1. **详细解释什么是Flask？说一说这个框架的优点？**

Flask 是一个 web 框架。也就是说 Flask 为你提供工具，库和技术来允许你构建一个 web 应用程序。这个 wdb 应用程序可以使一些 web 页面、博客、wiki、基于 web 的日历应用或商业网站。

Flask 属于微框架（micro-framework）这一类别，微架构通常是很小的不依赖于外部库的框架。这既有优点也有缺点，优点是框架很轻量，更新时依赖少，并且专注安全方面的 bug，缺点是，你不得不自己做更多的工作，或通过添加插件增加自己的依赖列表。Flask 的依赖如下：

[Werkzeug](http://werkzeug.pocoo.org/" \t "/home/anwc/文档\\x/_blank) 一个 WSGI 工具包

[jinja2](http://jinja.pocoo.org/" \t "/home/anwc/文档\\x/_blank) 模板引擎

1. **Flask的默认依赖库是什么？简单介绍介绍**

Flask 的依赖如下：

[Werkzeug](http://werkzeug.pocoo.org/" \t "/home/anwc/文档\\x/_blank) 一个 WSGI 工具包

[jinja2](http://jinja.pocoo.org/" \t "/home/anwc/文档\\x/_blank) 模板引擎

Werkzeug是一个WSGI工具包，他可以作为一个Web框架的底层库。这里稍微说一下， werkzeug 不是一个web[服务器](https://www.baidu.com/s?wd=%E6%9C%8D%E5%8A%A1%E5%99%A8&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd" \t "/home/anwc/文档\\x/_blank)，也不是一个web框架，而是一个工具包，官方的介绍说是一个 WSGI 工具包，它可以作为一个 Web 框架的底层库，因为它封装好了很多 Web 框架的东西，例如 Request，Response 等等。

Jinja2是Python下一个被广泛应用的模版引擎，他的设计思想来源于[Django](https://baike.baidu.com/item/Django/61531" \t "/home/anwc/文档\\x/_blank)的模板引擎，并扩展了其语法和一系列强大的功能。其中最显著的一个是增加了沙箱执行功能和可选的自动转义功能，这对大多应用的安全性来说是非常重要的。

1. **Django和Flask的区别在哪？**

Flask

Flask确实很“轻”，不愧是Micro Framework，从Django转向Flask的开发者一定会如此感慨，除非二者均为深入使用过

Flask自由、灵活，可扩展性强，第三方库的选择面广，开发时可以结合自己最喜欢用的轮子，也能结合最流行最强大的Python库

入门简单，即便没有多少web开发经验，也能很快做出网站

非常适用于小型网站

非常适用于开发web服务的API

开发大型网站无压力，但代码架构需要自己设计，开发成本取决于开发者的能力和经验

各方面性能均等于或优于Django

Django自带的或第三方的好评如潮的功能，Flask上总会找到与之类似第三方库

Flask灵活开发，Python高手基本都会喜欢Flask，但对Django却可能褒贬不一

Flask与关系型数据库的配合使用不弱于Django，而其与NoSQL数据库的配合远远优于Django

Flask比Django更加Pythonic，与Python的philosophy更加吻合

Django

Django太重了，除了web框架，自带ORM和模板引擎，灵活和自由度不够高

Django能开发小应用，但总会有“杀鸡焉用牛刀”的感觉

Django的自带ORM非常优秀，综合评价略高于SQLAlchemy

Django自带的模板引擎简单好用，但其强大程度和综合评价略低于Jinja

Django自带ORM也使Django与关系型数据库耦合度过高，如果想使用MongoDB等NoSQL数据，需要选取合适的第三方库，且总感觉Django+SQL才是天生一对的搭配，Django+NoSQL砍掉了Django的半壁江山

Django目前支持Jinja等非官方模板引擎

Django自带的数据库管理app好评如潮

Django非常适合企业级网站的开发：快速、靠谱、稳定

Django成熟、稳定、完善，但相比于Flask，Django的整体生态相对封闭

Django是Python web框架的先驱，用户多，第三方库最丰富，最好的Python库，如果不能直接用到Django中，也一定能找到与之对应的移植

Django上手也比较容易，开发文档详细、完善，相关资料丰富

1. **Flask-WTF是什么，有什么特点？**

是一个关于表单的扩展库，可以自动生成表单的HTML代码和验证提交的表单数据，并且提供跨站请求伪造（Cross-Site Request Forgery）保护的功能，使用非常方便。

安装：pip install flask-wtf

使用：

1. 定义表单类:

# 设置秘钥  
app.config[**'SECRET\_KEY'**] = **'123456'**​  
# 导入表单基类  
from flask\_wtf import FlaskForm  
# 导入相关字段  
from wtforms import StringField, SubmitField  
# 导入相关验证器类  
from wtforms.validators import DataRequired  
​  
class NameForm(FlaskForm):  
    name = StringField(**'用户名'**, validators=[DataRequired()])  
    submit = SubmitField(**'提交'**)

1. 添加视图函数:

@app.route(**'/'**)  
def index():  
 # 创建表单对象  
 form = NameForm()  
 # 渲染时分配到模板文件  
 return render\_template(**'form.html'**, form=form)

1. 表单渲染

{% if form.name.errors %}  
   {{ form.name.errors[0] }}  
{% endif %}  
<h1>Hello {{ name }}</h1>  
<form action=**"{{ url\_for('wtform') }}"** method=**"post"** >  
 # {{form.hidden\_tag()}}里面放的也就是csrf\_token  
   {{ form.csrf\_token }}  
   {{ form.name.label() }}{{ form.name(id=**'xx'**,class=**'yyy'**) }}  
   {{ form.submit() }}  
</form>  
​  
{# bootstrap渲染 #}  
{% extends **'bootstrap/base.html'** %}  
{% import **'bootstrap/wtf.html'** as wtf %}  
{% block content %}  
    <div class=**"container"**>{{ wtf.quick\_form(form) }}</div>  
{% endblock %}

1. 表单校验

@app.route(**'/'**, methods=[**'GET'**, **'POST'**])  
def index():  
    # 创建表单对象  
    form = NameForm()  
    name = None  
    # 表单校验  
    if form.validate\_on\_submit():  
        # 通过校验，则取出数据  
        name = form.name.data  
        form.name.data = **''**    # 渲染时分配到模板文件  
    return render\_template(**'form.html'**, form=form, name=name)

1. POST重定向GET解决重复提交的问题

@app.route(**'/wtform/'**, methods=[**'GET'**,**'POST'**])  
def wtform():  
    form = MyForm()  
    # 判断form表单中的数据是否通过校验  
    if form.validate\_on\_submit():  
        # 通过校验，则取出数据。  
        name = form.name.data  
        last\_name = session.get(**'name'**) or None  
        # 判断这次提交的name是否和上次的相等  
        if last\_name and last\_name == name:  
            return **'老铁别重复提交了'**        else:  
            # 不是重复提交，设置session  
            session[**'name'**] =  name  
            return render\_template(**'wtform.html'**,form=form, name=name)  
    return render\_template(**'wtform.html'**, form=form)

常见字段类型:



常见验证器类:



1. **如何在Flask中访问会话？**

flask\_session是flask框架实现session功能的一个插件，用来替代flask自带的session实现机制。

配置参数详解:

