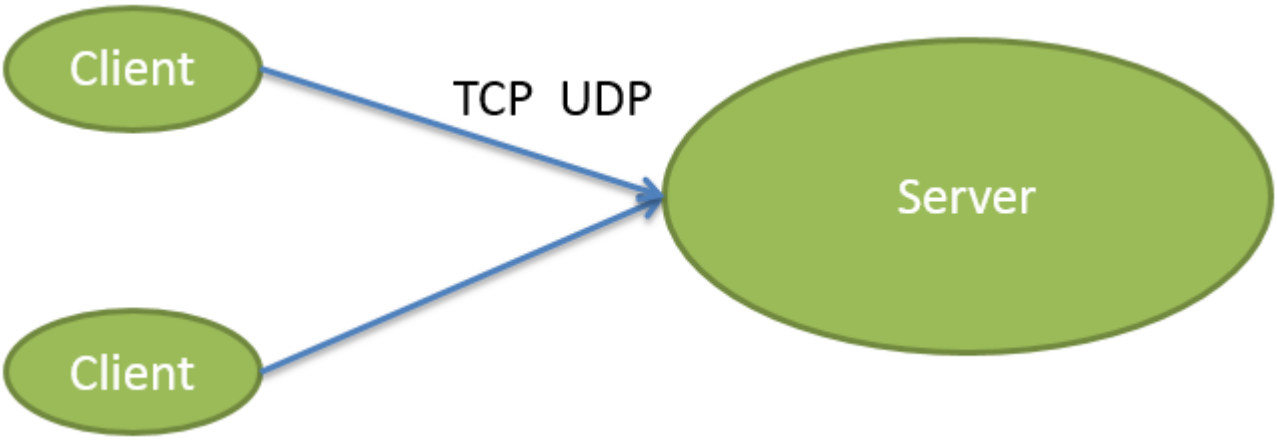
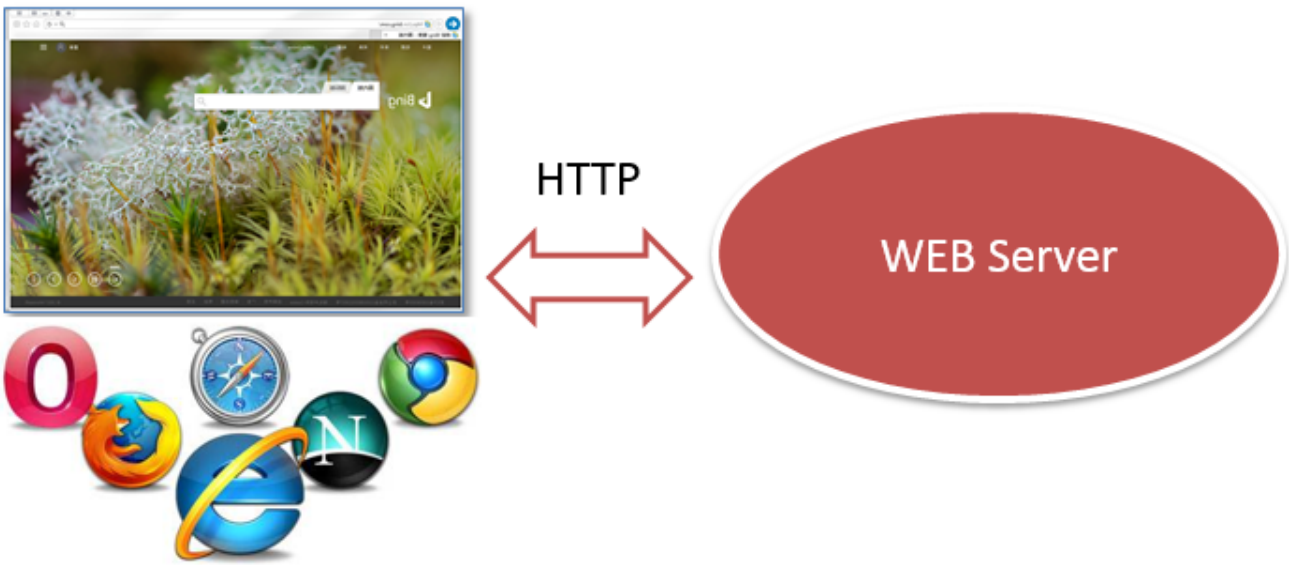
WEB开发



CS即客户端、服务器编程。
客户端、服务端之间需要使用Socket，约定协议、版本（往往使用的协议是TCP或者UDP），指定地址和端口，就可以通信了。
客户端、服务端传输数据，数据可以有一定的格式，双方必须先约定好。



BS编程，即Browser、Server开发。
Browser浏览器，一种特殊的客户端，支持HTTP(s)协议，能够通过URL向服务端发起请求，等待服务端返回HTML等数据，并在浏览器内可视化展示的程序。
Server，支持HTTP(s)协议，能够接受众多客户端发起的HTTP协议请求，经过处理，将HTML等数据返回给浏览器。
本质上来说，BS是一种特殊的CS，即客户端必须是一种支持HTTP协议且能解析并渲染HTML的软件，服务端必须是能够接收多客户端HTTP访问的服务器软件。
HTTP协议底层基于TCP协议实现。

- BS开发分为两端开发
  - 客户端开发，或称前端开发。HTML、CSS、JavaScript等
  - 服务端开发，Python有WSGI、Django、Flask、Tornado等

HTTP协议

安装httpd

可以安装httpd或nginx等服务端服务程序，通过浏览器访问，观察http协议

无状态，有连接和短连接

无状态：指的是服务器无法知道2次请求之间的联系，即使是前后2次同一个浏览器也没有任何数据能够判断出是同一个浏览器的请求。后来可以通过cookie、session来判断。

有连接：是因为它基于TCP协议，是面向连接的，需要3次握手、4次断开。

短连接：Http 1.1之前，都是一个请求一个连接，而Tcp的连接创建销毁成本高，对服务器有很大的影响。所以，自Http 1.1开始，支持keep-alive，默认也开启，一个连接打开后，会保持一段时间（可设置），浏览器再访问该服务器就使用这个Tcp连接，减轻了服务器压力，提高了效率。

协议

Http协议是无状态协议。
同一个客户端的两次请求之间没有任何关系，从服务器端角度来说，它不知道这两个请求来自同一个客户端。

URL组成

URL可以说就是地址，uniform resource locator 统一资源定位符，每一个链接指向一个资源供客户端访问。
`schema://host[:port#]/path/.../[;url-params][?query-string][#anchor]`

例如，通过下面的URL访问网页
`http://www.magedu.com/pathon/index.html?id=5&name=python`
访问静态资源时，通过上面这个URL访问的是网站的某路径下的index.html文件，而这个文件对应磁盘上的真实的文件。就会从磁盘上读取这个文件，并把文件的内容发回浏览器端。

scheme 模式、协议
http、ftp、https、file、mailto等等。

host:port
`www.magedu.com:80`，80端口是默认端口可以不写。域名会使用DNS解析，域名会解析成IP才能使用。实际上会对解析后返回的IP的TCP的80端口发起访问。

/path/to/resource
path，指向资源的路径。

?key1=value1&key2=value2
query string，查询字符串，问号用来和路径分开，后面key=value形式，且使用&符号分割。

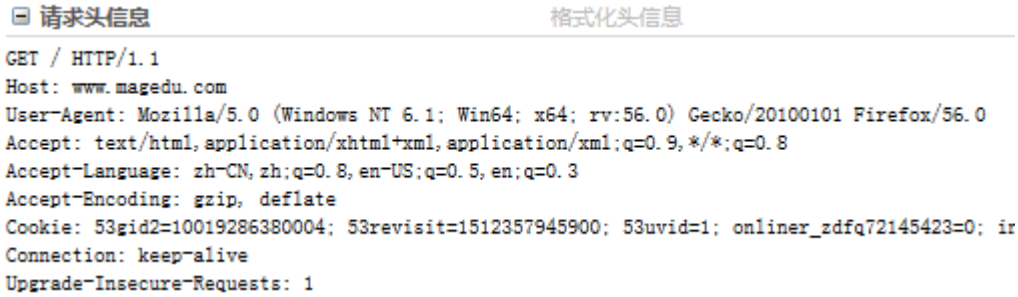
HTTP消息

消息分为Request、Response。
Request：浏览器向服务器发起的请求
Response：服务器对客户端请求的响应

请求和响应消息都是由请求行、Header消息报头、Body消息正文组成。

请求

请求消息行：请求方法Method 请求路径 协议版本CRLF



请求方法Method

```
GET     请求获取URL对应的资源
POST    提交数据至服务器端
HEAD    和GET类似，不过不返回消息正文
```

常见传递信息的方式

1、GET方法使用Query String

`http://www.magedu.com/pathon/index.html?id=5&name=python`

通过查询字符串在URL中传递参数，而URL在请求报文的头部的第一行

2、POST方法提交数据

```
http://127.0.0.1:9999/xxx/yyy?id=5&name=magedu
使用表单提交数据，文本框<input>的name属性分别为age、weight、height

请求消息如下
POST /xxx/yyy?id=5&name=magedu HTTP/1.1
HOST: 127.0.0.1:9999
content-length: 26
content-type: application/x-www-form-urlencoded

age=5&weight=80&height=170
```

请求时提交的数据是在请求报文的正文Body部分。

3、URL中本身就包含着信息

`http://www.magedu.com/python/student/001`

响应

响应消息行：协议版本 状态码 消息描述CRLF

status code状态码

```
状态码在响应头第一行
    1xx 提示信息，表示请求已被成功接收，继续处理
    2xx 表示正常响应
        200 正常返回了网页内容
    3xx 重定向
        301 页面永久性移走，永久重定向。返回新的URL，浏览器会根据返回的url发起新的request请求
        302 临时重定向
        304 资源未修改，浏览器使用本地缓存
    4xx 客户端请求错误
        404 Not Found，网页找不到，客户端请求的资源有错
        400 请求语法错误
        401 请求要求身份验证
        403 服务器拒绝请求
    5xx 服务器端错误
        500 服务器内部错误
        502 上游服务器错误，例如nginx反向代理的时候
```

cookie

键值对信息。
是一种客户端、服务器端传递数据的技术。
一般来说cookie信息是在服务器端生成，返回给浏览器端的。
浏览器端可以保持这些值，浏览器发起每一请求时，都会把cookie信息发给服务器端。
服务端收到浏览器端发过来的Cookie，处理这些信息，可以用来判断这次请求是否和之前的请求有关联。
曾经Cookie唯一在浏览器端存储数据的手段，目前浏览器端存储数据的方案很多，Cookie正在被淘汰。

当服务器收到HTTP请求时，服务器可以在响应头里面添加一个Set-Cookie选项。浏览器收到响应后通常会保存下Cookie，之后对该服务器每一次请求中都通过Cookie请求头部将Cookie信息发送给服务器。另外，Cookie的过期时间、域、路径、有效期、适用站点都可以根据需要来指定。
可以使用`Set - Cookie: NAME=VALUE; Expires=DATE; Path=PATH; Domain=DOMAIN_NAME; SECURE`

例如：

```
Set-Cookie:aliyungf_tc=AQAAAJDwJ3Bu8gkAHbrHb4zlNZGw4Y57; Path=/; HttpOnly
set-cookie:test_cookie=CheckForPermission; expires=Tue, 19-Mar-2018 15:53:02 GMT; path=/;
domain=.doubleclick.net

Set-Cookie: BD_HOME=1; path=/
```

Cookie过期	Cookie可以设定过期终止时间，过期后将被清除。如果缺省，Cookie不会持久化，浏览器关闭Cookie消失
Cookie域	域确定有哪些域可以存取这个Cookie。缺省设置属性值为当前主机，例如`www.magedu.com`。如果设置为`magedu.com`表示包含子域
Path	确定哪些目录及子目录访问可以使用该Cookie。例如`Domain=www.magedu.com; path=/webapp`，表示访问`http://www.magedu.com/webapp/a.html`的时候回发送Cookie信息
Secure	表示Cookie使用HTTPS协议发送给服务端。有些浏览器已经不允许http://使用Secure了。这个Secure不能保证Cookie是安全的传输，Cookie中不要传输敏感信息
HttpOnly	标记该Cookie，不能被JavaScript访问，只能发给服务端

Cookie一般明文传输，安全性极差，不要传输敏感数据。有4kB大小限制。每次请求中都会发送Cookie，增加了流量。

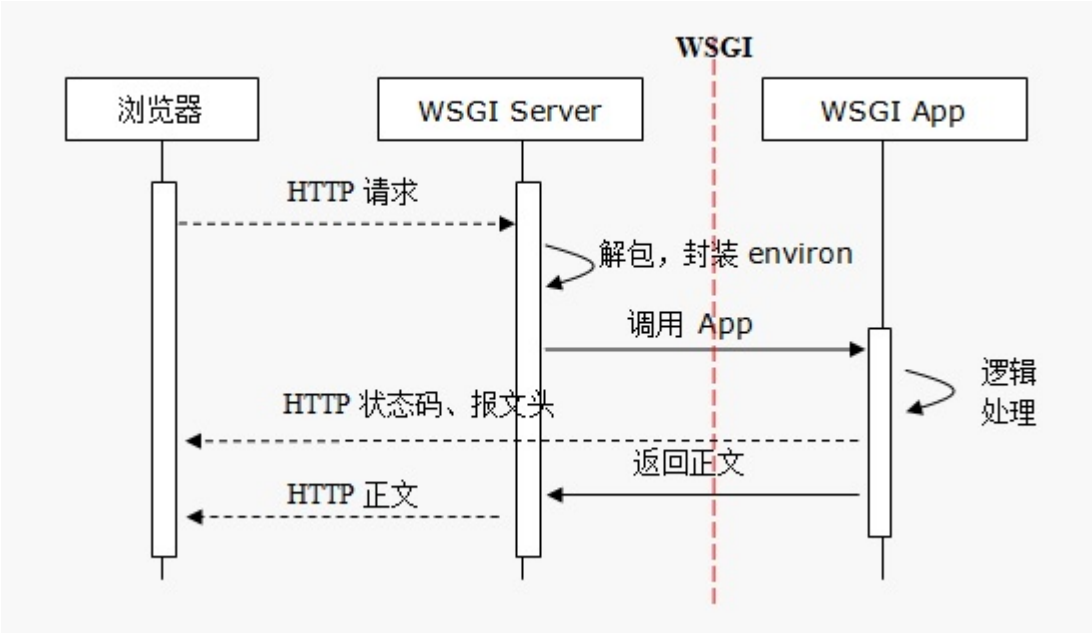
Session技术

WEB 服务器端，尤其是动态网页服务端Server，有时需要知道浏览器方是谁？但是HTTP是无状态的，怎么办？

服务端会为每一次浏览器端第一次访问生成一个SessionID，用来唯一标识该浏览器，通过Set-Cookie发送到浏览器端。
浏览器端收到之后并不永久保持这个Cookie，只是会话级的。浏览器访问服务端时，会使用Cookie，也会带上这个SessionID的Cookie值。
服务端会维持这个SessionID一段时间，如果超时，会清理这些超时没有人访问的SessionID。如果浏览器端发来的SessionID无法在服务端找到，就会自动再次分配新的SessionID，并通过Set-Cookie发送到浏览器端以覆盖原有的存在浏览器中的会话级的SessionID。

推荐图书《HTTP权威指南》

WSGI



WSGI主要规定了服务器端和应用程序间的接口。

WSGI服务器——wsgiref

wsgiref是Python提供的一个WSGI参考实现库，参考而已不适合生产环境。

wsgiref.simple_server 模块实现一个简单的WSGI HTTP服务器。

```
wsgiref.simple_server.make_server(host, port, app, server_class=WSGIServer,
handler_class=WSGIRequestHandler) # 启动一个WSGI服务器

wsgiref.simple_server.demo_app(environ, start_response)
# 一个两参数函数，小巧完整的WSGI的应用程序的实现
```

```
# 返回文本例子
from wsgiref.simple_server import make_server, demo_app

ip = '127.0.0.1'
port = 9999
server = make_server(ip, port, demo_app) # demo_app应用程序，可调用
server.serve_forever() # server.handle_request() 执行一次
```

WSGI 服务器作用

- 监听HTTP服务端口（TCPServer，默认端口80）
- 接收浏览器端的HTTP请求并解析封装成environ环境数据
- 负责调用应用程序，将environ数据和start_response方法两个参数传入给Application
- 将应用程序响应的正文封装成HTTP响应报文返回浏览器端

WSGI APP应用程序端

1、应用程序应该是一个可调用对象
Python中应该是函数、类、实现了`__call__`方法的类的实例
2、这个可调用对象应该接收两个参数

```
# 1 函数实现
def application(environ, start_response):
    pass

# 2 类实现
class Application:
    def __init__(self, environ, start_response):
        pass

# 3 类实现
class Application:
    def __call__(self, environ, start_response):
        pass
```

3、以上的可调用对象实现，都必须返回一个可迭代对象

```
res_str = b'www.magedu.com\n'

# 1 函数实现
def application(environ, start_response):
    start_response("200 OK", [('Content-Type', 'text/plain; charset=utf-8')])
    return [res_str]

# 2 类实现
class Application:
    def __init__(self, environ, start_response):
        self.env = environ
        self.start_response = start_response

    def __iter__(self): # 对象可迭代
        self.start_response('200 OK', [('Content-Type', 'text/plain; charset=utf-8')])
        yield res_str


# 3 类实现，可调用对象
class Application:
    def __call__(self, environ, start_response):
        start_response('200 OK', [('Content-Type', 'text/plain; charset=utf-8')])
        return [res_str]
```

environ和start_response这两个参数名可以是任何合法名，但是一般默认都是这2个名字。
应用程序端还有些其他的规定，暂不用关心

注意：第2、第3种实现调用时的不同

environ

environ是包含Http请求信息的dict字典对象

名称	含义
REQUEST_METHOD	请求方法，GET、POST等
PATH_INFO	URL中的路径部分
QUERY_STRING	查询字符串
SERVER_NAME，SERVER_PORT	服务器名、端口
HTTP_HOST	地址和端口
SERVER_PROTOCOL	协议
HTTP_USER_AGENT	UserAgent信息

```
CONTENT_TYPE = 'text/plain'
HTTP_HOST = '127.0.0.1:9999'
HTTP_USER_AGENT = 'Mozilla/5.0 (Windows; U; Windows NT 6.1; zh-CN) AppleWebKit/537.36 (KHTML,
like Gecko) Version/5.0.1 Safari/537.36'
PATH_INFO = '/'
QUERY_STRING = ''
REMOTE_ADDR = '127.0.0.1'
REMOTE_HOST = ''
REQUEST_METHOD = 'GET'
SERVER_NAME = 'DESKTOP-D34H5HF'
SERVER_PORT = '9999'
SERVER_PROTOCOL = 'HTTP/1.1'
SERVER_SOFTWARE = 'WSGIServer/0.2'
```

start_response

它是一个可调用对象。有3个参数，定义如下：
`start_response(status, response_headers, exc_info=None)`

参数名称	说明
status	状态码和状态描述，例如`200 OK`
response_headers	一个元素为二元组的列表，例如[('Content-Type', 'text/plain;charset=utf-8')]
exc_info	在错误处理的时候使用

start_response应该在返回可迭代对象之前调用，因为它返回的是Response Header。返回的可迭代对象是Response Body。

服务器端

服务器程序需要调用符合上述定义的可调用对象APP，传入environ、start_response，APP处理后，返回响应头和可迭代对象的正文，由服务器封装返回浏览器端。

```python
# 返回网页的例子
from wsgiref.simple_server import make_server

def application(environ, start_response):
    status = '200 OK'
    headers = [('Content-Type', 'text/html;charset=utf-8')]
    start_response(status, headers)
    # 返回可迭代对象
    html = '<h1>马哥教育欢迎你</h1>'.encode("utf-8")
    return [html]

ip = '127.0.0.1'
port = 9999
server = make_server(ip, port, application)
server.serve_forever() # server.handle_request() 一次
```

simple_server 只是参考用，不能用于环境。

测试用命令

```
$ curl -I http://192.168.142.1:9999/xxx?id=5
$ curl -X POST http://192.168.142.1:9999/yyy -d '{"x":2}'
```

-I 使用HEAD方法
-X 指定方法，-d传输数据

到这里就完成了一个简单的WEB 程序开发。
WSGI WEB服务器

- 本质上就是一个TCP服务器，监听在特定端口上
- 支持HTTP协议，能够将HTTP请求报文进行解析，能够把响应数据进行HTTP协议的报文封装并返回浏览器端。
- 实现了WSGI协议，该协议约定了和应用程序之间接口（参看PEP333，https://www.python.org/dev/peps/pep-0333/）

WSGI APP应用程序

- 遵从WSGI协议
- 本身是一个可调用对象
- 调用start_response，返回响应头部
- 返回包含正文的可迭代对象

WSGI 框架库往往可以看做增强的更加复杂的Application。

多人博客项目

分析

多人使用的博客系统。采用BS架构实现。

博客系统，需要用户管理、博文管理。

用户管理：注册、增删改查用户
博文管理：增删改查博文

需要数据库，本次使用Mysql 5.5， InnoDB引擎。

需要支持多用户登录，各自可以管理自己的博文（增删改查），管理是不公开的，但是博文是不需要登录就可以公开浏览的。

先不要思考过多的功能，先完成最小的核心需求代码。

数据库设计

创建数据库

```
CREATE DATABASE IF NOT EXISTS blog;
```

需要用户表、文章表

用户表user

id：唯一标识
name：用户姓名，描述性字段
email：电子邮箱，注册用信息，应该唯一，用作登录名。
password：存储密码。注意，不能存储明文，一般采用单向加密，例如MD5。

```
CREATE TABLE `user` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(48) NOT NULL,
  `email` varchar(64) NOT NULL,
  `password` varchar(128) NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `email` (`email`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

文章post

id：唯一标识
title：标题，描述字段
author：博文作者要求必须是注册用户，那这里就是用户userid了，这是外键
postdate：发布日期，日期类型
content：文章内容，博文内容可能很长，一般来说不会小于256个字符的。

一对多关系：一篇博文属于一个作者，一个作者有多篇博文

content字段的问题
1、博客选取什么字段类型?
2、多大合适?
3、博文中图片如何处理?
4、适合和其它字段放在同一张表吗?

思考
1、字段类型
博文一般很长，不可能只有几百个字符，需要大文本字段。MySQL中，选择TEXT类型，而不是char或者varchar类型。

2、大小
text类型是65535个字符，如果不够用，选择longtext，有2^32-1个字符长度，足够使用了。

3、图片存储
博文就像HTML一样，图片是通过路径信息将图片是嵌入在内容中的，所以保存的内容还是字符串。
图片来源有2种：
外链，通过URL链接访问，本站不用存储该图片，但容易引起盗链问题。
本站存储，需要提供博文的在线文本编辑器，提供图片上传到网站存储，并生成图片URL，这个URL嵌入博客正文中。不会有盗链的问题，但要解决众多图片存储问题、水印问题、临时图片清理、在线压缩问题等等。
本次博客项目不实现图片功能。

4、字段考虑
content字段存储文本类型大字段，一般不和数据频繁查询的字段放在一张表中，需要拆到另一张表中。

