

Django手册

Django整体请求流程

```
客户端 ----> process_request[]  
process_request[] ----> url  
url ----> process_view[]  
process_view[] ----> view  
view ----> model  
model ----> view  
view ----> template  
template ----> view  
view ----> response  
response ----> process_response[]  
----> 最后回到客户端
```

注意：

在view和template之间还有： process_template_response

process_exception函数是捕获异常函数，类比于flask中的
@app.errorhandler装饰器

基础知识

Linux安装软件

Linux系统中有两种常用系统包管理工具 yum 和 apt。

低版本中安装包使用 apt-get，新的现在只需要写 apt就ok了。

apt 指令（兼容apt-get和apt-cache）

apt install xxx 安装xxx软件

apt remove xxx移除xxx软件

apt autoremove xxx移除xxx软件和自动安装且不使用的包

虚拟化技术

虚拟化技术

- 虚拟机
- 虚拟容器
 - Docker
- 虚拟环境
 - Python专用
 - 将Python依赖隔离

指定源下载包 `pip install pymysql -i https://pypi.douban.com/simple`

在window下搭建虚拟环境需要下载

`pip install virtualenv`

`pip install virtualenvwrapper-win`

```
pip install virtualenvwrapper-win
```

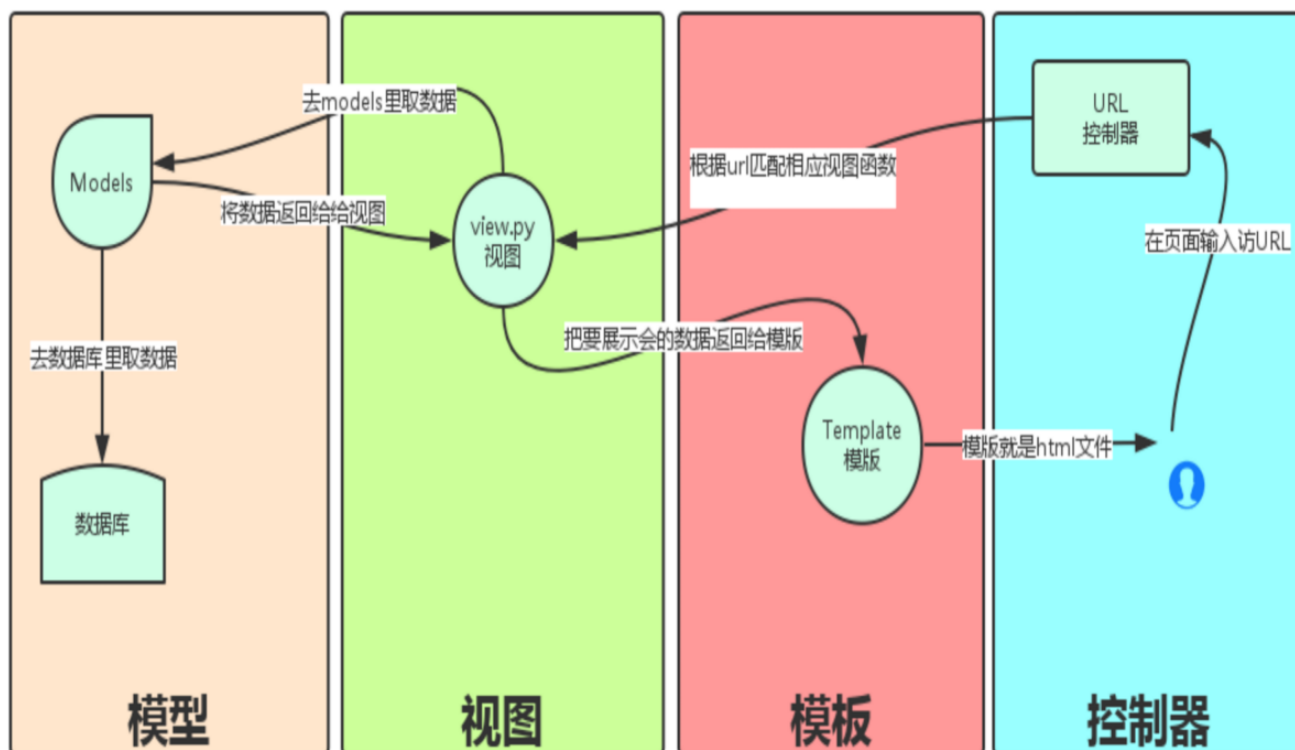
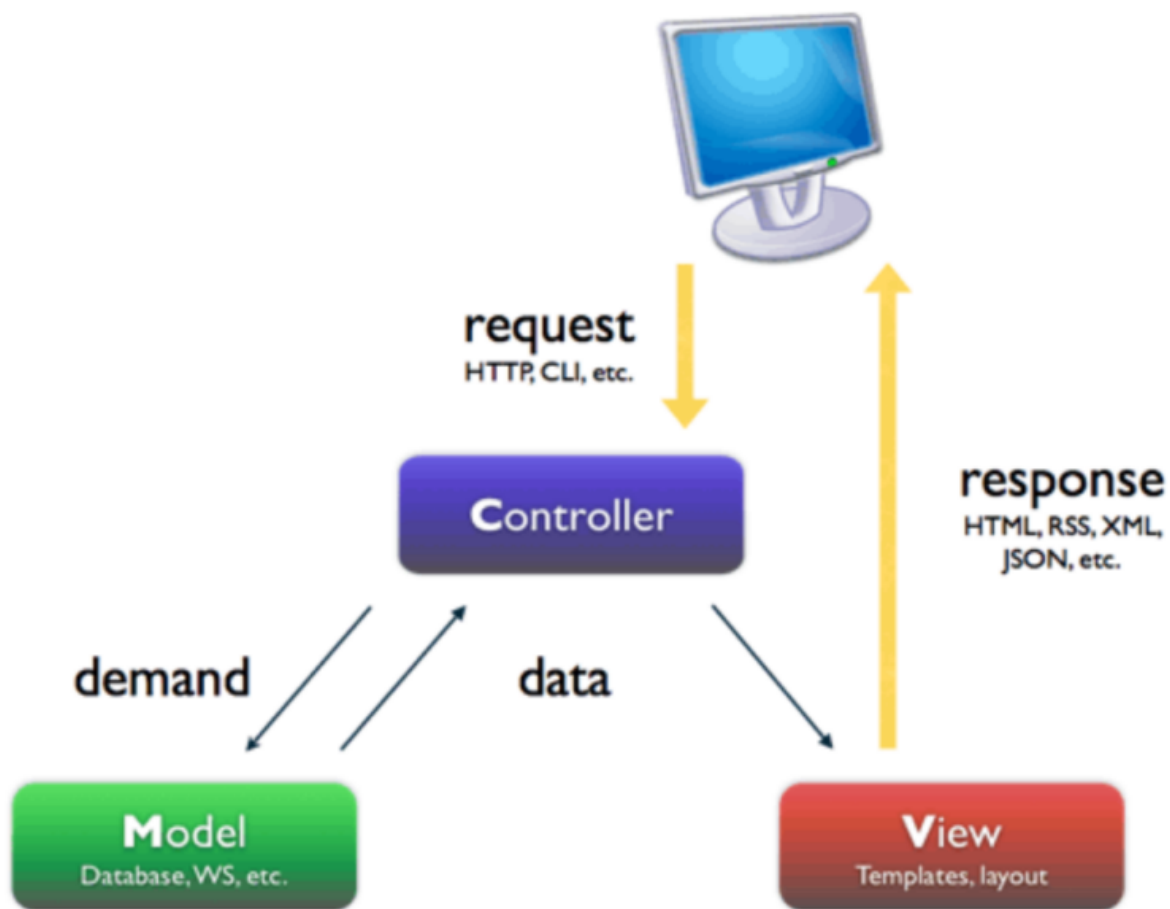
虚拟环境的使用

创建：

虚拟环境

- virtualenvwrapper
 - 对virtualenv的包装
 - mkvirtualenv
 - rmvirtualenv
 - workon
 - deactivate
- virtualenv
 - virtualenv xxx
 - source /xx/xx/activate
 - deactivate
 - rm -rf xxx
 - 指令在哪调用，虚拟就在哪生成

图解mvc和mtv



Django环境搭建

1、虚拟环境操作

创建:mkvirtualenv 虚拟环境名称 -p python的路径 /usr/bin/pythonX

删除:rmvirtualenv 虚拟环境名称

进入:workon 虚拟环境名称

退出:deactivate

pip install xxx:安装xxx依赖包

pip list:查看所有依赖包

pip freeze:查看虚拟环境新安装的包

2、安装django

安装Django:pip install django (也可指定某一版本 django==1.8.2)

测试Django是否安装成功

进入python环境

import django

django.get_version()

创建项目

创建项目： `django-admin startproject xxx` 创建一个名字为xxx的工程
创建应用： `python manage.py startapp 应用名称`

Django目录介绍

`manage.py`:是Django用于管理本项目的命令行工具，之后进行站点运行，数据库自动生成等都是通过本文件完成。

`HelloDjango/__init__.py`告诉python该目录是一个python包，暂无内容，后期一些工具的初始化可能会用到

`HelloDjango/settings.py` Django项目的配置文件，默认状态其中定义了本项目引用的组件，项目名，数据库，静态资源等。

`HelloDjango/urls.py` 维护项目的URL路由映射，即定义当客户端访问时由哪个模块进行响应。

`HelloDjango/wsgi.py` 定义WSGI的接口信息，主要用于服务器集成，通常本文件生成后无需改动。

启动开发服务器

```
python manager.py runserver [ip:port]
```

可以直接进行服务运行 默认执行起来的端口是8000

也可以自己指定ip和端口

ip指定为0.0.0.0的时候，匹配本机的全部ip

浏览器访问:localhost:8000 可以看到服务器启动成功

数据文件的迁移

迁移的概念:就是将模型映射到数据库的过程

生成迁移:python manager.py makemigrations

执行迁移:python manager.py migrate

介绍DataBase模块，可以直接连接操作数据库Sqlite和MySQL

```
'ENGINE':'django.db.backends.mysql',  
'NAME':'Learn',  
'USER':'root',  
'PASSWORD':'rock1204',  
'HOST':'127.0.0.1',  
'PORT':'3306',
```

安装连接数据库驱动

在python 2.x

```
pip install mysql-python
```

在python 3.x

```
pip install pymysql
```

安装后还需要在项目文件__init__.py中添加初始化代码

```
import pymysql
```

```
pymysql.install_as_MySQLdb()
```

应用的创建

```
python manager.py startapp XXX
```

创建名称为XXX的应用

创建之后可以通过tree来看一下目录结构

使用应用前需要将应用配置到项目中，在settings.py中将应用加入到INSTALLED_APPS选项中

应用目录介绍

__init__.py:其中暂无内容，使得app成为一个包

admin.py:管理站点模型的声明文件，默认为空

apps.py:应用信息定义文件，在其中生成了AppConfig，该类用于定义应用

apps.py:应用信息定义文件，在其中生成了 AppConfig，该类用于定义应用名等数据

models.py:添加模型层数据类文件

views.py:定义URL相应函数（路由规则）

migrations包:自动生成，生成迁移文件的

tests.py:测试代码文件

整体书写流程简介

基本视图

首先我们在views.py中建立一个路由响应函数

```
from django.http import HttpResponse
```

```
def welcome(request):  
    return HttpResponse('HelloDjango');
```

接着我们在urls中进行注册

```
from App import views  
url(r'^welcome/',views.welcome)
```

基于模块化的设计，我们通常会在每个app中定义自己的urls

在项目的urls中将app的urls包含进来

```
from django.conf.urls import include  
url(r'^welcome/',include('App.urls'))
```

模板使用

模板实际上就是我们用HTML写好的页面

创建模板文件夹,两种, 在工程目录的需要注册

settings中的TEMPLATES中的DIRS中添加

```
os.path.join(BASE_DIR, 'templates')
```

在模板文件夹中创建模板文件

在views中去加载渲染模板

1. from django.template import loader

```
template = loader.get_template('xxx')
```

```
return HttpResponse(template.render())
```

2 return render(request, 'xxx')

定义模型

定义年级

```
class Grade (models.Model) :
```

```
    gname = models.CharField(max_length=10)
```

```
    gdate = models.DateTimeField()
```

```
    ggirlnum = models.IntegerField()
```

```
    gboynum = models.IntegerField()
```

```
    isDelete = models.BooleanField()
```

定义学生

```
class Students(models.Model):
```

```
    sname = models.CharField(max_length=20)
```

```
    sgender = models.BooleanField(default=True)
```

```
    sage = models.IntegerField()
```

```
    sinfo = models.CharField(max_length=20)
```

```
sino = models.CharField(max_length=20)
isDelete = models.BooleanField(default=False)
# 关联外键
sgrade = models.ForeignKey(Grade)
```

进行迁移操作

结合使用

在urls配置路由规则

在views中调用models中的函数进行查询

在views中对模板进行渲染

模板获取views中传过来的数据

渲染后的模板显示给用户

挖坑

```
{{ xxx }}
```

表达式 {% for student in students %}

```
    {{student}}
```

```
{% endfor %}
```

配置文件全套配置

- 28行 `ALLOWED_HOSTS = ["*"]`
- 40行 添加APP
- 59行 项目目录下创建templates模板，然后添加 `os.path.join(BASE_DIR,"templates")`
- 82行 配置数据库

-

```
'ENGINE': 'django.db.backends.mysql',  
'NAME': 'GP1DjangoDay05',  
'USER': 'root',  
'PASSWORD': 'rock1204',  
'HOST': 'localhost',  
'PORT': 3306
```

