

公告

wiki和教程: www.pythontav.com

免费教学视频: B站: [凸头统治地球](#)

高级专题教程: [网易云课堂: 武沛齐](#)

聊技术, 加武Sir微信



武沛齐



扫一扫上面的二维码图案, 加我微信



Python技术交流群: 737658057

软件测试开发交流群: 721023555

昵称: 武沛齐

园龄: 8年2个月

粉丝: 10098

关注: 44

+加关注

我的标签

Python(17)

ASP.NET MVC(15)

python之路(7)

Tornado源码分析(5)

每天一道Python面试题(5)

crm项目(4)

面试都在问什么? (2)

Python开源组件 - Tyrion(1)

Python面试315题(1)

Python企业面试题讲解(1)

积分与排名

积分 - 433382

排名 - 814

随笔分类

JavaScript(1)

MVC(15)

Python(17)

随笔 - 140 文章 - 164 评论 - 903

Python之路【第十七篇】： Django【进阶篇】

Model

到目前为止, 当我们的程序涉及到数据库相关操作时, 我们一般都会这么搞:

- 创建数据库, 设计表结构和字段
- 使用 MySQLdb 来连接数据库, 并编写数据访问层代码
- 业务逻辑层去调用数据访问层执行数据库操作

曰



```
import MySQLdb

def GetList(sql):
    db = MySQLdb.connect(user='root', db='wupeiqidb', passwd='1234', host='127.0.0.1')
    cursor = db.cursor()
    cursor.execute(sql)
    data = cursor.fetchall()
    db.close()
    return data

def GetSingle(sql):
    db = MySQLdb.connect(user='root', db='wupeiqidb', passwd='1234', host='127.0.0.1')
    cursor = db.cursor()
    cursor.execute(sql)
    data = cursor.fetchone()
    db.close()
    return data
```



django为使用一种新的方式, 即: 关系对象映射 (Object Relational Mapping, 简称 ORM) 。

PHP: activerecord

Java: Hibernate

C#: Entity Framework

django中遵循 Code Frist 的原则, 即: 根据代码中定义类来自动生成数据库表。

一、创建表

1、基本结构

```
1 from django.db import models
2
3 class userinfo(models.Model):
4     name = models.CharField(max_length=30)
5     email = models.EmailField()
6     memo = models.TextField()
```

曰



```
AutoField(Field)
- int自增列, 必须填入参数 primary_key=True
```

面试都在问什么系列？【图】(2)

其他(37)

企业面试题及答案(1)

请求响应(6)

设计模式(9)

微软C#(34)

随笔档案

- 2020年6月(1)
- 2020年5月(1)
- 2019年11月(1)
- 2019年10月(1)
- 2019年9月(4)
- 2018年12月(1)
- 2018年8月(1)
- 2018年5月(2)
- 2018年4月(1)
- 2017年8月(1)
- 2017年5月(1)
- 2017年3月(1)
- 2016年10月(1)
- 2016年7月(1)
- 2015年10月(1)
- 2015年8月(1)
- 2015年7月(1)
- 2015年6月(2)
- 2015年4月(2)
- 2014年3月(3)
- 2014年1月(3)
- 2013年12月(2)
- 2013年11月(2)
- 2013年10月(7)
- 2013年8月(17)
- 2013年7月(1)
- 2013年6月(14)
- 2013年5月(23)
- 2013年4月(3)
- 2013年3月(13)
- 2013年2月(1)
- 2012年11月(26)

相册

git(14)

最新评论

1. Re:CRM【第一篇】：权限组件之权限控制

为什么不用moduleformset? 会不会更简单

--刘岩61
2. Re:CRM【第一篇】：权限组件之权限控制

@两个小黄鹌 我看着也是觉得这里错了，应该是pre_namespace...

--刘岩61

```
BigAutoField(AutoField)
- bigint自增列，必须填入参数 primary_key=True

注：当model中如果没有自增列，则会自动创建一个列名为id的列
from django.db import models

class UserInfo(models.Model):
    # 自动创建一个列名为id的且为自增的整数列
    username = models.CharField(max_length=32)

class Group(models.Model):
    # 自定义自增列
    nid = models.AutoField(primary_key=True)
    name = models.CharField(max_length=32)

SmallIntegerField(IntegerField):
- 小整数 -32768 ~ 32767

PositiveSmallIntegerField(PositiveIntegerRelDbTypeMixin, IntegerField)
- 正小整数 0 ~ 32767

IntegerField(Field)
- 整数列(有符号的) -2147483648 ~ 2147483647

PositiveIntegerField(PositiveIntegerRelDbTypeMixin, IntegerField)
- 正整数 0 ~ 2147483647

BigIntegerField(IntegerField):
- 长整型(有符号的) -9223372036854775808 ~ 9223372036854775807

自定义无符号整数字段

class UnsignedIntegerField(models.IntegerField):
    def db_type(self, connection):
        return 'integer UNSIGNED'

PS：返回值为字段在数据库中的属性，Django字段默认的值为：
'AutoField': 'integer AUTO_INCREMENT',
'BigAutoField': 'bigint AUTO_INCREMENT',
'BinaryField': 'longblob',
'BooleanField': 'bool',
'CharField': 'varchar(%(max_length)s)',
'CommaSeparatedIntegerField': 'varchar(%(max_length)s)',
'DateField': 'date',
'DatetimeField': 'datetime',
'DecimalField': 'numeric(%(max_digits)s, %(decimal_places)s)',
'DurationField': 'bigint',
'FileField': 'varchar(%(max_length)s)',
'FilePathField': 'varchar(%(max_length)s)',
'FloatField': 'double precision',
'IntegerField': 'integer',
'BigIntegerField': 'bigint',
'IPAddressField': 'char(15)',
'GenericIPAddressField': 'char(39)',
'NullBooleanField': 'bool',
'OneToOneField': 'integer',
'PositiveIntegerField': 'integer UNSIGNED',
'PositiveSmallIntegerField': 'smallint UNSIGNED',
'SlugField': 'varchar(%(max_length)s)',
'SmallIntegerField': 'smallint',
'TextField': 'longtext',
'TimeField': 'time',
'UUIDField': 'char(32)',

BooleanField(Field)
- 布尔值类型
```

