Django

- 单词

  migrate 迁移
  brief 摘要/简短的
  publish 公布
  paginator 页面/分页

- Django基本命令

  新建Django项目：django-admin.py startproject
  新建app：python manage.py startapp 或 django-admin.py startapp
  同步数据库：
  python manage.py makemigrations
  python manage.py migrate
  这种方法可以创建表，当你在models.py中新增了类时，运行它就可以自动在数据库中创建表了，不用手动创建。
  使用开发服务器运行：python manage.py runserver
  使用其它端口：python manage.py runserver 8001
  监听所有可用ip：python manage.py runserver 0.0.0.0:8000
  开发服务器，由于性能问题，建议只用来测试，不要用在生产环境。
  清空数据库：python manage.py flush
  此命令会询问是 yes 还是 no, 选择 yes 会把数据全部清空掉，只留下空表
  创建超级管理员：python manage.py createsuperuser
  按照提示输入用户名和对应的密码就好了邮箱可以留空，用户名和密码必填
  修改用户密码：python manage.py changepassword username
  数据导入导出：
  导出：python manage.py dumpdata appname > appname.json
  导入：python manage.py loaddata appname.json
  进入项目环境终端：python manage.py shell
  这个命令和 直接进入 shell 的区别是：你可以在这个 shell 里面调用当前项目的 models.py 中的 API，对于操作数据，还有一些小测试非常方便
  数据库命令行：python manage.py dbshell
  Django 会自动进入在settings.py中设置的数据库，如果是 MySQL 或 postgreSQL,会要求输入数据库用户密码。在这个终端可以执行数据库的SQL语句。
  更多命令：终端上输入 python manage.py 可以看到详细的列表

- 创建项目

  Django-admin startproject 项目名

- 项目目录

  wsgi.py--->WSGI(python web server gateway interface)--->python服务器网关接口--->python应用与web服务器之间的接口
  urls.py--->url配置文件
  settings.py--->项目的总配置文件,包含数据库,web应用,时间
  __init__.py--->声明模块的文件,把项目当成模块引用

- 应用目录

views.py--->执行响应的代码所在模块/代码逻辑处理的主要地点
apps.py--->声明应用

- 启动Django项目

  python manage.py runserver 8080

- 创建应用

  python manage.py startapp blog

- 使用API

  python manage.py shell

- 运行 python manage.py makemigrations blog为模型的改变生成迁移文件--->生成文件
- 运行 python manage.py migrate 来应用数据库迁移--->生成数据库结构
- 查看命令--->Django-admin/python manage.py
- settings--->urls--->views
- Django项目与应用的关系

  一个Django项目就是一个基于Django的Web应用。
  一个Django项目中包含一组配置和若干个Django应用。
  一个Django应用就是一个可重用的Python软件包，提供一定的功能。
  一个Django应用中可以包含models, views, templates, template tags, static files, URLs等。
  一个Django项目可以包含多个Django应用。
  一个Django应用也可以被包含到多个Django项目中，因为Django应用是可重用的Python软件包。

- 创建应用流程

  0.进入manage.py 同级目录
  1.创建一个新的应用--->python manage.py startapp newname
  2.写视图函数--->views.py
  3.配置路由--->urls.py
  4. 配置settings--->项目配置文件中--->settingspy--->INSTALLED_APPS--->应用名.apps.应用名Config--->blog.apps.BlogConfig
  5. 启动项目--->控制台中输入--->python manage.py runserver (8080)---->默认端口8000

- Django 模型

  why-->1. 屏蔽不同数据库之间的差异
  2.开发者更加专注于业务逻辑的开发
  3.提供很多便捷工具有助开发(数据库迁移/备份/表备份)

  模型层定义字段--->数字类型--->IntegerField
  文本类型--->TextField
  日期类型--->DateTimeField(auto_now=True) # 如果没有指定日期,默认当前日期为发布日期--->auto_now
  主键自增ID--->AutoField(primary_key=True)

  0.创建模型流程--->设计模型--->模型层定义字段
  1. 写好model--->models.py--->创建数据结构类型(继承models.Model)