- 语言、时间、时间标准配置

-

```
LANGUAGE_CODE = 'zh-hans'  
TIME_ZONE = 'Asia/Shanghai'  
USE_TZ = False
```

- 静态资源的配置

-

```
STATICFILES_DIRS = [  
    os.path.join(BASE_DIR, 'static'),  
]
```

- 媒体资源根目录配置

-

```
MEDIA_ROOT = os.path.join(BASE_DIR, 'static/upload/')
```

Model

ORM

图片1

数据的增、删、改、查

数据测试增删改查

查询: 类名.objects.all()

Grade.objects.all()

插入数据: 对象.save()

grade_one = Grade()

grade_one.gname='python1705'

grade_one.gdate=datetime(year=2017,month=7,day=28)

grade_one.ggirlnum=5

grade_one.gboynum = 60

grade_one.save()

查看某个指定对象

类名.objects.get(pk=1)

Grade.objects.get(pk=1)

修改数据

```
模型对象.属性=属性值  
模型对象.save()  
grade_one.gboynum = 55  
grade_one.save()
```

删除数据

```
模型对象.delete()  
grade_one.delete()
```

测试关联对象

```
stu = Students()  
stu.sname='rock'  
stu.sgender=True  
stu.sage=20  
stu.sinfo='小天才'  
stu.sgrade=grade_one  
stu.save()
```

获得关联对象集合:获取班级中的所有学生

```
对象名.关联的类名小写_set.all()  
grade_one.student_set.all()
```

Model字段类型

字段类型

·AutoField

·一个根据实际ID自动增长的IntegerField，通常不指定如果不指定，一个主键字段将自动添加到模型中

CharField(max_length=字符串长度)

·CharField(max_length=字符串长度)

·字符串，默认的表单样式是 TextInput

·TextField

·大文本字段，一般超过4000使用，默认的表单控件是Textarea

·IntegerField

·整数

·DecimalField(max_digits=None, decimal_places=None)

·使用python的Decimal实例表示的十进制浮点数

·参数说明

·DecimalField.max_digits

·位数总数

·DecimalField.decimal_places

·小数点后的数字位数

·FloatField

·用Python的float实例来表示的浮点数

·BooleanField

·true/false 字段，此字段的默认表单控制是CheckboxInput

·NullBooleanField

·支持null、true、false三种值

·DateField([auto_now=False, auto_now_add=False])

·使用Python的datetime.date实例表示的日期

·参数说明

·DateField.auto_now

·每次保存对象时，自动设置该字段为当前时间，用于"最后一次修改"的时间戳，它总是使用当前日期，默认为false

·DateField.auto_now_add

- DateField.auto_now_add

- 当对象第一次被创建时自动设置当前时间，用于创建的时间戳，它总是使用当前日期，默认为false

- 说明

- 该字段默认对应的表单控件是一个TextInput. 在管理员站点添加了一个JavaScript写的日历控件，和一个“Today”的快捷按钮，包含了一个额外的invalid_date错误消息键

- 注意

- auto_now_add, auto_now, and default 这些设置是相互排斥的，他们之间的任何组合将会发生错误的结果

- TimeField

- 使用Python的datetime.time实例表示的时间，参数同DateField

- DateTimeField

- 使用Python的datetime.datetime实例表示的日期和时间，参数同DateField

- FileField

- 一个上传文件的字段

- ImageField

- 继承了FileField的所有属性和方法，但对上传的对象进行校验，确保它是个有效的image

字段选项

字段选项

- 概述

通过字段选项，可以实现对字段的约束

- 通过子段选项，可以实现对子段的约束

- 在字段对象时通过关键字参数指定

- null

- 如果为True, Django 将空值以NULL 存储到数据库中, 默认值是 False

- blank

- 如果为True, 则该字段允许为空白, 默认值是 False

- 注意

- null是数据库范畴的概念, blank是表单验证范畴的

- db_column

- 字段的名称, 如果未指定, 则使用属性的名称

- db_index

- 若值为 True, 则在表中会为此字段创建索引

- default

- 默认值

- primary_key

- 若为 True, 则该字段会成为模型的主键字段

- unique

- 如果为 True, 这个字段在表中必须有唯一值

- 模型中的元信息

- class Meta:

- db_table = xxx 定义数据表名, 推荐使用小写字母

- ordering = [] -----注意: id升序 -id降序

查询、聚合函数、F对象、Q对象

基本概念：

查询集表示从数据库获取的对象集合

查询集可以有多个过滤器

过滤器就是一个函数，基于所给的参数限制查询集结果

从SQL角度来说，查询集合和select语句等价，过滤器就像where条件

在管理器上调用方法返回查询集

查询经过过滤器筛选后返回新的查询集，所以可以写成链式调用

返回查询集的方法称为过滤器

all() 返回所有数据

filter() 返回符合条件的数据

exclude() 过滤掉符合条件的数据

order_by() 排序

values() 一条数据就是一个字典，返回一个列表

返回单个对象

get(): 返回一个满足条件的对象

如果没有找到符合条件的对象，会引发 模型类.DoesNotExist异常

如果找到多个，会引发 模型类.MultiObjectsReturned 异常

first(): 返回查询集中的第一个对象

last(): 返回查询集中的最后一个对象

count(): 返回当前查询集中的对象个数

exists(): 判断查询集中是否有数据, 如果有数据返回True没有反之

限制查询集和查询集的缓存

限制查询集, 可以使用下标的方法进行限制, 等同于sql中limit

studentList = Student.objects.all()[0:5] 下标不能是负数

查询集的缓存: 每个查询集都包含一个缓存, 来最小化对数据库的访问

在新建的查询集中, 缓存首次为空, 第一次对查询集求值, 会发生数据缓存, django会将查询出来的数据做一个缓存, 并返回查询结构, 以后的查询直接使用查询集的缓存。

比较运算符

exact: 判断, 大小写敏感, filter(isDelete = False)

contains: 是否包含, 大小写敏感, filter(sname__contains='赵')

startswith,endswith: 以values开头或结尾, 大小写敏感

以上四个在运算符前加上 i(ignore)就不区分大小写了 iexact...

isnull,isnotnull: 是否为空, filter(sname__isnull=False)

in: 是否包含在范围内,filter(pk__in=[2,4,6,8])

gt,gte,lt,lte: 大于, 大于等于, 小于小于等于filter(sage__gt=30)

时间的

year month day week day hour minute second:

```
year,month,day,week_day,hour,minute,second.  
filter(lasttime__year=2017)
```

查询快捷:

pk: 代表主键, filter(pk=1)

跨关系查询:

模型类名__属性名__比较运算符, 实际上就是处理的数据库中的join
grade=Grade.objects.filter(student__scontend__contains='楚 人美')
描述中带有'楚人美'这三个字的数据属于哪个班级

聚合函数

使用aggregate()函数返回聚合函数的值

Avg: 平均值

Count: 数量

Max: 最大

Min: 最小

Sum: 求和

例如: Student.objects().aggregate(Max('sage'))

F对象

可以使用模型的A属性与B属性进行比较

```
grades = Grade.objects.filter(ggirlnum__gt=F('gboynum'))
```

F对象支持算术运算

```
grades = Grade.objects.filter(ggirlnum__gt=F('gboynum') + 10)
```

Q对象

过滤器的方法中的关键参数, 常用于组合条件

年龄小于25

```
Student.objects.filter(Q(sage__lt=25))
```

Q对象语法支持 | (or), & (and), ~(取反)

年龄大于等于的

```
Student.objects.filter(~Q(sage__lt=25))
```

模型成员

模型成员

类属性

显性: 自己写的那些

隐性: objects 是一个Manager类型的一个对象, 作用于数据库进行交互

当模型类没有指定管理器的时候, Django会自动为我们创建模型管理器

当然我们也可以自定义管理器, 重写get_queryset()方法就可以

```
class Student(models.Model):
```

```
    stuManager = models.Manager()
```

当自定义模型管理器时, objects就不存在了, Django就不会为我们自动生成模型管理器

自定义管理器类

模型管理器是Django的模型与数据库进行交互的接口, 一个模型可以有多个模型管理器

自定义模型管理器作用:

可以向管理器中添加额外的方法 • 修改管理器返回的原始查询集
提供创建对象的方式

```
class StudentManager(models.Manager):
```

```
class StudentManager(models.Manager):
    def get_queryset(self):
        return
super(StudentManager,self).get_queryset.filter(isDelete=False)

    def createStudent(self):
        stu = self.model()
        # 设置属性
        return stu
```

使用查询时的常见坑

获取单个对象

- 查询条件没有匹配的对象，会抛异常，DoesNotExist
- 如果查询条件对应多个对象，会抛异常，MultipleObjectsReturned

first和last

- 默认情况下可以正常从QuerySet中获取
- 隐藏bug
 - 可能会出现 first和last获取到的是相同的对象
 - 显式，手动写排序规则

切片

和python中的切片不太一样

- 和python中的切片不太一样
- QuerySet[5:15] 获取第五条到第十五条数据
- 相当于SQL中limit和offset

缓存集

- filter
- exclude
- all
- 都不会真正的去查询数据库
- 只有我们在迭代结果集，或者获取单个对象属性的时候，它才会去查询数据库
- 懒查询
- 为了优化我们结构和查询

查询条件

- 属性_运算符=值
- gt
- lt
- gte
- lte
- in 在某一集合中
- contains 类似于 模糊查询 like
- startswith 以xx开始 本质也是like
- endswith 以 xx 结束 也是like
- exact
- 前面同时添加i , ignore 忽略
- iexact
- icontains

istartswith

- istartswith
- iendswith
- django中查询条件有时区问题
 - 关闭django中自定义的时区
 - 在数据库中创建对应的时区表

F

- 可以获取我们属性的值
- 可以实现一个模型的不同属性的运算操作
- 还可以支持算术运算

Q

- 可以对条件进行封装
- 封装之后，可以支持逻辑运算
 - 与 & and
 - 或 | or
 - 非 ~ not

模型成员

- 显性属性
 - 开发者手动书写的属性
- 隐性属性
 - 开发者没有书写，ORM自动生成的

如果你把隐性属性手动去掉了，系统就不会再为你立生隐性属性了

- 如果你把隐属性声明了，系统就会认为你声明了隐属性

模型关系

- 数据库知识

- CASCADE

在父表上update/delete记录时，同步update/delete掉子表的匹配记录

- SET NULL

在父表上update/delete记录时，将子表上匹配记录的列设为null (要注意子表的外键 列不能为-not null)

- NO ACTION

如果子表中有匹配的记录,则不允许对父表对应候选键进行update/delete操作

- RESTRICT

同no action, 都是立即检查外键约束

- SET NULL

父表有变更时,子表将外键列设置成一个默认的值 但InnoDB不能识别

- 1:1

- 定义：id_person = models.OneToOneField(Person, null=True, blank=True, on_delete=models.SET_NULL)

- 实现：主表、从表，从表定义外键关系，并且是外键唯一

- 创建主表（人）、创建从表（身份证）--注意：设置外键null = True和black=True、绑定两张表
- 数据删除
 - 默认创建：
 - 主表删除，从表跟着删除，从表删除，而主表不变
 - on_delete=models.PROTECT: (在定义外键的时候加上)
 - 主表上只要有关联数据没有删除，就不能删除，除非关联的所有数据都删除完毕即可删除主表数据
 - SET
 - SET_NULL:删除主表字段时，将从表的关联字段设置为空（字段允许为NULL）
 - SET_DEFAULT:删除主表字段时，将从表的关联字段设置为默认值（字段设置默认值）
 - SET():直接传值，删除主表字段时，将从表的关联字段设置成为传入的值（传入值）
- 数据的获取
 - 从表获取主表
 - 主表某个对象 = 找到从表中的某个对象.外键属性
 - ex: person = idcard.i_person

- 从表某个对象 = 主表某个对象.从表类名小写 (通过隐性属性获得)

- ex: idcard = person.idcard

- 1: n

- 定义: s_grade = models.ForeignKey(grade,null=True)

- 实现:

- 主表、从表: 直接用ForeignKey实现、绑定

- 数据删除: 同1:1

- 数据获取:

- 主获取从隐性属性 主表对象.从表类名小写
_set (是一个Manager子类) 支持all()、filter()等操作
 - 从获取主直接显性属性获取

- n:m

- 定义: g_customer = ManyToManyField(customer)

- 实现:

- 主从可跌倒, 申明ManyToManyField即可, 生成一张新表, 通过两个Manger子类进行管理

- 数据获取:

- 从表获取通过显性属性 (Manger子类)
 - 主表获取通过级联类名_set(Manger子类)
 - 操作方法
 - add(对象): 添加关系
 - remove(对象): 移除关系

- `clear()`: 移除全部关系
- `set(一组对象)`: 同时添加多个关系

模型逆关联

- `python manage.py inspectdb`
- 可以直接生成Model
 - `python manage.py inspectdb > models.py`
 - 生成的Model拥有元信息 `manage=False`

模型继承

- 直接继承
 - 父类生成一张表，子类生成一张表，然后通过一对一关系连接两张表
 - 缺点：父表中的数据会越来越多，关系也越来越复杂，导致对数据操作时效率变慢
- 在类中添加Meta指定是否抽象
 - 不产生父表，在每个子表中拥有全部字段，例子如下
 - `class Animal(models.Model):`
 - `xxx`
 - `class Meta:`
 - `abstract = True/False`
 - `class Dog(Animal):`
 - `xxx`