3. Re:jwt揭秘（含源码示例和视频）
武老师，想看一个自定义令牌刷新机制

--HHMLXL

4. Re:Python开发【第十七篇】： MySQL
（一）
好文 很详细

--风必摧之

5. Re:CRM【第一篇】： 权限组件之权限控制
为什么用户列表相关路由 放到了项目路由
下面 不是应该也放到rbac里面么

--CrazyDemo

```
NullBooleanField(Field):
    - 可以为空的布尔值

CharField(Field)
    - 字符类型
    - 必须提供max_length参数， max_length表示字符长度

TextField(Field)
    - 文本类型

EmailField(CharField):
    - 字符串类型， Django Admin以及ModelForm中提供验证机制

IPAddressField(Field)
    - 字符串类型， Django Admin以及ModelForm中提供验证 IPV4 机制

GenericIPAddressField(Field)
    - 字符串类型， Django Admin以及ModelForm中提供验证 Ipv4和Ipv6
    - 参数:
        protocol, 用于指定Ipv4或Ipv6, 'both', "ipv4", "ipv6"
        unpack_ipv4, 如果指定为True, 则输入::ffff:192.0.2.1时候, 可解析为192.0.2.1

URLField(CharField)
    - 字符串类型， Django Admin以及ModelForm中提供验证 URL

SlugField(CharField)
    - 字符串类型， Django Admin以及ModelForm中提供验证支持 字母、数字、下划线、连接符

CommaSeparatedIntegerField(CharField)
    - 字符串类型， 格式必须为逗号分割的数字

UUIDField(Field)
    - 字符串类型， Django Admin以及ModelForm中提供对UUID格式的验证

FilePathField(Field)
    - 字符串， Django Admin以及ModelForm中提供读取文件夹下文件的功能
    - 参数:
        path, 文件夹路径
        match=None, 正则匹配
        recursive=False, 递归下面的文件夹
        allow_files=True, 允许文件
        allow_folders=False, 允许文件夹

FileField(Field)
    - 字符串， 路径保存在数据库， 文件上传到指定目录
    - 参数:
        upload_to = "" 上传文件的保存路径
        storage = None 存储组件， 默认django.core.files.storage.FileSystemStorage

ImageField(FileField)
    - 字符串， 路径保存在数据库， 文件上传到指定目录
    - 参数:
        upload_to = "" 上传文件的保存路径
        storage = None 存储组件， 默认django.core.files.storage.FileSystemStorage
        width_field=None, 上传图片的高度保存的数据库字段名（字符串）
        height_field=None 上传图片的宽度保存的数据库字段名（字符串）

DateTimeField(DateField)
    - 日期+时间格式 YYYY-MM-DD HH:MM[:ss[.uuuuuu]][TZ]

DateField(DateTimeCheckMixin, Field)
    - 日期格式 YYYY-MM-DD

TimeField(DateTimeCheckMixin, Field)
    - 时间格式 HH:MM[:ss[.uuuuuu]]
```

```
DurationField(Field)
    - 长整数，时间间隔，数据库中按照bigint存储，ORM中获取的值为datetime.timedelta

FloatField(Field)
    - 浮点型

DecimalField(Field)
    - 10进制小数
    - 参数：
        max_digits, 小数总长度
        decimal_places, 小数位长度

BinaryField(Field)
    - 二进制类型
```

```

null                数据库中字段是否可以空
db_column           数据库中字段的列名
db_tablespace
default             数据库中字段的默认值
primary_key         数据库中字段是否为主键
db_index            数据库中字段是否可以建立索引
unique              数据库中字段是否可以建立唯一索引
unique_for_date     数据库中字段【日期】部分是否可以建立唯一索引
unique_for_month    数据库中字段【月】部分是否可以建立唯一索引
unique_for_year     数据库中字段【年】部分是否可以建立唯一索引

verbose_name        Admin中显示的字段名称
blank               Admin中是否允许用户输入为空
editable            Admin中是否可以编辑
help_text           Admin中该字段的提示信息
choices             Admin中显示选择框的内容，用不变动的数据放在内存中从而避免跨表查询
                    如: gf = models.IntegerField(choices=[(0, '何穗'), (1,

error_messages       自定义错误信息（字典类型），从而定制想要显示的错误信息；
                    字典键: null, blank, invalid, invalid_choice, unique,
                    如: {'null': "不能为空.", 'invalid': '格式错误'}

validators           自定义错误验证（列表类型），从而定制想要的验证规则
                    from django.core.validators import RegexValidator
                    from django.core.validators import EmailValidator,URLValidator,
                    MaxLengthValidator,MinLengthValidator,MaxValueValidator,MinValueValidator
                    如:
                    test = models.CharField(
                        max_length=32,
                        error_messages={
                            'c1': '优先错信息1',
                            'c2': '优先错信息2',
                            'c3': '优先错信息3',
                        },
                        validators=[
                            RegexValidator(regex='root_\d+', message=
                            RegexValidator(regex='root_112233\d+', m
                            EmailValidator(message='又错误了', code='c
```

```
class UserInfo(models.Model):
    nid = models.AutoField(primary_key=True)
```

```

username = models.CharField(max_length=32)

class Meta:
    # 数据库中生成的表名称 默认 app名称 + 下划线 + 类名
    db_table = "table_name"

    # 联合索引
    index_together = [
        ("pub_date", "deadline"),
    ]

    # 联合唯一索引
    unique_together = (("driver", "restaurant"),)

    # admin中显示的表名称
    verbose_name

    # verbose_name加s
    verbose_name_plural

```

更多: <https://docs.djangoproject.com/en/1.10/ref/models/options/>



1. 触发Model中的验证和错误提示有两种方式:

- Django Admin中的错误信息会优先根据Admin内部的ModelForm错误信息提示, 如果
- 调用Model对象的 `clean_fields` 方法, 如:

```

# models.py
class UserInfo(models.Model):
    nid = models.AutoField(primary_key=True)
    username = models.CharField(max_length=32)

    email = models.EmailField(error_messages={'invalid': '格式错了'})

# views.py
def index(request):
    obj = models.UserInfo(username='11234', email='uu')
    try:
        print(obj.clean_fields())
    except Exception as e:
        print(e)
    return HttpResponse('ok')

# Model的clean方法是一个钩子, 可用于定制操作, 如: 上述的异常处理。

```

2. Admin中修改错误提示

```

# admin.py
from django.contrib import admin
from model_club import models
from django import forms

class UserInfoForm(forms.ModelForm):
    username = forms.CharField(error_messages={'required': '用户名不能'})
    email = forms.EmailField(error_messages={'invalid': '邮箱格式错误'})
    age = forms.IntegerField(initial=1, error_messages={'required':

class Meta:
    model = models.UserInfo
    # fields = ('username',)
    fields = "__all__"

class UserInfoAdmin(admin.ModelAdmin):
    form = UserInfoForm

```

```
admin.site.register(models.UserInfo, UserInfoAdmin)
```



2、连表结构

- 一对多：models.ForeignKey(其他表)
- 多对多：models.ManyToManyField(其他表)
- 一对一：models.OneToOneField(其他表)

