原因是因为防止浏览器一访问服务器就直接提交post的请求，如果第一次访问服务器上来就发送post请求，就不会获得该码值，该请求就会被拦截，防止页面攻击。

##### 3.自定义过滤器和标签

1、在settings中的INSTALLED\_APPS配置当前app，不然django无法找到自定义的simple\_tag.  
2、在app中创建templatetags模块(模块名（文件夹名）只能是templatetags，注意建立的是一个模块，而不是文件夹。)  
3、创建任意 .py 文件，如：my\_tags.py

from django import template

from django.utils.safestring import mark\_safe

register = template.Library()   #register的名字是固定的,不可改变

**以上为建立my\_tags文件所必须写的内容。**

@register.filter 加上此装饰器，变为过滤器

def filter\_multi(v1,v2):

    return  v1 \* v2

@register.simple\_tag 加上此装饰器，变为标签

def simple\_tag\_multi(v1,v2):

    return  v1 \* v2

@register.simple\_tag

def my\_input(id,arg):

    result = "<input type='text' id='%s' class='%s' />" %(id,arg,)

    return mark\_safe(result)

###### （3）inclusion\_tag将模板语法和数据结合起来的装饰器

注意，这个函数也需要在templatetags模块中，前面的代码和上面必须的是一样的

@register.inclusion\_tag(**"base/panel.html"**)

def myfunc（personsite）

return {‘dict’:dict}

将需要数据的html放入该装饰器中，

装饰器下面为自定义的函数，该函数需要有返回值。

该返回值会自动传入装饰器的html中。里面为该html所需使用的数据。

这个html文档就是一个封装好数据的文档模块，可供其他html加载。包括供继承的base标签，也可以加载该模块。

在使用时，在另外一个需要该模块的html中的需要该模块的位置输入：

{% **load my\_tags** %} my\_tag为该函数所在的文件名  
{% **myfunc personsite** %} myfunc为该函数名。

其中该装饰器装饰的函数，personsite为该函数需要的数据，这个数据需要视图函数传入(如request)。

4、在使用自定义simple\_tag和filter的html文件中导入之前创建的 my\_tags.py

{% load 自定义过滤器所在的文件名（不加扩展名） %}

{% **load my\_tag\_filter** %} #导入该文件  
<**p**>{{ **i**|**multi\_fliter**:20 }}</**p**>  
  
*{#<p>{% multi\_tag 7 9 6 %}</p>#}*{% **if i**|**multi\_fliter**:10 > 100 %}  
 <**p**>100</**p**>  
{% **else** %}  
 <**p**>{{ **i** }}</**p**>  
{% **endif** %}

5、使用simple\_tag和filter（如何调用）

-------------------------------.html

{% load filter\_multi %}

# num=12

{{ num|filter\_multi:2 }} #24

{{ num|filter\_multi:"[22,333,4444]" }}

{% simple\_tag\_multi 2 5 %} 参数不限,用空格分隔，但不能放在if for语句中

{% simple\_tag\_multi num 5 %}

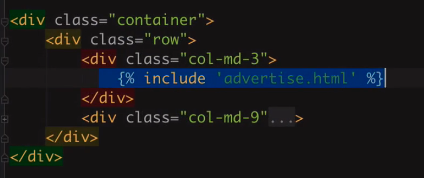
注意：filter可以用在if等语句后，simple\_tag不可以用在if等判断条件中。

在if标签中嵌套过滤器。

|  |  |
| --- | --- |
| 1  2  3 | {% if num|filter\_multi:30 > 100 %}      {{ num|filter\_multi:30 }}  {% endif %} |

##### 4.模板的继承标签

###### 1.include



include用于引入其他的html片段文件。

可以把公用的html一部分代码写入一个文件，然后其他文件用include进行引用。类似于函数。但是会把所有数据一模一样。

###### 2.extend与block

建立一个模板文件，并预留扩展接口。

模板文件：

建立一个模板文件，并在不确定内容的地方预留

{% block 接口名 %}

模板文件默认内容（可省略）

{% endblock %}

引入模板文件（假如模板文件叫 base.html）：

在文件开头引入

{% extend ‘base.html’ %}

然后对接口进行补充。

会把原接口内的内容覆盖掉。

{% block 接口名 %}

加入html标签内容

{% endblock %}

如果不想覆盖，保留原内容

{% block 接口名 %}

{{ block.super }}

加入html标签内容

{% endblock %}

也可以给endblock加入接口名，提高可读性。

### （八）模型层 数据库的操作

#### 1.ORM简介

对象-关系-映射 的简称。

ORM只能操作到table一级，无法新建或删除数据库。



<https://www.cnblogs.com/yuanchenqi/articles/8933283.html>

#### 2.建新表

创建名为book的app，在book下的models.py中创建模型：

from django.db import models

# Create your models here.

class Book(models.Model):

id=models.AutoField(primary\_key=True) #设置自增、主键

title=models.CharField(max\_length=32,verbose\_neme=’文章标题’) #字符串 指定字段名称。

state=models.BooleanField() #布尔值

pub\_date=models.DateField() #日期类型

price=models.DecimalField(max\_digits=8,decimal\_places=2)#浮点型，共8位，小数点后保留2位。

publish=models.CharField(max\_length=32)

创建数据表

python manage.py makemigrations

python manage.py migrate

不管主键字段叫什么，主键都可以用 对象名.pk 来选取。

在此处，book.id=book.pk

##### 更多字段：

###### <1> CharField

        字符串字段, 用于较短的字符串.

        CharField 要求必须有一个参数 maxlength, 用于从数据库层和Django校验层限制该字段所允许的最大字符数.

###### <2> IntegerField

       #用于保存一个整数.

###### <3> FloatField

        一个浮点数. 必须 提供两个参数:

        参数    描述

        max\_digits    总位数(不包括小数点和符号)

        decimal\_places    小数位数

                举例来说, 要保存最大值为 999 (小数点后保存2位),你要这样定义字段:

                models.FloatField(..., max\_digits=5, decimal\_places=2)

                要保存最大值一百万(小数点后保存10位)的话,你要这样定义:

                models.FloatField(..., max\_digits=19, decimal\_places=10)

                admin 用一个文本框(<input type="text">)表示该字段保存的数据.

###### <4> AutoField

        一个 IntegerField, 添加记录时它会自动增长. 你通常不需要直接使用这个字段;

        自定义一个主键：my\_id=models.AutoField(primary\_key=True)

        如果你不指定主键的话,系统会自动添加一个主键字段到你的 model.

###### <5> BooleanField

        A true/false field. admin 用 checkbox 来表示此类字段.

###### <6> TextField

        一个容量很大的文本字段.

        admin 用一个 <textarea> (文本区域)表示该字段数据.(一个多行编辑框).

###### <7> EmailField

        一个带有检查Email合法性的 CharField,不接受 maxlength 参数.

###### <8> DateField

        一个日期字段. 共有下列额外的可选参数:

        Argument    描述

        auto\_now    当对象被保存时,自动将该字段的值设置为当前时间.通常用于表示 "last-modified" 时间戳.

        auto\_now\_add    当对象首次被创建时,自动将该字段的值设置为当前时间.通常用于表示对象创建时间.

        （仅仅在admin中有意义...)

###### <9> DateTimeField

         一个日期时间字段. 类似 DateField 支持同样的附加选项.

###### <10> ImageField

        类似 FileField, 不过要校验上传对象是否是一个合法图片.#它有两个可选参数:height\_field和width\_field,

        如果提供这两个参数,则图片将按提供的高度和宽度规格保存.

###### <11> FileField

     一个文件上传字段.

     要求一个必须有的参数: upload\_to, 一个用于保存上载文件的本地文件系统路径. 这个路径必须包含 strftime #formatting,

     该格式将被上载文件的 date/time

     替换(so that uploaded files don't fill up the given directory).

     admin 用一个<input type="file">部件表示该字段保存的数据(一个文件上传部件) .

     注意：在一个 model 中使用 FileField 或 ImageField 需要以下步骤:

            （1）在你的 settings 文件中, 定义一个完整路径给 MEDIA\_ROOT 以便让 Django在此处保存上传文件.

            (出于性能考虑,这些文件并不保存到数据库.) 定义MEDIA\_URL 作为该目录的公共 URL. 要确保该目录对

             WEB服务器用户帐号是可写的.

            （2） 在你的 model 中添加 FileField 或 ImageField, 并确保定义了 upload\_to 选项,以告诉 Django

             使用 MEDIA\_ROOT 的哪个子目录保存上传文件.你的数据库中要保存的只是文件的路径(相对于 MEDIA\_ROOT).

             出于习惯你一定很想使用 Django 提供的 get\_<#fieldname>\_url 函数.举例来说,如果你的 ImageField

             叫作 mug\_shot, 你就可以在模板中以 {{ object.#get\_mug\_shot\_url }} 这样的方式得到图像的绝对路径.

###### <12> URLField

      用于保存 URL. 若 verify\_exists 参数为 True (默认), 给定的 URL 会预先检查是否存在( 即URL是否被有效装入且

      没有返回404响应).

      admin 用一个 <input type="text"> 文本框表示该字段保存的数据(一个单行编辑框)

###### <13> NullBooleanField

       类似 BooleanField, 不过允许 NULL 作为其中一个选项. 推荐使用这个字段而不要用 BooleanField 加 null=True 选项

       admin 用一个选择框 <select> (三个可选择的值: "Unknown", "Yes" 和 "No" ) 来表示这种字段数据.