2.为模型的改变生成迁移文件(生成迁移文件)---> python manage.py makemigrations 应用名
3.应用数据库迁移--->生成数据库结构同步到到数据库--->python manage.py migrate

- Django Shell

what--->python shell 用于交互式的python编程
Django Shell 类似,并继承Django项目环境
why--->方便开发/测试/Debug
临时性操作使用Django Shell更加方便
小范围Debug更简单,不需要运行整个项目来测试
how--->控制台输入--->python manage.py shell--->进入Django shell环境

--->数据存储--->

```
>>> from blog.models import Article
>>> a = Article()
>>> a.title = 'Test Django Shell'
>>> a.brief_content = 'Test Django Shell, By Titanxz 1904.'
>>> a.content = 'Test Django Shell, New Articel, Main content.'
>>> print(a)  # Article object (None)
>>> a.save() --->保存到数据库
```

--->数据查看--->

```
>>> articles = Article.objects.all()--->查看全部
>>> article = articles[0]--->取第一个
>>> print(article.title)  # Test Django Shell --->打印标题
```

- Django Admin 模块

开启debug=True,否则后台会失去css样式
what--->Django的后台管理工具
读取定义的模型元数据,提供强大的管理使用页面
why--->Django Shell新增文章太复杂了
管理页面是基础设施中重要的部分
提供认证用户/显示管理模型/效验输入等类似功能
how--->
1.创建管理员用户

--->控制台输入
--->python manage.py createsuperuser
--->username--->root/pwd--->password

2.登录页面进行管理

数据的操作界面的展示
--->将模型注册到admin中
--->from .models import Article
--->admin.site.register(Article)

--->后台列表显示为文章的标题

--->models.py
--->def __str__(self):
--->   return self.title

- 实现博客数据返回页面

----->1.views.py
--->导包--->from blog.models import Article
--->编写视图函数
--->def article_content(request):
    article = Article.objects.all()[0]
    title = article.title
    brief_content = article.brief_content
    content = article.content
    article_id = article.article_id
    publish_date = article.publish_date
    return_str = f'title: {title}, brief_content: {brief_content}, content: {content}, article_id: {article_id}, publish_date: {publish_date}'
    return HttpResponse(return_str)
--->2.应用urls.py--->添加路径--->path('content', blog.views.article_content)

- Django视图

使用Bootstrap实现静态博客页面
--->页面布局设计
    博客首页
    文章详情页
--->Bootstrap以及Bootstrap的栅格系统
    来自美国Twitter的前端框架
    提供非常多的控件及源码
    栅格系统把页面均分为十二等分
--->实现静态页面

- Django模板

模板简介
    --->what
        HTML文件
        模板系统的表现形式是文本
        分离文档的表现形式和表现内容
        模板系统定义了特有的标签占位符
    --->why
        视图文件不适合编码HTML
        页面设计改变需要修改python代码
        网页逻辑和网页视图应该分开设计
基本语法
--->how
变量标签: {{ 变量 }}
for循环标签: {%for x in list%}, {% endfor %}
if-else分支标签: {% if %}, {% else %}, {% endif %}

- 使用模板系统渲染博客页面

  博客首页
  文章详情页

- 修改数据库时间

```
TIME_ZONE = 'Asia/Shanghai'
USE_TZ = False
```

- 实现文章详情页面跳转

```
设计文章详情页URL
--->urls.py--->
path('detail/<int:article_id>', blog.views.get_detail_page)
完善视图函数逻辑
--->views.py--->
def get_detail_page(request, article_id):
    all_article = Article.objects.all()
    curr_article = None
    for article in all_article:
        if article.article_id == article_id:
            curr_article = article
            break
    section_list = curr_article.content.split('\n')
    return render(request, 'blog/detail.html',
                  {
                      'curr_article': curr_article,
                      'section_list': section_list
                  }
                  )
实现首页跳转
--->index.html--->
<h2><a href="/blog/detail/{{ article.article_id }}">{{ article.title }}</a></h2>
```