应用场景：

- 一对多：当一张表中创建一行数据时，有一个单选的下拉框（可以被重复选择）
例如：创建用户信息时候，需要选择一个用户类型【普通用户】【金牌用户】【铂金用户】等。
- 多对多：在某表中创建一行数据是，有一个可以多选的下拉框
例如：创建用户信息，需要为用户指定多个爱好
- 一对一：在某表中创建一行数据时，有一个单选的下拉框（下拉框中的内容被用过一次就消失了）
例如：原有含10列数据的一张表保存相关信息，经过一段时间之后，10列无法满足需求，需要为原来的表再添加5列数据



```
ForeignKey(ForeignObject) # ForeignKey(RelatedField)
to, # 要进行关联的表名
to_field=None, # 要关联的表中的字段名称
on_delete=None, # 当删除关联表中的数据时，当前表与其关联的行的行
                - models.CASCADE, 删除关联数据，与之关联
                - models.DO_NOTHING, 删除关联数据，引发错误
                - models.PROTECT, 删除关联数据，引发错误
                - models.SET_NULL, 删除关联数据，与之关联
                - models.SET_DEFAULT, 删除关联数据，与之
                - models.SET, 删除关联数据，
                  a. 与之关联的值设置为指定值
                  b. 与之关联的值设置为可执行

def func():
    return 10

class MyModel(models
    user = models.Fo
        to="User",
        to_field="id
        on_delete=mo

related_name=None, # 反向操作时，使用的字段名，用于代替 【表名_set
related_query_name=None, # 反向操作时，使用的连接前缀，用于替换【表名】
limit_choices_to=None, # 在Admin或ModelForm中显示关联数据时，提供的条
# 如：
    - limit_choices_to={'nid__gt': 5
    - limit_choices_to=lambda : {'ni

from django.db.models import Q
    - limit_choices_to=Q(nid__gt=10)
    - limit_choices_to=Q(nid=8) | Q(
    - limit_choices_to=lambda : Q(Q(

db_constraint=True # 是否在数据库中创建外键约束
parent_link=False # 在Admin中是否显示关联数据
```

```
OneToOneField(ForeignKey)
```

```

to,
to_field=None
on_delete=None,

##### 对于一对一 #####
# 1. 一对一其实就是 一对多 + 唯一索引
# 2. 当两个类之间有继承关系时，默认会创建一个一对一
# 如下会在A表中额外增加一个c_ptr_id列且唯一：

class C(models.Model):
    nid = models.AutoField(primary_key=True)
    part = models.CharField(max_length=100)

class A(C):
    id = models.AutoField(primary_key=True)
    code = models.CharField(max_length=100)

ManyToManyField(RelatedField)
to,
related_name=None,
related_query_name=None,
limit_choices_to=None,

symmetrical=None,

through=None,
through_fields=None,

# 要进行关联的表名
# 要关联的表中的字段名称
# 当删除关联表中的数据时，当前表与其关联的行的行

##### 对于一对一 #####
# 1. 一对一其实就是 一对多 + 唯一索引
# 2. 当两个类之间有继承关系时，默认会创建一个一对一
# 如下会在A表中额外增加一个c_ptr_id列且唯一：

class C(models.Model):
    nid = models.AutoField(primary_key=True)
    part = models.CharField(max_length=100)

class A(C):
    id = models.AutoField(primary_key=True)
    code = models.CharField(max_length=100)

# 要进行关联的表名
# 反向操作时，使用的字段名，用于代替 【表名_set】
# 反向操作时，使用的连接前缀，用于替换【表名】
# 在Admin或ModelForm中显示关联数据时，提供的名称
# 如：
- limit_choices_to={'nid__gt': 5}
- limit_choices_to=lambda : {'nid__gt': 5}

from django.db.models import Q
- limit_choices_to=Q(nid__gt=10)
- limit_choices_to=Q(nid=8) | Q(nid__gt=10)
- limit_choices_to=lambda : Q(Q(nid__gt=10) | Q(nid=8))

# 仅用于多对多自关联时，symmetrical用于指定内联
# 做如下操作时，不同的symmetrical会有不同的可
models.BB.objects.filter(...)

# 可选字段有: code, id, m1
class BB(models.Model):
    code = models.CharField(max_length=100)
    m1 = models.ManyToManyField('sel

# 可选字段有: bb, code, id, m1
class BB(models.Model):
    code = models.CharField(max_length=100)
    m1 = models.ManyToManyField('sel

# 自定义第三张表时，使用字段用于指定关系表
# 自定义第三张表时，使用字段用于指定关系表中那些
from django.db import models

class Person(models.Model):
    name = models.CharField(max_length=100)

class Group(models.Model):
    name = models.CharField(max_length=100)
    members = models.ManyToManyField(
        Person,
        through='Membership',
        through_fields=('group', 'pe

)

class Membership(models.Model):
    group = models.ForeignKey(Group,
    person = models.ForeignKey(Perso
    inviter = models.ForeignKey(
        Person,
    on_delete=models.CASCADE,

```

```

        related_name="membership_inv
    )

    invite_reason = models.CharField(
        db_constraint=True,          # 是否在数据库中创建外键约束
        db_table=None,              # 默认创建第三张表时，数据库表中表的名称
    )

```



二、操作表

1、基本操作



```

# 增
#
# models.Tb1.objects.create(c1='xx', c2='oo')  增加一条数据，可以接受字典类型

# obj = models.Tb1(c1='xx', c2='oo')
# obj.save()

# 查
#
# models.Tb1.objects.get(id=123)          # 获取单条数据，不存在则报错（不建议）
# models.Tb1.objects.all()                # 获取全部
# models.Tb1.objects.filter(name='seven') # 获取指定条件的数据

# 删
#
# models.Tb1.objects.filter(name='seven').delete() # 删除指定条件的数据

# 改
# models.Tb1.objects.filter(name='seven').update(gender='0') # 将指定条件
# obj = models.Tb1.objects.get(id=1)
# obj.c1 = '111'
# obj.save()                                     # 修改单条数据

```



2、进阶操作（了不起的双下划线）

利用双下划线将字段和对应的操作连接起来



```

# 获取个数
#
# models.Tb1.objects.filter(name='seven').count()

# 大于, 小于
#
# models.Tb1.objects.filter(id__gt=1)          # 获取id大于1的值
# models.Tb1.objects.filter(id__gte=1)         # 获取id大于等于1的值
# models.Tb1.objects.filter(id__lt=10)         # 获取id小于10的值
# models.Tb1.objects.filter(id__lte=10)        # 获取id小于等于10的值
# models.Tb1.objects.filter(id__lt=10, id__gt=1) # 获取id大于1 且 小于10

# in
#
# models.Tb1.objects.filter(id__in=[11, 22, 33]) # 获取id等于11、22、33
# models.Tb1.objects.exclude(id__in=[11, 22, 33]) # not in

# isnull
# Entry.objects.filter(pub_date__isnull=True)

```



```
# contains
#
# models.Tb1.objects.filter(name__contains="ven")
# models.Tb1.objects.filter(name__icontains="ven") # icontains大小写不
# models.Tb1.objects.exclude(name__icontains="ven")

# range
#
# models.Tb1.objects.filter(id__range=[1, 2]) # 范围between and

# 其他类似
#
# startswith, istartswith, endswith, iendswith,

# order by
#
# models.Tb1.objects.filter(name='seven').order_by('id') # asc
# models.Tb1.objects.filter(name='seven').order_by('-id') # desc

# group by
#
# from django.db.models import Count, Min, Max, Sum
# models.Tb1.objects.filter(c1=1).values('id').annotate(c=Count('num'))
# SELECT "app01_tb1"."id", COUNT("app01_tb1"."num") AS "c" FROM "app

# limit 、 offset
#
# models.Tb1.objects.all()[10:20]

# regex正则匹配, iregex 不区分大小写
#
# Entry.objects.get(title__regex=r'^(An?|The) +')
# Entry.objects.get(title__iregex=r'^(an?|the) +')

# date
#
# Entry.objects.filter(pub_date__date=datetime.date(2005, 1, 1))
# Entry.objects.filter(pub_date__date__gt=datetime.date(2005, 1, 1))

# year
#
# Entry.objects.filter(pub_date__year=2005)
# Entry.objects.filter(pub_date__year__gte=2005)

# month
#
# Entry.objects.filter(pub_date__month=12)
# Entry.objects.filter(pub_date__month__gte=6)

# day
#
# Entry.objects.filter(pub_date__day=3)
# Entry.objects.filter(pub_date__day__gte=3)

# week_day
#
# Entry.objects.filter(pub_date__week_day=2)
# Entry.objects.filter(pub_date__week_day__gte=2)

# hour
#
# Event.objects.filter(timestamp__hour=23)
# Event.objects.filter(time__hour=5)
# Event.objects.filter(timestamp__hour__gte=12)

# minute
```

```
#
# Event.objects.filter(timestamp__minute=29)
# Event.objects.filter(time__minute=46)
# Event.objects.filter(timestamp__minute__gte=29)

# second
#
# Event.objects.filter(timestamp__second=31)
# Event.objects.filter(time__second=2)
# Event.objects.filter(timestamp__second__gte=31)
```