###### <14> SlugField

       "Slug" 是一个报纸术语. slug 是某个东西的小小标记(短签), 只包含字母,数字,下划线和连字符.#它们通常用于URLs

       若你使用 Django 开发版本,你可以指定 maxlength. 若 maxlength 未指定, Django 会使用默认长度: 50.  #在

       以前的 Django 版本,没有任何办法改变50 这个长度.

       这暗示了 db\_index=True.

       它接受一个额外的参数: prepopulate\_from, which is a list of fields from which to auto-#populate

       the slug, via JavaScript,in the object's admin form: models.SlugField

       (prepopulate\_from=("pre\_name", "name"))prepopulate\_from 不接受 DateTimeFields.

###### <13> XMLField

        一个校验值是否为合法XML的 TextField,必须提供参数: schema\_path, 它是一个用来校验文本的 RelaxNG schema #的文件系统路径.

###### <14> FilePathField

        可选项目为某个特定目录下的文件名. 支持三个特殊的参数, 其中第一个是必须提供的.

        参数    描述

        path    必需参数. 一个目录的绝对文件系统路径. FilePathField 据此得到可选项目.

        Example: "/home/images".

        match    可选参数. 一个正则表达式, 作为一个字符串, FilePathField 将使用它过滤文件名.

        注意这个正则表达式只会应用到 base filename 而不是

        路径全名. Example: "foo.\*\.txt^", 将匹配文件 foo23.txt 却不匹配 bar.txt 或 foo23.gif.

        recursive可选参数.要么 True 要么 False. 默认值是 False. 是否包括 path 下面的全部子目录.

        这三个参数可以同时使用.

        match 仅应用于 base filename, 而不是路径全名. 那么,这个例子:

        FilePathField(path="/home/images", match="foo.\*", recursive=True)

        ...会匹配 /home/images/foo.gif 而不匹配 /home/images/foo/bar.gif

###### <15> IPAddressField

        一个字符串形式的 IP 地址, (i.e. "24.124.1.30").

###### <16> CommaSeparatedIntegerField

        用于存放逗号分隔的整数值. 类似 CharField, 必须要有maxlength参数.

##### 更多参数：

###### (1)null

如果为True，Django 将用NULL 来在数据库中存储空值。 默认值是 False.

###### (1)blank

如果为True，该字段允许不填。默认为False。

要注意，这与 null 不同。null纯粹是数据库范畴的，null是真的空值，在数据库中表现为null

而 blank 是数据验证范畴的。

如果一个字段的blank=True，表单的验证将允许该字段是空值。其实是存入了空字符串。

如果字段的blank=False，该字段就是必填的。

###### (2)default

字段的默认值。可以是一个值或者可调用对象。如果可调用 ，每有新对象被创建它都会被调用。

###### (3)primary\_key

如果为True，那么这个字段就是模型的主键。如果你没有指定任何一个字段的primary\_key=True，

Django 就会自动添加一个IntegerField字段做为主键，所以除非你想覆盖默认的主键行为，

否则没必要设置任何一个字段的primary\_key=True。

###### (4)unique

如果该值设置为 True, 这个数据字段的值在整张表中必须是唯一的

###### (5)choices

由二元组组成的一个可迭代对象（例如，列表或元组），用来给字段提供选择项。 如果设置了choices ，默认的表单将是一个选择框而不是标准的文本框，而且这个选择框的选项就是choices 中的选项。choices=[…,….,..]

###### （6）verbose\_name

指定字段名

#### django字段详解

**字段类型选择：**  
 AutoField(Field)

- int自增列，必须填入参数 primary\_key=True

BigAutoField(AutoField)

- bigint自增列，必须填入参数 primary\_key=True

注：当model中如果没有自增列，则自动会创建一个列名为id的列

from django.db import models

class UserInfo(models.Model):

# 自动创建一个列名为id的且为自增的整数列

username = models.CharField(max\_length=32)

class Group(models.Model):

# 自定义自增列

nid = models.AutoField(primary\_key=True)

name = models.CharField(max\_length=32)

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(PositiveIntegerRelDbTypeMixin, IntegerField)

- 正整数 0 ～ 2147483647

BigIntegerField(IntegerField):

- 长整型(有符号的) -9223372036854775808 ～ 9223372036854775807

自定义无符号整数字段

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字段默认的值为：

'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)',

BooleanField(Field)

- 布尔值类型

NullBooleanField(Field):

- 可以为空的布尔值

CharField(Field)

- 字符类型

- 必须提供max\_length参数， max\_length表示字符长度

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，开启刺功能，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）

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]]

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

DecimalField(Field)

- 10进制小数

- 参数：

max\_digits，小数总长度

decimal\_places，小数位长度

BinaryField(Field)

- 二进制类型  
  
字段中的参数详解：

null 数据库中字段是否可以为空（null=True）

db\_column 数据库中字段的列名(db\_column="test")

db\_tablespace

default 数据库中字段的默认值

primary\_key 数据库中字段是否为主键(primary\_key=True)

db\_index 数据库中字段是否可以建立索引(db\_index=True)

unique 数据库中字段是否可以建立唯一索引(unique=True)

unique\_for\_date 数据库中字段【日期】部分是否可以建立唯一索引

unique\_for\_month 数据库中字段【月】部分是否可以建立唯一索引

unique\_for\_year 数据库中字段【年】部分是否可以建立唯一索引

auto\_now 更新时自动更新当前时间  
 auto\_now\_add 创建时自动更新当前时间

verbose\_name Admin中显示的字段名称

blank Admin中是否允许用户输入为空 表单提交时可以为空

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)

error\_messages 自定义错误信息（字典类型），从而定制想要显示的错误信息；

字典健：null, blank, invalid, invalid\_choice, unique, and unique\_for\_date

如：{'null': "不能为空.", 'invalid': '格式错误'}

validators 自定义错误验证（列表类型），从而定制想要的验证规则

from django.core.validators import RegexValidator

from django.core.validators import EmailValidator,URLValidator,DecimalValidator,\

MaxLengthValidator,MinLengthValidator,MaxValueValidator,MinValueValidator

如：

test = models.CharField(

max\_length=32,

error\_messages={

'c1': '优先错信息1',

'c2': '优先错信息2',

'c3': '优先错信息3',

},

validators=[

RegexValidator(regex='root\_\d+', message='错误了', code='c1'),

RegexValidator(regex='root\_112233\d+', message='又错误了', code='c2'),

EmailValidator(message='又错误了', code='c3'), ]

)

#### 3.在setting中配置数据库

若想将模型转为mysql数据库中的表，需要在settings中配置：

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.mysql',

'NAME':'bms', 　　 　 # 要连接的数据库，连接前需要创建好

'USER':'root',　　　　　　 # 连接数据库的用户名

'PASSWORD':'',　　　　　　 # 连接数据库的密码

'HOST':'127.0.0.1', # 连接主机，默认本级

'PORT'：3306 　　　 # 端口 默认3306

}

}

注意1：NAME即数据库的名字，在mysql连接前该数据库必须已经创建，而上面的sqlite数据库下的db.sqlite3则是项目自动创建 USER和PASSWORD分别是数据库的用户名和密码。设置完后，再启动我们的Django项目前，我们需要激活我们的mysql。然后，启动项目，会报错：

no module named MySQLdb 。

这是因为django默认你导入的驱动是MySQLdb，可是MySQLdb 对于py3有很大问题，所以我们需要的驱动是PyMySQL 所以，我们只需要找到项目名文件下的\_\_init\_\_,在里面写入：

|  |  |
| --- | --- |
| 1  2 | import pymysql  pymysql.install\_as\_MySQLdb() |

最后通过两条数据库迁移命令即可在指定的数据库中创建表 ：

python manage.py makemigrations

python manage.py migrate

注意2:确保配置文件中的INSTALLED\_APPS中写入我们创建的app名称

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

"book"

]

注意3:如果报错如下：

|  |  |
| --- | --- |
| 1 | django.core.exceptions.ImproperlyConfigured: mysqlclient 1.3.3 or  newer is required; you have 0.7.11.None |

MySQLclient目前只支持到python3.4，因此如果使用的更高版本的python，需要修改如下：

通过查找路径C:\Programs\Python\Python36-32\Lib\site-packages\Django-2.0-py3.6.egg\django\db\backends\mysql  
这个路径里的文件把

|  |  |
| --- | --- |
| 1  2 | if version < (1, 3, 3):       raise ImproperlyConfigured("mysqlclient 1.3.3 or newer is required;  you have %s" % Database.\_\_version\_\_) |

注释掉 就OK了。

注意4: 如果想打印orm转换过程中的sql，需要在settings中进行如下配置：

LOGGING = {

    'version': 1,

    'disable\_existing\_loggers': False,

    'handlers': {

        'console':{

            'level':'DEBUG',

            'class':'logging.StreamHandler',

        },

    },

    'loggers': {

        'django.db.backends': {

            'handlers': ['console'],

            'propagate': True,

            'level':'DEBUG',

        },

    }

}

#### 4.单表操作

##### （1）添加表记录

###### 方式1：

导入应用models中建立的表结构类

from app01.models import Book

Book为自定义的数据库表类，继承了models.Model

book=Book(id=1,tittle=’python红宝书’,price=100,…..)

