# django

### django简单布局

pip3 install django

django是MTV模型：modal数据库，templates存放模板网页，views接收用户请求

#### 新建django项目

django-admin startproject mysite

在pycharm中新建项目

新建完有这么几个文件：setting.py,urls.py,wsgi.py

Setting.py：存放全局变量：

BASE\_DIR = os.path.dirname(os.path.dirname(os.path.abspath(\_\_file\_\_)))

INSTALLED\_APPS = [  
 'django.contrib.admin',  
 'django.contrib.auth',  
 'django.contrib.contenttypes',  
 'django.contrib.sessions',  
 'django.contrib.messages',  
 'django.contrib.staticfiles',  
]

Templates目录的信息

TEMPLATES = [  
 {  
 'BACKEND': 'django.template.backends.django.DjangoTemplates',  
 'DIRS': [os.path.join(BASE\_DIR, 'templates')]  
 ,  
 'APP\_DIRS': True,  
 'OPTIONS': {  
 'context\_processors': [  
 'django.template.context\_processors.debug',  
 'django.template.context\_processors.request',  
 'django.contrib.auth.context\_processors.auth',  
 'django.contrib.messages.context\_processors.messages',  
 ],  
 },  
 },  
]

数据库目录的信息,表示存放在db.sqlite3中

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

静态文件的url

STATIC\_URL = '/static/'

url.py：设定url对应的函数

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

#### 新建django app

一个project可以创建多个app

命令如下

python manage.py startapp cmdb

创建完的app又有许多文件：

Admins.py---后台管理系统

Models.py---数据库

Views.py---处理用户请求

Apps.py

Tests.py ----测试用

设置用户请求

当用户输入<http://127.0.0.1/index/>会有对应的函数来执行，不同的url可对应不同的函数，这些函数存放在views里

from django.http import HttpResponse

def index(request):  
 return HttpResponse("123")

这个函数应该要对应某个url，url的设定在urls.py里

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

必须加逗号

然后用户输入<http://127.0.0.1/index/>，就会显示123

#### 创建模板网页

在templates下创建一个index.html

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <title>Title</title>  
</head>  
<body>  
 <h2>注册界面</h2>  
</body>  
</html>

在index函数内修改：

def index(request):  
 return render(request,'index.html')

自动会去templates下找index.html

当用户输入上面url时显示网页<h2>注册界面</h2>

#### 导入静态文件

Statics里存放静态文件，js，css，图片都是静态文件

在statics中写一个inp.js

function f1() {  
 document.write("<h1>888</h1>");  
}

在html里导入：

<script src="/static/inp.js"></script>

static/指向哪里：setting中指定静态文件在根目录下的statics里

STATICFILES\_DIRS=(  
 os.path.join(BASE\_DIR,"statics"),  
)

这样就成功导入了

#### 用户提交数据：

Forbidden(403)

CSRF verification failed. Request aborted.

如果出现这个错误：

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',  
]

在此处把红色的注释掉

<form action="/index/" method="post">  
 用户名：<input type="text" name="user" />  
 邮箱：<input type="email" name="email" />  
 <input type="submit" value="提交" />  
</form>

用户发送用户名和邮箱的表单过来---服务器接收到post请求，判断---接收----执行

def index(request):  
 #print(request.method)  
 if(request.method=="POST"):  
 user=request.POST.get("user",None)  
 email=request.POST.get("email",None)  
 print(user,email)  
 return render(request,'index.html')

#### 服务器处理完，返回给用户

1. 数据打包
2. 发送给客户端
3. 在客户端展示出来

在render中传入给context就能把修改请求发送给客户端，context接收的参数是字典

def index(request):  
 #print(request.method)  
 if(request.method=="POST"):  
 user=request.POST.get("user",None)  
 email=request.POST.get("email",None)  
 temp={"user":user,"email":email}  
 USER\_INPUT.append(temp)  
 return render(request,'index.html',{'data':USER\_INPUT})

html中页面展示

<f2>数据展示</f2>  
<table>  
 {% for item in data %}  
 <tr>  
 <td>{{ item.user }}</td>  
 <td>{{ item.email }}</td>  
 </tr>  
 {% endfor %}  
</table>

以上操作都是在内存中完成的，如果服务器意外重启，数据全都丢失。所以要建一个数据库

#### 建数据库

Django默认连接sqlite

创建表

class UserInfo(models.Model):  
 user=models.CharField(max\_length=32)  
 email=models.CharField(max\_length=32)

注册app

python manage.py makemigrations

python manage.py migrate

插入数据

def index(request):  
 #print(request.method)  
 if(request.method=="POST"):  
 u=request.POST.get("user",None)  
 e=request.POST.get("email",None)  
 models.UserInfo.objects.create(user=u,email=e)  
 data\_list=models.UserInfo.objects.all()  
 return render(request,'index.html',{'data':data\_list})

不管什么时候，如果数据库出现问题，无法治疗的，删除app下的migrations

#### 启动服务：

python manage.py runserver 127.0.0.1:8000

#### redirect跳转

redirect("http://www.baidu.com")

### 路由系统

#### 处理url

从上到下匹配

当用户输入<http://127.0.0.1:8000/index>

按回车后，服务器接到这个请求，找urls中index/匹配的函数：

url(r'^index/', views.f1),

所以请求的是f1函数，然后把用户请求的信息放到F1的第一个参数request

Request就是一系列的请求报头的对象

Request.method是请求方式：get/post/head等等

Request.path为请求的路径/index/

resolve

可以采用正则表达式匹配：

<http://127.0.0.1:8000/detail/123/456>

它的映射关系：

url(r'^detail/(\d+)/(\d+)',views.f2),

然后在f2中需要接受三个参数，如果参数不满足：

f2() takes 2 positional arguments but 3 were given

指定参数

url(r'^login/(?P<p1>\d+)/(?P<p2>\d+)',views.f2),

在函数中就必须定义固定形参：p1,p2

如果一个url一开始有/表示从根开始：/index/表示127.0.0.1:8000/index/

如果url没有从/开始表示从当前url开始

name别名

别名的作用是在模板中自动生成url

如：

urlpatterns=[

url(r'^index/',views.index,name="index"),

]

#template

{% url "index" %}

#### url分组

一个项目可以有多个app，每个app对应的url可隔离

一个urls可包含另一个urls模块，使用include

from django.conf.urls import url,include

把项目的urls中的url映射到app的urls，再由app的urls映射到每一个函数

#product url.py

urlpatterns = [  
 #url(r'^admin/', admin.site.urls),  
 url(r'^web/', include('app01.urls')),  
]

http://127.0.0.1:8000/add\_user\_type/?q=555

这样一个url表示

向add\_user\_type传一个GET数据q

http://127.0.0.1:8000/add\_user\_type/?q=555&t=22

表示传一个GET数据q和t

在view内取：

request.GET.get("q",None)

通过reverse解析别名

from django.shortcuts import reverse

reverse(view\_name,args)

#### 视图函数：

试图函数的模式有CBV,FBV。

CBV让url映射到一个类

from django.views import View

class Base(View):  
 def dispatch(self, request, \*args, \*\*kwargs):  
 pass  
  
 def get(self):  
 return HttpResponse("hello")  
   
 def post(self):  
 pass

url调用：

url(r'^home/$', views.Base.as\_view(), name='home'),

FBV让url映射到一个函数

#### 静态文件

STATIC\_URL = '/static/' #标识url  
STATIC\_ROOT=os.path.join(BASE\_DIR,'static/') # 静态文件根目录  
STATICFILES\_DIRS=(  
 os.path.join(BASE\_DIR,'static/'), # 静态文件目录列表 /static/ == static/

("downloads", "/opt/webfiles/stats"), # /static/downloads/...

)

在url中指定静态文件

from django.conf import settings

from django.conf.urls.static import static

urlpatterns = [

# ... the rest of your URLconf goes here ...

] + static(settings.STATIC\_URL, document\_root=settings.STATIC\_ROOT)

#### request包含的内容：

request.method判断用户提交的方法GET/POST

request.POST/request.GET/request.FILE

request.COOKIE/request.session

request.META包含用户端的环境，头部信息，服务端的环境等

request.body 请求主体

request.get\_raw\_uri() 完整的uri地址

request.path/request.path\_info当前url路径

request.user 系统admin的当前用户

request的GET默认每一项是一个列表，如

<QueryDict: {'source': [''], 'meaning': ['1'], 'status': [''], 'name': ['']}>

如果直接dict转换成字典，每个value都是列表，但是get取出来的是具体的值，所以只能遍历。

#### 头部设置

def index(request):  
 response=render(request,'index.html')  
 response.setdefault("Cache-Control","public")

response.setdefault(**"a"**,**"b"**)

return response

#### MTV和MVC

MVC有模型，展示给用户的视图，控制器接收用户和发送给用户

MTV是模型，模板，视图

django就是MTV结构，还有一个url路由分发

### CBV

以类表示一个视图

view.py

class IndexView(TemplateView):  
 template\_name = 'index.html'  
 #get请求处理  
 def get(self, request, \*args, \*\*kwargs):  
 return super(IndexView, self).get(request, \*args, \*\*kwargs)  
 #返回模板  
 def get\_context\_data(self, \*\*kwargs):  
 kwargs.update({"aa":"10","bb":"20"})  
  
 return super(IndexView,self).get\_context\_data(\*\*kwargs)

urls.py

from app01.views import IndexView  
  
urlpatterns = [  
 。。。  
 url(r'^$', IndexView.as\_view(),name="index"),  
]

请求方法

http\_method\_names = ['get', 'post', 'put', 'patch', 'delete', 'head', 'options', 'trace']

### response设置

response发送的形式有HTTPResponse/render/redirect

发送的内容有响应头部和响应主体

响应头部设置：

Content-Type

Set-Cookie

resp01 = render(request, 'ajax\_test.html')  
resp01["aa"] = "bb"

resp01["Set-Cookie"] = "token=fdsfdsf;expire=xxx"

resp01["Content-Type"]="application/json;charset=UTF-8"

### 常见模块位置：

| Q | django.db.models |
| --- | --- |
| F | django.db.models |
| atomic | django.db.transaction |
| View | django.views |
| csrf\_exempt | django.views.decorators.csrf |
|  |  |
|  |  |
|  |  |

### 模板

模板是在第一次拿请求，取网页时解析

在render里面，第三个参数传给模板字典，模板调用字典的key来输出value

def template(request):  
 return render(request,'template.html',{"k1":"v1","k2":[11,22,33]})