```
CREATE TABLE `post` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `title` varchar(256) NOT NULL,
  `author_id` int(11) NOT NULL,
  `postdate` datetime NOT NULL,
  PRIMARY KEY (`id`),
  KEY `author_id` (`author_id`),
  CONSTRAINT `fk_post_user` FOREIGN KEY (`author_id`) REFERENCES `user` (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

```
CREATE TABLE `content` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `content` text NOT NULL,
  PRIMARY KEY (`id`),
  CONSTRAINT `fk_content_post` FOREIGN KEY (`id`) REFERENCES `post` (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

注：这里的SQL脚本本次不要用来生成表，使用ORM工具来创建表，用来检查实体类构建是否正确。

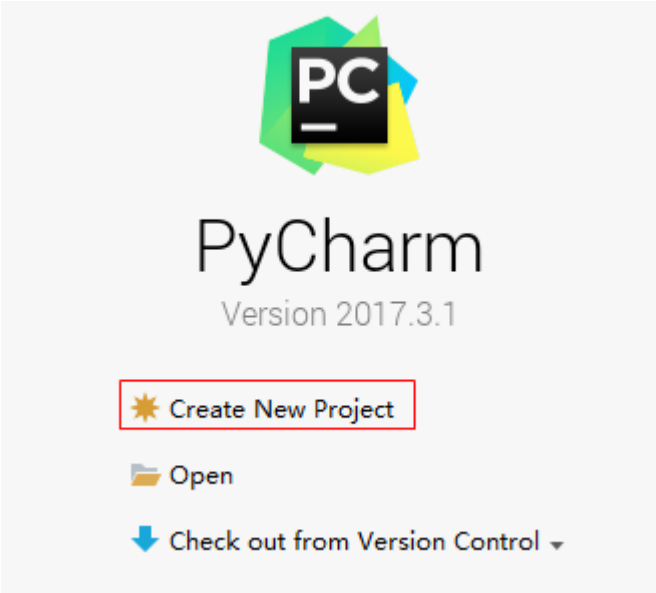
用户完成的功能有：
登录、注册、登出
user表基本满足

博客
用户发文、文章列表、文章详情
post、content表基本满足。

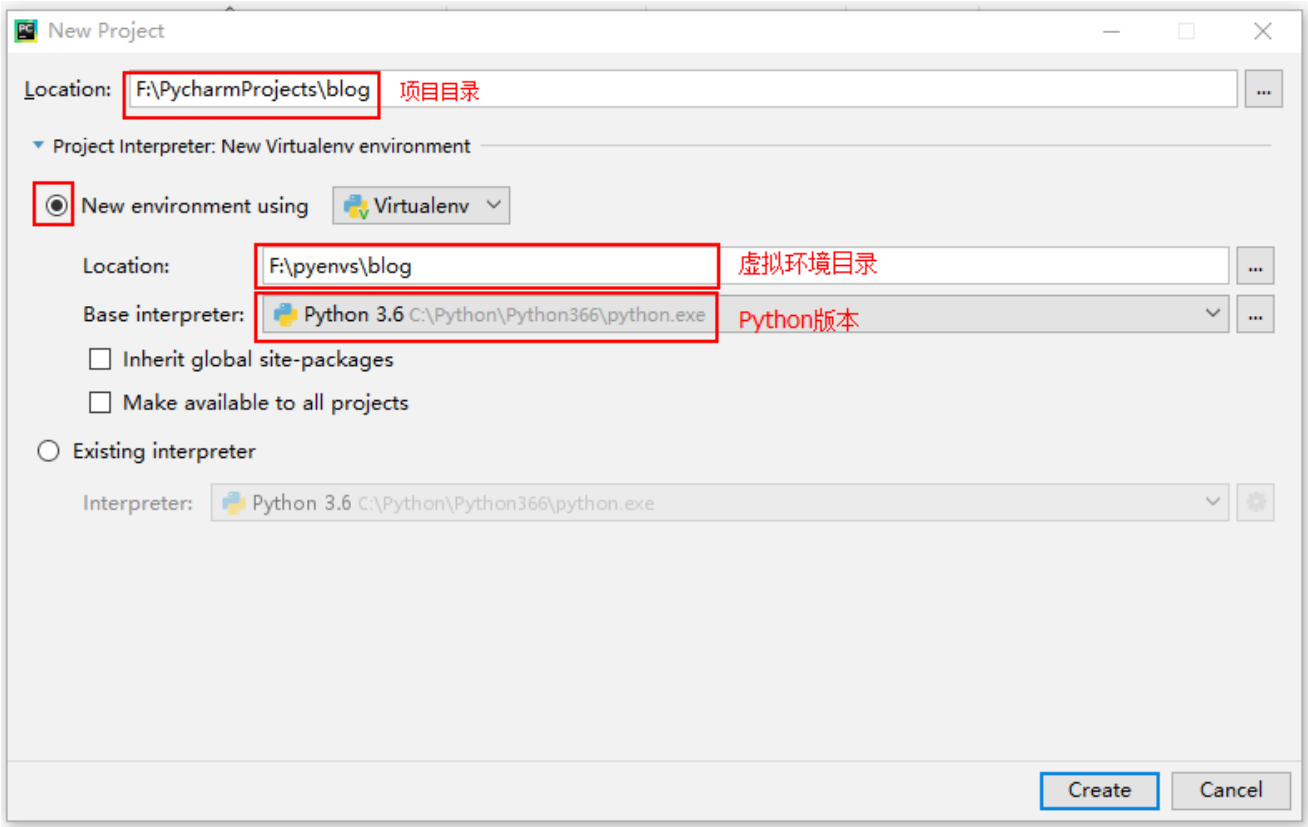
项目

项目构建

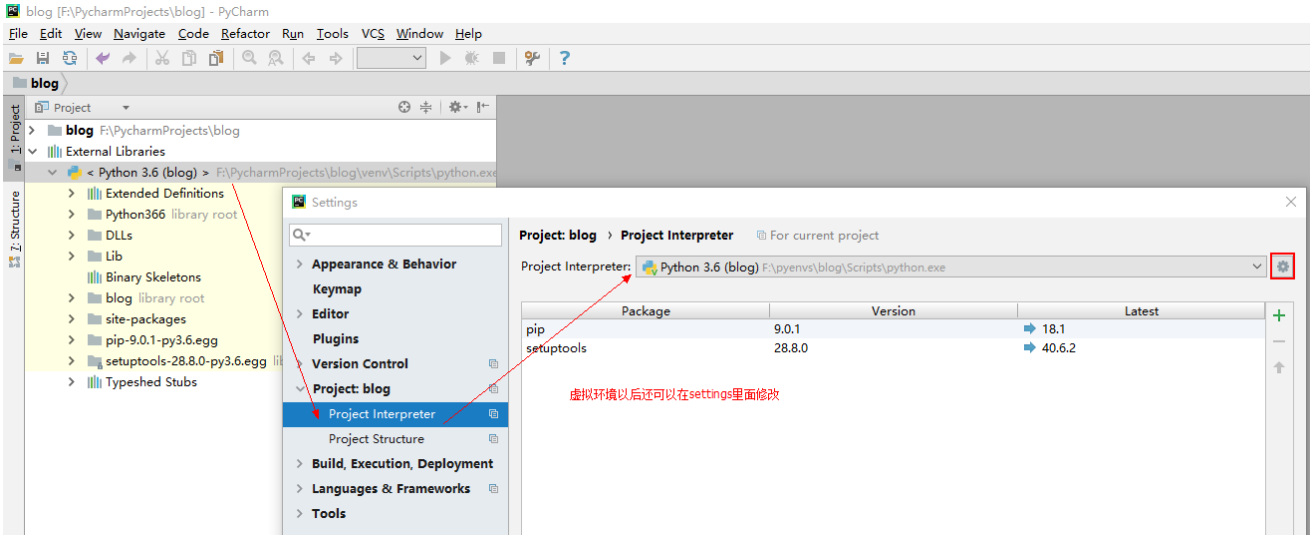
在Pycharm中新建一个项目



使用虚拟环境，Python挑选喜欢的版本。



注：Pycharm中可以通过settings重新设置虚拟环境。



本次项目使用Django开发后台，下面就开始Django之旅。

多人博客项目

概述

Django采用MVC架构设计的开源的WEB快速开发框架。

优点：
能够快速开发，自带ORM, Template, Form, Auth核心组件
MVC设计模式
实用的管理后台Admin
简洁的url设计
周边插件丰富

缺点：
框架重，因为东西大而全
同步阻塞

所以Django的设计目标就是一款大而全，便于企业快速开发项目的框架，因此企业应用较广。

Django版本

Django Version	Python Version
1.8	2.7， 3.2， 3.3，3.4， 3.5
1.9， 1.10	2.7， 3.4， 3.5
1.11	2.7， 3.4， 3.5，3.6，3.7
2.0	3.4， 3.5， 3.6，3.7
2.1，2.2	3.5， 3.6， 3.7

安装Django

Python 使用3.6.x

Django的下载地址 https://www.djangoproject.com/download/

Python版本依赖，参看https://docs.djangoproject.com/en/1.11/faq/install/#faq-python-version-support

Django version	Python版本
1.8	2.7, 3.2 (until the end of 2016), 3.3, 3.4, 3.5
1.9, 1.10	2.7, 3.4, 3.5
1.11(LTS)	2.7, 3.4, 3.5, 3.6, 3.7 (added in 1.11.17)
2.0	3.4, 3.5, 3.6, 3.7
2.1	3.5, 3.6, 3.7

```
$ pip install django==1.11.20
```

本次使用Django 1.11版本，它也是长期支持版本LTS，请在虚拟环境中安装。

在虚拟环境路径中，Lib/site-packages/django/bin下有一个django-admin.py，一起从它开始。

```
$ django-admin --version
$ django-admin

Type 'django-admin help <subcommand>' for help on a specific subcommand.

Available subcommands:

[django]
    check
    compilemessages
    createcachetable
    dbshell
    diffsettings
    dumpdata
    flush
    inspectdb
    loaddata
    makemessages
    makemigrations
    migrate
    runserver
    sendtestemail
    shell
    showmigrations
    sqlflush
    sqlmigrate
    sqlsequencereset
    squashmigrations
    startapp
    startproject
    test
    testserver
```

```
$ django-admin startproject --help
```

注意：本文如若未特殊声明，所有的命令操作都在项目根目录下

创建django项目

`$ django-admin startproject blog .`
上句命令就在当前项目根目录中构建了Django项目的初始文件。`点`代表项目根目录。

```
F:\CLASSES\TPROJECTS\BLOG10
├─ manage.py
└─ blog
    ├─  settings.py
    ├─  urls.py
    ├─  wsgi.py
    └─  __init__.py
```

重要文件说明

- manage.py：本项目管理的命令行工具。应用创建、数据库迁移等都使用它完成
- blog/settings.py：本项目的配置文件。数据库参数等
- blog/urls.py：URL路径映射配置。默认情况下，只配置了/admin的路由。
- blog/wsgi：定义WSGI接口信息。一般无须改动。

数据库配置

使用数据库，需要修改默认的数据库配置。
在主项目的settings.py下的DATABASES。默认使用的sqlite，修改为mysql。

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'blog',
        'USER': 'wayne',
        'PASSWORD': 'wayne',
        'HOST': '192.168.142.140',
        'PORT': '3306',
    }
}
```

HOST 数据库主机。缺省是空字符串，代表localhost。如果是'/'开头表示使用Unix Socket连接。
POST 端口
USER 用户名
PASSWORD 密码
NAME 库名
OPTIONS 选项，字典，参考MySQL文档

数据库引擎ENGINE
内建的引擎有

- 'django.db.backends.postgresql'
- 'django.db.backends.mysql'
- 'django.db.backends.sqlite3'
- 'django.db.backends.oracle'

MySQL数据库驱动

https://docs.djangoproject.com/en/2.0/ref/databases/

Django支持MySQL 5.5+

Django官方推荐使用本地驱动mysqlclient 1.3.7 +

`$ pip install mysqlclient`

```
windows下安装错误 error: Microsoft Visual C++ 14.0 is required.解决方法
1、下载Visual C++ Redistributable Packages 2015、2017安装，但是即使安装后，确实看到了V14库，也不保证安装mysqlclient就成功
2、直接安装编译好的wheel文件
mysqlclient-1.3.13-cp35-cp35m-win_amd64.whl , python 3.5使用
mysqlclient-1.3.13-cp36-cp36m-win_amd64.whl , python 3.6使用
mysqlclient-1.4.2-cp37-cp37m-win_amd64.whl,   python 3.7使用

$ pip install mysqlclient-1.3.13-cp35-cp35m-win_amd64.whl
参考 https://stackoverflow.com/questions/29846087/microsoft-visual-c-14-0-is-required-unable-to-find-vcvarsall-bat
```

下载地址
https://www.lfd.uci.edu/~gohlke/pythonlibs/

创建应用

创建用户应用`$ python manage.py startapp user`

该应用完成以下功能

- 用户注册
- 用户登录

创建应用后，项目根目录下产生一个user目录，有如下文件：

- admin.py：管理站点模型的声明文件
- models.py：模型层Model类定义
- views.py：定义URL响应函数
- migrations包：数据迁移文件生成目录
- apps.py：应用的信息定义文件

注册应用

在settings.py中，增加user应用。
目的是为了**后台管理**admin使用，或**迁移**migrate使用

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'user',
]
```

模型Model

字段类	说明
AutoField	自增的整数字段。 如果不指定，django会为模型类自动增加主键字段
BooleanField	布尔值字段，True和False 对应表单控件CheckboxInput
NullBooleanField	比BooleanField多一个null值
CharField	字符串，max_length设定字符长度 对应表单控件TextInput
TextField	大文本字段，一般超过4000个字符使用 对应表单控件Textarea
IntegerField	整数字段
BigIntegerField	更大整数字段，8字节
DecimalField	使用Python的Decimal实例表示十进制浮点数。max_digits总位数，decimal_places小数点后的位数
FloatField	Python的Float实例表示的浮点数
DateField	使用Python的datetime.date实例表示的日期 auto_now=False每次修改对象自动设置为当前时间。 auto_now_add=False对象第一次创建时自动设置为当前时间。auto_now_add、auto_now、default互斥 对应控件为TextInput，关联了一个js编写的日历控件
TimeField	使用Python的datetime.time实例表示的时间，参数同上
DateTimeField	使用Python的datetime.datetime实例表示的时间，参数同上
FileField	一个上传文件的字段
ImageField	继承了FileField的所有属性和方法，但是对上传的文件进行校验，确保是一个有效的图片

字段选项

值	说明
db_column	表中字段的名称。如果未指定，则使用属性名
primary_key	是否主键
unique	是否是唯一键
default	缺省值。这个缺省值不是数据库字段的缺省值，而是新对象产生的时候被填入的缺省值
null	表的字段是否可为null，默认为False
blank	Django表单验证中，是否可以不填写，默认为False
db_index	字段是否有索引

关系类型字段类

类	说明
ForeignKey	外键，表示一对多 ForeignKey('production.Manufacturer') 自关联ForeignKey('self')
ManyToManyField	表示多对多。
OneToOneField	表示一对一。

一对多时，自动创建会增加_id后缀。

- 从一访问多，使用`对象.小写模型类_set`
- 从一访问一，使用`对象.小写模型类`

访问id `对象.属性_id`

创建User的Model类

- 基类 models.Model
- 表名不指定默认使用`<appname>_<model_name>`。使用Meta类修改表名

```
from django.db import models

# Create your models here.

class User(models.Model):
    class Meta:
        db_table = 'user'
    id = models.AutoField(primary_key=True)
    name = models.CharField(max_length=48, null=False)
    email = models.CharField(max_length=64, unique=True, null=False)
    password = models.CharField(max_length=128, null=False)

    def __repr__(self):
```

```
        return '<user {} {}>'.format(self.id, self.name)

    __str__ = __repr__
```

Meta类的使用，参考 https://docs.djangoproject.com/en/1.11/ref/models/options/#django.db.models.Options.db_table

迁移Migration

迁移：从模型定义生成数据库的表

1、生成迁移文件``

```
$ python manage.py makemigrations
Migrations for 'user':
  user\migrations\0001_initial.py
    - Create model User

生成如下文件
user
├─ migrations
    ├─  0001_initial.py
    └─  __init__.py
```

修改过Model类，还需要调用makemigrations，然后migrate，迁移文件的序号会增加。

注意：
迁移的应用必须在settings.py的INSTALLED_APPS中注册。
不要随便删除这些迁移文件，因为后面的改动都是要依据这些迁移文件的。

```
# 0001_initial.py 文件内容如下
class Migration(migrations.Migration):

    initial = True

    dependencies = [
    ]

    operations = [
        migrations.CreateModel(
            name='User',
            fields=[
                ('id', models.AutoField(primary_key=True, serialize=False)),
                ('name', models.CharField(max_length=48)),
                ('email', models.CharField(max_length=64, unique=True)),
                ('password', models.CharField(max_length=128)),
            ],
            options={
                'db_table': 'user',
            },
        ),
    ]
```

2、执行迁移生成数据库的表

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

执行了迁移，还同时生成了admin管理用的表。

查看数据库，user表创建好了，字段设置完全正确。

Django后台管理

1、创建管理员

管理员用户名admin
密码adminadmin

```
$ manage.py createsuperuser
Username (leave blank to use 'wayne'):admin
Email address:
Password:
Password (again):
Superuser created successfully.
```

2、本地化

settings.py中设置语言、时区

语言名称可以查看 django\contrib\admin\locale 目录

```
LANGUAGE_CODE = 'zh-Hans' #'en-us'
USE_TZ = True
TIME_ZONE = 'Asia/Shanghai' #'UTC'
```

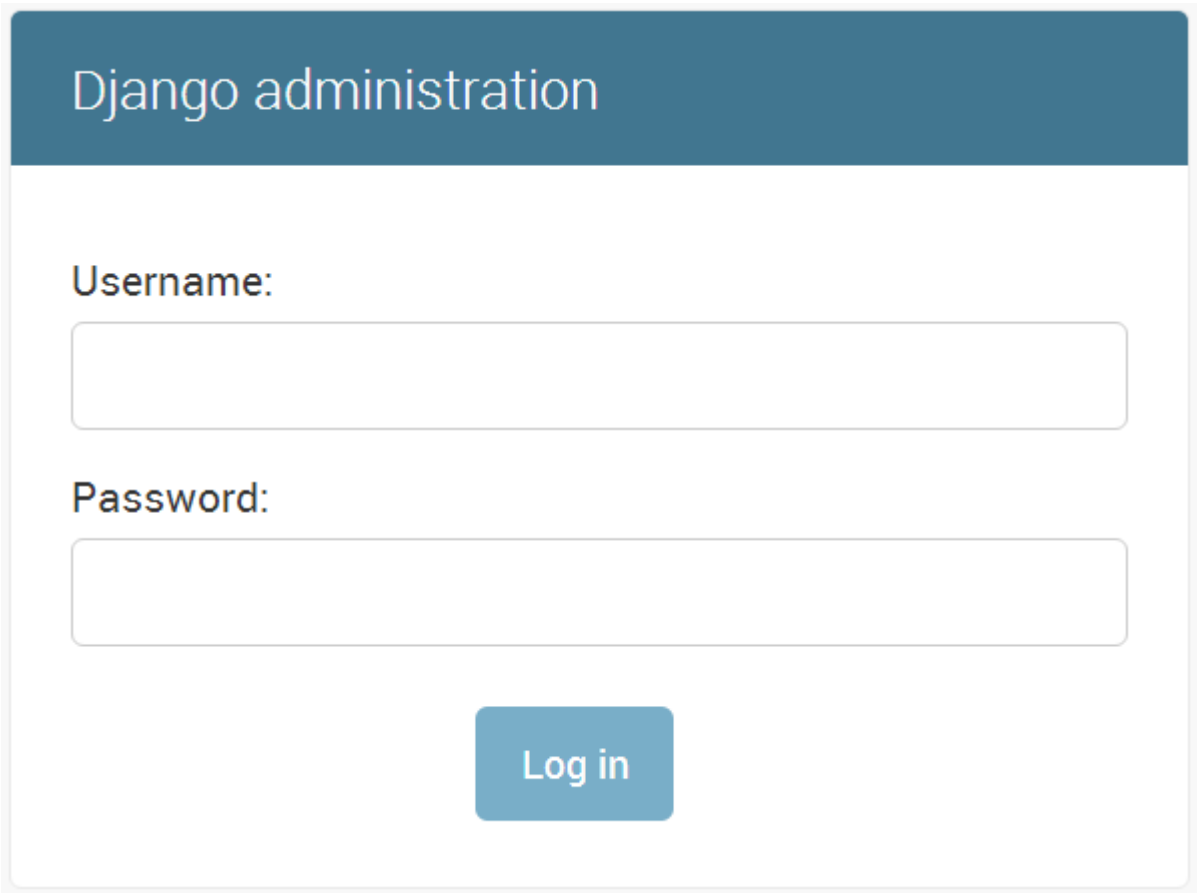
3、启动WEB Server

```
$ python manage.py runserver
```

默认启动8000端口

4、登录后台管理

后台登录地址`http://127.0.0.1:8000/admin`



5、注册应用模块

在user应用的admin.py添加

```
from django.contrib import admin
from .models import User
# Register your models here.

admin.site.register(User) # 注册
```

user就可以在后台进行增删改了。

路由**

路由功能就是实现URL模式匹配和处理函数之间的映射。

路由配置要在项目的urls.py中配置，也可以多级配置，在每一个应用中，建立一个urls.py文件配置路由映射。

url函数
url(regex, view, kwargs=None, name=None)，进行模式匹配
regex: 正则表达式，与之匹配的 URL 会执行对应的第二个参数 view
view: 用于执行与正则表达式匹配的 URL 请求
kwargs: 视图使用的字典类型的参数
name: 用来反向获取 URL

urls.py内容如下

```
from django.conf.urls import url
from django.contrib import admin

from django.http import HttpRequest, HttpResponse
def index(request:HttpRequest):
    """视图函数：请求进来返回响应"""
    return HttpResponse(b'welcome to magedu.com')

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

url(r'^index/$', index)
http://127.0.0.1:8000/index/ 可以访问
http://127.0.0.1:8000/index 可以访问，但会补一个/
url(r'^index$', index)
http://127.0.0.1:8000/index 可以访问
http://127.0.0.1:8000/index/ 不可以访问