SESSION\_COOKIE\_NAME

设置返回给客户端的cookie的名称，默认是“session”;放置在response的头部；

SESSION\_COOKIE\_DOMAIN

设置会话的域，默认是当前的服务器，因为Session是一个全局的变量，可能应用在多个app中；

SESSION\_COOKIE\_PATH

设置会话的路径，即哪些路由下应该设置cookie，如果不设置，那么默认为‘/’，所有的路由都会设置cookie；

SESSION\_COOKIE\_HTTPONLY

cookie应该和httponly标志一起设置，默认为True，这个一般采用默认。

SESSION\_COOKIE\_SECURE

cookie是否和安全标志一起设置，默认为false，这个一般采用默认。

PERMANENT\_SESSION\_LIFETIME

设置session的有效期，即cookie的失效时间，单位是s。这个参数很重要，因为默认会话是永久性的。

SESSION\_TYPE

设置session保存的位置，可以有多种配置，

SESSION\_TYPE = ‘null’ : 采用flask默认的保存在cookie中；

SESSION\_TYPE = ‘redis’ : 保存在redis中

SESSION\_TYPE = ‘memcached’ : 保存在memcache

SESSION\_TYPE = 'filesystem' : 保存在文件

SESSION\_TYPE = 'mongodb' : 保存在MongoDB

SESSION\_TYPE = 'sqlalchemy' : 保存在关系型数据库

SESSION\_PERMANENT

是否使用永久会话，默认True，但是如果设置了PERMANENT\_SESSION\_LIFETIME，则这个失效；

SESSION\_USE\_SIGNER

是否为cookie设置签名来保护数据不被更改，默认是False；如果设置True,那么必须设置flask的secret\_key参数；

1. **Flask是MVC模型吗？请详细解释一下**

不是,flask采用的是MTV模式

1. **Flask中的数据库连接有哪两种方式？**

flask中是没有ORM的，如果在flask里面连接数据库有两种方式

1.pymysql

2.SQLAlchemy

是python 操作数据库的一个库。能够进行 orm 映射官方文档 sqlchemy

SQLAlchemy“采用简单的Python语言，为高效和高性能的数据库访问 设计，实现了完整的企业级持久模型”。SQLAlchemy的理念是，SQL数据库的 量级和性能重要于对象集合；而对象集合的抽象又重要于表和行。

**数据库链接池原理**

BDUtils数据库链接池

模式一：基于threaing.local实现为每一个线程创建一个连接，关闭是伪关闭，当前线程可以重复

模式二：连接池原理

可以设置连接池中最大连接数:9

默认启动时，连接池中创建连接:5

如果有三个线程来数据库中获取连接：

如果三个同时来的，一人给一个链接

如果一个一个来，有时间间隔，用一个链接就可以为三个线程提供服务

说不准

有可能:1个链接就可以为三个线程提供服务

有可能:2个链接就可以为三个线程提供服务

有可能:3个链接就可以为三个线程提供服务

PS、：maxshared在使用pymysql中均无用。链接数据库的模块：只有threadsafety>1的时候才有用

1. **列举常见的http请求状态码并说明其含义**

1XX ：信息状态码

　　100 Continue 继续，一般在发送 post 请求时，已发送了 http header 之后服 务端 将返回此信息，表示确认，之后发送具体参数信息

2XX ：成功状态码

　　200 OK 正常返回信息 201 Created 请求成功并且服务器创建了新的资源

　　202 Accepted 服务器已接受请求，但尚未处理

3XX ：重定向

　　301 Moved Permanently 请求的网页已永久移动到新位置。

　　302 Found 临时性重定向。 303 See Other 临时性重定向，且总是使用 GET 请 求新的 URI 。

　　304 Not Modified 自从上次请求后，请求的网页未修改过。

4XX ：客户端错误

　　400 Bad Request 服务器无法理解请求的格式，客户端不应当尝试再次使用相同 的 内容发起请求。

　　401 Unauthorized 请求未授权。

　　403 Forbidden 禁止访问。

　　404 Not Found 找不到如何与 URI 相匹配的资源。

5XX: 服务器错误

　　500 Internal Server Error 最常见的服务器端错误。

　　503 Service Unavailable 服务器端暂时无法处理请求（可能是过载或维护）。

1. **Flask框架依赖的组件有哪些？每个组件的功能是什么**

Route(路由)：程序实例需要知道每个url请求所对应的运行代码是谁。所以程序中必须要创建一个 url 请求地址 到 python运行函数的一个映射。处理url和函数之间的关系的程序就是"路由"。在 Flask 中，路由是通过 @app.route 装饰器来表示的

templates(模板)

Models(orm模型)

blueprint(蓝图)

Jinja2模板引擎：是Python Web编程中的主流模板语言

1. **Flask蓝图的作用是什么？**

蓝图Blueprint实现模块化的应用

book\_bp = Blueprint('book', \_\_name\_\_）创建蓝图对象

蓝图中使用路由@book\_bp.route('url')

在另一.py文件里导入和注册蓝图from book import book\_bp app.register\_blueprint(book\_bp)

1. **你使用过哪些Flask的第三方插件？简要说明**

flask\_bootstrap

flask-WTF

flask\_sqlalchemy

1. **Flask框架的默认session处理机制是什么？**

Flask的默认session利用了Werkzeug的SecureCookie，把信息做序列化(pickle)后编码(base64)，放到cookie里了。

过期时间是通过cookie的过期时间实现的。

为了防止cookie内容被篡改，session会自动打上一个叫session的hash串，这个串是经过session内容、SECRET\_KEY计算出来的，看得出，这种设计虽然不能保证session里的内容不泄露，但至少防止了不被篡改

1. **ORM的实现原理详细说明一下**

概念： 对象关系映射（Object Relational Mapping，简称ORM，或O/RM，或O/R mapping），是一种程序技术，用于实现面向对象编程语言里不同类型系统的数据之间的转换。

详细介绍： 让我们从O/R开始。字母O起源于”对象”(Object),而R则来自于”关系”(Relational)。几乎所有的程序里面，都存在对象和关系数据库。在业务逻辑层和用户界面层中，我们是面向对象的。当对象信息发生变化的时候，我们需要把对象的信息保存在关系数据库中。

当你开发一个应用程序的时候(不使用O/R Mapping),你可能会写不少数据访问层的代码，用来从数据库保存，删除，读取对象信息，等等。你在DAL中写了很多的方法来读取对象数据，改变状态对象等等任务。而这些代码写起来总是重复的。

ORM解决的主要问题是对象关系的映射。域模型和关系模型分别是建立在概念模型的基础上的。域模型是面向对象的，而关系模型是面向关系的。一般情况下，一个持久化类和一个表对应，类的每个实例对应表中的一条记录，类的每个属性对应表的每个字段。

1. **ORM技术有什么特点？**

ORM技术特点：

1.提高了开发效率。由于ORM可以自动对Entity对象与数据库中的Table进行字段与属性的映射，所以我们实际可能已经不需要一个专用的、庞大的数据访问层。

2.ORM提供了对数据库的映射，不用sql直接编码，能够像操作对象一样从数据库获取数据。

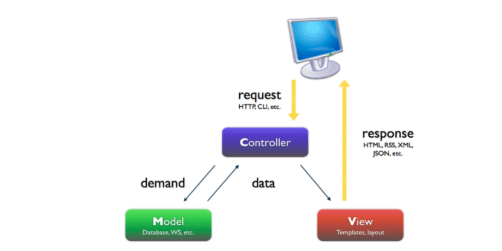
1. **你了解什么叫restful么？**

Flask-RESTful 是一个 Flask 扩展，它添加了快速构建 REST APIs 的支持。它当然也是一个能够跟你现有的ORM/库协同工作的轻量级的扩展。

简述MVC模式和MTV模式

著名的MVC模式：所谓MVC就是把web应用分为模型(M),控制器(C),视图(V)三层；他们之间以一种插件似的，松耦合的方式连接在一起。

模型负责业务对象与数据库的对象(ORM),视图负责与用户的交互(页面)，控制器(C)接受用户的输入调用模型和视图完成用户的请求。



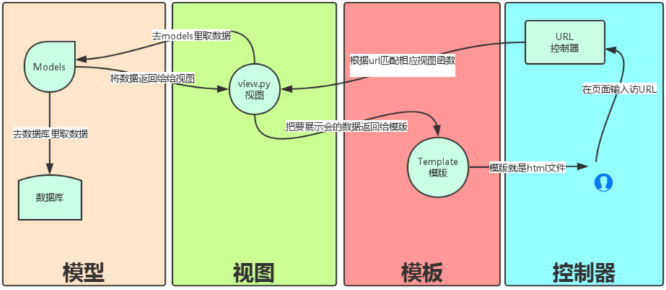
Django的MTV模式本质上与MVC模式没有什么差别，也是各组件之间为了保持松耦合关系，只是定义上有些许不同，Django的MTV分别代表：