- 实现上下篇文章跳转

```
--->detail.html
<div>
    <nav aria-label="...">
        <ul class="pager">
        <li><a href="/blog/detail/{{ previous_article.article_id }}">上一篇: {{ previous_article }}</a></li>
        <li><a href="/blog/detail/{{ next_article.article_id }}">下一篇: {{ next_article }}</a></li>
        </ul>
    </nav>
</div>
--->views.py
def get_detail_page(request, article_id):
    all_article = Article.objects.all()
    curr_article = None
    previous_index = 0
    next_index = 0
```

```
    previous_article = None
    next_article = None
    for index, article in enumerate(all_article):
        if index == 0:
            previous_index = 0
            next_index = index + 1
        elif index == len(all_article) - 1:
            previous_index = index - 1
            next_index = index
        else:
            previous_index = index - 1
            next_index = index + 1
        if article.article_id == article_id:
            curr_article = article
            previous_article = all_article[previous_index]
            next_article = all_article[next_index]
            break
    section_list = curr_article.content.split('\n')
    return render(request, 'blog/detail.html',
                  {
                      'curr_article': curr_article,
                      'section_list': section_list,
                      'previous_article': previous_article,
                      'next_article': next_article
                  }
                  )
```

- 实现分页功能

Bootstrap实现分页按钮
设计分页url
使用Django分页组件实现分页功能

```
>>> from django.core.paginator import Paginator  # 分页组件
>>> list=[1,2,3,4]
>>> print(l)
[1, 2, 3, 4]
>>> p=Paginator(list,2)  # 两个参数(列表,每一页的张数)
>>> p.num_pages  # 分页的数量
2
>>> p.count  # 分页总数--->列表的长度
4
>>> page1=p.page(1)  # 获取第一页的分页
>>> page1.object_list  # 获取第一页的内容
[1, 2]
>>> page2=p.page(2)
>>> page2.object_list
[3, 4]
>>> page2.has_next()  # 是否有下一页
False
```

```
>>> page1.has_next()
True
>>> page1.has_previous()  # 是否有上一页
False
>>> exit()
```

- 实现最近文章列表

```
--->views.py
top5_article_list = Article.objects.order_by('-publish_date')[:5]  # 已发布日期为准倒序取前5  -号表示倒序
```

- 查看sql语句

```
python manage.py sqlmigrate app_name 文件ID(0001)
```

- 修改admin后台

```
--->admin.py
class ArticleAdmin(admin.ModelAdmin):
   list_display = ('title', 'publish_date')   # 后台显示标题,日期
   list_filter = ('publish_date',)   # 过滤器
admin.site.register(Article, ArticleAdmin)  # 将模型注册到admin中
```

- Django模型层使用
  - 跨DB迁移--->sqlite3--->mysql

```
why--->sqlite3是文件数据库,性能跟不上
        --->MySQL是工业界常用(免费开源)
迁移过程--->重要的东西--->数据/表结构
how--->1--->数据备份---> python manage.py dumpdata (appname)>appname.json--->可以不指定应用名
        --->2.--->表结构同步
                    --->1.pip install mysqlclient
                    --->2.更改settings.py--->mysql创建好数据库--->blog
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    },
    'slave': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'blog',
        'USER': 'root',
        'PASSWORD': 'password',
        'HOST': 'localhost',
        'PORT': '3306',
    }
}
                    --->3.python manage.py migrate --run-syncdb --database slave  --->将表结构导入mysql
                    --->4. 将settings.py---->设置为
DATABASES = {
    'default': {
```

```
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'blog',
        'USER': 'root',
        'PASSWORD': 'password',
        'HOST': 'localhost',
        'PORT': '3306',
    }
}
```