{{输出语句}}

<h2>{{ k1 }}</h2>

输出v1

python中使用模板

from django import template

t=template.Template("My name is {{ aa }}")  
c=template.Context({"aa":"bb"})  
print(t.render(c))

user\_list=["alex","jack"]

#从列表中取值{{ user\_list.0 }}

user\_dict={"username":"alex","password":"123456"}

#从字典中取值{{ user\_ dict.username }}

class User:

username="alex"

password="123"

u=User()

#从对象中取值{{ u.username }}

在模板中都是用.来取元素

{%控制语句%}

For循环：

{% for item in k2 %}  
<li>{{ item }}</li>  
{% endfor %}

カウンター、ループ回数計算

{{ forloop.counter }}

If else

{% if 1 == 2 %}  
 <p>132</p>

{% elif 1 == 2 %}

...  
{% else %}  
 <p>456</p>  
{% endif %}

#### 内置方法

date

格式：Y-m-d H:i:s

#### 自定义函数：

Filter

Simple\_tag

1. 在app中创建templatetags
2. 在templatetags下创建任意xx.py

from django import template

#代替safe  
from django.utils.safestring import mark\_safe  
#from django.template.base import resolve\_variable, Node, TemplateSyntaxError  
  
register = template.Library()  
  
@register.filter  
def f1(item,tag):  
 return item + "666" + tag  
  
@register.simple\_tag  
def f2(s1,s2,s3,s4):  
 return s1+s2+s3+s4

1. 导入模块到html

{% load xx %}

{% load <模块名> %}

1. 使用函数

{{ 'dsddf'|f1:"ddd" }} <!--filter使用方法-->  
{% f2 1 2 3 4 %} <!--simple\_tag使用方法-->

Filter和simple\_tag的区别

Filter支持模板语言的if条件

Simple\_tag不支持if条件

{% if "alex"|f3 %}  
 <h1>true</h1>  
{% else %}  
 <h1>false</h1>  
{% endif %}

1. mark\_safe

知道在tag中返回一串html，要在页面上展示结果，必须mark\_safe这串字符串。有时候mark出来的结果不是我想要的如：<h1><model: user></h1> 不想把括model:user的尖括号转义，但是转义了，页面上就看不到了。

这就必须了解mark\_safe的本质：

mark\_safe(s) 传一个字符串或者字节，如果s是一个SafeText对象就不用转换，判断是否是SafeText对象，是判断是否有\_\_html\_\_方法

mark\_safe本质上是返回一个SafeText对象或者是SafeBytes对象

SafeText类在django.utils.safestring，SafeText继承text\_type(本质是继承str或者unicode)

#### 静态文件

导入静态文件

#setting.py

STATIC\_URL="/static/"

STATICFILES\_DIRS=(

os.path.join(BASE\_DIR,"static/"),

)

#template

{% load staticfiles %}

{% static "js/common.js" %}

#### 母板的继承

子模块放到母模块

母—index.html 做大效果

{% block <blockName> %}

{% endblock %}

子---child.html 填充内容，对应到blockname相同的地方

{% extends ‘index.html’ %}

{% block <blockName> %}

...

{% endblock %}

#### 小组件包含

{% include ‘index.html’ %}

模板语言在服务器端执行，最终给浏览器的都是渲染完成的字符串

mark\_safe

默认：模板引擎向下转义一层，浏览器向上转义一层，等到的是原来的字符串  
mark\_safe: 如果模板引擎没有向下转换，但浏览器向上转换，会被翻译成html语言  
所以直接return回去的字符串或者在模板内(两个大括号或{%%}之间的)会原样输出，  
但是被mark\_safe修饰过就能被浏览器翻译成html语言  
  
默认机制的好处是：  
如果黑客在可输入框中输入html语句，提交后存到数据库，再次拿到时不会被翻译成html语句，html代码就无法植入  
如果不这么做，那么他想要插入什么语句就是什么语句，会被植入任何广告等等

flask的jinja也有转义

模板python2注意：

python2如果要打印对象的字符串格式，注意不是\_\_str\_\_，是\_\_unicode\_\_，python2的模板会把一个str(bytes)转换成unicode(str)

unicode("你好")和unicode(u"你好")一样

但是打印一个自定义对象

class C:

def \_\_init\_\_(self, name):

self.name = name

def \_\_str\_\_(self):

return self.name

obj = C("你好")

str(obj)或print(obj)只是调用\_\_str\_\_

python2默认字符串编码是unicode

### django数据库---ORM

1. 创建数据库表
2. 操作数据库数据

#### 数据库配置

项目的\_\_init\_\_.py中配置

import pymysql  
pymysql.install\_as\_MySQLdb()

setting.py中配置

DATABASES = {

'default': {

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

'NAME': 's16madking',

'USER': 'root',

'PASSWORD': '123456',

'HOST': '10.0.0.20',

'PORT': '3306',

}

}

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

数据库创建

**from** django.db **import** models  
  
**class UserList**(models.Model):  
 nid=models.AutoField(primary\_key=**True**)  
 name=models.CharField(max\_length=35)  
 email=models.EmailField()  
 text=models.TextField()

#### 类型

| 类型 | 参数 | 解释 | 说明 |
| --- | --- | --- | --- |
| AutoField | Primary key | 自增列 |  |
| CharField | Max\_length | 字符串列 |  |
| Boolean |  | 布尔列 | 相当于tinyint(1) |
| CommaSeparatedIntegerField | Max\_length | 逗号分隔的数字 | 相当于varchar() |
| DateField | Auto\_now | 时间列 | Auto\_now=True表示每次更新时间 |
| DateTimeField |  |  | 同dateField |
| Decimal |  | 十进制小数 |  |
| EmailField | Max\_length | 邮箱 | 实质Varchar() |
| FloatField |  | 浮点类型 | Double |
| IntegerField |  | 整数 | 整数系列：Bigxxx，smallxxx  positivexxx |
| GenericIPAddressField |  | Ipv4或ipv6 | Varchar |
| TextField |  | 长字符串 | longtext |
| URLField |  | 地址 | varchar |
| BinaryField |  | 二进制 |  |
| ImageField |  | 图片 |  |
| FielPathField |  | 文件路径 |  |

python manage.py makemigrations

python manage.py migrate

#### 时间字段

class TestModel(models.Model):  
 date=models.DateTimeField()

如果添加一条记录直接用datetime，会有提示：

models.TestModel.objects.create(date=datetime.datetime.now())

>>> models.TestModel.objects.create(date=datetime.datetime.now())

E:\MySoft\python35\lib\site-packages\django\db\models\fields\\_\_init\_\_.py:1447: **RuntimeWarning:** DateTimeField TestModel.date received a naive datetime (2017-08-03 20:35:19.116747) while time zone support is active.

RuntimeWarning)

为了避免这一问题，可以用django自带的时间格式

>>> from django.utils import timezone

>>> models.TestModel.objects.create(date=timezone.now())

加auto\_now属性是指在创建记录的时候自动添加当前时间

auto\_now\_add属性是在每次修改记录的时候添加当前时间

#### 参数：

| 参数 | 说明 |
| --- | --- |
| Null | 数据库字段中是否可以为空 |
| Blank | Admin中是否可以为空 |
| default | 默认值 |
| Primary\_key | 主键 |
| Auto\_now | 在DateTimeField中无论创建或添加，都为当前时间 |
| Auto\_now\_add | 自动设置为创建的时间 |
| choice | 元祖：数据库中的数据转换成可读信息  Gender=(  (1,boy),  (2,girl),  )  Choice=Gender |
| Verbose\_name | Admin中字段显示的名称 |
| name|Db\_column | 数据库中字段显示的名称 |
| unique | 不允许重复 |
| Db\_index | Db\_index=True 索引 |
| editable | Admin中是否可编辑 |
| Error\_massage | Admin中设置错误提示 |
| Help\_text | 在admin中显示帮助 |

#### 一对多

**class UserType**(models.Model):  
 caption=models.CharField(max\_length=20,verbose\_name="用户类型")  
 **def** \_\_str\_\_(self):  
 **return** '%s表'% (self.caption)

**class UserList**(models.Model):  
 user=models.CharField(max\_length=20,verbose\_name="用户名")  
 pwd=models.CharField(max\_length=20,verbose\_name="密码")

UserList连接UserType

在UserList中找该记录的UserType值

model.UserList.objects.filter(user\_type\_\_caption="xxx")

创建外键：

ForeignKey(DataClass)创建外键

user\_type=models.ForeignKey('UserType',verbose\_name="用户类型")

创建的外键实际上是外键的类的id

models.UserList.objects.filter(user\_type\_\_caption='普通用户').values()

<QuerySet [{'user\_type\_id': 3, 'pwd': '123456', 'user': 'euewrqe', 'id': 1}]>

models.UserList.objects.create(user=user\_dict['user'],pwd=user\_dict['pwd'],  
 user\_type\_id=3)

查找所有类型为普通用户的所有信息

models.UserList.objects.filter(user\_type\_id=3) #知道普通用户的id

双下划线

models.UserList.objects.filter(user\_type\_\_caption='普通用户').values() #不知道普通用户的id

filter和value里面用双下划线来取外键的字段，对象用点来取外键

#### 使用choice，固定值只要放在一张表内

get\_choice字段\_display

auto\_now\_add 添加时候的时间

auto\_now 添加和修改时候的时间

#### 多对多表

方法一：自己创建

**class Boys**(models.Model):  
 name=models.CharField(max\_length=32)  
**class Girls**(models.Model):  
 name=models.CharField(max\_length=32)  
  
**class B2G**(models.Model):  
 boy\_id=models.ForeignKey("Boys")  
 girl\_id=models.ForeignKey("Girls")

方法二：django自动创建

**class Boys**(models.Model):  
 name=models.CharField(max\_length=32)  
**class Girls**(models.Model):  
 name=models.CharField(max\_length=32)  
 boy=models.ManyToManyField("Boys")

如果boys有个对象b1,id=1,Girls对象g1关联boys：

1，使用对象

g1.conn.add(b1)

1. 使用id

G1.conn.add(1)

添加多个：

g1.conn.add(\*[])

查找：

G1.conn.all()/filter()/get()...