 Model(模型)：负责业务对象与数据库的对象(ORM)

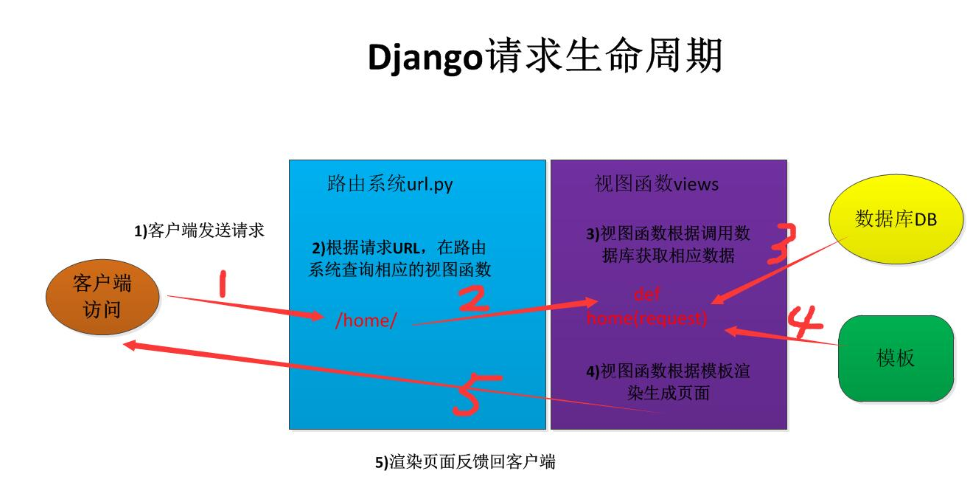
 Template(模版)：负责如何把页面展示给用户

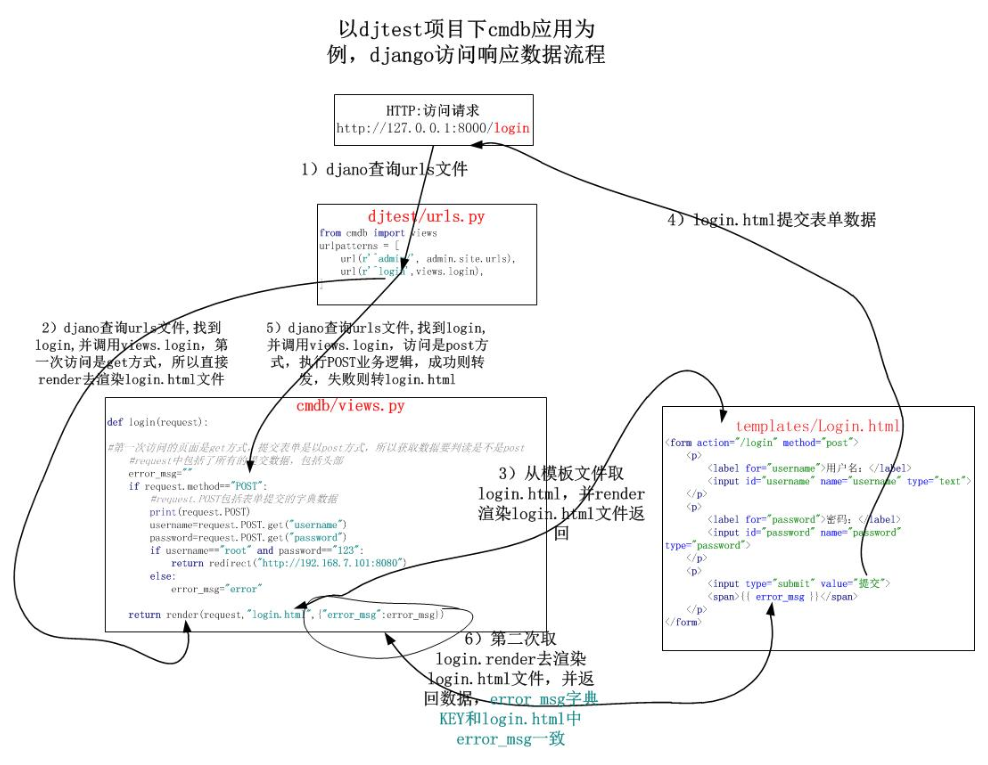
 View(视图)：负责业务逻辑，并在适当的时候调用Model和Template

 此外，Django还有一个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



1. **简述一下Django请求的生命周期**

****

****

1. **简单说明什么是FBV和CBV**

FBV:就是在视图里使用函数处理请求

看代码：

Urls.py

from django.conf.urls import url, include  
# from django.contrib import admin  
from mytest import views  
  
urlpatterns = [  
 # url(r‘^admin/‘, admin.site.urls),  
 url(r‘ ^ index /‘, views.index),  
]

Views.py

from django.shortcuts import render  
  
  
def index(req):  
 if req.method == ‘POST‘:  
 print(‘method is:‘ + req.method)  
 elif req.method == ‘GET‘:  
 print(‘method is:‘ + req.method)  
 return render(req, ‘index.html‘)

Index.html

<!DOCTYPE html>  
<html lang=**"en"**>  
<head>  
 <meta charset=**"UTF-8"**>  
 <title>index</title>  
</head>  
<body>  
 <form action=**""** method=**"post"**>  
 <input type=**"text"** name=**"A"** />  
 <input type=**"submit"** name=**"b"** value=**"提交"** />  
 </form>  
</body>  
</html>

CBV:就是在视图里使用类处理请求

Urls.py

from mytest import views  
  
urlpatterns = [  
 # url(r‘^index/‘, views.index),  
 url(r‘ ^ index /‘, views.Index.as\_view()),  
]

Views.py

from django.views import View  
  
  
class Index(View):  
 def get(self, req):  
 print(‘method is:‘ + req.method)  
 return render(req, ‘index.html‘)  
  
 def post(self, req):  
 print(‘method is:‘ + req.method)  
 return render(req, ‘index.html‘)

1. **说一说Django中间件的生命周期**

答案如上。

1. **在Django中如何写原生的sql**

在Django中使用原生Sql主要有以下几种方式:

extra:结果集修改器，一种提供额外查询参数的机制

raw:执行原始sql并返回模型实例

直接执行自定义Sql(这种方式完全不依赖model,前两种还是要依赖于model)

1. **如何使用Django ORM批量创建数据？**

数据模型定义：

from django.db import models  
# Create your models here.  
class Test(models.Model):  
 name = models.CharField(max\_length=32, null=True, default=None)  
 age = models.IntegerField(max\_length=32, null=True, default=None)

在urls.py中添加一个路径：

urlpatterns = [  
 path(**'admin/'**, admin.site.urls),  
 path(**"favicon.ico"**, RedirectView.as\_view(url=**'static/favicon.ico'**)),  
 re\_path(**'^index/'**,views.index),  
]

在views.py添加数据：

def index(request):  
 # ############### 添加数据 ###############  
 import random  
 product\_list\_to\_insert = list()  
 for x in range(100):  
 product\_list\_to\_insert.append(Test(name=**'apollo'**+str(x), age=random.randint(18,89)))  
 Test.objects.bulk\_create(product\_list\_to\_insert)  
return render(request, **'index.html'**)

批量更新数据  
批量更新数据时，先进行数据过滤，然后再调用update方法进行一次性地更新。  
下面的语句将生成类似update....frrom....的SQL语句。  
  
# ############### 更新数据 ###############  
Test.objects.filter(name\_\_contains=**'apollo1'**).update(name=**'Jack'**)  
 批量删除数据  
批量更新数据时，先是进行数据过滤，然后再调用delete方法进行一次性删除。  
下面的语句讲生成类似delete from ... where ... 的SQL语句。  
  