Template模板

模板处理

- 模板的处理分过程
 - 加载模板
 - 渲染模板

模板语法

- 模板语法
 - 变量：{{ grade }}
 - 模板中的点语法
 - 字典查询 dicts.name
属性或者方法 grade.gname
索引 grades.0.gname
 - if语法
 - {% if true %}
 - {% elif true %}
 - {% else %}
 - {% endif %}
 - for语法
 - {% for 变量 in 列表 %} 语句1 {% empty %} 语句2{% endfor %}

当列表为空或不存在时,执行empty之后的语句 {{ forloop.counter }} 表示当前是第几次循环, 从1数数 {{ forloop.counter0 }}表示当前是第几次循环, 从0数数 {{ forloop.revcounter }}表示当前是第几次循环, 倒着数数, 到1停 {{ forloop.revcounter0 }} 表示当前第几次循环, 倒着数, 到0停 {{ forloop.first }} 是否是第一个 布尔值 {{ forloop.last }} 是否是最后一个 布尔值

- 注释
 - 单行注释
 - {# 被注释掉的内容 #}
 - 多行注释
 - {% comment %} 内容 {% endcomment %}
- 乘除
 - {% widthratio 数 分母 分子 %}
 - 整除
 - {% if num|divisibleby:2 %}
- 如果相等
 - {% ifequal value1 value2 %} 语句 {% endifequal %}
- 如果不相等
 - ifnotequal

- url反向解析
 - {% url 'namespace:name' p1 p2 %}
- csrf_token 防止跨站攻击
 - {% csrf_token %}
- 过滤器
 - 作用：在变量显示前修改
 - {{ var|过滤器 }}
- 加减法
 - {{ p.page | add : 5 }} (加法)
 - {{ p.page | add : -5 }} (减法)
- lower
 - {{ p.pname|lower }} (全部转化为小写)
 - {{ p.pname|upper }} (全部转化为大写)
- 过滤器可以传递参数，参数需要用引号引起来
 - 比如join: {{ students|join '=' }}
- 默认值
 - 作用：如果变量没有被提供或者为False，空，会使用默认值
 - {{ var|default value }}
- 处理data
 - {{ dateVal | date:'y-m-d' }}

- HTML渲染
 - 渲染成HTML
 - `html:{{ code|safe}}`
`{% autoescape off%} code {%`
`endautoescape %}`
 - 不想渲染
 - `{% autoescape on%} code {%`
`endautoescape %}`
- 防跨站攻击
 - 在表单中添加`{% csrf_token %}`
 在settings中的中间件MIDDLEWARE中配置打开
`'django.middleware.csrf.CsrfViewMiddleware'`,
- 模板继承
 - 挖坑
 - `{% block XXX%} code {% endblock %}`
 - 继承
 - `extends` 继承, 写在开头位置 `{%`
`extends '父模板路径' %}`
 - 组装
 - `{% include '模板文件' %}`
- 静态资源
 - 先`{% load static %}`
 - 然后`{% static 'css/bootstrap.css' %}`

- 加载模板技巧

-

```
html_message =  
loader.get_template('user/activate.html').render(data)
```

视图

双R

- Request
 - method: 获取访问方式 (常见有POST、GET)
 - path: 获取访问路径
 - GET
 - 类字典结构
 - 一个key允许对应多个值
 - get: 获取GET方式传入的数据
 - getlist: 获取某个键对应的多个值
 - POST
 - 同GET
 - META
 - 客户端元信息
 - REMOTE_ADDR来客IP地址.....

- session
 - 类似字典，表示会话
- COOKIES
 - 字典，包含了所有的COOKIE
- FILES
 - 类似字典，包含了上传文件
- encoding: 编码方式
- is_ajax()
 - 判断是否是ajax(),通常用在移动端和js中
- Response
 - 属性
 - content 返回的内容
 - charset 编码格式
 - status_code 相应状态码
 - content-type MIME类型
 - 方法
 - init 初始化内容
 - write("hellow") 直接写出文本
 - flush() 冲刷缓冲区
 - set_cookie(key,value="xxx",max_age=None,exprise=None)

- max_age: 整数, 指定cookie过期时间
expries : 整数, 指定过期时间, 还支持是一个datetime或 timedelta, 可以指定一个具体日期时间max_age和expries两个选一个指定
- 过期时间的几个关键时间
 - max_age 设置为 0 浏览器关闭失效
 - 设置为None永不过期
 - expires=timedelta(days=10) 10天后过期
- delete_cookie(key) 删除cookie
- 分类
 - HttpResponseRedirect
 - 实现服务器内部跳转, 重定向
 - 简写: redirect配合reverse使用
 - JsonResponse
 - 返回json数据
 - 使用: JsonResponse (dict)
 - 类型: Content-type是application/json

COOKIE、SESSION、TOKEN

cookie、session、token

cookie 客户端云站技术

用于和服务端进行验证

使用：

添加

```
response.set_cookie(键,值)
```

获取

```
request.COOKIES.get(键)
```

删除

```
response.delete_cookie('sessionid')
```

session 服务器会话技术

理解：

django框架中不用手动设置cookie的sessionid值，django框架会帮我们自动生成

当跳转页面需要验证身份时，也不需要手动验证，框架帮我们做了session和 cookie的验证工作，我们只需要获取session中的值即可，当获取到的值为空时，

说明身份不符，如果获取到值，说明是此用户的信息

使用方法：

添加

```
request.session["id"] = u_id
```

存储到数据库中使用了Base64编码

cookie中自动生成key为sessionid的键值对

获取

```
request.session.get(键)
```

如果没有找到对应的返回None

删除

```
request.session.flush()
```

```
request.session.clear()清除所有会话
```

token 服务器会话技术

需要手动验证cookie中的token值和数据库中的token是否对应

自定义错误视图

- 在工程的templates目录中创建对应的错误的文件，在关闭Debug的模式下可以测试

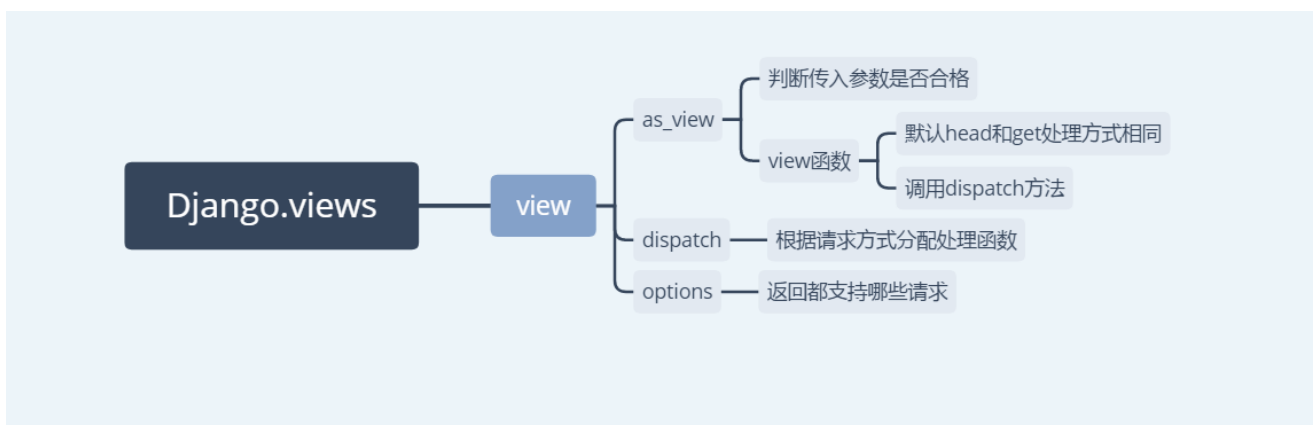
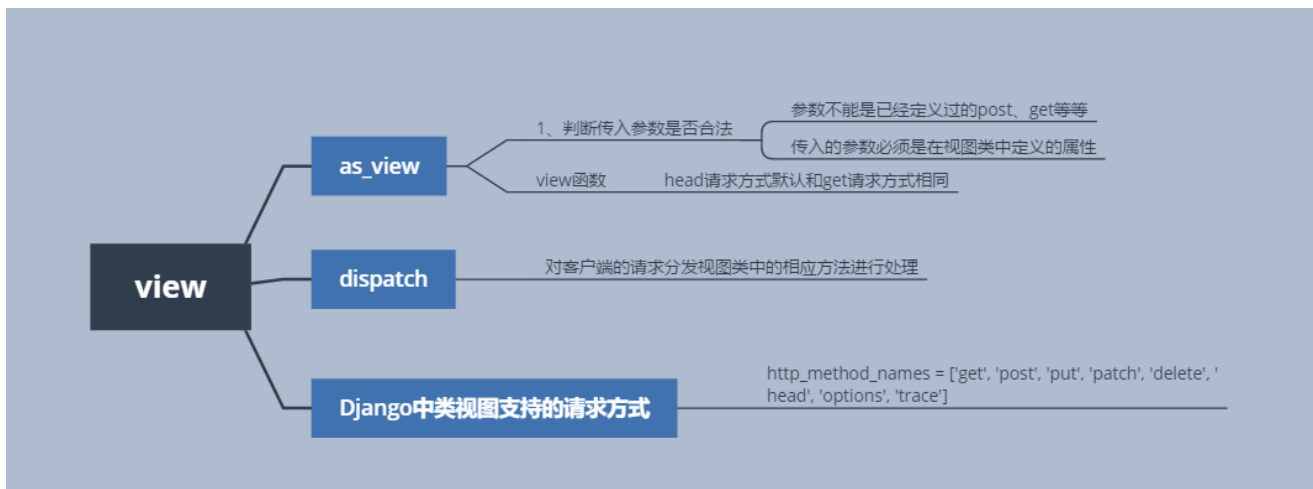
URL反向解析

- 从url中获取参数
 - 非关键字参数
 - `url(r'^grade/(\d+)$',views.getStudents)`
 - `url(r'^news/(\d{4})/(\d+)/(\d+)$',views.getNews)`, 匹配多个参数
 - 关键字参数
 - `url(r'^news/(?P\d{4})/(?P\d)+/(?P\d+)$',views.getNews)`
- 在根urls中
 - `url(r'^views/',include('app.views',namespace='app'))`
- 在子urls中
 - `url(r'^index/',views.index,name="index")`
- 在模板中使用
 - `<a href="{% url 'app:index' arg1 arg2%}"` (传非关键字参数)

- `<a href="{% url 'app:index' name1=arg1 name2=arg2 %}"` (传关键字参数)
- 在视图中使用
 - `HttpResponseRedirect(reverse('view:sayhello',kwargs={}))`
kwargs是字典

Django类视图

- 原码视图类继承的view类源码分析



Django高级

中间件

概念：

是一个轻量级的，底层的插件，可以介入Django的请求和相应过程（面向切面编程）

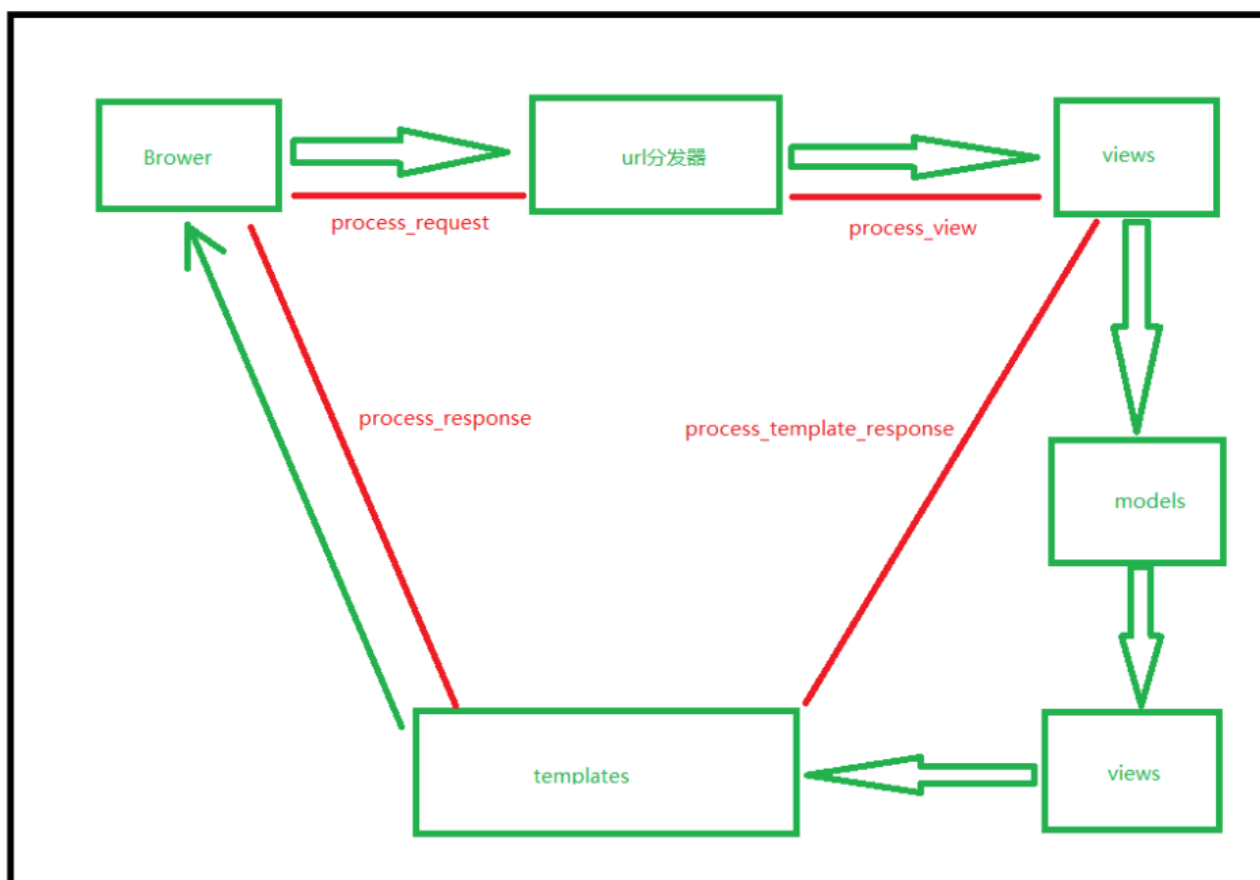
本质：

是一个python类

作用：

面向切面编程（Aspect Oriented Programming aop）对视图逻辑解耦合

中间件可切入点



切入函数说明

`__init__`:

