**WEB框架开发**

Django框架开发

Django 中文官方文档：https://docs.djangoproject.com/zh-hans/2.0/

# 1.WEB 应用

Web应用程序是一种可以通过种可以通过Web访问的应用程序，程序的最大好处是用户很容易访问应用程序，用户只需要有浏览器即可，不需要再安装其他软件。应用程序有两种模式C/S、B/S。C/S是客户端/服务器端程序，也就是说这类程序一般独立运行。而B/S就是浏览器端/服务器端应用程序，这类应用程序一般借助谷歌，火狐等浏览器来运行。WEB应用程序一般是B/S模式。Web应用程序首先是“应用程序”，和用标准的程序语言，如java，python等编写出来的程序没有什么本质上的不同。在网络编程的意义下，浏览器是一个socket客户端，服务器是一个socket服务端。

# 2.http协议

## 1.协议简介

HTTP协议是Hyper Text Transfer Protocol（超文本传输协议）的缩写,是用于万维网服务器与本地浏览器之间传输超文本的传送协议。

HTTP是一个属于应用层的面向对象协议，由于其简捷、快速的方式，适用于分布式超媒体信息系统。HTTP协议工作于客户端-服务端架构为上。浏览器作为HTTP客户端通过URL向HTTP服务端即WEB服务器发送所有请求。Web服务器根据接收到的请求后，向客户端发送响应信息。

## 2.http协议特性

1）基于TCP/IP ：HTTP协议是基于TCP/IP协议上的应用层协议。

2）基于请求-响应模式 ：HTTP协议规定，请求从客户端发出，最后服务器端响应该请求并返回。换句话说，肯定是先从客户端开始建立通信的，服务端在没有接收到请求之前不会响应。

3）无状态保存 ： HTTP是一种不保存状态,即无状态(stateless)协议。HTTP协议 自身不对请求和响应之间的通信状态进行保存。也就是说在HTTP这个 级别,协议对于发送过的请求或响应都不做持久化处理。

使用HTTP协议,每当有新的请求发送时,就会有对应的新响应产生。协议本身并不保留之前一切的请求或响应报文的信息。这是为了更快地处理大量事务,确保协议的可伸缩性,而特意把HTTP协议设计成 如此简单的。可是,随着Web的不断发展,因无状态而导致业务处理变得棘手 的情况增多了。比如,用户登录到一家购物网站,即使他跳转到该站的 其他页面后,也需要能继续保持登录状态。针对这个实例,网站为了能 够掌握是谁送出的请求,需要保存用户的状态。HTTP/1.1虽然是无状态协议,但为了实现期望的保持状态功能, 于是引入了Cookie技术。有了Cookie再用HTTP协议通信,就可以管 理状态了。有关Cookie的详细内容稍后讲解。

4）无连接 ：无连接的含义是限制每次连接只处理一个请求。服务器处理完客户的请求，并收到客户的应答后，即断开连接。采用这种方式可以节省传输时间。

## 3.http请求协议与响应协议

http协议包含由浏览器发送数据到服务器需要遵循的请求协议与服务器发送数据到浏览器需要遵循的请求协议。用于HTTP协议交互的信被为HTTP报文。请求端(客户端)的HTTP报文做请求报文,响应端(服务器端)的做响应报文。HTTP报文本身是由多行数据构成的字文本。



### 1）请求协议

请求格式：



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包的请求体中.
* 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。

### 2）响应协议

响应格式：



响应状态码：状态码的值是当客户端向服务端发送请求时，返回的请求结果。借助状态码，用户可以知道服务器端是否正常处理了请求。



实例：

# -\*- coding: utf-8 -\*-  
*import* socket  
sock = socket.socket()  
sock.bind(("127.0.0.1",8801))  
sock.listen(5)  
*while True*:  
 # conn 客户端套接字  
 conn,addr = sock.accept()  
 data = conn.recv(1024)  
 # HTTP/1.1 200 OK 相应首行  
 # 读取html文件  
 # POST 请求才有请求头数据  
 # 按照http请求协议解析数据  
 # 专注于web业务开发  
 # 按照http响应协议封装数据  
 *with* open('login.html','r') *as* f:  
 data = f.read()  
 conn.send(("HTTP/1.1 200 OK\r\n\r\n%s"%data).encode('utf8'))  
 conn.close()

login.html

<!DOCTYPE html>  
<**html** lang=**"en"**>  
<**head**>  
 <**meta** charset=**"UTF-8"**>  
 <**title**>**login**</**title**>  
</**head**>  
<**body**>  
<**form** action=**"http://127.0.0.1:8801"** method=**"post"**>  
 **用户名：**<**input** type=**"text"** name=**"user"**>  
 **密码：**<**input** type=**"password"** name=**"pwd"**>  
 <**input** type=**"submit"** value=**"提交"**>  
</**form**>  
</**body**>  
</**html**>

# 3.wsgiref模块

WSGI（Web Server Common Interface）是专门为Python语言制定的web服务器与应用程序之间的网关接口规范，通俗的来说，只要一个服务器拥有一个实现了WSGI标准规范的模块（例如apache的mod\_wsgi模块），那么任意的实现了WSGI规范的应用程序都能与它进行交互。因此，WSGI也主要分为两个程序部分：服务器部分和应用程序部分。

其实 wsgiref 最基本的功能就是封装了socket，不需要自己再去写socket。因为我们不希望接触到TCP连接、HTTP原始请求和响应格式，所以，需要一个统一的接口协议来实现这样的服务器软件，让我们专心用Python编写Web业务。这个接口就是WSGI：Web Server Gateway Interface。而wsgiref模块就是python基于wsgi协议开发的服务模块。

*from* wsgiref.simple\_server *import* make\_server  
  
*def* application(*environ*, *start\_response*):  
 *start\_response*('200 OK', [('Content-Type', 'text/html')])  
 *return* [b'<h1>Hello, web!</h1>']

httpd = make\_server('', 8080, application)  
print('Serving HTTP on port 8000...')  
# 开始监听HTTP请求:  
httpd.serve\_forever()

# 4.DIY一个WEB框架

利用wsgiref模块实现一个简单的WEB框架

目录结构如下：



DIY 的web框架 功能简介：  
main.py 启动文件 ，封装socket   
1. urls.py ：路径与视图函数映射关系 ---- url 控制器  
2. views.py ： 视图函数，固定有一个形参：environ ---- 视图函数  
3. templates文件夹：存放html文件 ---- 模板  
4.modules : 在项目启动前，在数据库中创建数据库表结构的 ----与数据相关

main.py：

*from* wsgiref.simple\_server *import* make\_server  
*from* urls *import* url\_patterns  
*def* application(*environ*, *start\_response*):  
 path = *environ*.get("PATH\_INFO")  
 start\_response('200 OK', [('Content-Type', 'text/html')])  
 *print*(path)  
 func = None  
 *for* item *in* url\_patterns:  
 *if* path == item[0]:  
 func = item[1]  
 *break  
 if* func:  
 *return* [func(*environ*)]  
 *else*:  
 *return* [b'404']  
httped=make\_server("",8066,application) # 封装了socket ,application 是回调函数  
# 等待用户连接：conn,addr = sock.accept()  
httped.serve\_forever()

urls.py:

# -\*- coding: utf-8 -\*-  
# \_\_title\_\_ = 'urls.py'  
# \_\_author\_\_ = 'YangYang'  
# \_\_mtime\_\_ = '2018.06.19'  
*from* views *import* \*  
url\_patterns = [  
 ("/login", login),  
 ("/index", index),  
 ("/favicon.ico",fav),  
 ("/timer",timer),  
 ("/auth",auth),  
]

views.py

# -\*- coding: utf-8 -\*-  
# \_\_title\_\_ = 'views.py'  
# \_\_author\_\_ = 'YangYang'  
# \_\_mtime\_\_ = '2018.06.19'  
*import* pymysql  
*from* urllib.parse *import* parse\_qs  
  
*def* login(environ):  
 *with* open('templates/login.html','rb') *as* f:  
 data = f.read()  
 *return* data  
  
*def* index(environ):  
 *with* open('templates/index.html','rb') *as* f:  
 data = f.read()  
 *return* data  
  
*def* fav(environ):  
 *with* open('templates/favicon.ico','rb') *as* f:  
 data = f.read()  
 *return* data  
  
*def* timer(environ):  
 *import* datetime  
 now = datetime.datetime.now().strftime("%y-%m-%d %X")  
 *return* now.encode('utf8')  
  
*def* auth(*request*):  
  
 *try*:  
 request\_body\_size = int(*request*.get('CONTENT\_LENGTH', 0))  
 *except* (ValueError):  
 request\_body\_size = 0  
  
 request\_body = *request*['wsgi.input'].read(request\_body\_size)  
 data = parse\_qs(request\_body)  
  
 user = data.get(b"user")[0].decode("utf8")  
 pwd = data.get(b"pwd")[0].decode("utf8")  
 conn = pymysql.connect(host='101.132.161.180', port=3306, user='root', passwd='123456', db='new\_web') # db：库名  
 # 创建游标  
 cur = conn.cursor()  
 SQL = "select \* from userinfo WHERE NAME ='%s' AND PASSWORD ='%s'" % (user, pwd)  
 cur.execute(SQL)  
  
 *if* cur.fetchone():  
 f = open("templates/index.html", "rb")  
 data = f.read()  
 data = data.decode("utf8")  
 *return* data.encode("utf8")  
 *else*:  
 *return* b"user or pwd is wrong"

modules.py

# -\*- coding: utf-8 -\*-  
# \_\_title\_\_ = 'modules.py'  
# \_\_author\_\_ = 'YangYang'  
# \_\_mtime\_\_ = '2018.06.19'  
# 生成用户表  
*import* pymysql  
#连接数据库  
conn = pymysql.connect(host='101.132.161.180',port= 3306,user = 'root',passwd='123456',db='new\_web') #db：库名  
#创建游标  
cur = conn.cursor()  
sql='''  
create table userinfo(  
 id INT PRIMARY KEY ,  
 name VARCHAR(32) ,  
 password VARCHAR(32)  
)  
'''  
cur.execute(sql)  
#提交  
conn.commit()  
#关闭指针对象  
cur.close()  
#关闭连接对象

# 5.MVC与MTV模型

## 1.MVC

Web服务器开发领域注明的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：



## 2.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：

* M代表模型（Model）：负责业务对象和数据库的关系映射（ORM）。
* T 代表模板（Template）：负责如何把页面展示给用户（html）。
* 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，最后返回网页给用户。