请求信息测试和JSON响应

```
from django.http import HttpRequest, HttpResponse, JsonResponse
def index(request:HttpRequest):
    """视图函数：请求进来返回响应"""
    d = {}
    d['method'] = request.method
    d['path'] = request.path
    d['path_info'] = request.path_info
    d['GETparams'] = request.GET

    return JsonResponse(d)
```

在项目中首页多数使用HTML显示，为了加载速度快，一般多使用静态页面。如果首页内容多，还有部分数据需要变化，将变化部分使用AJAX技术从后台获取数据。

本次，为了学习模板技术，只将首页采用Django的模板技术实现。

模板

如果使用react实现前端页面，其实Django就没有必须使用模板，它其实就是一个后台服务程序，接收请求，响应数据。接口设计就可以是纯粹的Restful风格。

模板的目的就是为了可视化，将数据按照一定布局格式输出，而不是为了数据处理，所以一般不会有复杂的处理逻辑。模板的引入实现了业务逻辑和显示格式的分离。这样，在开发中，就可以分工协作，页面开发完成页面布局设计，后台开发完成数据处理逻辑的实现。

Python的模板引擎默认使用Django template language (DTL)构建

模板配置

在settings.py中，设置模板项目的路径

```
BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__))) # 这一句取项目根目录

TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR, 'templates')],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

- DIRS：列表，定义模板文件的搜索路径顺序。os.path.join(BASE_DIR, 'templates')即项目根目录下templates目录，请构建这个目录。
- APP_DIRS：是否运行在每个已经安装的应用中查找模板。应用自己目录下有templates目录，例如django/contrib/admin/templates。如果应用需要可分离、可重用，建议把模板放到应用目录下
- BASE_DIR 是 项目根目录，os.path.join(BASE_DIR, 'templates')就是在manage.py这一层建立一个目录templates。这个路径就是以后默认找模板的地方。

模板渲染

模板页

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>马哥教育Django首页</title>
</head>
<body>
我是模板，数据是{{content}}
</body>
</html>
```

将模板index.html放入到templates目录下

模板处理

2个步骤
1、加载模板
模板是一个文件，需要从磁盘读取并加载。要将模板放置在
2、渲染
模板需要使用内容数据来渲染，生成HTML文件内容

```
from django.template import loader

def index(request:HttpRequest):
    """视图函数：请求进来返回响应"""
    template = loader.get_template('index.html') # 加载器模块搜索模板并加载它
    print(template.origin) # 显示模板路径
    context = {'content':'www.magedu.com'} # 数据字典
    return HttpResponse(template.render(context, request))
```

render快捷渲染函数

上面2个步骤代码编写繁琐，Django提供了对其的封装——快捷函数render。
render(request, template_name, context=None)
返回HttpResponse对象
template_name 模板名称
context 数据字典
render_to_string()是其核心方法，其实就是拿数据替换HTML中的指定位置后返回一个字符串

```
from django.shortcuts import render

def index(request:HttpRequest):
    """视图函数：请求进来返回响应"""
    return render(request, 'index.html',{'content': 'www.magedu.com'})
```

使用浏览器访问首页，可以正常显示

DTL语法

- 变量
- 标签
- 注释
- 过滤器

1 变量

`语法 {{ variable }}`

变量名由字母、数字、下划线、点号组成。

点号使用的时候，例如foo.bar，遵循以下顺序：

1. 字典查找，例如foo["bar"]，把foo当做字典，bar当做key
2. 属性或方法的查找，例如foo.bar，把foo当做对象，bar当做属性或方法
3. 数字索引查找，例如foo[bar]，把foo当做列表一样，使用索引访问

```
def index(request:HttpRequest):
    """视图函数：请求进来返回响应"""
    my_dict = {
        'a':100,
        'b':0,
        'c':list(range(10,20)),
        'd':'abc', 'date':datetime.datetime.now()
    }
    context = {'content':'www.magedu.com', 'my_dict':my_dict}
    return render(request, 'index.html', context)
```

如果**变量未能找到，则缺省插入空字符串''**
在模板中调用方法，**不能加小括号**，自然也不能传递参数。
{{my_dict.a}}符合第一条，当做字典的key就可以访问到了
{{ my_dict.keys }} 这样是对的，不能写成{{ my_dict.keys() }}。符合第二条，当做my_dict对象的属性或方法。

2 模板标签

if/else 标签
基本语法格式如下：

```
{% if condition %}
     ... display
{% endif %}
```

或者：

```
{% if condition1 %}
   ... display 1
{% elif condition2 %}
   ... display 2
{% else %}
   ... display 3
{% endif %}
```

条件也支持and、or、not
注意，因为这些标签是断开的，所以不能像Python一样使用缩进就可以表示出来，必须有个结束标签，例如endif、endfor。

for 标签
https://docs.djangoproject.com/en/2.0/ref/templates/builtins/#for

```
<ul>
{% for athlete in athlete_list %}
    <li>{{ athlete.name }}</li>
{% endfor %}
</ul>

{%  for person  in  person_list %}
<li>    {{  person.name }}  </li>
{%  endfor  %}
```

变量	说明
forloop.counter	当前循环从1开始的计数
forloop.counter0	当前循环从0开始的计数
forloop.revcounter	从循环的末尾开始倒计数到1
forloop.revcounter0	从循环的末尾开始到计数到0
forloop.first	第一次进入循环
forloop.last	最后一次进入循环
forloop.parentloop	循环嵌套时，内层当前循环的外层循环

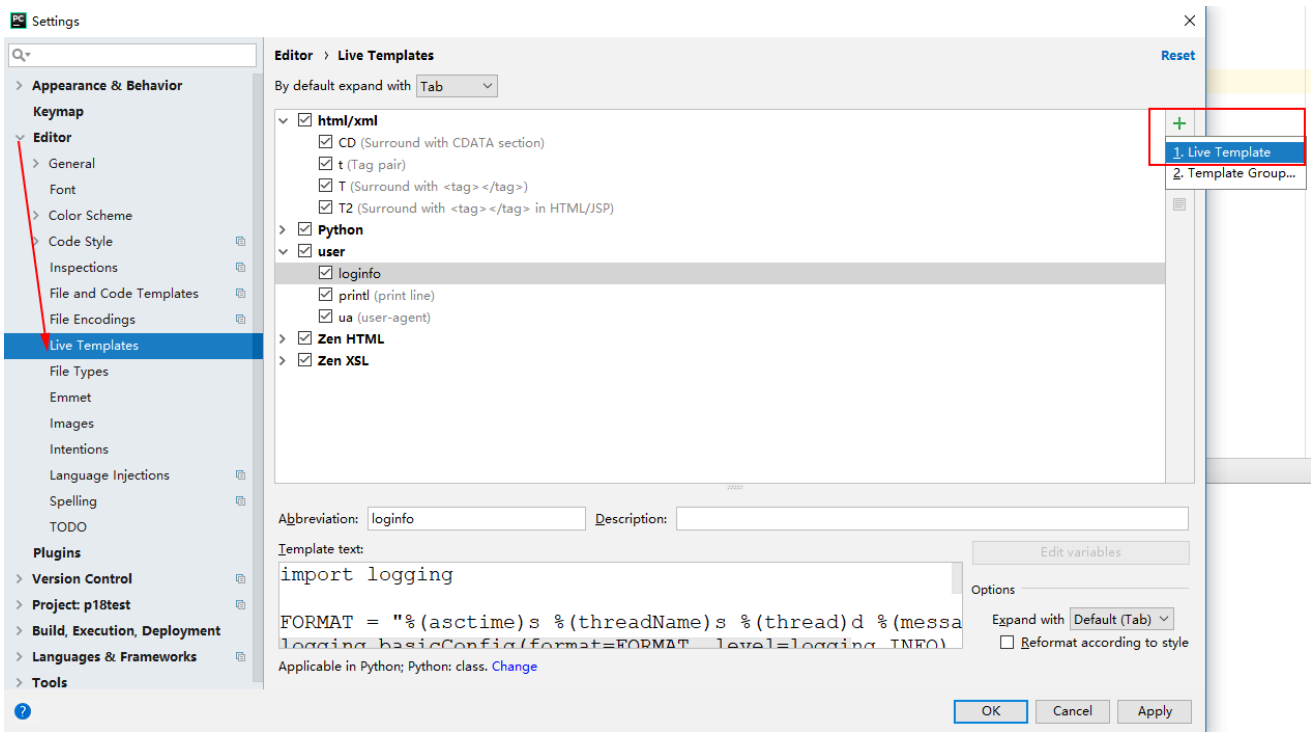
给标签增加一个 reversed 使得该列表被反向迭代：

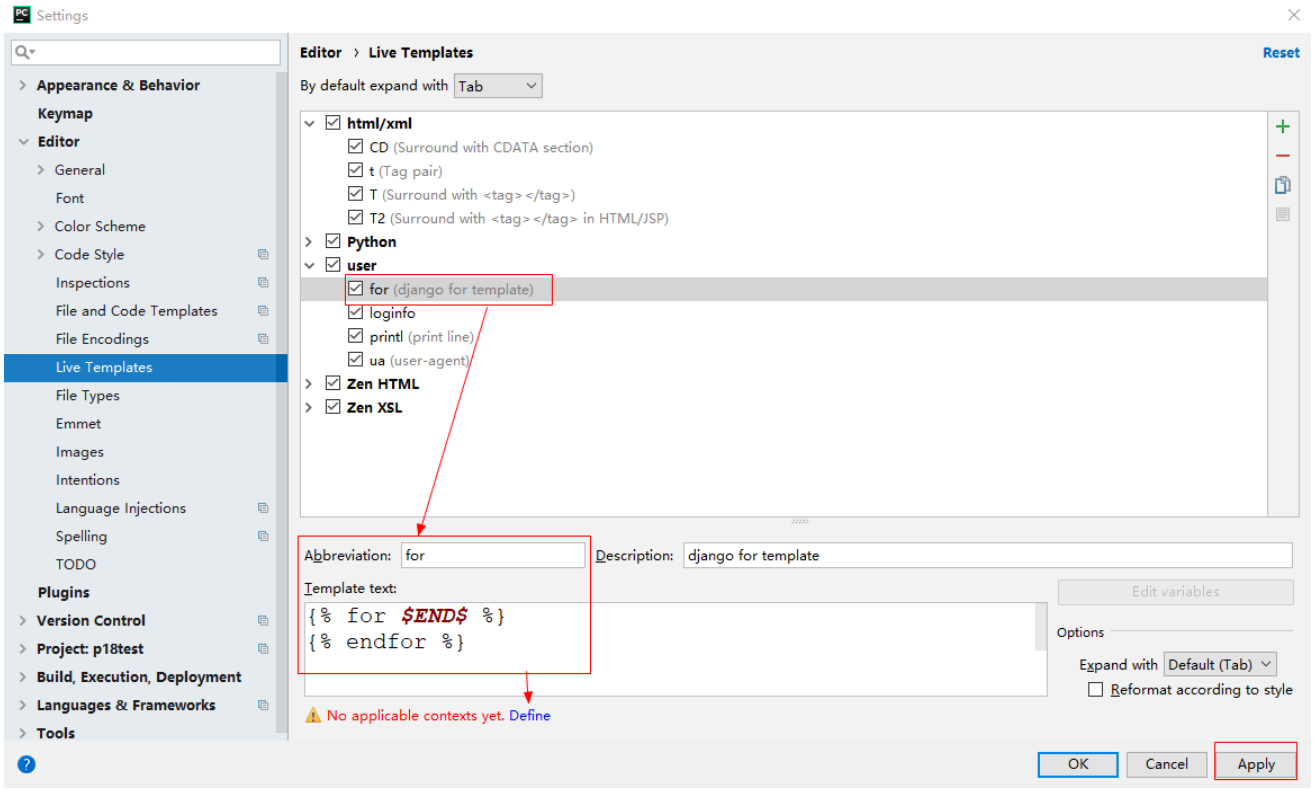
```
{% for athlete in athlete_list reversed %}
...
{% empty %}
... 如果被迭代的列表是空的或者不存在，执行empty
{% endfor %}
```

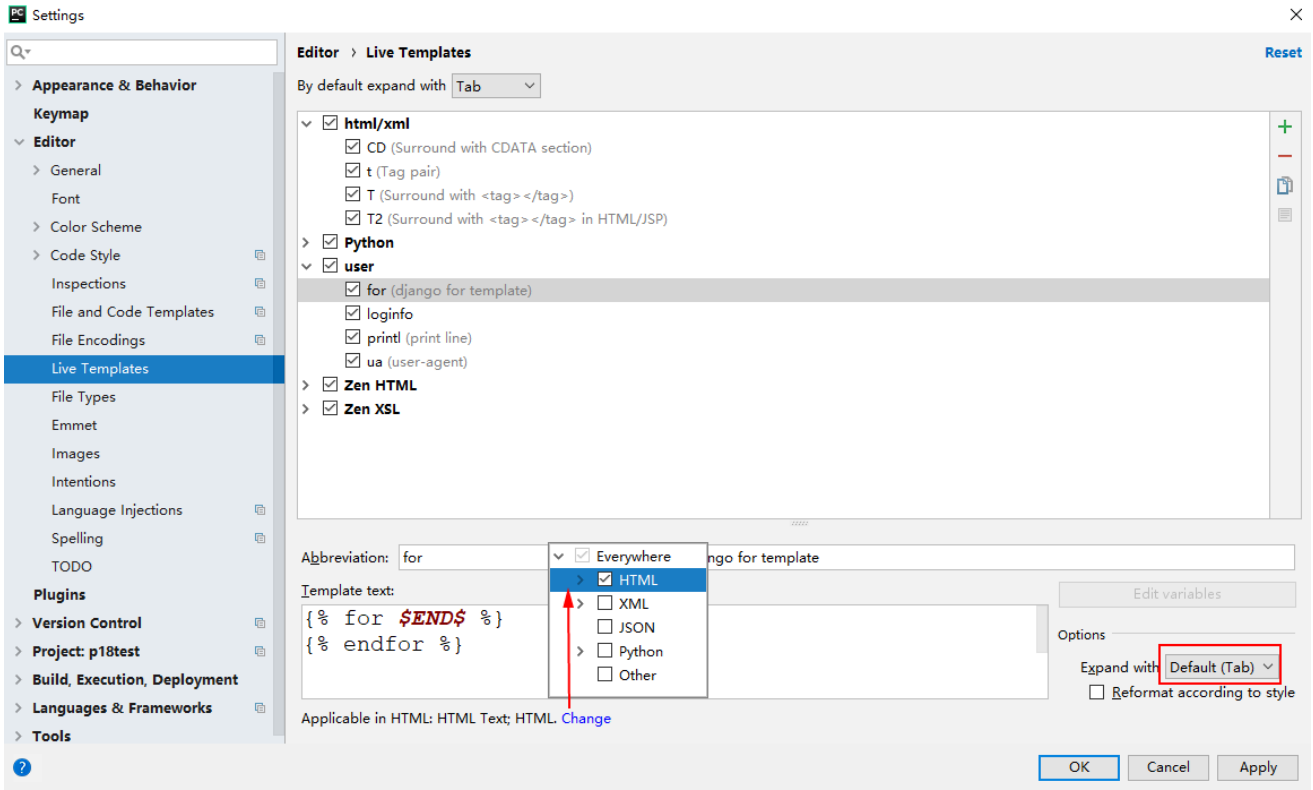
可以嵌套使用 {% for %} 标签：

```
{% for athlete in athlete_list %}
    <h1>{{ athlete.name }}</h1>
    <ul>
    {% for sport in athlete.sports_played %}
        <li>{{ sport }}</li>
    {% endfor %}
    </ul>
{% endfor %}
```

Pycharm模板定义







按照上述方法可以定义诸多Django模板标签。

testfor.html模板

```
<!DOCTYPE html>
<html lang="en">
<head>
```

```
    <meta charset="UTF-8">
    <title>测试for</title>
</head>
<body>
字典是dict(zip('abced', range(1,6)))
<ul>
{% for k,v in dct.items %}
    <li>{{forloop.counter}} {{k}} {{v}}</li>
{% endfor %}
</ul>
<hr>

<ul>
{% for k,v in dct.items %}
    <li>{{forloop.counter0}} {{k}} {{v}}</li>
{% endfor %}
</ul>
<hr>

<ul>
{% for k,v in dct.items %}
    {{ forloop.first }}
    {{ forloop.last }}
    <li>{{forloop.revcounter0}} {{k}} {{v}}</li>
{% endfor %}
</ul>
<hr>

<ul>
{% for k,v in dct.items %}
    <li>{{forloop.revcounter}} {{k}} {{v}}</li>
{% endfor %}
</ul>
<hr>

</body>
</html>
```

ifequal/ifnotequal 标签
{% ifequal %} 标签比较两个值，当他们相等时，显示在 {% ifequal %} 和 {% endifequal %} 之中所有的值。
下面的例子比较两个模板变量 user 和 currentuser :

```
{% ifequal user currentuser %}
    <h1>Welcome!</h1>
{% endifequal %}
```

和 {% if %} 类似， {% ifequal %} 支持可选的 {% else%} 标签:

```
{% ifequal section 'sitenews' %}
    <h1>Site News</h1>
{% else %}
    <h1>No News Here</h1>
{% endifequal %}
```

其他标签

csrf_token 用于跨站请求伪造保护，防止跨站攻击的。
{% csrf_token %}

3 注释标签

单行注释 {# #}。
多行注释 {% comment %} ... {% endcomment %}.

```
{# 这是一个注释 #}

{% comment %}
这是多行注释
{% endcomment %}.
```

4 过滤器

模板过滤器可以在变量被显示前修改它。

语法
`{{ 变量|过滤器 }}`
过滤器使用管道字符 | ，例如 `{{ name|lower }}` ， `{{ name }}` 变量被过滤器 lower 处理后，文档大写转换文本为小写。

过滤管道可以被**套接**，一个过滤器管道的输出又可以作为下一个管道的输入。
例如 `{{ my_list|first|upper }}` ，将列表第一个元素并将其转化为大写。

过滤器传参
有些过滤器可以传递参数，过滤器的参数跟随冒号之后并且总是以双引号包含。
例如： `{{ bio|truncatewords:"30" }}` ，截取显示变量 bio 的前30个词。
`{{ my_list|join:"," }}` ，将my_list的所有元素使用,逗号连接起来

其他过滤器

过滤器	说明	举例
first	取列表第一个元素	
last	取列表最后元素	
yesno	变量可以是True、False、None yesno的参数给定逗号分隔的三个值，返回3个值中的一个。 True对应第一个 False对应第二个 None对应第三个 如果参数只有2个，None等效False处理	{{ value | yesno:"yeah,no,maybe" }}
add	加法。参数是负数就是减法	数字加{{ value | add:"100"}} 列表合并{{mylist | add:newlist}}
divisibleby	能否被整除	{{ value | divisibleby:"3" }}能被3整除返回True
addslashes	在反斜杠、单引号或者双引号前面加上反斜杠	{{ value | addslashes }}
length	返回变量的长度	{% if my_list | length > 1 %}
default	变量等价False则使用缺省值	{{ value | default:"nothing" }}
default_if_none	变量为None使用缺省值	{{ value | default_if_none:"nothing" }}
date	格式化 date 或者 datetime 对象	实例：{{my_dict.date|date:'Y n j'}} Y 2000 年 n 1~12 月 j 1~31 日

时间的格式字符查看 https://docs.djangoproject.com/en/2.0/ref/templates/builtins/#date

模板习题

1、奇偶行列表输出

使用下面字典my_dict的c的列表，在模板网页中列表ul输出多行数据

- 要求奇偶行颜色不同
- 每行有行号（从1开始）
- 列表中所有数据都增大100

```python
from django.http import HttpResponse, HttpRequest
from django.shortcuts import render

def index(request:HttpRequest):
    """视图函数：请求进来返回响应"""
    my_dict = {
        'a':100,
        'b':0,
        'c':list(range(10,20)),
        'd':'abc', 'date':datetime.datetime.now()
    }
    context = {'content':'www.magedu.com', 'my_dict':my_dict}
    return render(request, 'index.html', context)
```

模板中如何实现?