--->5. **python manage.py loaddata blog.json --->将数据加载到MySQL**

- 数据库索引

加快检索数据的速度
降低插入删除更新的速度
排序/比较/过滤--->的字段应该被索引

索引底层结构: B+树

添加索引两种方法

1.db_index=True---->给字段添加索引
2. 添加Meta类

```
class User(models.Model):
    openid = models.CharField(max_length=64, unique=True)
    nickname = models.CharField(max_length=64)
    # nickname = models.CharField(max_length=64, db_index=True)
    def __str__(self):
        return self.nickname
    class Meta:
        """
        元: 描绘本身
        """
        # db_table = 'abc'   # 该变表名
        # app_label = 'User'   # 定义模型类属于哪一个应用
        indexes = [
            models.Index(fields=['nickname'], name='nickname'),
        ]
```

3.控制台输入--->1.python manage.py makemigrations--->2.python manage.py migrate

查看表的索引--->show index from tablename;

- 数据库关系(表与表之间的关系)

表是由模型类映射而来

模型类与模型类之间的关系--->

一对一 : OneToField
一对多 : ForeignKey, 写在多的这一边
多对多 : ManyToManyField
ondelete 删除一个对象关系时,有关系的怎么处理
--->级联删除(CASCADE),相关的值也需要进行删除
---->DO_NOTHING 只删除这个数据
on_delete=models.DO_NOTHING
on_delete=models.CASCADE

- 数据库操作函数

```
import django
import os
import random
os.environ['DJANGO_SETTINGS_MODULE'] = 'mysite.settings'
django.setup()  # 环境配置

def ranstr(length):  # 随机生成
    CHS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789'
    salt = ''
    for i in range(length):
        salt += random.choice(CHS)
    return salt
from blog.models import User
from django.db.models import Q  #  使用 Q 对象构建复杂的查询语句
# 查询
res1 = User.objects.filter(nickname__contains='张')  # 模糊查询
# res = User.objects.get(nickname='1')  # 精确查询
users = User.objects.filter(open_id__contains='test_').order_by('open_id')  # 连锁查询
res2 = User.objects.get(Q(openid='test_openid') | Q(nickname='test_nickname'))
res3 = User.objects.filter(Q(openid='test_openid') & Q(nickname='test_nickname'))
print(res1)
print(users)
# 增加
def add_one():
    user = User(openid='test_openid', nickname='test_nickname')  # 第一种
    # User.objects.create(openid='test_openid2', nickname='test_nickname2')  # 第二种
    user.save()
# add_one()
# 批量添加
def add_batch():
    new_user_list = []
    for i in range(10):
        open_id = ranstr(32)
        nickname = ranstr(10)
        user = User(open_id=open_id, nickname=nickname)
        new_user_list.append(user)
    User.objects.bulk_create(new_user_list)
```

```
# add_batch()
# 修改一个
def modify_one():
    user = User.objects.get(open_id='test_open_id')
    user.nickname = 'modify_username'
    user.save()
# modify_one()
# 批量改
def modify_batch():
    User.objects.filter(open_id__contains='test_').update(nickname='modify_uname')
# 删一个
def delete_one():
    User.objects.get(open_id='test_open_id').delete()
# 批量删除
def delete_batch():
    User.objects.filter(open_id__contains='test_').delete()
# 全部删除
def delete_all():
    User.objects.all().delete()
    # User.objects.delete()
# ---------数据库函数----------
# 数据库函数
# 字符串拼接：Concat
from django.db.models import Value
from django.db.models.functions import Concat
# annotate创建对象的一个属性, Value,如果不是对象中原有属性
def concat_function():
    user = User.objects.filter(open_id='test_open_id').annotate(
        # open_id=(open_id), nickname=(nickname)
        screen_name=Concat(
            Value('open_id='),
            'open_id',
            Value(', '),
            Value('nickname='),
            'nickname')
    )[0]
    print('screen_name = ', user.screen_name)
# 字符串长度： Length
from django.db.models.functions import Length
def length_function():
    user = User.objects.filter(open_id='test_open_id').annotate(
        open_id_length=Length('open_id'))[0]
    print(user.open_id_length)
# 大小写函数
from django.db.models.functions import Upper, Lower
def case_function():
    user = User.objects.filter(open_id='test_open_id').annotate(
```

```
        upper_open_id=Upper('open_id'),
        lower_open_id=Lower('open_id')
    )[0]
    print('upper_open_id:', user.upper_open_id, ', lower_open_id:', user.lower_open_id)
    pass
# 日期处理函数
# Now()
from blog.models import Article
from django.db.models.functions import Now
from datetime import datetime
dt = datetime(day=1,year=2020,month=3)
# print(dt)
def now_function():
    # 当前日期之前发布的所有应用
    apps = Article.objects.filter(publish_date__lte=Now())
    for app in apps:
        print(app)
# 时间截断函数
# Trunc
from django.db.models import Count
from django.db.models.functions import Trunc
def trunc_function():
    # 打印每一天发布的应用数量
    article_per_day = Article.objects.annotate(publish_day=Trunc('publish_date', 'month')) \
        .values('publish_day') \
        .annotate(publish_num=Count('article_id'))
    for article in article_per_day:
        print('date:', article['publish_day'], ', publish num:', article['publish_num'])
```