删除：

Clear()清空所有

Remove()指定删除

#### 数据库操作

创建一条数据

models.UserList.objects.create(user=user,pwd=pwd,email=email)

get\_or\_create()

models.NewAssetApprovalZone.objects.get\_or\_create(sn="00426-OEM-8992662-00401")

(<NewAssetApprovalZone: <资产sn号:00426-OEM-8992662-00400;资产信号:Win32\_OperatingSystem>>, False)

如果该内容已经有，则不再创建，返回False 如果没有创建 ，返回True

获取单条数据

models.UserList.objects.get(Users=user)

获取列：

userFlag.Pwds

修改列：

userFlag.Pwds='2254823'

values获取组成字典：

userFlag.values("Pwds","Emails")

得出：{'Pwds': '12345678', 'Emails': 'aaa@bbb.com'}

Values\_list获取组成元祖

userFlag.values\_list("Pwds","Emails")

得出：('12345678', 'aaa@bbb.com')

models数据库的操作分为

单个对象的操作----**理解为普通对象的操作，未直接对表进行操作，需要save()才会记录到表中**

增加一条数据：

obj=Model(name="xxx",age="xxx")

obj.save()

修改

obj.name="xxx"

obj.save()

删除

obj.delete() ----**这个是真正删除数据库的这条记录**

在python中对一切的del仅仅删除引用:

a=10

b=a

del a

print(b)

-----

b=10

对象是存放在内存中的，a和b仅仅是引用，提供访问的该内存的接口，当引用被删除后对象仍然在内存中，python中一切标识符皆是引用，所有引用都是引用对象，python对于垃圾回收的处理是，当该对象的引用数为0时就回收该对象。还有就是该引用在该定义域结束后删除

[]

C语言中

int a=10 a就是10存在的内存

int\* pa=&a pa存放a的内存地址，取用

query\_set的操作

获取所有符合的数据的数组

userFlag=models.UserList.objects.filter(Users=user)

>>返回UserList对象

获取列：

userFlag[0].Pwds

获取所有

userFlag=models.UserList.objects.all()

获取其它：

userFlag=models.UserList.objects.exclude(Users=user)

filter和all返回的类型是query\_set类似列表，query\_set内部是一个个model对象

get只返回一个model对象

count()显示一共有几条数据

delete()删除

update()更新数据

order\_by()按照那一列排序

userFlag=models.UserList.objects.filter(Users=user)  
userFlag.update(Pwds="12345678")

按照id列排序

userFlag.order\_by("id") esc

userFlag.order\_by("-id") desc

first()/last()从query\_set中找第一个或最后一个

**用双下划线获取返回**

userFlag=models.UserList.objects.filter(nid\_\_gt=20) #表示nid大于20的取出

类似的有\_\_gt

\_\_gte 大于等于

\_\_lt

\_\_lte

id\_\_in=[11,22,33]

\_\_contains="name" 模糊匹配，大小写敏感

\_\_icontains 模糊匹配，大小写不敏感

\_\_range=[1,10] 之间

startswith/endswith...

name\_\_regex=r"^game[0-9]$" 正则匹配

\_\_iregex

userFlag=models.UserList.objects.filter(nid\_\_gt=20,nid\_\_lt=30) 逗号表示且

limit\_by

userFlag=models.UserList.objects.all()[5:10]

转换成sql语句

userFlag.query

#### 数据库F和Q操作

from django.db.models import F

models.UserInfo.objects.update(salary=F("salary")+10)

在原来的基础上加数字

Q用来构建搜索条件

q1=Q()  
q1.connector = 'OR'  
q1.children.append(("id",1))  
q1.children.append(("id",2))

Q(xx=xx)|~Q(xx=xx)  
print(models.UserInfo.objects.filter(q1).values())

**append的内容必须是元祖或列表**

h1=Q()  
h1.add(q1,'AND')  
print(models.UserInfo.objects.filter(h1).values())

总结：构建搜索条件的方法

1. 传值

print(models.UserInfo.objects.filter(id=1,user="由岐姐").values())

1. 传字典

dic1={"id":1,"user":"由岐姐"}  
print(models.UserInfo.objects.filter(\*\*dic1).values())

1. 传Q对象

Q的或操作

models.wordBook.objects.filter(Q(word=srch)|Q(explanation=srch))

#### 数据库格式queryset转换成字典

from django.forms.models import model\_to\_dict

model\_to\_dict(query\_obj[0])

或用序列号的方式转换

#### 连表操作

连表预览

prefetch\_realated(...)

select\_realated(...)

一对多

class UserInfo(models.Model):  
 name=models.CharField(max\_length=32)  
 desc=models.TextField(max\_length=200)  
 email=models.CharField(max\_length=64,default="euewrqe@gmail.com")  
 fk\_usertype=models.ForeignKey('UserType',default=1)  
  
class UserType(models.Model):  
 name=models.CharField(max\_length=32)

UserInfo对象查UserType字段

mod\_ret=models.UserInfo.objects.filter(name\_\_regex=r"^game[0-9]$")  
print(mod\_ret.values("name","fk\_usertype\_\_name"))

print(mod\_ret.first().fk\_userinfo.name)

#已知外键查本体

mod\_ret=models.UserInfo.objects.filter(fk\_userinfo\_\_id=2)

已知UserType查UserInfo

ut=models.UserType.objects.filter(id=1)  
print(ut.values("name","userinfo\_\_name"))

#### 反向查找（也使用于多对多）

print(ut[0].userinfo\_set.all()) #查看所有关联userinfo对象

反响查找类

ut[0].userinfo\_set.model

#### 联合唯一

class Favor(models.Model):  
 nid = models.AutoField(primary\_key=True)  
 user\_info = models.ForeignKey('UserInfo')  
 news = models.ForeignKey('News')  
 ctime = models.DateTimeField()  
 #联合唯一  
 class Meta:  
 unique\_together = (("user\_info", "news"),)

修改一张表的表名

class Meta:  
 verbose\_name\_plural="用户表"

#每条记录名

def \_\_str\_\_(self)

return 'dddd'

修改记录名称：

class UserType(models.Model):  
 caption=models.CharField(max\_length=20,verbose\_name="用户类型")

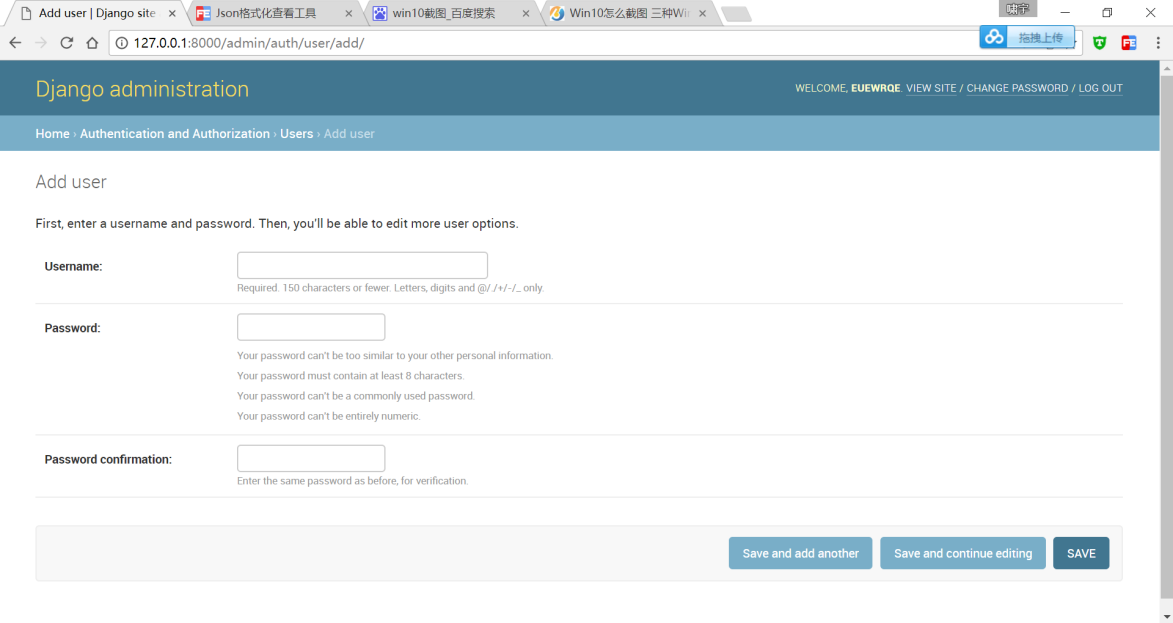
#verbose\_name在修改界面，每一项显示的名字  
 def \_\_str\_\_(self):  
 return '%s'% (self.caption)

#### obj.\_meta和class META

unique\_together=() 联合唯一

admin中的用户也是一张表，如果要继承这张表需要导入

from django.contrib.auth.models import User



model\_obj.\_meta.get\_field(field\_name)->field\_obj 用字符串拿字段对象

field\_obj.save\_form\_data(model\_obj,new\_data) 更新数据库，需要save

拿表名

>>> obj=models.Asset.objects.get(id\_\_gt=2)

>>> obj.\_meta.object\_name

'Asset'

知道model对象，取app名字:app\_label；取model名：model\_name

遍历所有字段

models.Account.\_meta.get\_fields()

数据库相关属性

models.Account.\_meta

#### 字段对象的属性：

choices和get\_choices()

choices只是拿到choice字段

get\_choices()可以拿到choice或外键的列表

>>> role\_field=models.Account.\_meta.get\_field("role")

>>> role\_field.get\_choices()

[('', '---------'), (1, 'manager'), (2, 'seller'), (3, 'teacher'), (4, 'student')]

app\_label

#### 数据迁移

#### 数据库日志记录

### admin

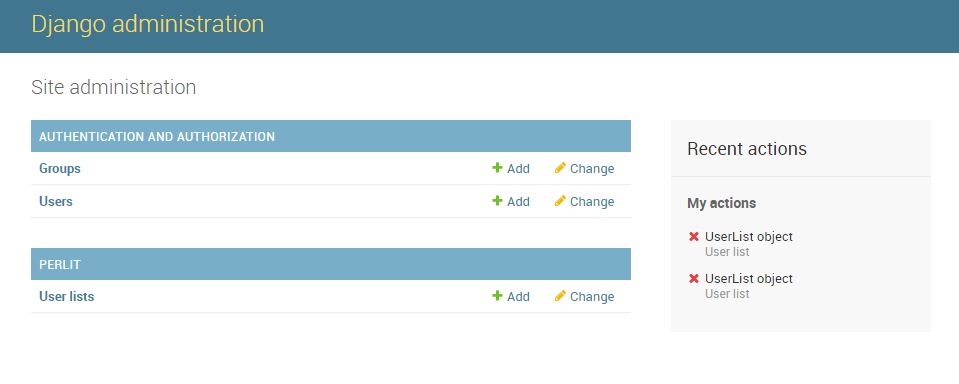
**from** app01 **import** models  
admin.site.register(models.UserType)