2、打印九九方阵

```
1*1=1 1*2=2 ... 1*9=9
...
9*1=1 9*2=18... 9*9=81
```

使用把上面所有的数学表达式放到HTML表格对应的格子中。如果可以，请实现奇偶行颜色不同。

模板习题

1、奇偶行列表输出

使用下面字典my_dict的c的列表，在模板网页中列表ul输出多行数据

- 要求奇偶行颜色不同
- 每行有行号（从1开始）
- 列表中所有数据都增大100

```
from django.http import HttpResponse, HttpRequest
from django.shortcuts import render

def index(request:HttpRequest):
    """视图函数：请求进来返回响应"""
    my_dict = {
        'a':100,
        'b':0,
        'c':list(range(10,20)),
        'd':'abc', 'date':datetime.datetime.now()
    }
    context = {'content':'www.magedu.com', 'my_dict':my_dict}
    return render(request, 'index.html', context)
```

模板代码如下

```
<ul>
{%for line in my_dict.c %}
    <li style='color:{{forloop.counter|divisibleby:"2"|yesno:"red,blue"}}'>
        {{forloop.counter}} {{line|add:"100"}}
    </li>
{%endfor%}
</ul>
```

2、打印九九方阵

```
1*1=1 1*2=2 ... 1*9=9
...
9*1=1 9*2=18... 9*9=81
```

使用把上面所有的数学表达式放到HTML表格对应的格子中

方法一、由视图函数提供数据

为了简单，直接准备一个排好输出顺序的列表

```
from django.http import HttpResponse, HttpRequest
from django.shortcuts import render

def index(request:HttpRequest):
    data = ['{}*{}={}'.format(j, i, j*i) for i in range(1, 10) for j in range(1, 10)]
    return render(request, 'matrix.html', {'data':data})
```

matrix.html模板如下

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>九九方阵</title>
</head>
<body>
<table>
    {% for x in data %}
    {% if forloop.counter0|divisibleby:9 %}<tr>{% endif %}
        <td>{{x}}</td>
    {% if forloop.counter|divisibleby:9 %}</tr>{% endif %}
    {% endfor %}
</table>
</body>
</html>
```

方法二、内建标签 `widthratio`

widthratio 本意是计算宽度比率的。

widthratio 用法：`{% widthratio value max_value max_width %}`

举例
`{% widthratio 175 200 100 %}`
value是175，max_value是200，max_width是100。
因此，175/200=0.875得到数值比值，再利用比率计算出宽度0.875*100=87.5，四舍五入到88

{% widthratio i 1 j as product %} 别名就可以在后面引用这个变量product

```
from django.http import HttpResponse, HttpRequest
from django.shortcuts import render

def index(request:HttpRequest):
    context = {'loop':list(range(1,10))}
    return render(request, 'matrix.html', context)
```

matrix.html模板如下

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>九九方阵</title>
</head>
<body>
九九方阵表格打印--widthratio
<hr>
<table>
    {% for i in loop %}
    <tr style="background-color:{{forloop.counter|divisibleby:2|yesno:'#F0F0F0,#CCC'}}">
        {% for j in loop %}
        <td>
            {{forloop.counter}}*{{forloop.parentloop.counter}}={% widthratio forloop.counter 1
forloop.parentloop.counter %}
        </td>
        {% endfor %}
    </tr>
    {% endfor %}
</table>
<hr>
</body>
</html>
```

使用cycle标签来替换奇偶行变色代码。cycle标签，就是轮询所有其后给出的所有参数。

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>九九方阵</title>
</head>
<body>
九九方阵表格打印--widthratio

<hr>
<table>
    {% for i in loop %}
    <tr style="background-color:{% cycle '#CCC' '#F0F0F0' '#FFF' %}">
        {% for j in loop %}
        <td>
            {{forloop.counter}}*{{forloop.parentloop.counter}}={% widthratio forloop.counter 1
forloop.parentloop.counter %}
        </td>
        {% endfor %}
    </tr>
    {% endfor %}
</table>
<hr>
</body>
</html>
```

方法三、自定义filter
1）构建自定义的模板的包和模块

在应用user下构建templatetags包，一定要有`__init__.py`文件。

构建自己的filter的模块，这里起名为`myfilters.py`。其中代码如下

```
from django.template import Library

register = Library()

@register.filter('multiply') # 使用装饰器注册
def multiply(a, b):# 只能1到2个参数
    return int(a)*int(b)
```

2）定义模板
matrix.html模板中要load myfilters模块，使用已经注册的filter multiply，模板内容如下

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>九九方阵</title>
</head>
<body>
九九方阵表格打印
{% load myfilters %}
<hr>
<table>
    {% for i in loop %}
    <tr style="background-color:{{forloop.counter|divisibleby:2|yesno:'#F0F0F0,#CCC'}}">
        {% for j in loop %}
        <td>
            {{forloop.counter}}*{{forloop.parentloop.counter}}=
{{forloop.counter|multiply:forloop.parentloop.counter}}
        </td>
        {% endfor %}
    </tr>
    {% endfor %}
</table>
<hr>
</body>
</html>
```

3）使用模板

```
from django.http import HttpResponse, HttpRequest
from django.shortcuts import render

def index(request:HttpRequest):
    context = {'loop':list(range(1,10))}
    return render(request, 'matrix.html', context)
```

用户功能设计与实现

提供用户注册处理
提供用户登录处理
提供路由配置

用户注册接口设计

接收用户通过Post方法提交的注册信息，提交的数据是JSON格式数据
检查email是否已存在与数据库表中，如果存在返回错误状态码，例如4xx，如果不存在，将用户提交的数据存入表中
整个过程都采用AJAX异步过程，用户提交JSON数据，服务端获取数据后处理，返回JSON。

URL：/user/reg
METHOD：POST

路由配置

为了避免项目中的urls.py条目过多，也为了让应用自己管理路由，采用多级路由

```
# blog/urls.py中
urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url(r'^$', index),
    url(r'^index/$', index),
    url(r'^user/', include('user.urls'))
]
```

include函数参数写`应用.路由模块`，该函数就会动态导入指定的包的模块，从模块里面读取urlpatterns，返回三元组。
url函数第二参数如果不是可调用对象，如果是元组或列表，则会从路径中除去已匹配的部分，将剩余部分与应用中的路由模块的urlpatterns进行匹配。

```
# 新建user/urls.py
from django.conf.urls import url

# 临时测试用reg视图函数
from django.http import HttpRequest, HttpResponse
def reg(request:HttpRequest):
    return HttpResponse(b'user.reg')

urlpatterns = [
    url(r'^reg$', reg)
]
```

浏览器中输入`http://127.0.0.1:8000/user/reg`测试一下，可以看到响应的数据。下面开始完善视图函数。

视图函数

在user/views.py中编写视图函数reg，路由做响应的调整。

测试JSON数据

使用POST方法，提交的数据类型为application/json，json字符串要使用双引号
这个数据是登录和注册用的，由客户端提交

```
{
  "password":"abc",
  "name":"wayne",
  "email":"wayne@magedu.com"
}
```

JSON数据处理

simplejson 比标准库方便好用，功能强大。

`$ pip install simplejson`

浏览器端提交的数据放在了请求对象的body中，需要使用simplejson解析，解析的方式同json模块，但是simplejson更方便。

错误处理

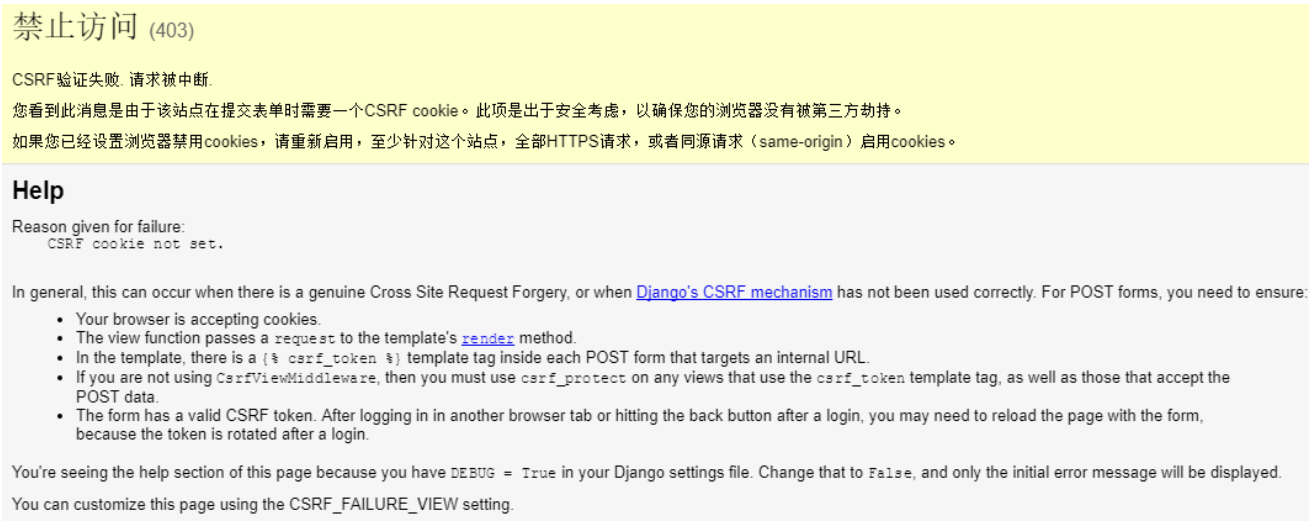
Django中有很多异常类，定义在django.http下，这些类都继承自HttpResponse。

```
# user/views.py中
from django.http import HttpRequest, HttpResponse, HttpResponseBadRequest, JsonResponse
import simplejson

def reg(request:HttpRequest):
    print(request.POST)
    print(request.body)
    payload = simplejson.loads(request.body)
    try:
        email = payload['email']

        name = payload['name']
        password = payload['password']
        print(email, name, password)
        return JsonResponse({}) # 如果正常，返回json数据
    except Exception as e: # 有任何异常，都返回
        return HttpResponseBadRequest() # 这里返回实例，这不是异常类
```

将上面代码增加邮箱检查、用户信息保存功能，就要用到Django的模型操作。

CSRF处理**

在Post数据的时候，发现出现了下面的提示



原因是，默认Django会对所有POST信息做CSRF校验。

CSRF（Cross-site request forgery）跨站请求伪造，通常缩写为CSRF或者XSRF，是一种对网站的恶意利用。
CSRF则通过伪装来自受信任用户的请求来利用受信任的网站。
CSRF攻击往往难以防范，具有非常大的危险性。

> Django 提供的 CSRF 机制
> Django 第一次响应来自某个客户端的请求时，会在服务器端随机生成一个 token，把这个 token 放在 cookie 里。然后浏览器端每次 POST 请求带上这个 token，Django的中间件验证，这样就能避免被 CSRF 攻击

解决办法

1. 关闭CSRF中间件（不推荐）

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    #'django.middleware.csrf.CsrfViewMiddleware',# 注释掉
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
```

2. 在POST提交时，需要发给服务器一个csrf_token （在Postman 中增加 X-CSRFTOKEN 字段之后在通过post 提交)
    - 模板中的表单Form中增加{% csrf_token %}，它返回到了浏览器端就会为cookie增加 csrftoken 字段，还会在表单中增加一个名为csrfmiddlewaretoken隐藏控件`<input type='hidden' name='csrfmiddlewaretoken' value='jZTxU0v5mPoLvugcfLbS1B6vT8COYrKuxMzodWv8oNAr3a4ouWlb5AaYG2tQi3dD' />`
3. 如果使用AJAX进行POST，需要在请求Header中增加X-CSRFTOKEN，其值来自cookie中获取的csrftoken值

为了测试方便，可以选择第一种方法先禁用中间件，测试完成后开启。

注册代码 V1

```python
from django.http import HttpRequest, HttpResponse, HttpResponseBadRequest, JsonResponse
import simplejson
from .models import User

# 注册函数
def reg(request:HttpRequest):
    print(request.POST)
    print(request.body)
    payload = simplejson.loads(request.body)
    try:
        # 有任何异常，都返回400，如果保存数据出错，则向外抛出异常
        email = payload['email']
        query = User.objects.filter(email=email)
        print(query)
        print(type(query),query.query) # 查看SQL语句
        if query.first():
            return HttpResponseBadRequest()  # 这里返回实例，这不是异常类

        name = payload['name']
        password = payload['password']
        print(email, name, password)

        user = User()
        user.email = email
        user.name = name
        user.password = password

        try:
            user.save()
            return JsonResponse({'user':user.id})  # 如果正常，返回json数据
        except:
            raise
    except Exception as e: # 有任何异常，都返回
        print(e)
        return HttpResponseBadRequest() # 这里返回实例，这不是异常类
```

邮箱检查

邮箱检查需要查user表，需要使用User类的filter方法。
email=email，前面是字段名email，后面是email变量。查询后返回结果，如果查询有结果，则说明该email已经存在，邮箱已经注册，返回400到前端。

用户信息存储

创建User类实例，属性存储数据，最后调用save方法。Django默认是在save()、delete()的时候事务**自动提交**。如果提交抛出任何错误，则捕获此异常做相应处理。

异常处理

- 出现获取输入框提交信息异常，就返回异常
- 查询邮箱存在，返回异常
- save()方法保存数据，有异常，则向外抛出，捕获返回异常
- 注意一点，Django的异常类继承自HttpResponse类，所以不能raise，只能return

- 前端通过状态码判断是否成功

下面我们说说模型类的操作。

Django日志

Django的日志配置在settings.py中。
必须DEBUG=True，否则logger的级别够也不打印日志。

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'handlers': {
        'console': {
            'class': 'logging.StreamHandler',
        },
    },
    'loggers': {
        'django.db.backends': {
            'handlers': ['console'],
            'level': 'DEBUG',
        },
    },
}
```

配置后，就可以在控制台看到执行的SQL语句。

模型操作

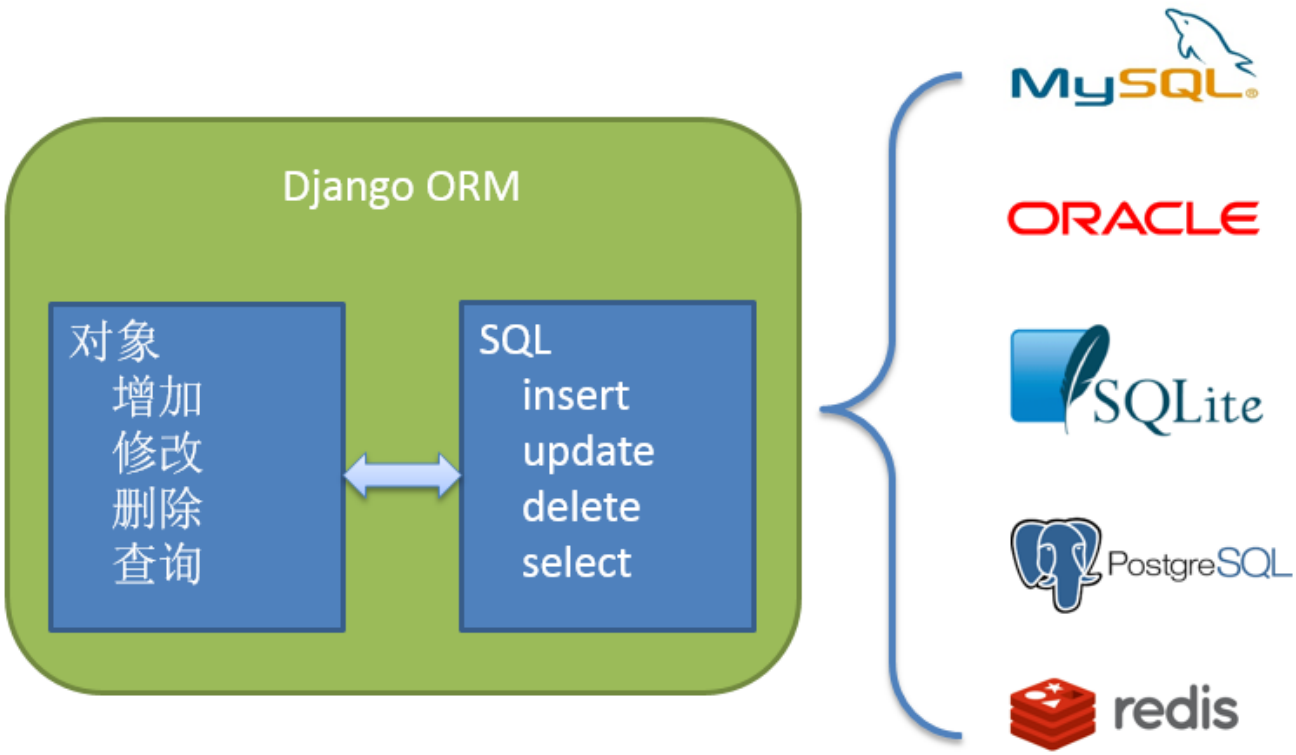
管理器对象

Django会为模型类提供一个**objects对象**，它是django.db.models.manager.Manager类型，用于与数据库交互。
当定义模型类的时候没有指定管理器，则Django会为模型类提供一个objects的管理器。
如果在模型类中手动指定管理器后，Django不再提供默认的objects的管理器了。

管理器是Django的模型进行数据库**查询**操作的接口，Django应用的每个模型都至少拥有一个管理器。

Django ORM

数据的校验validation是在对象的Save、update方法上



对模型对象的CRUD，被Django ORM转换成相应的SQL语句操作不同的数据源。

查询***

查询集

查询会返回结果的集，它是django.db.models.query.QuerySet类型。
它是惰性求值，和sqlalchemy一样。结果就是查询的集。
它是可迭代对象。

1、惰性求值：
创建查询集不会带来任何数据库的访问，直到调用方法使用数据时，才会访问数据库。在迭代、序列化、if语句中都会立即求值。

2、缓存：
每一个查询集都包含一个缓存，来最小化对数据库的访问。
新建查询集，缓存为空。首次对查询集求值时，会发生数据库查询，Django会把查询的结果存在这个缓存中，并返回请求的结果，接下来对查询集求值将使用缓存的结果。

观察下面的2个例子是要看真正生成的语句了
1) 没有使用缓存，每次都要去查库，查了2次库

```
[user.name for user in User.objects.all()]
[user.name for user in User.objects.all()]
```

2) 下面的语句使用缓存，因为使用同一个结果集

```
qs = User.objects.all()
[user.name for user in qs]
[user.name for user in qs]
```

限制查询集（切片）

查询集对象可以直接使用索引下标的方式（不支持负索引），相当于SQL语句中的limit和offset子句。
注意使用索引返回的新的结果集，依然是惰性求值，不会立即查询。

```
qs = User.objects.all()[20:40]
# LIMIT 20 OFFSET 20
qs = User.objects.all()[20:30]
# LIMIT 10 OFFSET 20
```

过滤器

返回**查询集**的方法，称为过滤器，如下：

名称	说明
all()	
filter()	过滤，返回满足条件的数据
exclude()	排除，排除满足条件的数据
order_by()	
values()	返回一个对象字典的列表，列表的元素是字典，字典内是字段和值的键值对

filter(k1=v1).filter(k2=v2) 等价于 filter(k1=v1, k2=v2)

filter(pk=10) 这里pk指的就是主键， 不用关心主键字段名，当然也可以使用使用主键名filter(emp_no=10)

返回**单个值**的方法

名称	说明
get()	仅返回单个满足条件的对象 如果未能返回对象则抛出DoesNotExist异常；如果能返回多条，抛出MultipleObjectsReturned异常
count()	返回当前查询的总条数
first()	返回第一个对象
last()	返回最后一个对象
exist()	判断查询集中是否有数据，如果有则返回True

```
user = User.objects.filter(email=email).get() # 期待查询集只有一行，否则抛出异常
user = User.objects.get(email=email) # 返回不是查询集，而是一个User实例，否则抛出异常
user = User.objects.get(id=1) # 更多的查询使用主键，也可以使用pk=1

user = User.objects.first() # 使用limit 1查询，查到返回一个实例，查不到返回None
user = User.objects.filter(pk=3, email=email).first() # and条件
```

字段查询（Field Lookup）表达式

字段查询表达式可以作为filter()、exclude()、get()的参数，实现where子句。

语法：`属性（字段）名称__比较运算符=值`

注意：属性名和运算符之间使用双下划线

比较运算符如下

名称	举例	说明
exact	filter(isdeleted=False) filter(isdeleted__exact=False)	严格等于，可省略不写
contains	exclude(title__contains='天')	是否包含，大小写敏感，等价于 `like '%天%'`
statswith endswith	filter(title__startswith='天')	以什么开头或结尾，大小写敏感
isnull isnotnull	filter(title__isnull=False)	是否为null
iexact icontains istartswith iendswith		i的意思是忽略大小写
in	filter(pk__in=[1,2,3,100])	是否在指定范围数据中

gt、gte
lt、lte
	`filter(id__gt=3)`
	`filter(pk__lte=6)`
	`filter(pub_date__gt=date(2000,1,1))`
		大于、小于等

year、month、day week_day hour、minute、 second	filter(pub_date__year=2000)	对日期类型属性处理

Q对象

虽然Django提供传入条件的方式，但是不方便，它还提供了Q对象来解决。

Q对象是django.db.models.Q，可以使用&（and）、|（or）操作符来组成逻辑表达式。`~`表示not。

```
from django.db.models import Q
User.objects.filter(Q(pk__lt=6)) # 不如直接写User.objects.filter(pk<6)

User.objects.filter(pk__gt=6).filter(pk_lt=10) # 与
User.objects.filter(Q(pk_gt=6) & Q(pk_lt=10)) # 与
User.objects.filter(Q(pk=6) | Q(pk=10)) # 或
User.objects.filter(~Q(pk__lt=6)) # 非
```

可使用&|和Q对象来构造复杂的逻辑表达式
过滤器函数可以使用一个或多个Q对象
如果混用关键字参数和Q对象，那么Q对象必须位于关键字参数的前面。所有参数都将and在一起

注册接口设计完善

认证

HTTP协议是无状态协议，为了解决它产生了cookie和session技术。

传统的session-cookie机制
浏览器发起第一次请求到服务器，服务器发现浏览器没有提供session id，就认为这是第一次请求，会返回一个新的session id给浏览器端。浏览器只要不关闭，这个session id就会随着每一次请求重新发给服务器端，服务器端查找这个session id，如果查到，就认为是同一个会话。如果没有查到，就认为是新的请求。
session是会话级的，可以在这个会话session中创建很多数据，连接断开session清除，包括session id。
这个session id还得有过期的机制，一段时间如果没有发起请求，认为用户已经断开，就清除session。浏览器端也会清除响应的cookie信息。
服务器端保存着大量session信息，很消耗服务器内存，而且如果多服务器部署，还要考虑session共享的问题，比如redis、memcached等方案。

无session方案
既然服务端就是需要一个ID来表示身份，那么不使用session也可以创建一个ID返回给客户端。但是，要保证客户端不可篡改。
服务端生成一个标识，并使用某种算法对标识签名。
服务端收到客户端发来的标识，需要检查签名。
这种方案的缺点是，加密、解密需要消耗CPU计算资源，无法让浏览器自己主动检查过期的数据以清除。
这种技术称作JWT（Json WEB Token）。

JWT

JWT（Json WEB Token）是一种采用Json方式安装传输信息的方式。
这次使用PyJWT，它是Python对JWT的实现。

包 https://pypi.python.org/pypi/PyJWT/1.5.3
文档 https://pyjwt.readthedocs.io/en/latest/

安装
`$ pip install pyjwt`

jwt原理

```
```

```python
import jwt


key = 'secret'
token = jwt.encode({'payload':'abc123'}, key, 'HS256')
print(token)
print(jwt.decode(token, key, algorithms=['HS256']))
#
b'eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJkYXRhIjoiYWJjMTIzIn0.recDeRSRirvucWKGtGPGDWkAfY4XQRK7w
pw8bJd6gB8'
# token分为3部分，用.断开

header, payload, signature = token.split(b'.')
print(header)
print(payload)
print(signature)


import base64
def addeq(b:bytes):
    '''为base64编码补齐等号'''
    rest = 4 - len(b) % 4
    return b + b'=' * rest

print('header=', base64.urlsafe_b64decode(addeq(header)))
print('payload=', base64.urlsafe_b64decode(addeq(payload)))
print('signature=', base64.urlsafe_b64decode(addeq(signature)))

# 根据jwt算法，重新生成签名
# 1 获取算法对象
from jwt import algorithms
alg = algorithms.get_default_algorithms()['HS256']
newkey = alg.prepare_key(key) # key 为 secret

# 2 获取前两部分 header.payload
signing_input,_,_ = token.rpartition(b'.')
print(signing_input)

# 3 使用key 签名
signature = alg.sign(signing_input, newkey)
print('-------------------------')
print(signature)
print(base64.urlsafe_b64encode(signature))

import json
print(base64.urlsafe_b64encode(json.dumps({'payload':'abc123'}).encode()))
```

由此，可知jwt生成的token分为三部分
1、header，由数据类型、加密算法构成
2、payload，负载就是要传输的数据，一般来说放入python对象即可，会被json序列化的
3、signature，签名部分。是前面2部分数据分别base64编码后使用点号连接后，加密算法使用key计算好一个结果，再被base64编码，得到签名