将相应的记录添加进入。

book.save()

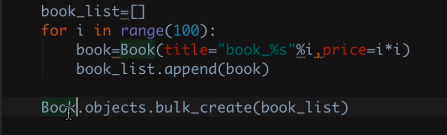
执行插入操作。

###### 方式二：

book=Book.objects.create(表字段内容)

将表记录插入数据库，并返回该记录。返回值为一个model实例，各字段为其属性。

###### 批量插入



先批量生成实例化对象，然后通过bulk\_create（）来批量生成。相当于mysql中一条insert语句，后面跟多条记录。

##### （2）单表查询

<1> all(): 查询所有结果

<2> filter(\*\*kwargs): 它包含了与所给筛选条件相匹配的对象

<3> 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。

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

<5> order\_by(\*field): 对查询结果排序

<6> reverse(): 对查询结果反向排序

<8> count(): 返回数据库中匹配查询(QuerySet)的对象数量。

<9> first(): 返回第一条记录

<10> last(): 返回最后一条记录

<11> exists(): 如果QuerySet包含数据，就返回True，否则返回False

<12> 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

<13> 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

<14> distinct(): 从返回结果中剔除重复纪录

###### ①all（）

book\_list=book.objects.all()

相当于select \* from book

将查询的所有记录返回给book\_list,是一个QuerySet的类型，处理时类似列表，但是方法比list多。

列表里放入的是每条记录实例。

**注意：通过orm取出的queryset类型，为[{},{}]列表嵌套字典的形式，数据保存在字典中，字典的关键字key为：表名\_\_字段名**

###### ②first（）、last（）

QuerySet的方法。

book\_list=book.objects.all().first()

book\_list=book.objects.all().last()

返回第一个、最后一个实例，返回一条记录实例，即model对象。

相当于book\_list[0] book\_list[-1]

###### ③filter（）

返回QuerySet类型。可以使用QuerySet方法。如first（）、last（）

相当于where

book\_list=Book.objects.filter(price=100) 可以设置多条件，逗号隔开。

相当于 select \* from Book where price=100

返回一个QuerySet的类型，处理时和列表一样。

filter（字段\_\_isnull=False）筛选该字段不为空的数据

###### ④get（）

返回值为model对象，即一条记录实例。

用法和filter一样，但是当查询结果有且只有一条时，get才有效，不然会报错。

要查询的的条件必须只有一条记录满足。

###### ⑤exclude（）

返回一个QuerySet类型

排除条件，和filter用法一模一样，只不过filter是满足条件选中，而exclude是不满足条件选中。相当于不等于……

###### ⑥order\_by(字段)

QuerySet类型的方法，对查询结果按某字段排序。默认按照升序排列，字段前加 – 号，则为降序。如“-id”，为按照id降序排列。

返回QuerySet类型。

可以进行组合。

Book.objects.all().order\_by(‘id’,’-price’)

优先按照id升序排列，如果id相等，则按照price降序排列。

###### ⑦reverse

QuerySet的方法，返回一个QuerySet类型。

对查询结果反向排序。

###### ⑧count()

QuerySet的方法，返回int类型，对符合条件的结果计数。



###### ⑨exist（）

QuerySet的方法，判断是否存在记录。



其实就是从选择出的记录中limit 1 条，判断是否存在。返回布尔值。

相当于：



###### ⑩values(字段一，字段二)

取查询结果的特定字段。

QuerySet的方法。



返回一个QuerySet类型，

这个 QuerySet类型比较特殊，处理数据时，相当于是一个列表，里面存放着字典，

[{‘prace’:Decimal(100),’id’:… },{ }]

有几个字典就有几个记录查询结果，字典里有几个key，values（）里面就有几个key。

相当于

select price from book

value里面的字段可以有多个，有几个字段，字典里面就有几个key。

###### ⑪values\_list

和values用法一样，返回也是QuerySet类型，返回的是个列表。而values返回的是个字典

但是数据处理时，相当于列表里面放着元组，有几个字段，元祖里面就有几个元素。

###### ⑫distinct

从返回结果中去掉重复记录。

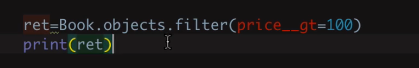
QuerySet方法，返回QuerySet类型。



##### （3）单表模糊查询\_\_

双下划线的模糊查询。

大于：



字段\_\_gt=….. 表示大于。



大于10 小于200

字段\_\_lt=…… 表示小于

\_\_startwith=’h’ 以h开头

Book.objects.filter(price\_\_in=[100,200,300]) #相当于mysql的 in，三个数据中的一个。

Book.objects.filter(price\_\_gt=100) #大于

Book.objects.filter(price\_\_lt=100) #小于

Book.objects.filter(price\_\_range=[100,200]) #范围内

Book.objects.filter(title\_\_contains="python") #包含

Book.objects.filter(title\_\_icontains="python") #忽略大小写

Book.objects.filter(title\_\_startswith="py")#开头

Book.objects.filter(pub\_date\_\_year=2012)#年份 需要时date类型。

\_\_month:月份

##### （4）删除记录

delete（）：

QuerySet的方法，也是model的方法

先查到再删除



有返回值，返回值为一个元组，有两个元素

第一个为删除记录的个数，第二个为一个字典，数据库表名为key，删除记录的条数为值。

返回值：



##### （5）修改记录

update（）

QuerySet的方法，只能QuerySet对象来调用。

先查到再更新



返回值也是一个元组，和delete（）返回值相似。

#### 5.多表操作

##### （1）orm生成关联表模型

from django.db import models

# Create your models here.

class Author(models.Model):

nid = models.AutoField(primary\_key=True)

name=models.CharField( max\_length=32)

age=models.IntegerField()

# 与AuthorDetail建立一对一的关系

authorDetail=models.OneToOneField(to="AuthorDetail",on\_delete=models.CASCADE)

class AuthorDetail(models.Model):

nid = models.AutoField(primary\_key=True)

birthday=models.DateField()

telephone=models.BigIntegerField()

addr=models.CharField( max\_length=64)

class Publish(models.Model):

nid = models.AutoField(primary\_key=True)

name=models.CharField( max\_length=32)

city=models.CharField( max\_length=32)

email=models.EmailField()

class Book(models.Model):

nid = models.AutoField(primary\_key=True)

title = models.CharField( max\_length=32)

publishDate=models.DateField()

price=models.DecimalField(max\_digits=5,decimal\_places=2)

# 与Publish建立多对一的关系,外键字段建立在多的一方

publish=models.ForeignKey(to="Publish",to\_field="nid",on\_delete=models.CASCADE)

# 与Author表建立多对多的关系,ManyToManyField可以建在两个模型中的任意一个，自动创建第三张表

authors=models.ManyToManyField(to='Author',)

###### on\_delete

on\_delete=models.CASCADE 级联删除，即删除外键关联的内容时，相应的级联内容也同时删除。

 author = models.ForeignKey(AuthModel, null=True, blank=True, on\_delete=models.SET\_NULL)

price = models.FloatField(verbose\_name='价格')

create\_time = models.DateTimeField(auto\_now\_add=True, verbose\_name='添加时间')

 2、关于别的属性的介绍

* CASCADE:这就是默认的选项，级联删除，你无需显性指定它。
* PROTECT: 保护模式，如果采用该选项，删除的时候，会抛出ProtectedError错误。
* SET\_NULL: 置空模式，删除的时候，外键字段被设置为空，前提就是blank=True, null=True,定义该字段的时候，允许为空。即：取消级联删除。
* SET\_DEFAULT: 置默认值，删除的时候，外键字段设置为默认值，所以定义外键的时候注意加上一个默认值。
* SET(): 自定义一个值，该值当然只能是对应的实体了

应该在多的那个表中建立外键，关联一的那个表



pulish为建立的外键，关联Publish表中的nid字段，注意，Django会自动为外键名添加\_id，因此publish字段名实际为publish\_id。表名Publish最好加引号。

如果不加to\_field，则默认关联主键。

###### 一对一的关系

无论在哪张表建立外键都可以，外键应该唯一。





最后还要加个参数：

onetoone方法会自动将外键唯一。

###### 多对多关系

需要建立第三张关联表。

在需要建立多对多关系的其中一张表的类下写：

假设Book表与Author表示多对多关系，那么在Book表类中写下：

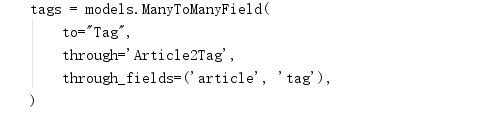


Django会自动建立第三张表，表名为book\_authors，里面字段为id、book\_id、author\_id

也可以自己建立第三张表：

假设是tag、article两个表有多对多关系，

在多对多的其中一个表（article表）中写下：

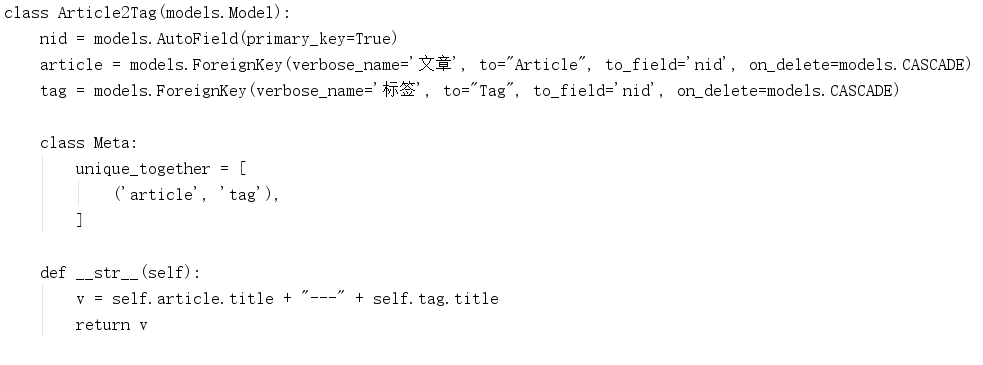


参数：to=’tag’是和tag形成多对多关系

througb=’Article2Tag’使我们要建立的第三张表

through\_fields=(‘article’,’tag’)是要关联的第三张表中的两个字段，此处一定要article在前，因为这个关联是在Atricle中建立的。