创建超级用户：

python manage.py createsuperuser

django的admin用于管理数据库，在admin模块中导入modals模块，然后添加一条数据库信息。

admin.site.register(models.UserList)



显示多个字段

list\_display=[]

让每张表显示更多字段

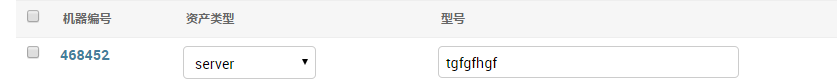
list\_display = ('sn', 'asset\_type', 'model')

分类显示

list\_filter = ('asset\_type', )

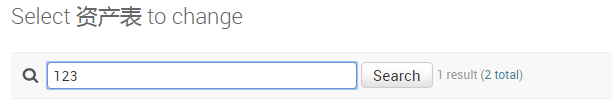


list\_editable = ('asset\_type','model',) 批量修改



搜索条按照先后顺序搜索

search\_fields = ('sn', 'asset\_type')

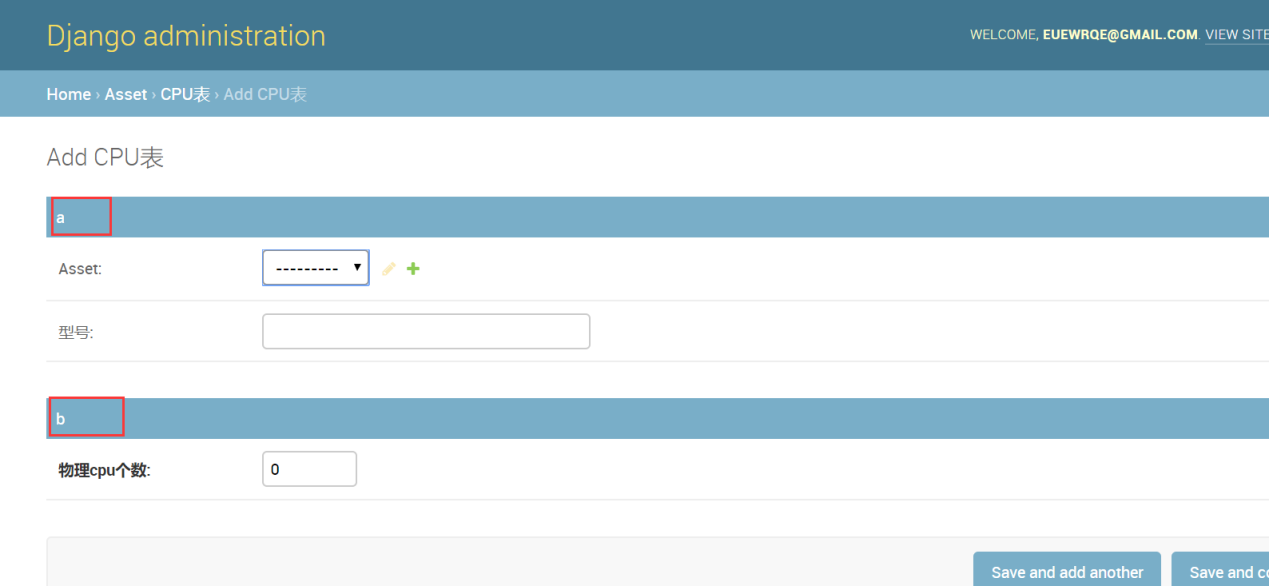


在批量操作框中自定义

actions = ['asset\_approval', ]  
  
def asset\_approval(self, request, querysets):  
 pass

field\_set

class CPUAdmin(admin.ModelAdmin):  
 fieldsets = (  
 ("a",{"fields":("asset","model",)}),  
 ("b",{"fields":("cpu\_count",)})  
 )



ordering = ("cpu\_count",) #按什么排序

### model高级方法

一个model

class Account(models.Model):

name=models.CharField(max\_length=128)

class\_at=models.ManyToManyField()

class SchoolClass(models.Model):

name=models.CharField(max\_length=128)

mod\_obj=Account.objects.first() #从数据库中拿第一条记录

mod\_obj=Account() #创建一条新的记录，记录是类Account的实例，可以对字段进行操作

mod\_obj.name='alex'

mod\_obj.save() #保存这条新纪录

\_meta属于该类的属性，所以用类或对象都能访问\_meta，这里面的属性在class Meta中有一部分会定义，比如unique\_togethor /index\_together

**pk是实例的主键，如果不知道主键的名字，可以用这个**

可以看一下该属性的所有成员

#### \_meta

['FORWARD\_PROPERTIES', 'REVERSE\_PROPERTIES', '\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_

\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_module\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_siz

eof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_weakref\_\_', '\_expire\_cache', '\_forward\_fields\_map', '\_get\_fields', '\_get\_fields\_cache', '\_ordering\_clash', '\_popula

te\_directed\_relation\_graph', '\_prepare', '\_property\_names', '\_relation\_tree', 'abstract', 'add\_field', 'add\_manager', 'app\_config', 'app\_label', 'apps', 'aut

o\_created', 'auto\_field', 'base\_manager', 'base\_manager\_name', 'can\_migrate', 'concrete\_fields', 'concrete\_model', 'contribute\_to\_class', 'db\_table', 'db\_tab

lespace', 'default\_apps', 'default\_manager', 'default\_manager\_name', 'default\_permissions', 'default\_related\_name', 'fields', 'fields\_map', 'get\_ancestor\_lin

k', 'get\_base\_chain', 'get\_field', 'get\_fields', 'get\_latest\_by', 'get\_parent\_list', 'get\_path\_from\_parent', 'get\_path\_to\_parent', 'has\_auto\_field', 'index\_t

ogether', 'indexes', 'installed', 'label', 'label\_lower', 'local\_concrete\_fields', 'local\_fields', 'local\_managers', 'local\_many\_to\_many', 'managed', 'manage

r\_inheritance\_from\_future', 'managers', 'managers\_map', 'many\_to\_many', 'model', 'model\_name', 'object\_name', 'order\_with\_respect\_to', 'ordering', 'original\_

attrs', 'parents', 'permissions', 'pk', 'private\_fields', 'proxy', 'proxy\_for\_model', 'related\_fkey\_lookups', 'related\_objects', 'required\_db\_features', 'req

uired\_db\_vendor', 'select\_on\_save', 'setup\_pk', 'setup\_proxy', 'swappable', 'swapped', 'unique\_together', 'verbose\_name', 'verbose\_name\_plural', 'verbose\_nam

e\_raw', 'virtual\_fields']

model.\_meta.app\_label---向上找app

db\_table---找app\_自身类名字

model\_name----表名字 (无大写)

model----表

object\_name 类名字

fields所有可见的字段

many\_to\_many/local\_many\_to\_many所有可见的m2m字段

local\_fields所有本表可见的字段

**permissions权限控制**

找到所有被关联的外键(set)--related\_name，多对多，一对一

fields\_map，当中的每个值是对方的rel对象，to属性是对方的属性，所以永远指向mod\_obj的类。对方的该对象有什么属性，该对象就有什么属性，比如related\_name，但还是本类的成员，model也是本类的名字

>>> mod\_obj.\_meta.fields\_map

{'logentry': <ManyToOneRel: admin.logentry>, 'account\_salary': <ManyToOneRel: crm.salaryprocessor>, 'Account\_class\_at+': <ManyToOneRel: crm.account\_class\_at>

, 'customer': <ManyToOneRel: crm.customer>, 'sessionrecord': <ManyToOneRel: crm.sessionrecord>, 'recommand': <ManyToOneRel: crm.recommand>, 'schoolclass': <M

anyToOneRel: crm.schoolclass>, 'Account\_good\_at+': <ManyToOneRel: crm.account\_good\_at>, 'account\_success': <ManyToOneRel: crm.sellersuccesser>, 'account\_talk

': <ManyToOneRel: crm.sellerprocessor>, 'stu\_to\_stu': <ManyToOneRel: crm.recommand>, 'studyrecord': <ManyToOneRel: crm.studyrecord>}

#### ManyToOneRel:

['auto\_created', 'concrete', 'db\_type', 'editable', 'field', 'field\_name', 'get\_accessor\_name', 'get\_cache\_name', 'get\_choices', 'ge

t\_extra\_restriction', 'get\_internal\_type', 'get\_joining\_columns', 'get\_lookup', 'get\_path\_info', 'get\_related\_field', 'hidden', 'is\_hidden', 'is\_relation', '

limit\_choices\_to', 'many\_to\_many', 'many\_to\_one', 'model', 'multiple', 'name', 'null', 'on\_delete', 'one\_to\_many', 'one\_to\_one', 'parent\_link', 'related\_mode

l', 'related\_name', 'related\_query\_name', 'remote\_field', 'set\_field\_name', 'symmetrical', 'target\_field', 'to']

target\_field --- 对方的类的主键

field --- 找到对方的该关联字段

get\_choice --- 直接获取对方的所有id和name

related\_model --- 直接找到对方的类

get\_internal\_type --- 该字段的类型

get\_field("role") ---通过字段名拿到字段对象

get\_fields 获取所有字段的对象

>>> mod\_obj.\_meta.get\_fields()

(<ManyToOneRel: admin.logentry>, <ManyToOneRel: crm.customer>, <ManyToOneRel: crm.sellerprocessor>, <ManyToOneRel: crm.sellersuccesser>, <ManyToOneRel: crm.s

alaryprocessor>, <ManyToOneRel: crm.recommand>, <ManyToOneRel: crm.recommand>, <ManyToOneRel: crm.schoolclass>, <ManyToOneRel: crm.sessionrecord>, <ManyToOne

Rel: crm.studyrecord>, <django.db.models.fields.AutoField: id>, <django.db.models.fields.CharField: password>, <django.db.models.fields.DateTimeField: last\_l

ogin>, <django.db.models.fields.EmailField: email>, <django.db.models.fields.CharField: name>, <django.db.models.fields.IntegerField: salary>, <django.db.mod

els.fields.related.ForeignKey: role>, <django.db.models.fields.CharField: directory>, <django.db.models.fields.DateTimeField: date\_of\_birth>, <django.db.mode

ls.fields.BooleanField: is\_active>, <django.db.models.fields.BooleanField: is\_admin>, <django.db.models.fields.related.ManyToManyField: good\_at>, <django.db.

models.fields.related.ManyToManyField: class\_at>)