> 所有数据都是明文传输的，只是做了base64，如果是敏感信息，请不要使用jwt。
> 数据签名的目的不是为了隐藏数据，而是保证数据不被篡改。如果数据篡改了，发回到服务器端，服务器使用自己的key再计算一遍，然后进行签名比对，一定对不上签名。

Jwt使用场景

认证：这是Jwt最常用的场景，一旦用户登录成功，就会得到Jwt，然后请求中就可以带上这个Jwt。服务器中Jwt验证通过，就可以被允许访问资源。甚至可以在不同域名中传递，在单点登录（Single Sign On）中应用广泛。

数据交换：Jwt可以防止数据被篡改，它还可以使用公钥、私钥，确保请求的发送者是可信的

密码

使用邮箱 + 密码方式登录。
邮箱要求唯一就行了，但是，密码如何存储？

早期，都是明文的密码存储。
后来，使用MD5存储，但是，目前也不安全，网上有很多MD5的网站，使用反查方式找到密码。

加盐，使用hash(password + salt)的结果存入数据库中，就算拿到数据库的密码反查，也没有用了。如果是固定加盐，还是容易被找到规律，或者从源码中泄露。随机加盐，每一次盐都变，就增加了破解的难度。

暴力破解，什么密码都不能保证不被暴力破解，例如穷举。所以要使用慢hash算法，例如bcrypt，就会让每一次计算都很慢，都是秒级的，这样穷举的时间就会很长，为了一个密码破解的时间在当前CPU或者GPU的计算能力下可能需要几十年以上。

bcrypt

安装
`$ pip install bcrypt`

```
import bcrypt
import datetime

password = b'123456'

# 每次拿到盐都不一样
print(1, bcrypt.gensalt())
print(2, bcrypt.gensalt())

salt = bcrypt.gensalt()

# 拿到的盐相同，计算等到的密文相同
print('==========same salt ==========')
x = bcrypt.hashpw(password, salt)
print(3, x)
x = bcrypt.hashpw(password, salt)
print(4, x)

# 每次拿到的盐不同，计算生成的密文也不一样
print('==========different salt ==========')
x = bcrypt.hashpw(password, bcrypt.gensalt())
print(5, x)
x = bcrypt.hashpw(password, bcrypt.gensalt())
```

```
print(6, x)

# 校验
print(bcrypt.checkpw(password, x), len(x))
print(bcrypt.checkpw(password + b' ', x), len(x))

# 计算时长
start = datetime.datetime.now()
y = bcrypt.hashpw(password, bcrypt.gensalt())
delta = (datetime.datetime.now() - start).total_seconds()
print(10, 'duration={}'.format(delta))


# 检验时长
start = datetime.datetime.now()
y = bcrypt.checkpw(password, x)
delta = (datetime.datetime.now() - start).total_seconds()
print(y)
print(11, 'duration={}'.format(delta))

start = datetime.datetime.now()
y = bcrypt.checkpw(b'1', x)
delta = (datetime.datetime.now() - start).total_seconds()
print(y)
print(12, 'duration={}'.format(delta))
```

从耗时看出，bcrypt加密、验证非常耗时，所以如果穷举，非常耗时。而且碰巧攻破一个密码，由于盐不一样，还得穷举另一个。

```
salt=b'$2b$12$jwBD7mg9stvIPydF2bqoPO'
b'$2b$12$jwBD7mg9stvIPydF2bqoPOodPwWYVvdmZb5uWWuWvlf9iHqNlKSQO'

$是分隔符
$2b$，加密算法
12，表示2^12 key expansion rounds
这是盐b'jwBD7mg9stvIPydF2bqoPO'，22个字符，Base64
这是密文b'odPwWYVvdmZb5uWWuWvlf9iHqNlKSQO'，31个字符，Base64
```

注册代码 V2

全局变量

项目的settings.py文件实际上就是全局变量的配置文件。
SECRET_KEY 一个强密码

```
from django.conf import settings
print(settings.SECRET_KEY)
```

使用jwt和bcrypt，修改注册代码

```python
from django.http import HttpRequest, HttpResponse, HttpResponseBadRequest, JsonResponse
import simplejson
from .models import User
from django.conf import settings
import bcrypt
import jwt
import datetime

def gen_token(user_id):
    '''生成token'''
    return jwt.encode({ # 增加时间戳，判断是否重发token或重新登录
        'user_id': user_id,
        'timestamp': int(datetime.datetime.now().timestamp()) # 要取整
    } , settings.SECRET_KEY, 'HS256').decode() # 字符串

def reg(request:HttpRequest):
    print(request.POST)
    print(request.body)
    payload = simplejson.loads(request.body)
    try:
        # 有任何异常，都返回400，如果保存数据出错，则向外抛出异常
        email = payload['email']
        query = User.objects.filter(email=email)
        print(query)
        print(type(query),query.query) # 查看SQL语句
        if query.first():
            return HttpResponseBadRequest()  # 这里返回实例，这不是异常类

        name = payload['name']
        password = bcrypt.hashpw(payload['password'].encode(), bcrypt.gensalt())
        print(email, name, password)

        user = User()
        user.email = email
        user.name = name
        user.password = password

        try:
            user.save()
            return JsonResponse({'token':gen_token(user.id)}) # 如果正常，返回json数据
        except:
            raise
    except Exception as e: # 有任何异常，都返回
        print(e)
        return HttpResponseBadRequest() # 这里返回实例，这不是异常类
```

用户功能设计与实现

提供用户注册处理
提供用户登录处理
提供路由配置

用户登录接口设计

接收用户通过POST方法提交的登录信息，提交的数据是JSON格式数据

```
{
  "password":"abc",
  "email":"wayne@magedu.com"
}
```

从user表中email找出匹配的一条记录，验证密码是否正确。
验证通过说明是合法用户登录，显示欢迎页面。
验证失败返回错误状态码，例如4xx
整个过程都采用AJAX异步过程，用户提交JSON数据，服务端获取数据后处理，返回JSON。

URL：/user/login
METHOD：POST

路由配置

```
from django.conf.urls import url
from .views import reg, login

urlpatterns = [
    url(r'^reg$', reg),
    url(r'^login$', login),
]
```

登录代码

```
    try:
        payload = simplejson.loads(request.body) # 获取登录信息数据
        print(payload)
        email = payload['email']

        user = User.objects.get(email=email) # only one
        if bcrypt.checkpw(payload['password'].encode(), user.password.encode()):
            # 验证成功
            token = gen_token(user.id)
            res =  JsonResponse({
                'user':{
                    'user_id':user.id,
```

```
                    'name':user.name,
                    'email':user.email
                }, 'token':token
            })
            res.set_cookie('jwt', token) # 成功登陆，response报文设置set-cookie
            return res
        else:
            return HttpResponseBadRequest()
    except Exception as e:
        print(e)
        return HttpResponseBadRequest()
```

认证接口

如何获取浏览器提交的token信息?

1、使用Header中的Authorization
通过这个header增加token信息。
通过header发送数据，方法可以是Post、Get

2、自定义header
在Http请求头中使用JWT字段来发送token
我们选择第二种方式

认证
基本上所有的业务都需要认证用户的信息。
在这里比较时间戳，如果过期，就直接抛未认证401，客户端收到后就该直接跳转到登录页。
如果没有提交user id，就直接重新登录。如果用户查到了，填充user对象。

request -> 时间戳比较 -> user id 比较 -> 向后执行

Django的认证

django.contrib.auth中提供了许多方法，这里主要介绍其中的三个:

1、authenticate(**credentials)
提供了用户认证，即验证用户名以及密码是否正确
user = authentica(username='someone',password='somepassword')

2、login(HttpRequest, user, backend=None)
该函数接受一个HttpRequest对象，以及一个认证了的User对象
此函数使用django的session框架给某个已认证的用户附加上session id等信息。

3、logout(request)
注销用户
该函数接受一个HttpRequest对象，无返回值。
当调用该函数时，当前请求的session信息会全部清除
该用户即使没有登录，使用该函数也不会报错

还提供了一个装饰器来判断是否登录django.contrib.auth.decorators.login_required

本项目使用了无session机制，且用户信息自己建表管理，所以，认证需要自己实现。

中间件技术Middleware

官方定义，在Django的request和response处理过程中，由框架提供的hook钩子

中间件技术在1.10后发生了改变，我们当前使用1.11版本，可以使用新的方式定义。
参看 https://docs.djangoproject.com/en/1.11/topics/http/middleware/#writing-your-own-middleware

```
#
class BlogAuthMiddleware(object):
    '''自定义认证中间件'''
    def __init__(self, get_response):
        self.get_response = get_response
        # 初始化执行一次

    def __call__(self, request):
        # 视图函数之前执行
        # 认证
        print(type(request), '++++++++++++++++++')
        print(request.GET)
        print(request.POST)
        print(request.body) # json数据
        print('-'*30)

        response = self.get_response(request)

        # 视图函数之后执行

        return response
# 要在setting的MIDDLEWARE中注册
```

但是，这样所有的请求和响应都拦截，我们还得判断是不是访问的想要拦截的view函数，所以，考虑其他方法。

中间件有很多用途，适合拦截所有请求和响应。例如浏览器端的IP是否禁用、UserAgent分析、异常响应的统一处理。

思考：使用中间件实现访问IP统计

装饰器*

在需要认证的view函数上增强认证功能，写一个装饰器函数。谁需要认证，就在这个view函数上应用这个装饰器。

```
AUTH_EXPIRE = 8 * 60 * 60 # 8小时过期

def authenticate(view):
    def wrapper(request:HttpRequest):
        # 自定义header jwt
        payload = request.META.get('HTTP_JWT') # 会被加前缀HTTP_且全大写
```

```
        if not payload: # None没有拿到，认证失败
            return HttpResponse(status=401)
        try: # 解码
            payload = jwt.decode(payload, settings.SECRET_KEY, algorithms=['HS256'])
            print(payload)
        except:
            return HttpResponse(status=401)
        # 验证过期时间
        current = datetime.datetime.now().timestamp()
        if (current - payload.get('timestamp', 0)) > AUTH_EXPIRE:
            return HttpResponse(status=401)
        print('-'*30)
        try:
            user_id = payload.get('user_id', -1)
            user = User.objects.get(pk=user_id)
            request.user = user # 如果正确，则注入user
            print('-'*30)
        except Exception as e:
            print(e)
            return HttpResponse(status=401)

        ret = view(request) # 调用视图函数
        # 特别注意view调用的时候，里面也有返回异常
        return ret
    return wrapper


@authenticate # 很自由的应用在需要认证的view函数上
def test(request:HttpRequest):
    return HttpResponse('test')
```

Jwt过期问题

pyjwt支持过期设定，在decode的时候，如果过期，则抛出异常。
需要在payload中增加claim exp。
exp要求是一个整数int的时间戳。

```
import jwt
import datetime
import threading

event = threading.Event()

key = 'magedu'

# 在jwt的payload中增加exp claim
data = jwt.encode({'name':'tom', 'age':20, 'exp': int(datetime.datetime.now().timestamp())+10},
key)
print(jwt.get_unverified_header(data)) # 不校验签名提取header
try:
    while not event.wait(1):
        print(jwt.decode(data, key)) # 过期，校验就会抛出异常
```

```
        print(datetime.datetime.now().timestamp())
except jwt.ExpiredSignatureError as e:
    print(e)
```

重写Jwt过期

```
AUTH_EXPIRE = 8 * 60 * 60 # 8小时过期

def gen_token(user_id):
    '''生成token'''
    return jwt.encode({ # 增加时间戳，判断是否重发token或重新登录
        'user_id': user_id,
        'exp': int(datetime.datetime.now().timestamp()) + AUTH_EXPIRE # 要取整
    } , settings.SECRET_KEY, 'HS256').decode() # 字符串


def authenticate(view):
    def wrapper(request:HttpRequest):
        # 自定义header jwt
        payload = request.META.get('HTTP_JWT') # 会被加前缀HTTP_且全大写
        if not payload: # None没有拿到，认证失败
            return HttpResponse(status=401)
        try: # 解码，同时验证过期时间
            payload = jwt.decode(payload, settings.SECRET_KEY, algorithms=['HS256'])
            print(payload)
        except:
            return HttpResponse(status=401)

        try:
            user_id = payload.get('user_id', -1)
            user = User.objects.get(pk=user_id)
            request.user = user # 如果正确，则注入user
            print('-'*30)
        except Exception as e:
            print(e)
            return HttpResponse(status=401)

        ret = view(request) # 调用视图函数
        return ret
    return wrapper
```

博文相关接口

功能分析

功能	函数名	Request方法	路径
发布（增）	pub	POST	/pub
看文章（查）	get	GET	/(\d+)
列表(分页)	getall	GET	/

创建博文应用

```
$ python manage.py startapp post
```

注意：**一定要把应用post加入到settings.py中**，否则不能迁移

模型

```
from django.db import models
from user.models import User


class Post(models.Model):
    class Meta:
        db_table = 'post'
    id = models.AutoField(primary_key=True)
    title = models.CharField(max_length=256, null=False)
    postdate = models.DateTimeField(null=False)
    # 从post查作者，从post查内容
    author = models.ForeignKey(User) # 指定外键，migrate会生成author_id字段
    # self.content可以访问Content实例，其内容是self.content.content

    def __repr__(self):
        return '<Post {} {} {} {} >'.format(
            self.id, self.title, self.author_id, self.content)

    __str__ = __repr__


class Content(models.Model):
    class Meta:
        db_table = 'content'
    # 没有主键，会自动创建一个自增主键
    post = models.OneToOneField(Post) # 一对一，这边会有一个外键post_id引用post.id
    content = models.TextField(null=False)

    def __repr__(self):
```

```
        return '<Content {} {}>'.format(self.post.id, self.content[:20])

    __str__ = __repr__
```

路由

全局settings.py

```
urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url(r'^$', index),
    url(r'^index/$', index),
    url(r'^user/', include('user.urls')),
    url(r'^post/', include('post.urls')),
]
```

post应用urls.py

```
from django.conf.urls import url
from .views import pub, get, getall

urlpatterns = [
    url(r'^pub$', pub),
    url(r'^(\d+)$', get), # 给get传入一个参数str类型
    url(r'^$', getall),
]
```

pub接口实现

用户从浏览器端提交Json数据，包含title，content。
提交博文需要认证用户，从请求的header中验证jwt。

`request: POST-> @authenticate pub -> return post_id`

```
from django.http import HttpResponse, HttpRequest, JsonResponse, HttpResponseBadRequest,
HttpResponseNotFound
from user.views import authenticate
from user.models import User
import simplejson
import datetime
from .models import Post, Content


@authenticate
def pub(request:HttpRequest):
    post = Post() # 新增
    content = Content() # 新增
    try:
        payload = simplejson.loads(request.body)
        post.title = payload['title']
```

```
        post.author = User(id=request.user.id) # 注入的
        #post.author = request.user
        post.postdate = datetime.datetime.now(
            datetime.timezone(datetime.timedelta(hours=8)))

        post.save()

        content.content = payload['content']
        content.post = post
        content.save()

        return JsonResponse({'post_id':post.id})
    except Exception as e:
        print(e)
        return HttpResponseBadRequest()
```

显式事务处理

Django中每一次save()调用就会自动提交，那么上例中第一次事务提交后如果第二次提交前出现异常，则post.save()不会回滚。

解决办法可以使用事务的原子方法，参考https://docs.djangoproject.com/zh-hans/2.1/topics/db/transactions/#controlling-transactions-explicitly

```
from django.db import transaction

@transaction.atomic # 装饰器用法
def viewfunc(request):
    # This code executes inside a transaction.
    do_stuff()
```

```
from django.db import transaction

def viewfunc(request):
    # This code executes in autocommit mode (Django's default).
    do_stuff()

    with transaction.atomic(): # 上下文用法
        # This code executes inside a transaction.
        do_more_stuff()
```

上面2种用法都可以，我们这一次采用第二种方法。

```
@authenticate
def pub(request:HttpRequest):
    post = Post() # 新增
    content = Content() # 新增
    try:
        payload = simplejson.loads(request.body)
```

```
        post.title = payload['title']
        post.author = User(id=request.user.id) # 注入的
        #post.author = request.user
        post.postdate = datetime.datetime.now(
            datetime.timezone(datetime.timedelta(hours=8)))

        with transaction.atomic(): # 原子操作
            post.save() # save方法必须在前，save后才有post.id

            content.content = payload['content']
            content.post = post
            content.save()


        return JsonResponse({'post_id':post.id})
    except Exception as e:
        print(e)
        return HttpResponseBadRequest()
```

get接口实现

根据post_id查询博文并返回。

这里需要认证吗?
如果博文只能作者看，就需要认证。我们这里的公开给所有人看，所以不需要认证，同样，下面的list接口也是不需要认证的。

`request: GET-> get Post by id -> return post+content`

```
def get(request:HttpRequest, id): # 分组捕获传入
    try:
        id = int(id)
        post = Post.objects.get(pk=id)
        print(post, '~~~~~~~~')
        if post:
            return JsonResponse({
                'post':{
                    'post_id':post.id,
                    'title':post.title,
                    'author':post.author.name,
                    'author_id':post.author_id, # post.author.id
                    'postdate':post.postdate.timestamp(),
                    'content':post.content.content
                }
            })
        # get方法保证必须只有一条记录，否则抛异常
    except Exception as e:
        print(e)
        return HttpResponseNotFound()
```

getall接口实现

发起get请求，通过查询字符串`http://url/post/?page=2`查询第二页数据

`request: GET-> get all (page=1) -> return post list`

```
def getall(request:HttpRequest):
    try: # 页码
        page = int(request.GET.get('page', 1))
        page = page if page > 0 else 1
    except:
        page = 1

    try: # 页码行数
        # 注意，这个数据不要轻易让浏览器端改变，如果允许改变，一定要控制范围
        size = int(request.GET.get('size', 20))
        size = size if size > 0 and size < 101 else 20
    except:
        size = 20

    try:
        # 按照id倒排
        start = (page - 1) * size
        posts = Post.objects.order_by('-id')[start:start+size]
        #posts = Post.objects.order_by('-pk')[start:start+size]
        print(posts.query)
        return JsonResponse({
            'posts':[
                {
                    'post_id': post.id,
                    'title': post.title
                } for post in posts
            ]
        })

    except Exception as e:
        print(e)
        return HttpResponseBadRequest()
```

完善分页

分页信息，一般有：当前页/总页数、行限制数、记录总数。
当前页：page
行限制数：size，每页最多多少行
总页数：pages = math.ceil(count/size)
记录总数：count，从select * from table来

```
def getall(request: HttpRequest):
    try:  # 页码
        page = int(request.GET.get('page', 1))
        page = page if page > 0 else 1
```

```
    except:
        page = 1

    try:  # 页码行数
        # 注意，这个数据不要轻易让浏览器端改变，如果允许改变，一定要控制范围
        size = int(request.GET.get('size', 20))
        size = size if size > 0 and size < 101 else 20
    except:
        size = 20

    try:
        # 按照id倒排
        start = (page - 1) * size
        posts = Post.objects.order_by('-id')
        print(posts.query)
        count = posts.count()

        posts = posts[start:start + size]
        print(posts.query)

        return JsonResponse({
            'posts': [
                {
                    'post_id': post.id,
                    'title': post.title
                } for post in posts
            ], 'pagination': {
                'page': page,
                'size': size,
                'count': count,
                'pages': math.ceil(count / size)
            }
        })

    except Exception as e:
        print(e)
        return HttpResponseBadRequest()
```

也可以使用Django提供的Paginator类来完成。
Paginator文档 https://docs.djangoproject.com/en/1.11/topics/pagination/。
但是，还是自己处理更加简单明了些。

改写校验函数

```
def validate(d:dict, name:str, type_func, default, validate_func):
    try:  # 页码
        result = type_func(d.get(name, default))
        result = validate_func(result, default)
    except:
        result = default
    return result
```

```python
def getall(request: HttpRequest):
    # 页码
    page = validate(request.GET, 'page', int, 1, lambda x,y: x if x>0 else y)
    # 注意，这个数据不要轻易让浏览器端改变，如果允许改变，一定要控制范围
    size = validate(request.GET, 'size', int, 20, lambda x,y: x if x>0 and x<101 else y)

    try:
        # 按照id倒排
        start = (page - 1) * size
        posts = Post.objects.order_by('-id')
        print(posts.query)
        count = posts.count()

        posts = posts[start:start + size]
        print(posts.query)

        return JsonResponse({
            'posts': [
                {
                    'post_id': post.id,
                    'title': post.title
                } for post in posts
            ], 'pagination': {
                'page': page,
                'size': size,
                'count': count,
                'pages': math.ceil(count / size)
            }
        })

    except Exception as e:
        print(e)
        return HttpResponseBadRequest()
```

前端开发

开发环境设置

使用react-mobx-starter-master脚手架，解压更名为frontend。
在src中新增component、service、css目录。

注：没有特别说明，js开发都在src目录下

```
目录结构
frontend/
    |-.babelrc
    |-.gitignore
    |-.npmrc
    |-index.html
    |-jsconfig.json
    |-LICENSE
    |-package-lock.json
    |-package.json
    |-README.md
    |-webpack.config.dev.js
    |-webpack.config.prod.js
    └-src/
        |- componet/
        |- service/
        |- css/
        |- index.html
        |- index.js
```

修改项目信息

```
{
  "name": "blog",
  "description": "blog project",
  "author": "wayne"
}
```

webpack.config.dev.js

```
    devServer: {
        compress: true, /* gzip */
        //host: '192.168.142.1', // IP设置
        port: 3000,
        publicPath: '/assets/', /* 设置bundled files浏览器端访问地址 */
        hot: true, /* 开启HMR热模块替换 */
```

```
        inline: true, /* 控制浏览器控制台是否显示信息 */
        historyApiFallback: true,
        stats: {
            chunks: false
        },
        proxy: { // 代理
            '/api': {
                target: 'http://127.0.0.1:8000',
                changeOrigin: true
            }
        }
    }
```

安装依赖

```
$ npm install
```

npm会安装package.json中依赖的包。

也开始使用新的包管理工具yarn安装模块

```
yarn安装
$ npm install -g yarn
或者，去 https://yarn.bootcss.com/docs/install/

相当于 npm install
$ yarn

相当于npm install react-router
$ yarn add react-router
$ yarn add react-router-dom
```

开发

前端路由

前端路由使用react-router组件完成

官网文档 https://reacttraining.com/react-router/web/guides/quick-start

基本例子 https://reacttraining.com/react-router/web/example/basic

使用react-router组件，更改src/index.js

```
import React from 'react';
import ReactDom from 'react-dom';
import {Route, Link, BrowserRouter as Router} from 'react-router-dom';
```

```
const Home = () => (
  <div>
    <h2>Home</h2>
  </div>
);

const About = () => (
  <div>
    <h2>About</h2>
  </div>
);

const App = () => (
  <Router>
    <div>
      <Route exact path="/" component={Home} />
      <Route path="/about" component={About} />
    </div>
  </Router>
);

ReactDom.render(<App />, document.getElementById('root'));
```