# 6.Django的下载与基本命令

## 1.安装Django

windows : python3 -m pip install django===2.0.1

django安装路径地址：C:\Python\Python36\Scripts

linux : python3 install django===2.0.1

django安装路径地址：/usr/local/python36/bin

## 2.创建一个Django project

创建一个文件夹用于存放django项目：mkdir my\_pro，然后执行

windows：django-admin startproject mysite

linux： django-admin.py startproject mysite

## 3.创建mysite后生成的文件的作用

[root@dev mysite]# tree

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

└── mysite

├── \_\_init\_\_.py

├── 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的变量。

├── urls.py ----- 负责把URL模式映射到应用程序。路由相关

└── wsgi.py ----- 封装的socket，实际上，django走的第一环就是这个文件

## 4.创建一个应用属于mysite项目

cd mysite  
python3 manage.py startapp blog  
  
[root@dev mysite]# tree blog/  
blog/  
├── admin.py  
├── apps.py  
├── \_\_init\_\_.py  
├── migrations  
│   └── \_\_init\_\_.py  
├── models.py ---- 数据库相关  
├── tests.py  
└── views.py ---视图函数

## 5.创建一个templates目录，用于存放html文件



## 6.启动django项目

python3 manage.py runserver 8080 (默认8000端口)

# 7.Django简单示例及静态文件配置

## 1.创建一个 static\_file\_configuration 项目和一个app01的应用



## 2.需求说明

利用django实现一个页面展示当前时间，并且点击以后不断变色的方案

## 3.django访问顺序

首先，当用户通过URL访问服务器django项目的时候，首先通过urls控制器去匹配一个视图函数，然后，如果匹配到视图函数，那么会跳转到views.py里面找到对应的函数名称，然后，views.py会把要展示的数据给到templates，templates里面的页面在经过处理后，会把html页面发送给用户。因此，接下来的的步骤就是：urls.py ---> views.py ---->template --return-->用户

## 4.urls.py控制器配置

*from* django.contrib *import* admin  
*from* django.urls *import* path  
  
*from* app01 *import* views #导入对应应用的视图  
  
urlpatterns = [  
 path('admin/', admin.site.urls),  
 path('timer/',views.timer), #添加路由转发  
]

## 5.views.py 创建视图（逻辑业务处理）

*def* timer(*request*):  
 *import* time  
 ctime = time.time()  
 *return* render(*request*,'timer.html',{"ctime":ctime})

## 6.创建页面

注意：django的views.py 传送变量给html的时候，在html里面用 {{变量}} 去使用变量。

<!DOCTYPE html>  
<**html** lang=**"en"**>  
<**head**>  
 <**meta** charset=**"UTF-8"**>  
 <**title**>**timer**</**title**>  
 <**link** rel=**"stylesheet"** href=**"/static/app01/timer.css"**>  
</**head**>  
<**body**>  
<**h4**>**hi** {{ ctime }}</**h4**>  
</**body**>  
 <**script** src=**"/static/jquery-3.3.1.js"**></**script**>  
 <**script** type=**"text/javascript"** src=**"/static/app01/timer.js"**></**script**>  
</**html**>

## 7.创建css js

css文件：

h4{  
 *color*: **red**;  
}

js文件：

$(function () {  
 var a = 0;  
 $('h4').click(function () {  
 a++;  
 if(a%4===1){  
 $(this).css('color','green')  
 }else if(a%4===2) {  
 $(this).css('color','yellow')  
 }else if (a%4===3){  
 $(this).css('color','blue')  
 }else{  
 $(this).css('color','red')  
 }  
 })  
});

## 8.静态文件说明（重要）

静态文件我们一般会在该项目下面创建一个static的目录，然后用应用名称来解耦合，存放各自应用的js和css文件。如下图所示：



但是，此刻会发现，在该项目下，创建的是一个 statics的目录去存放，但是，为啥在html文件里面引入的是 static呢。这里就需要配置项目的setting.py文件

# statics的别名  
STATIC\_URL = '/static/'  
  
STATICFILES\_DIRS=[  
 os.path.join(BASE\_DIR,"statics")  
]

STATIC\_URL = '/static/' 这个是是静态文件目录的别名。这里可以更改为别的，但是一般情况下都是使用static。

STATICFILES\_DIRS ：这个是静态文件在该项目下的路径。

BASE\_DIR 可以在setting.py 的文件上面找到。

## 9.启动django项目

第一种方法：

点击箭头启动

第二种方法：python3 manage.py runserver 8000

## 10.页面展示



# 8.路由配置

## 1.路由控制之简单配置（基于正则表达式）

*from* django.contrib *import* admin  
*from* django.urls *import* path,re\_path  
# path django===2.0.\*  
# re\_path django===1.0.\*  
*from* app01 *import* views #导入对应应用的视图  
urlpatterns = [  
 path('admin/', admin.site.urls),  
 path('timer/',views.timer), #添加路由转发  
 # 路由配置： 路径----> 视图函数  
 re\_path(r'^articles/2003/$', views.special\_case\_2003), #special\_case\_2003(request) request是请求对象  
 # 只要url里面有个正则分组，它就会作为一个位置参数，就要多传一个参数到对应的视图函数里面  
 re\_path(r'^articles/([0-9]{4})/$', views.year\_archive),  
 re\_path(r'^articles/([0-9]{4})/([1-9]{1}|1[0-2]{1})/$', views.month\_archive),  
 # re\_path(r'^articles/([0-9]{4})/([1-9]{1}|1[0-2]{1})/([0-9]+)/$', views.article\_detail),  
]

1.首先导入一个 re\_path 。

path是django 2.\* 版本以上的，但是，django兼容1.\* 版本的 re\_path

2.上面的re\_path 中的配置项是基于正则表达式的匹配规则。

一个路径（或者叫做匹配项）对应一个视图函数，要精确匹配 那就要让正则前后闭合“^” “$” 。并且，匹配规则为从上到下匹配，当匹配到一个re\_path的路径以后，就不在往下匹配。

3.要从URL中捕获一个值，只需要在它周围放置一个圆括号。

4.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。

5. 每个正则表达式前面的'r' 是可选的但是建议加上。它告诉Python 这个字符串是“原始的” —— 字符串中任何字符都不应该转义

## 2.路由控制之有名分组

urls.py

*from* django.contrib *import* admin  
*from* django.urls *import* path,re\_path  
  
*from* app01 *import* views   
urlpatterns = [  
 path('admin/', admin.site.urls),  
 path('timer/',views.timer), #添加路由转发  
  
 re\_path(r'^articles/(?P<year>[0-9]{4})/(?P<month>[1-9]{1}|1[0-2]{1})/(?P<day>[1-9]{1}|1[0-9]{1}|2[0-9]{1}|3[0-1]{1})/$', views.article\_detail),  
]

views.py

*from* django.shortcuts *import* render,HttpResponse

*def* article\_detail(*request*,*month*,*day*,*year*):  
 *return* HttpResponse(*year*+'-'+*month*+'-'+*day*)

在Python 正则表达式中，命名正则表达式组的语法是(?P<name>pattern)，其中name 是组的名称，pattern 是要匹配的模式。做了分组以后，在views.py 里面的变量位置可以随便改变。

## 3.路由控制之分发

如果一个项目下有多个应用，那把所有的应用的url都放在主项目下的urls.py 里面，那就非常混乱，代码耦合太高。因此，把各自的应用的URL放在各自的应用目录下面。

1.首先在应用下面创建一个urls.py(名字随便取) .



2.在主项目下导入一个include方法，并添加一条路由规则，此时include导入的文件就是，对应的应用名以及1所创建的路由文件。

*from* django.contrib *import* admin  
*from* django.urls *import* path,re\_path,**include**  
*from* app01 *import* views #导入对应应用的视图  
urlpatterns = [  
 path('admin/', admin.site.urls),  
 path('timer/',views.timer), #添加路由转发  
 # 分发  
 # re\_path(r"app01/",include("app01.urls"))  
 re\_path(r"^",include("app01.urls"))  
]

3.把应用的URL写在自己应用的app01 下的 urls.py 里面

*from* django.contrib *import* admin  
*from* django.urls *import* path,re\_path  
*from* app01 *import* views #导入对应应用的视图  
urlpatterns = [  
 re\_path(r'^articles/([0-9]{4})/$', views.year\_archive),  
 re\_path(r'^articles/([0-9]{4})/([1-9]{1}|1[0-2]{1})/$', views.month\_archive),  
 re\_path(r'^articles/(?P<year>[0-9]{4})/(?P<month>[1-9]{1}|1[0-2]{1})/(?P<day>[1-9]{1}|1[0-9]{1}|2[0-9]{1}|3[0-1]{1})/$', views.article\_detail),  
]

4.在浏览器里面访问就可以得到自己的页面

views.py的视图：

*def* article\_detail(*request*,*month*,*day*,*year*):  
 *return* HttpResponse(*year*+'-'+*month*+'-'+*day*)

URL访问页面在控制器里面的特别说明：

re\_path(r"app01/",include("app01.urls")),

如果在主项目下配置的项目URL，那么就要在url路径当中加app01（项目名称）：

例如：<http://127.0.0.1:8000/app01/articles/2013/4/12/>

re\_path(r"^",include("app01.urls")),

如果在主项目下的urls.py里面配置成这样的，那在url中就直接使用规则去访问即可：

例如：http://127.0.0.1:8000/articles/2013/4/12/

## 4.路由控制之登陆验证示例

1.在主项目的urls.py中加入一个login的路由：

urlpatterns = [  
 path('admin/', admin.site.urls),  
 path('timer/',views.timer), #添加路由转发  
 path('login/',views.login)  
]

意思就是：当URL 后面是 login 的时候跳转到指定的视图 login 里面去进行逻辑处理以及获取页面

2.创建一个login.html的页面

<!DOCTYPE html>  
<**html** lang=**"en"**>  
<**head**>  
 <**meta** charset=**"UTF-8"**>  
 <**title**>**login.html**</**title**>  
</**head**>  
<**body**>  
<**form** action=**"http://127.0.0.1:8000/login/"** method=**"post"**>  
 **用户名：** <**input** type=**"text"** name=**"user"**>  
 **密码：** <**input** type=**"password"** name=**"pwd"**>  
 <**input** type=**"submit"**>  
</**form**>  
</**body**>  
</**html**>

3.对用户的请求做逻辑处理

特别提示：用户请求包含GET请求和POST请求。其中GET请求是用户访问login 的URL以后，服务器返回一个login页面，此时就完成了GET请求。然后，用户输入了用户名密码以后，点击提交，此时就是一个POST请求。因此：

GET请求：用户拿到login.html 页面

POST请求：用户发送密码信息给服务器端，然后服务器端进行逻辑处理（校验用户名密码）

4.views.py里面的逻辑处理等

*def* login(*request*):  
 # get请求直接拿到 login.html  
 # post 请求直接验证 用户信息  
 print("methon:",*request*.method)  
 *if request*.method == "GET":  
 print("GET:", *request*.GET)  
 *return* render(*request*,"login.html")  
 *elif request*.method == "POST":  
 print("POST:", *request*.POST) # POST: <QueryDict: {'user': ['ryan'], 'pwd': ['1234']}>  
 user = *request*.POST.get("user")  
 pwd = *request*.POST.get("pwd")  
 *if* user == 'ryan' *and* pwd == '123':  
 *return* HttpResponse('登陆成功')  
 *else*:  
 *return* HttpResponse('登陆失败')