3、其他操作



```
# extra
#
# extra(self, select=None, where=None, params=None, tables=None, order_b
# Entry.objects.extra(select={'new_id': "select col from sometable wh
# Entry.objects.extra(where=['headline=%s'], params=['Lennon'])
# Entry.objects.extra(where=["foo='a' OR bar = 'a'", "baz = 'a'"])
# Entry.objects.extra(select={'new_id': "select id from tb where id >

# F
#
# from django.db.models import F
# models.Tbl.objects.update(num=F('num')+1)

# Q
#
# 方式一:
# Q(nid__gt=10)
# Q(nid=8) | Q(nid__gt=10)
# Q(Q(nid=8) | Q(nid__gt=10)) & Q(caption='root')
# 方式二:
# con = Q()
# q1 = Q()
# q1.connector = 'OR'
# q1.children.append(('id', 1))
# q1.children.append(('id', 10))
# q1.children.append(('id', 9))
# q2 = Q()
# q2.connector = 'OR'
# q2.children.append(('c1', 1))
# q2.children.append(('c1', 10))
# q2.children.append(('c1', 9))
# con.add(q1, 'AND')
# con.add(q2, 'AND')
#
# models.Tbl.objects.filter(con)

# 执行原生SQL
#
# from django.db import connection, connections
# cursor = connection.cursor() # cursor = connections['default'].cursor
# cursor.execute("""SELECT * from auth_user where id = %s""", [1])
# row = cursor.fetchone()
```



4、连表操作（了不起的双下划线）

利用双下划线和 `_set` 将表之间的操作连接起来

```
class UserProfile(models.Model):
    user_info = models.OneToOneField('UserInfo')
    username = models.CharField(max_length=64)
    password = models.CharField(max_length=64)

    def __unicode__(self):
        return self.username

class UserInfo(models.Model):
    user_type_choice = (
        (0, u'普通用户'),
        (1, u'高级用户'),
    )
    user_type = models.IntegerField(choices=user_type_choice)
    name = models.CharField(max_length=32)
    email = models.CharField(max_length=32)
    address = models.CharField(max_length=128)

    def __unicode__(self):
        return self.name

class UserGroup(models.Model):

    caption = models.CharField(max_length=64)

    user_info = models.ManyToManyField('UserInfo')

    def __unicode__(self):
        return self.caption

class Host(models.Model):
    hostname = models.CharField(max_length=64)
    ip = models.GenericIPAddressField()
    user_group = models.ForeignKey('UserGroup')

    def __unicode__(self):
        return self.hostname
```

```
user_info_obj = models.UserInfo.objects.filter(id=1).first()
print user_info_obj.user_type
print user_info_obj.get_user_type_display()
print user_info_obj.userprofile.password

user_info_obj = models.UserInfo.objects.filter(id=1).values('email', 'userpr
print user_info_obj.keys()
print user_info_obj.values()
```

类似一对一

- 1、搜索条件使用 `__` 连接
- 2、获取值时使用 `.` 连接

```

user_info_obj = models.UserInfo.objects.get(name=u'武沛齐')
user_info_objs = models.UserInfo.objects.all()

group_obj = models.UserGroup.objects.get(caption='CEO')
group_objs = models.UserGroup.objects.all()

# 添加数据
#group_obj.user_info.add(user_info_obj)
#group_obj.user_info.add(*user_info_objs)

# 删除数据
#group_obj.user_info.remove(user_info_obj)
#group_obj.user_info.remove(*user_info_objs)

# 添加数据
#user_info_obj.usergroup_set.add(group_obj)
#user_info_obj.usergroup_set.add(*group_objs)

# 删除数据
#user_info_obj.usergroup_set.remove(group_obj)
#user_info_obj.usergroup_set.remove(*group_objs)

# 获取数据
#print group_obj.user_info.all()
#print group_obj.user_info.all().filter(id=1)

# 获取数据
#print user_info_obj.usergroup_set.all()
#print user_info_obj.usergroup_set.all().filter(caption='CEO')
#print user_info_obj.usergroup_set.all().filter(caption='DBA')

```

扩展:

a、自定义上传

```

def upload_file(request):
    if request.method == "POST":
        obj = request.FILES.get('fafafa')
        f = open(obj.name, 'wb')
        for chunk in obj.chunks():
            f.write(chunk)
        f.close()
    return render(request, 'file.html')

```

b、Form上传文件实例

```

class FileForm(forms.Form):
    ExcelFile = forms.FileField()

```

```

from django.db import models

class UploadFile(models.Model):
    userid = models.CharField(max_length = 30)
    file = models.FileField(upload_to = './upload/')
    date = models.DateTimeField(auto_now_add=True)

```

```
def UploadFile(request):
    uf = AssetForm.FileForm(request.POST, request.FILES)
    if uf.is_valid():
        upload = models.UploadFile()
        upload.userid = 1
        upload.file = uf.cleaned_data['ExcelFile']
        upload.save()

    print upload.file
```

Form

django中的Form一般有两种功能:

- 输入html
- 验证用户输入

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
import re
from django import forms
from django.core.exceptions import ValidationError

def mobile_validate(value):
    mobile_re = re.compile(r'^(13[0-9]|15[012356789]|17[678]|18[0-9]|14[57])'
    if not mobile_re.match(value):
        raise ValidationError('手机号码格式错误')

class PublishForm(forms.Form):

    user_type_choice = (
        (0, u'普通用户'),
        (1, u'高级用户'),
    )

    user_type = forms.IntegerField(widget=forms.widgets.Select(choices=user_
        attrs={'cl

    title = forms.CharField(max_length=20,
        min_length=5,
        error_messages={'required': u'标题不能为空',
            'min_length': u'标题最少为5个字符',
            'max_length': u'标题最多为20个字符',
        widget=forms.TextInput(attrs={'class': "form-con
            'placeholder': u'标

    memo = forms.CharField(required=False,
        max_length=256,
        widget=forms.widgets.Textarea(attrs={'class': "fo

    phone = forms.CharField(validators=[mobile_validate, ],
        error_messages={'required': u'手机不能为空'},
        widget=forms.TextInput(attrs={'class': "form-con
            'placeholder': u'手

    email = forms.EmailField(required=False,
```

```
error_messages={'required': u'邮箱不能为空', 'invalid': u'无效的邮箱地址'}
widget=forms.TextInput(attrs={'class': "form-control"})
```