- 迁移的详情
  - 单词

    > dependencies 依赖关系

  - 在models.py--->更改模型类
  - makemigrations--->生成迁移文件(Django框架记录一下)
  - migrate--->将迁移文件应用到数据库,改变表结构
  - 显示某个应用的模型变更和迁移历史--->python manage.py showmigratioins (appname)--->x(执行过的文件)
  - 显示每次迁移执行的实际sql语句--->python manage.py sqlmigrate appname 0001(迁移文件num)
  - dependencie字段--->依赖哪个APP的哪次迁移
  - operations字段--->增删改查,更改模型
  - 懒加载 ---- 预加载

    > 懒加载的问题根源---->ORM的不透明---->一次查询经过ORM框架会变成两次sql语句
    > 懒加载--->存在于外键与多对多关系--->默认懒加载

--->不检索关联对象的数据
----->调用关联对象会再次查询数据库
查看Django ORM 数据加载----->通过shell来验证----->通过对象的Query属性来看查询对象的sql语句
预加载的方法----->预加载单个关联对象----->select_related
----->预加载多个关联对象----->prefetch_related

- 长连接 ---- 短连接

show variables like '%max_connections%'; # 查看连接数
set global max_connections=200; # 更改最大连接数
长连接--->省时间/效率高
短连接的缺点
每个请求都将重复连接数据库
处理高并发请求给服务带来巨大压力
无法承受更高并发的服务
避免负优化
存储数据库连接的位置: 线程局部变量
CONN_MAX_AGE--->开发过程中不配置--->每次请求创建一个连接,给服务器带来巨大压力

- 数据库操作规范
  - 使用正确的优化策略优化查询
    - 正确的使用查询
    - 索引该索引的列
      - 如何使用索引
      - 什么列该索引
    - 不索引不该索引的列
  - 使用iterator迭代器代替QuerySet

    QuerySet 非常大 迭代器节省内存
  - 理解对象的属性缓存

    不可调用的属性会被ORM框架缓存
    可调用的属性不会被ORM框架缓存
  - 数据库的工作留给数据库做

    过滤:使用filter,exclude属性
    聚合: 使用annotate函数进行聚合
    必要时,使用原始sql
  - 正确检索行数据

    使用被索引的列字段检索
    使用被unique修饰的列字段索引
  - 不要进行不必要的检索

    QuerySet使用values, value_list()函数返回python结构容器
    查询结果长度使用QuerySer.count()而不是len(QuerySet)
    判断是否为空使用QuerySer.exists()而不是if QuerySet

不进行不必要的排序--->考虑数据足够小了吗/使用排序的列是索引吗

- 批量操作

  大量数据的时候

- 日志模块

DEBUG：用于调试目的的低级系统信息
INFO：一般系统信息
WARNING：描述已发生的小问题的信息。
ERROR：描述已发生的主要问题的信息。
CRITICAL：描述已发生的严重问题的信息。