在地址栏里面输入`http://127.0.0.1:3000/`或`http://127.0.0.1:3000/about`试试看，能够看到页面的变化

App中，使用了Router路由组件，Router是根，且它只能有一个元素，所以加了div。

Route负责静态路由
path是匹配路径，没有path总是匹配。
component是目标组件，
    exact：布尔值，true时，要求路径完全匹配。
    strict：布尔值，true时，要求严格匹配，但是url字符串可以是自己的子串。

地址变化，Router组件会匹配路径，然后使用匹配的组件渲染。

登录组件

在component目录下构建react组件。

登录页模板
https://codepen.io/colorlib/pen/rxddKy?q=login&limit=all&type=type-pens

```
<div class="login-page">
  <div class="form">
    <form class="register-form">
      <input type="text" placeholder="name"/>
      <input type="password" placeholder="password"/>
      <input type="text" placeholder="email address"/>
      <button>create</button>
      <p class="message">Already registered? <a href="#">Sign In</a></p>
    </form>
    <form class="login-form">
      <input type="text" placeholder="username"/>
```

```
      <input type="password" placeholder="password"/>
      <button>login</button>
      <p class="message">Not registered? <a href="#">Create an account</a></p>
    </form>
  </div>
</div>
```

使用这个HTML模板来构建组件。

特别注意

- 搬到React组件中的时候，要将**class属性改为className。**
- 所有标签，需要闭合。

login.js

在component目录下建立login.js的登录组件。
使用上面的模板的HTML中的登录部分，挪到render函数中。

- 修改class为className
- 将`<a>`标签替换成`<Link to="?">`组件
- 注意标签闭合问题

```
import React from 'react';
import {Link} from 'react-router-dom';

export default class Login extends React.Component {
    render() {
        return (<div className="login-page">
            <div className="form">
                <form className="login-form">
                    <input type="text" placeholder="邮箱" />
                    <input type="password" placeholder="密码" />
                    <button>登录</button>
                    <p className="message">还未注册? <Link to="/reg">请注册</Link></p>
                </form>
            </div>
        </div>);
    }
}
```

index.js
在路由中增加登录组件

```
import Login from './component/login';

const App = () => (
  <Router>
    <div>
      <Route path="/about" component={About} />
      <Route path="/login" component={Login} />
      <Route exact path="/" component={Home} />
    </div>
  </Router>
);
```

访问`http://127.0.0.1:3000/login`就可以看到登录界面了。但是没有样式。

样式表

在src/css中，创建login.css，放入一下内容，然后在login.js中导入样式

```
body {
  background: #456;
  font-family: SimSun;
  font-size: 14px;
}

.login-page {
  width: 360px;
  padding: 8% 0 0;
  margin: auto;
}
.form {
  font-family: "Microsoft YaHei", SimSun;
  position: relative;
  z-index: 1;
  background: #FFFFFF;
  max-width: 360px;
  margin: 0 auto 100px;
  padding: 45px;
  text-align: center;
  box-shadow: 0 0 20px 0 rgba(0, 0, 0, 0.2), 0 5px 5px 0 rgba(0, 0, 0, 0.24);
}
.form input {
  outline: 0;
  background: #f2f2f2;
  width: 100%;
  border: 0;
  margin: 0 0 15px;
  padding: 15px;
  box-sizing: border-box;
  font-size: 14px;
}
.form button {
  text-transform: uppercase;
```

```
  outline: 0;
  background: #4CAF50;
  width: 100%;
  border: 0;
  padding: 15px;
  color: #FFFFFF;
  font-size: 14px;
  cursor: pointer;
}
.form button:hover,.form button:active,.form button:focus {
  background: #43A047;
}
.form .message {
  margin: 15px 0 0;
  color: #b3b3b3;
  font-size: 12px;
}
.form .message a {
  color: #4CAF50;
  text-decoration: none;
}
```

在login.js导入样式表

```
import React from 'react';
import '../css/login.css';
```

如有需要，重启dev server。可以看到界面，如下(http://127.0.0.1:3000/login)

注册组件

与登录组件编写方式差不多，创建component/reg.js，使用login.css

```
import React from 'react';
import { Link } from 'react-router-dom';
import '../css/login.css'

export default class Reg extends React.Component {
    render() {
        return (
            <div className="login-page">
                <div className="form">
                    <form className="register-form">
                        <input type="text" placeholder="姓名" />
                        <input type="text" placeholder="邮箱" />
                        <input type="password" placeholder="密码" />
                        <input type="password" placeholder="确认密码" />
                        <button>注册</button>
                        <p className="message">如果已经注册 <Link to="/login">请登录</Link></p>
                    </form>
                </div>
            </div>
        );
    }
}
```

在index.js中增加一条静态路由

```
import Reg from './component/reg';

const App = () => (
  <Router>
    <div>
      <Route path="/about" component={About} />
      <Route path="/login" component={Login} />
      <Route path="/reg" component={Reg} />
      <Route exact path="/" component={Home} />
    </div>
  </Router>
);
```

可以看到注册界面，如下(http://127.0.0.1:3000/reg)

导航栏链接

在index.js中增加导航栏链接，方便页面切换

```
const App = () => (
  <Router>
    <div>
      <div>
        <ul>
          <li><Link to="/">主页</Link></li>
          <li><Link to="/login">登录</Link></li>
          <li><Link to="/reg">注册</Link></li>
          <li><Link to="/about">关于</Link></li>
        </ul>
      </div>
      <Route path="/about" component={About} />
      <Route path="/login" component={Login} />
      <Route path="/reg" component={Reg} />
      <Route exact path="/" component={Home} />
    </div>
  </Router>
);
```

分层

层次	作用	路径
视图层	负责数据呈现，负责用户交互界面	src/component/xxx.js
服务层	负责业务逻辑处理	src/service/xxx.js
Model层	数据持久化	

登录功能实现

view层，登录组件和用户交互。当button点击触发onClick，调用事件响应函数handleClick，handleClick中调用服务service层的login函数。

service层，负责业务逻辑处理。调用Model层数据操作函数

在src/service/user.js

```
export default class UserService {
    login (email, password) {
        // TODO
    }
}
```

src/component/login.js

```
import React from 'react';
import {Link} from 'react-router-dom';
import '../css/login.css'

export default class Login extends React.Component {
    handleClick(event) {
        console.log(event.target)
    }

    render() {
        return (<div className="login-page">
            <div className="form">
                <form className="login-form">
                    <input type="text" placeholder="邮箱" />
                    <input type="password" placeholder="密码" />
                    <button onClick={this.handleClick.bind(this)}>登录</button>
                    <p className="message">还未注册? <Link to="/reg">请注册</Link></p>
                </form>
            </div>
        </div>);
    }
}
```

问题:

- 页面提交
  这一次发现有一些问题，按钮点击会提交，导致页面刷新了。
  要阻止页面刷新，其实就是阻止提交。使用event.preventDefault()。
- 如何拿到邮箱和密码?
  event.target.form返回按钮所在表单，可以看做一个数组。
  fm[0].value和fm[1].value就是文本框的值。
- 如何在Login组件中使用UserService实例呢?
  使用全局变量，虽然可以，但是不好。
  可以在Login的构造器中通过属性注入。
  也可以在外部使用props注入。使用这种方式。

```
import React from 'react';
import {Link} from 'react-router-dom';
import '../css/login.css'
import UserService from '../service/user';

const userService = new UserService();

export default class Login extends React.Component {
    render () {
        return <_Login service={userService} />;
    }
}
```

```
class _Login extends React.Component {
    handleClick(event) {
        event.preventDefault();
        let fm = event.target.form;
        this.props.service.login(
            fm[0].value, fm[1].value
        );
    }

    render() {
        return (<div className="login-page">
            <div className="form">
                <form className="login-form">
                    <input type="text" placeholder="邮箱" />
                    <input type="password" placeholder="密码" />
                    <button onClick={this.handleClick.bind(this)}>登录</button>
                    <p className="message">还未注册? <Link to="/reg">请注册</Link></p>
                </form>
            </div>
        </div>);
    }
}
```

UserService的login方法实现

代理配置

修改webpack.config.dev.js文件中proxy部分，要保证proxy的target是后台服务的地址和端口，且要开启后台服务。
注意，修改这个配置，需要重启dev server

```
    devServer: {
        compress: true,
        port: 3000,
        publicPath: '/assets/',
        hot: true,
        inline: true,
        historyApiFallback: true,
        stats: {
            chunks: false
        },
        proxy: {
            '/api': {
                target: 'http://127.0.0.1:8000',
                changeOrigin: true
            }
        }
    }
```

axios异步库

axios是一个基于Promise的HTTP异步库，可以用在浏览器或nodejs中。

使用axios发起异步调用，完成POST、GET方法的数据提交。可参照官网的例子

中文说明 https://www.kancloud.cn/yunye/axios/234845

安装npm

`$ npm install axios` 或 `$ yarn add axios`

注意，如果使用yarn安装，就不要再使用npm安装包了，以免出现问题。

导入

`import axios from 'axios';`

service/user.js修改如下

```
import axios from 'axios';

export default class UserService {
    login (email, password) {
        console.log(email, password);

        axios.post('/api/user/login', {
            email:email,
            password:password
        })/* dev server会代理 */
        .then(
            function (response) {
                console.log(response);
                console.log(response.data);
                console.log(response.status);
                console.log(response.statusText);
                console.log(response.headers);
                console.log(response.config);
            }
        ).catch(
            function (error) {
                console.log(error);
            }
        )
    }
}
```

问题：

- 404

  填入邮箱、密码，点击登录按钮，返回404，查看Python服务端，访问地址是`/api/user/login`，也就是多了/api。如何解决?

  1、修改blog server的代码的路由匹配规则
  不建议这么做，影响有点大

  2、rewrite

  类似httpd、nginx等的rewrite功能。本次测试使用的是dev server，去官方看看。

https://webpack.js.org/configuration/dev-server/#devserver-proxy

可以查到pathRewrite可以完成路由重写。

```
devServer: {
    compress: true,
    port: 3000,
    publicPath: '/assets/',
    hot: true,
    inline: true,
    historyApiFallback: true,
    stats: {
        chunks: false
    },
    proxy: {
        '/api': {
            target: 'http://127.0.0.1:8000',
            pathRewrite: {"^/api" : ""},
            changeOrigin: true
        }
    }
}
```

重启dev server。

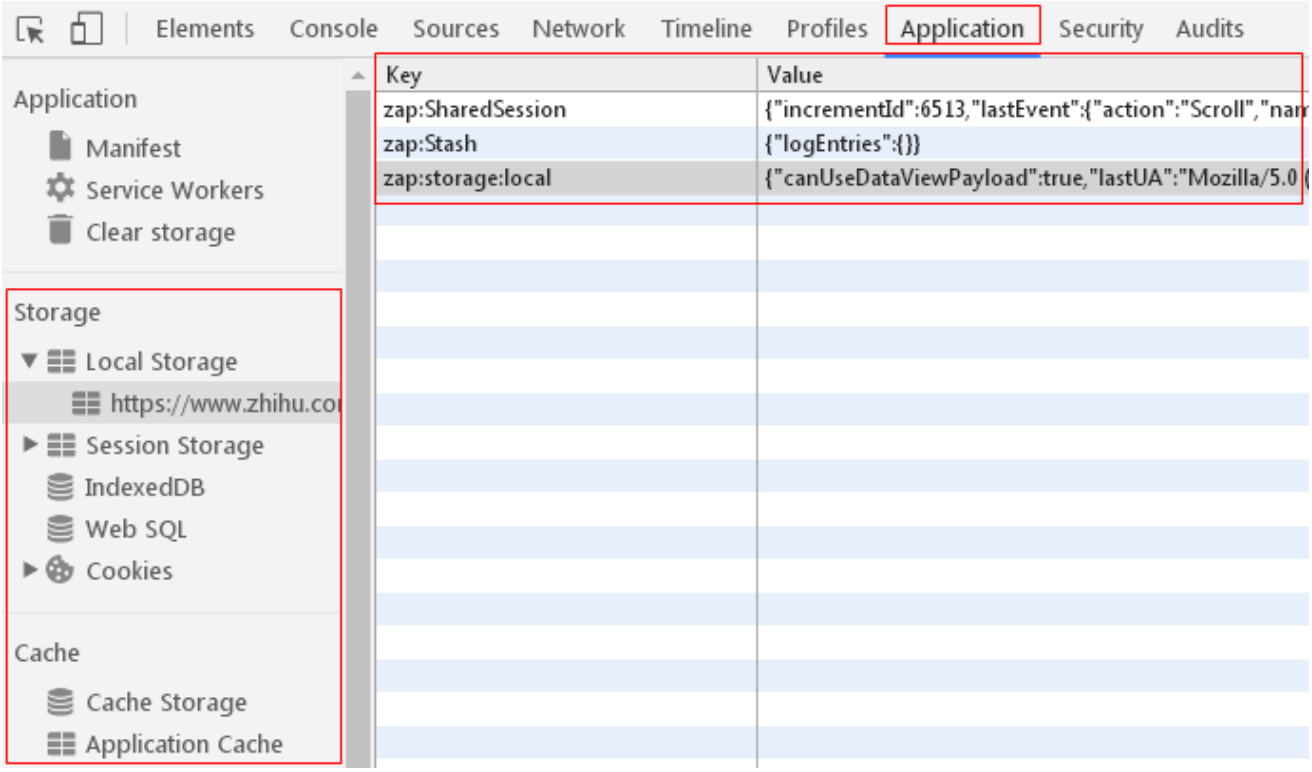
使用正确的邮箱、密码登录，返回了json数据，在response.data中可以看到token、user。

---

token持久化——LocalStorage

使用LocalStorage来存储token。
LocalStorage 是HTML5标准增加的技术，是浏览器端持久化方案之一。
LocalStorage 是为了存储浏览器得到的数据，例如Json。
数据存储是键值对。
数据会存储在不同的域名下面。
不同浏览器对单个域名下存储数据的长度支持不同，有的最多支持2MB。

Chrome浏览器中查看，如下



SessionStorage和LocalStorage功能差不多，只不过SessionStorage是会话级的，浏览器关闭，会话结束，数据清除。而LocalStorage可以持久保存。

IndexedDB

- 一个域一个datatable
- key-value检索方式
- 建立在关系型的数据模型之上，具有索引表、游标、事务等概念

store.js

store.js 是一个兼容所有浏览器的 LocalStorage 包装器，不需要借助 Cookie 或者 Flash。
store.js 会根据浏览器自动选择使用 localStorage、globalStorage 或者 userData 来实现本地存储功能。

安装

`$ npm i store` 或 `yarn add store`

测试代码

编写一个test.js，使用node exec插件按F8执行

```
let store = require('store');

store.set('user', 'wayne');
console.log(store.get('user'));

store.remove('user');
console.log(store.get('user')); // undefined
console.log(store.get('user','a')); // a
```

```
store.set('user', {name:'wayne',age:30});
console.log(store.get('user').name);

store.set('school', {name:'magedu'});

store.each(function(value, key) { // 注意这里key、value是反的
    console.log(key, '-->', value)
});

store.clearAll();

console.log(store.get('user')); // undefined
```

安装store的同时，也安装了expire过期插件，可以在把kv对存储到LS中的时候顺便加入过期时长。

```
const store = require('store');
// 一定要加载插件，否则不会key不会过期
store.addPlugin(require('store/plugins/expire'));

let d = new Date()
store.set('user', 'wayne', (new Date()).getTime() + (10 * 1000)) // 注意时间单位

setInterval(() => console.log(store.get('user', 'abc')), 1000);
```

下面是准备写在service中的代码

```
import store from 'store';
import expire from 'store/plugins/expire'; // 过期插件

// 存储token
store.set('token',res.data.token, (new Date()).getTime() + (8*3600*1000));
```

Mobx状态管理

Redux和Mobx

社区提供的状态管理库，有Redux和Mobx。

Redux代码优秀，使用严格的函数式编程思想，学习曲线陡峭，小项目使用的优势不明显。
Mobx，非常优秀稳定的库，简单方便，适合中小项目使用。使用面向对象的方式，容易学习和接受。现在在中小项目中使用也非常广泛。Mobx和React也是一对强力组合。

Mobx官网 https://mobx.js.org/
中文网 http://cn.mobx.js.org/

MobX 是由 Mendix、Coinbase、Facebook 开源，它实现了观察者模式。

观察者模式
观察者模式，也称为发布订阅模式，观察者观察某个目标，目标对象(Obserable)状态发生了变化，就会通知自己内部注册了的观察者Observer。

状态管理

需求

一个组件的onClick触发事件响应函数，此函数会调用后台服务。但是后台服务比较耗时，等处理完，需要引起组件的渲染操作。
要组件渲染，就需要改变组件的props或state。

1、同步调用

同步调用中，实际上就是等着耗时的函数返回

```
import React from 'react';
import ReactDom from 'react-dom';

class Service {
  handle(n) {
    // 同步
    console.log('pending~~~~~~~~')
    // 同步死循环
    for (let s=new Date(); new Date()-s < n*1000;);
    console.log('done');
    return Math.random();
  }
}

class Root extends React.Component {
  state = {ret:null}
  handleClick(event){
    // 同步返回值
    let ret = this.props.service.handle(4);
    this.setState({ret:ret});
  }

  render() {
    console.log('**********************');
    return (
      <div>
        <button onClick={this.handleClick.bind(this)}>触发handleClick函数</button><br />
        <span style={{color:'red'}}>{new Date().getTime()} Service的handle函数返回值是:
{this.state.ret}</span>
      </div>);
  }
}

ReactDom.render(<Root service={new Service()} />, document.getElementById('root'));
```

这里使用一个死循环来模拟同步调用，来模拟耗时的等待返回的过程。

在调用过程中，整个页面鼠标不能点击了。

2、异步调用

思路一、使用setTimeout

使用setTimeout，有2个问题。
1、无法向内部的待执行函数传入参数，比如传入Root实例。
2、延时执行的函数的返回值无法取到，所以无法通知Root

思路二、Promise异步执行

Promise异步执行，如果成功执行，将调用回调。

```
import React from 'react';
import ReactDom from 'react-dom';

class Service {
  handle(obj) {
    // Promise
    new Promise((resolve, reject) => {
      setTimeout(()=>resolve('OK'), 5000);
    }).then(
      value => { // 使用obj
        obj.setState({ret:(Math.random()*100)});
      }
    )
  }
}

class Root extends React.Component {
  state = {ret:null}
  handleClick(event){
    // 异步不能直接使用返回值
    this.props.service.handle(this);
  }

  render() {
    console.log('**********************');
    return (
      <div>
        <button onClick={this.handleClick.bind(this)}>触发handleClick函数</button><br />
        <span style={{color:'red'}}>{new Date().getTime()} Service中修改state的值是:
{this.state.ret}</span>
      </div>);
  }
}

ReactDom.render(<Root service={new Service()} />, document.getElementById('root'));
```

不管render中是否显示state的值，只要state改变，都会触发render执行。

3、Mobx实现

observable装饰器：设置被观察者
observer装饰器：设置观察者，将React组件转换为响应式组件

```
import React from 'react';
import ReactDom from 'react-dom';
import {observable} from 'mobx';
import {observer} from 'mobx-react';

class Service {
  @observable ret = -100;
  handle() {
    // Promise
    new Promise((resolve, reject) => {
      setTimeout(()=>resolve('OK'), 2000);
    }).then(
      value => {
        this.ret = (Math.random()*100);
        console.log(this.ret);
      }
    )
  }
}

@observer // 将react组件转换为响应式组件
class Root extends React.Component {
  //state = {ret:null} // 不使用state了
  handleClick(event){
    // 异步不能直接使用返回值
    this.props.service.handle(this);
  }

  render() {
    console.log('**********************');
    return (
      <div>
        <button onClick={this.handleClick.bind(this)}>触发handleClick函数</button><br />
        <span style={{color:'red'}}>{new Date().getTime()} Service中修改state的值是:
{this.props.service.ret}</span>
      </div>);
  }
}

ReactDom.render(<Root service={new Service()} />, document.getElementById('root'));
```

Service中被观察者ret变化，导致了观察者调用了render函数。

被观察者变化不引起渲染的情况：将上例中的`{this.props.service.ret}`注释`{/*this.props.service.ret*/}`。可以看到，如果在render中不使用这个被观察者，render函数就不会调用。

在观察者的render函数中，一定要使用这个被观察对象。

跳转

如果service中ret发生了变化，观察者Login就会被通知到。一般来说，就会跳转到用户界面，需要使用Redirect组件。

```
// 导入Redirect
import {Link, Redirect} from 'react-router-dom';

//render函数中return
return <Redirect to='/' />; //to表示跳转到哪里
```

---

login登录功能代码实现

```
// service/user.js
import axios from 'axios';
import store from 'store';
import expire from 'store/plugins/expire';
import {observable} from 'mobx';

// 过期插件
store.addPlugin(expire);

export default class UserService {
    @observable loggedin = false; //+ 被观察者

    login (email, password) {
        console.log(email, password);

        axios.post('/api/user/login', {
            email:email,
            password:password
        })/* dev server会代理 */
        .then(
            response => { // 此函数要注意this的问题
                console.log(response.data);
                console.log(response.status);
                //+ 存储token, 注意需要重开一次chrome的调试窗口才能看到
                store.set('token',
                response.data.token,
                (new Date()).getTime() + (8*3600*1000));
                this.loggedin = true; //+ 修改被观察者
            }
        ).catch(
            function (error) {
                console.log(error);
            }
        )
    }
}
```

```
//  component/login.js
import React from 'react';
```

```
import {Link, Redirect} from 'react-router-dom';
import '../css/login.css'
import UserService from '../service/user';
import {observer} from 'mobx-react';

const userService = new UserService();

export default class Login extends React.Component {
    render () {
        return <_Login service={userService} />;
    }
}

@observer
class _Login extends React.Component {
    handleClick(event) {
        event.preventDefault();
        let fm = event.target.form;
        this.props.service.login(
            fm[0].value, fm[1].value
        );
    }

    render() {
        if (this.props.service.loggedin) {
            return <Redirect to='/' />; //+ 跳转
        }
        return (<div className="login-page">
            <div className="form">
                <form className="login-form">
                    <input type="text" placeholder="邮箱" value='tom@magedu.com' />
                    <input type="password" placeholder="密码" value='abc' />
                    <button onClick={this.handleClick.bind(this)}>登录</button>
                    <p className="message">还未注册? <Link to="/reg">请注册</Link></p>
                </form>
            </div>
        </div>);
    }
}
```