# ############### 删除数据 ###############  
Test.objects.filter(name\_\_contains=**'jack'**).delete()

1. **路由匹配原则是什么？**

一、URL的基本配置如下：

from django.conf.urls import url

from . import views

urlpatterns = [

url(r’^blog/[0-9]+/$’, views.page), # 捕捉到的参数不命名

url(r’^blog/page(?P[0-9]+)/$’, views.page), # 正则表达式匹配到的参数命名为num

]

views.py中的方法如下：

def page(request, num):

print “page\_num”, num

\*正则表达式最好以结尾，这样能根据URL的内容进行精准的匹配，如果不加结尾，这样能根据URL的内容进行精准的匹配，如果不加，会出现路由错误的情况：

例如：

第一种情况：

urlpatterns = [

url(r’^query/’, views.query),

url(r’^query/supply\_hotel/’, views.supply\_hotel),

]

访问query/supply\_hotel/会路由到query/，因为query/在前，此种情况，如果把query/supply\_hotel/’写在第一列就不会出现问题。

第二种情况：

urlpatterns = [

url(r’^query/′,views.query),url(r′query/supplyhotel/′,views.query),url(r′query/supplyhotel/’, views.supply\_hotel),

]

访问query/supply\_hotel/会正确路由

二、django处理URL参数命名冲突的方式

It’s possible to have a URL pattern which captures named keyword arguments, and also passes arguments with the same names in its dictionary of extra arguments. When this happens, the arguments in the dictionary will be used instead of the arguments captured in the URL.

从URL中通过正则获得的命名参数如果与额外传入的字典参数中有冲突，则使用字典中的参数

例如：

urlpatterns = [

url(r’^blog/page(?P[0-9]+)/$’, views.page，{“num”:12}),

]

则访问blog/page/16/时，传入的参数num会为12

三、是否捕捉URL参数

官方文档原文：

As a rule of thumb, only capture the values the view needs to work with and use non-capturing arguments when the regular expression needs an argument but the view ignores it.

理解：

当view中需要用到URL中的参数时才捕捉参数

当路由需要，但view不需要时，不需要捕捉URL中的参数

四、嵌套URL参数

from django.conf.urls import url

urlpatterns = [

url(r’blog/(page-(\d+)/)?’, blog\_articles),                  # bad

    url(r’comments/(?:page-(?P\d+)/)?’, blog\_articles),                  # bad      url(r’comments/(?:page-(?P\d+)/)?’, comments), # good

]

假设链接为blog/page-2/

第一条路由，将传递两个位置参数，分别为page-2和2

假设链接为comments/page-2/

第二条路由，传递的变量名称为page\_number，值为2，注意，不会捕捉(?:……)的内容

1. **解决跨域的常用方法都有哪些？**

看如下例子：

<!DOCTYPE HTML>  
<html>  
<head>  
 <meta charset=**"utf-8"**>  
 <meta name=**"viewport"** content=**"maximum-scale=1.0,minimum-scale=1.0,user-scalable=0,width=device-width,initial-scale=1.0"**/>  
 <title>test</title>  
</head>  
<body>  
<button onclick=**"showPersonInfo()"**>点我获取数据</button>  
</body>  
<script src=**"http://libs.baidu.com/jquery/1.9.0/jquery.js"**></script>  
 <script>  
 function showPersonInfo(){  
 $.getJSON(  
 **'http://xxx/account/getuserinfo/'**,  
 {username: **"abc"**},  
 function(json) {  
 var html=**'<br>'**+**'用户名：'**+json.username+**'<br>'**+**'姓：'**+json.first\_name+**'<br>'**+**'名:'**+json.last\_name+**'<br>'**+**'邮箱'**+json.email;   
 document.write(html);  
 }  
 )  
 }  
 </script>  
</html>

但是，Chrome浏览器提示错误：

No 'Access-Control-Allow-Origin' header is present on the requested resource.

经过一番Google发现这个问题是——CORS导致的。

什么是CORS？  
 CORS（跨域资源共享，Cross-Origin Resource Sharing）是一种跨域访问的机制，可以让Ajax实现跨域访问。  
 其实，在服务器的response header中，加入“Access-Control-Allow-Origin: \*”即可支持CORS，非常的简单，apache/nginx等怎么配置，见参考文档。  
举个例子：

API部署在DomainA上；

Ajax文件部署在DomainB上，Ajax文件会向API发送请求，返回数据；

用户通过DomainC访问DomainB的Ajax文件，请求数据

以上过程就发生了跨域访问。如果直接使用Ajax来请求就会失败，就像Chrome提示的：

No 'Access-Control-Allow-Origin' header is present on the requested resource.

如何解决Ajax跨域访问问题？  
 解决跨域问题，有两个方法：

1. 使用jsonp
2. 使CORS生效

使用jsonp方法，需要让服务器端放回jsonp格式的response，如Django可以加jsonp相关的decorator，如：

<https://coderwall.com/p/k8vb_a/returning-json-jsonp-from-a-django-view-with-a-little-decorator-help>

由于我不太喜欢这种方式，所以这里略过了，可看后面的参考资料。  
 使用CORS：这个用起来比较方便，现在大多数浏览器都支持了，且我web服务器完全开放给别人调用，所以比较推荐CORS。  
 1.使用JSONP  
 使用Ajax获取json数据时，存在跨域的限制。不过，在Web页面上调用js的script脚本文件时却不受跨域的影响，JSONP就是利用这个来实现跨域的传输。因此，我们需要将Ajax调用中的dataType从JSON改为JSONP（相应的API也需要支持JSONP）格式。JSONP只能用于GET请求。

2.直接修改Django中的views.py文件  
修改views.py中对应API的实现函数，允许其他域通过Ajax请求数据：

def myview(\_request):  
 response = HttpResponse(json.dumps({**"key"**: **"value"**, **"key2"**: **"value"**}))  
 response[**"Access-Control-Allow-Origin"**] = **"\*"** response[**"Access-Control-Allow-Methods"**] = **"POST, GET, OPTIONS"** response[**"Access-Control-Max-Age"**] = **"1000"** response[**"Access-Control-Allow-Headers"**] = **"\*"** return response

3.安装django-cors-headers  
这里还有一各发现！在Django中，有人开发了CORS-header的middleware，只在settings.py中做一些简单的配置即可，见：

<https://github.com/ottoyiu/django-cors-headers/>

现在用起来服务器端完全开放，开启CORS，没有跨域烦恼，真爽！~  
 安装django-cors-headers：

pip install django-cors-headers

在settings.py中增加：

INSTALLED\_APPS = (  
 ...  
 **'corsheaders'**,  
 ...  
)  
...  
  
MIDDLEWARE\_CLASSES = (  
 ...  
 **'corsheaders.middleware.CorsMiddleware'**,  
 **'django.middleware.common.CommonMiddleware'**,  
 ...  
)

可以配置允许跨域访问的白名单或者直接设置为允许所有的跨域访问，具体的配置可以看看他们的github页说明。

1. **Django的model继承有几种方式？分别是什么？**

Django 中的 model 继承和 Python 中的类继承非常相似，只不过你要选择具体的实现方式：让父 model 拥有独立的数据库；还是让父 model 只包含基本的公共信息，而这些信息只能由子 model 呈现。

Django中有三种继承关系：

1.通常，你只是想用父 model 来保存那些你不想在子 model 中重复录入的信息。父类是不使用的也就是不生成单独的数据表，这种情况下使用抽象基类继承 Abstract base classes。

2.如果你想从现有的Model继承并让每个Model都有自己的数据表，那么使用多重表继承Multi-table inheritance。

3.最后，如果你只想在 model 中修改 Python-level 级的行为，而不涉及字段改变。 代理 model (Proxy models) 适用于这种场合。

