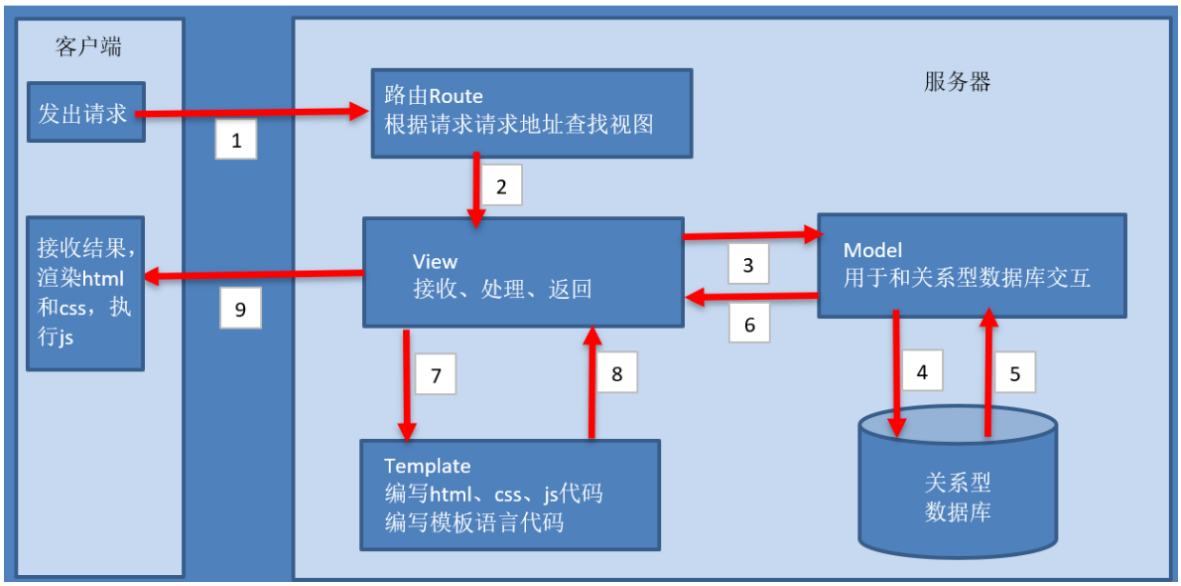
Django进阶

Django的MVT

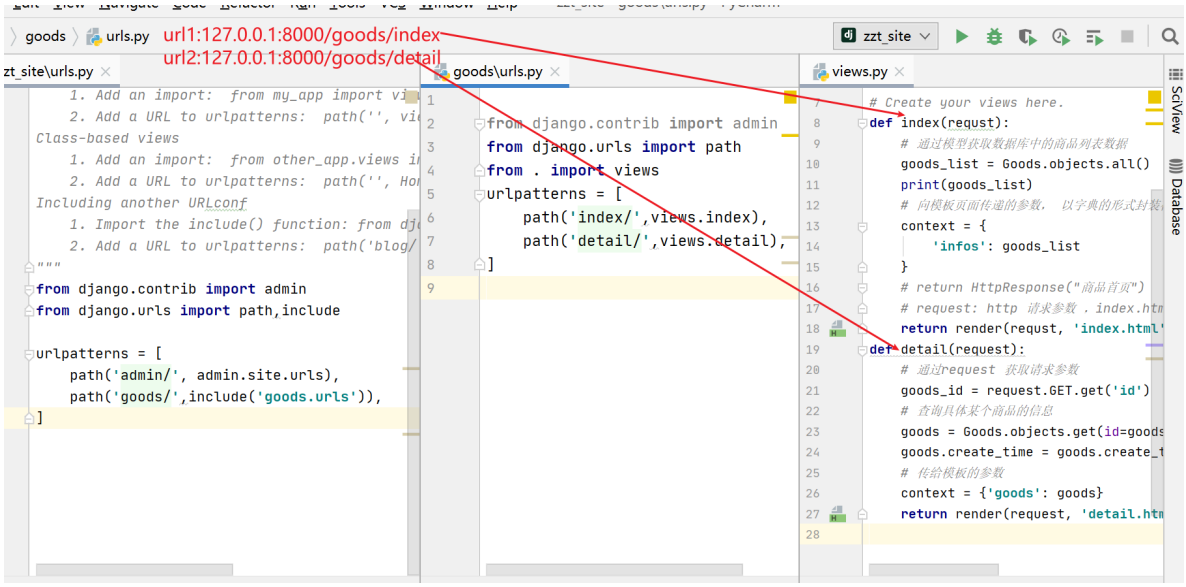


一、Django路由系统

1.认识路由系统

通过URL（Uniform Resource Locator，统一资源定位符）可以访问互联网上的资源——用户通过浏览器向指定URL发起请求，Web服务器接收请求并返回用户请求的资源，因此可以将URL视为用户与服务器之间交互的桥梁。

在Django当中，路由系统的本质就是URL与要调用URL的视图函数的一个映射表，它的主要作用就是让views里面的执行函数与请求的url建立映射关系，当请求来了的时候根据这个url映射来调用对应的执行函数，从而返回给客户端相应的信息



1) Django处理请求的流程

当用户向你的Django站点请求一个页面时，系统会采用一个算法来确定要执行哪一段Python代码：

1. 首先，Django会使用根路由解析模块(root URLconf)来解析路由。
2. Django加载该Python模块并查找变量`urlpatterns`。 它应该是`django.urls.path()`或者`django.urls.re_path()`实例的Python列表。
3. Django按顺序遍历每个URL pattern，并在第一个匹配的请求URL被匹配时停下。
4. 一旦某个URL pattern成功匹配，Django会导入并调用给定的视图，该视图是一个简单的Python函数（或基于类的视图）。

   这个视图会被传以以下参数：

   - 一个`HttpRequest`的实例。
   - 如果所匹配的正则表达式返回的是若干个无名组，那么该正则表达式所匹配的内容将被作为位置参数提供给该视图。
   - 关键字参数是由路径表达式匹配的任何指定部件组成的，在可选的`kwargs`参数中指定的任何参数覆盖到`django.urls.path()`或`django.urls.re_path()`。
5. 如果请求的URL没有匹配到任何一个表达式，或者在匹配过程的任何时刻抛出了一个异常，那么Django 将调用适当的错误处理视图进行处理

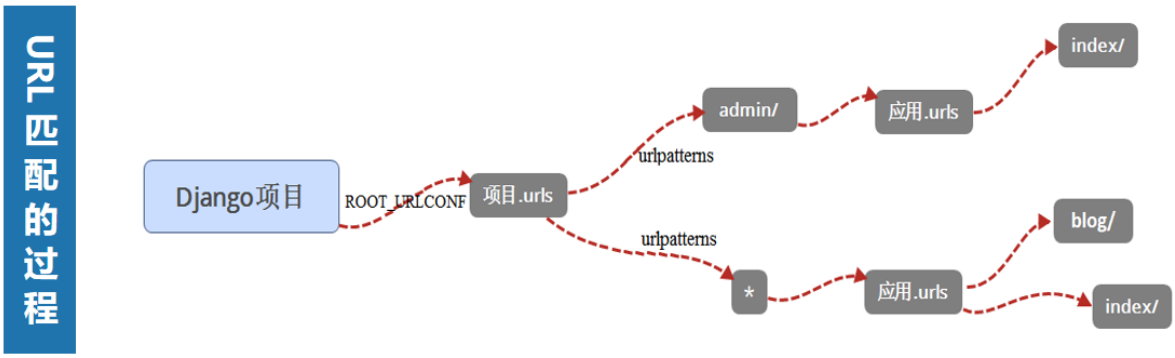
2) Django URL配置示例

> 一个项目允许有多个urls.py，但Django需要一个urls.py作为入口，这个特殊的urls.py就是根URLconf（根路由配置），它由settings.py文件中的ROOT_URLCONF指定

```
ROOT_URLCONF = 'mysite.urls'
```

以上示例通过ROOT_URLCONF指定了mysite目录下的urls.py作为根URLconf

为保证项目结构清晰，开发人员通常在Django项目的每个应用下创建urls.py文件，在其中为每个应用配置子URL。路由系统接收到HTTP请求后，先根据请求的URL地址匹配根URLconf，找到匹配的子应用，再进一步匹配子URLconf，直到匹配完成



3) 示例

urldemo应用的url配置示例:

```
from django.urls import path

from . import views

urlpatterns = [
    path('articles/2021/', views.special_case_2021),
    path('articles/<int:year>/', views.year_archive),
    path('articles/<int:year>/<int:month>/', views.month_archive),
    path('articles/<int:year>/<int:month>/<slug:slug>/', views.article_detail),
]
```

views.py视图

```
from django.shortcuts import render
from django.http import HttpResponse


def special_case_2021(request):
    return HttpResponse("special_case_2021")

def year_archive(request, year):
    return HttpResponse(f"year_archive{year}")

def month_archive(request, year, month):
    return HttpResponse(f"month_archive 年：{year} 月：{month}")

def article_detail(request, year, month, slug):
    return HttpResponse(f"article_detail 年：{year} 月：{month} {slug}")
```

提示:

- 要从URL捕获某个值，使用尖角括号。
- 捕获的值可以选择包含一个转换器类型。 例如, 使用 `<int:name>` 来捕获一个整形参数。 如果没有包含转换器, 那么除了 `/` 字符外, 会匹配任意字符串。
- 因为每个URL都有前导斜杠，所以使用时，每个URL没有必要添加一个/。 例如, path函数中URL部分应该使用 `articles`, 而不是 `/articles`。

示例请求：

- 对 `/articles/2021/03/` 的请求将与列表中的第三个条目匹配。 Django会调用函数 `views.month_archive（request, year=2021, month=3）`。
- `/articles/2021/` 将与列表中的第一个模式相匹配，而不是第二个，因为这些模式是按顺序匹配的，第一个模式会首先测试是否匹配。 请像这样自由插入一些特殊的情况来探测匹配的次序。 在这里，Django会调用函数 `views.special_case_2021（request）`
- `/articles/2020` 不会匹配到任何一个模式，因为每个模式都要求URL以斜杠结尾。
- `/articles/2021/03/building-a-django-site/` 会匹配到最后一个模式。 Django会调用函数 `views.article_detail（request, year=2021, month=3, slug ="building-a-django-site"）`。