注意，测试时，开启Django编写的后台服务程序。

测试成功，成功登录，写入Localstorage，也实现了跳转

前端开发

前端注册功能实现

在service/user.js中增加reg注册函数

```
import axios from 'axios';
import store from 'store';
import expire from 'store/plugins/expire';
import { observable } from 'mobx';

// 过期插件
store.addPlugin(expire);

export default class UserService {
  @observable loggedin = false; //+ 被观察者

  login(email, password) {
    console.log(email, password);

    axios.post('/api/user/login', {
      email: email,
      password: password
    })/* dev server会代理 */
      .then(
        response => { // 此函数要注意this的问题
          console.log(response.data);
          console.log(response.status);
          //+ 存储token，注意需要重开一次chrome的调试窗口才能看到
          store.set('token',
            response.data.token,
            (new Date()).getTime() + (8 * 3600 * 1000));
          this.loggedin = true; //+ 修改被观察者
        }
      ).catch(
        error => {
          console.log(error);
        }
      )
  }

  reg(name, email, password) {
    console.log(name, email, password);

    axios.post('/api/user/reg', {
      email: email,
      password: password,
      name:name
    })/* dev server会代理 */
```

```
      .then(
        response => { // 此函数要注意this的问题
          console.log(response.data);
          console.log(response.status);
          //+ 存储token，注意需要重开一次chrome的调试窗口才能看到
          store.set('token',
            response.data.token,
            (new Date()).getTime() + (8 * 3600 * 1000));
          this.loggedin = true; //+ 修改被观察者
        }
      ).catch(
        error => {
          console.log(error);
        }
      )
  }
}
```

Reg组件类

```
import React from 'react';
import '../css/login.css';
import {Link, Redirect} from 'react-router-dom';
import UserService from '../service/user';
import { observer } from 'mobx-react';

const userService = new UserService;

export default class Reg extends React.Component {
    render() {
        return <_Reg service={userService} />;
    }
}

@observer
class _Reg extends React.Component {
    validate(password, confirmpwd) {
        // 表单验证函数
        return password.value === confirmpwd.value;
    }

    handleClick(event) {
        event.preventDefault();
        const [name, email, password, confirmpwd] = event.target.form;
        console.log(this.validate(password, confirmpwd)); // +要验证表单数据后才发往后台
        this.props.service.reg(name.value, email.value, password.value);
    }
    render() {
        if (this.props.service.loggedin){ // 已经登录的用户不允许注册
            return <Redirect to="/" />;
        }
        return (
```

```
            <div className="login-page">
                <div className="form">
                <form className="register-form">
                    <input type="text" placeholder="姓名"/>
                    <input type="text" placeholder="邮箱"/>
                    <input type="password" placeholder="密码"/>
                    <input type="password" placeholder="确认密码"/>
                    <button onClick={this.handleClick.bind(this)}>注册</button>
                    <p className="message">如果已经注册<Link to='/login'>请登录</Link></p>
                </form>
                </div>
            </div>
        );
    }
}
```

测试

重新登录，登录成功后，点击注册链接，看看是否跳转到了首页？或先注册，注册成功后，点击登录，看看是否还能登录？

原因是什么？

原因就是构造并使用了不同的UserService实例。如何使用同一个？
在service/user.js中导出唯一的UserService实例即可，其他模块直接导入并使用这个实例即可。

```
// service/user.js
class UserService {
    // 此处省略
}

const userService = new UserService();

export {userService};
```

Ant Design

Ant Design 蚂蚁金服开源的 React UI 库。
官网 https://ant.design/index-cn
官方文档 https://ant.design/docs/react/introduce-cn

安装

```
$ npm install antd
```

使用

```
import { List } from 'antd'; // 加载antd的组件
import 'antd/lib/list/style/css';  // 加载 组件的样式 CSS

ReactDOM.render(<List />, mountNode);
```

每一种组件的详细使用例子，官网都有，需要时查阅官方文档即可

信息显示

网页开发中，不管操作成功与否，有很多提示信息，目前信息都是控制台输出，用户看不到。
使用Antd的message组件显示友好信息提示。

在service/user.js中增加一个被观察对象

```
import axios from 'axios';
import store from 'store';
import expire from 'store/plugins/expire';
import { observable } from 'mobx';

// 过期插件
store.addPlugin(expire);

class UserService {
  @observable loggedin = false; //+ 被观察者
  @observable errMsg = '' //+ 被观察者

  login(email, password) {
    console.log(email, password);

    axios.post('/api/user/login', {
      email: email,
      password: password
    })/* dev server会代理 */
      .then(
        response => { // 此函数要注意this的问题
          console.log(response.data);
          console.log(response.status);
          //+ 存储token，注意需要重开一次chrome的调试窗口才能看到
          store.set('token',
            response.data.token,
            (new Date()).getTime() + (8 * 3600 * 1000));
          this.loggedin = true; //+ 修改被观察者
        }
      ).catch(
        error => {
          console.log(error);
          this.errMsg = '登录失败' //+ 信息显示
        }
      )
  }
```

```
  reg(name, email, password) {
    console.log(name, email, password);

    axios.post('/api/user/reg', {
      email: email,
      password: password,
      name: name
    })/* dev server会代理 */
      .then(
        response => { // 此函数要注意this的问题
          console.log(response.data);
          console.log(response.status);
          //+ 存储token，注意需要重开一次chrome的调试窗口才能看到
          store.set('token',
            response.data.token,
            (new Date()).getTime() + (8 * 3600 * 1000));
          this.loggedin = true; //+ 修改被观察者
        }
      ).catch(
        error => {
          console.log(error);
          this.errMsg = '登录失败' //+ 信息显示
        }
      )
  }
}

const userService = new UserService();

export { userService };
```

component/_Login组件，增加Antd的message组件

```
import React from 'react';
import { Link, Redirect } from 'react-router-dom';
import '../css/login.css'
import { userService } from '../service/user';
import { observer } from 'mobx-react';
import { message } from 'antd';

import 'antd/lib/message/style';

export default class Login extends React.Component {
    render() {
        return <_Login service={userService} />;
    }
}

@observer
class _Login extends React.Component {
    handleClick(event) {
        event.preventDefault();
```

```
        let fm = event.target.form;
        this.props.service.login(
            fm[0].value, fm[1].value
        );
    }

    render() {
        if (this.props.service.loggedin) {
            return <Redirect to='/' />; //+ 跳转
        }

        if (this.props.service.errMsg) {
            message.info(this.props.service.errMsg, 3,
                setTimeout(() => this.props.service.errMsg = '', 1000))
        }

        return (<div className="login-page">
            <div className="form">
                <form className="login-form">
                    <input type="text" placeholder="邮箱" value='wayne@magedu.com' />
                    <input type="password" placeholder="密码" value='abc1' />
                    <button onClick={this.handleClick.bind(this)}>登录</button>
                    <p className="message">还未注册? <Link to="/reg">请注册</Link></p>
                </form>
            </div>
        </div>);
    }
}
```

上面代码执行到用户登录失败时，浏览器控制台会抛出一个警告

```
Warning: _renderNewRootComponent(): Render methods should be a pure function of props and state; triggering nested component updates from render is not allowed. If necessary, trigger nested updates in componentDidUpdate. Check the render method of _Login.
```

解决方法

将消息的清除代码移动到 componentDidUpdate 中。

```
//component/login.js
import React from 'react';
import { Link, Redirect } from 'react-router-dom';
import '../css/login.css'
import { userService } from '../service/user';
import { observer } from 'mobx-react';
import { message } from 'antd';

import 'antd/lib/message/style';

export default class Login extends React.Component {
    render() {
        return <_Login service={userService} />;
```

```
    }
}

@observer
class _Login extends React.Component {
    handleClick(event) {
        event.preventDefault();
        let fm = event.target.form;
        this.props.service.login(
            fm[0].value, fm[1].value
        );
    }

    render() {
        if (this.props.service.loggedin) {
            return <Redirect to='/' />; //+ 跳转
        }

        let em = this.props.service.errMsg;

        return (<div className="login-page">
            <div className="form">
                <form className="login-form">
                    <input type="text" placeholder="邮箱" value='wayne@magedu.com' />
                    <input type="password" placeholder="密码" value='abc1' />
                    <button onClick={this.handleClick.bind(this)}>登录</button>
                    <p className="message">还未注册? <Link to="/reg">请注册</Link></p>
                </form>
            </div>
        </div>);
    }

    componentDidUpdate(prevProps, prevState) { //+ 渲染后显示消息组件
        if (prevProps.service.errMsg) {
            message.info(prevProps.service.errMsg, 3, () => prevProps.service.errMsg = '');
        }
    }
}
```

component/_Reg组件同样增加message组件

```
// reg.js
import React from 'react';
import '../css/login.css';
import {Link, Redirect} from 'react-router-dom';
import { userService } from '../service/user';
import { observer } from 'mobx-react';

export default class Reg extends React.Component {
    render() {
        return <_Reg service={userService} />;
```

```
    }
}

@observer
class _Reg extends React.Component {
    validate(password, confirmpwd) {
        // 表单验证函数
        return password.value === confirmpwd.value;
    }

    handleClick(event) {
        event.preventDefault();
        const [name, email, password, confirmpwd] = event.target.form;
        console.log(this.validate(password, confirmpwd)); // +要验证表单数据后才发往后台
        this.props.service.reg(name.value, email.value, password.value);
    }
    render() {
        if (this.props.service.loggedin){ // 已经登录的用户不允许注册
            return <Redirect to="/" />;
        }

        let em = this.props.service.errMsg;

        return (
            <div className="login-page">
                <div className="form">
                <form className="register-form">
                    <input type="text" placeholder="姓名"/>
                    <input type="text" placeholder="邮箱"/>
                    <input type="password" placeholder="密码"/>
                    <input type="password" placeholder="确认密码"/>
                    <button onClick={this.handleClick.bind(this)}>注册</button>
                    <p className="message">如果已经注册<Link to='/login'>请登录</Link></p>
                </form>
                </div>
            </div>
        );
    }

    componentDidUpdate(prevProps, prevState) { //+ 渲染后显示消息组件
        if (prevProps.service.errMsg) {
            message.info(prevProps.service.errMsg, 3, () => prevProps.service.errMsg = '');
        }
    }
}
```

高阶组件装饰器

装饰器函数的整个演变过程如下

```
import React from 'react';

// 1 组件原型
class Reg extends React.Component {
    render() {
        return <_Reg service={service} />;
    }
}

// 2 匿名组件
const Reg = class extends React.Component {
    render() {
        return <_Reg service={service} />;
    }
};

// 3 提参数
function inject(Comp) {
    return class extends React.Component {
        render() {
            return <Comp service={service} />;
        }
    };
}

// 继续提参数
function inject(service, Comp) {
    return class extends React.Component {
        render() {
            return <Comp service={service} />;
        }
    };
}

// 4 props
function inject(obj, Comp) {
    return class extends React.Component {
        render() {
            return <Comp {...obj} />;
        }
    };
}

// 5 柯里化
function inject(obj) {
    function wrapper(Comp) {
```

```
        return class extends React.Component {
            render() {
                return <Comp {...obj} />;
            }
        };
    }
    return wrapper;
}

// 变形
function inject(obj) {
    return function wrapper(Comp) {
        return class extends React.Component {
            render() {
                return <Comp {...obj} />;
            }
        };
    };
}

// 6 箭头函数简化
const inject = obj => {
    return Comp => {
        return class extends React.Component {
            render() {
                return <Comp {...obj} />;
            }
        };
    };
}

// 继续简化
const inject = obj => Comp => {
    return class extends React.Component {
        render() {
            return <Comp {...obj} />;
        }
    };
};
/**
 * 本质上就是被包装的组件Comp作为了返回的新匿名组件的子组件
 * 换句话说，也就是原来的组件Comp被包装成一个新的组件，增强的功能通过obj属性注入
 */

// 7 函数式组件简化
const inject = obj => Comp => {
    return props => <Comp {...obj} />;
}

const inject = obj => Comp => props => <Comp {...obj} />;

const inject = obj => Comp => props => <Comp {...obj} {...props}/>;
```

新建src/utils.js，放入以下内容

```
import React from 'react';

const inject = obj => Comp => props => <Comp {...obj} {...props}/>;

export {inject};
```

将登陆、注册组件装饰一下

```
// reg.js修改如下
import React from 'react';
import { Link, Redirect } from 'react-router-dom';
import '../css/login.css'
import { observer } from 'mobx-react';
import { userService as service } from '../service/user';
import {inject} from '../utils';

// 注意装饰器顺序
@inject({service}) //+ 装饰器注入属性
@observer
export default class Reg extends React.Component {
    validatePwd(pwd1, pwd2) {
        return pwd1.value === pwd2.value
    }

    handleClick(event) {
        event.preventDefault();
        const [name, email, password, confirm] = event.target.form;
        console.log(this.validatePwd(password, confirm)) // +要验证表单数据后才发往后台
        this.props.service.reg(name.value, email.value, password.value);
    }

    render() {
        if (this.props.service.loggedin) return <Redirect to="/" />

        let em = this.props.service.errMsg; //+ 引用这个值留作观察

        return (
            <div className="login-page">
                <div className="form">
                    <form className="register-form">
                        <input type="text" placeholder="姓名" />
                        <input type="text" placeholder="邮箱" />
                        <input type="password" placeholder="密码" />
                        <input type="password" placeholder="确认密码" />
                        <button onClick={this.handleClick.bind(this)}>注册</button>
                        <p className="message">如果已经注册 <Link to="/login">请登录</Link></p>
                    </form>
                </div>
            </div>
        );
```

```
    }

    componentDidUpdate(nextProps, nextState) { //+ 渲染后显示消息组件
        if (nextProps.service.errMsg) {
            message.info(this.props.service.errMsg, 3, () => nextProps.service.errMsg = '');
        }
    }
}
```

```
// login.js修改如下
//component/login.js
import React from 'react';
import {Link, Redirect} from 'react-router-dom';
import '../css/login.css'
import {observer} from 'mobx-react';
import {userService as service} from '../service/user';
import { message } from 'antd'; // 增加消息组件
import {inject} from '../utils';
import 'antd/lib/message/style'; // 增加样式

// 注意装饰器顺序
@inject({service}) //+ 装饰器注入属性
@observer
export default class _Login extends React.Component {
    handleClick(event) {
        event.preventDefault();
        let fm = event.target.form;
        this.props.service.login(
            fm[0].value, fm[1].value
        );
    }

    render() {
        console.log("~~~~~~~~~~~")
        if (this.props.service.loggedin) {
            return <Redirect to='/' />; // 跳转
        }

        let em = this.props.service.errMsg; // 引用这个值留作观察

        return (<div className="login-page">
            <div className="form">
                <form className="login-form">
                    <input type="text" placeholder="邮箱" />
                    <input type="password" placeholder="密码" />
                    <button onClick={this.handleClick.bind(this)}>登录</button>
                    <p className="message">还未注册? <Link to="/reg">请注册</Link></p>
                </form>
            </div>
        </div>);
    }
```

```
    componentDidUpdate(nextProps, nextState) { //+ 渲染后显示消息组件
        if (nextProps.service.errMsg) {
            message.info(this.props.service.errMsg, 3, ()=>nextProps.service.errMsg='');
        }
    }
}
```

Mobx的observer装饰器有要求，所以装饰的顺序要注意一下

前端开发

导航菜单

菜单，https://ant.design/components/menu-cn/

Menu 菜单组件
　　mode有水平、垂直、内嵌
Menu.Item 菜单项
　　key 菜单项item的唯一标识

```
// src/index.js
import React from 'react';
import ReactDom from 'react-dom';
import { Route, Link, BrowserRouter as Router } from 'react-router-dom';
import {Menu, Icon} from 'antd';
import Login from './component/login';
import Reg from './component/reg';
import Pub from './component/pub'; // 发布页
//import L from './component/list'; // 列表页
//import Post from './component/post'; // 详情页

import 'antd/lib/menu/style';
import 'antd/lib/icon/style';

const Home = () => (
  <div>
    <h2>Home</h2>
  </div>
);

const About = () => (
  <div>
    <h2>About</h2>
  </div>
);

const App = () => (
  <Router>
    <div>
      <div>
        <Menu mode="horizontal">
          <Menu.Item key="home"><Link to="/"><Icon type="home" />主页</Link></Menu.Item>
          <Menu.Item key="login"><Link to="/login"><Icon type="login" />登录</Link></Menu.Item>
          <Menu.Item key="reg"><Link to="/reg">注册</Link></Menu.Item>
          <Menu.Item key="pub"><Link to="/">发布</Link></Menu.Item>
          <Menu.Item key="list"><Link to="/"><Icon type="bars" />文章列表</Link></Menu.Item>
          <Menu.Item key="about"><Link to="/about">关于</Link></Menu.Item>
        </Menu>
```

```
      </div>
      <Route path="/about" component={About} />
      <Route path="/login" component={Login} />
      <Route path="/reg" component={Reg} />
      <Route exact path="/" component={Home} />
    </div>
  </Router>
);

ReactDom.render(<App />, document.getElementById('root'));
```

布局

采用上中下布局，参考 https://ant.design/components/layout-cn/

```
import React from 'react';
import ReactDom from 'react-dom';
import { Route, Link, BrowserRouter as Router } from 'react-router-dom';
import {Layout, Menu, Icon} from 'antd';
import Login from './component/login';
import Reg from './component/reg';
import Pub from './component/pub'; // 发布页
// import L from './component/list'; // 列表页
// import Post from './component/post'; // 详情页

import 'antd/lib/layout/style';
import 'antd/lib/menu/style';
import 'antd/lib/icon/style';

const { Header, Content, Footer } = Layout; // 上中下

const Home = () => (
  <div>
    <h2>Home</h2>
  </div>
);

const About = () => (
  <div>
    <h2>About</h2>
  </div>
);

const App = () => (
  <Router>
    <Layout>
      <Header>
        <Menu mode="horizontal" theme="dark">
          <Menu.Item key="home"><Link to="/"><Icon type="home" />主页</Link></Menu.Item>
          <Menu.Item key="login"><Link to="/login"><Icon type="login" />登录</Link></Menu.Item>
          <Menu.Item key="reg"><Link to="/reg">注册</Link></Menu.Item>
          <Menu.Item key="pub"><Link to="/pub">发布</Link></Menu.Item>
          <Menu.Item key="list"><Link to="/"><Icon type="bars" />文章列表</Link></Menu.Item>
```

```
          <Menu.Item key="about"><Link to="/about">关于</Link></Menu.Item>
        </Menu>
      </Header>
      <Content style={{ padding: '8px 50px' }}>
        <div style={{ background: '#fff', padding: 24, minHeight: 280 }}>
          <Route path="/about" component={About} />
          <Route path="/login" component={Login} />
          <Route path="/reg" component={Reg} />
          <Route path="/pub" component={Pub} />
          <Route exact path="/" component={Home} />
        </div>
      </Content>
      <Footer style={{ textAlign: 'center' }}>
        马哥教育©2008-2018
      </Footer>
    </Layout>
  </Router>
);

ReactDom.render(<App />, document.getElementById('root'));
```

注意，`<Menu.Item key="list"><Link to="/"><Icon type="bars" />文章列表</Link></Menu.Item>`这里Link需要包着Icon，否则会错位。

博文业务

url	method	说明
/post/pub	POST	提交博文的title、content，成功返回json，post_id
/post/id	GET	返回博文详情，返回json，post_id、title、author、author_id、postdate（时间戳）、content
/post/	GET	返回博文标题列表

业务层

创建service/post.js文件，新建PostService类。

```
import axios from 'axios';
import { observable } from 'mobx';


class PostService{
    @observable msg = '';

    pub(title, content) {
        console.log(title);
        axios.post('/api/post/pub', {
            title, content
        })/* dev server会代理 */
```

```
        .then(
            response => { // 此函数要注意this的问题
                console.log(response.data);
                console.log(response.status);
                this.msg = '博文提交成功';
            }
        ).catch(
            error=> {
                console.log(error);
                this.msg = '博文提交失败'; //+ 信息显示
            }
        )
    }
}


const postService = new PostService();
export {postService};
```

发布组件

使用Form组件，https://ant.design/components/form-cn/

```
// component/pub.js
import React from 'react';
import { observer } from 'mobx-react';
import { Form, Icon, Input, Button, message } from 'antd';
import {inject} from '../utils';
import {postService as service} from '../service/post';

import 'antd/lib/form/style';
import 'antd/lib/icon/style';
import 'antd/lib/input/style';
import 'antd/lib/button/style';
import 'antd/lib/message/style';

const FormItem = Form.Item;
const { TextArea } = Input;

@inject({service})
@observer
export default class Pub extends React.Component {
    handleSubmit(event){
        event.preventDefault();
        const [title, content] = event.target; // event.target返回form, 而form是表单控件的数组
        this.props.service.pub(title.value, content.value);
    }

    render() {
        let msg = this.props.service.msg;
        return (
            <Form layout="vertical" onSubmit={this.handleSubmit.bind(this)}>
```

```
                <FormItem label="标题" labelCol={{ span: 4 }} wrapperCol={{ span: 14 }}>
                    <Input placeholder="标题" />
                </FormItem>
                <FormItem label="内容" labelCol={{ span: 4 }} wrapperCol={{ span: 14 }}>
                    <TextArea rows={10} />
                </FormItem>
                <FormItem wrapperCol={{ span: 14, offset: 4 }}>
                    <Button type="primary" htmlType="submit">提交</Button>
                </FormItem>
            </Form>
        );
    }

    componentDidUpdate(prevProps, prevState) { //+ 渲染后显示消息组件
        if (prevProps.service.msg) {
            message.info(prevProps.service.msg, 3, ()=>prevProps.service.msg='');
        }
    }
}
```

- Form 表单组件，layout是垂直，onSubmit提交，注意这个提交的this就是表单自己
- FormItem 表单项
  - label设置控件前的标题
  - labelCol设置label的宽度，wrapperCol是label后占用的宽度，这些单位都是栅格系统的宽度
  - 参考 Antd/Form表单/表单布局 https://ant.design/components/form-cn/#components-form-demo-layout
  - 栅格系统：AntD提供了一个类似于Bootstrap的栅格系统，它将页面分成了24等分的列
  - span代表占几个格子；offset表示左边空出多少个格子
- Input 输入框，placeholder提示字符
- TextArea 文本框，rows行数
- Button 按钮
  - type是按钮的类型，也决定它的颜色
  - htmlType使用HTML中的type值，submit是提交按钮会触发提交行为，一定要写，但是handleSubmit中要阻止默认行为。

富文本编辑器(可选)

https://github.com/margox/braft-editor

```
使用yarn安装
$ yarn add braft-editor
使用npm安装
$ npm install braft-editor --save

组件使用
import BraftEditor from 'braft-editor';
import 'braft-editor/dist/index.css';
```