# 9.流程控制之反向解析

在使用Django项目时，一个常见的需求是获取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）。人们强力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模块标签。
* 在python中：使用from django.urls import reverse 的reverse() 函数

## 1.html中使用反向解析

在主项目的urls.py中配置一个login的路由：

urlpatterns = [  
 path('admin/', admin.site.urls),  
 path('timer/',views.timer), #添加路由转发  
 path('login.html/',views.login,name="Login"),  
]

其中，name=“Login”是 login.html/ 的别名；

在对应项目中的 views.py 中的逻辑处理：

*def* login(*request*):  
 *if request*.method == "GET":  
 *return* render(*request*,"login.html")  
 *elif request*.method == "POST":  
 user = *request*.POST.get("user")  
 pwd = *request*.POST.get("pwd")  
 *if* user == 'ryan' *and* pwd == '123':  
 *return* HttpResponse('登陆成功')  
 *else*:  
 *return* HttpResponse('登陆失败')

在html页面 用url模块标签去导入对应的别名：

<!DOCTYPE html>  
<**html** lang=**"en"**>  
<**head**>  
 <**meta** charset=**"UTF-8"**>  
 <**title**>**login.html**</**title**>  
</**head**>  
<**body**>  
<**form** action={% **url** 'Login' %} **method="post"**>  
 **用户名：** <**input** type=**"text"** name=**"user"**>  
 **密码：** <**input** type=**"password"** name=**"pwd"**>  
 <**input** type=**"submit"**>  
</**form**>  
</**body**>  
</**html**>

注意：其中的 {% url 'Login' %} 这个是url模块去调用一个在Django 项目组中 urls.py 下面的别名。



然后现在来了需求，说 不在使用 login.html 去显示登陆页，如果不用 url模块去获取地址，那就会去修改html标签中 form的action的地址，这样就导致代码修改困难，地址不正确等。所以，现在要使用/login/去访问登陆页面，只需要修改urls.py中的路由即可；

urlpatterns = [  
 path('admin/', admin.site.urls),  
 path('timer/',views.timer), #添加路由转发  
 path('login/',views.login,name="Login"),  
]



## 2.在python代码中使用反向解析

在主项目中有一个导入的 app01 的web应用：

urlpatterns = [  
 path('admin/', admin.site.urls),  
 path('timer/',views.timer), #添加路由转发  
 path('login/',views.login,name="Login"),  
 path('app01/',include('app01.urls'))  
]

include 就是导入app01应用的路由转发，主要是为了让结构代码看上去清晰。

APP01 的urls.py

*from* django.urls *import* path,re\_path  
*from* app01 *import* views #导入对应应用的视图  
urlpatterns = [  
 re\_path(r'^articles/2003/$', views.special\_case\_2003,name='s\_c\_2003'),  
 re\_path(r'^articles/([0-9]{4})/$',views.year\_archive,name='y\_a'),  
 re\_path(r'^articles/([0-9]{4})/([1-9]{1}|1[0-2]{1})/$', views.month\_archive),  
 re\_path(r'^articles/(?P<year>[0-9]{4})/(?P<month>[1-9]{1}|1[0-2]{1})/(?P<day>[1-9]{1}|1[0-9]{1}|2[0-9]{1}|3[0-1]{1})/$', views.article\_detail),  
]

在app01 的views.py中导入 反向解析的函数模块：

*from* django.urls *import* reverse  
*def* special\_case\_2003(*request*):  
 # HttpResponse 响应对象  
 url = reverse('s\_c\_2003')  
 print(url) # /app01/articles/2003/  
 *return* HttpResponse('special\_case\_2003')  
  
*def* year\_archive(*request*,*year*):  
 url = reverse('y\_a',args=(*year*,))  
 print(url) # /app01/articles/2012/  
 *return* HttpResponse('<p style="color: blue">%s year</p>'%*year*)  
  
*def* month\_archive(*request*,*month*,*year*):  
 url = reverse('y\_a',args=(1007,))  
 print(url) #/app01/articles/1007/  
 *return* HttpResponse('<p style="color: blue">%s year %s month</p>'%(*year*,*month*))

在python中反向解析的用法总结：

1、如果在路由转发中的URL不带不确定项（也就是精确匹配），那么在reverse进行反向解析的时候就不需要使用参数，直接使用别名即可。

2、如果在路由转发中的URL带了不确定项,就像上列中的 r'^articles/([0-9]{4})/$.那么在反向解析的时候，就要加入一个参数，不然会报错。

3、可以在views.py 中的任何地方使用别名获取到对应的地址。

# 10.路由器控制之名称空间

命名空间（英语：Namespace）是表示标识符的可见范围。一个标识符可在多个命名空间中定义，它在不同命名空间中的含义是互不相干的。这样，在一个新的命名空间中可定义任何标识符，它们不会与任何已有的标识符发生冲突，因为已有的定义都处于其它命名空间中。

由于name没有作用域，Django在反解URL时，会在项目全局顺序搜索，当查找到第一个name指定URL时，立即返回

我们在开发项目时，会经常使用name属性反解出URL，当不小心在不同的app的urls中定义相同的name时，可能会导致URL反解错误，为了避免这种事情发生，引入了命名空间。

在urls中新增加一个namespace：

urlpatterns = [  
 path('admin/', admin.site.urls),  
 path('timer/',views.timer), #添加路由转发  
 path('login/',views.login,name="Login"),  
 path('app01/',include(('app01.urls',"app01"))),  
 path('app02/',include(('app02.urls',"app02"))),  
]

include(('app01.urls',"app01")) 。此时的app01 就是设置的namespace。app02 也一样。

在app01.urls.py 中

urlpatterns = [  
 re\_path("index/",views.index,name="index"),

]

在app01.views.py中

*from* django.urls *import* reverse

*def* index(*request*):  
 *return* HttpResponse(reverse("app01:index"))

在app02.urls.py中

*from* django.urls *import* path,re\_path  
*from* app02 *import* views #导入对应应用的视图  
urlpatterns = [  
 re\_path("index/",views.index,name="index")  
]

在app02.views.py 中

*from* django.shortcuts *import* render,HttpResponse  
*from* django.urls *import* reverse  
# Create your views here.  
*def* index(*request*):  
 *return* HttpResponse(reverse("app02:index"))

网站访问结果：



如果不设置namespace ，那么 访问到的结果，就会是一个相同的结果。

# 11.django2.0版本的path

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),

]

考虑下面这样两个问题：

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。

<https://docs.djangoproject.com/en/2.0/topics/http/urls/#example> 官方文档 URLS

## 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.注册自定义转化器

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：在URL里面匹配 月份 （1-9 10 11 12）

在项目中的任意一个地方创建一个自定义的匹配规则（转化器），此例是在app01 下面创建的

1.创建一个 urlconverter.py

*class* MonConvert(object):  
 regex = "[1-9]{1}|1[0-2]{1}"  
 *def* to\_python(self,*value*):  
 *return* int(*value*)  
 *def* to\_url(self,*value*): #反向解析  
 *return* '%04d'%*value*

2.定义视图函数 views.py中

*def* path\_month(*request*,*month*):  
 *return* HttpResponse("month:%d"%*month*)

3.在项目的urls中进行导入-注册-使用

*from* django.contrib *import* admin  
*from* django.urls *import* path,re\_path,include,register\_converter  
*from* app01 *import* views #导入对应应用的视图  
# 导入  
*from* app01.urlconverter *import* MonConvert  
# 注册  
register\_converter(MonConvert,"mm")  
urlpatterns = [  
 path('admin/', admin.site.urls),  
 # 匹配  
 path('month/<mm:month>',views.path\_month)  
]

此时的mm就是注册的一个转化器，month就相当于分组中的名称。

4.网页访问



# 12.django视图层

## 1.视图函数

一个视图函数，简称视图，是一个简单的Python 函数，它接受Web请求并且返回Web响应。响应可以是一张网页的HTML内容，一个重定向，一个404错误，一个XML文档，或者一张图片. . . 是任何东西都可以。无论视图本身包含什么逻辑，都要返回响应。代码写在哪里也无所谓，只要它在你的Python目录下面。除此之外没有更多的要求了——可以说“没有什么神奇的地方”。为了将代码放在某处，约定是将视图放置在项目或应用程序目录中的名为views.py的文件中。

下面是一个返回当前日期和事件作为HTML文档的视图：

*from* django.shortcuts *import* render, HttpResponse, HttpResponseRedirect, redirect  
*import* datetime  
  
*def* current\_datetime(request):  
 now = datetime.datetime.now()  
 html = "<html><body>It is now %s.</body></html>" % now  
 *return* HttpResponse(html)

代码解释：

1.从django.shortcuts模块导入了HTTPResponse 类，以及Python的datetime库

2. 定义了current\_datetime函数。它就是视图函数。每个视图函数都使用HttpRequest对象作为第一个参数，并且通常称之为request。

3. 视图会返回一个HttpResponse对象，其中包含生成的响应。每个视图函数都负责返回一个HttpResponse对象。

视图层，熟练掌握两个对象即可：请求对象(request)和响应对象(HttpResponse)

## 2.HTTPRequest对象

来源：http://www.cnblogs.com/yuanchenqi/articles/8876856.html

django将请求报文中的请求行、首部信息、内容主体封装成 HttpRequest 类中的属性。 除了特殊说明的之外，其他均为只读的。

例如：HttpRequest.GET HttpRequest.POST HttpRequest.body HttpRequest.path HttpRequest.method 等等，使用的时候查询即可。

request常用方法：

1.HttpRequest.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.HttpRequest.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。

*def* index(*request*):  
 # 请求对象  
 print("method:",*request*.method) # GET请求 地址栏发请求默认都是GET请求  
 print("request.GET >>",*request*.GET) # 存储所有GET请求信息  
 print("GET name value: ",*request*.GET.get("name"))  
 print("GET age value: ",*request*.GET.get("age"))  
  
 print("request.POST >>",*request*.POST) # 存储所有POST请求信息  
 print("POST name value: ",*request*.POST.get("name"))  
 print("POST age value: ",*request*.POST.get("age"))  
 '''  
 URL包含: 协议://IP:PORT/路径?参数(GET请求的数据)  
 '''  
 print("request directory:",*request*.path) # 获取请求路径；/index/  
 print("get\_full\_path:",*request*.get\_full\_path()) # 获取请求路径 /index/?name=ryan&age=19  
 *return* HttpResponse("OK")

## 3.HttpResponse对象

响应对象主要有三种形式：

* HttpResponse()
* render()
* redirect()