----

一部分是ManyToOneRel 是被关联的外键字段

反向查找的ManyToManyRel through 查找中介表

但是正向查找不是Rel，多对多字段有rel属性，调到对方的Rel字段再进行rel相关操作

#### field

['\_check\_backend\_specific\_checks', '\_check\_choices', '\_check\_db\_index', '\_check\_deprecation\_details', '\_check\_field\_name

', '\_check\_null\_allowed\_for\_primary\_keys', '\_check\_primary\_key', '\_clear\_cached\_lookups', '\_description', '\_error\_messages', '\_get\_default', '\_get\_flatchoice

s', '\_get\_lookup', '\_get\_val\_from\_obj', '\_unique', '\_unregister\_lookup', '\_validators', '\_verbose\_name', 'attname', 'auto\_created', 'auto\_creation\_counter',

'blank', 'cached\_col', 'check', 'choices', 'class\_lookups', 'clean', 'clone', 'column', 'concrete', 'contribute\_to\_class', 'creation\_counter', 'db\_check', 'd

b\_column', 'db\_index', 'db\_parameters', 'db\_tablespace', 'db\_type', 'db\_type\_suffix', 'deconstruct', 'default', 'default\_error\_messages', 'default\_validators

', 'description', 'editable', 'empty\_strings\_allowed', 'empty\_values', 'error\_messages', 'flatchoices', 'formfield', 'get\_attname', 'get\_attname\_column', 'ge

t\_cache\_name', 'get\_choices', 'get\_col', 'get\_db\_converters', 'get\_db\_prep\_save', 'get\_db\_prep\_value', 'get\_default', 'get\_filter\_kwargs\_for\_object', 'get\_in

ternal\_type', 'get\_lookup', 'get\_lookups', 'get\_pk\_value\_on\_save', 'get\_prep\_value', 'get\_transform', 'has\_default', 'help\_text', 'hidden', 'is\_relation', 'm

any\_to\_many', 'many\_to\_one', 'max\_length', 'merge\_dicts', 'model', 'name', 'null', 'one\_to\_many', 'one\_to\_one', 'pre\_save', 'primary\_key', 'register\_lookup',

'rel', 'rel\_db\_type', 'related\_model', 'remote\_field', 'run\_validators', 'save\_form\_data', 'select\_format', 'serialize', 'set\_attributes\_from\_name', 'system

\_check\_deprecated\_details', 'system\_check\_removed\_details', 'to\_python', 'unique', 'unique\_for\_date', 'unique\_for\_month', 'unique\_for\_year', 'validate', 'val

idators', 'value\_from\_object', 'value\_to\_string', 'verbose\_name']

**字段是类的字段---是全局的**

verbose\_name----

model---通过字段找类

column---该字段的名字

on\_delete ---外键相关，对方是否能直接删除

get\_internal\_type --- 该字段的类型(CharField,ForiegnField...)

value\_from\_object(model\_obj) ---取到该字段的值

value\_to\_string(model\_obj)

save\_form\_data(model\_obj,value) ---保存value给model\_obj

字段被修改后必须save()生效

**foreignkey字段的属性：**

related\_name 用来让多对一反查用

一张表为M1，字段有m2=ForeignKey("M2")关联M2表

obj\_m2=M2.objects.first()

想要查找关联obj\_m2该记录的所有m1的表，表名加\_set

obj\_m2.m1\_set.all() #有all必定有filter，delete属性

related\_name取代m1\_set作为别名：如related\_name="mm1",可以obj\_m2.mm1.all()

用了别名后\_set无效

related\_query\_name #filter反查外键符合条件的本表记录

如果反向filter，是表名\_\_字段

如：

BModel.objects.filter(testmodel\_\_name="alex")

定义了fk的related\_query\_name=bm

BModel.objects.filter(bm\_\_name="alex")

**manytomany字段的属性：**

对于many\_to\_many的related\_name和related\_query\_name和外键的一样

though 中介表相关

>>> mod\_obj.\_meta.fields\_map['customer'] #本类的rel成员

<ManyToOneRel: crm.customer>

>>> mod\_obj.\_meta.fields\_map['customer'].field #对方的field成员

<django.db.models.fields.related.ForeignKey: account>

### auth自定制

auth相关源码在django.contrib.auth下面

该文档相关：

[https://docs.djangoproject.com/en/1.11/topics/auth/customizing/#a-full-example](https://docs.djangoproject.com/en/1.11/topics/auth/customizing/%23a-full-example)

自定义用户之前，在setting中配上：项目名.用户类

AUTH\_USER\_MODEL = 'app01.UserInfo'

1. 用户需要自定制user类，需要继承AbstractBaseUser类，在user类下面按照一般model定义字段
2. 需要重写几个函数：

用来显示用户：

get\_full\_name

get\_short\_name---必重写

\_\_str\_\_

-----该两项用来显示app和表

has\_perm

has\_module\_perms

需要重写的几个字段：

is\_active 能否使用该用户

is\_staff 能否使用该用户登陆后台

is\_admin

USERNAME\_FIELD="" 根据那个字段作为用户名

REQUIRED\_FIELDS=[] 创建超级用户时自动添加哪几项，可以没有

**class** Account(AbstractBaseUser,PermissionsMixin):  
 *'''登陆用户'''*

email = models.EmailField(  
 verbose\_name=**'email address'**,  
 max\_length=255,  
 unique=**True** )  
 name = models.CharField(max\_length=128,unique=**True**)  
 date = models.DateTimeField(auto\_now=**True**)  
 is\_active = models.BooleanField(default=**True**)  
 is\_staff=models.BooleanField(default=**True**,)  
 is\_admin = models.BooleanField(default=**False**)  
 objects = MyUserManager()  
 *#这个必须是unique* USERNAME\_FIELD = **'email'** *#创建时要填的内容* REQUIRED\_FIELDS = [**'name'**]  
 bindhostuser=models.ManyToManyField(**'BindHostUser'**)  
 host\_group=models.ManyToManyField(**'HostGroup'**)  
  
 *# def get\_full\_name(self):  
 # # The user is identified by their email address  
 # return self.email* **def** get\_short\_name(self):  
 *# The user is identified by their email address* **return** self.email  
  
 **def** \_\_str\_\_(self): *# \_\_unicode\_\_ on Python 2* **return** self.email  
 *#* **def** has\_perm(self, perm, obj=**None**):  
 *"Does the user have a specific permission?"  
 # Simplest possible answer: Yes, always* **return True** *#* **def** has\_perms(self, perm, obj=**None**):  
 *"Does the user have a specific permission?"  
 # Simplest possible answer: Yes, always* **return True  
  
 def** has\_module\_perms(self, app\_label):  
 *"Does the user have permissions to view the app `app\_label`?"  
 # Simplest possible answer: Yes, always* **return True**

创建用户和超级用户时配置：输入的密码一定要set\_password加密

**class** MyUserManager(BaseUserManager):  
 **def** create\_user(self,email,name, password=**None**):  
 *"""  
 Creates and saves a User with the given email, date of  
 birth and password.  
 """* user = self.model(  
 email=email,  
 name=name,  
 )  
 user.set\_password(password)  
  
 user.save(using=self.\_db)  
 **return** user  
  
 **def** create\_superuser(self,email, password, name):  
 *"""  
 Creates and saves a superuser with the given email, date of  
 birth and password.  
 """* user = self.create\_user(  
 email=email,  
 password=password,  
 name=name,  
 )  
 user.is\_admin = **True** user.save(using=self.\_db)  
 **return** user

在admin中配置auth

1. 定制添加时表单 继承form.ModelForm
2. 定制修改时表单

from django.contrib.auth.admin import UserAdmin as BaseUserAdmin

### 借用系统用户登陆

### Form表单处理

#### form模块的成员

**from** django.core.exceptions **import** ValidationError *# NOQA***from** django.forms.boundfield **import** \* *# NOQA***from** django.forms.fields **import** \* *# NOQA***from** django.forms.forms **import** \* *# NOQA***from** django.forms.formsets **import** \* *# NOQA***from** django.forms.models **import** \* *# NOQA***from** django.forms.widgets **import** \* *# NOQA*

#### 生成表单验证

Django自带表单处理

from django import forms

新建一个表单验证规则的类

class LoginForm(forms.Form):  
 user=forms.CharField(required=True)  
 pwd=forms.CharField(required=True)

字段名为html的name值

拿到表单验证对象

obj=LoginForm(request.POST)

判断验证结果

ret=obj.is\_valid()

返回true或false

当表单正确的时候，会把表单内容以字典的形式返回

obj.clean()

当表单验证错误时，把表单详细错误列出来

obj.errors

ErrorDict为错误信息字典，有几个方法：as\_data，as\_json，as\_ul，as\_text

默认输出是as\_ul

print(obj.errors.as\_json())

得出

{"pwd": [{"code": "required", "message": "This field is required."}]}

然后把此json内容发送给客户端，由js来解析，放到html页面上

$.each(data,function (k,v) {  
 var insertHt="<span class='error'>"+v[0].message+"</span>";  
 $("[name="+k+"]").after(insertHt);  
})

#### 字段类型：

\_\_all\_\_ = (  
 'Field', 'CharField', 'IntegerField',  
 'DateField', 'TimeField', 'DateTimeField', 'DurationField',  
 'RegexField', 'EmailField', 'FileField', 'ImageField', 'URLField',  
 'BooleanField', 'NullBooleanField', 'ChoiceField', 'MultipleChoiceField',  
 'ComboField', 'MultiValueField', 'FloatField', 'DecimalField',  
 'SplitDateTimeField', 'GenericIPAddressField', 'FilePathField',  
 'SlugField', 'TypedChoiceField', 'TypedMultipleChoiceField', 'UUIDField',  
)

常用

CharField

IntergerField

EmailField

ChoiceField

FileField

参数

required是否可以为空

widget定制表单标签

error\_message 错误映射

使用

obj=UserForm()

obj可以直接传到模板渲染成html代码

当提交数据时，传入参数给UserForm同时能后台验证，验证的结果

obj=is\_valid()判断是否有错误

错误信息都放在obj.errors对象中

用户传入的键值都放在obj.cleaned\_data对象中

初始化时可以将一些值传入前端，可以做到更新页面的同时用户输入内容保留。

obj=UserForm(initial={"name":"fdsfds"})

选择框：

#### 配置规则error\_message

user=forms.CharField(required=True,min\_length=6,max\_length=12,error\_messages={"required":"用户不能为空","min\_length":"最小长度为6","max\_length":"最大长度为12"})

内置规则：

EmailField必须为邮箱格式，IntegerField必须为数字

默认规则的格式错误对应的名称为invalid

email=forms.EmailField(error\_messages={"invalid":"邮箱格式错误"})

#### 自定制规则：

from django.core.exceptions import ValidationError  
import re  
def phone\_format(value):  
 phone\_re=re.compile(r"^[0-9]{3}-[0-9]{3}-[0-9]{4}$")  
 if not phone\_re.match(value):  
 raise ValidationError("手机格式错误")

导入

phone=forms.CharField(validators=[phone\_format,],)

验证结果：数据库的内容都放在instance，表单传入的结果都放在clean\_data里

clean用来验证每个值，

def clean(self, value):  
 """  
 Validates the given value and returns its "cleaned" value as an  
 appropriate Python object.  
  