**Abstract base classes**

如果你想把某些公共信息添加到很多 model 中，抽象基类就显得非常有用。你编写完基类之后，在 Meta 内嵌类中设置 abstract=True ，该类就不能创建任何数据表。然而如果将它做为其他 model 的基类，那么该类的字段就会被添加到子类中。抽象基类和子类如果含有同名字段，就会导致错误(Django 将抛出异常)。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class CommonInfo(models.Model):      name = models.CharField(max\_length=100)      age = models.PositiveIntegerField()        class Meta:          abstract = True    class Student(CommonInfo):      home\_group = models.CharField(max\_length=5) |

sqlall结果:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | CREATE TABLE "myapp\_student" (      "id" integer NOT NULL PRIMARY KEY,      "name" varchar(100) NOT NULL,      "age" integer unsigned NOT NULL,      "home\_group" varchar(5) NOT NULL  ) |

只为Student model 生成了数据表，而CommonInfo不能做为普通的 Django model 使用，因为它是一个抽象基类。他即不生成数据表，也没有 manager ，更不能直接被实例化和保存。

对很多应用来说，这种继承方式正是你想要的。它提供一种在 Python 语言层级上提取公共信息的方式，但在数据库层级上，每个子类仍然只创建一个数据表，在JPA中称作TABLE\_PER\_CLASS。这种方式下，每张表都包含具体类和继承树上所有父类的字段。因为多个表中有重复字段，从整个继承树上来说，字段是冗余的。

**Meta继承**

创建抽象基类的时候，Django 会将你在基类中所声明的有效的 Meta 内嵌类做为一个属性。如果子类没有声明它自己的 Meta 内嵌类，它就会继承父类的 Meta 。子类的 Meta 也可以直接继承父类的 Meta 内嵌类，对其进行扩展。例如：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | class CommonInfo(models.Model):      name = models.CharField(max\_length=100)      age = models.PositiveIntegerField()      class Meta:          abstract = True          ordering = ['name']    class Student(CommonInfo):      home\_group = models.CharField(max\_length=5)      class Meta(CommonInfo.Meta):          db\_table = 'student\_info' |

sqlall结果：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | CREATE TABLE "student\_info" (      "id" integer NOT NULL PRIMARY KEY,      "name" varchar(100) NOT NULL,      "age" integer unsigned NOT NULL,      "home\_group" varchar(5) NOT NULL  ) |

按照我们指定的名称student\_info生成了table。

继承时，Django 会对基类的 Meta 内嵌类做一个调整：在安装 Meta 属性之前，Django 会设置 abstract=False。 这意味着抽象基类的子类不会自动变成抽象类。当然，你可以让一个抽象类继承另一个抽象基类，不过每次都要显式地设置 abstract=True 。

对于抽象基类而言，有些属性放在 Meta 内嵌类里面是没有意义的。例如，包含 db\_table 将意味着所有的子类(是指那些没有指定自己的 Meta 内嵌类的子类)都使用同一张数据表，一般来说，这并不是我们想要的。

小心使用 related\_name (Be careful with related\_name)

如果你在 ForeignKey 或 ManyToManyField 字段上使用 related\_name 属性，你必须总是为该字段指定一个唯一的反向名称。但在抽象基类上这样做就会引发一个很严重的问题。因为 Django 会将基类字段添加到每个子类当中，而每个子类的字段属性值都完全相同 (这里面就包括 related\_name)。注：这样使用 ForeignKey 或 ManyToManyField 反向指定时就无法确定是指向哪个子类了。

当你在(且仅在)抽象基类中使用 related\_name 时，如果想绕过这个问题，就要在属性值中包含  '%(app\_label)s' 和 '%(class)s'字符串。

1.'%(class)s'会被子类的名字取代。

2.'%(app\_label)s'会被子类所在的app的名字所取代。

举例，在app common中，common/models.py：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | class Base(models.Model):      m2m = models.ManyToManyField(OtherModel, related\_name="%(app\_label)s\_%(class)s\_related")        class Meta:          abstract = True    class ChildA(Base):      pass    class ChildB(Base):      pass |

在另外一个app中，rare/models.py：

|  |  |
| --- | --- |
| 1  2 | class ChildB(Base):      pass |

那么common.ChildA.m2m字段的反向名称为common\_childa\_related, common.ChildB.m2m字段的反向名称为common\_childb\_related, rare app中rare.ChildB.m2m字段的反向名称为rare\_childb\_related.

如果你没有在抽象基类中为某个关联字段定义 related\_name 属性，那么默认的反向名称就是子类名称加上 '\_set'，它能否正常工作取决于你是否在子类中定义了同名字段。例如，在上面的代码中，如果去掉 related\_name 属性，在 ChildA 中，m2m 字段的反向名称就是 childa\_set；而 ChildB 的 m2m 字段的反向名称就是 childb\_set 。

**多表继承(Multi-table inheritance)**

这是 Django 支持的第二种继承方式。使用这种继承方式时，同一层级下的每个子 model 都是一个真正意义上完整的 model 。每个子 model 都有专属的数据表，都可以查询和创建数据表。继承关系在子 model 和它的每个父类之间都添加一个链接 (通过一个自动创建的 OneToOneField 来实现)。 例如：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class Place(models.Model):      name = models.CharField(max\_length=50)      address = models.CharField(max\_length=80)    class Restaurant(Place):      serves\_hot\_dogs = models.BooleanField()      serves\_pizza = models.BooleanField() |

sqlall:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | BEGIN;  CREATE TABLE "myapp\_place" (      "id" integer NOT NULL PRIMARY KEY,      "name" varchar(50) NOT NULL,      "address" varchar(80) NOT NULL  )  ;  CREATE TABLE "myapp\_restaurant" (      "place\_ptr\_id" integer NOT NULL PRIMARY KEY REFERENCES "myapp\_place" ("id"),        "serves\_hot\_dogs" bool NOT NULL,      "serves\_pizza" bool NOT NULL  )  ;    COMMIT; |

父类和子类都生成了单独的数据表，Restaurant中存储了Place的id，也就是通过OneToOneField链接在一起。继承关系通过表的JOIN操作来表示。在JPA中称作JOINED。这种方式下，每个表只包含类中定义的字段，不存在字段冗余，但是要同时操作子类和所有父类所对应的表。

Place 里面的所有字段在 Restaurant 中也是有效的，只不过数据保存在另外一张数据表当中。所以下面两个语句都是可以运行的：

|  |  |
| --- | --- |
| 1  2 | >>> Place.objects.filter(name="Bob's Cafe")  >>> Restaurant.objects.filter(name="Bob's Cafe") |

如果你有一个 Place，那么它同时也是一个 Restaurant， 那么你可以使用子 model 的小写形式从 Place 对象中获得与其对应的 Restaurant 对象：

|  |  |
| --- | --- |
| 1  2  3  4 | >>> p = Place.objects.filter(name="Bob's Cafe")  # If Bob's Cafe is a Restaurant object, this will give the child class:  >>> p.restaurant  <Restaurant: ...> |

但是，如果上例中的 p 并不是 Restaurant (比如它仅仅只是 Place 对象，或者它是其他类的父类)，那么在引用 p.restaurant 就会抛开Restaurant.DoesNotExist 异常：

|  |  |
| --- | --- |
| 1  2  3  4 | >>> from myapp.models import Place,Restaurant  >>> p=Place.objects.create(name='Place',address='Place')  >>> p.restaurant  DoesNotExist: Place has no restaurant. |

也就是说，创建Place实例的同时不会创建Restaurant，但是创建Restaurant实例的同时会创建Place实例：

|  |  |
| --- | --- |
| 1  2  3  4 | >>>Restaurant.objects.create(name='M',address='M',serves\_hot\_dogs=True,serves\_pizza=True)  <Restaurant: Restaurant object>  >>> Place.objects.get(name='M')  <Place: Place object> |

多表继承中的Meta (Meta and multi-table inheritance)

在多表继承中，子类继承父类的 Meta 内嵌类是没什么意见的。所有的 Meta 选项已经对父类起了作用，再次使用只会起反作用。(这与使用抽象基类的情况正好相反，因为抽象基类并没有属于它自己的内容)

所以子 model 并不能访问它父类的 Meta 内嵌类。但是在某些受限的情况下，子类可以从父类继承某些 Meta ：如果子类没有指定 django.db.models.Options.ordering 属性或 django.db.models.Options.get\_latest\_by 属性，它就会从父类中继承这些属性。