2. 路由转换器

Django内置路由转换器可以显式地指定路由中参数的数据类型，Django中内置了5种路由转换器，分别为str、int 、slug 、uuid 和path

- str：匹配任何非空字符串，但不包含路由分隔符“/”。如果URL中没有指定参数类型，默认使用该类型。
- int：匹配0或任何正整数。并作为 int 返回。
- slug：匹配由字母、数字、连字符和下划线组成的URL。例如，`building-your-1st-django-site`
- uuid：匹配一个uuid格式的字符串。为了防止多个URL映射到同一页面中，该转换器必须包含连字符，且所有字母均为小写。例如，`075194d3-6885-417e-a8a8-6c931e272f00`. 返回一个 `UUID` 实例
- path：匹配包含路径分隔符 `'/'` 在内的任意非空字符串。 相对于 `str`，这允许你匹配一个完整的URL路径，而不仅仅是URL路径的一部分。

3. 使用正则表达式

如果路径和转换器语法不足以定义你的URL pattern，你还可以使用正则表达式。 为了使用正则表达式，请使用 `re_path()`，而不要使用 `path()`。

在Python正则表达式中，命名正则表达式组的语法是 `(?Ppattern)`， 这里 `name` 是表达式组的名字而 `pattern` 是要匹配的模式。

语法：`re_path(route, view, kwargs=None, name=None)`

以下是使用正则表达式重写的前面的示例URLconf：

```
from django.contrib import admin
from django.urls import path, include, re_path
from . import views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('users/', include('users.urls')),
    re_path('articles/2021/$', views.special_case_2021),
    re_path('articles/(?P<year>[0-9]{4})/$', views.year_archive),
    re_path('articles/(?P<year>[0-9]{4})/(?P<month>[0-9]{2})/$',
views.month_archive),
    re_path('articles/(?P<year>[0-9]{4})/(?P<month>[0-9]{2})/(?P<slug>[-
_\w]+)/$', views.article_detail),
]
```

提示：

- 无论正则表达式匹配什么类型，每个捕获的参数都会以字符串形式传给视图。
- 正则表达式格式为：(?Ppattern)，其中name表示分组名，pattern表示匹配的正则表达式。URL匹配成功后，捕获到的参数会作为关键字参数传递给对应的视图，因此视图中的形式参数必须和正则表达式中的分组名相同
- 若正则表达式只通过小括号“()”来捕获URL的参数，但未为其命名，则它是一个未命名正则表达式，此时捕获的参数并将其以位置参数形式传递给对应视图

4. 向视图传递额外参数

- path()函数、re_path()函数允许向视图传递额外参数，这些参数存放在一个字典类型的数据中，该数据的键代表参数名，值代表参数值。re_path()函数与path()函数传递额外参数方式相同，以path()函数为例介绍如何向视图传递额外参数。
- 使用path()函数的第三个参数可以向视图传递额外参数

urls.py

```
 path('blog/<int:year>/', views.blog_year_archive, {'major': 'python'})
 re_path('blog/(?P<year>[0-9]{4})/$', views.blog_year_archive, {'major':
'python'}),
```

views.py

```
def blog_year_archive(request, year, major):
    return HttpResponse(f"blog_year_archive 年 {year} 额外参数:{major}")
```

提示

- 路由解析顺序

  Django在接收到一个请求时，从主路由文件中的urlpatterns列表中以由上至下的顺序查找对应路由规则，如果发现规则为include包含，则再进入被包含的urls中的urlpatterns列表由上至下进行查询

- 反斜线问题

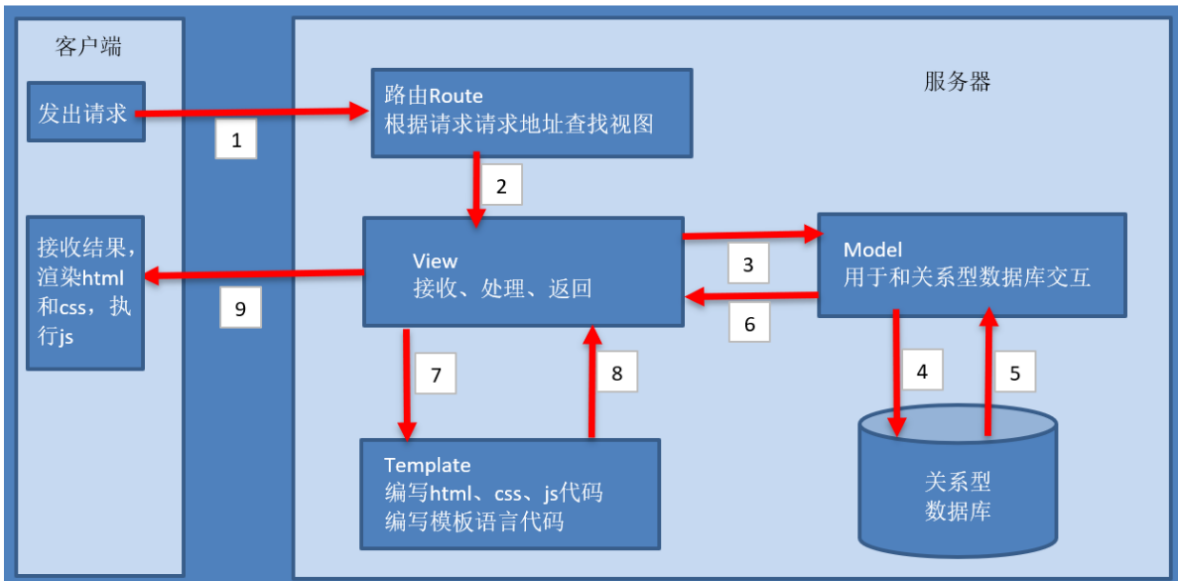
  Django中定义路由时，通常以斜线/结尾，其好处是用户访问不以斜线/结尾的相同路径时，Django会把用户重定向到以斜线/结尾的路径上，而不会返回404不存在

二、视图

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。

视图的第一个参数必须为HttpRequest实例，且视图必须返回一个HttpResponse对象或子对象作为响应

Django的MVT



2.1 请求HttpRequest

回想一下，利用HTTP协议向服务器传参有几种途径?

- 提取URL的特定部分，如/weather/beijing/2021，可以用路由转换器或者正则表达式截取;
- 查询字符串（query string），形如`?key1=value1&key2=value2`;
- 请求体（body）中发送的数据，比如表单数据、json、xml;

- 在http报文的头（header）中。

1 URL路径参数

在定义路由URL时，可以使用正则表达式提取参数的方法从URL中获取请求参数，Django会将提取的参数直接传递到视图的传入参数中。

url路由

- 路由转换器提取

```
path('weather/<str:city>/<int:year>', views.weather),
```

- 正则表达式未命名参数按定义顺序传递， 如

```
re_path('weather/([a-z]+)/(\d{4})/', views.weather),
```

- 命名参数按名字传递，如

```
re_path('weather/(?P<city>[a-z]+)/(?P<year>\d{4})/', views.weather),
```

视图views.py

```
from django.shortcuts import render
from django.http import HttpResponse
# Create your views here.
def weather(request, city, year):
    return HttpResponse(f'weather :城市 {city} 年份:{year}')
```

2 Django中的QueryDict对象

定义在django.http.QueryDict

HttpRequest对象的属性GET、POST都是QueryDict类型的对象

与python字典不同，QueryDict类型的对象用来处理同一个键带有多个值的情况

- 方法get()：根据键获取值

  如果一个键同时拥有多个值将获取最后一个值

  如果键不存在则返回None值，可以设置默认值进行后续处理

```
dict.get('键',默认值)
可简写为
dict['键']
```

- 方法getlist()：根据键获取值，值以列表返回，可以获取指定键的所有值

  如果键不存在则返回空列表[]，可以设置默认值进行后续处理

```
dict.getlist('键',默认值)
```

3. 查询字符串Query String

获取请求路径中的查询字符串参数（形如?k1=v1&k2=v2），可以通过request.GET属性获取，返回QueryDict对象。

视图views.py

```
# /qs?a=xxx&b=yyy
def qs(request):
    """查询字符串获取"""
    a = request.GET.get("a")
    b = request.GET.get("b")
    b_list = request.GET.getlist("b")
    return HttpResponse(f'参数a：{a}, 参数b：{b} 参数b(列表)：{b_list}')
```

重要：查询字符串不区分请求方式，即使客户端进行POST方式的请求，依然可以通过request.GET获取请求中的查询字符串数据。

4 请求体：

请求体数据格式不固定，可以是表单类型字符串，可以是JSON字符串，可以是XML字符串，应区别对待。

可以发送请求体数据的请求方式有**POST**、**PUT**、**PATCH**、**DELETE**

Django默认开启了CSRF防护，会对上述请求方式进行CSRF防护验证，在测试时可以关闭CSRF防护机制，方法为在settings.py文件中注释掉CSRF中间件，如：

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    # 'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
```

4.1 表单类型 Form Data

前端发送的表单类型的请求体数据，可以通过request.POST属性获取，返回QueryDict对象。

注册表单模板页面 regForm.html

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>注册表单</title>
</head>
<body>
<form method="post" action="/viewdemo/register/">
    姓名：<input type="text" name="uname"/><br>
    密码：<input type="password" name="upwd"/><br>
    性别：<input type="radio" name="ugender" value="1"/>男
    <input type="radio" name="ugender" value="0"/>女<br> 爱好：
    <input type="checkbox" name="uhobby" value="抽烟"/>抽烟
    <input type="checkbox" name="uhobby" value="喝酒"/>喝酒
    <input type="checkbox" name="uhobby" value="烫头"/>烫头<br>
    <input type="submit" value="提交"/>
</form>
```