没有参数，服务器响应第一个请求的时候自动调用，用户确定是否启用该中间件

`process_request(self,request)`:

在执行视图前被调用，每个请求上都会调用，不主动进行返回或返回 `HttpResponse` 对象

`process_view(self,request,view_func,view_args,view_kwargs)`:

调用视图之前执行，每个请求都会调用，不主动进行返回或返回 `HttpResponse` 对象

`process_template_response(self,request,response)`:

`process_template_response(self,request,response):`

在视图刚好执行完后进行调用，每个请求都会调用，不主动进行返回或返回HttpResponse对象

`process_response(self,request,response):`

所有响应返回浏览器之前调用，每个请求都会调用，不主动进行返回或返回HttpResponse对象

`process_exception(self,request,exception):`

当视图抛出异常时调用，不主动进行返回或返回HttpResponse对象

调用顺序

- 中间件注册的时候是一个列表
- 如果我们没有在切点处直接进行返回，中间件会依次执行
- 如果我们直接进行了返回，后续中间件就不再执行了

自定义中间件

- 在工程目录下创建middleware目录
- 目录创建一个python文件
- 在python总导入中间件基类

```
from django.utils.deprecation import MiddlewareMixin
```

- 在类中根据功能需求，创建切入需求类，重写切入点方法

```
class LearnAOP(MiddlewareMixin):
    def process_request(self,request):
        print(request.GET.path)
```

- 启用中间件，在settings中进行配置，MIDDLEWARE中添加middleware.文件名.类名

密码加密

```
make_password
check_password
```

图片、文件上传

- 文件数据存储在request.FILES属性中
- form表单上传文件需要添加enctype='multipart/form-data'，文件上传必须使用POST请求方式
- 存储

```
在static文件夹下创建uploadfiles用于存储接收上传的文件，在
settings中配置，
MEDIA_ROOT=os.path.join(BASE_DIR,r'static/uploadfiles')
```

- 在开发中通常是存储的时候，我们要存储到关联用户的表中

- 原生写法

```
<form method='post' action='xxx' enctype='multipart/form-  
data'>  
    {% csrf_token %}  
    <input type='file' name='icon'>  
    <input type='submit' value='上传'>  
</form>
```

```
def saveIcon(request):  
    if request.method == 'POST'  
        f = request.FILES['icon']  
        filePath = os.path.join(settings.MEDIA_ROOT,f.name)  
        with open(filePath,'wb') as fp:  
            for part in f.chunks():  
                fp.write(part)
```

分页

django分页

django提供了分页的工具，存在于django.core中

Django: 数据分页工具

Paginator:数据分页工具

Page: 具体某一页面

Paginator:

对象的创建: Paginator(数据集, 每一页数据数)

属性:

count: 对象总数

num_pages:页面总数

page_range: 页码列表, 从1开始

方法:

page (整数): 获得一个page对象

常见错误:

InvalidPage: page()传递无效页码

PageNotAnInteger: page()传递的不是整数

Empty: page()传递的值有效, 但是没有数据

Page:

对象获得: 通过Paginator的page()方法获得

属性:

object_list: 当前页面上所有数据对象

number: 当前的页码值

paginator: 当前page关联的Paginator对象

方法:

has_next(): 判断是否有下一页

has_previous():判断是否有上一页

has_other_pages():判断是否有上一页或下一页

next_page_number():返回下一页的页码

previous_page_number():返回上一页的页码

len(): 返回当前页的数据的个数

代码实现:

```
def get_students_with_page(request):
```

```
    page = int(request.GET.get("page", 1))
```

```
    per_page = int(request.GET.get("per_page", 10))
```

```

per_page = int(request.GET.get( per_page , 10))
students = Student.objects.all()
paginator = Paginator(students, per_page)
page_object = paginator.page(page)
data = {
    "page_object": page_object,
    'page_range': paginator.page_range
}
return render(request, 'students_with_page.html', context=data)

```

滚动加载

```

//为窗体添加滚动事件
$(window).scroll(function(){
    //bot是距离底部还有多少像素触发事件，一般设置为1
    var boot = 1;
    if((bot + $(window).scrollTop())>=
    ($(document).height()-$(window).height())){
        //发起ajax请求
        $.ajax({
            url:"http://127.0.0.1:8000/app/scroll/",
            type:"POST",
            data:{
                "num":num,

                "startPage":startPage
            }
        })
    }
})

```

```
        startPage :startPage,  
    },  
    dataType:"json",  
    success:function (data,textStatus) {  
        //创建元素、添加元素  
    }  
}  
});
```

加缓存

数据库缓存

- 目的
 - 提升服务器响应速度
 - 将执行过的操作数据 存储下来，在一定时间内，再次获取数据的时候，直接从缓存中获取
 - 比较理想的方案，缓存使用内存级缓存
 - Django内置缓存框架
- 实现步骤
 - 执行python manage.py createcachetable my_cache_table
 - 缓存配置

```
CACHES = {
    'default':{

'BACKEND':'django.core.cache.backends.db.DatabaseCa
che',
        'LOCATION':'my_cache_table',
        'TIMEOUT':'60'
    }
}
```

- 缓存使用
 - 在视图中使用（使用最多的场景）
 - @cache_page()
 - time 秒 60*5缓存5分钟
 - cache 缓存配置，默认default
 - key_prefix前置字符串
- 缓存底层
 - 配有多个缓存的获取

```
from django.core.cache import caches
cache = caches['cache_name']
```

- 只有一个缓存的获取

```
from django.core.cache import cache
```

- 缓存操作

- `cache.set`
 - `key`
 - `value`
 - `timeout`
- `get`
- `add`
- `get_or_set`
- `get_many`
- `set_many`
- `delete`
- `delete_many`
- `clear`
- `incr` 增加
 - `incr(key, value)` key对应的值上添加 `value`
- `decr` 减少
 - `decr(key, value)` key对应的值上减少 `value`
 - 如果 `value` 不写, 默认变更为1

使用Redis做缓存

- 安装三方模块
 - `pip install django-redis`
 - `pip install django-redis-cache`

- 配置和内置数据库缓存配置基本一致

```
CACHES = {
    "default": {
        "BACKEND": "django_redis.cache.RedisCache",
        "LOCATION": "redis://127.0.0.1:6379/1",
        "OPTIONS": {
            "CLIENT_CLASS": "django_redis.client.DefaultClient",
            "PASSWORD": "fanding",
        }
    }
}
```

- 用法和内置缓存使用一样

验证码

视图函数代码：

```
def get_code(request):

    # 初始化画布，初始化画笔

    mode = "RGB"

    size = (200, 100)

    red = get_color()
```

```
    green = get_color()
```

```
green = get_color()

blue = get_color()

color_bg = (red, green, blue)

image = Image.new(mode=mode, size=size, color=color_bg)

imagedraw = ImageDraw(image, mode=mode)

imagefont = ImageFont.truetype(settings.FONT_PATH, 100)

verify_code = generate_code()

request.session['verify_code'] = verify_code

for i in range(4):
    fill = (get_color(), get_color(), get_color())
    imagedraw.text(xy=(50*i, 0), text=verify_code[i], font=imagefont,
fill=fill)

for i in range(10000):
    fill = (get_color(), get_color(), get_color())
    xy = (random.randrange(201), random.randrange(100))
    imagedraw.point(xy=xy, fill=fill)

fp = BytesIO()

image.save(fp, "png")

return HttpResponse(fp.getvalue(), content_type="image/png")
```

JS代码：

```
$("#img").click(function () {  
  
    console.log("点到我了");  
  
    $(this).attr("src", "/app/getcode/?t=" + Math.random());  
  
})
```

富文本

富文本：Rich Text Format（RTF），是有微软开发的跨平台文档格式，大多数的文字处理软件都能读取和保存RTF文档，其实就是可以添加样式的文档，和HTML有很多相似的地方

两种插件：

tinymce 插件

django的插件

pip install django-tinymce

用处：

1. 在后台管理中使用
2. 在页面中使用，通常用来作博客

后台使用：

配置settings.py文件

INSTALLED_APPS 添加 tinymce 应用

添加默认配置

TINYMCE_DEFAULT_CONFIG = {

```
TINYMCE_DEFAULT_CONFIG = {  
    'theme':'advanced',  
    'width':800,  
    'height':600,  
}
```

创建模型类：

```
from tinymce.models import HTMLField  
class Blog(models.Model):  
    sBlog = HTMLField()
```

配置站点：

```
admin.site.register
```

在视图中使用：

使用文本域盛放内容

```
<form method='post' action='url'>  
    <textarea></textarea>  
</form>
```

在head中添加script

```
<script src='/static/tiny_mce/tiny_mce.js'></script>  
<script>  
    tinyMCE.init({  
        'mode':'textareas', 'theme':'advanced',  
        'width':800,'height':600,  
    })  
</script>
```

免除csrf_token验证

- 加装饰器@csrf_exempt

跨域问题解决

为response添加如下属性即可实现

```
response["Access-Control-Allow-Origin"] = "*"
```

```
response["Access-Control-Allow-Methods"] = "POST, GET, OPTIONS"
```

```
response["Access-Control-Max-Age"] = "1000"
```

```
response["Access-Control-Allow-Headers"] = "*"
```

最好写成中间件

```
重写process_response(self,request,response):
```

需求，统计用户

- 自己统计
 - 通过中间件直接实现
- 使用专用统计分析工具
 - 百度统计
 - 注册、添加一段js代码即可
 - 极光统计
 - 友盟统计
- 具体看官方文档

发送邮件

- 配置文件配置字段

```
#SEND_EMAIL
EMAIL_HOST = "smtp.163.com"
EMAIL_PORT = 465
EMAIL_HOST_USER = "fand1024@163.com"
EMAIL_HOST_PASSWORD = "fand102487"
EMAIL_USE_SSL = True
```

Mail is sent using the SMTP host and port specified in the `EMAIL_HOST` and `EMAIL_PORT` settings. The `EMAIL_HOST_USER` and `EMAIL_HOST_PASSWORD` settings, if set, are used to authenticate to the SMTP server, and the `EMAIL_USE_TLS` and `EMAIL_USE_SSL` settings control whether a secure connection is used.

- 视图函数实现逻辑

```
from django.core.mail import send_mail
```

```
send_mail(
    'Subject here',
    'Here is the message.',
    'from@example.com',
    ['to@example.com'],
    fail_silently=False,
)
```

例子：

```
send_mail(subject="FAND用户激活",
from_email=EMAIL_HOST_USER, recipient_list=
["2094531487@qq.com", ], message="nihao",
        html_message="<a href='www.baidu.com'>激活</a>")
```

html_message和message同时写，但前者覆盖了后者

num_message和message同时与，但前者覆盖了后者

异步消息处理

- 消息队列
 - 异步任务
 - 定时任务
- 需要了解的知识
 - 选择并安装消息容器（载体）
 - redis
 - 安装Celery并创建第一个任务
 - 开启工作进程并调用任务
 - 记录工作状态和返回的结果

整体流程：

通用：

1、pip install -U celery[redis] 和pip install celery

2、创建task.py文件

```
from celery import Celery
```

```
app = Celery('tasks', broker='pyamqp://guest@localhost//')
```

```
@app.task
```

```
def add(x, y):
```

```
    return x + y
```

```
return x + y
```

3、启动celery

```
celery -A tasks worker --loglevel=info
```

django中使用：

1、在工程与setting同级目录下创建celery.py文件

```
import os
```

```
from celery import Celery
```

```
os.environ.setdefault('DJANGO_SETTINGS_MODULE',  
'rest_school.settings') # 将最后一个参数修改称为自己项目
```

```
app = Celery('rest_school') #修改成为自己项目
```

```
app.config_from_object('django.conf:settings',  
namespace='CELERY')
```

```
app.autodiscover_tasks()
```

```
@app.task(bind=True)
```

```
def debug_task(self):
```

```
    print('Request: {0!r}'.format(self.request))
```

2、修改同setting同一级文件__init__文件，添加代码

```
from .celery import app as celery_app
```

```
__all__ = ['celery_app']
```

3、setting文件添加内容：

添加： `pip install django-celery-results` 如下图

安装： pip install django-celery-results:如下图所示

CELERY_BROKER_URL =

'redis://fanding@www.fand.wang:6379/1'

CELERY_TASK_SERIALIZER = 'json'

CELERY_ACCEPT_CONTENT = ['json']

CELERY_RESULT_BACKEND = 'django-db'

4、注册result结果管理模块

在setting中的注册app的地方注册

'django_celery_results',

5、执行迁移： python manage.py migrate

django_celery_results

生成对应的数据库表

6、在app中添加task.py文件，编写异步执行函数

用@shared_task装饰函数

调用时如： add.delay()

注意： window10系统下还不能使用：

报错： Celery ValueError: not enough values to unpack
(expected 3, got 0)的 解决方案：

安装： pip install eventlet

启动worker时候： celery -A <mymodule> worker -l info -P
eventlet

多加-P eventlet参数

a result backend

The [django-celery-results](#) extension provides result backends using either the Django ORM, or the Django Cache framework.