如果父类有了排序设置，而你并不想让子类有任何排序设置，你就可以显式地禁用排序：

|  |  |
| --- | --- |
| 1  2  3  4  5 | class ChildModel(ParentModel):      # ...      class Meta:          # Remove parent's ordering effect          ordering = [] |

**继承与反向关联(Inheritance and reverse relations)**

因为多表继承使用了一个隐含的 OneToOneField 来链接子类与父类，所以象上例那样，你可以用父类来指代子类。但是这个 OnetoOneField 字段默认的 related\_name 值与 django.db.models.fields.ForeignKey 和 django.db.models.fields.ManyToManyField 默认的反向名称相同。如果你与其他 model 的子类做多对一或是多对多关系，你就必须在每个多对一和多对多字段上强制指定 related\_name 。如果你没这么做，Django 就会在你运行 验证(validate) 或 同步数据库(syncdb) 时抛出异常。

例如，仍以上面 Place 类为例，我们创建一个带有 ManyToManyField 字段的子类：

|  |  |
| --- | --- |
| 1  2  3 | class Supplier(Place):      # Must specify related\_name on all relations.      customers = models.ManyToManyField(Restaurant, related\_name='provider') |

指定链接父类的字段(Specifying the parent link field)

之前我们提到，Django 会自动创建一个 OneToOneField 字段将子类链接至非抽象的父 model 。如果你想指定链接父类的属性名称，你可以创建你自己的 OneToOneField 字段并设置 parent\_link=True ，从而使用该字段链接父类。

**代理model (Proxy models)**

使用 多表继承(multi-table inheritance) 时，model 的每个子类都会创建一张新数据表，通常情况下，这正是我们想要的操作。这是因为子类需要一个空间来存储不包含在基类中的字段数据。但有时，你可能只想更改 model 在 Python 层的行为实现。比如：更改默认的 manager ，或是添加一个新方法。

而这，正是代理 model 继承方式要做的：为原始 model 创建一个代理(proxy)。你可以创建，删除，更新代理 model 的实例，而且所有的数据都可以象使用原始 model 一样被保存。不同之处在于：你可以在代理 model 中改变默认的排序设置和默认的 manager ，更不会对原始 model 产生影响。

声明代理 model 和声明普通 model 没有什么不同。设置Meta 内置类中 proxy 的值为 True，就完成了对代理 model 的声明。

举个例子，假设你想给 Django 自带的标准 User model (它被用在你的模板中)添加一个方法：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | class Person(models.Model):      first\_name = models.CharField(max\_length=30)      last\_name = models.CharField(max\_length=30)    class MyPerson(Person):      class Meta:          proxy = True        def do\_something(self):          # ...          pass |

sqlall：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | CREATE TABLE "myapp\_person" (      "id" integer NOT NULL PRIMARY KEY,      "first\_name" varchar(30) NOT NULL,      "last\_name" varchar(30) NOT NULL  )  ; |

MyPerson 类和它的父类 Person操作同一个数据表。特别的是，Person 的任何实例也可以通过 MyPerson 访问，反之亦然：

|  |  |
| --- | --- |
| 1  2  3 | >>> p = Person.objects.create(first\_name="foobar")  >>> MyPerson.objects.get(first\_name="foobar")  <MyPerson: foobar> |

你也可以使用代理 model 给 model 定义不同的默认排序设置。Django 自带的 User model 没有定义排序设置(这是故意为之，是因为排序开销极大，我们不想在获取用户时浪费额外资源)。你可以利用代理对 username 属性进行排序，这很简单：

|  |  |
| --- | --- |
| 1  2  3  4 | class OrderedPerson(Person):      class Meta:          ordering = ["last\_name"]          proxy = True |

普通的 User 查询，其结果是无序的；而 OrderedUser 查询的结果是按 username 排序。

查询集只返回请求时所使用的 model (Querysets still return the model that was requested)

无论你何时查询Person 对象，Django 都不会返回 MyPerson 对象。针对 Person 对象的查询集只返回 Person 对象。代理对象的精要就在于依赖原始 User 的代码仅对它自己有效，而你自己的代码就使用你扩展的内容。不管你怎么改动，都不会在查询 Person 时得到 MyPerson。

**基类的限制(Base class restrictions)**

代理 model 必须继承自一个非抽象基类。你不能继承自多个非抽象基类，这是因为一个代理 model 不能连接不同的数据表。代理 model 也可以继承任意多个抽象基类，但前提是它们没有定义任何 model 字段。

代理 model 从非抽象基类中继承那些未在代理 model 定义的 Meta 选项。

代理 model 的 manager (Proxy model managers)

如果你没有在代理 model 中定义任何 manager ，代理 model 就会从父类中继承 manager 。如果你在代理 model 中定义了一个 manager ，它就会变成默认的 manager ，不过定义在父类中的 manager 仍是有效的。

继续上面的例子，你可以改变默认 manager，例如：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class NewManager(models.Manager):      # ...      pass    class MyPerson(Person):      objects = NewManager()        class Meta:          proxy = True |

如果你想给代理添加一个新的 manager ，却不想替换已有的默认 manager ，那么你可以参考 自定义 manager (custom manager) 中提到的方法：创建一个包含新 manager 的基类，然后放在主基类后面继承：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class ExtraManagers(models.Model):      secondary = NewManager()        class Meta:          abstract = True    class MyPerson(Person, ExtraManagers):      class Meta:          proxy = True |

你可能不需要经常这样做，但这样做是可行的。

代理 model 与非托管 model 之间的差异(Differences between proxy inheritance and unmanaged models)

代理 model 继承看上去和使用 Meta 内嵌类中的 managed 属性的非托管 model 非常相似。但两者并不相同，你应当考虑选用哪种方案。

一个不同之处是你可以在 Meta.managed=False 的 model 中定义字段(事实上，是必须指定，除非你真的想得到一个空 model )。在创建非托管 model 时要谨慎设置 Meta.db\_table ，这是因为创建的非托管 model 映射某个已存在的 model ，并且有自己的方法。因此，如果你要保证这两个 model 同步并对程序进行改动，那么就会变得繁冗而脆弱。

另一个不同之处是两者对 manager 的处理方式不同。这对于代理 model 非常重要。代理 model 要与它所代理的 model 行为相似，所以代理 model 要继承父 model 的 managers ，包括它的默认 manager 。但在普通的多表继承中，子类不能继承父类的 manager ，这是因为在处理非基类字段时，父类的 manager 未必适用。

我们实现了这两种特性(Meta.proxy和Meta.unmanaged)之后，曾尝试把两者结合到一起。结果证明，宏观的继承关系和微观的 manager 揉在一起，不仅导致 API 复杂难用，而且还难以理解。由于任何场合下都可能需要这两个选项，所以目前二者仍是各自独立使用的。

所以，一般规则是：

1.如果你要镜像一个已有的 model 或数据表，且不想涉及所有的原始数据表的列，那就令 Meta.managed=False。通常情况下，对数据库视图创建 model 或是数据表不需要由 Django 控制时，就使用这个选项。  
 2.如果你想对 model 做 Python 层级的改动，又想保留字段不变，那就令 Meta.proxy=True。因此在数据保存时，代理 model 相当于完全复制了原始 model 的存储结构。

**多重继承(Multiple inheritance)**

和 Python 一样，Django 的 model 也可以做多重继承。这里要记住 Python 的名称解析规则。如果某个特定名称 (例如，Meta) 出现在第一个基类当中，那么子类就会使用第一个基类的该特定名称。例如，如果多重父类都包含 Meta 内嵌类，只有第一个基类的 Meta 才会被使用，其他的都被会忽略。

一般来说，你没必要使用多重继承。

不允许"隐藏"字段(Field name "hiding" is not permitted)

普通的 Python 类继承允许子类覆盖父类的任何属性。但在 Django 中，重写 Field 实例是不允许的(至少现在还不行)。如果基类中有一个 author 字段，你就不能在子类中创建任何名为 author 的字段。