```
def publish(request):
    ret = {'status': False, 'data': '', 'error': '', 'summary': ''}
    if request.method == 'POST':
        request_form = PublishForm(request.POST)
        if request_form.is_valid():
            request_dict = request_form.clean()
            print request_dict
            ret['status'] = True
        else:
            error_msg = request_form.errors.as_json()
            ret['error'] = json.loads(error_msg)
    return HttpResponse(json.dumps(ret))
```



扩展：ModelForm

在使用Model和Form时，都需要对字段进行定义并指定类型，通过ModelForm则可以省去Form中字段的定义



```
class AdminModelForm(forms.ModelForm):

    class Meta:
        model = models.Admin
        #fields = '__all__'
        fields = ('username', 'email')

        widgets = {
            'email' : forms.PasswordInput(attrs={'class':"alex"}),
        }
```



跨站请求伪造

一、简介

django为用户实现防止跨站请求伪造的功能，通过中间件 django.middleware.csrf.CsrfViewMiddleware 来完成。而对于django中设置防跨站请求伪造功能有分为全局和局部。

全局：

中间件 django.middleware.csrf.CsrfViewMiddleware

局部：

- @csrf_protect, 为当前函数强制设置防跨站请求伪造功能，即便settings中没有设置全局中间件。
- @csrf_exempt, 取消当前函数防跨站请求伪造功能，即便settings中设置了全局中间件。

注：from django.views.decorators.csrf import csrf_exempt, csrf_protect

二、应用

1、普通表单

```

1  veiw中设置返回值:
2      return render_to_response('Account/Login.html',data,context_instance=Re
3      或者
4      return render(request, 'xxx.html', data)
5
6  html中设置Token:
7      {% csrf_token %}

```

2、Ajax

对于传统的form，可以通过表单的方式将token再次发送到服务端，而对于ajax的话，使用如下方式。

view.py

```

1  from django.template.context import RequestContext
2  # Create your views here.
3
4
5  def test(request):
6
7      if request.method == 'POST':
8          print request.POST
9          return HttpResponse('ok')
10     return render_to_response('app01/test.html',context_instance=RequestCo

```

text.html

```

1  <!DOCTYPE html>
2  <html>
3  <head lang="en">
4      <meta charset="UTF-8">
5      <title></title>
6  </head>
7  <body>
8      {% csrf_token %}
9
10     <input type="button" onclick="Do();" value="Do it"/>
11
12     <script src="/static/plugin/jquery/jquery-1.8.0.js"></script>
13     <script src="/static/plugin/jquery/jquery.cookie.js"></script>
14     <script type="text/javascript">
15         var csrftoken = $.cookie('csrftoken');
16
17         function csrfSafeMethod(method) {
18             // these HTTP methods do not require CSRF protection
19             return /^(GET|HEAD|OPTIONS|TRACE)$/.test(method);
20         }
21         $.ajaxSetup({
22             beforeSend: function(xhr, settings) {
23                 if (!csrfSafeMethod(settings.type) && !this.crossDomain) {
24                     xhr.setRequestHeader("X-CSRFToken", csrftoken);
25                 }
26             }
27         });
28         function Do(){
29
30             $.ajax({
31                 url:"/app01/test/",
32                 data:{id:1},
33                 type:'POST',
34                 success:function(data){
35                     console.log(data);
36                 }
37             });
38

```

```

39         }
40     </script>
41 </body>
42 </html>

```

更多: <https://docs.djangoproject.com/en/dev/ref/csrf/#ajax>

Cookie

1、获取Cookie:

```

1 request.COOKIES['key']
2 request.get_signed_cookie(key, default=RAISE_ERROR, salt='', max_age=None)
3 参数:
4     default: 默认值
5     salt: 加密盐
6     max_age: 后台控制过期时间

```

2、设置Cookie:

```

1 rep = HttpResponse(...) 或 rep = render(request, ...)
2
3 rep.set_cookie(key,value,...)
4 rep.set_signed_cookie(key,value,salt='加密盐',...)
5 参数:
6     key,          键
7     value='',     值
8     max_age=None, 超时时间
9     expires=None, 超时时间(IE requires expires, so set it if hasn't
10    path='/',      Cookie生效的路径, / 表示根路径, 特殊的: 跟路径的cook
11    domain=None,   Cookie生效的域名
12    secure=False,  https传输
13    httponly=False  只能http协议传输, 无法被JavaScript获取 (不是绝对, 底

```

由于cookie保存在客户端的电脑上, 所以, JavaScript和jquery也可以操作cookie。

```

1 <script src='/static/js/jquery.cookie.js'></script>
2 $.cookie("list_pager_num", 30,{ path: '/' });

```

Session

Django中默认支持Session, 其内部提供了5种类型的Session供开发者使用:

- 数据库 (默认)
- 缓存
- 文件
- 缓存+数据库
- 加密cookie

1、数据库Session

```

1 Django默认支持Session, 并且默认是将Session数据存储在数据库中, 即: django_sessio
2
3 a. 配置 settings.py
4
5     SESSION_ENGINE = 'django.contrib.sessions.backends.db' # 引擎 (默认)
6
7     SESSION_COOKIE_NAME = "sessionid" # Session的coo
8     SESSION_COOKIE_PATH = "/" # Session的coo
9     SESSION_COOKIE_DOMAIN = None # Session的coo
10    SESSION_COOKIE_SECURE = False # 是否Https传输
11    SESSION_COOKIE_HTTPONLY = True # 是否Session部
12    SESSION_COOKIE_AGE = 1209600 # Session的coo
13    SESSION_EXPIRE_AT_BROWSER_CLOSE = False # 是否关闭浏览器
14    SESSION_SAVE_EVERY_REQUEST = False # 是否每次请求者
15

```



```

16
17
18 b. 使用
19
20 def index(request):
21     # 获取、设置、删除Session中数据
22     request.session['k1']
23     request.session.get('k1',None)
24     request.session['k1'] = 123
25     request.session.setdefault('k1',123) # 存在则不设置
26     del request.session['k1']
27
28     # 所有 键、值、键值对
29     request.session.keys()
30     request.session.values()
31     request.session.items()
32     request.session.iterkeys()
33     request.session.itervalues()
34     request.session.iteritems()
35
36
37     # 用户session的随机字符串
38     request.session.session_key
39
40     # 将所有Session失效日期小于当前日期的数据删除
41     request.session.clear_expired()
42
43     # 检查 用户session的随机字符串 在数据库中是否
44     request.session.exists("session_key")
45
46     # 删除当前用户的所有Session数据
47     request.session.delete("session_key")
48
49     request.session.set_expiry(value)
50         * 如果value是个整数，session会在些秒数后失效。
51         * 如果value是个datetime或timedelta，session就会在这个时间后失效。
52         * 如果value是0,用户关闭浏览器session就会失效。
53         * 如果value是None,session会依赖全局session失效策略。

```

2、缓存Session