然后再建立第三张表，第三张表与另外两张表是多对一个关系。



##### （2）多表添加记录

###### ①一对多

方式一

像单表添加一样，外键也手动赋值

book\_obj=Book.objects.create(title="金瓶眉",publishDate="2012-12-12",price=100,publish\_id=1)

方式二：

把相关联的Publish的一个model对象找出来，直接相等。

publish\_obj=Publish.objects.get(nid=1)

book\_obj=Book.objects.create(title= "金瓶梅",publishDate="2012-12-12",price=100,publish=

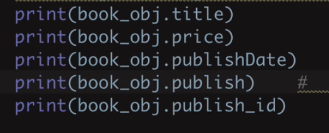
publish\_obj)

方式一和方式二实现的功能一样，但是实现过程有区别。方式一是直接输入，方式二是向找到与book相关的出版社对象publish\_obj，然后通过对象来关联输入。所以方式一是publish\_id，而方式二是publish。



publish\_id是添加外键时，djang自动根据变量建立的。

不论方式一还是方式二，最终都可以调用publish。



注意book对象都有一个publish对象属性，即是该外键关联的对象。这个属性是关联的出版社对象，包含了关联出版社的所有信息。在连表查询时很有用。

###### ②绑定多对多关系

对于自动生成的多对多的表，不能手动添加记录。

假设Book表与Author表示多对多关系，那么在Book表类中写下：



上面是在Book类中设与Author的多对多关系。

在往自动生成的表中添加记录时，想找到需要绑定作者的书籍。

第一步：获取书籍对象

book\_obj=Book.objects.filter(…).first()

或者在添加时直接指定

book\_obj= Book.objects.create(title=‘水浒传’,…….)

返回的是一个model的实例对象。

第二步：获取相关作者的对象：



如果知道了作者对象的id，也可以不用获取作者对象。假设id为1和2。

第三步 添加



或者：

：这是在知道作者id的前提下。

注意这个authors是上面设置多对多关系时在Book类中设置的属性。

会自动往关联表中添加数据。找到各自的id，并添加到关联表中。

也可以，这是python的解包

注意如果是自己建立而非自动建立的多表关系，则需要对多表关系表进行直接添加操作。

###### ③解绑

多对多解绑关系和绑定一样，

方法由add变为remove。用于有选择的进行解绑。0

而clear则是全部解绑。

对于多对多数据库的更新，一般是先解绑关系，再 重新绑定。

如果:

解除所有与该book实例有关的作者。



注意，这个book.authors对象包括了正向查询的所有信息，book----🡪author。

他是book类下的一个对象属性。

##### ④多对多关系更新

对于多对多数据库的更新，一般是先解绑关系，再 重新绑定。

方法一：先clear 再add

方法二：直接set，注意set里面放的是列表。

book\_obj.authors.set([….])

set执行两步，一个是解绑关系，然后绑定关系。

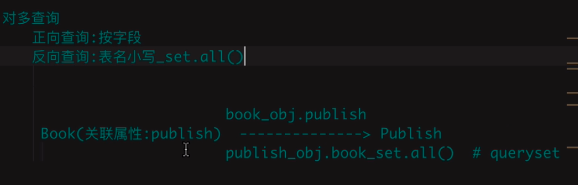
##### （3）多表查询 ：基于对象

正向查询：关联属性在A表中，通过A查B，属于正向查询。

反向查询：通过B找A。

如Book与publish是多对一的关系，关联属性在多方，即Book中，则通过书籍找出版社为正向查询，通过出版社找书籍为反向查询。

多对一：



正向查询：

book\_obj=book.objects.filter(….).first() 先拿到书籍对象，为model对象。

book\_obj.publish 再找相关联的出版社对象。为model对象。

正向查询，一本书籍只对应一个出版社，所以是一对一的关系。返回的是一个model对象。

反向查询：通过出版社找书籍：

1.先拿到相关出版社的对象：publish\_obj=publish.objects.filter(…).first() 是个model对象。

2.在找相关书籍：

publish\_obj.book\_set.all() 是个queryset对象。

注意：book\_set：为规定用法，表名\_set

反向查询，一个出版社能对应多本书，是\_set，返回的也是个queryset对象。

多对多

Book与Author为多对多关系，我们把关联属性authors放在了Book中，则通过书籍找作者为正向查询，通过作者找书籍为方向查询。

正向查询

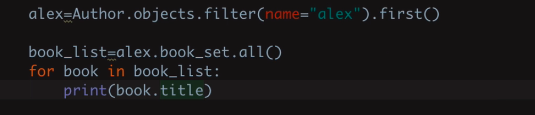


正向查询和多对一的一样。但是由于正向查询时一对多的关系，即一本书可以对应多名作者，所以返回的是个queryset对象。

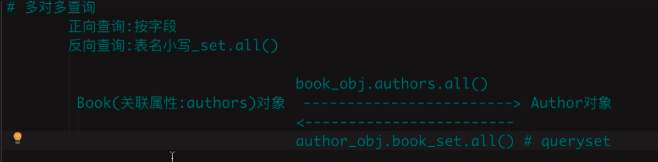
反向查询：

和多对一反向查询一样。这里的book\_set的命名方式也是规定用法。

反向查询也是一对多的关系，一个作者可以对应多本书。



总结：



一对一



注意，反向查询时，就是表名author，不加\_set。正反向查询都是返回model对象。

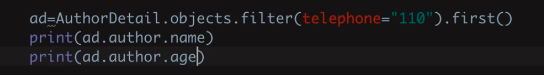
正向查询



注意，此处authordetail是Author类中带的对象属性，不是表名。

正向查询时一对一的关系，返回的是一个model对象。

反向查询



反向查询也是一对一的关系，所以表名不加\_set，返回的也是model对象。即ad.author为model对象。

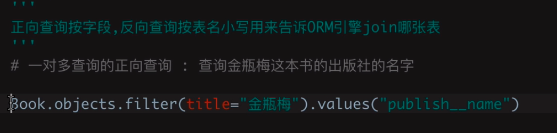
##### （4）多表查询：基于双下划线

基于join。

###### 一对多

方式一：正向查询按字段，即连表是按照字段来指示的

通过 常规的查询方向：从book表入手，通过book来找book的出版社。



以book为基础连表:先连表，publish\_\_ :book表与publish表inner join表。

此处的publish为book表中的那个字段，即book中的那个外键。

publish是建立多对一关系时，Book类下的一个属性对象。

然后通过title=金瓶梅 找到该记录，再查下name。

返回的是queryset类型，是values特有的那种特殊的 [{},{}]这种列表嵌套字典的queryset类型。

publish\_\_name:publish为表名小写，name为要取的字段。

方式二：

反向查询按表名



以publish为基础连表：book\_\_ ：publish表与book连表，此时book\_\_为表名

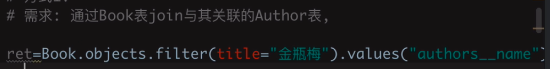
然后从title字段找出金瓶梅，然后找到name字段出版社。

###### 多对多

多对多需要用关联的第三张表，因此连表的时候需要连两次：先和关联的第三张表连表，然后再和另外一张表连表。

这两次连表，Django自动处理。一次 \_\_ 即可完成。

方式一：



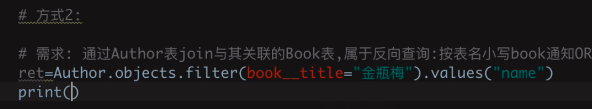
正向：以book为基础连表：

authors为建立多对多关系时，在Book类中建立的一个对象属性。

name是取author表中的name。

返回的是queryset类型，是values特有的那种特殊的 [{},{}]这种列表嵌套字典的queryset类型。

方式二：



以Author为基础连表。

###### 一对一

# 一对一查询的查询 : 查询alex的手机号  
  
# 方式1:  
# 需求: 通过Author表join与其关联的AuthorDetail表,属于正向查询:按字段authordetail通知ORM引擎join Authordetail表  
  
# ret=Author.objects.filter(name="alex").values("authordetail\_\_telephone")  
# print(ret) # <QuerySet [{'authordetail\_\_telephone': 110}]>  
#  
# # 方式2:  
# # 需求: 通过AuthorDetail表join与其关联的Author表,属于反向查询:按表名小写author通知ORM引擎join Author表  
# ret=AuthorDetail.objects.filter(author\_\_name="alex").values("telephone")  
# print(ret) # <QuerySet [{'telephone': 110}]>