HttpResponse()括号内直接跟一个具体的字符串作为响应体，比较直接很简单，所以这里主要介绍render

1. render(request, template\_name[, context]）

*def* index(*request*):  
 *import* time  
 ctime = time.time()  
 dic = {"ctime":ctime}  
 *return* render(*request*,'index.html',dic) # index.html:模板文件  
 # render 先拿到模板文件，渲染成模板文件，转换成HTML文件以后才发送给用户。

结合一个给定的模板和一个给定的上下文字典，并返回一个渲染后的HttPResponse对象

参数：

request : 用于生成响应的请求对象

template\_name ：要使用的模板的完整名称，可选的参数。

context：添加到模板上下文的一个字典。默认是一个空字典。如果字典中的某个值是可调用的，视图将在渲染模板之前调用它。

render方法就是将一个模板页面中的模板语法进行渲染，最终渲染成一个html页面作为响应体。然后把响应体返回给浏览器。

# 13.模板语法之变量

在 Django 模板中遍历复杂数据结构的关键是句点字符, 语法: {{var\_name}}

在views.py 中：

*def* index(*request*):  
 name = 'ryan'  
 i = 10  
 l = [11,22,33]  
 info = {"name":"ryan","age":22}  
 b = *True  
 class* Person(object):  
 *def* \_\_init\_\_(self,*name*,*age*):  
 self.name=*name* self.age = *age* ryan = Person("ryan",22)  
 cherry = Person("cherry",18)  
 person\_list = [ryan,cherry]  
 *return* render(*request*,'index.html',locals())

把index 函数中的变量传到html模板当中去，表里包括，数字、字符串、列表、字典以及object对象等

其中最重要的就是 **locals()**它表示把所有变量都传到html中。就不用把所有的变量都放到一个大的字典里面去。

在模板html中去获取变量的方法，index.html

<!DOCTYPE html>  
<**html** lang=**"en"**>  
<**head**>  
 <**meta** charset=**"UTF-8"**>  
 <**title**>**index**</**title**>  
</**head**>  
<**body**>  
 <**h3**>**模板语法**</**h3**>  
 <**p**>{{ name }}</**p**>  
 <**p**>{{ i }}</**p**>  
 <**p**>{{ l }}</**p**>  
 <**p**>{{ info }}</**p**>  
 <**p**>{{ ryan }}</**p**>  
 <**p**>{{ person\_list }}</**p**>  
 <**p**>{{ b }}</**p**>  
 <**hr**>  
 **深度查询** <**p**>{{ l.1 }}</**p**>  
 <**p**>{{ info.name }}</**p**>  
 <**p**>{{ ryan.age }}</**p**>  
 <**p**>{{ person\_list.1.age }}</**p**>  
</**body**>  
</**html**>

直接使用变量名称获取的时候，是获取整个object的值。

如果在html里要获取列表里面的值，使用“列表名.索引”

如果在html里要获取字典的值，使用 “字典名.key”

如果在html里要获取对象的属性，使用 “对象.属性”

注意：句点符也可以用来引用对象的方法(无参数方法):

<**p**>{{ person\_list.1.name.upper }}</**p**>

# 14.模板之过滤器

语法： {{ obj|filter\_name:param }}

## 1.date

在views.py 中

*def* index(*request*):  
 *import* datetime  
 now = datetime.datetime.now() # 2018-06-21 14:02:42.585698  
 *return* render(*request*,'index.html',locals())

本来now 等于”2018-06-21 14:02:42.585698”，但是在网页中显示为“June 21, 2018, 2:02 p.m.”。这是经过渲染以后造成的。因此，要在html模板中进行过滤：

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

然后就会显示：2018-06-21

## 2.defalut：“default string”

person\_list = []

html标签中：

<**p**>{{ person\_list|default:"数据为空" }}</**p**>

网页显示数据为空，如果person\_list是一个未定义的变量，也可以这样使用。

## 3.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）。

{{ value|filesizeformat }}

最小单位是byte，依次网上，KB，MB， GB

## 4.truncatechars && truncatewords

truncatechars是截断字符串。

truncatewords 是一单词截断。

## 5.link

在views.py中

link = "<a href=''>click</a>"

在Html中

<**p**>{{ link|safe }}</**p**>

如果不使用safe，会把link中的字符串，按照字符串处理，特殊符号按照特殊符号处理，页面显示的效果为：<a href=''>click</a>。

# 15.模板之标签

在views.py中的变量定义

*class* Person(object):  
 *def* \_\_init\_\_(self,*name*,*age*):  
 self.name=*name* self.age = *age*ryan = Person("ryan",22)  
cherry = Person("cherry",18)

*def* index(*request*):

l = [11,22,33]

info = {"name":"ryan","age":22}

person\_list = [ryan,cherry]

## 1.for标签

for标签循环一个列表和字典：

{% **for** i in l %}  
 <**p**>{{ i }}</**p**>  
{% **endfor** %}  
  
{% **for** i in info %}  
 <**p**>{{ i }}</**p**>  
{% **endfor** %}

for标签循环一个object 列表

{% **for** person in person\_list %}  
<**p**>**序号:**{{ forloop.counter0 }} **姓名：**{{ person.name }}**年龄：**{{ person.age }}</**p**>  
{% **endfor** %}

forloop.counter 表示序号，从1开始计数

forloop.counter0 ，表示从0开始计算

for标签循环一个对象，如果对象为空，那就显示别的内容

{% **for** person in person\_list\_undefine %}  
 <**p**>**序号:**{{ forloop.counter0 }} **姓名：**{{ person.name }}**年龄：**{{ person.age }}</**p**>  
{% **empty** %}  
 <**p**>**undefine or list is empty**</**p**>  
{% **endfor** %}

如果循环的列表为空，那么显示 undefne or list is empty

## 2.if标签

{% **if** user %}  
 <**p**>  
 <**a** href=**"#"**>**hi** {{ user }}</**a**>  
 </**p**>  
{% **elif** user == None %}  
 <**p**>**没有登录**</**p**>  
{% **endif** %}

## 3.with标签 -取别名

{% **with** person\_list.1.name as n %}  
 <**p**>**姓名：**{{ n }}</**p**>  
{% **endwith** %}

n就表示 person\_list.1.name

## 4.csrf\_token标签

这个标签用于跨站请求伪造保护

当用户访问login页面的时候，如果不使用django自带的安全检查中间件 csrf 就需要注释掉setting里面的 'django.middleware.csrf.CsrfViewMiddleware',

如果要使用django自带的安全模块，那就要在form表单在提交post请求的时候加上 csrf\_token标签。

views.py 的代码：

*def* login(*request*):  
 *if request*.method == "POST":  
 *return* render(*request*,'welcome.html')  
 *return* render(*request*,"login.html")

在html中的代码：

<!DOCTYPE html>  
<**html** lang=**"en"**>  
<**head**>  
 <**meta** charset=**"UTF-8"**>  
 <**title**>**login**</**title**>  
</**head**>  
<**body**>  
<**form** action=**""** method=**"post"**>  
  **{% csrf\_token %}**  
 <**input** type=**"text"** name=**"user"**>  
 <**input** type=**"submit"**>  
</**form**>  
</**body**>  
</**html**>

在点击submit的时候，就是发送的一个POST请求，然后视图拿到这个请求，就会然后跳转。

实际上，在请求走到views.py视图时，视图去渲染html模板，然后会把 csrf\_token 生成一个token的标签，标签里面的value值就是 django自动生成的值。value值每次访问也是不一样的



# 16.自定义过滤器和标签

1.查看setting的INSTALLED\_APPS配置里面是否有当前的APP，不然django无法找到自定义的过滤器或者标签。在setting里面可以查看到如下内容：

'app01.apps.App01Config',

更新django1.11.3后，django创建应用的时候已经帮你注册了xx.apps.XXConfig了。当然，没有引入，那就使用 “app01”添加到INSTALLED\_APPS中即可。

2．在app中创建templatetags模块(模块名只能是templatetags)

3.创建任意 .py 文件，如： my\_tags\_filter.py

*from* django *import* template  
# register 名称不可修改  
register=template.Library()  
# 自定义过滤器只能有两个参数  
@register.filter  
*def* multi\_filter(*x*,*y*):  
 *return x*\**y*# 自定义标签可以有多个参数  
@register.simple\_tag  
*def* multi\_tag(*x*,*y*):  
 *return x*\**y*

注意：

1.register 只能是这个名称

2.自定义过滤器要在函数前面加上装饰器 @register.filter

3.自定义标签要在函数前面加上装饰器 @register.simple\_tag

4.如果要在html模板文件当中使用自定义的过滤器或者标签，那么在模板文件html中要引入（3）所创建的py文件。

{% **load** my\_tags\_filter %}

5.使用自定义的过滤器和标签

5.1使用自定义过滤器

<**p**>{{ i | multi\_filter:20}}</**p**>

这里定义的过滤器是乘法运算，因此在网页中显示的是 200, 因为在views.py 中定义的i=10，表示 10乘以20 得到200；

5.2使用自定义标签

<**p**>{% **multi\_tag** 7 9 %}</**p**>

这里定义的标签也是乘法运算，因此，在网页中显示的内容是 63

6.自定义标签和过滤器比较。

1.自定义的过滤器只能有两个参数，而自定义的标签可以有多个参数。

2.自定义的过滤器可以使用 if 等标签语句，而自定义的标签却不可以。

{% **if** i|multi\_filter:18 > 100 %}  
 <**p**>**100**</**p**>  
{% **else** %}  
 <**p**>{{ i }}</**p**>  
{% **endif** %}

i=10 过滤器后面是18 所以 得到 180 大于100 因此网页显示100.