```

1 a. 配置 settings.py
2
3     SESSION_ENGINE = 'django.contrib.sessions.backends.cache' # 引擎
4     SESSION_CACHE_ALIAS = 'default' # 使用的缓存类
5
6
7     SESSION_COOKIE_NAME = "sessionid" # Session的cookie的名称
8     SESSION_COOKIE_PATH = "/" # Session的cookie保存的路径
9     SESSION_COOKIE_DOMAIN = None # Session的cookie域
10    SESSION_COOKIE_SECURE = False # 是否Https传输cookie
11    SESSION_COOKIE_HTTPONLY = True # 是否Session cookie只被浏览器访问
12    SESSION_COOKIE_AGE = 1209600 # Session的cookie过期时间：秒
13    SESSION_EXPIRE_AT_BROWSER_CLOSE = False # 是否关闭浏览器时Session过期
14    SESSION_SAVE_EVERY_REQUEST = False # 是否每次请求都保存Session
15
16
17
18 b. 使用
19
20 同上

```

3、文件Session

```

1 a. 配置 settings.py

```

```

2
3     SESSION_ENGINE = 'django.contrib.sessions.backends.file' # 引擎
4     SESSION_FILE_PATH = None # 缓存文件路
5
6
7     SESSION_COOKIE_NAME = "sessionid" # Session的
8     SESSION_COOKIE_PATH = "/" # Session的
9     SESSION_COOKIE_DOMAIN = None # Session的
10    SESSION_COOKIE_SECURE = False # 是否Https
11    SESSION_COOKIE_HTTPONLY = True # 是否Sessi
12    SESSION_COOKIE_AGE = 1209600 # Session的
13    SESSION_EXPIRE_AT_BROWSER_CLOSE = False # 是否关闭浏
14    SESSION_SAVE_EVERY_REQUEST = False # 是否每次请
15
16    b. 使用
17
18    同上

```

4、缓存+数据库Session

```

1    数据库用于做持久化，缓存用于提高效率
2
3    a. 配置 settings.py
4
5        SESSION_ENGINE = 'django.contrib.sessions.backends.cached_db' #
6
7    b. 使用
8
9    同上

```

5、加密cookie Session

```

1    a. 配置 settings.py
2
3        SESSION_ENGINE = 'django.contrib.sessions.backends.signed_cookies' #
4
5    b. 使用
6
7    同上

```

更多参考：[猛击这里](#) 和 [猛击这里](#)

扩展：Session用户验证

```

1    def login(func):
2        def wrap(request, *args, **kwargs):
3            # 如果未登陆，跳转到指定页面
4            if request.path == '/test/':
5                return redirect('http://www.baidu.com')
6            return func(request, *args, **kwargs)
7        return wrap

```

分页

一、Django内置分页

```

from django.shortcuts import render
from django.core.paginator import Paginator, EmptyPage, PageNotAnInteger

L = []
for i in range(999):
    L.append(i)

def index(request):

```

```

current_page = request.GET.get('p')

paginator = Paginator(L, 10)
# per_page: 每页显示条目数量
# count: 数据总个数
# num_pages:总页数
# page_range:总页数的索引范围, 如: (1,10), (1,200)
# page: page对象
try:
    posts = paginator.page(current_page)
    # has_next 是否有下一页
    # next_page_number 下一页页码
    # has_previous 是否有上一页
    # previous_page_number 上一页页码
    # object_list 分页之后的数据列表
    # number 当前页
    # paginator paginator对象
except PageNotAnInteger:
    posts = paginator.page(1)
except EmptyPage:
    posts = paginator.page(paginator.num_pages)
return render(request, 'index.html', {'posts': posts})

```



```

<!DOCTYPE html>
<html>
<head lang="en">
    <meta charset="UTF-8">
    <title></title>
</head>
<body>
<ul>
    {% for item in posts %}
        <li>{{ item }}</li>
    {% endfor %}
</ul>

<div class="pagination">
    <span class="step-links">
        {% if posts.has_previous %}
            <a href="?p={{ posts.previous_page_number }}">Previous</a>
        {% endif %}
        <span class="current">
            Page {{ posts.number }} of {{ posts.paginator.num_pages }}.
        </span>
        {% if posts.has_next %}
            <a href="?p={{ posts.next_page_number }}">Next</a>
        {% endif %}
    </span>
</div>
</body>
</html>

```



```

from django.shortcuts import render
from django.core.paginator import Paginator, EmptyPage, PageNotAnInteger

```

```

class CustomPaginator(Paginator):
    def __init__(self, current_page, max_pager_num, *args, **kwargs):
        """
        :param current_page: 当前页
        :param max_pager_num: 最多显示的页码个数
        :param args:
        :param kwargs:
        :return:
        """
        self.current_page = int(current_page)
        self.max_pager_num = max_pager_num
        super(CustomPaginator, self).__init__(*args, **kwargs)

    def page_num_range(self):
        # 当前页面
        # self.current_page
        # 总页数
        # self.num_pages
        # 最多显示的页码个数
        # self.max_pager_num
        print(1)
        if self.num_pages < self.max_pager_num:
            return range(1, self.num_pages + 1)
        print(2)
        part = int(self.max_pager_num / 2)
        if self.current_page - part < 1:
            return range(1, self.max_pager_num + 1)
        print(3)
        if self.current_page + part > self.num_pages:
            return range(self.num_pages + 1 - self.max_pager_num, self.num_p
        print(4)
        return range(self.current_page - part, self.current_page + part + 1)

L = []
for i in range(999):
    L.append(i)

def index(request):
    current_page = request.GET.get('p')
    paginator = CustomPaginator(current_page, 11, L, 10)
    # per_page: 每页显示条目数量
    # count: 数据总个数
    # num_pages: 总页数
    # page_range: 总页数的索引范围, 如: (1,10), (1,200)
    # page: page对象
    try:
        posts = paginator.page(current_page)
        # has_next 是否有下一页
        # next_page_number 下一页页码
        # has_previous 是否有上一页
        # previous_page_number 上一页页码
        # object_list 分页之后的数据列表
        # number 当前页
        # paginator paginator对象
    except PageNotAnInteger:
        posts = paginator.page(1)
    except EmptyPage:
        posts = paginator.page(paginator.num_pages)

    return render(request, 'index.html', {'posts': posts})

```



```

<!DOCTYPE html>
<html>
<head lang="en">
    <meta charset="UTF-8">
    <title></title>
</head>
<body>

<ul>
    {% for item in posts %}

        <li>{{ item }}</li>
    {% endfor %}
</ul>

<div class="pagination">
<span class="step-links">
{% if posts.has_previous %}
    <a href="?p={{ posts.previous_page_number }}">Previous</a>
{% endif %}

    {% for i in posts.paginator.page_num_range %}
        <a href="?p={{ i }}">{{ i }}</a>
    {% endfor %}

    {% if posts.has_next %}
        <a href="?p={{ posts.next_page_number }}">Next</a>
    {% endif %}
</span>

<span class="current">
Page {{ posts.number }} of {{ posts.paginator.num_pages }}.
</span>

</div>
</body>
</html>

```

二、自定义分页

分页功能在每个网站都是必要的，对于分页来说，其实就是根据用户的输入计算出应该在数据库表中的起始位置。

- 1、设定每页显示数据条数
- 2、用户输入页码（第一页、第二页...）
- 3、根据设定的每页显示条数和当前页码，计算出需要取数据表的起始位置
- 4、在数据表中根据起始位置取值，页面上输出数据

需求又来了，需要在页面上显示分页的页面。如： [上一页] [1] [2] [3] [4] [5] [下一页]

- 1、设定每页显示数据条数
- 2、用户输入页码（第一页、第二页...）
- 3、设定显示多少页号
- 4、获取当前数据总条数
- 5、根据设定显示多少页号和数据总条数计算出，总页数
- 6、根据设定的每页显示条数和当前页码，计算出需要取数据表的起始位置
- 7、在数据表中根据起始位置取值，页面上输出数据