产生日志
--->渲染格式(格式化--->formatter)
---->匹配过滤(--->filter)
--->持久化(保存文件--->handler)
---->loggers
---->生成一个实例日志,跑起来

```
# 日志配置
LOGGING = {
    'version': 1,
    # 渲染格式--->格式化
    'formatters': {
        'standard': {
            'format': '%(asctime)s [%(threadName)s: %(thread)d]'  # 时间/线程名字/线程ID
                      '%(pathname)s:%(funcName)s:%(lineno)d %(levelname)s - %(message)s'
        }
    },
    # 过滤器--->匹配过滤
    'filters': {
        'xxx': {
            '()': 'ops.XXXFilter'
        }
    },
    # 持久化
    'handlers': {
        # 输出控制台
        'console_handler': {
            'level': 'INFO',
            'class': 'logging.StreamHandler',
            'formatter': 'standard'
        },
        # 输出文件
        'file_handler': {
            'level': 'DEBUG',
            'class': 'logging.handlers.RotatingFileHandler',
            # todo maybe loging file no exist
```

```
            'filename': os.path.join(BASE_DIR, 'ops.log'),
            'maxBytes': 100 * 1024 * 1024,  # 文件最大100M
            'backupCount': 3,  # 备份数量
            'formatter': 'standard',
            'encoding': 'utf8'
        }
    },
    # 生成一个实例日志 django
    'loggers': {
        'django': {
            'handlers': ['console_handler', 'file_handler'],
            'filters': ['xxx'],
            'level': 'DEBUG'
        }
    }
}

--->import django
import os
import logging
os.environ['DJANGO_SETTINGS_MODULE'] = 'mysite.settings'
django.setup()  # 配置环境

def logdemo():
    # 得到配置的实例对象
    logger = logging.getLogger('django')
    logger.info('i am info log')
if __name__ == '__main__':
    logdemo()
# 日志过滤器
from logging import Filter
class XXXFilter(Filter):
    def filter(self, record):
        if 'lc' in record.msg:
            return False
        else:
            return True
```

- admin模块

```
认证用户--->权限验证--->页面展示--->数据管理
models--->定义模型的地方
admin--->把模型注册到后台的地方,不注册不显示

# 装饰器写法 效果相当于注册模型--->admin.site.register(User, UserAdmin)
@admin.register(User)
class UserAdmin(admin.ModelAdmin):
    exclude = ['openid'] # 不包含openid
```

```
    # 可以定义一些规则来控制插入模型字段的值
    def save_model(self, request, obj, form, change):
        print('--->',obj)
        print(request.method)
        obj.openid = obj.name + str(random.randint(1,1000))
        return super(UserAdmin, self).save_model(request, obj, form, change)
```

- 缓存

```
电脑---->内存就是缓存--->介于CPU与硬盘线之间---->两者之间速度差异大
what--->高速缓存 Cache
缓存--->为了解决速度不一致情况
        --->根本目的是为了加快数据访问速度,提高性能
         --->协调两者数据传输速度的差异
缺点--->成本太高
缓存算法--->FIFO(队列--->最不经常使用)       /  LRU /LFU--->最近最少使用

缓存类型--->基于内存/文件/数据库/缓存框架的
缓存配置
CACHES = {
    'default': {
        # 1. MemCache
        # 'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        # 'LOCATION': '127.0.0.1:11211',
        # 2. DB Cache
        # 'BACKEND': 'django.core.cache.backends.db.DatabaseCache',
        # 'LOCATION': 'my_cache_table',
        # 3. Filesystem Cache
        # 'BACKEND': 'django.core.cache.backends.filebased.FileBasedCache',
        # 'LOCATION': '/var/tmp/django_cache',
        # 4. Local Mem Cache
        'BACKEND': 'django.core.cache.backends.locmem.LocMemCache',
        'LOCATION': 'backend-cache'
    }
}
缓存存储获取
import django
import os
from django.core.cache import cache
import time
os.environ['DJANGO_SETTINGS_MODULE'] = 'mysite.settings'
django.setup()
def get1():
    time.sleep(6)
    res = cache.get('coke')  # cache 全局缓存
    print('get1--->', res)
if __name__ == '__main__':
    # 存数据
```

```
    cache.set('coke', 'cookies', 5)  # 缓存存在时间5秒
    # 获取数据
    print(cache.get('coke'))
    get1()
```