To use this with your project you need to follow these steps:

1. Install the [django-celery-results](#) library:

```
$ pip install django-celery-results
```

2. Add `django_celery_results` to `INSTALLED_APPS` in your Django project's `settings.py`:

```
INSTALLED_APPS = (  
    ...,  
    'django_celery_results',  
)
```

Note that there is no dash in the module name, only underscores.

3. Create the Celery database tables by performing a database migrations:

```
$ python manage.py migrate django_celery_results
```

4. Configure Celery to use the [django-celery-results](#) backend.

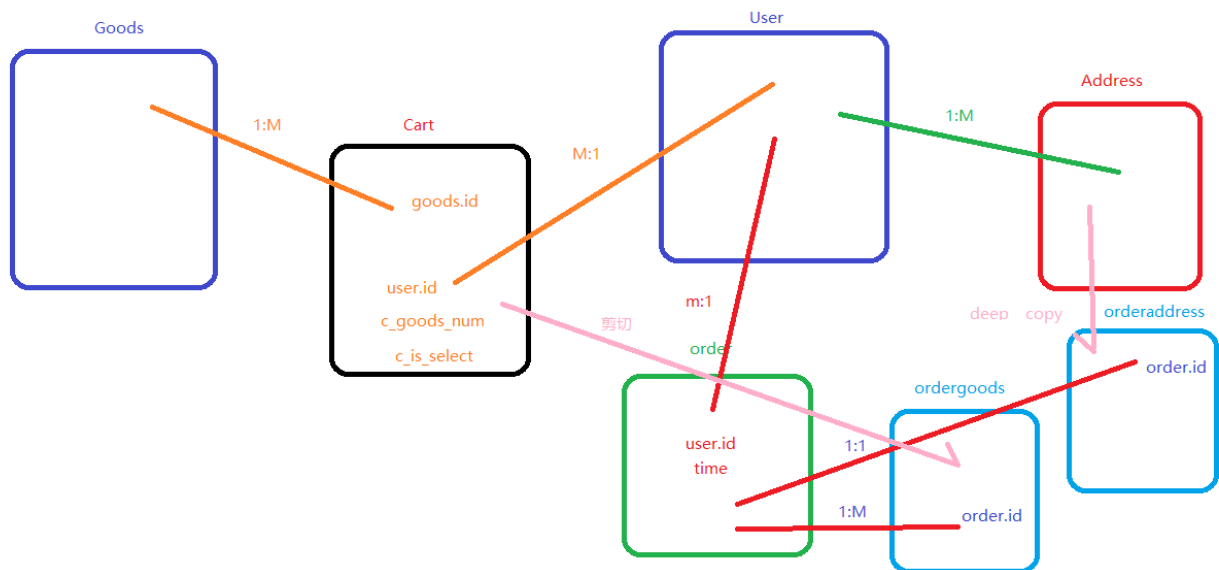
Assuming you are using Django's `settings.py` to also configure Celery, add the following settings:

```
CELERY_RESULT_BACKEND = 'django-db'
```

For the cache backend you can use:

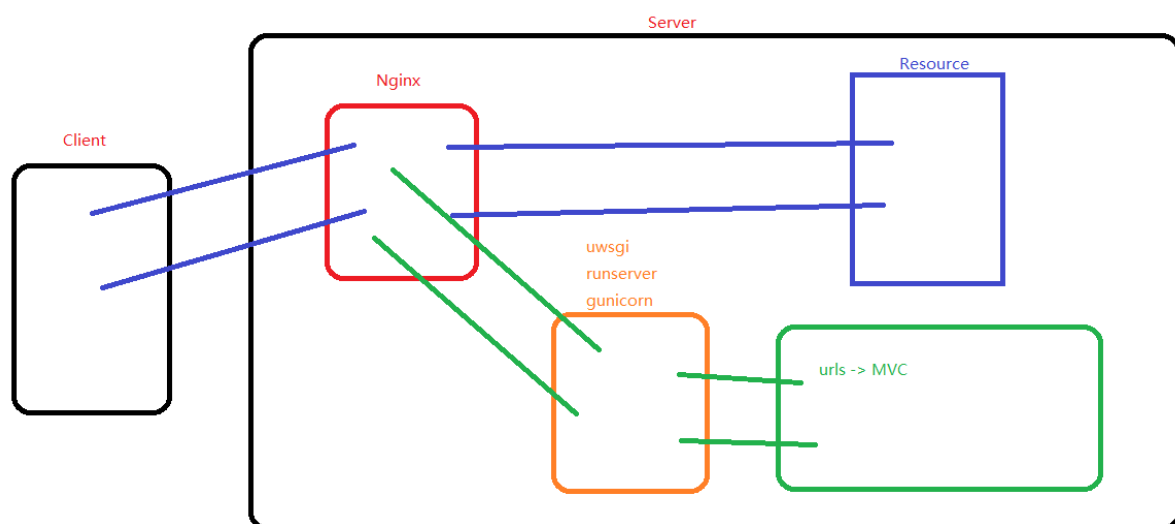
```
CELERY_RESULT_BACKEND = 'django-cache'
```

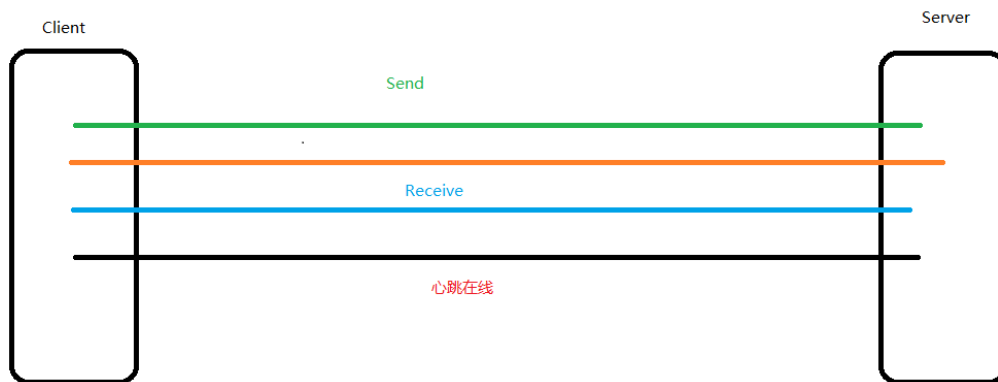
项目经验



项目部署

概念理解





Nginx安装

源码安装

1. 下载源码压缩包
2. 安装源码编译依赖包 gcc,zlib,make...
3. 配置编译模块
4. make && make test
5. make install

包管理工具安装

1. 去官网将所使用依赖添加到包管理工具中
2. 更新包管理工具资源
3. 使用包管理工具安装

阅读官方文档

官网

<http://nginx.org/>

中文资料

<http://www.nginx.cn/doc/index.html>

<http://tengine.taobao.org/book/>

<http://tengine.taobao.org/book/>

1、/etc/apt/sources.list在此文件中添加如下两行：（ubuntu16.04版本，对应修改codename（xenial））

```
deb http://nginx.org/packages/ubuntu/ xenial nginx
```

```
deb-src http://nginx.org/packages/ubuntu/ xenial nginx
```

2、下载钥匙并添加钥匙

```
sudo apt-key add nginx_signing.key
```

3、输入命令

```
apt-get update
```

```
apt-get install nginx
```

启动Nginx

```
nginx [-c configpath]
```

信息查看

```
nginx -v
```

```
nginx -V
```

控制Nginx

```
nginx -s signal
```

```
stop    快速关闭
```

```
quit    优雅的关闭
```

```
reload  重新加载配置
```

通过系统管理

```
systemctl status nginx 查看nginx状态
```

```
systemctl start  nginx 启动nginx服务
```

```
systemctl stop   nginx    关闭nginx服务
```

```
systemctl enable nginx 设置开机自启
```

```
systemctl disable nginx 禁止开机自启
```

配置文件启动

```
nginx -t    不运行，仅测试配置文件
```

```
nginx -c configpath 从指定路径加载配置文件
```

```
nginx -t -c configpath 测试指定配置文件
```

```
nginx -t -c configpath 测试指定配置文件
```

配置文件语法

1、总体说明

main 全局设置

events{ 工作模式, 连接配置

...

}

http{ http的配置

...

upstream xxx{ 负载均衡配置

...

}

server{ 主机设置

...

location xxx{ URL匹配

...

}

}

}

2、main

user nginx; worker进程运行的用户和组

worker_processes 1; 指定Nginx开启的子进程数, 多核CPU建议设置和CPU数量一样的进程数

`error_log xxx level;` 用来定义全局错误日志文件，通常放在var中，
level有 debug, info, notice, warn, error, crit

`pid xxx;` 指定进程id的存储文件位置

3、events

指定工作模式和以及连接上限

```
events{  
    use epoll;  
    worker_connections 1024;  
}
```

use 指定nginx工作模式

epoll 高效工作模式，linux

kqueue 高效工作模式，bsd

poll 标准模式

select 标准模式

`worker_connections` 定义nginx每个进程的最大连接数

正向代理 连接数 * 进程数

反向代理 连接数 * 进程数 / 4

linux系统限制最多能同时打开65535个文件，默认上限就是65535，可解除 `ulimit -n 65535`

4、http

最核心的模块，主要负责http服务器相关配置，包含server，upstream子模块

`include mime.types;`设置文件的mime类型

`include xxx.conf;` 包含其它配置文件 公开规划解耦

`include xxxconf; 包含其它配置文件，为开规划解耦`

`default_type xxx;` 设置默认类型为二进制流，文件类型未知时就会使用默认

`log_format` 设置日志格式

`sendfile` 设置高效文件传输模式

`keepalive_timeout` 设置客户端连接活跃超时

`gzip` `gzip`压缩

5、server

用来指定虚拟主机

`listen 80;` 指定虚拟主机监听的端口

`server_name localhost;` 指定ip地址或域名，多个域名使用空格隔开

`charset utf-8;` 指定网页的默认编码格式

`error_page 500 502 /50x.html` 指定错误页面

`access_log xxx main;` 指定虚拟主机的访问日志存放路径

`error_log xxx main;` 指定虚拟主机的错误日志存放路径

`root xxx;` 指定这个虚拟主机的根目录

`index xxx;` 指定默认首页

6、location

6、location

核心中的核心，以后的主要配置都在这

主要功能:定位url，解析url，支持正则匹配，还能支持条件，实现动静分离

语法

```
location [modifier] uri{  
    ...  
}
```

modifier 修饰符

- = 使用精确匹配并且终止搜索
- ~ 区分大小写的正则表达式
- ~* 不区分大小写的正则表达式
- ^~ 最佳匹配，不是正则匹配，通常用来匹配目录

常用指令

alias 别名，定义location的其他名字，在文件系统中能够找到，如果location指定了正则表达式，alias将会引用正则表达式中的捕获，alias替代location中匹配的部分，没有匹配的部分将会在文件系统中搜索

负载均衡配置

1、项目部署完整流程：

django 服务器

runserver

runserver

wsgi

uwsgi : web服务器,多线程处理的不错

1. pip install uwsgi
2. 工程目录下创建uwsgi.ini 配置文件
3. 书写配置信息
4. 使用uwsgi服务器
 - 启动 uwsgi --ini uwsgi.ini
 - 停止 uwsgi --stop uwsgi.pid

nginx配置

```
location /static{  
    alias xxx/static/  
}  
location / {  
    include uwsgi_params;  
    uwsgi_pass localhost:8000;  
}
```

2、反向代理:

proxy_pass URL; 反向代理转发地址, 默认不转发header, 需要转发header则设置

proxy_set_header HOST \$host;

proxy_method POST; 转发的方法名

proxy_hide_header Cache-Control; 指定头部不被转发

proxy_pass_header Cache-Control; 设置哪些头部转发

proxy_pass_request_header on; 设置转发http请求头

proxy_pass_request_body on; 设置转发请求体

3、upstream负载均衡配置：

负载均衡模块，通过一个简单的调度算法来实现客户ip到后端服务器的负载均衡

写法 upstream myproject{

```
    ip_hash;
    server 127.0.0.1:8000;
    server 127.0.0.1:8001 down;
    server 127.0.0.1:8002 weight=3;
    server 127.0.0.1:8003 backup;
    fair;
}
```

负载均衡算法

weight 负载权重

down 当前server不参与负载均衡

backup 其它机器全down掉或满载使用此服务

ip_hash 按每个请求的hash结果分配

fair 按后端响应时间来分（第三方的）

自己做服务器配置文件

```
user nginx;
```

```
worker_processes 1;
```

```
error_log /var/log/nginx/error.log warn;
```

```
error_log /var/log/nginx/error.log warn;

pid /var/run/nginx.pid;


events {
    worker_connections 1024;
}


http {
    include /etc/nginx/mime.types;
    default_type application/octet-stream;

    log_format main '$remote_addr - $remote_user [$time_local]
"$request" '
        '$status $body_bytes_sent "$http_referer" '
        '"$http_user_agent" "$http_x_forwarded_for"';

    access_log /var/log/nginx/access.log main;

    sendfile on;
    #tcp_nopush on;

    keepalive_timeout 65;

    #gzip on;

    server {
        listen 80;
        server_name localhost;

        root /home/rock/GP1/Day10/GPAXF;
```

```
location /static {  
    alias /home/rock/GP1/Day10/GPAXF/static;  
}  
  
location / {  
    include /etc/nginx/uwsgi_params;  
    uwsgi_pass 127.0.0.1:8888;  
}  
  
}  
  
}
```