```

#!/usr/bin/env python
# -*- coding:utf-8 -*-
from django.utils.safestring import mark_safe

class PageInfo(object):

    def __init__(self, current, totalItem, peritems=5):
        self.__current=current
        self.__peritems=peritems
        self.__totalItem=totalItem

    def From(self):
        return (self.__current-1)*self.__peritems

    def To(self):
        return self.__current*self.__peritems

    def TotalPage(self): #总页数
        result=divmod(self.__totalItem,self.__peritems)
        if result[1]==0:
            return result[0]
        else:
            return result[0]+1

def Custompager(baseUrl,currentPage,totalpage): #基础页, 当前页, 总页数
    perPager=11
    #总页数<11
    #0 -- totalpage
    #总页数>11
        #当前页大于5 currentPage-5 -- currentPage+5
            #currentPage+5是否超过总页数, 超过总页数, end就是总页数
        #当前页小于5 0 -- 11
    begin=0
    end=0
    if totalpage <= 11:
        begin=0
        end=totalpage
    else:
        if currentPage>5:
            begin=currentPage-5
            end=currentPage+5
            if end > totalpage:
                end=totalpage
        else:
            begin=0
            end=11
    pager_list=[]
    if currentPage<=1:
        first="<a href=''>首页</a>"
    else:
        first="<a href='%s%d'>首页</a>" % (baseUrl,1)
    pager_list.append(first)

    if currentPage<=1:
        prev="<a href=''>上一页</a>"
    else:
        prev="<a href='%s%d'>上一页</a>" % (baseUrl,currentPage-1)
    pager_list.append(prev)

    for i in range(begin+1,end+1):
        if i == currentPage:
            temp="<a href='%s%d' class='selected'>%d</a>" % (baseUrl,i,i)
        else:
            temp="<a href='%s%d'>%d</a>" % (baseUrl,i,i)
        pager_list.append(temp)
    if currentPage>=totalpage:

```

```

        next="<a href='#'>下一页</a>"
    else:
        next="<a href='%s%d'>下一页</a>" % (baseurl,currentPage+1)
    pager_list.append(next)
    if currentPage>=totalpage:
        last="<a href=''>末页</a>"
    else:
        last="<a href='%s%d'>末页</a>" % (baseurl,totalpage)
    pager_list.append(last)
    result=''.join(pager_list)
    return mark_safe(result)    #把字符串转成html语言

```

总结，分页时需要做三件事：

- 创建处理分页数据的类
- 根据分页数据获取数据
- 输出分页HTML，即： [上一页] [1] [2] [3] [4] [5] [下一页]

缓存

由于Django是动态网站，所有每次请求均会去数据进行相应的操作，当程序访问量小时，耗时必然会更加明显，最简单解决方式是使用：缓存，缓存将一个某个views的返回值保存至内存或者memcache中，5分钟内再有人来访问时，则不再去执行view中的操作，而是直接从内存或者Redis中之前缓存的内容拿到，并返回。

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)

1、配置

a、开发调试

```

# 此为开始调试用，实际内部不做任何操作
# 配置：
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.dummy.DummyCache',
        'TIMEOUT': 300,
        'OPTIONS': {
            'MAX_ENTRIES': 300,
            'CULL_FREQUENCY': 3,
        },
        'KEY_PREFIX': '',
        'VERSION': 1,
        'KEY_FUNCTION' 函数名
    }
}

# 自定义key
def default_key_func(key, key_prefix, version):
    """
    Default function to generate keys.

    Constructs the key used by all other methods. By default it prepends

```

```
the `key_prefix`. KEY_FUNCTION can be used to specify an alternate
function with custom key making behavior.
"""
return '%s:%s:%s' % (key_prefix, version, key)

def get_key_func(key_func):
    """
    Function to decide which key function to use.

    Defaults to ``default_key_func``.
    """
    if key_func is not None:
        if callable(key_func):
            return key_func
        else:
            return import_string(key_func)
    return default_key_func
```



b、内存



```
# 此缓存将内容保存至内存的变量中
# 配置:
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.locmem.LocMemCache',
        'LOCATION': 'unique-snowflake',
    }
}

# 注: 其他配置同开发调试版本
```



c、文件



```
# 此缓存将内容保存至文件
# 配置:
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.filebased.FileBasedCa
        'LOCATION': '/var/tmp/django_cache',
    }
}

# 注: 其他配置同开发调试版本
```



d、数据库



```
# 此缓存将内容保存至数据库

# 配置:
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.db.DatabaseCache',
        'LOCATION': 'my_cache_table', # 数据库表
```



```

    }
}

# 注：执行创建表命令 python manage.py createcachetable

```



e、Memcache缓存 (python-memcached模块)



```

# 此缓存使用python-memcached模块连接memcache

CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': '127.0.0.1:11211',
    }
}

CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': 'unix:/tmp/memcached.sock',
    }
}

CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': [
            '172.19.26.240:11211',
            '172.19.26.242:11211',
        ]
    }
}

```



f、Memcache缓存 (pylibmc模块)



```

# 此缓存使用pylibmc模块连接memcache

CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.PyLibMCCache',
        'LOCATION': '127.0.0.1:11211',
    }
}

CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.PyLibMCCache',
        'LOCATION': '/tmp/memcached.sock',
    }
}

CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.PyLibMCCache',
        'LOCATION': [
            '172.19.26.240:11211',
            '172.19.26.242:11211',
        ]
    }
}

```

```
}
}
```



g. Redis缓存 (依赖: pip3 install django-redis)



```
CACHES = {
    "default": {
        "BACKEND": "django_redis.cache.RedisCache",
        "LOCATION": "redis://127.0.0.1:6379",
        "OPTIONS": {
            "CLIENT_CLASS": "django_redis.client.DefaultClient",
            "CONNECTION_POOL_KWARGS": {"max_connections": 100}
            # "PASSWORD": "密码",
        }
    }
}
```



```
from django_redis import get_redis_connection
conn = get_redis_connection("default")
```

2、应用

a. 全站使用



使用中间件，经过一系列的认证等操作，如果内容在缓存中存在，则使用FetchFromCacheMiddle

```
MIDDLEWARE = [
    'django.middleware.cache.UpdateCacheMiddleware',
    # 其他中间件...
    'django.middleware.cache.FetchFromCacheMiddleware',
]
```

```
CACHE_MIDDLEWARE_ALIAS = ""
CACHE_MIDDLEWARE_SECONDS = ""
CACHE_MIDDLEWARE_KEY_PREFIX = ""
```



b. 单独视图缓存



方式一:

```
from django.views.decorators.cache import cache_page

@cache_page(60 * 15)
def my_view(request):
    ...
```

方式二:

```
from django.views.decorators.cache import cache_page

urlpatterns = [
    url(r'^foo/([0-9]{1,2})/$', cache_page(60 * 15)(my_view)),
]
```



c、局部视图使用



更多：[猛击这里](#)

序列化

关于Django中的序列化主要应用在将数据库中检索的数据返回给客户端用户，特别的Ajax请求一般返回的为Json格式。

1、serializers