重写父类的字段会导致很多麻烦，比如：初始化实例(指定在 Model.\_\_init\_\_ 中被实例化的字段) 和序列化。而普通的 Python 类继承机制并不能处理好这些特性。所以 Django 的继承机制被设计成与 Python 有所不同，这样做并不是随意而为的。

这些限制仅仅针对做为属性使用的 Field 实例，并不是针对 Python 属性，Python 属性仍是可以被重写的。在 Python 看来，上面的限制仅仅针对字段实例的名称：如果你手动指定了数据库的列名称，那么在多重继承中，你就可以在子类和某个父类当中使用同一个列名称。(因为它们使用的是两个不同数据表的字段)。

如果你在任何一个父类中重写了某个 model 字段，Django 都会抛出 FieldError 异常。

1. **Django中想验证表单提交是否格式正确需要用到form中的哪个函数？**

form.is\_valid() :返回布尔值

1. **ORM如何取消级联？**

User = models.ForeignKey(user,blank = True,null = True,on\_delete = models.SET\_NULL)

并且SET\_NULL只有在null为True的时候，才可以使用。

1. **Django中如何读取和保存session？整个session的运行机制是什么？**

1、存放数据到session

request.session['some\_id'] = some\_id

2、从session中读取存放的数据

session.get('some\_id', False)

3、从session中将数据删除

del request.session['some\_id']

4、让session过期

request.session.set\_expiry()

5、在命令行访问sessin数据

# 生成并保存session（利用SessionStore）

from django.contrib.sessions.backends.db import SessionStore

sessionStore = SessionStore()

sessionStore["str"] = "hello" # 字串映射

sessionStore["dict"] = {}; # 可以定义多级的字典结构

sessionStore["dict"]["key1"]="value1"

sessionStore["dict"]["key2"]="value2"

sessionStore.save();

print(sessionStore.session\_key);

print(sessionStore.keys());

session\_key = sessionStore.session\_key;

# 读取保存的session

from django.contrib.sessions.models import Session

session = Session.objects.get(pk=session\_key)

print(session.session\_data); # 返回session的存储（加密过）

print(session.get\_decoded()); # 返回session的数据结构（加过解码）

print(session.expire\_date);

注意：

（1）保存数据的使用使用的是SessionStore读取数据使用使用的是Session。

（2）在使用多级字典时session["dict"]["key1"] = "something" django默认不会对多级对象进行保存，需要显示的使用代码

1. **简述Django对http请求的执行流程**

Django 和其他 Web 框架的 HTTP 处理的流程大致相同，Django 处理一个 Request 的过程是首先通过中间件，然后再通过默认的 URL 方式进行的。我们可以在 Middleware 这个地方把所有 Request 拦截住，用我们自己的方式完成处理以后直接返回 Response。  
        1. 加载配置

Django 的配置都在 “Project/settings.py” 中定义，可以是 Django 的配置，也可以是自定义的配置，并且都通过 django.conf.settings 访问，非常方便。

 2. 启动

最核心动作的是通过 django.core.management.commands.runfcgi 的 Command 来启动，它运行 django.core.servers.fastcgi 中的 runfastcgi，runfastcgi 使用了 flup 的 WSGIServer 来启动 fastcgi 。而 WSGIServer 中携带了 django.core.handlers.wsgi 的 WSGIHandler 类的一个实例，通过 WSGIHandler 来处理由 Web 服务器（比如 Apache，[Lighttpd](https://www.baidu.com/s?wd=Lighttpd&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd" \t "https://blog.csdn.net/dapeng0802/article/details/_blank) 等）传过来的请求，此时才是真正进入 Django 的世界。

3. 处理 Request

当有 HTTP 请求来时，WSGIHandler 就开始工作了，它从 BaseHandler 继承而来。WSGIHandler 为每个请求创建一个 WSGIRequest 实例，而 WSGIRequest 是从 http.HttpRequest 继承而来。接下来就开始创建 Response 了。

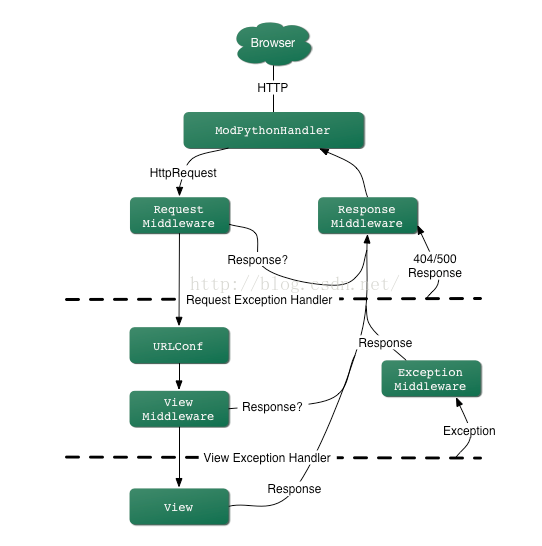
4. 创建 Response

BaseHandler 的 get\_response 方法就是根据 request 创建 response，而具体生成 response 的动作就是执行 urls.py 中对应的view函数了，这也是 Django 可以处理“友好 URL ”的关键步骤，每个这样的函数都要返回一个 Response 实例。此时一般的做法是通过 loader 加载 template 并生成页面内容，其中重要的就是通过 ORM 技术从数据库中取出数据，并渲染到 Template 中，从而生成具体的页面了。

5. 处理 Response

Django 返回 Response 给 flup，flup 就取出 Response 的内容返回给 Web 服务器，由后者返回给浏览器。

总之，Django 在 fastcgi 中主要做了两件事：处理 Request 和创建 Response，而它们对应的核心就是“ urls 分析”、“模板技术”和“ ORM 技术”。

****

如图所示，一个 HTTP 请求，首先被转化成一个 HttpRequest 对象，然后该对象被传递给 Request 中间件处理，如果该中间件返回了Response，则直接传递给 Response 中间件做收尾处理。否则的话 Request 中间件将访问 URL 配置，确定哪个 view 来处理，在确定了哪个 view 要执行，但是还没有执行该 view 的时候，系统会把 request 传递给 view 中间件处理器进行处理，如果该中间件返回了 Response，那么该 Response 直接被传递给 Response 中间件进行后续处理，否则将执行确定的 view 函数处理并返回 Response，在这个过程中如果引发了异常并抛出，会被 Exception 中间件处理器进行处理。

1. **Django中当用户登录到A服务器进入登陆状态，下次被nginx代理到B服务器会出现什么影响**

如果用户在A应用服务器登陆的session数据没有共享到B应用服务器，那么之前的登录状态就没有了。

1. **如何获取django urlpatterns里面注册的所有url?**

from django.conf.urls import url, include  
from arya.service.sites import site  
from django.urls.resolvers import RegexURLPattern  
from django.urls.resolvers import RegexURLResolver  
from django.shortcuts import HttpResponse  
  
  
def index(request):  
 print(get\_all\_url(urlpatterns, prev=**'/'**))  
 return HttpResponse(**'...'**)  
  
  
def get\_all\_url(urlparrentens, prev, is\_first=False, result=[]):  
 if is\_first:  
 result.clear()  
 for item in urlparrentens:  
 v = item.\_regex.strip(**'^'**) # 去掉url中的^和  
 if isinstance(item, RegexURLPattern):  
 result.append(prev + v)  
 else:  
 get\_all\_url(item.urlconf\_name, prev + v)  
 return result  
  
  
urlpatterns = [  
 url(**r'^arya/'**, site.urls),  
 url(**r'^index/'**, index),  
]