###### 进阶：连续跨表，多表join

进阶练习:

注意：有些 \_\_ 是表示连表，有些不是。  
  
# 练习: 手机号以110开头的作者出版过的所有书籍名称以及书籍出版社名称

# 方式1:  
# 需求: 通过Book表join AuthorDetail表, Book与AuthorDetail无关联,所以必需连续跨表  
# ret=Book.objects.filter(authors\_\_authordetail\_\_telephone\_\_startswith="110").values("title","publish\_\_name")

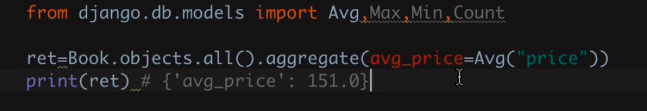
# print(ret)  
#  
# # 方式2:  
# ret=Author.objects.filter(authordetail\_\_telephone\_\_startswith="110").values("book\_\_title","book\_\_publish\_\_name")  
# print(ret)

#### 6.聚合查询

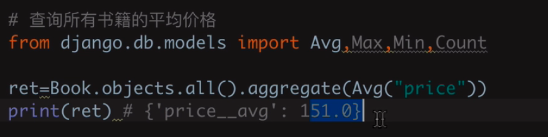
aggregate（）聚合函数统计

Avg、Max、Min、Count

返回值是一个字典。



字典主键可以设定，如果省略，则默认为是 字段\_\_avg



ret = Book.objects.all().aggregate(avg\_price=Avg(**"price"**), max\_price=Max(**"price"**))  
print(ret) # {'avg\_price': 151.0, 'max\_price': Decimal('301.00')}

可以有多个参数，在一个字典里显示。

#### 7.分组查询

##### （1）单表分组查询

语法：



annotate（）分组

例：



先指定values，分组。Emp.objects.values(‘dep’)是选中的字段，按照选中的字段进行分组。参数为分组后要查询的内容。可以有聚合函数。

返回值为一个queryset对象，是一个数组里包含字典。和values返回的样式一样。此处，只查询一个内容，字典中含有两个键值，一个是dep，一个是avg\_salary。但是也可以在里面拿到其他的内容，只要在后面加values（字段名），也可以取到其他字段的内容。



#### 8.跨表分组查询

Count需要导入

**from** django.db.models **import** Count



就是将连表和分组结合起来。

以publish为基础的连表，所以统计书名时需要加上表名book。



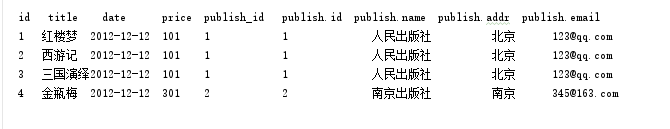
ret = Publish.objects.values(**"nid"**).annotate(c=Count(**"book\_\_title"**)).values(**"name"**, **"c"**)

values里面的c是annotate参数中的那个c。

注意，虽然分组时，未统计name，但是由于分组是按照出版社id分组的，相同的id，name是相同的，其实也就是按照name分组的（相同name，不同id分为两组），所以也可以取name值。

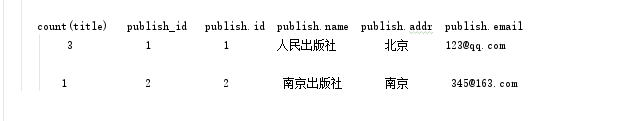
从头分析：

下图是以book为基础的连表（和以publish为基础的连表是一样的）



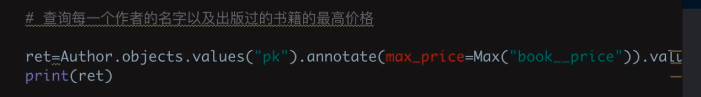
连表后进行统计输的个数：

分组后，将组内不统一的内容自动去掉，留下统一的



这些就是可以查询的内容。

多对多的例子：

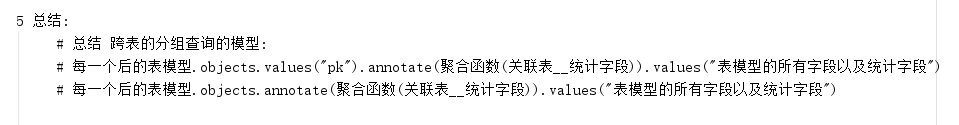


只要连表正确，分组就很容易。

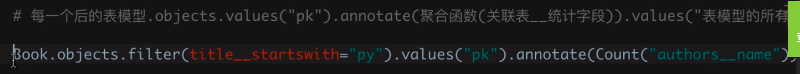
进阶：

分组也可以按照多个字段进行分组，多个字段内容完全相同的，分为一组。

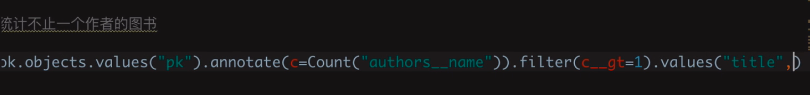
publish.object.values(‘address’,’name’).annotate(..)…



示例：



先对要分组的字段进行筛选，然后分组，然后统计。



对分组后的数据进行筛选，然后取值。

分组之后的筛选、过滤，其实就是having。

#### 9.F查询、Q查询

**from** django.db.models **import** F

##### F查询：

当字段和字段进行比较时，需要用F查询。

假设book表有两个字段，分别为price1和price2

book.objects.filter(price1\_\_gt=F(‘price2’)):查找price1大于price2的字段

##### Q查询

当条件间存在与或非关系时，使用Q查询。

与：&

或：|

非：~

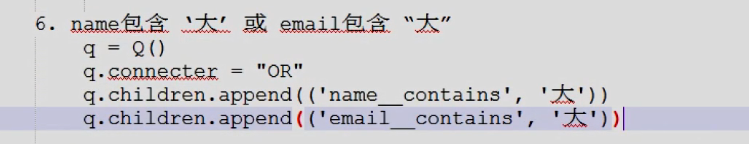
注意，条件之间用逗号隔开，本身就代表与的关系。但是或、非必须使用Q查询。

如果逗号隔开方式和Q查询联合使用，Q查询需要放在前面。

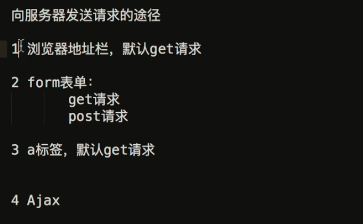
book.objects.filter(Q(price1\_\_gt=100)|Q(price2\_\_lt=150))

Q(条件1) 逻辑连接 Q（条件2）……..

book.objects.filter(Q(price1\_\_gt=100)|Q(price2\_\_lt=150), id\_\_lt=10)



### （九）Ajax



#### 1.Ajax特点：

##### 异步请求

同步交互：客户端发出一个请求后，需要等待服务器响应结束后，才能发出第二个请求；

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。

##### 局部刷新

#### 2.基于jquery实现Ajax



里面有六个参数，其中url、type、success是必须的。success是回调函数。

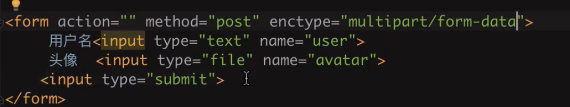
data数据为js的objec类型。在服务端收到的格式为：字典，内容为列表，列表中都是字符串。

如：data：{‘a’:1,’b’:2} 主键可加引号，可不加引号，效果一样。

服务端收到的内容为：{‘a’:[‘1’],’b’:[‘2’]}

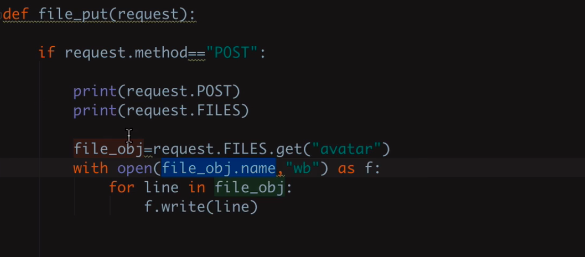
#### 3.基于form表单的文件上传

html代码：



注意，使用form表单上传文件时，需要加上enctype属性

Django代码：



上传的文件存入request.FILES中。同时，request.FILES有一个name属性，保存着上传的文件名。

通过读取request.FILES，并把其写到硬盘中，完成文件的上传。

#### 4. 请求头之contentType

规定编码格式，数据传输格式。

在form表单中，

<**form action=""enctype="application/x-www-form-urlencoded"**></**form**>

enctype规定contentType，自动转换为http协议中contenttype。

**application/x-www-form-urlencoded：按照a=5&b=4这种格式传输。 默认编码**

**multipart/form-data：传输文件。**

也是告诉后端根据什么格式进行解码。（自动过程）。

#### 5.Ajax传输json数据

js的json反序列化为:js\_data=JSON.parse(data)