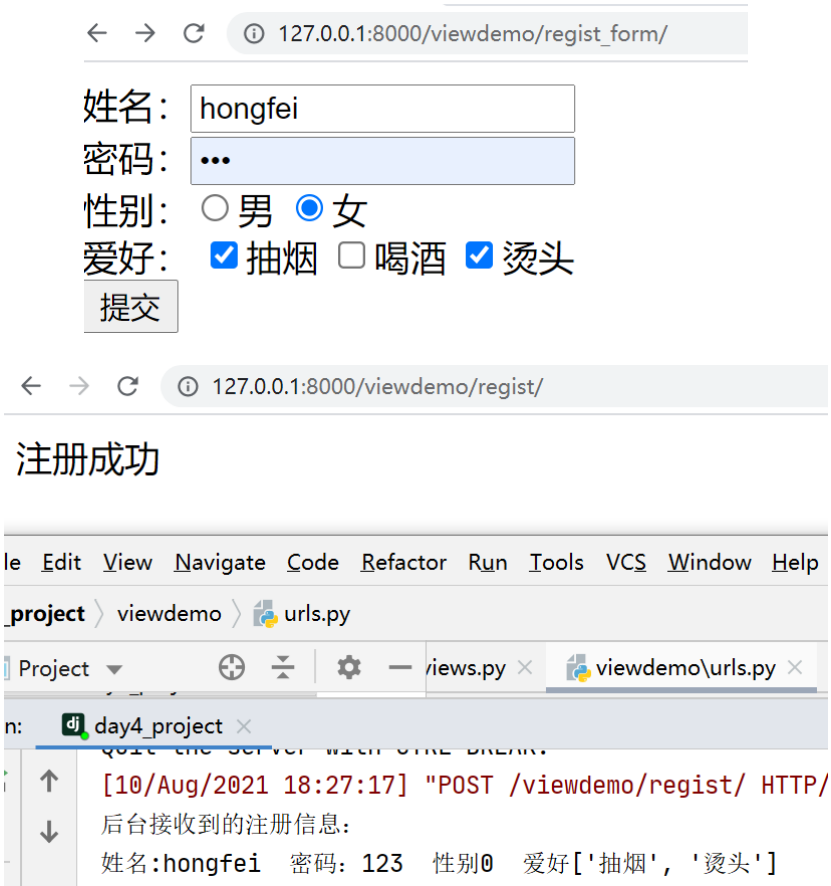
```
</body>
</html>
```

实现注册功能的视图

```
#views.py
def get_reg_form(request):
    """获取注册表单页面的视图"""
    return render(request, 'reg_form.html')
def regist(request):
    """注册功能实现的视图"""
    p_dict = request.POST  # 获取post请求参数
    uname = p_dict.get('uname')
    upwd = p_dict.get('upwd')
    ugender = p_dict.get('ugender')
    uhobby = p_dict.getlist('uhobby')  # 爱好是一键多值，使用getlist方法获取所有爱好
    print(f'后台接收到的注册信息：')
    print(f'姓名:{uname}  密码：{upwd}  性别{ugender}  爱好{uhobby}')
    return HttpResponse("注册成功")
```

路由urls.py

```
urlpatterns = [
    path('register_form/', views.get_reg_form),  # 显示注册表单界面的 请求
    path('register/', views.register), # 注册
]
```



提示：request.POST只能用来获取POST方式的请求体表单数据。

4.2 非表单类型 Non-Form Data

非表单类型的请求体数据 (ajax提交数据)，Django无法自动解析，可以通过**request. body**属性获取最原始的请求体数据，自己按照请求体格式（JSON、XML等）进行解析。**request.body返回bytes类型。**

例如要获取请求体中的如下JSON数据

```
{"a": 1, "b": 2}
```

可以进行如下方法操作：

```
import json

def get_body_json(request):
    json_str = request.body  #{"a": 1, "b": 2}
    json_str = json_str.decode()  # python3.6 无需执行此步
    req_data = json.loads(json_str)# 字典
    print(req_data['a'])
    print(req_data['b'])
    return HttpResponse('OK')
```

5 请求头

可以通过**request.META**属性获取请求头headers中的数据，**request.META为字典类型。**

常见的请求头如：

- `CONTENT_LENGTH` – The length of the request body (as a string).
- `CONTENT_TYPE` – The MIME type of the request body.
- `HTTP_ACCEPT` – Acceptable content types for the response.
- `HTTP_ACCEPT_ENCODING` – Acceptable encodings for the response.
- `HTTP_ACCEPT_LANGUAGE` – Acceptable languages for the response.
- `HTTP_HOST` – The HTTP Host header sent by the client.
- `HTTP_REFERER` – The referring page, if any.
- `HTTP_USER_AGENT` – The client's user-agent string.
- `QUERY_STRING` – The query string, as a single (unparsed) string.
- `REMOTE_ADDR` – The IP address of the client.
- `REMOTE_HOST` – The hostname of the client.
- `REMOTE_USER` – The user authenticated by the Web server, if any.
- `REQUEST_METHOD` – A string such as `"GET"` or `"POST"`.
- `SERVER_NAME` – The hostname of the server.
- `SERVER_PORT` – The port of the server (as a string).

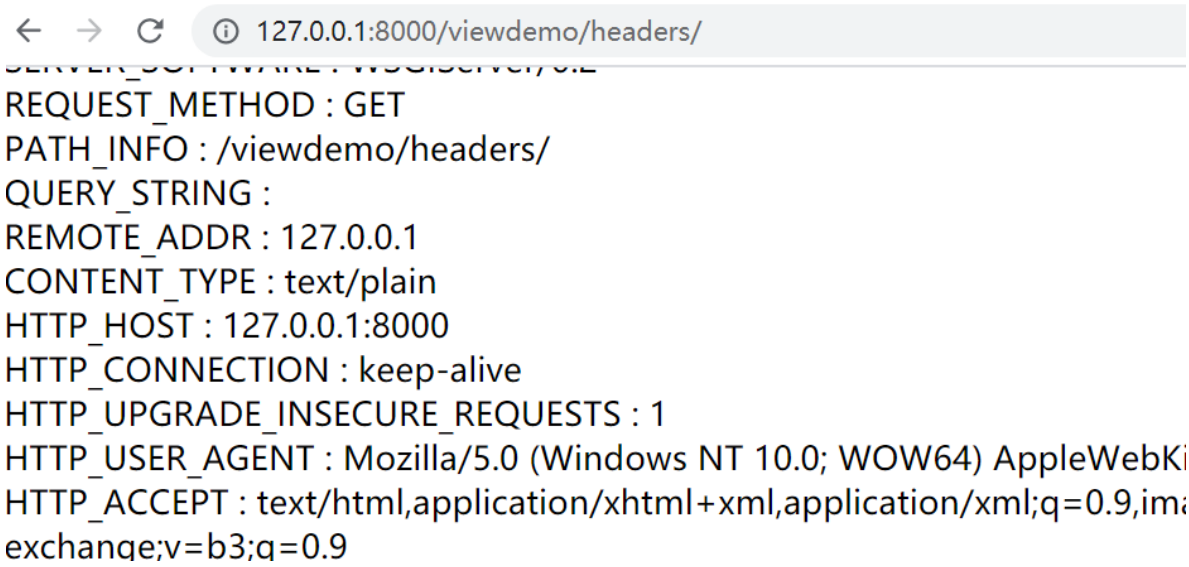
视图设置如下:

```
def get_headers(request):
    """获取请求头数据"""
    m = request.META
    str = ''
    # 将http请求头拼接返回给浏览器，查看http请求头信息
    for key, value in m.items():
        str += '%s : %s' % (key, value) + '</br>'
```

```
    # request其他属性
    # print(request.method)
    # print(request.user)
    # print(request.path)
    # print(request.encoding)
    # print(request.FILES)
    # return HttpResponse(str)
    return HttpResponse(str)
```

路由配置如下：

```
    path('headers/', views.get_headers)
```



6 其他常用HttpRequest对象属性

requtest.GET

requtest.POST

request.META

- **method**：一个字符串，表示请求使用的HTTP方法，常用值包括：'GET'、'POST'。
- **user：请求的用户对象。**（如获取user属性值，需访问django_session表 makemigrations migrate）
- path：一个字符串，表示请求的页面的完整路径，不包含域名和参数部分。
- encoding：一个字符串，表示提交的数据的编码方式。
    - 如果为None则表示使用浏览器的默认设置，一般为utf-8。
    - 这个属性是可写的，可以通过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将使用新的encoding值。
- FILES：一个类似于字典的对象，包含所有的上传文件。

2.2 响应 Response

视图在接收请求并处理后，必须返回HttpResponse对象或子对象。HttpRequest对象由Django创建，HttpResponse对象由开发人员创建。

1) HttpResponse

可以使用**django.http.HttpResponse**来构造响应对象。

```
HttpResponse(content=响应体, content_type=响应体数据类型, status=状态码)
```

也可通过HttpResponse对象属性来设置响应体、状态码：

- content：表示返回的内容。
- status_code：返回的HTTP响应状态码。

响应头可以直接将HttpResponse对象当做字典进行响应头键值对的设置：

```
response = HttpResponse()
response['zzt'] = 'Python'  # 自定义响应头zzt, 值为Python
```

示例：

视图Views

```
from django.http import HttpResponse

def resp_view(request):
    # return HttpResponse(content='zzt python',content_type='text/html',
status=400)
    # 或者
    resp = HttpResponse(content_type='text/html', charset='GBK')
    resp.write('大家好')  # content
    resp.status_code = 200  # 状态码
    resp['major'] = 'python'  # 设置响应头
    return resp  # 返回响应对象
```

路由urls.py

```
path('resp/', views.resp_view),
```

2) HttpResponse子类

Django提供了一系列HttpResponse的子类，可以快速设置状态码

- HttpResponseRedirect 301
- HttpResponsePermanentRedirect 302
- HttpResponseNotModified 304
- HttpResponseBadRequest 400
- HttpResponseNotFound 404
- HttpResponseForbidden 403
- HttpResponseNotAllowed 405
- HttpResponseGone 410
- HttpResponseServerError 500

3) JsonResponse

若要返回json数据，可以使用JsonResponse来构造响应对象，作用：

- 帮助我们将数据转换为json字符串
- 设置响应头**Content-Type**为 **application/json**

```
from django.http import JsonResponse

def demo_view(request):
    return JsonResponse({'city': 'beijing', 'subject': 'python'})
```