- 迁移代码 开发环境(windows)--->生产环境(Linux)

代码: git
开发环境: push
服务器: pull
环境:
python:
  开发服务器: pip freeze> requirement.txt
  服务器: pip install -r requirement.txt
select version()
windows--->cmd(不进入mysql)--->本地开发环境把表结构和内容导出来---->mysqldump -h localhost -uroot -ppassword data > data.sql

服务器导入表结构和内容--->sudo mysql -uroot -ppassword data < data.sql
apt-get install libmysqlclient-dev
安装mysql:
  sudo apt-get install mysql-server mysql-client
  然后mysql -V查看mysql是否安装成功
  sudo apt-get install libmysqlclient-dev python3-dev
  然后
  pip install mysqlclient就不会报错找不到'mysql_config'了
sudo apt-get update
sudo apt-get install libmysql-dev
sudo apt-get install libmysqlclient-dev
sudo apt-get install python-dev #python3要装python3-dev
pip install mysqlclient

- Linux下的crontab命令

用于提交和管理用户的周期性任务
crond进程每分钟定时检查
时间间隔格式
minutes hours day month week command

用户所建立的crontab文件中，每一行都代表一项任务，每行的每个字段代表一项设置，它的格式共分为六个字段，前五段是时间设定段，第六段是要执行的命令段，格式如下：
minute hour day month week command
其中：
minute： 表示分钟，可以是从0到59之间的任何整数。
hour：表示小时，可以是从0到23之间的任何整数。
day：表示日期，可以是从1到31之间的任何整数。
month：表示月份，可以是从1到12之间的任何整数。
week：表示星期几，可以是从0到7之间的任何整数，这里的0或7代表星期日。
command：要执行的命令，可以是系统命令，也可以是自己编写的脚本文件。

在以上各个字段中，还可以使用以下特殊字符：
星号（*）：代表所有可能的值，例如month字段如果是星号，则表示在满足其它字段的制约条件后每月都执行该命令操作。
逗号（,）：可以用逗号隔开的值指定一个列表范围，例如，“1,2,5,7,8,9”
中杠（-）：可以用整数之间的中杠表示一个整数范围，例如“2-6”表示“2,3,4,5,6”
正斜线（/）：可以用正斜线指定时间的间隔频率，例如“0-23/2”表示每两小时执行一次。同时正斜线可以和星号一起使用，例如*/10，如果用在minute字段，表示每十分钟执行一次。

使用实例
实例1：每1分钟执行一次command
命令：
* * * * * command

实例2：每小时的第3和第15分钟执行
命令：
3,15 * * * * command

实例3：在上午8点到11点的第3和第15分钟执行
命令：
3,15 8-11 * * * command

实例4：每隔两天的上午8点到11点的第3和第15分钟执行
命令：
3,15 8-11 */2 * * command

实例5：每个星期一的上午8点到11点的第3和第15分钟执行
命令：
3,15 8-11 * * 1 command

实例6：每晚的21:30重启smb
命令：
30 21 * * * /etc/init.d/smb restart

实例7：每月1、10、22日的4 : 45重启smb
命令：
45 4 1,10,22 * * /etc/init.d/smb restart

实例8：每周六、周日的1 : 10重启smb
命令：
10 1 * * 6,0 /etc/init.d/smb restart

实例9：每天18 : 00至23 : 00之间每隔30分钟重启smb
命令：
0,30 18-23 * * * /etc/init.d/smb restart

实例10：每星期六的晚上11 : 00 pm重启smb

命令：
0 23 * * 6 /etc/init.d/smb restart

实例11：每一小时重启smb
命令：
* */1 * * * /etc/init.d/smb restart

实例12：晚上11点到早上7点之间，每隔一小时重启smb
命令：
* 23-7/1 * * * /etc/init.d/smb restart

实例13：每月的4号与每周一到周三的11点重启smb
命令：
0 11 4 * mon-wed /etc/init.d/smb restart

实例14：一月一号的4点重启smb
命令：
0 4 1 jan * /etc/init.d/smb restart
实例15：每小时执行/etc/cron.hourly目录内的脚本
命令：
01 * * * * root run-parts /etc/cron.hourly
说明：
run-parts这个参数了，如果去掉这个参数的话，后面就可以写要运行的某个脚本名，而不是目录名了