如果传过来的数据在python中是字典，则在js中是object，可以传各种类型，不仅限于字符串。

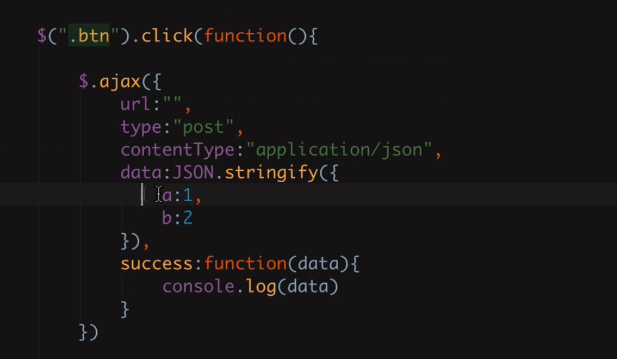
js的json序列化：JSON.stringify(data)

使用

from django.http import JsonResponse

return JsonResponse（data）

在返回数据时，直接返回，不用序列化名，而前端也直接拿到的是个object类型的数据，不用反序列化。



contentType格式为json时，传输json数据。

此时，服务端接收时，数据不存放于request.POST中，而是存放于request.body中。

注意：当contentType为：

urlencoded时：数据存放于request.POST中

json时： 数据存放于request.body中

formdata时，数据存放于request.FILES中。



发送时数据为{‘a’：‘1’，‘b’：‘2’}

接收后，进行反序列化：{‘a’：‘1’，‘b’：‘2’}

#### 6.Ajax上传文件

*// ajax上传文件***$**(**".btn"**).click(**function**(){  
 **var** formdata=**new *FormData***();  
 formdata.append(**"user"**,**$**(**"#user"**).val());  
 formdata.append(**"avatar"**,**$**(**"#avatar"**)[0].files[0]);  
 **$**.ajax({  
 **url**:**""**,  
 **type**:**"post"**,  
 **contentType**:**false**,  
 **processData**:**false**,  
 **data**:formdata,  
 success:**function**(data){  
 ***console***.log(data)  
 }  
 })  
})

新建一个FormData类的变量，存储想要发送的数据。

模式为formdata.append(键名,值)。

设置两个参数：

contentType：false

processData：false

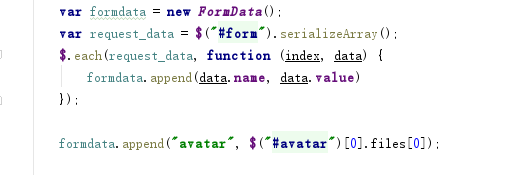
接收时，该数据以字典形式存入request.FILES。文件时以文件对象的形式存入。

不用再用json序列和反序列。

注意，和文件一起上传的在formdata中的普通数据，仍然通过request.POST来取。

便捷的将相同格式的form表单内容添加到formdata中。

serializeArray是将form表单中提交的内容的对象汇总到一个列表中



### （十）分页器

#### 1.导入分页器



#### 2. 分页器的使用:

book\_list=Book.objects.all()

paginator = Paginator(book\_list, 10) #要显示的所有数据对象，每页显示的条数。

print("count:",paginator.count) #数据总数

print("num\_pages",paginator.num\_pages) #总页数

print("page\_range",paginator.page\_range) #页码的列表，顾头不顾尾，总13页，则为[1,14]

page1=paginator.page(1) #第1页的page对象

for i in page1: #遍历第1页的所有数据对象

print(i)

print(page1.object\_list) #第1页的所有数据

page2=paginator.page(2)

print(page2.has\_next()) #是否有下一页

print(page2.next\_page\_number()) #下一页的页码

print(page2.has\_previous()) #是否有上一页

print(page2.previous\_page\_number()) #上一页的页码

# 抛错

#page=paginator.page(12) # error:EmptyPage

#page=paginator.page("z") # error:PageNotAnInteger

'''

book\_list=Book.objects.all()

paginator = Paginator(book\_list, 10)

page = request.GET.get('page',1)

currentPage=int(page)

try:

print(page)

book\_list = paginator.page(currentPage)

except PageNotAnInteger:

book\_list = paginator.page(1)

except EmptyPage:

book\_list = paginator.page(paginator.num\_pages)

return render(request,"index.html",{"book\_list":book\_list,"paginator":paginator,"currentPage":currentPage})

#### 3.对分页加页码

利用bootcss框架及python代码。

py代码：

**def** index(request):  
 *'''  
 批量导入:  
 book\_list=[]  
 for i in range(100):  
 book=Book(title="book\_%s"%i,price=i\*i)  
 book\_list.append(book)  
  
 Book.objects.bulk\_create(book\_list)* **:param** *request:* **:return***:  
 '''* book\_list=Book.objects.all()  
  
 *# 分页器* paginator=Paginator(book\_list,3)  
  
 print(**"count:"**,paginator.count) *#数据总数* print(**"num\_pages"**,paginator.num\_pages) *#总页数* print(**"page\_range"**,paginator.page\_range) *#页码的列表* current\_page\_num=int(request.GET.get(**"page"**,1))  
  
 **if** paginator.num\_pages>11:  
  
 **if** current\_page\_num-5<1: 如果当前页码小于5，则显示前11页页码  
 page\_range=range(1,12)  
 **elif** current\_page\_num+5>paginator.num\_pages: 如果当前页码距离最后页码小于5页，显示最后11页页码。  
 page\_range=range(paginator.num\_pages-10,paginator.num\_pages+1)  
  
 **else**: #将当前页码居中，前后各五个页码。  
 page\_range=range(current\_page\_num-5,current\_page\_num+6)  
 **else**:  
 page\_range=paginator.page\_range #如果总的页码小于11页，全部显示。  
  
  
  
 **try**:  
  
  
 current\_page=paginator.page(current\_page\_num)  
  
 *# 显示某一页具体数据的两种方式:* print(**"object\_list"**,current\_page.object\_list)  
 **for** i **in** current\_page:  
 print(i)  
  
 **except** EmptyPage **as** e:  
 current\_page=paginator.page(1)  
  
  
 **return** render(request,**"index.html"**,locals())

html代码：

<**ul**>  
 {% for book in current\_page %}  
 <**li**>{{ book.title }}:{{ book.price }}</**li**>  
 {% endfor %}  
  
</**ul**>  
  
  
<**nav aria-label="Page navigation"**>  
 <**ul class="pagination"**>  
 {% if current\_page.has\_previous %}  
 <**li**><**a href="?page={{ current\_page.previous\_page\_number }}" aria-label="Previous"**><**span aria-hidden="true"**>上一页</**span**></**a**></**li**>  
 {% else %}  
 <**li class="disabled"**><**a href="" aria-label="Previous"**><**span aria-hidden="true"**>上一页</**span**></**a**></**li**> //利用bootstrap框架，class=disable，是灰色的按钮，不能点击。  
 {% endif %}  
  
  
  
 {% for item in page\_range %}  
   
 {% if current\_page\_num == item %}  
 <**li class="active"**><**a href="?page={{ item }}"**>{{ item }}</**a**></**li**>  
 {% else %}  
 <**li**><**a href="?page={{ item }}"**>{{ item }}</**a**></**li**> //在页面不改变的情况下，可以只传输数据。  
 {% endif %}  
  
 {% endfor %}  
  
  
  
 {% if current\_page.has\_next %}  
 <**li**><**a href="?page={{ current\_page.next\_page\_number }}" aria-label="Next"**><**span aria-hidden="true"**>下一页</**span**></**a**>  
 {% else %}  
 <**li class="disabled"**><**a href="" aria-label="Next"**><**span aria-hidden="true"**>下一页</**span**></**a**>  
 {% endif %}  
</**li**>  
 </**ul**>  
</**nav**>

### （十一）form组件

#### 1.创建form组件

先引入

**from** django **import** forms  
  
**from** django.forms **import** widgets



定义一个类，属性为想校验的数据名，并定义好数据的类型。

验证时，将校验的值以字典形式作为参数，实例化form。

如果所有值通过验证，则form.is\_valid()为True，反之为False。

传入参数实例化时，传入不相干的参数将会被无视，但UserForm定义的参数必须传入，传入的字典的键名就是UserForm的属性值，不然将返回False。



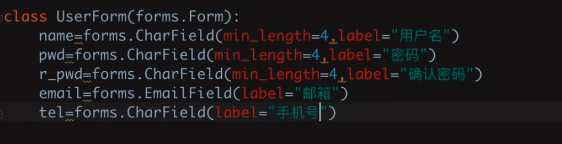
符合规则的，将会放入form.cleaned\_data中，以字典形式保存，不符合规则的，放入form.errors中，以变量名字符串为键值，错误信息为内容，内容为列表，可以有多个错误。