4) redirect重定向

```
from django.shortcuts import redirect

def demo_view(request):
    return redirect('/viewdemo/register_form')
```

2.3 Cookie

> Cookie，有时也用其复数形式Cookies，指某些网站为了辨别用户身份、进行session跟踪而储存在用户本地终端上的数据（通常经过加密）。

Cookie最早是网景公司的前雇员Lou Montulli在1993年3月的发明。Cookie是由服务器端生成，发送给User-Agent（一般是浏览器），浏览器会将Cookie的key/value保存到某个目录下的文本文件内，下次请求同一网站时就发送该Cookie给服务器（前提是浏览器设置为启用cookie）。Cookie名称和值可以由服务器端开发自己定义，这样服务器可以知道该用户是否是合法用户以及是否需要重新登录等。服务器可以利用Cookies包含信息的任意性来筛选并经常性维护这些信息，以判断在HTTP传输中的状态。Cookies最典型**记住用户名。**

Cookie是存储在浏览器中的一段纯文本信息，建议不要存储敏感信息如密码，因为电脑上的浏览器可能被其它人使用。

Cookie的特点

- Cookie以键值对Key-Value形势进行信息的存储。
- Cookie基于域名安全，不同域名的Cookie是不能互相访问的

1 设置Cookie

可以通过**HttpResponse**对象中的**set_cookie**方法来设置cookie。

```
HttpResponse.set_cookie(cookie名, value=cookie值, max_age=cookie有效期)
```

- **max_age** 单位为秒，默认为**None**。如果是临时cookie，可将max_age设置为None。

2 读取Cookie

可以通过**HttpRequest**对象的**COOKIES**属性来读取本次请求携带的cookie值。**request.COOKIES为字典类型。**

示例

路由urls.py

```
urlpatterns = [
    path('set/cookie/', views.set_cookie),# 服务端设置cookie传递给客户端
    path('send/cookie/', views.send_cookie),# 客户端上传cookie 到服务端
]
```

视图views.py

```
from django.shortcuts import render
from django.http import HttpResponse

# Create your views here.
def set_cookie(request):
    response = HttpResponse('<a href="/viewdemo/send/cookie">客户端默认上传
cookie...</a>')
    response.status_code = 200
    # 服务端设置cookie ，传递给客户端
    response.set_cookie('user_info', value="xiao_wang")
    return response

def send_cookie(request):
    """服务端获取客户端传递过来的cookie"""
    for item in request.COOKIES.items():
        print(item)
    return HttpResponse("ok!")
```

2.4 Session

对于敏感、重要的信息不能存在浏览器，也就不能使用cookie，如用户名、余额、等级、验证码等信息 如何确保数据的安全性呢，那就是将数据保存在服务端。这就利用到session技术。

Session 对象存储特定用户会话所需的属性及配置信息。这样，当用户在应用程序的 Web 页之间跳转时，存储在 Session 对象中的变量将不会丢失，而是在整个用户会话中一直存在下去。当用户请求来自应用程序的 Web页时，如果该用户还没有会话，则 Web 服务器将自动创建一个 Session 对象。当会话过期或被放弃后，服务器将终止该会话。

1 Session介绍

django项目默认启用Session: settings.py文件，在项MIDDLEWARE_CLASSES中启用Session中间件

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
```

存储方式: 设置SESSION_ENGINE项指定Session数据存储的方式，可以存储在数据库、缓存、Redis等存储在数据库中，如下设置可以写，也可以不写，这是默认存储方式

```
SESSION_ENGINE='django.contrib.sessions.backends.db'
```

存储在缓存中：存储在本机内存中，如果丢失则不能找回，比数据库的方式读写更快

```
SESSION_ENGINE='django.contrib.sessions.backends.cache'
```

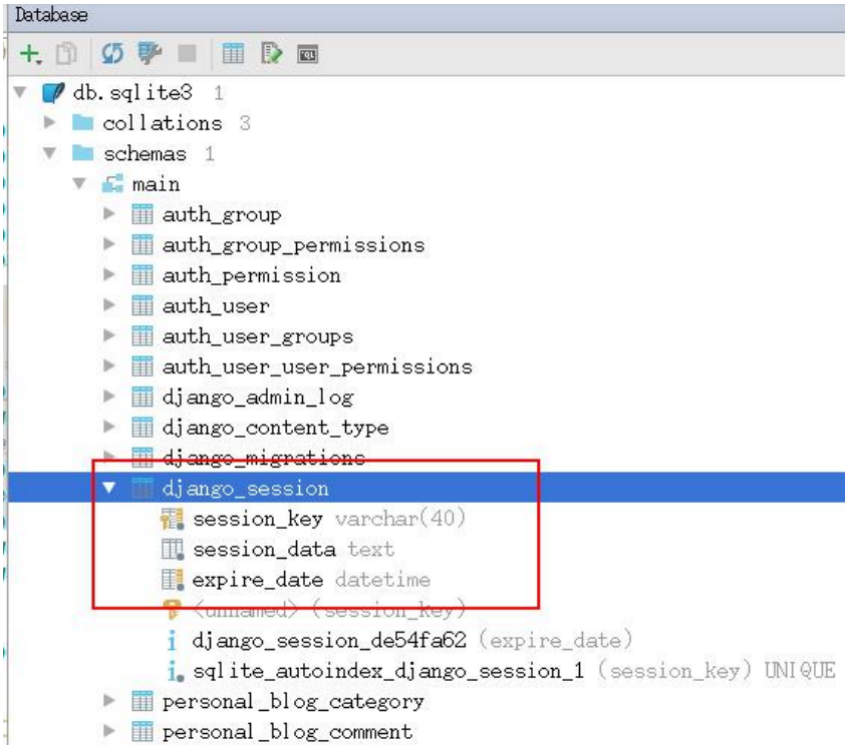
混合存储：优先从本机内存中存取，如果没有则从数据库中存取

```
SESSION_ENGINE='django.contrib.sessions.backends.cached_db'
```

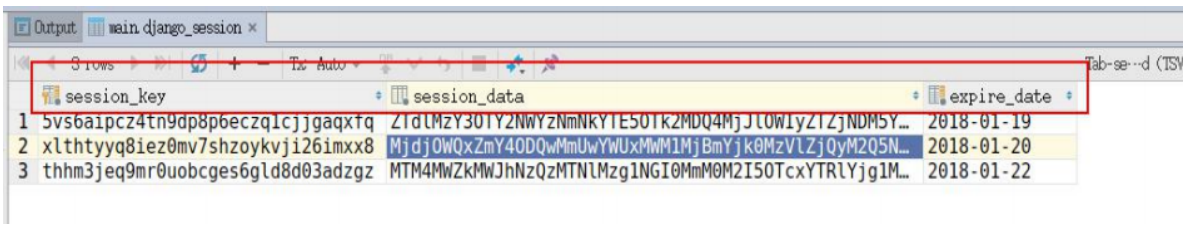
如果存储在数据库中，需要在项INSTALLED_APPS中安装Session应用

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'urldemo.apps.UrldemoConfig',
]
```

django中默认的session保存在数据库，迁移(makemigrations migrate)之后会默认生成django_session表



Session包括三个数据：键，值，过期时间



Session依赖于Cookie

所有请求者(三个不同用户)的Session都会存储在服务器中,启用Session后，会在Cookie中存储一个sessionid，每次请求时浏览器都会将这个数据发给服务器，服务器在接收到sessionid后，会根据这个值找出这个请求者的Session。 所以如果想要使用session，浏览器必须支持cookie,一般浏览器都会开启cookie，除非是人为关闭。

2.session操作

通过HttpRequest对象的Session属性进行会话的读写操作 以键值对的格式写会话

设置session值:

```
request.session['键']=值
```

根据键读取值:

```
request.session.get('键',默认值)
```

清除所有会话，删除值部分

```
request.session.clear()
```

删除会话中的指定键及值，在存储中只删除某个键及对应的值

```
del request.session['键']
```

设置会话的超时时间，如果没有指定过期时间则两周后过期

```
request.session.set_expiry(value)
```

如果value是一个整数，会话将在value秒没有活动后过期 如果value为0，那么用户会话的session将在用户的浏览器关闭时过期如果value为None，那么会话永不过期

在视图中设置session

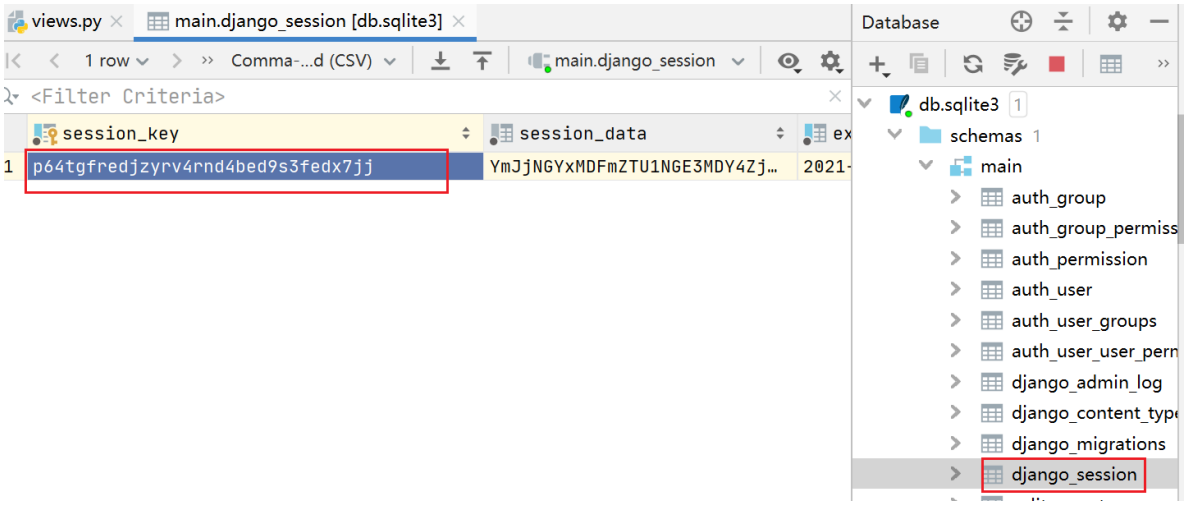
```
def set_session(request):
    """视图函数 ，测试session"""
    # 比如说 python，django，linux，java 都是 从前端用户通过表单传过来的信息
    request.session['h1'] = 'python' # 以键值对的方 式设置session
    request.session['h2'] = 'django'
    request.session['h3'] = 'linux'
    request.session['h4'] = 'java'
    return HttpResponse("设置session成功")
```