# 17.模板继承（extend）

Django模板引擎中最强大也是最复杂的部分就是模板继承了。模板继承可以让您创建一个进本的“骨架”模板，它包含您站点中的全部元素，并且可以定义能够被自子模板覆盖的blocks。

在urls控制器中：

urlpatterns = [  
 path('admin/', admin.site.urls),  
 path('login/',views.login),  
 path('orders/',views.orders),  
 re\_path(r"index/",views.index,name="index"),  
 re\_path(r"^$",views.index,name="index"),  
 re\_path(r"welcome/",views.welcome,name="welcome"),  
]

base.html 模板

<!DOCTYPE html>  
<**html** lang=**"en"**>  
<**head**>  
 <**meta** charset=**"UTF-8"**>  
 {% **block** title %}

<**title**>**base模板**</**title**>  
 {% **endblock** %}  
 <**link** rel=**"stylesheet"** href=**"https://cdn.bootcss.com/bootstrap/3.3.7/css/bootstrap.min.css"** integrity=**"sha384-BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u"** crossorigin=**"anonymous"**>  
 <**style** type=**"text/css"**>  
 \*{*margin*: 0;*padding*: 0}  
 .header{*width*: 100%;*height*: 50**px**;*background-color*: **cornflowerblue**;}  
 </**style**>  
</**head**>  
<**body**>  
<**div** class=**"header"**></**div**>  
 <**div** class=**"container"**>  
 <**div** class=**"row"**>  
 <**div** class=**"col-md-3"**>  
 {% **include** 'advertise.html' %}  
 </**div**>  
 <**div** class=**"col-md-9"**>  
 {% **block** con %}

<**a** href=**"#"**>**我是base.html 设置的content**</**a**>  
 {% **endblock** %}  
 </**div**>  
 </**div**>  
 </**div**>  
</**body**>  
</**html**>

在代码中可以看到两个部分，一个是 include，一个是 block部分。

include的意思是，引入advertise.html的全部代码到该html文件中。

block 中有一个title的模板，此时表示，在别的html模板使用base模板的时候，如果没有自定一个title的信息，那就默认使用base中的title的信息。

block 标签定义了三个可以被子模版内容填充的block。 block 告诉模版引擎： 子模版可能会覆盖掉模版中的这些位置。

例如有一个订单页面（ordrs.html）要使用base的模板信息：

{% **extends** 'base.html' %}  
  
{% **block** title %}  
 <**title**>**订单页面**</**title**>  
{% **endblock** %}  
  
{% **block** con %}  
 {{ block.super }}  
 <**h3**>**订单**</**h3**>  
 <**a** href=**"#"** style=**"***display*: **block"**>**我是a标签**</**a**>  
 {{ block.super }}  
{% **endblock** %}

1.extends标签是整理的关键。它告诉模板引擎，这个模板“继承”了另一个模板。当模块系统处理这个模板时，首先它将定位父模板。说就是在第一行就必须引入base.html的信息，不然，后面的block就不能够解析出来。

2.在子模板orders.html 中就使用block进行填充base.html 中的block。

这种方式使代码得到最大程度的复用，并且使得添加内容到共享的内容区域更加简单，如果部分范围内的导航。

使用继承的提示:

* 如果在模板中使用{% extends %}标签，它必须是模板中的第一个标签。其他的任何情况下，模板继承都将无法工作。
* 在base模板中设置越多的 {% block %}标签越好。请记住，子模板补习定义全部父模板的blocks，所以，你可以在大多数block中填充合理的默认内容，然后，之定义你需要的哪一个。多一点钩子总比少一点好。
* If you need to get the content of the block from the parent template, the {{ block.super }} variable will do the trick. This is useful if you want to add to the contents of a parent block instead of completely overriding it. Data inserted using {{ block.super }} will not be automatically escaped (see the next section), since it was already escaped, if necessary, in the parent template.

意思使用: {{ block.super }}可以展示父模板中的默认ner

* 为了更好的可读性，可以给 {% endblock %}标签一个名字。例如：
* {% **block** title %}  
   <**title**>**订单页面**</**title**>  
  {% **endblock** title%}

endblock后面的title就是endblock的名字。

* 不能再一个模板中定义多个相同的名字的block标签

# 18.Django模型层(重点)

## 1.ORM简介

* MVC或者MVC框架中包括一个重要的部分，就是ORM，它实现了数据模型与数据库的解耦，即数据模型的设计不需要依赖于特定的数据库，通过简单的配置就可以轻松更换数据库，这极大的减轻了开发人员的工作量，不需要面对因数据库变更而导致的无效劳动
* ORM是“对象-关系-映射”的简称。



## 2.单表操作-创建表

创建一个orm的项目，orm下面有一个app01的应用。

在app01 下的modules.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*)  
 title=models.CharField(max\_length=32)  
 pub\_date=models.DateField()  
 price=models.DecimalField(max\_digits=8,decimal\_places=2)  
 publish=models.CharField(max\_length=32)

### 1.创建表的字段和参数

1) 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.

2)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.

3) 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.

4) DecimalField(max\_digits=8,decimal\_places=2)：表示最长8位，小数点后有2位数字。

更多参数：http://www.cnblogs.com/yuanchenqi/articles/8933283.html

### 2.setting设置

若想将模型转为mysql数据库中的表，需要在setting中配置。

DATABASES = {  
 'default': {  
 'ENGINE': 'django.db.backends.mysql',  
 'NAME':'orm',# 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':'123456',# 连接数据库的密码  
 'HOST':'101.132.161.180',# 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，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：

*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INSTALED\_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',  
 'app01.apps.App01Config',  
]

注意3：如果报错如下：

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：

mac系统：

查看安装路径，import sys print(sys.path)

/usr/local/lib/python3.6/site-packages/django/db/backends/mysql/base.py 下的

if version < (1, 3, 3):

raise ImproperlyConfigured("mysqlclient 1.3.3 or newer is required; you have %s" % Database.\_\_version\_\_)

注释掉，就可以了。

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',  
 },  
 }  
}

## 3.添加表记录

在views.py中：

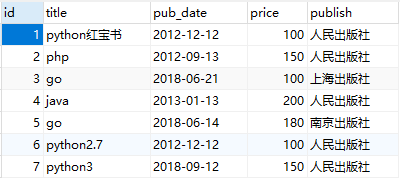
*from* django.shortcuts *import* render,HttpResponse  
# Create your views here.  
*from* app01.models *import* Book  
*def* index(*request*):  
 #添加表记录  
 # 方式一：  
 book\_obj1 = Book(id=1,title="python红宝书",price=100,pub\_date="2012-12-12",publish="人民出版社")  
 book\_obj1.save()

#方式二：create返回值就是当前生成的对象记录  
 book\_obj2 = Book.objects.create(title="java",price=200,pub\_date="2013-01-13",publish="人民出版社")  
 print(book\_obj2.title)  
 print(book\_obj2.price)  
 print(book\_obj2.publish)  
  
 *return* HttpResponse("OK")

注意：方式1 ，必须要保存。方式2，可以直接调用对象的属性。

## 4.查询表记录API

表结构与数据：



在使用方法之前一定要注意方法的 调用者是什么，返回值是什么

### 1.all() 方法

all方法返回一个QuerySet对象。QuerySet对象 类似一个列表放了一个个object对象

book\_list = Book.objects.all()  
print(book\_list)  
# 结果：<QuerySet [<Book: python红宝书>, <Book: php>, <Book: go>, <Book: java>, <Book: go>, <Book: python2.7>, <Book: python3>]>  
*for* obj *in* book\_list:  
 print(obj.title,obj.price)  
  
print(book\_list[1].title,book\_list[1].price) #php 150.00

可以直接获取整个QuerySet对象，可以用for循环，也可以用“.属性”的方式

### 2.first() && last() 方法

调用者是：QuerySet对象，返回值是：model对象

print(Book.objects.all().first()) # 结果：python红宝书 。相当于 Book.objects.all()[0]  
print(Book.objects.all().last()) # 结果：python3

### 3.filter() 方法

返回值是 QuerySet对象

print(Book.objects.filter(title='php')) # <QuerySet [<Book: php>]>  
print(Book.objects.filter(price=100)) # <QuerySet [<Book: python红宝书>, <Book: go>, <Book: python2.7>]>  
book\_obj = Book.objects.filter(price=100).first()  
print(book\_obj) # python红宝书  
print(Book.objects.filter(title='go',price=150)) # <QuerySet []> 过滤多个条件，逗号表示 and

### 4.get() 方法

get方法有且只有一个查询结果时才有意义。返回值：model对象。

get方法没有返回值会报错，超过一个返回值也报错。

book\_obj = Book.objects.get(title='java')  
print(book\_obj.price) #200.00

### 5.exclude() 方法

exclude表示包含了与所给筛选条件不匹配的对象（排除）。返回值是：QuerySet对象

print(Book.objects.exclude(price\_\_in=[100,150,200])) # <QuerySet [<Book: go>]>

### 6.order\_by() 方法

order\_by  对查询结果排序，默认是升序。调用者:QuerySet对象，返回值：QuerySet对象。

如果要使用降序，在字段前面加上 “-”

ret1 = Book.objects.all().order\_by("-id") #按照 ID 降序排列  
ret2 = Book.objects.all().order\_by("-price") # 价格排序  
ret3 = Book.objects.all().order\_by("price","id") # 按照价格,id排序，(先排序价格，如果相同，在排序ID)

### 7.reverse() 方法

对查询结果反向排序

### 8.count() 方法

count 是返回数据库中匹配查询QuerySet对象的数量。调用者是：QuerySet对象。返回值是：int

### 9.exists() 方法

exists 如果QuerySet包含数据，就返回True，否则返回False

ret1 = Book.objects.exclude(title='go').exists()  
ret2 = Book.objects.all().exists()  
print(ret1) # True  
print(ret2) # True

### 10.values() 方法 <重要>

values 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

'''values：工作原理  
temp = []  
for obj in Book.objects.all():  
 temp.append({  
 "price"=obj.price  
 "title"=obj.title  
 })  
return temp  
'''

ret = Book.objects.all().filter(price=150).values("price","title")  
print(ret)  
#结果： <QuerySet [{'price': Decimal('150.00'), 'title': 'php'}, {'price': Decimal('150.00'), 'title': 'python3'}]>

print(ret[0].get('price')) #150.00

### 11.values\_list() 方法

values\_list 它与values非常相似，它返回一个元祖序列，values返回的是一个字典序列。

ret = Book.objects.all().filter(price=150).values\_list("price","title")  
print(ret)  
# 结果：<QuerySet [(Decimal('150.00'), 'php'), (Decimal('150.00'), 'python3')]>

### 12.distinct() 方法

distinct 从返回结果中提出重复记录

ret = Book.objects.all().values('price').distinct()  
print(ret) #<QuerySet [{'price': Decimal('100.00')}, {'price': Decimal('150.00')}, {'price': Decimal('200.00')}, {'price': Decimal('180.00')}]>

## 5.查询表记录之模糊查询

### 1.大于&小于

返回一个QuerySet对象，例如：查询价格大于150小于200的书籍

ret = Book.objects.filter(price\_\_gt=150,price\_\_lt=200)   
print(ret) #<QuerySet [<Book: go>]>

### 2.查询以什么开头什么结尾的书籍

startswith && endswith

ret = Book.objects.filter(title\_\_startswith='py')   
ret2 = Book.objects.filter(title\_\_endswith='va')   
print(ret) # <QuerySet [<Book: python红宝书>, <Book: python2.7>, <Book: python3>]>  
print(ret2) # <QuerySet [<Book: java>]>

### 3.查询包含某字符串的字段

查询字段中包含py的数据

print(Book.objects.filter(title\_\_contains='py')) #<QuerySet [<Book: python红宝书>, <Book: python2.7>, <Book: python3>]>

### 4.取在范围里面的值（单点）

取出价格等于 150 和200 的书籍

print(Book.objects.filter(price\_\_in=[100,200])) # <QuerySet [<Book: php>, <Book: java>, <Book: python3>]>