做负载均衡配置文件

```
user nginx;  
worker_processes 1;  
  
error_log /var/log/nginx/error.log warn;  
pid /var/run/nginx.pid;  
  
events {  
    worker_connections 1024;  
}
```

```
http {
    include    /etc/nginx/mime.types;
    default_type application/octet-stream;

    log_format main '$remote_addr - $remote_user [$time_local]
"$request" '
        '$status $body_bytes_sent "$http_referer" '
        '"$http_user_agent" "$http_x_forwarded_for"';

    access_log /var/log/nginx/access.log main;

    sendfile    on;
    #tcp_nopush  on;

    keepalive_timeout 65;

    #gzip on;
    upstream myproject{
        server www.donogh.cn;
    }

    server {
        listen    80;
        server_name www.fand.wang;

        location / {
            proxy_pass http://myproject;
        }

    }
}
```

```
}
```

uwsgi安装

- 进入虚拟环境直接pip install uwsgi
- 注意（创建虚拟环境的时候需要指定python解释器）否则会出现错误-
- 具体可以到uwsgi官网上看
- 配置文件

```
[uwsgi]
# 使用nginx连接时 使用
socket=0.0.0.0:8888

# 直接作为web服务器使用，测试
# http=0.0.0.0:8888
# 配置工程目录
chdir=/home/rock/GP1/Day10/GPAXF

# 配置项目的wsgi目录。相对于工程目录
wsgi-file=GPAXF/wsgi.py

#配置进程，线程信息
processes=4

threads=10
```

```
enable-threads=True
```

```
master=True
```

```
pidfile=uwsgi.pid
```

```
daemonize=uwsgi.log
```

RESTful

什么是REST

一种软件架构风格、设计风格、而不是标准，只是提供了一组设计原则和约束条件。它主要用户客户端和服务端交互类的软件。基于这个风格设计的软件可以更简洁，更有层次，更易于实现缓存机制等。

REST全程是Representational State Transfer，表征性状态转移。首次在2000年Roy Thomas Fielding的博士论文中出现，Fielding是一个非常重要的人，他是HTTP协议（1.0版和1.1版）的主要设计者，Apache服务器软件的作者之一，Apache基金会的第一任主席。所以，他的这篇论文一经发表，就引起了广泛的关注。

论文：

本文研究计算机科学两大前沿----软件和网络----的交叉点。长期以来，软件研究主要关注软件设计的分类、设计方法的演化，很少客观地评估不同的设计选择对系统行为的影响。而相反地，网络研究主要关注系统之间通信行为的细节、如何改进特定通信机制的表现，常常忽视了一个事实，那就是改变应用程序的互动风格比改变互动协议，对整体表现有更大的影响。我这篇文

理解RESTful

要理解RESTful架构，最好的就是去理解它的单词 Representational State Transfer 到底是什么意思，它的每一个词到底要表达什么。

REST的释义，“表现层状态转化”，其实这省略了主语。“表现层”其实指的是“资源(Resource)”的“表现层”。

资源 (Resource)

所谓“资源”就是网络上的一个实体，或者说是网络上的一个具体信息

所谓“资源”，就是网络上的一个实体，或者说就是网络上的一个具体信息。

它可以是一段文本，一张图片，一首歌曲，一种服务，总之就是一个具体的实例。你可以使用一个URI（统一资源定位符）指向它，每种资源对应一个特定的URI。要获取这个资源，访问它的URI就可以了，因此URI就成了每一个资源的地址或独一无二的识别符。所谓“上网”就是与互联网上一系列的“资源”互动，调用它们的URI。

表现层（Representation）

“资源”是一种信息实体，它可以有多种外在表现形式。我们把“资源”具体呈现出来的形式，叫做它的“表现层”（Representation）。

URI只代表资源的实体，不代表它的形式。严格地说，有些网站最后的“.html”后缀名是不必要的，因为这个后缀表示格式，属于“表现层”范畴，而URI应该只代表“资源”的位置。它的具体表现形式，应该在HTTP请求头的信息中使用Accept和Content-Type字段指定。

状态转换（State Transfer）

访问一个网站，就代表客户端和服务端的一个互动过程。在这个过程中，势必涉及到数据和状态的变化。

互联网通信协议HTTP协议，是一个无状态协议。这意味着，所有的状态都保存在服务端。因此，如果客户端想要操作服务器，就必须通过某种手段，让服务器端发生“状态转换（State Transfer）”。而这种转换是建立在表现层之上的，所以就是“表现层状态转化”。

客户端用到的手段，只能是HTTP协议。具体来说，就是HTTP协议中，四个表示操作方式的动词：GET，POST，PUT，DELETE。它们分别对应四种基本操作：GET用来获取资源，POST用来新建资源（也可用于更新资源），PUT用来更新资源，DELETE用来删除资源

到底什么是RESTful架构

每一个URI代表一种资源

客户端和服务端之间，传递这种资源的某种表现形式

各客户端和服务端之间，传递这种资源的某种表现层

客户端通过四个HTTP动词，对服务端资源进行操作，实现“表现层状态转换”

RESTful API 设计

协议

API与用户的通信协议，通常使用HTTP(S)协议。

域名

应该尽量将API部署在专用域名之下。

```
http://api.rock.com
```

如果确定API很简单，不会有大规模扩充，可以考虑放在主域名之下。

```
http://www.rock.com/api/
```

版本

应该将API的版本号放入URL。

```
http://api.rock.com/v1/
```

也有做法是将版本号放在HTTP的头信息中，但不如放在URL中方便和直观。GITHUB是这么搞的。

路径 (Endpoint)

路径又称“终点” (endpoint)，表示API的具体网址。

在RESTful架构中，每个网址代表一种资源 (Resource)，所以网址中不能有动词，只能有名词，而且所用的名词往往与数据库的表格名对应。一般来说，数据库中的表都是同种记录的“集合” (collection)，所以API中的名词也应该使用复数。

HTTP动词

对于资源的具体操作类型，由HTTP动词表示。

HTTP常用动词

- GET (SELECT)：从服务器取出资源
- POST (CREATE or UPDATE)：在服务器创建资源或更新资源
- PUT (UPDATE)：在服务器更新资源（客户端提供改变后的完整资源）
- PATCH (UPDATE)：在服务器更新资源（客户端提供改变的属性）
- DELETE (DELETE)：从服务器删除资源
- HEAD：获取资源的元数据
- OPTIONS：获取信息，关于资源的哪些属性是客户端可以改变的、

示例

- GET /students：获取所有学生

- POST /students: 新建学生
- GET /students/id: 获取某一个学生
- PUT /students/id: 更新某个学生的信息 (需要提供学生的全部信息)
- PATCH /students/id: 更新某个学生的信息 (需要提供学生变更部分信息)
- DELETE /students/id: 删除某个学生

过滤信息 (Filtering)

如果记录数量过多, 服务器不可能将它们返回给用户。API应该提供参数, 过滤返回结果。

- ?limit=10
- ?offset=10
- ?page=2&per_page=20
- ?sortby=name&order=desc
- ?student_id=id

参数的设计允许存在冗余, 即允许API路径和URL参数偶尔有重复, 比如 GET /students/id 和 ? student_id=id

状态码

服务器向用户返回的状态码和提示信息, 常见的有以下一些地方

- 200 OK - [GET]: 服务器成功返回用户请求的数据
- 201 CREATED - [POST/PUT/PATCH]: 用户新建或修改数据成功
- 202 Accepted - [*]: 表示一个请求已经进入后台排队 (异步任务)
- 204 NO CONTENT - [DELETE]: 表示数据删除成功
- 400 INVALID REQUEST - [POST/PUT/PATCH]: 用户发出的请求有错误

- 401 Unauthorized - [*]：表示用户没有权限（令牌，用户名，密码错误）
- 403 Forbidden - [*]：表示用户得到授权，但是访问是被禁止的
- 404 NOT FOUND - [*]：用户发出的请求针对的是不存在的记录
- 406 Not Acceptable - [*]：用户请求格式不可得
- 410 Gone - [GET]：用户请求的资源被永久移除，且不会再得到的
- 422 Unprocesable entity -[POST/PUT/PATCH]：当创建一个对象时，发生一个验证错误
- 500 INTERNAL SERVER EROR - [*]：服务器内部发生错误

错误处理

如果状态码是4xx，就应该向用户返回出错信息。一般来说，返回的信息中将error做为键名

返回结果

针对不同操作，服务器想用户返回的结果应该符合以下规范

- GET /collection：返回资源对象的列表（数组，集合）
- GET /collection/id：返回单个资源对象
- POST /collection：返回新生成的资源对象
- PUT /collection/id：返回完整的资源对象
- PATCH /collection/id：返回完整的资源对象
- DELETE /collection/id：返回一个空文档

使用链接关联资源

RESTful API最好做到Hypermedia，即返回结果中提供链接，连向其他API方法，使得用户不查文档，也知道下一步应该做什么。

```
{
  "link": {
    "rel": "collection https://www.rock.com/zoostudents",
    "href": "https://api.rock.com/students",
    "title": "List of students",
    "type": "application/vnd.yourformat+json"
  }
}
```

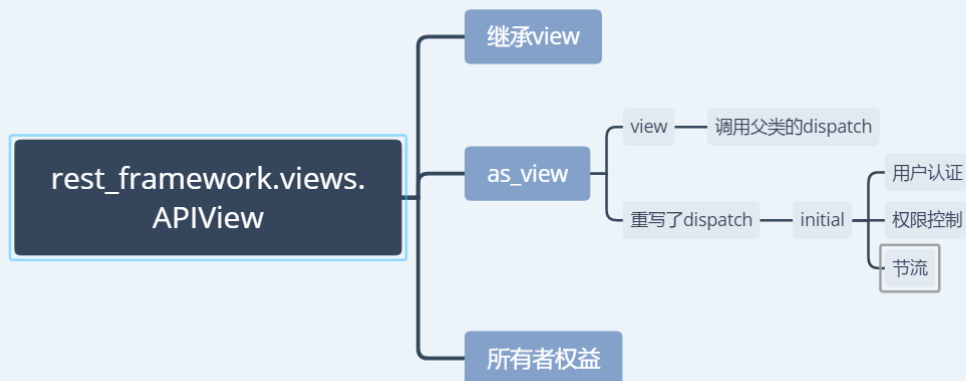
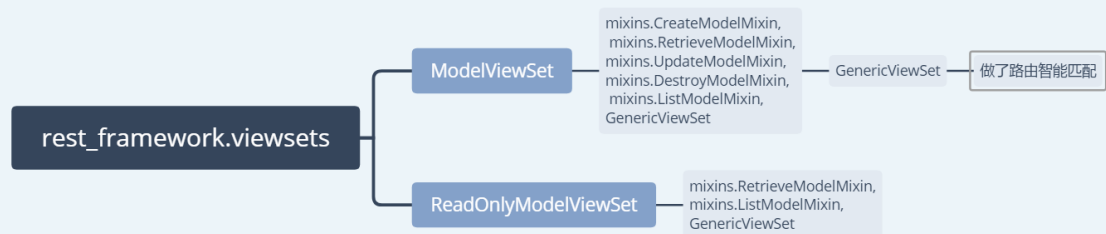
- rel: 表示这个API与当前网址的关系
- href: 表示API的路径
- title: 表示API的标题
- type: 表示返回的类型

其它

- 服务器返回的数据格式，应该尽量使用JSON
- API的身份认证应该使用OAuth2.0框架

Django-rest-framework

源码剖析



模型序列化

- django-rest-framework
- REST难点
 - 模型序列化
 - 正向序列化
 - 将模型转换成JSON
 - 反向序列化
 - 将JSON转换成模型
 - serialization
 - 在模块serializers

- HyperLinkedModelSerializer
 - 序列化模型，并添加超链接
- Serializer
 - 手动序列化
- ModelSerializer
 - 类似于
HyperLinkedModelSerializer
- 级联序列化

-

```
from rest_framework import
serializers

from App.models import UserModel,
Address

class
AddressSerializer(serializers.Hyperlin
kedModelSerializer):

    class Meta:
        model = Address
        fields = ('url', 'id', 'a_address')

class
```

```

class
UserSerializer(serializers.Hyperlinked
ModelSerializer):

    address_list =
AddressSerializer(many=True,
read_only=True)

    class Meta:
        model = UserModel
        fields = ('url', 'id', 'u_name',
'u_password', 'address_list')

```

注意：可以通过模型级联时修改隐性属性的名字如下：

```

a_user =
models.ForeignKey(UserModel,
related_name='address_list', null=True,
blank=True)

```

双R

- Request
 - rest_framework.request
 - 将Django中的Request作为了自己的一个属性_request
 - 属性和方法
 - content_type

- stream
- query_params
- data
 - 同时兼容 POST,PUT,PATCH
- user
 - 可以直接在请求上获取用户
 - 相当于在请求上添加一个属性，用户对象
- auth
 - 认证
 - 相当于请求上添加了一个属性，属性值是token
- successful_authenticator
 - 认证成功
- Response
 - 依然是HttpResponse的子类
 - 自己封装的
 - data 直接接受字典转换成JSON
 - status 状态码
 - 属性和方法
 - rendered_content
 - status_text