 Raises ValidationError for any errors.  
 """  
 value = self.to\_python(value)  
 self.validate(value)  
 self.run\_validators(value)  
 return value

run\_validator()

def run\_validators(self, value):  
 if value in self.empty\_values:  
 return  
 errors = []  
 for v in self.validators:  
 try:  
 v(value)  
 except ValidationError as e:  
 if hasattr(e, 'code') and e.code in self.error\_messages:  
 e.message = self.error\_messages[e.code]  
 errors.extend(e.error\_list)  
 if errors:  
 raise ValidationError(errors)

to\_python强制类型转换，self.validate验证是否为空:判断self.required布尔值，run\_validators在可以为空的字段，如果是空就排除。

self.validators下面，每一个都调用一遍。不断抛异常的过程

#### 自动生成HTML标签

可根据匹配规则自动生成HTML标签。

obj1=LoginForm()

return render(request,'login.html',{'obj1':obj1})

obj1是一个表单组，直接放到html就能生成表单，如果循环obj1可以拿出每一个表单

{% for form in form\_objs %}

{{ form.name }} //form.field&form.label\_tag可拿出label属性 直接form就是每一个表单。

{% endfor %}

在html中：

<div>  
 用户名：{{ obj1.user }}  
</div>

#### 自定义HTML标签

from django.forms import widgets

widgets.TextInput(attrs={'class': 'form-control'})

TextInput

EmailInput

UrlInput

PasswordInput

FileInput

TextArea

### ModelForm

**class** AccountForm(forms.ModelForm):  
  
 **class** Meta:  
 model = models.Account  
 fields = [**'email'**,**'name'**, **'password'**,**'salary'**,**'directory'**] #\_\_all\_\_

form带一个参数：novalidate

#### modelform一些自定制

与数据库字段有关的配置都在Meta里进行：

model： 参照的数据库

fields： 包含的字段

exclude： 排除哪些字段

labels： 提示信息

widgets： 对话框

error\_messages： 自定义错误信息

field\_classes： 自定义字段类

forms.UserSettingForm(data=request.POST, instance=user\_obj)

如果有instance传入，就是修改

如果没有instance就是添加

同时instance会让form保存每个字段的值，在前端，如果数据库有值，就会显示

initial只是单纯把值传给前端

在modelforms上可自定制额外字段

save在源码中做了这些事：

def save(self, commit=True):  
 if commit:  
 # If committing, save the instance and the m2m data immediately.  
 self.instance.save() #保存当前数据库对象  
 self.\_save\_m2m() #同时保存多对多字段  
 else:  
 # If not committing, add a method to the form to allow deferred  
 # saving of m2m data.  
 self.save\_m2m = self.\_save\_m2m  
 return self.instance

可重写save做一些自定制操作

### form高级方法

表单：

**from** django **import** forms  
  
**class** CommentForm(forms.Form):  
 name = forms.CharField()  
 url = forms.URLField()  
 comment = forms.CharField()

form\_obj=test.CommentForm() #整个表单的对象

def \_\_init\_\_(self, data=None, files=None, auto\_id='id\_%s', prefix=None,  
 initial=None, error\_class=ErrorList, label\_suffix=None,  
 empty\_permitted=False,field\_order=None,use\_required\_attribute=None, renderer=None):

实例化表单 \_\_str\_\_是整个表单的html

如果传入一组数据，会进行验证

>>> form\_obj=test.CommentForm(data={"name":"alex","url":"baidu.com/","comment":""})

>>> form\_obj.errors

{'comment': ['This field is required.']}

#### 整个表单的对象

['\_bound\_fields\_cache', '\_clean\_fields', '\_clean\_form', '\_errors', '\_html\_output', '\_post\_clean', 'add\_error', 'add\_initial\_prefix', 'add\_prefix', 'as\_p', 'as\_table', 'as\_ul', 'auto\_id', 'base\_fields', 'changed\_data', 'clean', 'data', 'declared\_fields', 'default\_renderer', 'empty\_permitted', 'error\_class', 'errors', 'field\_order', 'fields', 'files', 'full\_clean', 'get\_initial\_for\_field', 'has\_changed', 'has\_error', 'hidden\_fields', 'initial', 'is\_bound', 'is\_multipart', 'is\_valid', 'label\_suffix', 'media', 'non\_field\_errors', 'order\_fields', 'prefix', 'renderer', 'use\_required\_attribute', 'visible\_fields']

整个表单是由一个个表单元素组成：

form\_obj["name"]

[i for i in form\_obj] <obj1,obj2>

每个表单元素又有共通的属性

['as\_hidden', 'as\_text', 'as\_textarea', 'as\_widget', 'auto\_id', 'build\_widget\_attrs', 'css\_classes

', 'data', 'errors', 'field', 'form', 'help\_text', 'html\_initial\_id', 'html\_initial\_name', 'html\_name', 'id\_for\_label', 'initial', 'is\_hidden', 'label', 'lab

el\_tag', 'name', 'subwidgets', 'value']

name---name是表单定义时的字段名字，默认也是html表单的name属性

value()---传入前端的value的值

data---每一个元素传入的值，data不可修改，紧在对象创建时写入

label---为标签名字，是表单的字段名字(开头大写)，如果是ModelForm优先考虑数据库的字段的verbose

html\_name---可修改的name属性，也是id的id\_<name>的这个name

auto\_id时这个字段的id值

form---回到大form对象

tag---解析成html--表单元素(一个个input)

as\_widget---查看表单元素

**subwidgets---多个绑定的widget对象**

['choice\_label', 'data', 'id\_for\_label', 'parent\_widget', 'renderer', 'tag', 'template\_name']

parent\_widget----该绑定的widget对象的父级widget，如

>>> form\_obj["name"].subwidgets[0].parent\_widget

<django.forms.widgets.TextInput object at 0x0000027C9D222CC0>

**renderer---渲染对象**

['backend', 'engine', 'get\_template', 'render']

render(template\_name,context)

**field---该表单元素对应的字段，内部有定义该字段的属性**

#### form\_obj["name"].field 表单元素的字段对象

['bound\_data', 'clean', 'creation\_counter', 'default\_error\_messages', 'default\_validators', 'disabled', 'empty\_value', 'empty\_values', 'error\_messages', 'get\_bound\_field', 'has\_changed', 'help\_text', 'hidden\_widget', 'initial', 'label', 'label\_suffix', 'localize', 'max\_length', 'min\_length', 'prepare\_value', 'required', 'run\_validators', 'show\_hidden\_initial', 'strip', 'to\_python', 'validate', 'validators', 'widget', 'widget\_attrs']

required---判断该字段是否允许为空，类似验证方法max\_length/min\_length

clean()--- 强制转换，变成正确的表单值

to\_python()---clean() to\_python是每个字段的强制转换方法

validate() ---传入一个自定义验证

validator ---- 所有自定义验证的字典

#### field.widget 字段的表单配置

['\_format\_value', '\_render', 'attrs', 'build\_attrs', 'format\_value', 'get\_context', 'id\_for\_label', 'input\_type', 'is\_hidden', 'is\_localized', 'is\_required', 'media', 'needs\_multipart\_form', 'render', 'subwidgets', 'supports\_microseconds', 'template\_name', 'use\_required\_attribute', 'value\_from\_datadict', 'value\_omitted\_from\_data']

attr---定义时配置的html属性，是一个字典

format\_value---value的强制转换，方法，任何传入值，如果是TextInput则转换成字符串

is\_hidden/is\_required---html的一些属性属性

**input\_type---传入的值的类型text/url/email...**

>>> form\_obj["name"].field.widget.input\_type

'text'

**media--- 一些静态的配置**

['\_css', '\_js', 'absolute\_path', 'add\_css', 'add\_js', 'render', 'render\_css', 'render\_js']

needs\_multipart\_form ---- xxx

render---表单对象的渲染

>>> form\_obj["url"].field.widget.render("ddd","fff")

'<input type="url" name="ddd" value="fff" />'

form\_objs["<字段>"]=字段的form对象

<input type="text" name="name" value="真红老婆" required class="form-control" id="id\_name" maxlength="128" />

form\_obj.instance的所有属性，表中的记录的对象

['DoesNotExist', 'Meta', 'MultipleObjectsReturned', 'REQUIRED\_FIELDS', 'USERNAME\_FIELD', '\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_module\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_setstate\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_weakref\_\_', '\_check\_column\_name\_clashes', '\_check\_field\_name\_clashes', '\_check\_fields', '\_check\_id\_field', '\_check\_index\_together', '\_check\_local\_fields', '\_check\_long\_column\_names', '\_check\_m2m\_through\_same\_relationship', '\_check\_managers', '\_check\_model', '\_check\_model\_name\_db\_lookup\_clashes', '\_check\_ordering', '\_check\_swappable', '\_check\_unique\_together', '\_do\_insert', '\_do\_update', '\_get\_FIELD\_display', '\_get\_next\_or\_previous\_by\_FIELD', '\_get\_next\_or\_previous\_in\_order', '\_get\_pk\_val', '\_get\_unique\_checks', '\_meta', '\_password', '\_perform\_date\_checks', '\_perform\_unique\_checks', '\_save\_parents', '\_save\_table', '\_set\_pk\_val', '\_state', 'account\_salary', 'account\_success', 'account\_talk', 'check', 'check\_password', 'class\_at', 'clean', 'clean\_fields', 'customer\_set', 'date\_error\_message', 'date\_of\_birth', 'delete', 'directory', 'email', 'from\_db', 'full\_clean', 'get\_deferred\_fields', 'get\_email\_field\_name', 'get\_full\_name', 'get\_next\_by\_date\_of\_birth', 'get\_previous\_by\_date\_of\_birth', 'get\_session\_auth\_hash', 'get\_short\_name', 'get\_username', 'good\_at', 'has\_module\_perms', 'has\_perm', 'has\_perms', 'has\_usable\_password', 'id', 'is\_active', 'is\_admin', 'is\_anonymous', 'is\_authenticated', 'is\_staff', 'last\_login', 'logentry\_set', 'name', 'natural\_key', 'normalize\_username', 'objects', 'password', 'pk', 'prepare\_database\_save', 'recommand\_set', 'refresh\_from\_db', 'role', 'role\_id', 'salary', 'save', 'save\_base', 'schoolclass\_set', 'serializable\_value', 'sessionrecord\_set', 'set\_password', 'set\_unusable\_password', 'show\_field', 'stu\_to\_stu', 'studyrecord\_set', 'unique\_error\_message', 'validate\_unique']

form\_obj.add\_error

ValidationError

### CSRF

Csrf是跨站请求伪造，比方我有一个网站127.0.0.1，然后我再创一个表单网页，该表单提交到www.baidu.com就是跨站请求。

@csrf\_exempt

开启csrf后，模板网页必须要有以下代码，才可以提交数据

{% csrf\_token %}

当第一次用get请求过来，这段代码自动生成一个input标签，当post提交表单时，带着这个过去，后端就让通过

<input type="hidden" name="csrfmiddlewaretoken" value="vqiU1rQvA6fv6LeAZIg3zMIjVl9EKJOwLrj1RpJrVodFFZy2qWUUFfB2HNHxAjVy">