### 5.取区间范围里面的值

取出价格等于 150到200之间的书籍

print(Book.objects.filter(price\_\_range=[150,200])) #<QuerySet [<Book: php>, <Book: java>, <Book: go>, <Book: python3>]>

### 6.查询日期

\_\_year,\_\_month,\_\_day 年月日。

取出出版日期为2012年12月份的书籍：

print(Book.objects.filter(pub\_date\_\_year=2012,pub\_date\_\_month=12)) #<QuerySet [<Book: python红宝书>, <Book: python2.7>]>

## 6.单表删除表记录和修改表记录

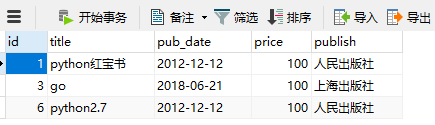
### 1.delete()方法删除表记录

delete 调用者：既可以是QuerySet对象，也可以是model对象

delete调用QuerySet对象进行删除

ret = Book.objects.filter(price\_\_range=[150,200]).delete()  
print(ret) #(4, {'app01.Book': 4})

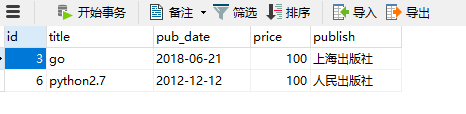
返回结果ret 表示：delete删除了4条app01.book表中的数据。可以看到下面数据表中部分数据已被删除



delete调用model对象

ret = Book.objects.filter(price=100).first().delete()  
print(ret) # (1, {'app01.Book': 1})

first方法返回的是一个model对象。可以看到下面数据表中部分数据已被删除



### 2.更新和修改（update）

update只能用QuerySet对象去调用。

根据上列剩下的数据，修改go 书籍的价格为150

ret = Book.objects.filter(title="go").update(price=150)  
print(ret,type(ret)) # 1 <class 'int'>

## 7.单表练习题

# 查询老男孩出版社出版过的价格大于200的书籍  
ret =Book.objects.filter(publish="老男孩出版社",price\_\_gt=200)  
# 查询2017年8月出版的所有以py开头的书籍名称  
ret = Book.objects.filter(title\_\_startswith='py',pub\_date\_\_year=2017,pub\_date\_\_month=8).values("title")  
# 查询价格为50, 100 或者150的所有书籍名称及其出版社名称  
Book.objects.filter(price\_\_in=[50,100,150]).values("title","publish")  
# 查询价格在100到200之间的所有书籍名称及其价格  
Book.objects.filter(price\_\_range=[100,200]).values("title","price")  
# 查询所有人民出版社出版的书籍的价格（从高到低排序，去重）  
Book.objects.filter(publish="人民出版社").values("price").distinct().order\_by("-price")

## 8.图书管理功能

实现功能：book表单，实现在页面展示书籍，增加书籍，删除书籍，修改书籍。

### 1. 创建项目和应用

创建一个bookms的项目，项目下右一个app01的应用

命令行操作：

创建项目：django-admin.py startproject bookms

创建应用：python3 manage.py startapp app01

### 2.创建数据库（MySQL）

创建一个bookdata的数据库用于存放书籍信息：

CREATE DATABASE IF NOT EXISTS bookdata DEFAULT CHARACTER SET utf8 COLLATE utf8\_general\_ci;

### 3.引入MySQL以及创建表和字段

在app01应用下的init.py文件中写入：

*import* pymysql  
pymysql.install\_as\_MySQLdb()

在app01应用下的models.py中

*from* django.db *import* models  
# Create your models here.  
*class* Book(models.Model):  
 id=models.AutoField(primary\_key=*True*)  
 title=models.CharField(max\_length=32)  
 pub\_date=models.DateField()  
 price=models.DecimalField(max\_digits=8,decimal\_places=2)  
 publish=models.CharField(max\_length=32)

### 4.配置使用MySQL数据库及console打印数据库语句

在setting.py中，注释掉默认的sqlite的数据库信息，添加以下信息：

DATABASES = {  
 'default': {  
 'ENGINE': 'django.db.backends.mysql',  
 'NAME':'bookdata',  
 'USER':'root',  
 'PASSWORD':'123456',  
 'HOST':'101.132.161.180',  
 'PORT':3306  
 }  
}

添加console打印日志配置

LOGGING = {  
 'version': 1,  
 'disable\_existing\_loggers': *False*,  
 'handlers': {  
 'console':{  
 'level':'DEBUG',  
 'class':'logging.StreamHandler',  
 },  
 },  
 'loggers': {  
 'django.db.backends': {  
 'handlers': ['console'],  
 'propagate': *True*,  
 'level':'DEBUG',  
 },  
 }  
}

### 5.urls路由控制配置信息

*from* django.contrib *import* admin  
*from* django.urls *import* path,re\_path  
*from* app01 *import* views  
urlpatterns = [  
 path('admin/', admin.site.urls),  
 path('books/', views.books), # 管理界面  
 path('addbook/', views.addbook), # 添加书籍界面  
 re\_path(r'books/(\d+)/delete', views.delbook), # 删除书籍路由  
 re\_path(r'books/(\d+)/change', views.changebook), # 修改书籍路由  
]

### 6.页面以及视图创建思路注意点

1、界面以模板的形式去创建页面

2、界面传参方式以及获取方式

3、发送请求时的方法（post or get）。注意 a标签默认是get请求

4、页面的跳转实现。

### 7. 所有操作视图逻辑

*from* django.shortcuts *import* render,redirect  
# Create your views here.  
*from* app01.models *import* Book  
# 书籍首页  
*def* books(*request*):  
 book\_list = Book.objects.all()  
 *return* render(*request*,"books.html",locals())  
# 添加书籍页面  
*def* addbook(*request*):  
 *if request*.method == "POST":  
 titile = *request*.POST.get("title")  
 price = *request*.POST.get("price")  
 date = *request*.POST.get("date")  
 publish = *request*.POST.get("publish")  
 # 书籍信息插入到数据库  
 Book.objects.create(title=titile,price=price,pub\_date=date,publish=publish)  
 *return* redirect("/books/")  
 *return* render(*request*,"addbook.html")  
# 删除书籍信息  
*def* delbook(*request*,*id*):  
 Book.objects.filter(id=*id*).delete()  
 *return* redirect("/books/")  
# 修改书籍信息  
*def* changebook(*request*,*id*):  
 book\_obj = Book.objects.filter(id=*id*).first()  
 *if request*.method == "POST":  
 titile = *request*.POST.get("title")  
 price = *request*.POST.get("price")  
 date = *request*.POST.get("date")  
 publish = *request*.POST.get("publish")  
 # 更新数据库书籍信息  
 Book.objects.filter(id=*id*).update(title=titile,price=price,pub\_date=date,publish=publish)  
 *return* redirect('/books/')  
 *return* render(*request*,"changebook.html",locals())

注意点：

1.把所有参数传递给模板用 locals()

2. redirect("/路径/") ,此时的路径是 urls.py中的路径。

3.根据用户的请求方式（post/get）去增删改查数据。

4.request.POST.get 去获取post传递的信息

### 8.创建一个展示页面books.html

<!DOCTYPE html>  
<**html** lang=**"en"**>  
<**head**>  
 <**meta** charset=**"UTF-8"**>  
 <**title**>**书籍展示**</**title**>  
 <**link** rel=**"stylesheet"** href=**"/static/bs/css/bootstrap.css"**>  
 <**style** type=**"text/css"**>  
 .container{*margin-top*: 50**px**;}  
 .btn{*margin-top*: 10**px**;*margin-bottom*: 10**px**;}  
 .table th, .table td {*text-align*: **center**;*vertical-align*: **middle!important**;}  
 </**style**>  
</**head**>  
<**body**>  
<**div** class=**"container"**>  
<**h3**>**书籍管理**<**small**>**信息查看**</**small**></**h3**>  
<**div** class=**"row"**>  
 <**div** class=**"col-md-6 col-md-offset-2"**>  
 <**a** href=**"/addbook"** class=**"btn btn-primary pull-right active"**>**添加书籍**</**a**>  
 <**table** class=**"table table-striped table-bordered table-hover table-condensed"**>  
 <**thead**>  
 <**tr**>  
 <**th**>**ID**</**th**>  
 <**th**>**书籍名称**</**th**>  
 <**th**>**价格**</**th**>  
 <**th**>**出版日期**</**th**>  
 <**th**>**出版社**</**th**>  
 <**th**>**删除操作**</**th**>  
 <**th**>**编辑操作**</**th**>  
 </**tr**>  
 </**thead**>  
 <**tbody**>  
 {% **for** book in book\_list %}  
 <**tr**>  
 <**td**>{{ forloop.counter }}</**td**>  
 <**td**>{{ book.title }}</**td**>  
 <**td**>{{ book.price }}</**td**>  
 <**td**>{{ book.pub\_date|date:'Y-m-d' }}</**td**>  
 <**td**>{{ book.publish }}</**td**>  
 <**td**><**a** href=**"/books/**{{ book.pk }}**/delete"** class=**"btn btn-sm btn-danger active"**>**删除**</**a**></**td**>  
 <**td**><**a** href=**"/books/**{{ book.pk }}**/change"** class=**"btn btn-sm btn-info active"**>**编辑**</**a**></**td**>  
 </**tr**>  
 {% **endfor** %}  
 </**tbody**>  
 </**table**>  
 </**div**>  
</**div**>  
</**div**>  
</**body**>  
</**html**>

特别注意：在点击删除的时候，怎么知道删除那一条？因此，就要在a标签里面去获取到对应的书籍的主键ID，因为主键唯一，所以，就可以获取到对应的书籍信息 就要使用book.pk 。其中的pk就表示主键，无论主键用什么名称，都可以使用pk获取。

### 9.创建base.html的父模板用于增改页面的继承

<!DOCTYPE html>  
<**html** lang=**"en"**>  
<**head**>  
 <**meta** charset=**"UTF-8"**>  
 {% **block** title %}  
 <**title**>**基础页面**</**title**>  
 {% **endblock** %}  
 <**link** rel=**"stylesheet"** href=**"/static/bs/css/bootstrap.css"**>  
 <**style** type=**"text/css"**>  
 .container{*margin-top*: 50**px**;}  
 .btn{*margin-top*: 10**px**;}  
 </**style**>  
</**head**>  
<**body**>  
<**div** class=**"container"**>  
{% **block** content %}  
<**h3**>**基础内容**</**h3**>  
{% **endblock** %}  
 <**div** class=**"row"**>  
 <**div** class=**"col-md-6 col-md-offset-3"**>  
 <**form** action=**""** method=**"post"**>  
 {% **csrf\_token** %}  
 <**div**>  
 <**label** for=**"book—name"**>**书籍名称**</**label**>  
 {% **block** book\_title\_name %}  
 {% **endblock** %}  
 </**div**>  
 <**div**>  
 <**label** for=**"price"**>**价格**</**label**>  
 {% **block** book\_price %}  
 {% **endblock** %}  
 </**div**>  
 <**div**>  
 <**label** for=**"pub\_date"**>**出版日期**</**label**>  
 {% **block** book\_pub\_date %}  
 {% **endblock** %}  
 </**div**>  
 <**div**>  
 <**label** for=**"publish"**>**出版社**</**label**>  
 {% **block** book\_publish %}  
 {% **endblock** %}  
 </**div**>  
 <**input** type=**"submit"** class=**"btn btn-success pull-right"**>  
 </**form**>  
 </**div**>  
 </**div**>  
</**div**>  
</**body**>  
</**html**>