```

1 from django.core import serializers
2
3 ret = models.BookType.objects.all()
4
5 data = serializers.serialize("json", ret)

```

2、json.dumps

```

1 import json
2
3 #ret = models.BookType.objects.all().values('caption')
4 ret = models.BookType.objects.all().values_list('caption')
5
6 ret=list(ret)
7
8 result = json.dumps(ret)

```

由于json.dumps时无法处理datetime日期，所以可以通过自定义处理器来做扩展，如：

[+ View Code](#)

信号

Django中提供了“信号调度”，用于在框架执行操作时解耦。通俗来讲，就是一些动作发生的时候，信号允许特定的发送者去提醒一些接受者。

1、Django内置信号

```

1 Model signals
2     pre_init                # django的model执行其构造方法前，自动触发
3     post_init               # django的model执行其构造方法后，自动触发
4     pre_save                 # django的model对象保存前，自动触发
5     post_save                # django的model对象保存后，自动触发
6     pre_delete               # django的model对象删除前，自动触发
7     post_delete              # django的model对象删除后，自动触发
8     m2m_changed              # django的model中使用m2m字段操作第三张表（add
9     class_prepared           # 程序启动时，检测已注册的app中model类，对于每
10 Management signals
11     pre_migrate              # 执行migrate命令前，自动触发
12     post_migrate             # 执行migrate命令后，自动触发
13 Request/response signals
14     request_started          # 请求到来前，自动触发

```

```

15     request_finished          # 请求结束后，自动触发
16     got_request_exception     # 请求异常后，自动触发
17 Test signals
18     setting_changed           # 使用test测试修改配置文件时，自动触发
19     template_rendered        # 使用test测试渲染模板时，自动触发
20 Database Wrappers
21     connection_created        # 创建数据库连接时，自动触发

```

对于Django内置的信号，仅需注册指定信号，当程序执行相应操作时，自动触发注册函数：

```

from django.core.signals import request_finished
from django.core.signals import request_started
from django.core.signals import got_request_exception

from django.db.models.signals import class_prepared
from django.db.models.signals import pre_init, post_init
from django.db.models.signals import pre_save, post_save
from django.db.models.signals import pre_delete, post_delete
from django.db.models.signals import m2m_changed
from django.db.models.signals import pre_migrate, post_migrate

from django.test.signals import setting_changed
from django.test.signals import template_rendered

from django.db.backends.signals import connection_created

def callback(sender, **kwargs):
    print("xxoo_callback")
    print(sender, kwargs)

xxoo.connect(callback)
# xxoo指上述导入的内容

```

```

from django.core.signals import request_finished
from django.dispatch import receiver

@receiver(request_finished)
def my_callback(sender, **kwargs):
    print("Request finished!")

```

2、自定义信号

a. 定义信号

```

1 import django.dispatch
2 pizza_done = django.dispatch.Signal(providing_args=["toppings", "size"])

```

b. 注册信号

```

1 def callback(sender, **kwargs):
2     print("callback")
3     print(sender, kwargs)
4
5 pizza_done.connect(callback)

```

c. 触发信号

```
1 from 路径 import pizza_done
2
3 pizza_done.send(sender='seven',toppings=123, size=456)
```

由于内置信号的触发者已经集成到Django中，所以其会自动调用，而对于自定义信号则需要开发者在任意位置触发。

更多：[猛击这里](#)



作者：武沛齐
出处：<http://www.cnblogs.com/wupeiqi/>
本文版权归作者和博客园共有，欢迎转载，但未经作者同意必须保留这段声明，且在文章页面明显位置给出原文连接。

好文要顶

关注我

收藏该文







武沛齐
关注 - 44
粉丝 - 10098
[+加关注](#)

341

posted @ 2016-03-06 06:12 武沛齐 阅读(58684) 评论(9) 编辑 收藏

评论列表

- #1楼 2016-08-19 19:32 我当道士那儿些年

回复 引用

这么好的文章都没人评论。
刘奶奶找牛奶买榴莲牛奶，牛奶给刘奶奶拿榴莲牛奶，刘奶奶说牛奶的榴莲牛奶不如柳奶奶的榴莲牛奶，牛奶说柳奶奶的榴莲牛奶会流奶，柳奶奶听见了大骂牛奶你的榴莲牛奶才会流奶。柳奶奶和牛奶泼榴莲牛奶吓坏了刘刘奶奶找牛奶买牛奶，牛奶给刘奶奶拿牛奶，刘奶奶说牛奶的牛奶不如柳奶奶的牛奶，牛奶说柳奶奶的牛奶会流奶柳奶奶听见了大骂牛奶你的才会流奶，柳奶奶和牛奶泼牛奶吓坏了刘奶奶，大骂再也不买柳奶奶和牛奶的牛奶

支持(11) 反对(7)
- #2楼 2017-09-18 19:30 卧槽，我是谁

回复 引用

好文章应该评论.....

支持(3) 反对(0)
- #3楼 2017-12-25 20:10 刘钊up

回复 引用

a a a

支持(2) 反对(0)
- #4楼 2018-01-18 16:34 MaryMaryTang

回复 引用

@ 我当道士那儿些年
绕口令不错诶

支持(2) 反对(1)
- #5楼 2018-01-30 18:44 MaryMaryTang

回复 引用

@ 非白
你的form表单的action的地方写错了，应该是action = "/zhuce/"

支持(0) 反对(1)
- #6楼 2018-02-05 16:09 Cool·

回复 引用

@ 非白
<form action="/zhuce.html/" method="POST">

url(r'^zhuce.html/', zhuce),

你改成这样试试

支持(0) 反对(0)
- #7楼 2018-02-05 16:20 非白

回复 引用

@ Cool·
还是不行_(:з] ∠)_

支持(0) 反对(0)

#8楼 2018-10-05 18:40 fangsheng420

回复 引用

@ 非白
fff

支持(0) 反对(0)

#9楼 2020-04-08 11:43 Fmaj-7

回复 引用





同一张表一对多忘了。。

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

发表评论

编辑 预览

B    

支持 Markdown

提交评论 退出 订阅评论

[Ctrl+Enter快捷键提交]

- 【推荐】超50万行VC++源码：大型组态工控、电力仿真CAD与GIS源码库
- 【推荐】为自己发“声”——声网RTC征文大赛在园子里征稿
- 【推荐】未知数的距离，毫秒间的传递，声网与你实时互动
- 【推荐】了不起的开发者，挡不住的华为，园子里的品牌专区
- 【推荐】电子签名认准大家签，上海CA权威认证

相关博文：

- Python之路【第十七篇】： Django【进阶篇】
- Python之路【第十七篇】： Django【进阶篇】
- Python之路【第十七篇】： Django【进阶篇】
- Python之路【第十七篇】： Django【进阶篇】
- Python之路【第十七篇】： Django【进阶篇】
- » 更多推荐...

最新 IT 新闻：

- 特朗普感染新冠，其治疗方案透露了哪些信息？
- 便利蜂公布双节大数据：十一当天服务人次超百万 销售额同比增50%
- 蒙古国捐赠的3万只羊真的要来了！都是活的
- SpaceX发射前2秒紧急叫停

· 吃货福音？全球首份AI生成的调味诞生！
» 更多新闻...