1. **django路由系统中include是干嘛用的？**

include路由转发

通常，我们会在每个app里，各自创建一个urls.py路由模块，然后从根路由出发，将app所属的url请求，全部转发到相应的urls.py模块中。

1. **django2.0中的path与django1.xx里面的url有什么区别？**

2.0内的path匹配正则时候无效，导入re\_path即可匹配正则

1. **urlpatterns中的name与namespace有什么作用？你是如何使用的？**

别名，给路由起一个别名

名称空间，防止多个应用之间的路由重复

1. **Django 获取用户前端请求数据的几种方式**

def get(self, key, default=None):  
 """  
 Returns the last data value for the passed key. If key doesn't exist  
 or value is an empty list, then default is returned.  
 """  
 try:  
 val = self[key]  
 except KeyError:  
 return default  
 if val == []:  
 return default  
 return val  
  
  
def getlist(self, key, default=None):  
 """  
 Returns the list of values for the passed key. If key doesn't exist,  
 then a default value is returned.  
 """  
 try:  
 return super(MultiValueDict, self).\_\_getitem\_\_(key)  
 except KeyError:  
 if default is None:  
 return []  
 return default  
  
  
def \_\_getitem\_\_(self, key):  
 """  
 Returns the last data value for this key, or [] if it's an empty list;  
 raises KeyError if not found.  
 """  
 try:  
 list\_ = super(MultiValueDict, self).\_\_getitem\_\_(key)  
 except KeyError:  
 raise MultiValueDictKeyError(repr(key))  
 try:  
 return list\_[-1]  
 except IndexError:  
 return []

1. **什么是Cookie、如何获取、设置Cookie**

Cookie 是在HTTP协议下，服务器或脚本可以维护客户工作站上信息的一种方式。Cookie 是由 Web

服务器保存在用户浏览器（客户端）上的小文本文件(内容通常经过加密)，它可以包含有关用户的信息。无论何时用户链接到服务器，Web站点都可以访问

Cookie 信息,可以看作是浏览器缓存.

**什么是Session**

Session的定义很抽象,在不同的场合中session一词的含义也很不相同.它可以代表服务器与浏览器的一次会话过程,指从一个浏览器窗口打开到关闭的这个期间.也可以用于指一类用来在客户端与服务器之间保持状态的解决方案.

Cookie需要解决三个问题：分发、内容和使用。

cookie的分发是通过扩展HTTP协议来实现的，服务器端通过在HTTP的响应由中加上一行特殊的指示以提示浏览器按照指示生成相应的cookie。

cookie的内容主要包括name(名字)、value(值)、maxAge(失效时间)、path(路径),domain(域)和secure

name：cookie的名字，一旦创建，名称不可更改。

value：cookie的值，如果值为Unicode字符，需要为字符编码。如果为二进制数据，则需要使用BASE64编码.

maxAge：cookie失效时间，单位秒。如果为正数，则该cookie在maxAge后失效。如果为负数，该cookie为临时cookie，关闭浏览器即失效，浏览器也不会以任何形式保存该cookie。如果为0，表示删除该cookie。默认为-1

path：该cookie的使用路径。如果设置为"/sessionWeb/"，则只有ContextPath为“/sessionWeb/”的程序可以访问该cookie。如果设置为“/”，则本域名下ContextPath都可以访问该cookie。

domain:域.可以访问该Cookie的域名。第一个字符必须为".",如果设置为".google.com",则所有以"google.com结尾的域名都可以访问该cookie",如果不设置,则为所有域名

secure：该cookie是否仅被使用安全协议传输。

cookie的使用是由浏览器按照一定的原则在后台自动发送给服务器。浏览器检查所有存储的cookie，如果某个cookie所声明的作用范围大于等于将要请求的资源所在的位置，则把该cookie附在请求资源的HTTP请求头上发送给服务器.

**Session机制**

Session机制是一种服务端的机制，服务器使用一种类似散列表的结构来保存信息。当程序需要为某个客户端的请求创建一个session的时候，服务器首先检查这个客户端里的请求里是否已包含了一个session标识--sessionID，如果已经包含一个sessionID，则说明以前已经为此客户端创建过session，服务器就按照sessionID把这个session检索出来使用（检索不到，可能会新建一个），如果客户端请求不包含sessionID，则为此客户端创建一个session并且声称一个与此session相关联的sessionID，sessionID的值应该是一个既不会重复，又不容易被找到规律以仿造的字符串(服务器会自动创建),这个sessionID将被在本次响应中返回给客户端保存。

保存sessionID的方式

1，使用Cookie.此时Cookie不再用作认证,只是作为载体,存储sessionID

2，URL重写：因为cookie可以被人为禁止，所以为了保证sessionID能够被传递给服务器，使用URL重写也是一种解决方法。如,

href="<%=response.encodeURL("index.jsp") %>"

写好后URL是这样的

href="index.jsp;jsessionid=0CCD096E7F8D97B0BE608AFDC3E1931E"

3, 隐藏表单提交.将sessionId放在隐藏表单中

实际上这几种方式最方便的还是cookie,其他两种方式都各有弊端,URL重写的弊端是要对所有的URL都重写,非常繁琐.表单隐藏字段的弊端是只能局限于表单提交,如果是单纯的文本链接则不能提供会话跟踪

**Cookie和Session之间的区别**

1.Cookie数据存放在客户的浏览器(本地),session数据放在服务器上

2.Cookie不如session安全，别人可以分析存放在本地的Cookie并进行Cookie欺骗,所以出于安全性的考虑应当使用Session

3.Session会在一定时间内保存在服务器上。当访问增多,会占用较多的服务器资源,所以出于性能考虑则应当使用cookie

单个cookie保存的数据不能超过4k,很多浏览器都限制一个站点最多保存20个cookie.实际上为了性能考虑,不论是cookie还是session,其中的信息都应当短小精悍

session因为是保存在服务器上,所以不支持跨域的访问

**扩展思考**

使用cookie和session的具体场景

一般来说,登陆验证信息,客户的私人信息,如姓名,电话等,应该放在Session中.Cookie则用于用户登陆网站时的自动登陆以及类似"购物车"的处理.使用Cookie保存信息时最好通过加密形式来保存数据,同时是否保存登陆信息,需要由用户自行选择

1. **什么是session，与cookie的对比、设置、获取、清空session**
2. **WSGI / uwsgi/ uWSGI区分**

uWSGI是一个Web服务器，它实现了WSGI协议、uwsgi、http等协议。Nginx中HttpUwsgiModule的作用是与uWSGI服务器进行交换。

要注意 WSGI / uwsgi / uWSGI 这三个概念的区分。

WSGI是一种Web服务器网关接口。它是一个Web服务器（如nginx，uWSGI等服务器）与web应用（如用Flask框架写的程序）通信的一种规范。

uwsgi是一种线路协议而不是通信协议，在此常用于在uWSGI服务器与其他网络服务器的数据通信。

而uWSGI是实现了uwsgi和WSGI两种协议的Web服务器。

uwsgi协议是一个uWSGI服务器自有的协议，它用于定义传输信息的类型（type of information），每一个uwsgi packet前4byte为传输信息类型描述，它与WSGI相比是两样东西。

1. **QueryDict和dict区别**

dict是字典对象，没有urlencode()方法；

QueryDict对象，有urlencode()方法，作用是将QueryDict对象转换为url字符串；

一般QueryDict通过params = copy.deepcopy(request.GET)得到的，这时params是QueryDict对象；

也能params[key]=value进行添加数据！！！

1. **Django重定向你是如何实现的？用的什么状态码？**

1.使用HttpResponseRedirect #from django.http import HttpResponseRedirect

2.使用redirect和reverse #状态码：301和302 #301和302的区别：

相同点：都表示重定向，浏览器在拿到服务器返回的这个状态码后会自动跳转到一个新的URL地址

不同点：

   301比较常用的场景是使用域名跳转。比如，我们访问 http://www.baidu.com 会跳转到 https://www.baidu.com 表示旧地址A的资源已经被永久地移除了

   302用来做临时跳转，比如未登陆的用户访问用户中心重定向到登录页面。表示旧地址A的资源还在（仍然可以访问），这个重定向只是临时地从旧地址A跳转到地址B

1. **简述Django下的（内建）缓存机制**

Django中提供了6种缓存方式：

开发调试

内存

文件

数据库

Memcache缓存（python-memcached模块）

Memcache缓存（pylibmc模块）