- **APIView**

- `renderer_classes`
 - 渲染的类
- `parser_classes`
 - 解析转换的类
- `authentication_classes`
 - 认证的类
- `throttle_classes`
 - 节流的类
 - 控制请求频率的
- `permission_classes`
 - 权限的类
- `content_negotiation_class`
 - 内容过滤类
- `metadata_class`
 - 元信息的类
- `versioning_class`

- 版本控制的类
- as_view()
 - 调用父类中的as_view -> dispatch
 - dispatch被重写
 - initialize_request
 - 使用django的request构建了一个REST中的Request
 - initial
 - perform_authentication
 - 执行用户认证
 - 遍历我们的认证器
 - 如果认证成功会返回一个元组
 - 元组中的第一个元素就是user
 - 第二个元素就是auth, token
 - check_permissions
 - 检查权限

- 遍历我们的权限检测器

- 只要有一个权限检测没通过
- 就直接显示权限被拒绝
- 所有权限都满足, 才算是拥有权限

- `check_throttles`

- 检测频率
- 遍历频率限制器
- 如果验证不通过, 就需要等待

- `csrf_exempt`

- 所有APIView的子类都是csrf豁免的

- 错误码

- 封装 `status` 模块中
- 实际上就是一个常量类

- 针对视图函数的包装

- CBV
 - APIView
- FBV
 - 添加 @api_view装饰器
 - 必须手动指定允许的请求方法

APIView子类

- 子类
 - generics包中
 - GenericAPIView
 - 增加的模型的获取操作
 - get_queryset
 - get_object
 - lookup_field 默认pk
 - get_serializer
 - get_serializer_class
 - get_serializer_context
 - filter_queryset
 - paginator
 - paginate_queryset

- get_paginated_response
- CreateAPIView
 - 创建的类视图
 - 继承自GenericAPIView
 - 继承自CreateModelMixin
 - 实现了post进行创建
- ListAPIView
 - 列表的类视图
 - 继承自GenericAPIView
 - 继承自ListModelMixin
 - 实现了get
- RetrieveAPIView
 - 查询单个数据的类视图
 - 继承自GenericAPIView
 - 继承自RetrieveModelMixin
 - 实现了get
- DestroyAPIView
 - 销毁数据的类视图，删除数据的类视图
 - 继承自GenericAPIView
 - 继承自DestroyModelMixin
 - 实现了delete
- UpdateAPIView
 - 更新数据的类视图
 - 继承自GenericAPIView
 - 继承自UpdateModelMixin

- 实现了 put,patch
- ListCreateAPIView
 - 获取列表数据，创建数据的类视图
 - 继承自GenericAPIView
 - 继承自ListModelMixin
 - 继承自CreateModelMixin
 - 实现了 get,post
- RetrieveUpdateAPIView
 - 获取单个数据，更新单个数据的类视图
 - 继承自GenericAPIView
 - 继承自RetrieveModelMixin
 - 继承自UpdateModelMixin
 - 实现了 get, put, patch
- RetrieveDestroyAPIView
 - 获取单个数据，删除单个数据
 - 继承自GenericAPIView
 - 继承自RetrieveModelMixin
 - 继承自DestroyModelMixin
 - 实现了 get, delete
- RetrieveUpdateDestroyAPIView
 - 获取单个数据，更新单个数据，删除单个数据的类视图
 - 继承自GenericAPIView
 - 继承自RetrieveModelMixin
 - 继承自UpdateModelMixin
 - 继承自DestroyModelMixin
 - 实现了 get, put, patch, delete

- mixins
 - CreateModelMixin
 - create
 - perform_create
 - get_success_headers
 - ListModelMixin
 - list
 - 查询结果集，添加分页，帮你序列化
 - RetrieveModelMixin
 - retrieve
 - 获取单个对象并进行序列化
 - DestroyModelMixin
 - destroy
 - 获取单个对象
 - 调用执行删除
 - 返回Respon 状态码204
 - perform_destroy
 - 默认是模型的delete
 - 如果说数据的逻辑删除
 - 重写进行保存
 - UpdateModelMixin
 - update
 - 获取对象，合法验证

- 执行更新
- perform_update
 - partial_update
 - 增量更新，对应的就是patch
- viewsets
 - ViewSetMixin
 - 重写as_view
 - GenericViewSet
 - 继承自GenericAPIView
 - 继承自ViewSetMixin
 - ViewSet
 - 继承自APIView
 - 继承自ViewSetMixin
 - 默认啥都不支持，需要自己手动实现
 - ReadOnlyModelViewSet
 - 只读的模型的视图集合
 - 继承自RetrieveModelMixin
 - 继承自ListModelMixin
 - 继承自GenericViewSet
 - ModelViewSet
 - 直接封装对象的所有操作
 - 继承自GenericViewSet
 - 继承自CreateModelMixin

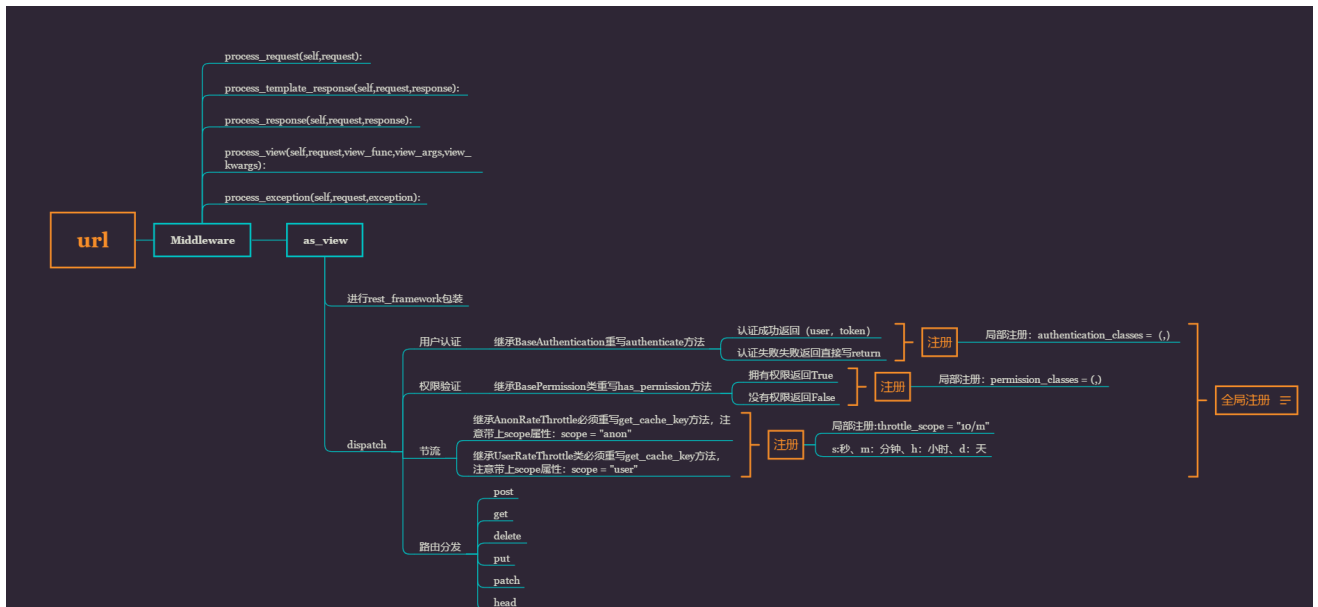
- 继承自RetrieveModelMixin
- 继承自UpdateModelMixin
- 继承自DestroyModelMixin
- 继承自ListModelMixin

用户模块

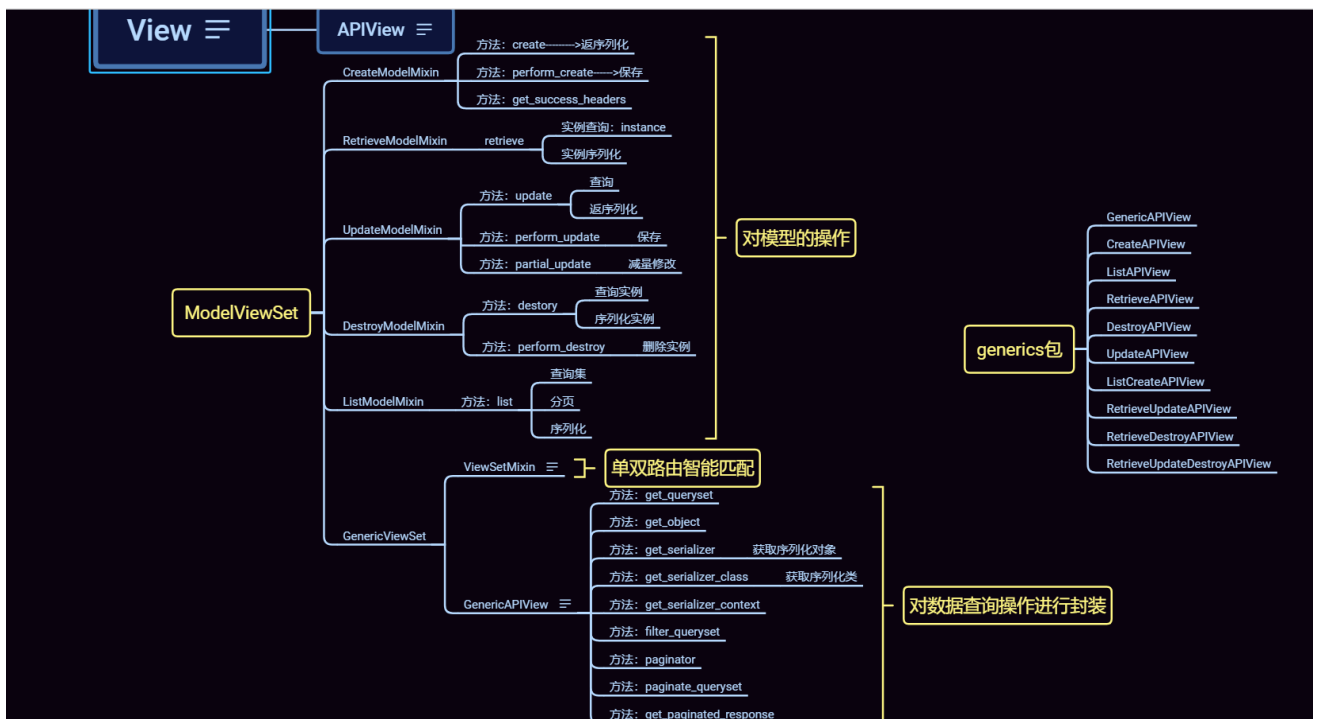
- 用户注册
 - RESTful
 - 数据开始
 - 模型，数据库
 - 创建用户
 - 用户身份
 - 管理员
 - 普通
 - 删除用户
 - 注册实现
 - 添加了超级管理员生成
- 用户登陆
 - 验证用户名密码
 - 生成用户令牌
 - 出现和注册公用post冲突
 - 添加action
 - path/?action=login

- path/?action=register
- 异常捕获尽量精确
- 用户认证
 - BaseAuthentication
 - authenticate
 - 认证成功会返回 一个元组
 - 第一个元素是user
 - 第二元素是令牌 token, auth
- 用户权限
 - BasePermission
 - has_permission
 - 是否具有权限
 - true拥有权限
 - false未拥有权限
- 用户认证和权限
 - 直接配置在视图函数上就ok了

DjangoRest_framework请求响应过程



类梳理



节流器

控制频率，访问次数

1、setting中设置REST_FRAMEWORK:

```

REST_FRAMEWORK = {
    "DEFAULT_THROTTLE_CLASSES": (
        # 此处是设置的throttle的路径
        "RESTEnd.throttles.ChildThrottle",
    ),
    "DEFAULT_THROTTLE_RATES": {

        # 此处的child对应的是scope的值
        'child': "5/m"
    }
}

```

2、定义一个throttle类，继承自Simple Throttle。

3、重写get_cache_key方法

```

class ChildThrottle(SimpleRateThrottle):

    scope = 'child'          #child

    def get_cache_key(self, request, view):
        if isinstance(request.user, ChildModel):
            ident = request.auth
        else:
            ident = self.get_ident(request)

        return self.cache_format % {
            'scope': self.scope,
            'ident': ident
        }

```

```
}
```

Leetcode

- 心情好就去刷几道题
- 心情不好就多刷几道

django事物



```
settings.py x views.py x
from django.db import transaction
from django.http import HttpResponse
from django.shortcuts import render

def hello_transaction(request):
    try:
        with transaction.atomic():
            pass
    except Exception as e:
        transaction.rollback()
    else:
        transaction.commit()

    return HttpResponse("I")
```