多余的参数，将被无视。

form=UserForm(request.POST)对form表单传过来的值进行验证。

#### 2.from组件渲染标签

主要是批量产生标签时比较便捷。



首先，在定义类的时候，填写label参数，这是form标签相关输入值得字段名。

视图函数中，在url get请求的条件下实例化：

form=UserForm()  
并返回：

**return** render(request,**"reg.html"**,locals())

其中包含{‘form’：form}

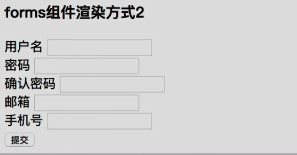


label 中for 可以赋值： {{ field.auto\_id }} 自动关联input该input标签。

<label for=”{{ field.auto\_id }}”>{{ field.label }}</label>

在html中利用循环进行书写。

Django会自动根据变量的类型，添加相关的input标签。



此为生成的表单。

另外一种方法：



直接调用as\_p方法，则会整体生成。

as\_p 是p标签中放label、input标签。

还有as\_ul是ul标签中放label、input标签



##### 添加错误信息：

if 登录验证合格：

跳转

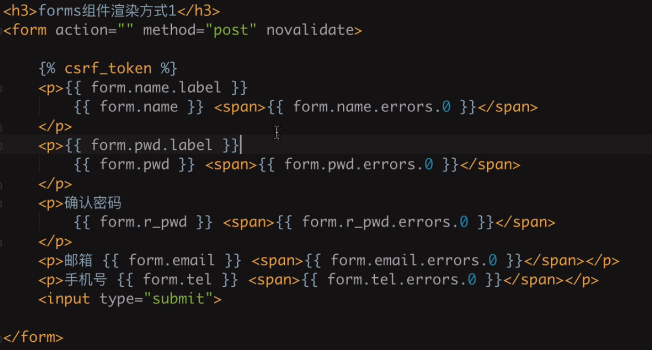
else：

返回登录界面。**return** render(request,**"reg.html"**,locals())

由于此时form实例有值，他会将值填入相应的表单之中。也就是前台你填写的数据不会被清空。

同时可以显示错误的信息：

form.属性名.errors.0 将显示这个字段错误信息。



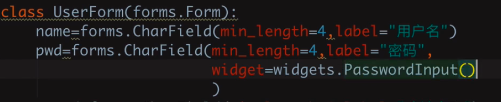
#### 3.form组件参数

##### （1）选择想要的input类型

导入widget模块



然后在参数设置中选择相应的input类型，默认为CharField类型对应的是text类型，可以在参数中指定，如pwd指定的是password类型：



##### （2）指定错误信息

参数error\_messages={'错误名（Django固定）’:‘显示的内容’，…..}



required:为空 错误，未填写

invalid：格式错误。

##### （3）调整表单属性：

如class、id等



为属性widget赋值，在指定input属性后，里面添加参数，attrs=字典形式。

#### 4.局部钩子

先进行form自有的校验，然后再进行自定义的校验。

定义一个判断条件，用户名是否已被注册：

在表单类 UserForm中定义一个方法：

注意这个方法的名称是固定的：clean\_字段名

**def** clean\_name(self):  
  
 val=self.cleaned\_data.get(**"name"**)  
  
 ret=UserInfo.objects.filter(name=val) #UserInfo为数据库表。  
  
 **if not** ret:  
 **return** val  
 **else**:  
 **raise** ValidationError(**"该用户已注册!"**)

注意：格式符合自定义的要求，就返回原值，

当格式不符合要求时，抛出ValidationError异常。

#### 5.全局钩子

用于数据间的互相比较

如密码的两次输入是否一致。需要将这两个数据进行比对。此时可以用全局钩子。

全局钩子错误信息保存在form.errors字典中的\_\_all\_\_键中



全局钩子名称为clean，在自定义的UserForm中定义。

**def** clean(self):  
 pwd=self.cleaned\_data.get(**'pwd'**)  
 r\_pwd=self.cleaned\_data.get(**'r\_pwd'**)  
  
 **if** pwd **and** r\_pwd:  
 **if** pwd==r\_pwd:  
 **return** self.cleaned\_data  
 **else**:  
 **raise** ValidationError(**'两次密码不一致'**)  
 **else**:  
  
 **return** self.cleaned\_data

### （十二）cookie

#### 1.http协议无状态保存

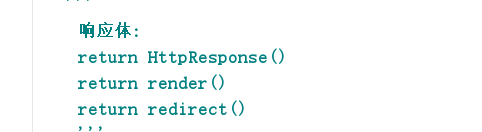
#### 2.cookie简介

是key-value结构，类似于一个python中的字典。随着服务器端的响应发送给客户端浏览器。然后客户端浏览器会把Cookie保存起来，当下一次再访问服务器时把Cookie再发送给服务器。 Cookie是由服务器创建，然后通过响应发送给客户端的一个键值对。客户端会保存Cookie，并会标注出Cookie的来源（哪个服务器的Cookie）。当客户端向服务器发出请求时会把所有这个服务器Cookie包含在请求中发送给服务器，这样服务器就可以识别客户端了！

#### 3.cookie设置与读取

需要用响应体来设置cookie。

响应体：



这三个属于响应体。

设置cookie并将其发送给浏览器：



设置有两个参数，第一个是键，第二个是值。

取cookie

在request.COOKIES中以字典形式保存。

#### 4.cookie参数设置

max\_age以秒为单位，指定cookie将要延续多长时间。

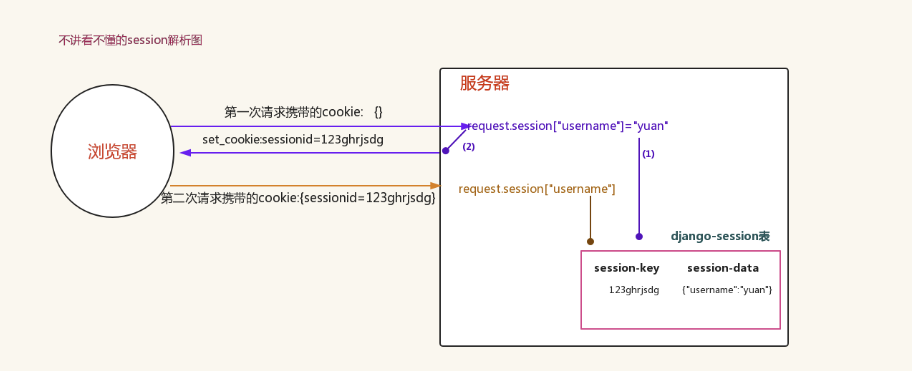
expire指定在什么时间失效。可以给expire赋值datatime类型。



删除cookie

response.delete\_cookie("cookie\_key键名",path="/",domain=name)

### （十三）session



相当于把要用到的信息存入cookie，由服务端保存起来，保存到django\_session数据表中，服务器发一个“钥匙”给客户端，通过钥匙取到信息。而钥匙的具体内容和保存的信息没关系，便于保密。

这个钥匙就是一个键为sessionid的cookie，值为django生成的随机字符串。

1、设置Sessions值

          request.session['session\_name'] ="admin"

2、获取Sessions值

          session\_name = request.session["session\_name"]

3、删除Sessions值

          del request.session["session\_name"]

4、flush()

     删除当前的会话数据并删除会话的Cookie。

     这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

5、get(key, default=None)

fav\_color = request.session.get('fav\_color', 'red')

6、pop(key)

fav\_color = request.session.pop('fav\_color')

7、keys()

8、items()

9、setdefault()

10 用户session的随机字符串

        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()

        # 检查 用户session的随机字符串 在数据库中是否

        request.session.exists("session\_key")

        # 删除当前用户的所有Session数据

        request.session.delete("session\_key")

        request.session.set\_expiry(value)

            \* 如果value是个整数，session会在些秒数后失效。

            \* 如果value是个datatime或timedelta，session就会在这个时间后失效。

            \* 如果value是0,用户关闭浏览器session就会失效。

            \* 如果value是None,session会依赖全局session失效策略。

对于同一浏览器，如果重新进行登录，则该session对应的值并不会变（即钥匙并不会改变），但是session表中保存的数据会改变。也就是锁没有变，但是保存的东西发生了改变。

删除session键值：

del request.session[‘key’]

s=request.session.pop(‘key’) 删除并返回

其实就是字典的删除，但是只删除了session中的一个字段，并没有完全删除session，不推荐这样做。

request.session.flush()：删除当前session并删除相应cookie

setting.py中配置session



### （十四）用户认证组件

#### 1.auth模块

from django.contrib import auth

django自己提供了一个用户表auth\_user，然后提供了基于这个用户表的认证方法。此认证方法更加严谨安全。

如：同一浏览器改变登录信息后，session的钥匙要随之改变，会把原session完全删除，重新建议一条session。

##### （1）authenticate

提供了用户认证，即验证用户名以及密码是否正确,一般需要username password两个关键字参数

如果认证信息有效，会返回一个 User 对象。authenticate()会在User 对象上设置一个属性标识那种认证后端认证了该用户，且该信息在后面的登录过程中是需要的。当我们试图登陆一个从数据库中直接取出来不经过authenticate()的User对象会报错的！！

user = authenticate(username='someone',password='somepassword')

其实就是取User表中查找有没有相关信息。如果有，则登录成功，返回user对象，没有就返回None。

##### （2）auth.login(request,user)

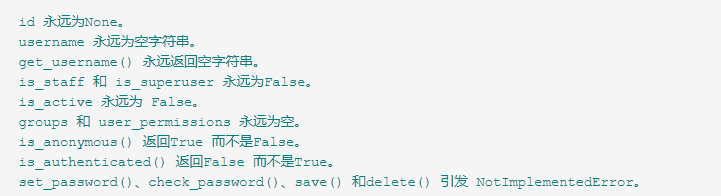
进行request.session的设置，同时request中一个user对象，和我们传入的user对象一样。