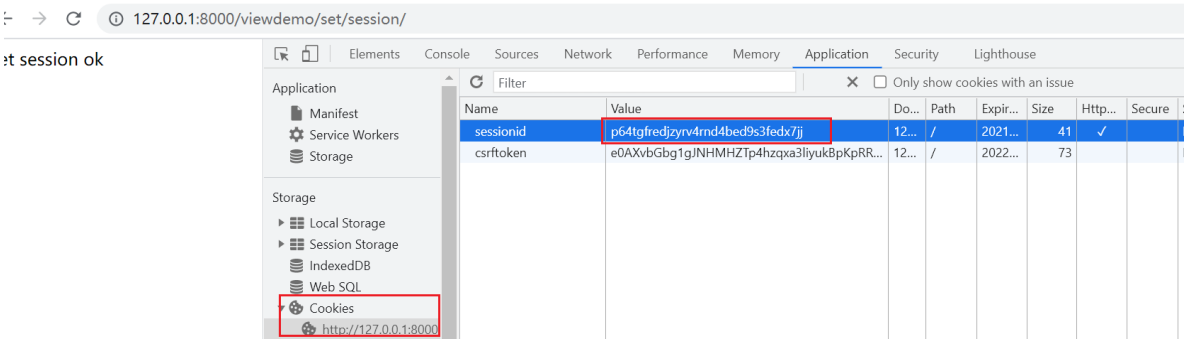
配置 url

```
path('set/session/', views.set_session),
```

浏览器访问：http://127.0.0.1:8000/blog/set/session/

可以看到cookie中保存的sessionid跟数据库中保存的session_key有一条相同的。就是这个会话保存在数据库中的 session





数据库中保存的session采用base64编码，解码之后可以看到session保存的键值对：

> 在线解码：https://base64.us/

2.5 类视图

虽然一个视图处理用户的一个请求，但HTTP提供了多种请求方式（GET、POST、PUT等），用户使用应用的某个功能时，该功能可能以任意一种方式发起请求，例如商品管理功能使用GET方式发起的呈现商品列表的请求、使用POST方式发起的修改商品请求等。此时视图需要结合条件分支，对每种请求方式分别进行处理。然而若所有请求方式的处理逻辑都定义在同一个视图中，视图很可能庞大且臃肿。为了解决这一问题，Django设计了类视图

2.5.1 定义及使用类视图

以函数的方式定义的视图称为**函数视图**，函数视图便于理解。但是遇到一个视图对应的路径提供了多种不同HTTP请求方式的支持时（get，post，put，delete...），便需要在一个函数中编写不同的业务逻辑，代码可读性与复用性都不佳。

/viewsdemo/regist

类视图允许在views.py的一个类中定义不同的方法，以处理同一功能以不同请求方式发送的请求

```
from django.views import View
def class_view(request):
    """不同请求的视图函数"""
    if request.method == 'GET':
        return HttpResponse("Get request")
    elif request.method == "POST":
        return HttpResponse("POST request")
# 类视图
class ClassView(View):
    """不同请求的类视图"""
    def get(self, request):
        return HttpResponse("类视图，get")
    def post(self, request):
        return HttpResponse("类视图，post")
```

配置路由时，使用类视图的 `as_view()` 方法来添加。

```
urlpatterns = [
    # 视图函数路由
    #path('classview/', views.class_view),
    # 类视图路由注册
    path('classview/', views.ClassView.as_view()),
]
```

可结合表单注册来测试

2.5.2 通用类视图

在开发网站的过程中，有一些视图函数虽然处理的对象不同，但是其大致的代码逻辑是一样的。比如一个博客和一个 论坛，通常其首页都是展示一系列的文章列表或者帖子列表。对处理首页的视图函数来说，虽然其处理的对象一个是 文章，另一个是帖子，但是其处理的过程是非常类似的。先从数据库取出文章或者帖子列表，然后将这些数据传递给 模板并渲染模板。 Django 把这些相同的逻辑代码抽取了出来，写成了一系列的通用的类视图。类视图被封装在 django.views.generic 包中

通用视图是视图开发中常见任务的抽象，使用通用视图，只需编写少量代码，便可快速开发数据的公共视图

各类通用类视图

各类通用视图

分类	内容	说明
基础视图	View TemplateView RedirectView	包含Django视图所需的大部分功能，既是通用视图，也是其它类的父视图；既可以被继承，也可以单独使用。
通用显示视图	DetailView ListView	用于显示数据，是许多项目中最常用的视图。
通用编辑视图	FormView CreateView UpdteView DeleteView	分别用于显示表单、创建表单、编辑已有表单和删除表单。包含了SuccessMessageMixin，这有助于呈现成功提交表单的消息。
通用日期视图	ArchiveIndexView YearArchiveView MonthArchiveView WeekArchiveView DayArchiveView TodayArchiveView DateDetailView	用于呈现基于日期的深层页面数据的视图，依次表示按日期显示“最新”对象的顶级索引页面、每年存档页面、每月存档页面、每周存档页面、每日存档页面、当天存档页面和单个对象的页面。

将模型作为额外参数传递给通用视图，可以快速开发视图。除了额外接收模型外，基于通用视图的开发流程与基于基础视图的开发流程基本相同

创建模型

```
class Publisher(models.Model):
    name = models.CharField(max_length=30)
    address = models.CharField(max_length=50)
    city = models.CharField(max_length=60)
    state_province = models.CharField(max_length=30)
    country = models.CharField(max_length=50)
    website = models.URLField()
    class Meta:
        ordering = ["-name"]
    def __str__(self):
        return self.name


# views.py
from django.views.generic import ListView
from .models import Publisher
class PublisherList(ListView):
    model = Publisher
```

路由：

```
urlpatterns = [
    path('publishers/', PublisherList.as_view()),
]
```

模板页

```
<h2>Publishers</h2><hr>
<ul>
    {% for publisher in object_list %}
        <li>{{ publisher.name }}</li>
    {% endfor %}
</ul>
```

设置上下文数据

通用视图通过get_context_data()方法返回嵌入模板中的上下文字典，重写该方法

```
def get_context_data(self, **kwargs):
        context = super(). get_context_data(**kwargs)
        # 将书籍信息的查询集加入上下文对象
        context['object_list'] = PublisherList.objects.all()
        return context
```

2.6 错误视图

Django内置处理HTTP错误的视图，主要错误及视图包括

- 404错误：page not found视图
- 500错误：server error视图
- 400错误：bad request视图
- 403错误：permission_denied视图

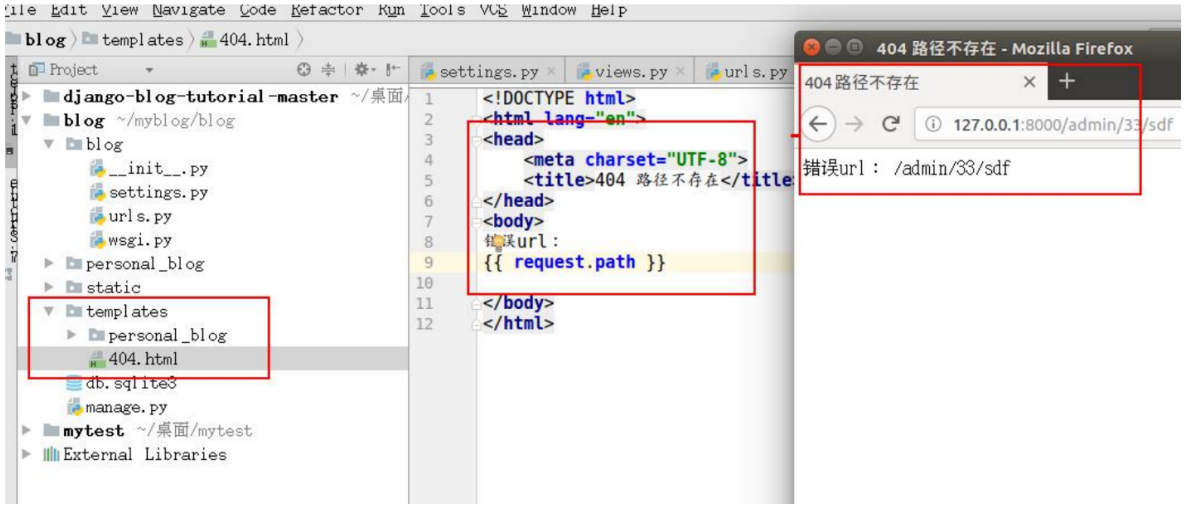
如果想要看到错误视图，而不是错误调试信息的需要设置setting.py配置文件的调试开关设置为False，ALLOWED_HOSTS允许所有主机访问。

```
DEBUG = Fasle
ALLOWED_HOSTS = ["*",]
```

404错误及视图

- 将请求地址进行url匹配后，没有找到匹配的正则表达式，则调用404视图，这个视图会调用404.html的模板 进行渲染
- 404视图默认情况下定义在'django.views.defaults.page_not_found'
- django 自带404模板
- 视图传递变量request_path给模板，是导致错误的URL

自定义错误模板： 在项目的templates目录下创建一个名字叫404.html的模板文件： 当发生404错误时，404 视图会调用templates中的模板文件而不再调用自带的模板。