### 10.创建书籍页面addbook.html

{% **extends** 'base.html' %}  
{% **block** title %}  
 <**title**>**添加书籍**</**title**>  
{% **endblock** %}  
{% **block** content %}  
 <**h3**>**添加书籍**</**h3**>  
{% **endblock** %}  
{% **block** book\_title\_name %}  
 <**input** type=**"text"** class=**"form-control"** name=**"title"**>  
{% **endblock** %}  
{% **block** book\_price %}  
 <**input** type=**"text"** class=**"form-control"** name=**"price"**>  
{% **endblock** %}  
{% **block** book\_pub\_date %}  
 <**input** type=**"date"** class=**"form-control"** name=**"date"**>  
{% **endblock** %}  
{% **block** book\_publish %}  
 <**input** type=**"text"** class=**"form-control"** name=**"publish"**>  
{% **endblock** %}

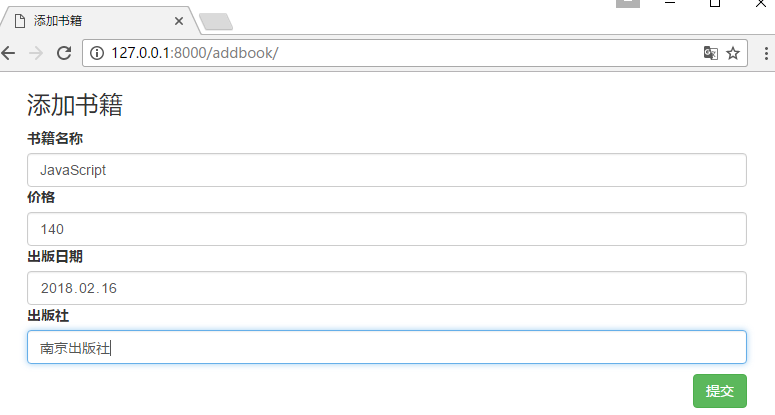
注意：要使用父模板，一定要先extends父模板，不然下面的block无法解析。

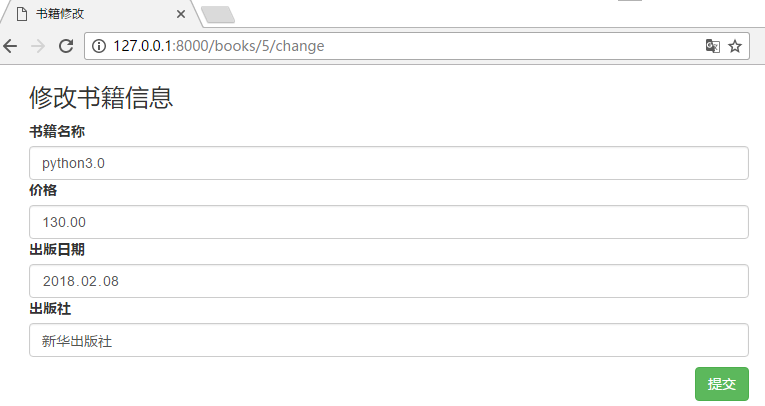
### 11.创建修改页面changebook.html

{% **extends** 'base.html' %}  
{% **block** title %}  
 <**title**>**书籍修改**</**title**>  
{% **endblock** %}  
{% **block** content %}  
 <**h3**>**修改书籍信息**</**h3**>  
{% **endblock** %}  
{% **block** book\_title\_name %}  
 <**input** type=**"text"** class=**"form-control"** name=**"title"** value=**"**{{ book\_obj.title }}**"**>  
{% **endblock** %}  
{% **block** book\_price %}  
 <**input** type=**"text"** class=**"form-control"** name=**"price"** value=**"**{{ book\_obj.price }}**"**/>  
{% **endblock** %}  
{% **block** book\_pub\_date %}  
 <**input** type=**"date"** class=**"form-control"** name=**"date"** value=**"**{{ book\_obj.pub\_date|date:'Y-m-d' }}**"**>  
{% **endblock** %}  
{% **block** book\_publish %}  
 <**input** type=**"text"** class=**"form-control"** name=**"publish"** value=**"**{{ book\_obj.publish }}**"**>  
{% **endblock** %}

### 12.最终页面展示









## 9.多表操作-创建表

### 1.数据库表之间关系简介

数据库的表之间存在，一对一、一对多、多对多 的关系。

图书管理系统有这几张表：作者详情表（authordetail）、出版社信息表（publish）、书籍信息表（book）、作者表（author）、作者书籍表（book\_authors）

关系：

出版社表与书籍信息表是一对多的关系：一个出版社可以出版多本书籍。就可以用书籍信息表的一个外键去对应出版社表的主键ID。

书籍信息表与作者表是多对多的关系：一本书可以有多个作者，一个作者可以有多本书。

作者表与作者详情表是一对一的关系：一个作者只能有一个详情地址。

### 2.创建一个Multi\_table\_operation项目及app01应用

创建项目：django-admin.py startproject Multi\_table\_operation

创建应用：python3 manage.py startapp app01

### 3.配置信息

1.配置urls.py的路由转发，添加一个add的路由转发

2.配置使用MySQL数据库

3.添加pymysql

### 4.创建表

*from* django.db *import* models  
# Create your models here.  
# 作者详情表  
*class* AuthorDetail(models.Model):  
 nid = models.AutoField(primary\_key=*True*)  
 birthday=models.DateField()  
 telephone=models.BigIntegerField()  
 addr=models.CharField( max\_length=64)  
# 出版社信息表  
*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()  
 *def* \_\_str\_\_(self):  
 *return* self.name  
  
*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建立一对多的关系,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',)  
 *def* \_\_str\_\_(self):  
 *return* self.title  
  
*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 to\_field 默认关联主键  
 authordetail=models.OneToOneField(to="AuthorDetail",to\_field="nid", on\_delete=models.CASCADE)

知识点：

1.建立一对多的关系，使用外键，外键字段建立在多的一张表

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

2.创建多对多的关系，可以在两张表中的任意一个创建，用代码实现是自动创建第三张表

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

3.建立一对一的关系，可以在两张表中的任意一个创建。

authordetail=models.OneToOneField(to="AuthorDetail",to\_field="nid", on\_delete=models.CASCADE)

如果不指定 to\_field的值，那么默认是关联另外一张表的主键。

4.\_\_str\_\_

打印对象的时候返回对象的信息。

5.注意事项

外键字段会在字段名上添加“\_id”来创建数据库中的列名。

外键字段 ForeignKey 有一个 null=True 的设置(它允许外键接受空值 NULL)，你可以赋给它空值 None 。

## 10.多表操作-添加数据

### 1.添加数据到无对应关系的表

*def* add(*request*):  
 Publish.objects.create(name="人民出版社",email="123qq.com",city="beijing")  
 Publish.objects.create(name="新华出版社",email="1234qq.com",city="beijing")

*return* HttpResponse("ok")

### 2.绑定一对多的关系

方式一：

为book表绑定出版社（publish）表：

book\_obj = Book.objects.create(title="python",price=100,publishDate="2012-12-11",publish\_id=1)  
print(book\_obj.title,book\_obj.publish\_id)

为下面的方式二理解下publish\_id=1 ：publish\_id 该字段是创建数据库表语句时通过外键创建的，此时就算用publish\_id=1 插入到数据库的表，django在执行该语句的时候，也会有一个publish对象产生。

方式二：publish等于一个实例模型对象

pub\_obj = Publish.objects.filter(nid=2).first()  
book\_obj = Book.objects.create(title="JavaScript",price=180,publishDate="2017-08-14",publish=pub\_obj)  
print(book\_obj.title)  
print(book\_obj.publish) #Publish object (1) 与这本书关联的的出版社对象  
print(book\_obj.publish.email)

要获取Publish中 book\_obj 别的值，可以直接使用 “book\_obj.publish.字段名” 去获取。

### 3.查询java出版社的邮箱

book\_obj = Book.objects.filter(title="java").first()  
print(book\_obj.publish.email) # 123qq.com

### 4.绑定多对多的关系

在绑定多对多之前要注意：在操作多对多的那张表时，该表创建的对象没有在modes.py 的名称空间中存在。因此，不能直接使用 “对象.objects” 去操作表。所以，Django提供了一个接口去操作多对多的那张表。

book\_obj = Book.objects.create(title="HTML", price=80, publishDate="2013-02-21", publish\_id=1)  
ryan = Author.objects.get(name="ryan")  
cherry = Author.objects.get(name="cherry")  
# 绑定多对多关系的API  
book\_obj.authors.add(ryan,cherry)  
# book\_obj.authors.add(1,2) # 此时的1和2 就是ryan和cherry对应的主键ID  
# book\_obj.authors.add(1,2) 等价于 book\_obj.authors.add(\*[1,2])

代码中的 add就是提供的API去操作多对多的关系。

### 5.解除多对多的关系

book = Book.objects.filter(nid=4).first()  
book.authors.remove(1) # 删除 nid=4 的书籍在 book\_authors中作者ID等于1的关系。  
# book.authors.remove(1，2) 等价于book.authors.remove(\*[1,2])

book.authors.clear() #清空所有book 的nid=4 对象的关系主键

remove就是解除多对多的关系。也可以用clear，但是clear就是清空所有。

### 6.多对多关系之all方法

book = Book.objects.filter(nid=4).first()

print(book.authors.all()) # 返回 <QuerySet [<Author: cherry>]> ; 获取与这本书关联的所有作者对象集合  
print(book.authors.all().values("name")) # <QuerySet [{'name': 'cherry'}]> 获取作者名  
print(book.authors.all().values("name")[0].get("name")) # cherry

# 11.多表操作-查询数据

## 1.跨表查询分类

跨表查询分为：

1.基于对象查询

2.基于双下划线查询

3.聚合和分组查询

4. F 与 Q查询

## 2.基于对象的跨表查询(子查询)

### 1.一对多查询

**一对多的正向查询**：查询python这本书出版社的名字  
book\_obj = Book.objects.filter(title="python").first() print(book\_obj.publish)  
print(book\_obj.publish.name)

解释：publish属性是 Book表和Publish表之间的关联字段。

**一对多的反向查询**：查询新华出版社出版了什么书籍

publish\_obj = Publish.objects.filter(name="新华出版社").first()  
print(publish\_obj.book\_set.all()) # QuerySet