即：request.user=user

request.user永远是当前登录对象。如果没有登录成功，则request.user是一个匿名对象。

匿名对象中

request.user



而正常的非匿名request.user. 上面的属性都是有正常的值。

所以可以通过判断id、username、get\_username()、is\_anonymous（注意，上面书写错误，没有括号）的状态来判断用户是否登录成功。

##### （3）auth.logout(request)

注销。

#### 2.uesr对象

User 对象属性：username， password（必填项）password用哈希算法保存到数据库

##### （1）is\_authenticated

如果是真正的 User 对象，返回值恒为 True 。 用于检查用户是否已经通过了认证。

登录成功，返回True，登录失败，则为false。

##### （2）创建用户

使用 create\_user 辅助函数创建用户:

from django.contrib.auth.models import User

user = User.objects.create\_user（username='',password='',email=''）

使用create\_user 会自动对password进行哈希计算，然后存入auth\_user表中。

check\_password(passwd)

创建超级用户：

使用cmd

python manage.py createsuperuser

然后输入用户名、邮箱、密码，邮箱可以省略。

##### （3）检测密码是否相符

check\_password(passwd)

用户需要修改密码的时候 首先要让他输入原来的密码 ，如果给定的字符串通过了密码检查，返回 True

##### （4）修改密码

使用 set\_password() 来修改密码

user = User.objects.get(username='') #获取user对象

user.set\_password(password='') #修改密码

user.save　#保存

##### （5）跳转的实现

1 用户登陆后才能访问某些页面，

2 如果用户没有登录就访问该页面的话直接跳到登录页面

3 用户在跳转的登陆界面中完成登陆后，自动访问跳转到之前访问的地址

步骤一：

方法1:

def my\_view(request):

if not request.user.is\_authenticated():

return redirect('%s?next=%s' % (settings.LOGIN\_URL, request.path))

方法2:

django已经为我们设计好了一个用于此种情况的装饰器：login\_requierd()

from django.contrib.auth.decorators import login\_required

@login\_required

def my\_view(request):

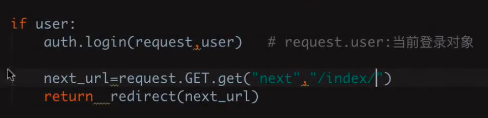
...

若用户没有登录，则会跳转到django默认的 登录URL '/accounts/login/ ' (这个值可以在settings文件中通过LOGIN\_URL进行修改)。并传递 当前访问url的绝对路径 (登陆成功后，会重定向到该路径)。

步骤二：

在登录逻辑判断处：

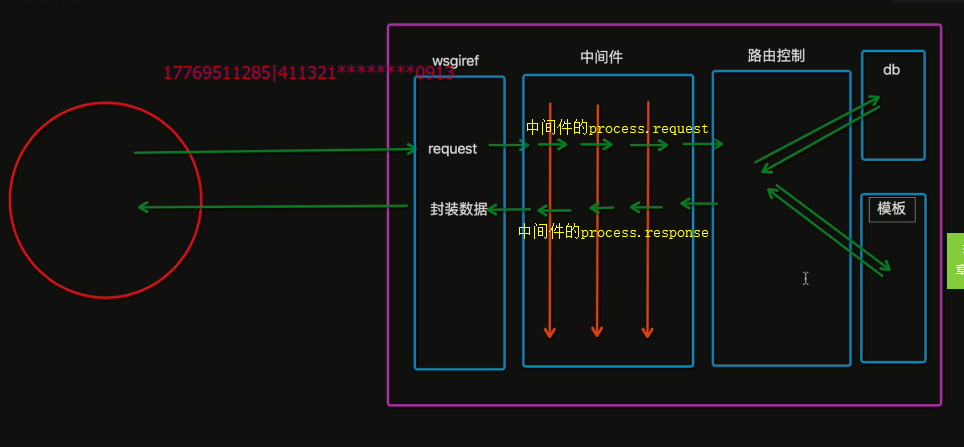
在登录成功逻辑地方 request.GET.get(‘next’)，在登录成功之后，重定向至next指向的url。



如果next为空，则取‘/index/’



### （十五）中间件



每个中间件不是必须得有request、response方法。

#### 创建自己的中间件

在setting.py中的MIDDLEWARE列表中添加自己定义的中间件。是一个类。

新建一个文件，定义中间件。仿照其他中间件，导入相关的库。定义的类要继承MiddlewareMixin类。

from django.utils.deprecation import MiddlewareMixin

定义process\_request（self,request）、process\_response(self,request,response)方法，不用全部定义。

中间件process\_request在数据进入时，肯定会被执行。相当于预处理。所以一些涉及到总体设计、限制、验证，可以通过中间件来实现。

中间件process\_response在数据输出时，肯定会被执行。

中间件中常用有四个方法：

process\_request

process\_view

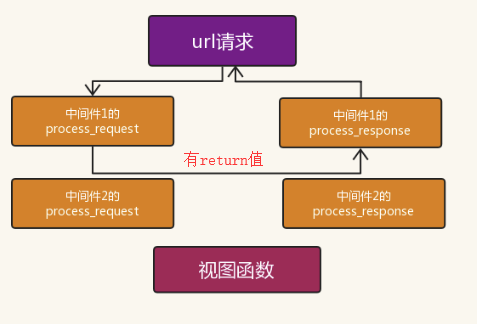
process\_exception

process\_response

##### ①process\_request（self,request）

一般没有返回值，或者返回值为None。（返回值为None与没有返回值等效）

如果添加了返回值，数据将不会继续进行下一步操作，而是就此返回至浏览器，可以做拦截使用。



##### ②process\_response(self,request,response)

必须有返回值

return response

response指的是视图函数中返回的响应体。

##### ③process\_view()

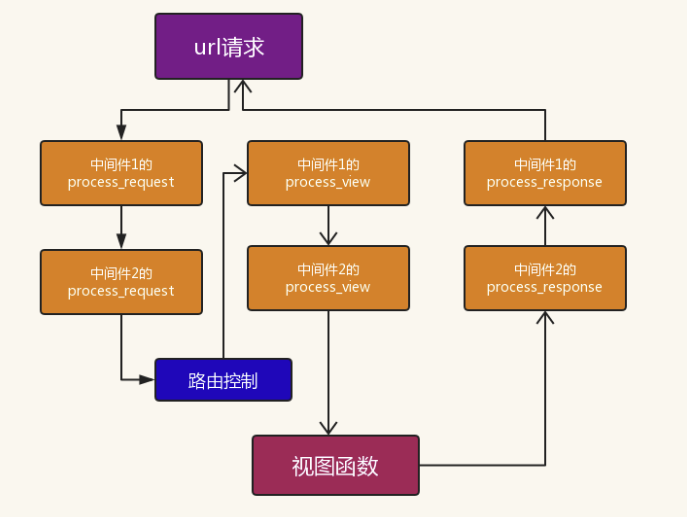
在路由控制之后，视图函数之前运行。一般没有返回值

注意：process\_view如果有返回值，会越过其他的process\_view以及视图函数，但是所有的process\_response都还会执行。所以可以利用此提前一步进行，可以进行视图函数执行的拦截。

process\_view(self,request,callback,callback\_args, callback\_kwargs)

callback为地址分发分配的视图函数。

callback\_args, callback\_kwargs为传入callback函数中的参数。



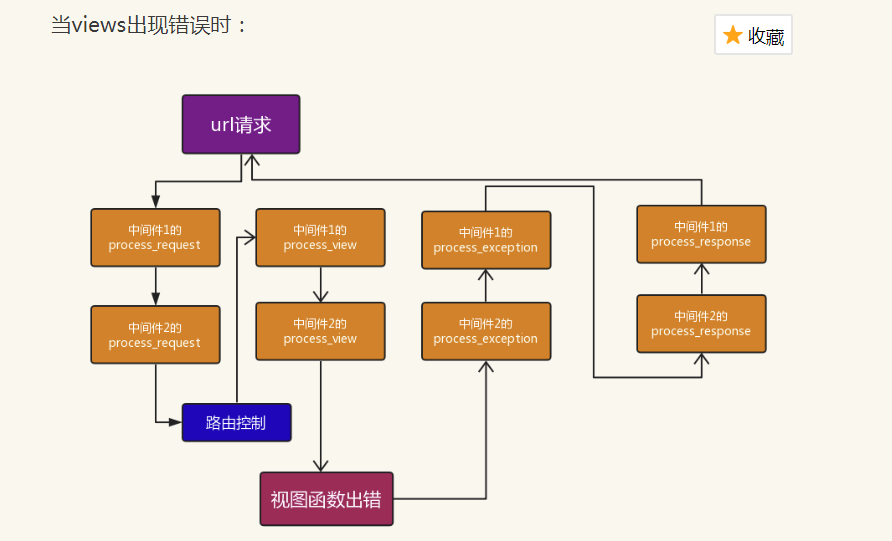
##### ④process\_exception

process\_exception(self, request, exception)

当视图函数出现错误时，会执行。

先执行

exception为接收到的错误信息。



注意，只要执行的process\_exception函数中有返回值，就会越过后面所有的process\_exception函数，直接执行process\_response函数。

函数执行顺序：以视图函数为分界线，进入的时候，按照setting.py中MIDDLEWARE列表由前往后的顺序执行各个中间件(process.request、process.view)，返回数据的时候，按照相反的顺序执行中间件（process.expection、process.respoonse）。

#### 中间件应用