其他错误处理方式相同。

三、模型与数据库

1 数据库介绍：

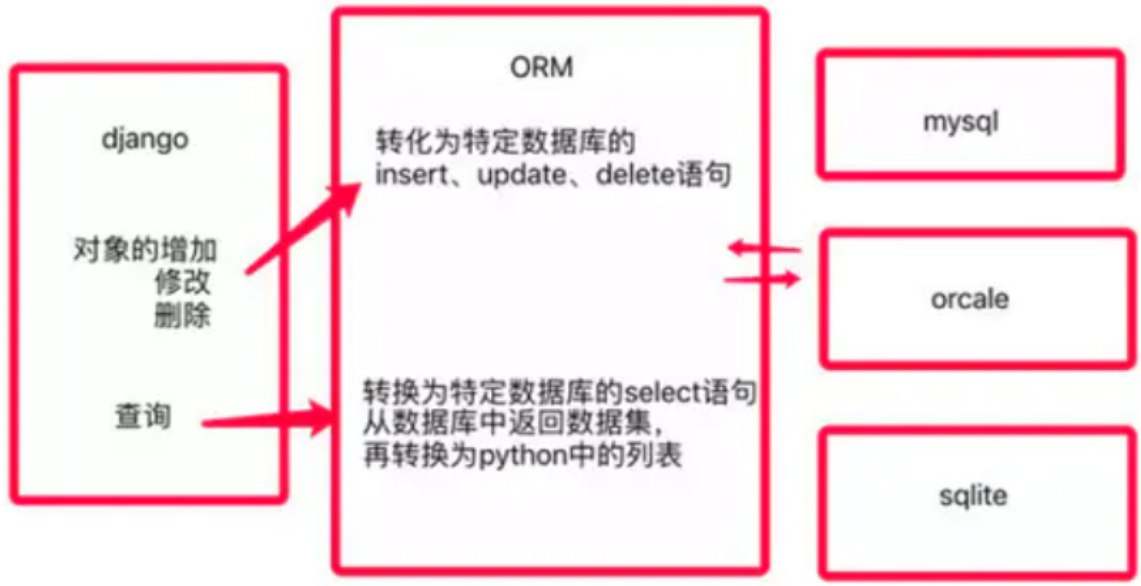
数据库分类: 关系型数据库(mysql,sqlite,oracle，db2)，sql语句

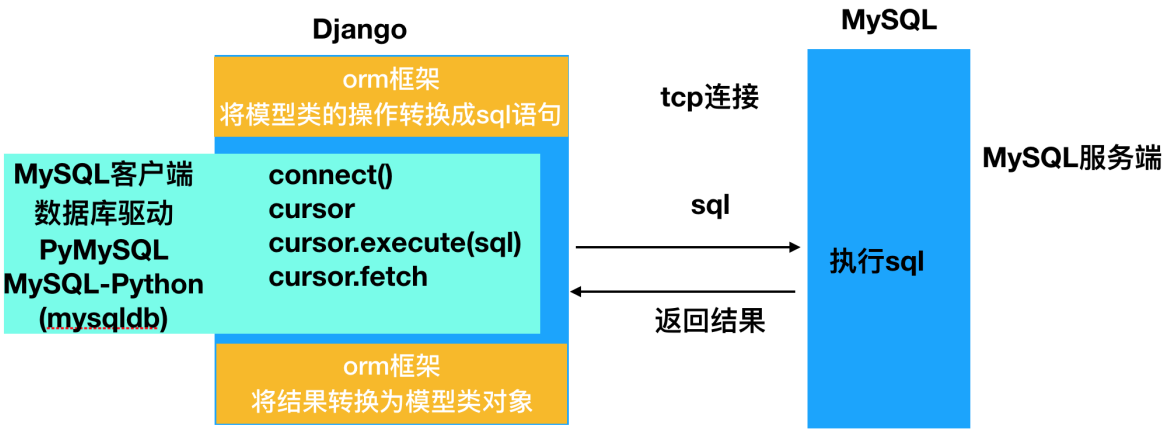
1) ORM框架

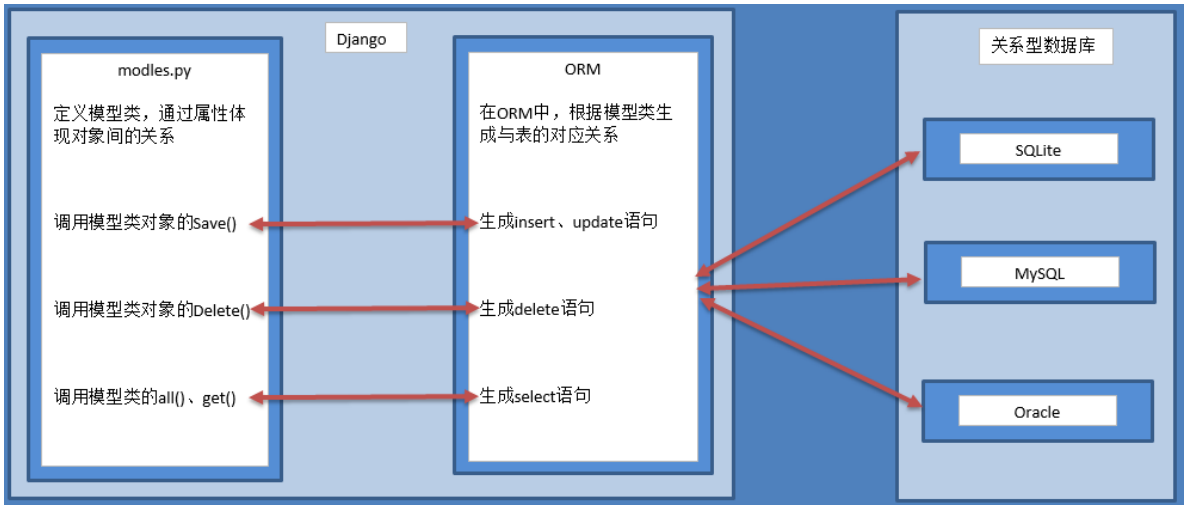
O是object，也就是**对象**的意思，R是relation，翻译成中文是关系，也就是关系数据库中**数据表**的意思，M是mapping，是**映射**的意思。在ORM框架中，它帮我们把类和数据表进行了一个映射，可以让我们**通过类和类对象就能操作它所对应的表中的数据。**ORM框架还有一个功能，它可以**根据我们设计的类自动帮我们生成数据库中的表**，省去了我们自己建表的过程。

django中内嵌了ORM框架，不需要直接面向数据库编程，而是定义模型类，通过模型类和对象完成数据表的增删改查操作。

2) ORM作用







使用django进行数据库开发的步骤如下：

1. 配置数据库连接信息
2. 在models.py中定义模型类
3. 迁移
4. 通过类和对象完成数据增删改查操作: 而不用在写 繁琐的sql语句

2 配置

在settings.py中保存了数据库的连接配置信息，Django默认初始配置使用**sqlite**数据库。

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

Sqlite数据库配置

MySQL数据库配置:

windows中 安装mysql

1. 使用**MySQL**数据库首先需要安装驱动程序

```
pip install PyMySQL
```

2. 在Django的工程同名子目录的__init__.py文件中添加如下语句 ,做向下兼容

```
from pymysql import install_as_MySQLdb

install_as_MySQLdb()
```

作用是让Django的ORM能以mysqldb的方式来调用PyMySQL

3. 修改**DATABASES**配置信息

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'HOST': '127.0.0.1',  # 数据库主机
        'PORT': 3306,  # 数据库端口
        'USER': 'root',  # 数据库用户名
        'PASSWORD': '1234',  # 数据库用户密码
        'NAME': 'db_books'  # 数据库名字
    }
}
```

4. 在MySQL中创建数据库

```
create database db_books default charset=utf8;
```

3 定义模型类

- 模型类被定义在"应用/models.py"文件中。
- 模型类必须继承自Model类，位于包django.db.models中。

接下来首先以"图书-英雄"管理为例进行演示

3.1 定义

创建子应用modelsdemo，并且注册子应用， 且在models.py 文件中定义模型类。

```
from django.db import models

#定义图书模型类BookInfo
class BookInfo(models.Model):
    btitle = models.CharField(max_length=20, verbose_name='名称')
    bpub_date = models.DateField(verbose_name='发布日期')
    bread = models.IntegerField(default=0, verbose_name='阅读量')
    bcomment = models.IntegerField(default=0, verbose_name='评论量')
    is_delete = models.BooleanField(default=False, verbose_name='逻辑删除')

    class Meta:
        db_table = 'tb_books'  # 指明数据库表名
        verbose_name = '图书'  # 在admin站点中显示的名称
        verbose_name_plural = verbose_name  # 显示的复数名称

    def __str__(self):
        """定义每个数据对象的显示信息"""
        return self.btitle

#定义英雄模型类HeroInfo
class HeroInfo(models.Model):
    GENDER_CHOICES = (
        (0, 'female'),
        (1, 'male')
    )
    hname = models.CharField(max_length=20, verbose_name='名称' )
    hgender = models.SmallIntegerField(choices=GENDER_CHOICES, default=0,
verbose_name='性别')
```

```
    hcomment = models.CharField(max_length=200, null=True, verbose_name='描述信息')
    hbook = models.ForeignKey(BookInfo, on_delete=models.CASCADE, verbose_name='图书')  # 外键
    is_delete = models.BooleanField(default=False, verbose_name='逻辑删除')

    class Meta:
        db_table = 'tb_heros'
        verbose_name = '英雄'
        verbose_name_plural = verbose_name

    def __str__(self):
        return self.hname
```

1） 数据表名

模型类如果未指明表名，Django默认以 **小写app应用名_小写模型类名** 为数据库表名。
modelsdemo_bookinfo

可通过**db_table** 指明数据库表名。

2） 关于主键

django会为表创建自动增长的主键列，每个模型只能有一个主键列，如果使用选项设置某属性为主键列后django不会再创建自动增长的主键列。

默认创建的主键列属性为id，可以使用pk代替，pk全拼为primary key。

3） 属性命名限制

- 不能是python的保留关键字。
- 不允许使用连续的下划线，这是由django的查询方式决定的。
- 定义属性时需要指定字段类型，通过字段类型的参数指定选项，语法如下：