反向查询：obj.表名小写\_set.all()

### 2.多对多查询

**多对多查询的正向查询** ：查询XHTML所有作者的名字  
book\_obj = Book.objects.filter(title="XHTML").first()  
author\_list = book\_obj.authors.all() # QuerySet 对象  
*for* author *in* author\_list:  
 print(author.name)

**多对多查询的反向查询** ：查询ryan出版过的所有书籍的名称  
author\_obj = Author.objects.filter(name='ryan').first()  
book\_list = author\_obj.book\_set.all()  
*for* book *in* book\_list:  
 print(book.title)

### 3.一对一查询

**一对一查询的正向查询**：查找ryan详情  
author\_obj = Author.objects.filter(name="ryan").first()  
print(author\_obj.authordetail.telephone)

**一对一查询的反向查询：**查询手机号为 911 的作者名  
author\_obj = AuthorDetail.objects.filter(telephone=911).first()  
print(author\_obj.author.name)

### 4.基于对象的跨表查询总结

A与B有关系：关联属性在A表中

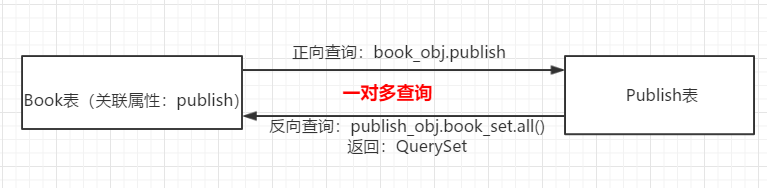
正向查询： A表 -----正向查询-------> B表

反向查询： B表 -----反向查询------->A表

一对多查询的正反向查询：

正向查询：按字段

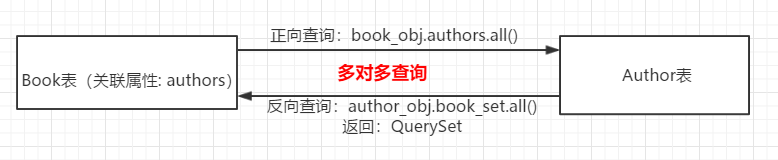
反向查询：表名小写\_set.all()



多对多查询的正反向查询

正向查询：按字段

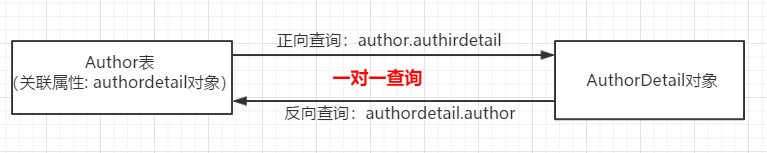
反向查询：表名小写\_set.all()



一对一查询的正反向查询

正向查询：按字段

反向查询：按表名小写



## 3.基于双下划线的跨表查询（join查询）

基于双下划线的跨表查询（join查询）

正向查询：按字段

反向查询：按表名小写

正向查询字段，反向查询按表名小写，用来告诉ORM引擎join哪张表

### 1.一对多查询

**一对多正向查询**： 查询XHTML这本书的出版社的名字  
book\_obj = Book.objects.filter(title="XHTML").values("publish\_\_name")  
print(book\_obj) # <QuerySet [{'publish\_\_name': '人民出版社'}]>

这句的意思就是，从Book表中过滤出XHTML的书籍，然后，Book表去join Publish表，然后得到publish表中的name值。

**一对多反向查询**： 查询XHTML这本书的出版社的名字  
ret = Publish.objects.filter(book\_\_title="XHTML").values("name")  
print(ret)

因为反向查询按表名，因此，这句话的意思就是，Publish表去join Book表，然后过滤出Book表中title等于XHTML的信息，然后在获取Publish表中的name 值。

### 2.多对多查询

**多对多正向查询** ：查询XHTML书籍所有作者的名字  
#需求：通过Book表join与其关联的Author表,属于正向查询：按照字段authors通知ORM引擎join book\_authors 与 author  
ret = Book.objects.filter(title="XHTML").values("authors\_\_name")  
*for* author *in* ret:  
 print(author.get('authors\_\_name'))

**多对多反向查询** ：查询XHTML书籍所有作者的名字  
# 需求：通过Author表join与其关联的Book表，属于反向查询：按表名小写book通知ORM引擎join book\_authors 与 author表  
ret = Author.objects.filter(book\_\_title="XHTML").values("name")  
print(ret)

### 3.一对一查询

**一对一正向查询**：查找作者ryan详情  
# 需求：通过Author表join与其关联的AuthorDetail表，属于正向查询：按照字段authordetail 通知ORM引擎 join author 表 与 AuthorDetail表  
author\_obj = Author.objects.filter(name="ryan").values("authordetail\_\_telephone")  
print(author\_obj)

**一对一反向查询**：查找作者ryan详情

# 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通知join author 表 与 AuthorDetail表  
author\_obj = AuthorDetail.objects.filter(author\_\_name="ryan").values("telephone")  
print(author\_obj)

### 4.基于下划线的连续跨表练习

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

需求：通过Book表join AuthorDetail表，Book与AuthorDetail无关联，所以必须连续跨表

解析：要查询书籍以及出版社，就要通过Author为中间表，Author通过AuthorDetail表去找到Book表，然后，Book表在去join Publish表。

方式一：

ret = Book.objects.filter(authors\_\_authordetail\_\_telephone\_\_startswith="110").values("title","publish\_\_name")  
  
print(ret)

语句解析：首先是Author表去join AuthorDetail表，属于正向查询，因此，按字段telephone去查询。Book表查询 Author表，也是正向查询，因此，通过关联属性 authors去找到书籍和作者的信息。然后，书籍去找出版社也是正向查询，因此，在values里面按照字段找即可。

方式二：

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

语句理解：通过Author表开始找，首先，Author与AuthorDetail表有关联字段，并且是正向查询，所以按照telephone字段去查询。然后Author表和Book表是反向查询，按照表名小写（book\_title）去找到作者出版过的书籍，最后，作者表和出版社没有关联，因此，要通过book表作为中间表，去找到Publish表中的出版社，就用（book\_\_publish\_\_name）就得到了出版社name的名称。

## 4.聚合查询aggregate

aggregate 称为聚合统计函数，它的返回值是一个字典。

示例：查询所有书籍的平均价格，最大价格和所有书籍总数

*from* django.db.models *import* Avg,Max,Min,Count  
ret = Book.objects.all().aggregate(avg\_price = Avg("price"),max\_price = Max("price"),count = Count("price"))  
print(ret)

也可以省去all（）方法。

## 5.分组查询 annotate

annotate 的返回值依然是 QuerySet

单表的分组查询的ORM语法：单表模型.objescts.values("group by的字段").annotate(聚合函数("统计字段"))

*class* Emp(models.Model):  
 name=models.CharField(max\_length=32)  
 age=models.IntegerField()  
 salary=models.DecimalField(max\_digits=8,decimal\_places=2)  
 dep=models.CharField(max\_length=32)  
 province=models.CharField(max\_length=32)

### 1.单表分组查询

示例一：查询每一个部门的名称以及员工的平均薪水

ret = Emp.objects.values("dep").annotate(avg\_salary = Avg("salary"))  
print(ret)

示例二：查询每一个省份的名称，以及对应的员工数

ret = Emp.objects.values("province").annotate(emp\_count = Count("id"))  
print(ret)

### 2.分组跨表查询

分组跨表的模型：

每一个对象的表模型.objects.values('pk').annotate(聚合函数(关联表\_\_统计字段)).values(表模型字段，以及统计的字段)

每一个对象的表模型.objects.all().annotate(聚合函数(关联表\_\_统计字段)).values(表模型字段，以及统计的字段) #注释：all() 可以去掉

提示：values中的字段，就是需要展示的字段

示例一：查找每一个出版社的名称以及出版的书籍个数

方式一：

ret = Publish.objects.values("name").annotate(book\_count=Count("book\_\_title"))  
print(ret) #<QuerySet [{'name': '人民出版社', 'book\_count': 4}, {'name': '新华出版社', 'book\_count': 2}]>

方式二：

ret = Publish.objects.values("nid").annotate(book\_count=Count("book\_\_title")).values("book\_\_publish\_\_name","book\_count","book\_\_publish\_\_email")  
ret = Publish.objects.values("nid").annotate(book\_count=Count("book\_\_title")).values("name","book\_count","email")  
print(ret) #<QuerySet [{'name': '人民出版社', 'email': '123qq.com', 'book\_count': 4}, {'name': '新华出版社', 'email': '1234qq.com', 'book\_count': 2}]>

两种ret 的方式都可以，一种是跨表，一种是不跨表。

示例二：查询每一个作者的名字以及出版过的书籍的最高价格

ret = Author.objects.values("pk").annotate(book\_max\_price=Max("book\_\_price")).values("name","book\_max\_price")  
print(ret) #<QuerySet [{'name': 'ryan', 'book\_max\_price': Decimal('120.00')}, {'name': 'cherry', 'book\_max\_price': Decimal('130.00')}]>

示例三：查询每本书籍的名称以及对应的作者个数

ret = Book.objects.values('pk').annotate(c=Count("authors\_\_name")).values("title","c")  
print(ret)  
# result：<QuerySet [{'title': 'java', 'c': 1}, {'title': 'HTML', 'c': 1}, {'title': 'XHTML', 'c': 2}, {'title': 'HTML5', 'c': 1}, {'title': 'python', 'c': 0}, {'title': 'JavaScript', 'c': 0}]>

示例四：查询每一个出版社的名称以及出版的书籍个数

ret = Publish.objects.values("pk").annotate(c=Count("book\_\_title")).values("name","email","c")  
print(ret)  
ret = Publish.objects.annotate(c=Count("book\_\_title")).values("name","email","c")  
print(ret)  
# 两个ret都是：<QuerySet [{'name': '人民出版社', 'email': '123qq.com', 'c': 4}, {'name': '新华出版社', 'email': '1234qq.com', 'c': 2}]>

Publish.objects.all().annotate(c=Count("book\_\_title")) 是一个Publish对象，因此，values就相当于调用该对象里面的属性。

示例五：统计每一本以HT开头的书籍的作者个数

ret = Book.objects.filter(title\_\_startswith="HT").values("pk").annotate(author\_count=Count("authors\_\_name")).values("title","author\_count")  
print(ret) #<QuerySet [{'title': 'HTML', 'author\_count': 1}, {'title': 'HTML5', 'author\_count': 1}]>

示例六：统计不止一个作者的书籍

ret = Book.objects.values("pk").annotate(c=Count("authors\_\_name")).filter(c\_\_gt=1).values("title","c")  
print(ret) # <QuerySet [{'title': 'XHTML', 'c': 2}]>