没有这一段，或者这段值被修改了，就通不过。

实质是一个csrf键和随机字符串的值

如果把这一段复制到我的网页上也是可以通过的，也就是只要带着这段过去，也能通过。

同时csrf还会生成一个cookie：

Django的csrf流程：

第一次请求，返回的时候到达中间件CSRF组件，如果是GET，返回csrf\_token放到cookie里，或者{% CSRF\_TOKEN %} 模板渲染

第二次请求，只要是POST，验证是否携带csrf\_token，如果没有，就是非法请求，返回403

#### 通过Ajax进行csrf认证

方法一：直接从生成的 input中取csrf的简直就行

方法二：jquery对于form表单有自动提取内部数据得功能：

$('form').serilize()

返回的内容是字符串：k1=v1&k2=v2

方法三：从cookie中取csrftoken的值

Jquery的cookie插件：

<script src="/static/plugin/jquery/jquery.cookie.js"></script>

var csrftoken = $.cookie('csrftoken'); #拿到指定cookie

得放到请求头

function csrfSafeMethod(method) {  
 // these HTTP methods do not require CSRF protection  
 return (/^(GET|HEAD|OPTIONS|TRACE)$/.test(method));  
}

//只要设置一遍，其它所有的ajax都会带上  
$.ajaxSetup({  
 beforeSend: function(xhr, settings) {  
 if (!csrfSafeMethod(settings.type) && !this.crossDomain) {  
 xhr.setRequestHeader("X-CSRFToken", csrftoken);  
 }  
 }  
});

放入请求头部

$.ajax({

headers:{"X-CSRFToken": csrftoken}

})

#### Csrf的影响范围设置

在setting.py的MIDDLEWARE中设置，csrf是一个中间件

'django.middleware.csrf.CsrfViewMiddleware',

表示全局都使用csrf

在view导入from django.views.decorators.csrf import csrf\_exempt,csrf\_protect

可以使用以下装饰器：

@csrf\_protect当前url强制设置成需要csrf认证

@csrf\_exempt取消当前url的csrf认证

如何判断这段字符符合

### CSRF攻击和XSS攻击

CSRF为跨站请求伪造，依靠信赖用户标识(cookie)。

1. cookie是表示用户和网站之间的信任的，用cookie可以进行持久化登录，但是cookie可以在任意浏览器和机器中使用，如果cookie被盗用，其他人也能无需帐号密码进行登录。

csrf攻击是指恶意网站盗用用户身份，以用户名义发送恶意请求，数据提交不在你控制的范围内。

原理：

A为受信任的网站 ，B为恶意网站，C为用户

1. C浏览登录A网站
2. 验证通过，在C的浏览器中产生cookie，
3. 用户在没有登出A的情况下，访问B网站
4. B网站发出一个请求，要求C访问A
5. 根据B的请求，C浏览器带着原来的cookie进行访问A网站

假设A网站是银行。

一种：

A网站以GET方式请求完成银行转账，如：

http://www.mybank.com/Transfer.php?toBankId=11&money=1000

B网站刚好有这段代码

<img src=http://www.mybank.com/Transfer.php?toBankId=11&money=1000>

首先你登录了A网站，拿到了cookie，然后访问B网站，你会发现你少了1000块钱。以上这种是极其不规范的HTTP写法。不应该把数据直接写在GET中

1. 手动请求B网站。
2. B网站自动让你的浏览器访问A，而你携带着cookie。所以这个请求就执行成功

第二种：

银行升级，改用POST进行提交

<?php  
　　　　session\_start();  
　　　　if (isset($\_POST['toBankId'] &&　isset($\_POST['money']))  
　　　　{  
　　　　    buy\_stocks($\_POST['toBankId'],　$\_POST['money']);  
　　　　}  
　　?>

<http://www.mybank.com/Transfer.php>

Body:

toBankId=11

Money = 1000

csrf

B网站也升级，这段还是自动提交，但是以POST

<html>  
　　<head>  
　　　　<script type="text/javascript">  
　　　　　　function steal()  
　　　　　　{  
          　　　　 iframe = document.frames["steal"];  
　　     　　      iframe.document.Submit("transfer");  
　　　　　　}  
　　　　</script>  
　　</head>  
  
　　<body onload="steal()">  
　　　　<iframe name="steal" display="none">  
　　　　　　<form method="POST" name="transfer"　action="http://www.myBank.com/Transfer.php">  
　　　　　　　　<input type="hidden" name="toBankId" value="11">  
　　　　　　　　<input type="hidden" name="money" value="1000">  
　　　　　　</form>  
　　　　</iframe>  
　　</body>  
</html>

你点击该网站还是自动以你的名义提交。你还是少了1000

预防措施：

服务器自动生成给你一个临时的csrf cookie，在你的form表单中，正常提交会携带该cookie的值，后端会对该值进行处理，如果认证成功，说明是你操作的。

如果是B网站生成的网页，B网站无法获知csrfcookie的值，这样就访问失败，也就是必须得登录A网站的 form输入界面才能提交成功。

如果B网站事先去A网站进行正常访问拿到csrf，再替换到B网站的表单也可以提交，但是C请求生成的csrf和B网站请求生成的csrf不一样。

用户的cookie很容易通过XSS获取，所以还是存在威胁

第二种方法：使用验证码

用户必须输入指定的随机字符串，才能访问，

我们先来看看CSRF和XSS的工作原理，先让大家把这两个分开来。

XSS：

攻击者发现XSS漏洞——构造代码——发送给受害人——受害人打开——攻击者获取受害人的cookie——完成攻击

CSRF：

攻击者发现CSRF漏洞——构造代码——发送给受害人——受害人打开——受害人执行代码——完成攻击

#### xss攻击

就是在用户浏览器植入js或html代码，xss的危害有：盗取各类用户帐号，控制企业数据：包括更改，删除等操作，盗窃企业重要资料，非法转账，强行发送电子邮件，网站挂马，控制受害者机器像其他网站发起攻击，

### Session和cookie

Session保存在服务器，cookie保存在客户端，是一段键值对，当用户请求过来后cookie给session提供信息，如果没有cookie说明是第一次，或者cookie已经过期。比如用户登录成功后，第二次请求后，判断用户已经登录成功，就不需要再次登录

Django设置cookie

obj=render(request,"cookie.html")  
obj.set\_cookie("k1","v1")  
return obj

当第二次请求时，头部自动带cookie

获取cookie

request.COOKIES

{'sessionid': 'hjl6tw8hqpuxhjvurkxe9h4rjwx0rps7', 'SULang': 'zh%2CCN', '\_ga': 'GA1.1.603777216.1467547540', 'domainid': '6755', 'csrftoken': 'fxCXwz4kGfC3xpy4dkhlPYgbt2YHCmanvyD4mxXg1xAd6DSwEyVcVr9UfuwAsWhp', 'domainname': 'live', 'k1': 'v1', 'domainodbc': '0', 'homelinktip': 'false', 'themename': 'vista'}

set\_cookie(self, key, value='', max\_age=None, expires=None, path='/',  
 domain=None, secure=False, httponly=False):

max\_age多少时间后失效，expires具体什么时间后失效

path指定url设置cookie，默认为/为全局生效

domain指定域名生效

用户认证用session

Django为session保存用户信息时，发送给用户的cookie为一个随机字符串，session在服务端内部可以保存很多值，随机字符串不能重复，django默认把session创建在数据库中

Session储存

request.session['user']=user

查询

user = request.session.get("user", None)

删除

del request.session['user']

django的session，如果给session赋值一个字典后，在字典里修改值，是不会生效的。只能修改session的值才会生效

#### session配置

session可放在数据库，缓存，文件，缓存加数据库，加密cookie放到客户端等

数据库

    SESSION\_ENGINE = 'django.contrib.sessions.backends.db'   # 引擎（默认）

    SESSION\_COOKIE\_NAME ＝ "sessionid"                       # Session的cookie保存在浏览器上时的key，即：sessionid＝随机字符串（默认）

    SESSION\_COOKIE\_PATH ＝ "/"                               # Session的cookie保存的路径（默认）

    SESSION\_COOKIE\_DOMAIN = None                             # Session的cookie保存的域名（默认）

    SESSION\_COOKIE\_SECURE = False                            # 是否Https传输cookie（默认）

    SESSION\_COOKIE\_HTTPONLY = True                           # 是否Session的cookie只支持http传输（默认）

    SESSION\_COOKIE\_AGE = 1209600                             # Session的cookie失效日期（2周）（默认）

    SESSION\_EXPIRE\_AT\_BROWSER\_CLOSE = False                  # 是否关闭浏览器使得Session过期（默认）

    SESSION\_SAVE\_EVERY\_REQUEST = False                       # 是否每次请求都保存Session，默认修改之后才保存（默认）

获取值：

request.session['k1']

request.session.get('k1',None)

修改值：

request.session['k1'] = 123

request.session.setdefault('k1',123) # 存在则不设置

删除值：

del request.session['k1']

获取所有的键值

        request.session.keys()

        request.session.values()

        request.session.items()

       #设置session有效时间

request.session.set\_expiry(time)

time可以是datetime, timedelta，也可以是直接某个值

datetime表示到具体某个时间

timedelta表示从当前开始到某个时间

某个值表示多少秒

如果设置成0表示浏览器关闭后失效

 # 将所有Session失效日期小于当前日期的数据删除

        request.session.clear\_expired()