```
属性=models.字段类型(选项)
```

4）字段类型

类型	说明
AutoField	自动增长的IntegerField，通常不用指定，不指定时Django会自动创建属性名为id的自动增长属性
BooleanField	布尔字段，值为True或False
NullBooleanField	支持Null、True、False三种值
CharField	字符串，参数max_length表示最大字符个数
TextField	大文本字段，一般超过4000个字符时使用
IntegerField	整数
DecimalField	十进制浮点数， 参数max_digits表示总位数， 参数decimal_places表示小数位数
FloatField	浮点数
	日期， 参数auto_now表示每次保存对象时，自动设置该字段为当前时间，

类型	说明
DateField	用于"最后一次修改"的时间戳，它总是使用当前日期，默认为False； 参数auto_now_add表示当对象第一次被创建时自动设置当前时间，用于创建的时间戳，它总是使用当前日期，默认为False; 参数auto_now_add和auto_now是相互排斥的，组合将会发生错误
TimeField	时间，参数同DateField
DateTimeField	日期时间，参数同DateField
FileField	上传文件字段
ImageField	继承于FileField，对上传的内容进行校验，确保是有效的图片

5） 选项

选项	说明
null	如果为True，表示允许为空，默认值是False
blank	如果为True，则该字段允许为空白，默认值是False
db_column	字段的名称，如果未指定，则使用属性的名称
db_index	若值为True, 则在表中会为此字段创建索引，默认值是False
default	默认
primary_key	若为True，则该字段会成为模型的主键字段，默认值是False，一般作为AutoField的选项使用
unique	如果为True, 这个字段在表中必须有唯一值，默认值是False

null是数据库范畴的概念，blank是表单验证范畴的

6） 外键

在设置外键时，需要通过**on_delete**选项指明主表删除数据时，对于外键引用表数据如何处理，在django.db.models中包含了可选常量：

- **CASCADE** 级联，删除主表数据时连通一起删除外键表中数据
- **PROTECT** 保护，通过抛出**ProtectedError**异常，来阻止删除主表中被外键应用的数据
- **SET_NULL** 设置为NULL，仅在该字段null=True允许为null时可用
- **SET_DEFAULT** 设置为默认值，仅在该字段设置了默认值时可用
- **DO_NOTHING** 不做任何操作，如果数据库前置指明级联性，此选项会抛出**IntegrityError**异常
- **SET()** 设置为特定值或者调用特定方法

3.2 数据迁移

将模型类同步到数据库中。

1）注册模型

```
#admin.py
from django.contrib import admin
from .models import *

admin.site.register(BookInfo)
admin.site.register(HeroInfo)
```

2）生成迁移文件

```
python manage.py makemigrations
```

3）同步到数据库中

```
python manage.py migrate
```

3.3 在mysql客户端添加测试数据

```
insert into tb_books(btitle,bpub_date,bread,bcomment,is_delete) values
('射雕英雄传','1980-5-1',12,34,0),
('天龙八部','1986-7-24',36,40,0),
('笑傲江湖','1995-12-24',20,80,0),
('雪山飞狐','1987-11-11',58,24,0);
insert into tb_heros(hname,hgender,hbook_id,hcomment,is_delete) values
('郭靖',1,1,'降龙十八掌',0),
('黄蓉',0,1,'打狗棍法',0),
('黄药师',1,1,'弹指神通',0),
('欧阳锋',1,1,'蛤蟆功',0),
('梅超风',0,1,'九阴白骨爪',0),
('乔峰',1,2,'降龙十八掌',0),
('段誉',1,2,'六脉神剑',0),
('虚竹',1,2,'天山六阳掌',0),
('王语嫣',0,2,'神仙姐姐',0),
('令狐冲',1,3,'独孤九剑',0),
('任盈盈',0,3,'弹琴',0),
('岳不群',1,3,'华山剑法',0),
('东方不败',0,3,'葵花宝典',0),
('胡斐',1,4,'胡家刀法',0),
('苗若兰',0,4,'黄衣',0),
('程灵素',0,4,'医术',0),
('袁紫衣',0,4,'六合拳',0);
```

4 数据库操作

提示： 进入Django shell 操作界面

- 在命令行输入

  python manage.py shell

- 执行Django shell

- 导入模型类

  from modelsdemo.models import *

4.1 增加操作

增加数据有两种方法。

1）save

通过创建模型类对象，执行对象的save()方法保存到数据库中。

```
>>> from datetime import date
>>> book = BookInfo(
    btitle='东游记',
    bpub_date=date(2013,1,1),
    bread=20,
    bcomment=20
)
>>> book.save()
>>> hero = HeroInfo(
    hname='孙悟空',
    hgender=0,
    hbook=book
)
>>> hero.save()
>>> hero2 = HeroInfo(
    hname='猪八戒',
    hgender=0,
    hbook_id=book.id
)
>>> hero2.save()
```

2）create

通过模型类.objects.create()保存。

```
>>> HeroInfo.objects.create(
    hname='如来佛祖',
    hgender=0,
    hbook_id=book.id
)
```

4.2 删除操作

删除有两种方法

1）模型类对象delete

```
hero = HeroInfo.objects.get(id=13)
hero.delete()
```

2）模型类.objects.filter().delete()

```
HeroInfo.objects.filter(id=14).delete()
```

4.3 修改操作

修改更新有两种方法

1）save

修改模型类对象的属性，然后执行save()方法

```
hero = HeroInfo.objects.get(hname='猪八戒')
hero.hname = '猪悟能'
hero.save()
```

2）update

使用模型类.objects.filter().update()，会返回受影响的行数

```
HeroInfo.objects.filter(hname='沙悟净').update(hname='沙僧')
```

4.4 查询操作

1. 基本查询

get 查询单一结果，如果不存在会抛出**模型类.DoesNotExist**异常。

all 查询多个结果。

count 查询结果数量。

```
>>> BookInfo.objects.all()
>>> book = BookInfo.objects.get(btitle='西游记')
>>> book.id
>>> BookInfo.objects.get(id=3)
>>> BookInfo.objects.get(pk=3)
>>> BookInfo.objects.get(id=100)
>>> BookInfo.objects.count()
```

2.过滤查询

实现SQL中的where功能，包括

- **filter** 过滤出多个结果
- **exclude** 排除掉符合条件剩下的结果
- **get** 过滤单一结果

对于过滤条件的使用，上述三个方法相同，故仅以**filter**进行讲解。

过滤条件的表达语法如下：

```
属性名称__比较运算符=值
# 属性名称和比较运算符间使用两个下划线，所以属性名不能包括多个下划线
```

1）相等

exact：表示判等。

例：查询编号为1的图书。

```
BookInfo.objects.filter(id__exact=1)
可简写为：
BookInfo.objects.filter(id=1)
```

2）模糊查询

contains：是否包含。

> 说明：如果要包含%无需转义，直接写即可。

例：查询heros中的名字包含'不'的人物。

```
HeroInfo.objects.filter(hname__contains='不')
```

startswith、endswith：以指定值开头或结尾。

例：查询书名以'部'结尾的图书

```
BookInfo.objects.filter(btitle__endswith='部')
```

> 以上运算符都区分大小写，在这些运算符前加上i表示不区分大小写，如iexact、icontains、istartswith、iendswith.

3） 空查询

isnull：是否为null。

例：查询书名不为空的图书。

```
BookInfo.objects.filter(btitle__isnull=False)
```

4） 范围查询

in：是否包含在范围内。

例：查询编号为1或3或5的图书

```
BookInfo.objects.filter(id__in=[1, 3, 5])
```

5）比较查询 数字类型　字段

- **gt** 大于 (greater then)
- **gte** 大于等于 (greater then equal)
- **lt** 小于 (less then)
- **lte** 小于等于 (less then equal)

例：查询编号大于3的图书

```
BookInfo.objects.filter(id__gt=3)
```

不等于的运算符，使用exclude()过滤器。

例：查询编号不等于3的图书

```
BookInfo.objects.exclude(id=3)
```

6）日期查询

year、month、day、week_day、hour、minute、second：对日期时间类型的属性进行运算。

例：查询1980年发表的图书。

```
BookInfo.objects.filter(bpub_date__year=1980)
```

例：查询1980年1月1日后发表的图书。

```
BookInfo.objects.filter(bpub_date__gt=date(1990, 1, 1))
```

3 F与Q对象

F对象

之前的查询都是对象的属性与常量值比较，两个属性（字段）怎么比较呢？ 答：使用F对象，被定义在django.db.models中。

语法如下：

```
F(属性名)
```

例：查询阅读量大于等于评论量的图书。

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

BookInfo.objects.filter(bread__gte=F('bcomment'))
```

可以在F对象上使用算数运算。

例：查询阅读量大于2倍评论量的图书。

```
BookInfo.objects.filter(bread__gt=F('bcomment') * 2)
```

Q对象

多个过滤器逐个调用表示逻辑与关系，同sql语句中where部分的and关键字。

例：查询阅读量大于20，并且编号小于3的图书。

```
BookInfo.objects.filter(bread__gt=20,id__lt=3)
或
BookInfo.objects.filter(bread__gt=20).filter(id__lt=3)
```

如果需要实现逻辑或or的查询，需要使用Q()对象结合|运算符，Q对象被义在django.db.models中。

```
Q(属性名__运算符=值)
```

例：查询阅读量大于20的图书，改写为Q对象如下。

```
from django.db.models import Q

BookInfo.objects.filter(Q(bread__gt=20))
```

Q对象可以使用&、|连接，&表示逻辑与，|表示逻辑或。

例：查询阅读量大于20，或编号小于3的图书，只能使用Q对象实现

```
BookInfo.objects.filter(Q(bread__gt=20) | Q(pk__lt=3))
```

Q对象前可以使用~操作符，表示非not。

例：查询编号不等于3的图书。

```
BookInfo.objects.filter(~Q(pk=3))
```

4. 聚合查询

1.聚合函数

使用aggregate()过滤器调用聚合函数。聚合函数包括：**Avg** 平均，**Count** 数量，**Max** 最大，**Min** 最小，**Sum** 求和，被定义在django.db.models中。

例：查询图书的总阅读量。

```
from modelsdemo.models import *
from django.db.models import *

BookInfo.objects.aggregate(Sum('bread'))
```