- django-crontab 命令的使用

只能在linux上使用
安装----> pip install django-crontab
配置--->
在settings的INSTALLED_APPS中 添加 'django_crontab'
在settings中添加--->
CRONJOBS = [
('*/2 * * * *', 'cron.jobs.demo'), # cron.jobs.demo是一个函数dosomething
('*/2 * * * *', 'echo "xxxxx"'),
('*/3 * * * *', '/bin/ls')
]
cron.jobs--->

```
import time
import datetime
import logging
log = logging.getLogger('django')
def demo():
    message = 'corntab..., now time is:' + str(datetime.datetime.now())
    log.info('corntab...')
    log.warning('corntab... warning')
    log.warning(message)
```

将内容同步到服务器--->git

在服务器查看定时任务列表---->pyhton manage.py crontab show
添加定时任务--->python manage.py crontab add
通过linux命令查看定时任务--->检查定时任务是否正常工作,看看有没有日志
crontab -l 查看时间表有哪些
crontab -e 以编辑状态打开时间表
其他操作--->**python manage.py crontab add/show/remove/run**

ubuntu 安装cron
安装---> sudo apt-get install cron
启动---> service cron start
重启---> service cron restart
停止---> service cron stop
检查状态--->service cron status
查询cron可用命令---> service cron
检查cronta工具是否安装--->crontab -l

- 中间件

一个类--->请求前后在合适的时机执行相应的方法
执行流程--->每次请求都会两次经过配置的中间件--->多个中间在存在执行顺序
实现自定义中间件
settings配置--->MIDDLEWARE =['mymiddleware.testmiddleware.TestMiddle',]
testmiddleware.py--->code

```
class TestMiddle():
    def __init__(self, get_response):  # 重写函数
        self.get_response = get_response
    def __call__(self, request):   # 重写函数
        print('------------------>before call')
        response = self.get_response(request)
        print('------------------>after call')
        return response
```

- 发送邮箱

```
# Email config
# QQ邮箱 SMTP 服务器地址
# EMAIL_HOST = 'smtp.qq.com'
EMAIL_HOST = "smtp.163.com"
# 端口 附加码25
EMAIL_PORT_SSL = 465  # 加密通道
EMAIL_PORT = 25
# 发送邮件的邮箱
EMAIL_HOST_USER = 'titanxz@163.com'
# 在邮箱中设置的客户端授权密码
EMAIL_HOST_PASSWORD = 'xxxx'
# 开启TLS
EMAIL_USE_TLS = True
# 收件人看到的发件人
EMAIL_FROM = 'titanxz@163.com'
```

```
code-->
import os
import django
import smtplib
from mysite import settings
from email.mime.text import MIMEText
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'myfirstproj.settings')
django.setup()
def send_mail():
    msg = MIMEText("邮件通道测试", "plain", "utf-8")
    msg['FROM'] = "Mail Test"
    msg['Subject'] = "【Mail Test】"
    receivers = ['titanxz@163.com']  # 接收人
    server = smtplib.SMTP_SSL(settings.EMAIL_HOST, settings.EMAIL_PORT_SSL)  # 加密通道
    # server = smtplib.SMTP(settings.EMAIL_HOST, settings.EMAIL_PORT)
    server.set_debuglevel(1)
    server.login(settings.EMAIL_HOST_USER, settings.EMAIL_HOST_PASSWORD)
    server.sendmail(settings.EMAIL_FROM, receivers, msg.as_string())
    server.close()
if __name__ == '__main__':
    send_mail()
```

幕布 - 极简大纲笔记 | 一键生成思维导图