文件

SESSION\_ENGINE = 'django.contrib.sessions.backends.file' # 引擎  
SESSION\_FILE\_PATH = None # 缓存文件路径，如果为None，则使用tempfile模块获取一个临时地址tempfile.gettempdir() # 如：/var/folders/d3/j9tj0gz93dg06bmwxmhh6\_xm0000gn/T  
  
SESSION\_COOKIE\_NAME = "sessionid" # Session的cookie保存在浏览器上时的key，即：sessionid＝随机字符串  
SESSION\_COOKIE\_PATH = "/" # Session的cookie保存的路径  
SESSION\_COOKIE\_DOMAIN = None # Session的cookie保存的域名  
SESSION\_COOKIE\_SECURE = False # 是否Https传输cookie  
SESSION\_COOKIE\_HTTPONLY = True # 是否Session的cookie只支持http传输  
SESSION\_COOKIE\_AGE = 1209600 # Session的cookie失效日期（2周）  
SESSION\_EXPIRE\_AT\_BROWSER\_CLOSE = False # 是否关闭浏览器使得Session过期  
SESSION\_SAVE\_EVERY\_REQUEST = False # 是否每次请求都保存Session，默认修改之后才保存

XSS攻击

### Django分页

#### django内置分页：

**from** django.core.paginator **import** Paginator, EmptyPage, PageNotAnInteger

导入这些：

Paginator(model\_obj,cord\_num) #传入数据对象和每页显示的条目，生成paginator对象

pageinator的属性：

count 数据总个数

per\_page 每页显示的条目

num\_pages 总页数

page\_range 页码生成器

page 具体页码对象

post=paginator.page(2) #具体第几页，这个对象

post属性：

has\_next() 是否有下一页

has\_prev() 是否有上一页

next\_page\_number() 下一页页码

prev\_page\_number() 上一页页码

number 当前页

PageNotInteger 如果传过来的页码不是整数，就报错

EmptyPage 如果传过来的页码为空，就报错

#### 自制分页

拿到一页的数据列表，和页码列表放到页面中。

需要传入当前页，所有数据(一张表)，一页显示多少记录，就能生成当前页的所有数据和页码列表。

大部分网站页码列表不是所有都列出来的，如果当前页是10会把前5个和后4个显示到页面上，然后有上一页和下一页，所以要定义一次显示页码的数量

**:param** *current\_page: 传入当前页码，如果没有默认1***:param** *rows: 一页的记录数***:param** *mod: 指定数据库***:param** *base\_url: 指定全路径***:param** *page\_size: 指定一次最多显示多少个页码*

先看目的：

*cp=ChPage(page\_current,rows,models.UserInfo,base\_url)*

*mod\_ret=models.UserInfo.objects.filter(id\_\_lt=100,id\_\_gt=10)  
model\_list=cp.get\_pm(mod\_ret) #当前页的数据列表，放到网页中*

*cp.page\_list # 生成的页码的列表，放到网页中*

### Django中间件

配置路径：

Setting.py->MIDDLEWARE

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',  
 'md.nd.nd1',  
]

自定义中间件例子

from django.utils.deprecation import MiddlewareMixin  
class nd1(MiddlewareMixin):  
 def process\_request(self,request):  
 print("nd1.process\_request")  
 def process\_response(self,request,response):  
 print("nd1.process\_response")  
 return response  
  
class nd2(MiddlewareMixin):  
 def process\_request(self,request):  
 print("nd2.process\_request")  
 def process\_response(self,request,response):  
 print("nd2.process\_response")  
 return response

到view之前执行process\_request，从上到下执行，

view执行完了之后执行process\_response，从下倒上执行

如果中间件中有process\_view，在process\_request执行完后执行view

def process\_view(self,request, callback, callback\_args, callback\_kwargs):  
 print("nd1.process\_view")

如果views请求出现异常，执行process\_exception(self, request, exception)

def process\_exception(self, request, exception):  
 print("nd1.process\_exception")

当process\_request有return时，不执行views，老版本把所有process\_response执行完再回去，新版本只执行一个

### Django缓存

#### 配置

1，调试缓存

CACHES = {  
 'default': {  
 'BACKEND': 'django.core.cache.backends.dummy.DummyCache', # 引擎  
 'TIMEOUT': 300, # 缓存超时时间（默认300，None表示永不过期，0表示立即过期）  
 'OPTIONS': {  
 'MAX\_ENTRIES': 300, # 最大缓存个数（默认300）  
 'CULL\_FREQUENCY': 3, # 缓存到达最大个数之后，剔除缓存个数的比例，即：1/CULL\_FREQUENCY（默认3）  
 },  
 'KEY\_PREFIX': '', # 缓存key的前缀（默认空）  
 'VERSION': 1, # 缓存key的版本（默认1）  
 'KEY\_FUNCTION' 函数名 # 生成key的函数（默认函数会生成为：【前缀:版本:key】）  
 }  
}

1. 文件级别缓存：

CACHES = {  
 'default': {  
 'BACKEND': 'django.core.cache.backends.filebased.FileBasedCache',  
 'LOCATION': os.path.join(BASE\_DIR,'cache'),  
 }  
}

1. 内存缓存

CACHES = {  
 'default': {  
 'BACKEND': 'django.core.cache.backends.locmem.LocMemCache',  
 'LOCATION': 'unique-snowflake', #key值  
 }  
}

1. 数据库缓存

CACHES = {  
 'default': {  
 'BACKEND': 'django.core.cache.backends.db.DatabaseCache',  
 'LOCATION': 'my\_cache\_table', # 数据库表  
 }  
}

具体某一个表上

1. memcache缓存—python\_memcached

CACHES = {  
 'default': {  
 'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',  
 'LOCATION': '10.0.0.20:11211',  
 }  
}

#### 应用

全局应用：中间件

MIDDLEWARE = [  
 'django.middleware.cache.UpdateCacheMiddleware',

...  
 'django.middleware.cache.FetchFromCacheMiddleware',  
]

单独视图缓存：

每个网站的views进行缓存

from django.views.decorators.cache import cache\_page  
@cache\_page(5)  
def cache1(request):

或urls.py内

from django.views.decorators.cache import cache\_page  
  
urlpatterns = [  
 url(r'^cache1/', cache\_page(60 \* 15)(views.cache1)),  
]

局部视图缓存：

对一个网页中的某个部分进行缓存

导入templatetag

{% load cache %}

缓存的部分

{% cache 5 c\_cache %}  
。。。  
{% endcache %}

### Django信号

信号在某些事件发生前或发生后触发关联函数，信号在django加载时加载

#### 内置信号

| Model signals | |
| --- | --- |
| pre\_init | 数据库类执行构造函数时 |
| post\_init |
| pre\_save | 数据库保存时 |
| Post\_save |
| Pre\_delete | 数据删除时 |
| Post\_delete |
| Pre\_migrate | 数据库执行migrate命令时 |
| Post\_migrate |
| Class\_prepared | 当程序加载时，检查已经注册的app中modal类 |
| Request/reponse signals | |
| Request\_started | 请求到来前 |
| Request\_finished | 请求结束后 |
| Got\_request\_exception | 请求错误后 |

### 图片验证码

原理：用pillow画一张，放到前端展示，这张画包含几个字母，这些字母在服务端的session处存放，客户端需要输入画中的这些字母才可进行帐号和密码的验证。为了防止简单人工智能的识别，要加一些干扰。

### Django文件上传

默认POST提交文件的get收到的是文件名

request.FILES.get("img")接收文件，变成对象

前端要想传文件，必须在form上定义enctype="multipart/form-data"

<form method="POST" action="/upload/" enctype="multipart/form-data">  
 <input type="text" name="user" />  
 <input type="file" name="img" />  
 <input type="submit" />  
</form>

img=request.FILES.get("img")

类型为：<class 'django.core.files.uploadedfile.InMemoryUploadedFile'>的对象，接收的是二进制

该类方法：

Chunks数据切片

上传文件到服务器储存：

f=open(os.path.join("static",img.name),"wb")  
for chunk in img.chunks():  
 f.write(chunk)  
f.close()

### rest\_framework

django给其他应用提供api的模块，收到一串url和一个动作，返回json格式的数据库内容

模块：django-rest-framework

pip install djangorestframework

客户端以get请求方式请求api地址经过http请求到达view，view根据serialize的，返回一串json字符串，

Serialize类，设置显示规则

# Serializers define the API representation.  
class UserSerializer(serializers.HyperlinkedModelSerializer):  
 class Meta:  
 model = User  
 fields = ('username', 'email', "phone")  
  
class GroupSerializer(serializers.HyperlinkedModelSerializer):  
 class Meta:  
 model = Group  
 fields = ('groupname', )

viewset类

# ViewSets define the view behavior.  
class UserViewSet(viewsets.ModelViewSet):  
 queryset = User.objects.all()  
 serializer\_class = UserSerializer  
  
class GroupViewSet(viewsets.ModelViewSet):  
 queryset = Group.objects.all()  
 serializer\_class = GroupSerializer

url配置

# Routers provide an easy way of automatically determining the URL conf.  
router = routers.DefaultRouter()  
router.register(r'users', UserViewSet)  
router.register(r'groups', GroupViewSet)

urlpatterns = [  
 url(r'^', include(router.urls)),  
 url(r'^api-auth/', include('rest\_framework.urls', namespace='rest\_framework'))  
]

### 惰性翻译ugettext\_lazy

无论何处你要使用一个unicode 字符串，都可以使用一个 ugettext\_lazy() 调用的结果

# Flask

### 路由

#### request

from flask import Flask  
app = Flask(\_\_name\_\_)  
  
@app.route('/')  
def hello\_world():  
 return 'Hello World!'  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 app.run()

request请求返回的结果，flask.request

request.url

request.path|request.full\_path

request.query\_string 解析前的get参数

request.cookie

request.headers

Cookie: aa=bb

User-Agent: PostmanRuntime/6.4.1

Connection: keep-alive

Postman-Token: 46334d2e-2ec0-49a3-a0ae-02e5b2f24d4c

Host: localhost:5000

Cache-Control: no-cache

Accept: \*/\*

Accept-Encoding: gzip, deflate

request.headers["User-Agent"]

内容:

url: request.args

body: request.form

变成能提取的字典

request.form.to\_dict()

#### response

jinja预处理

@app.route('/', methods=["GET", "POST", 'PUT'])  
def hello\_world():  
 name = "file"  
 return render\_template("index.html", name=name)

render\_template(file, \*\*context) -> string

render\_template\_string(string, \*\*context) -> string