注意aggregate的返回值是一个字典类型，格式如下：

```
  {'属性名__聚合类小写':值}
  如:{'bread__sum':156}
```

使用count时一般不使用aggregate()过滤器。

例：查询图书总数。

```
BookInfo.objects.count()
```

注意count函数的返回值是一个数字。

2. 排序

使用**order_by**对结果进行排序

```
BookInfo.objects.all().order_by('bread')  # 升序
BookInfo.objects.all().order_by('-bread')  # 降序
```

5.关联查询

1 关联查询

由一到多的访问语法： BookInfo（一方）， HeroInfo(多方)

一对应的模型类对象.多对应的模型类名小写_set 例：

```
b = BookInfo.objects.get(id=1)
b.heroinfo_set.all()
```

由多到一的访问语法:

多对应的模型类对象.多对应的模型类中的关系类属性名 例：

```
h = HeroInfo.objects.get(id=1)
h.hbook
```

访问一对应的模型类关联对象的id语法:

多对应的模型类对象.关联类属性_id

例：

```
h = HeroInfo.objects.get(id=1)
h.hbook_id
```

2 关联过滤查询

由多模型类条件查询一模型类数据:

语法如下：

```
关联模型类名小写__属性名__条件运算符=值
```

注意：如果没有"__运算符"部分，表示等于。

例：

查询图书，要求图书英雄为"孙悟空"

```
BookInfo.objects.filter(heroinfo__hname='孙悟空')
```

查询图书，要求图书中英雄的描述包含"八"

```
BookInfo.objects.filter(heroinfo__hcomment__contains='八')
```

由一模型类条件查询多模型类数据:

语法如下:

```
一模型类关联属性名__一模型类属性名__条件运算符=值
```

注意：如果没有"__运算符"部分，表示等于。

例:

查询书名为“天龙八部”的所有英雄。

```
HeroInfo.objects.filter(hbook__btitle__exact='天龙八部')
```

查询图书阅读量大于30的所有英雄

```
HeroInfo.objects.filter(hbook__bread__gt=30)
```

5 查询集 QuerySet

1 概念

Django的ORM中存在查询集的概念。

查询集，也称查询结果集、QuerySet，表示从数据库中获取的对象集合。

当调用如下过滤器方法时，Django会返回查询集（而不是简单的列表）：

- all()：返回所有数据。
- filter()：返回满足条件的数据。
- exclude()：返回满足条件之外的数据。
- order_by()：对结果进行排序。

对查询集可以再次调用过滤器进行过滤，如

```
BookInfo.objects.filter(bread__gt=30).order_by('bpub_date')
```

也就意味着查询集可以含有零个、一个或多个过滤器。过滤器基于所给的参数限制查询的结果。

从SQL的角度讲，查询集与select语句等价，过滤器像where、limit、order by子句。

判断某一个查询集中是否有数据:

- exists()：判断查询集中是否有数据，如果有则返回True，没有则返回False。

2 两大特性

1）惰性执行

创建查询集不会访问数据库，直到调用数据时，才会访问数据库，调用数据的情况包括迭代、序列化、与if合用

例如，当执行如下语句时，并未进行数据库查询，只是创建了一个查询集qs

```
qs = BookInfo.objects.all()
```

继续执行遍历迭代操作后，才真正的进行了数据库的查询

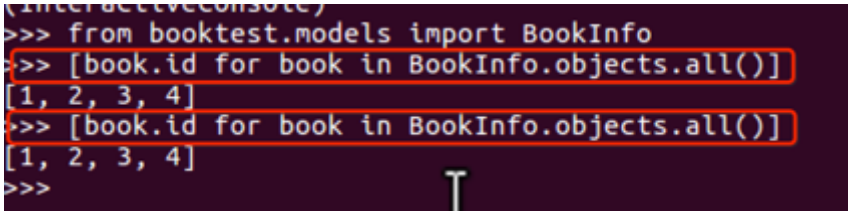
```
for book in qs:
    print(book.btitle)
```

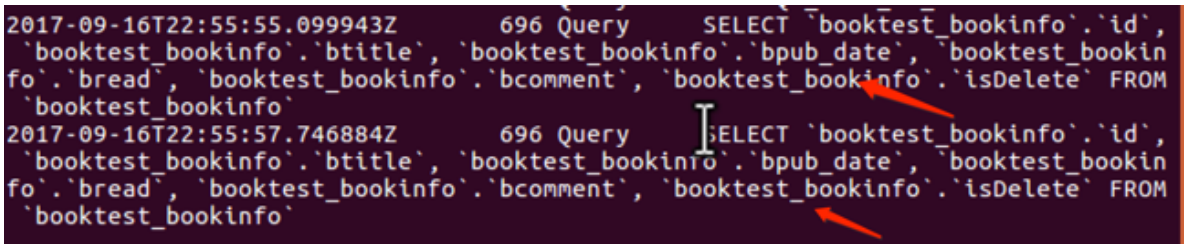
2）缓存

使用同一个查询集，第一次使用时会发生数据库的查询，然后Django会把结果缓存下来，再次使用这个查询集时会使用缓存的数据，减少了数据库的查询次数。

情况一：如下是两个查询集，无法重用缓存，每次查询都会与数据库进行一次交互，增加了数据库的负载。

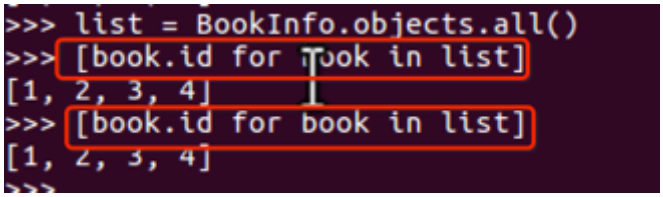
```
from booktest.models import BookInfo
[book.id for book in BookInfo.objects.all()]
[book.id for book in BookInfo.objects.all()]
```

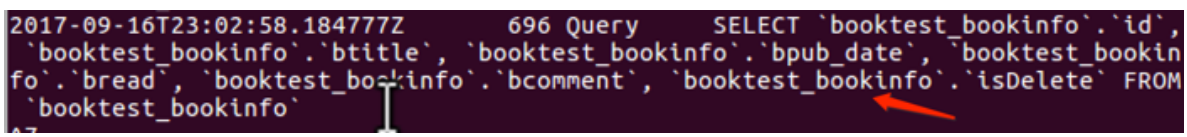




情况二：经过存储后，可以重用查询集，第二次使用缓存中的数据。

```
qs=BookInfo.objects.all()
[book.id for book in qs]
[book.id for book in qs]
```





3 限制查询集

可以对查询集进行取下标或切片操作，等同于sql中的limit和offset子句。

> 注意：不支持负数索引。

对查询集进行切片后返回一个新的查询集，不会立即执行查询。

如果获取一个对象，直接使用[0]，等同于[0:1].get()，但是如果没有数据，[0]引发IndexError异常，[0:1].get()如果没有数据引发DoesNotExist异常。

示例：获取第1、2项，运行查看。

```
qs = BookInfo.objects.all()[0:2]
```

6 执行原生SQL语句

管理器介绍

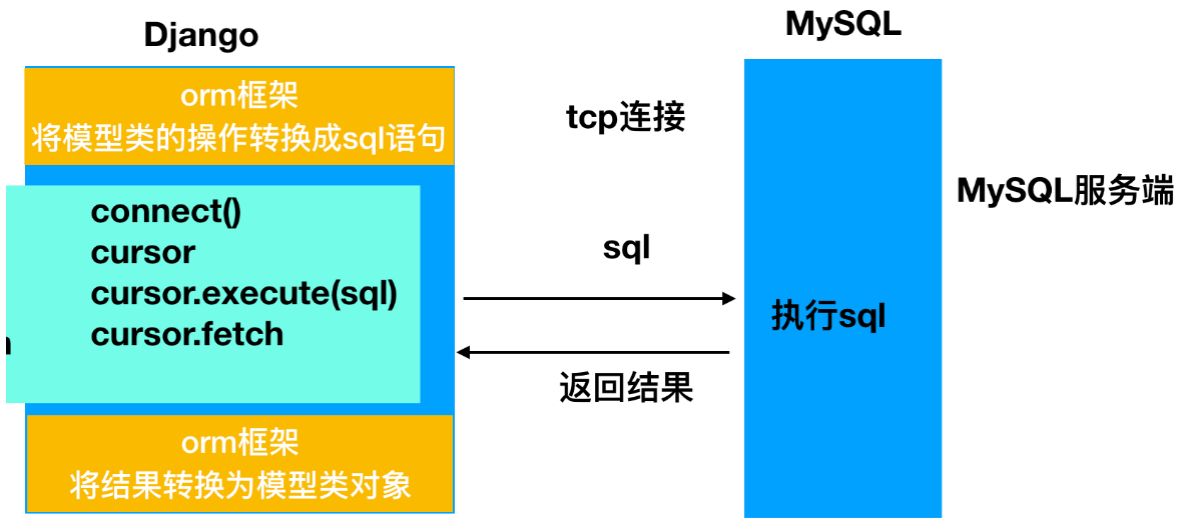
管理器是Django的模型进行数据库操作的接口，Django应用的一个模型类都拥有至少一个管理器。

我们在通过模型类的**objects**属性提供的方法操作数据库时，即是在使用一个管理器对象objects。当没有为模型类定义管理器时，Django会为每一个模型类生成一个名为objects的管理器，它是**models.Manager**类的对象。

1）**使用管理器执行原生SQL语句**

Manager.raw(raw_query, params=None, translations=None)

2）**使用 connection.cursor()执行原生SQL**



视图views.py

```
from django.shortcuts import render
from django.http import HttpResponse
from .models import *
from django.db import connection
# Create your views here.
def index(request):
    """模型与数据库测试"""
    # 1.使用管理器执行原生sql
    # qs = BookInfo.objects.raw("select * from tb_books")
    # for item in qs:
    #     print(item)
    #     print(item.__dict__)

    # 2.使用连接游标执行原生sql
    with connection.cursor() as cur:
        cur.execute('select * from tb_books')
```

```
        row = cur.fetchone()
        #rows = cur.fetchall()
        print(row)
    return HttpResponse("模型与数据库")
```

路由urls.py

```
from django.urls import path
from . import views

# 根路由的配置
urlpatterns = [
    path('index/', views.index),
]
```