设置回滚点


```
settings.py  views.py  transaction.py

from django.db import transaction
from django.http import HttpResponse
from django.shortcuts import render

def hello_transaction(request):
    try:
        with transaction.atomic():
            pass
    except Exception as e:
        transaction.rollback()
    else:
        transaction.commit()

    return HttpResponse("Hello")

def welcome_transaction(request):
    save_point = transaction.savepoint()

    try:
        pass
    except Exception as e:
        transaction.savepoint_rollback(save_point)
    else:
        transaction.savepoint_commit(save_point)

    return HttpResponse("Welcome")
```

orm扩展

- 执行原生sql

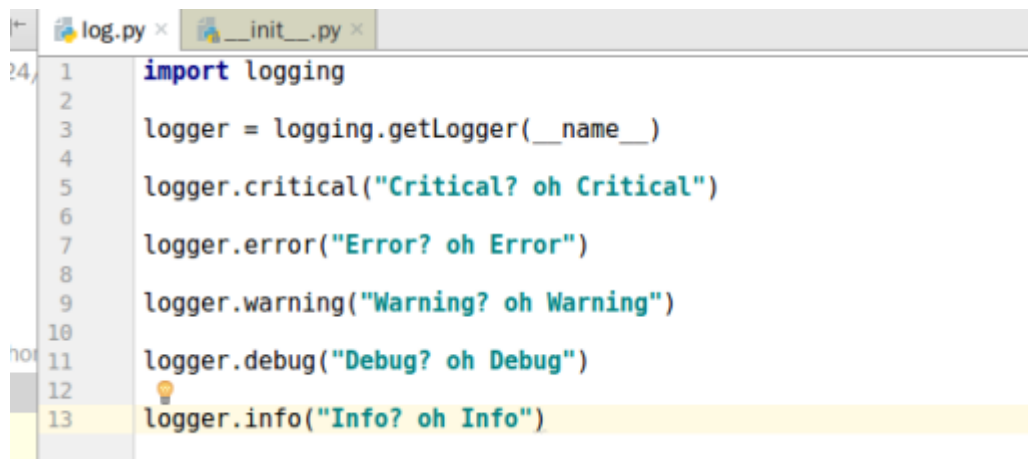
res = User.object.raw("select * from user")

- 查询指定字段:

- defer
 - 不要哪些字段
- only
 - 只要哪些字段

python原装中的日志管理

logging

A screenshot of a code editor with two tabs: 'log.py' and '__init__.py'. The 'log.py' tab is active, showing a Python script that uses the logging module. The code is as follows:

```
1 import logging
2
3 logger = logging.getLogger(__name__)
4
5 logger.critical("Critical? oh Critical")
6
7 logger.error("Error? oh Error")
8
9 logger.warning("Warning? oh Warning")
10
11 logger.debug("Debug? oh Debug")
12
13 logger.info("Info? oh Info")
```

The line numbers 1 through 13 are visible on the left side of the code. The line 'logger.info("Info? oh Info")' is highlighted in yellow. There is a lightbulb icon next to line 12.

```

import logging

logger = logging.getLogger(__name__)

logger.setLevel(level=logging.INFO)

file_handler = logging.FileHandler("log.txt")

file_handler.setLevel(level=logging.INFO)

file_formatter = logging.Formatter("%(levelname)s - %(asctime)s - %(message)s")

file_handler.setFormatter(file_formatter)

logger.addHandler(file_handler)

|
logger.critical("Critical? oh Critical")

logger.error("Error? oh Error")

logger.warning("Warning? oh Warning")

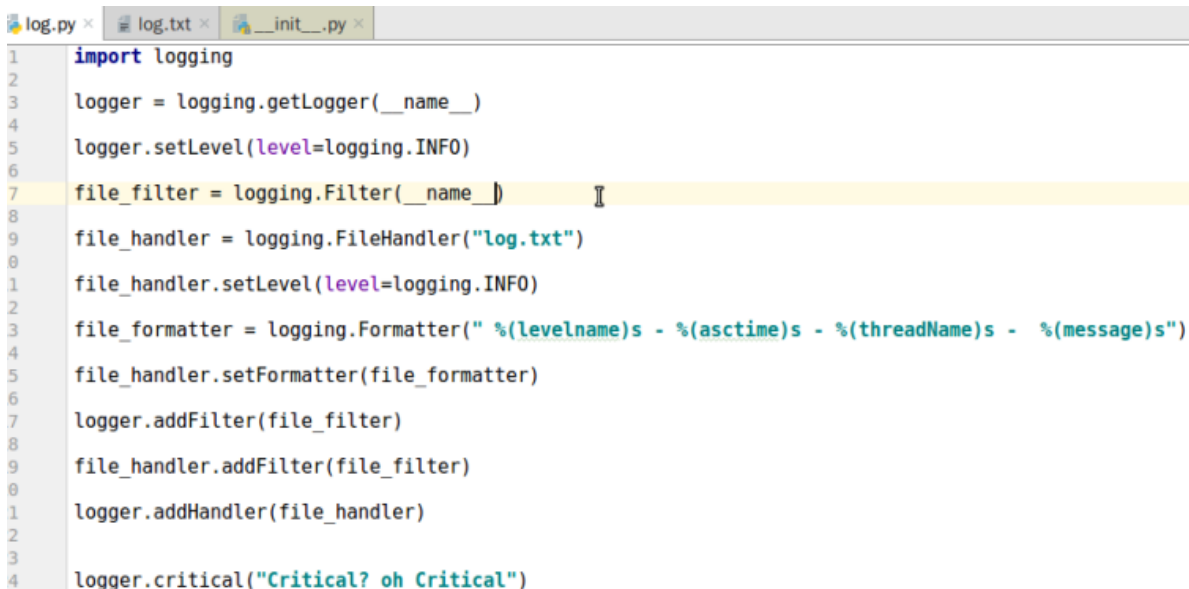
logger.debug("Debug? oh Debug")

logger.info("Info? oh Info")

```

日志处理流程

- 通过logger对象进行输出
- 交给handler处理者进行处理
 - 处理者可以对输出样式进行格式化
 - 也可对数据进行过滤



```

log.py x log.txt x __init__.py x
1 import logging
2
3 logger = logging.getLogger(__name__)
4
5 logger.setLevel(level=logging.INFO)
6
7 file_filter = logging.Filter(__name__)
8
9 file_handler = logging.FileHandler("log.txt")
10
11 file_handler.setLevel(level=logging.INFO)
12
13 file_formatter = logging.Formatter("%(levelname)s - %(asctime)s - %(threadName)s - %(message)s")
14
15 file_handler.setFormatter(file_formatter)
16
17 logger.addFilter(file_filter)
18
19 file_handler.addFilter(file_filter)
20
21 logger.addHandler(file_handler)
22
23
24 logger.critical("Critical? oh Critical")

```

LOG

Log简介

logging模块是Python内置的标准模块，主要用于输出运行日志，可以设置输出日志的等级、日志保存路径、日志文件回滚等；相比print，具备如下优点：

通过log的分析，可以方便用户了解系统或软件、应用的运行情况；如果你的应用log足够丰富，也可以分析以往用户的操作行为、类型喜好、地域分布或其他更多信息；如果一个应用的log同时也分了多个级别，那么可以很轻易地分析得到该应用的健康状况，及时发现问题并快速定位、解决问题，补救损失。

Log的用途

不管是使用何种编程语言，日志输出几乎无处不再。总结起来，日志大致有以下几种用途：

- 问题追踪：通过日志不仅仅包括我们程序的一些bug，也可以在安装配置时，通过日志可以发现问题。
- 状态监控：通过实时分析日志，可以监控系统的运行状态，做到早发现问题、早处理问题。
- 安全审计：审计主要体现在安全上，通过对日志进行分析，可以发现是否存在非授权的操作

Log等级

- DEBUG最详细的日志信息，典型应用场景是 问题诊断
- INFO信息详细程度仅次于DEBUG，通常只记录关键节点信息，用于确认一切都是按照我们预期的那样进行工作
- WARNING当某些不期望的事情发生时记录的信息（如，磁盘可用空间较低），但是此时应用程序还是正常运行的
- ERROR由于一个更严重的问题导致某些功能不能正常运行时记录的信息 如IO操作失败或者连接问题
- CRITICAL当发生严重错误，导致应用程序不能继续运行时记录的信息

Log模块的四大组件

- Loggers
提供应用程序代码直接使用的接口
- Handlers
用于将日志记录发送到指定的目的位置

FileHandler: logging.FileHandler; 日志输出到文件

RotatingHandler: logging.handlers.RotatingHandler; 日志回滚方式, 支持日志文件最大数量和日志文件回滚

SMTPHandler: logging.handlers.SMTPHandler; 远程输出日志到邮件地址

HTTPHandler: logging.handlers.HTTPHandler; 通过"GET"或者"POST"远程输出到HTTP服务器

等等

- Filters

提供更细粒度的日志过滤功能，用于决定哪些日志记录将会被输出（其它的日志记录将会被忽略）

- Formatters

用于控制日志信息的最终输出格式

`%(levelno)s`: 打印日志级别的数值

`%(levelname)s`: 打印日志级别的名称

`%(pathname)s`: 打印当前执行程序的路径，其实就是`sys.argv[0]`

`%(filename)s`: 打印当前执行程序名

`%(funcName)s`: 打印日志的当前函数

`%(lineno)d`: 打印日志的当前行号

`%(asctime)s`: 打印日志的时间

`%(thread)d`: 打印线程ID

`%(threadName)s`: 打印线程名称

`%(process)d`: 打印进程ID

`%(message)s`: 打印日志信息

`datefmt`: 指定时间格式，同`time.strftime()`;

`level`: 设置日志级别，默认为`logging.WARNNING`;

`stream`: 指定将日志的输出流，可以指定输出到`sys.stderr`，`sys.stdout`或者文件，默认输出到`sys.stderr`，当`stream`和`filename`同时指定时，`stream`被忽略

示例

```
import logging
logger = logging.getLogger(__name__)
logger.setLevel(level = logging.INFO)
handler = logging.FileHandler("log.txt")
# handler.setLevel(logging.INFO)
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s
- %(message)s')
handler.setFormatter(formatter)
logger.addHandler(handler)

logger.info("Start print log")
logger.debug("Do something")
logger.warning("Something maybe fail.")
logger.info("Finish")
```

Django中的配置

```
ADMINS = (
    ('tom', '*****@163.com'),
    \
```

```

)
LOGGING = {
    'version': 1,
    'disable_existing_loggers': True,
    'formatters': {
        'standard': {
            'format': '%(asctime)s [%(threadName)s:%(thread)d] [%(name)s:%
(lineno)d] [%(module)s:%(funcName)s] [%(levelname)s]- %(message)s'
        },
        'filters': {
            'require_debug_false': {
                '()': 'django.utils.log.RequireDebugFalse',
            }
        },
        'handlers': {
            'null': {
                'level': 'DEBUG',
                'class': 'logging.NullHandler',
            },
            'mail_admins': {
                'level': 'ERROR',
                'class': 'django.utils.log.AdminEmailHandler',
                'filters': ['require_debug_false'],
            },
            'debug': {
                'level': 'DEBUG',
                'class': 'logging.handlers.RotatingFileHandler',
                'filename': os.path.join(BASE_DIR, "log", 'debug.log'), #文件路径
                'maxBytes': 1024*1024*5,
                'backupCount': 5,
                'formatter': 'standard',
            },
            'console': {

```



```

console :{
    'level': 'DEBUG',
    'class': 'logging.StreamHandler',
    'formatter': 'standard',
},
},
'loggers': {
'django': {
    'handlers': ['console'],
    'level': 'DEBUG',
    'propagate': False
},
'django.request': {
    'handlers': ['debug','mail_admins'],
    'level': 'ERROR',
    'propagate': True,是否继承父类的log信息
},
# 对于不在 ALLOWED_HOSTS 中的请求不发送报错邮件
'django.security.DisallowedHost': {
    'handlers': ['null'],
    'propagate': False,
},
}
}
}

```

```

import logging
logger = logging.getLogger("django") # 为loggers中定义的名称
logger.info("some info...")

```

课堂修改

```

LOGGING = {
    'version': 1, #版本
    'disable_existing_loggers': False, #是否禁用其他的logger
    'formatters': {
        'standard': {
            'format': '%(asctime)s [%(threadName)s:%(thread)d] [%(name)s:%(lineno)d] [%(module)s:%(funcName)s] [%(levelname)s]- %(message)s',
            'myself':{
                'format': '%(asctime)s- %(message)s '
            }
        },
        'filters': {
            # 'require_debug_false': {
            #     '(): 'django.utils.log.RequireDebugFalse',
            # },
            'require_debug_true':{
                '(): 'django.utils.log.RequireDebugTrue',
            }
        },
        'handlers': {
            'null': {
                'level': 'DEBUG',
                'class': 'logging.NullHandler',
            },
            'mail_admins': {
                'level': 'ERROR',
                'class': 'django.utils.log.AdminEmailHandler',
                'filters': ['require_debug_true'],
            },
            'debug': {
                'level': 'DEBUG',
                'class': 'logging.handlers.RotatingFileHandler'
            }
        }
    }
}

```

```
        class : logging.handlers.RotatingFileHandler ,
        'filename': os.path.join(BASE_DIR, "log", 'debug.log'),#log文件地址
手动新建一个log目录下debug.log, 不然会报错
        'maxBytes': 1024 * 1024 * 5,# 5M
        'backupCount': 5,
        'formatter': 'myself',
    },
    'console': {
        'level': 'DEBUG',
        'class': 'logging.StreamHandler',
        'formatter': 'myself',
    },

},
'loggers': {
    'django': {
        'handlers': ['console', 'debug'],
        'level': 'DEBUG',
        'propagate': True
    },
    'django.request': {
        'handlers': ['debug', 'mail_admins'],
        'level': 'ERROR',
        'propagate': True, # 是否继承父类的log信息
    },
    # 对于不在 ALLOWED_HOSTS 中的请求不发送报错邮件
    'django.security.DisallowedHost': {
        'handlers': ['null'],
        'propagate': False,
    },
}
}
```