业务层改进

header中的Jwt
由于与后台Django Server通信，身份认证需要Jwt，这个要放到request header中。使用axios的API

`axios.post(url[, data[, config]])`可以使用第三个参数config。config是一个对象，字典中设置headers字段，该字段的值依然是对象，都是键值对形式。

```
// service/post.js
import axios from 'axios';
import { observable } from 'mobx';
import store from 'store';

class PostService {
    constructor() {
        this.axios = axios.create({
            baseURL: '/api/post/'
        });
    }

    @observable msg = '';

    getJwt(){
        return store.get('token', null);
    }

    pub(title, content) {
        console.log(title);
        axios.post('/api/post/pub', {
            title, content
        }, {
            headers: {'Jwt': this.getJwt()}
        })/* dev server会代理 */
            .then(
                response => { // 此函数要注意this的问题
                    console.log(response.data);
                    console.log(response.status);
                    this.msg = '博文提交成功';
                }
            ).catch(
                error => {
                    console.log(error);
                    this.msg = '博文提交失败'; //+ 信息显示
                }
            )
    }
}

const postService = new PostService();
export { postService };
```

编写文章内容并提交，注意jwt过期问题。

文章列表页组件

创建component/list.js，创建List组件。在index.js中提交菜单项和路由。

使用L是为了避免和AntD的List冲突。

```
// index.js
import L from './component/list'; // 列表页

const App = () => (
  <Router>
    <Layout>
      <Header>
        <Menu mode="horizontal" theme="dark">
          <Menu.Item key="home"><Link to="/"><Icon type="home" />主页</Link></Menu.Item>
          <Menu.Item key="login"><Link to="/login"><Icon type="login" />登录</Link></Menu.Item>
          <Menu.Item key="reg"><Link to="/reg">注册</Link></Menu.Item>
          <Menu.Item key="pub"><Link to="/pub">发布</Link></Menu.Item>
          <Menu.Item key="list"><Link to="/list"><Icon type="bars" />文章列表</Link></Menu.Item>
          <Menu.Item key="about"><Link to="/about">关于</Link></Menu.Item>
        </Menu>
      </Header>
      <Content style={{ padding: '8px 50px' }}>
        <div style={{ background: '#fff', padding: 24, minHeight: 280 }}>
          <Route path="/about" component={About} />
          <Route path="/login" component={Login} />
          <Route path="/reg" component={Reg} />
          <Route path="/list" component={L} />
          <Route path="/pub" component={Pub} />
          <Route exact path="/" component={Home} />
        </div>
      </Content>
      <Footer style={{ textAlign: 'center' }}>
        马哥教育©2008-2018
      </Footer>
    </Layout>
  </Router>
);
```

查询字符串处理

用户请求的URL是`http://127.0.0.1:3000/list?page=2`，url要被转换成`/api/post/?page=2`，如何提取查询字符串?
现在前端路由有react-router管理，它匹配路径后，才会路由。它提供了匹配项，它将匹配的数据注入组件的props中，也可以使用解构提取`const { match, location } = this.props`。
location也是一个对象，pathname表示路径，search表示查询字符串。`{pathName:"/list", search:"?page=2"}`。拿到查询字符串后，可以使用URLSearchParams解析它，但是它是实验性的，不建议用在生产环境中。本次将查询字符串直接拼接发往后端，由Django服务器端判断。

```
var params = new URLSearchParams(url.search);
console.log(params.get('page'), params.get('size'))
```

参考 https://reacttraining.com/react-router/core/api/location

List组件

Ant design的List，需要使用3.x版本，修改package.json的版本信息`"antd": "^3.1.5"`。
然后`$ npm update`，更新成功后，就可以使用List组件了。
或者`$ yarn upgrade antd@^3.1.5`。
注意：npm安装和yarn安装不要混用。

component/list.js代码如下

```
import React from 'react';
import { inject } from '../utils';
import { postService as service } from '../service/post';
import { List } from 'antd';
import { observer } from 'mobx-react';

import 'antd/lib/list/style';


@inject({ service })
@observer
export default class L extends React.Component {
    constructor(props) {
        super(props);
        // 不处理查询字符串，传到后台服务提取处理
        props.service.list(props.location.search); // ?page=1
    }

    render() {
        const {posts:data, pagination} = this.props.service.posts;
        console.log(data)
        if (data.length)
            return (
                <List
                    header="博文列表"
                    footer={<div>Footer</div>}
                    bordered
                    dataSource={data}
                    renderItem={item => (<List.Item>{item.title}</List.Item>)}
                />
            );
        else
            return (<div>无数据</div>);
    }
}
```

List 列表组件
bordered 有边线
dataSource 给定数据源
renderItem 渲染每一行，给定一个一参函数迭代每一行
List.Item 每一行的组件

使用Link组件增加链接

```
    <List bordered={true} dataSource={data} renderItem={
          item => (<List.Item>
              <Link to={'/post/' + item.post_id}>{item.title}</Link>
            </List.Item>)
     } />
```

如果需要根据复杂的效果可以这样

```
      <List bordered={true} dataSource={data} renderItem={
           item => (<List.Item>
              <List.Item.Meta title={<Link to={'/post/' + item.post_id}>{item.title}</Link>} />
            </List.Item>)
      } />
```

```
<Link to={'/post/' + item.post_id}>{item.title}</Link> 这是详情页的链接
```

PostService代码如下

```
import axios from 'axios';
import { observable } from 'mobx';
import store from 'store';

class PostService {
    constructor() {
        this.axios = axios.create({
            baseURL: '/api/post/'
        });
    }

    @observable msg = '';
    // 博文列表,分页信息
    @observable posts = {'posts':[], 'pagination':{page:1, size:20, count:0, pages:0}}

    getJwt(){
        return store.get('token', null);
    }

    pub(title, content) {
        console.log(title);
        axios.post('/api/post/pub', {
            title, content
```

```
        }, {
            headers: {'Jwt': this.getJwt()}
        })/* dev server会代理 */
            .then(
                response => { // 此函数要注意this的问题
                    console.log(response.data);
                    console.log(response.status);
                    this.msg = '博文提交成功';
                }
            ).catch(
                error => {
                    console.log(error);
                    this.msg = '博文提交失败'; //+ 信息显示
                }
            )
    }

    list(search) {
        this.axios.get(search).then(response => {
            this.posts = {
                'posts':response.data.posts,
                'pagination':response.data.pagination // 分页信息
            };
        }).catch(error => {
            console.log(error);
            this.msg = '文章列表加载失败'; //+ 信息显示
        });
    }
}

const postService = new PostService();
export { postService };
```

测试 http://localhost:3000/list?page=2&size=2

分页功能

分页还是需要解析查询字符串的，因此写一个解析函数，把这个函数放入utils.js中，需要时调用。

```
let url = '?id=5&page=1&size=20&id=&age-20&name=abc&name=汤姆=&测试=1'

function parse_qs(qs, re=/(\w+)=([^&]+)/){
    let obj = {};
    if (qs.startsWith('?'))
        qs = qs.substr(1)
    console.log(qs);
    qs.split('&').forEach(element => {
        let match = re.exec(element);
        //console.log(match)
        if (match) obj[match[1]] = match[2];
    });
    return obj;
```

```
}


console.log(parse_qs(url))
```

分页使用了Pagination组件，在L组件的render函数的List组件中使用pagination属性，这个属性内放入一个pagination对象，有如下属性

- current，当前页
- pageSize， 页面内行数
- total，记录总数
- onChange，页码切换时调用，回调函数为(pageNo, pageSize) => {}，即切换是获得当前页码和页内行数。

可以参考 https://ant.design/components/list-cn/#components-list-demo-vertical

component/list.js代码修改如下

```
import React from 'react';
import { inject } from '../utils';
import { postService as service } from '../service/post';
import { List, Pagination } from 'antd';
import { observer } from 'mobx-react';
import {Link} from 'react-router-dom';

import 'antd/lib/list/style';
import 'antd/lib/pagination/style';


@inject({ service })
@observer
export default class L extends React.Component {
    constructor(props) {
        super(props);
        // 不处理查询字符串，传到后台服务提取处理
        props.service.list(props.location.search); // ?page=1
    }

    handleChange(page, pageSize) {
        // 不管以前查询字符串是什么，重新拼接 查询字符串 向后传
        let search = `?page=${page}&size=${pageSize}`;
        this.props.service.list(search);
    }

    render() {
        const {posts:data=[], pagination={}} = this.props.service.posts;

        if (data.length) {
            const {page:current=1, count:total=0, size:pageSize=20} = pagination;

            return (
                <List
                    header="博文列表"
                    bordered
```

```
                    dataSource={data}
                    renderItem={item => (<List.Item>
                        <Link to={'/post/' + item.post_id}>{item.title}</Link>
                    </List.Item>)}
                    pagination={{
                        current:current,
                        total:total,
                        pageSize:pageSize,
                        onChange:this.handleChange.bind(this)
                    }}
                />
            );
        }
        else
            return (<div>无数据</div>);
    }
}
```

测试可以切换页面。但是鼠标放到左右两端发现上一页、下一页是英文，如何修改？国际化。

国际化

index.js修改如下(部分代码)

```
import { LocaleProvider } from 'antd';
import zhCN from 'antd/lib/locale-provider/zh_CN';

ReactDom.render(
  <LocaleProvider locale={zhCN}>
    <App />
  </LocaleProvider>
  , document.getElementById('root'));
```

将App这个根组件包裹住就行了，再看分页组件就显示中文了。

浏览器地址不变的问题（可选）

基本上没有什么问题了，但是，如果在地址栏里面输入`http://127.0.0.1:3000/list?size=2&page=2`后，再切换分页，地址栏URL不动，不能和当前页一致。
这个问题的解决有一定的难度。需要定义Pagination的itemRender属性，定义一个函数，这个函数有3个参数

- current，当前pageNo
- type，当前类型，有三种可能，上一页为prev，下一页为next，页码为page
- originalElement，不要动这个参数，直接返回就行了

```
const itemRender = (page, type: 'page' | 'prev' | 'next', originalElement) => React.ReactNode
```

```
// component/list.js
import React from 'react';
import { inject } from '../utils';
```

```
import { postService as service } from '../service/post';
import { List, Pagination } from 'antd';
import { observer } from 'mobx-react';
import {Link} from 'react-router-dom';

import 'antd/lib/list/style';
import 'antd/lib/pagination/style';

@inject({ service })
@observer
export default class L extends React.Component {
    constructor(props) {
        super(props);
        // 不处理查询字符串，传到后台服务提取处理
        props.service.list(props.location.search); // ?page=1
    }

    handleChange(page, pageSize) {
        // 不管以前查询字符串是什么，重新拼接 查询字符串 向后传
        let search = `?page=${page}&size=${pageSize}`;
        this.props.service.list(search);
    }

    getUrl(c){
        let obj = parse_qs(this.props.location.search);
        let {size=20} = obj;
        return `/list?page=${c}&size=${size}`;
    }

    itemRender(page, type, originalElement){
        console.log(originalElement)
        if (type === 'page')
            return <Link to={this.getUrl(page)}>{page}</Link>;
        return originalElement;
    }

    render() {
        const {posts:data=[], pagination={}} = this.props.service.posts;

        if (data.length) {
            const {page:current=1, count:total=0, size:pageSize=20} = pagination;

            return (
                <List
                    header="博文列表"
                    bordered
                    dataSource={data}
                    renderItem={item => (<List.Item>
                        <Link to={'/post/' + item.post_id}>{item.title}</Link>
                    </List.Item>)}
                    pagination={{
                        current:current,
                        total:total,
```

```
                        pageSize:pageSize,
                        onChange:this.handleChange.bind(this),
                        itemRender:this.itemRender.bind(this)
                    }}
                />
            );
        }
        else
            return (<div>无数据</div>);
    }
}
```

基本解决问题。但是，上一页、下一页点击不能改变浏览器地址栏。

```
import React from 'react';
import { inject } from '../utils';
import { postService as service } from '../service/post';
import { List, Pagination } from 'antd';
import { observer } from 'mobx-react';
import {Link} from 'react-router-dom';

import 'antd/lib/list/style';


@inject({ service })
@observer
export default class L extends React.Component {
    constructor(props) {
        super(props);
        // 不处理查询字符串，传到后台服务提取处理
        props.service.list(props.location.search); // ?page=1
    }

    handleChange(page, pageSize) {
        // 不管以前查询字符串是什么，重新拼接 查询字符串 向后传
        let search = `?page=${page}&size=${pageSize}`;
        this.props.service.list(search);
    }

    getUrl(c){
        let obj = parse_qs(this.props.location.search);
        let {size=20} = obj;
        return `/list?page=${c}&size=${size}`;
    }

    itemRender(page, type, originalElement){
        console.log(originalElement)
        if (page == 0) return originalElement;
        if (type === 'page')
            return <Link to={this.getUrl(page)}>{page}</Link>;
        if (type === 'prev')
```

```
            return <Link to={this.getUrl(page)} className="ant-pagination-item-link">&lt;
</Link>;
        if (type === 'next')
            return <Link to={this.getUrl(page)} className="ant-pagination-item-link">&gt;
</Link>;
    }

    render() {
        const {posts:data=[], pagination={}} = this.props.service.posts;

        if (data.length) {
            const {page:current=1, count:total=0, size:pageSize=20} = pagination;

            return (
                <List
                    header="博文列表"
                    bordered
                    dataSource={data}
                    renderItem={item => (<List.Item>
                        <Link to={'/post/' + item.post_id}>{item.title}</Link>
                    </List.Item>)}
                    pagination={{
                        current:current,
                        total:total,
                        pageSize:pageSize,
                        onChange:this.handleChange.bind(this),
                        itemRender:this.itemRender.bind(this)
                    }}
                />
            );
        }
        else
            return (<div>无数据</div>);
    }
}
```

至此，分页问题基本解决。

详情页组件

index.jsp

```
import Post from './component/post'; // 详情页

<Route exact path="/post/:id" component={Post} />
```

新建component/post.js，创建Post组件。使用antd Card布局。

```
import React from 'react';
```

```
import { inject } from '../utils';
import { postService as service } from '../service/post';
import { observer } from 'mobx-react';
import {message, Card} from 'antd';

import 'antd/lib/message/style';
import 'antd/lib/card/style';

@inject({ service })
@observer
export default class Post extends React.Component {
    constructor(props) {
        super(props);
        let { id = -1 } = props.match.params;
        this.props.service.getPost(id);
    }
    render() {
        let msg = this.props.service.msg;
        const {title="", content="", author, postdate} = this.props.service.post;
        if (title) {
            return (<Card title={title} bordered={false} style={{ width: 300 }}>
            <p>{author} {new Date(postdate*1000).toLocaleDateString()}</p>
            <p>{content}</p>
          </Card>);
        }
        else
            return (<div>无数据</div>);
    }

    componentDidUpdate(prevProps, prevState) { //+ 渲染后显示消息组件
        if (prevProps.service.msg) {
            message.info(prevProps.service.msg, 3, ()=>prevProps.service.msg='');
        }
    }
}
```

至此，前后端分离的博客系统基本框架搭好了，去看看页面的成果。

service/post.js代码如下

```
import axios from 'axios';
import { observable } from 'mobx';
import store from 'store';

class PostService {
    constructor() {
        this.axios = axios.create({
            baseURL: '/api/post/'
        });
    }

    @observable msg = '';
    // 博文列表,分页信息
```

```
    @observable posts = {'posts':[], 'pagination':{page:1, size:20, count:0, pages:0}}

    @observable post = {}; // 文章

    getJwt(){
        return store.get('token', null);
    }

    pub(title, content) {
        console.log(title);
        axios.post('/api/post/pub', {
            title, content
        }, {
            headers: {'Jwt': this.getJwt()}
        })/* dev server会代理 */
            .then(
                response => { // 此函数要注意this的问题
                    console.log(response.data);
                    console.log(response.status);
                    this.msg = '博文提交成功';
                }
            ).catch(
                error => {
                    console.log(error);
                    this.msg = '博文提交失败'; //+ 信息显示
                }
            )
    }

    list(search) {
        this.axios.get(search).then(response => {
            this.posts = {
                'posts':response.data.posts,
                'pagination':response.data.pagination // 分页信息
            };
        }).catch(error => {
            console.log(error);
            this.msg = '文章列表加载失败'; //+ 信息显示
        });
    }

    getPost(id) {
        this.axios.get(id).then(response => {
            this.post = response.data.post;
        }).catch(error => {
            console.log(error);
            this.post = {}
            this.msg = '文章加载失败'; //+ 信息显示
        });
    }
}

const postService = new PostService();
```

```
export { postService };
```

如果使用了富文本编辑器，那么显示的时候，发现不能按照网页标签显示。

原因是为了安全，防止XSS攻击，React不允许直接按照HTML显示。

使用dangerouslySetInnerHTML属性，这个名字提醒使用者很危险。

```
<p>{content}</p>
修改为
<p dangerouslySetInnerHTML={{__html:content}}></p>
```

Session

Session-Cookie机制

网景公司发明了Cookie技术，为了解决浏览器端数据存储问题。
每一次request请求时，会把此域名相关的Cookie发往服务器端。服务器端也可以使用response中的set-cookie来设置cookie值。

动态网页技术，也需要知道用户身份，但是HTTP是无状态协议，无法知道。必须提出一种技术，让客户端提交的信息可以表明身份，而且不能更改。这就是Session技术。
Session开启后，会为浏览器端设置一个Cookie值，即SessionID。
这个放置SessionID的Cookie是会话级的，浏览器不做持久化存储只放在内存中，并且浏览器关闭自动清除。
浏览器端发起HTTP请求后，这个SessionID会通过Cookie发到服务器端，服务器端就可以通过这个ID查到对应的一个字典结构。如果查无此ID，就为此浏览器重新生成一个SessionID，为它建立一个SessionID和空字典的映射关系。
可以在这个ID对应的Session字典中，存入键值对来保持与当前会话相关的信息。

- Session会定期过期清除
- Session占用服务器端内存
- Session如果没有持久化，如果服务程序崩溃，那么所有Session信息丢失
- Session可以持久化到数据库中，如果服务程序崩溃，那么可以从数据库中恢复

开启session支持

Django可以使用Session

- 在settings中，MIDDLEWARE设置中，启用'django.contrib.sessions.middleware.SessionMiddleware'。
- INSTALLED_APPS设置中，启用'django.contrib.sessions'。它是基于数据库存储的Session。
- Session不使用，可以关闭上述配置，以减少开销。
- 在数据库的表中的django_session表，记录session信息。但可以使用文件系统或其他cache来存储

登录登出实现

登录实现

原来登录成功，会使用Jwt的token发往客户端。现在不需要了，只需要在Session中记录登录信息即可。

```
def login(request:HttpRequest):
    # post json
    try:
        payload = simplejson.loads(request.body)
        email = payload['email']

        user = User.objects.get(email=email) # only one

        # 验证密码
        if bcrypt.checkpw(payload['password'].encode(), user.password.encode()):
            # 注释掉原来的jwt token代码
            # token = gen_token(user.id)
```

```
            # res = JsonResponse({
            #     'user':{
            #         'user_id':user.id,
            #         'name':user.name,
            #         'email':user.email,
            #     }, 'token':token
            # })
            # res.set_cookie('jwt', token)
            s = request.session
            # 只有登录成功才会在session中保存信息
            s.set_expiry(300) # 设置过期时长
            s['user_id'] = user.id # 用于判断是否登录
            s['user_info'] = '[**{} {} {}**]'.format(user.id, user.name, user.email)
            #s['user'] = user # user不可以json序列化，会报错
            return JsonResponse({
                'user':{
                    'user_id':user.id,
                    'name':user.name,
                    'email':user.email,
                }, 'user_info':s['user_info']
            })
        else:
            return HttpResponseBadRequest()

    except Exception as e:
        print(e)
        return HttpResponseBadRequest()
```

建议不要在Session中保存太多数据，也不要保存过于复杂的类型。

认证实现

取消原有判断HTTP header中是否提供了JWT信息，改为判断该SessionID是否能找到一个字典，这个字典中是否有登录成功后设置的user_id键值对信息。

```
def authenticate(viewfunc):
    def wrapper(request:HttpRequest):
        # 认证检测 "HTTP_JWT"
        try:
            # auth = request.META["HTTP_JWT"] # 会被加前缀HTTP_且全大写
            # payload = jwt.decode(auth, settings.SECRET_KEY, algorithms=[settings.ALG])# 篡改，
过期
            # print(payload, '~~~~~~~~~~~~~~~~~~')
            # print(datetime.datetime.now().timestamp(), '~~~~~~~~~~~~~~~')

            print('~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~')
            print(type(request.session), request.session)
            payload: SessionStore = request.session

            print(payload.items())
            print(payload['user_id']) # 登录成功才会有
```

```
            user = User.objects.get(pk=payload['user_id'])  # 以后要注意查询条件

            request.user = user
            print(user)
            print('~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~')

        except Exception as e:
            print(e)
            return HttpResponse(status=401)

        ret = viewfunc(request)# 调用视图函数
        # 特别注意view调用的时候，里面也有返回异常
        return ret
    return wrapper
```

登出代码

```
@authenticate # 没有登录成功过，就不用登出了，所以要认证
def logout(request):
    s = '{} logout ok.'.format(request.session['user_id'])
    #del request.session['user_id'] # 不会清除数据库中记录
    request.session.flush()  # 清空当前session， 删除对应表记录
    return HttpResponse(s)
```

登出时，需要调用flush方法，清除session，清除数据库记录。

登录成功，为当前session在django_session表中增加一条记录，如果没有登出操作，那么该记录不会消失。
Django也没有自动清除失效记录的功能。

但Django提供了一个命令clearsessions，建议放在cron中定期执行。

```
django-admin.py clearsessions
manage.py clearsessions
```

不需要认证的view函数中使用Session

在当前会话中，每次request请求中都会得到浏览器端发出的SessionID，因此都可以在服务器端找到该ID对应的Session字典，可以使用request.session访问。
所有请求都可以使用request.session对象，ID找不到可以认为返回个空字典。

```
def test1(request): # 视图函数，需要配置url映射
    if request.session.get('user_id'): # 访问Session字典中的值
        print(request.session.items())
        return HttpResponse('test1 ok')
    else:
        return HttpResponseBadRequest() # 没有此信息，说明没有登录成功过
```

部署

Django打包

构建setup.py文件

```
from distutils.core import setup
import glob

setup(name='blog',
      version='1.0',
      description='magedu blog',
      author='Wayne',
      author_email='wayne@magedu.com',
      url='http://www.magedu.com/python',
      packages=['blog', 'post', 'user'],
      py_modules=['manage'], # 可以不打包manage.py
      data_files=glob.glob('templates/*.html') + ['requirements']
     )
```

```
# 应用程序的根目录下打包
$ pip freeze > requirements
$ python setup.py sdist --formats=gztar # gz
```

在Linux系统中创建一个python虚拟环境目录，使用

```
$ pyenv virtualenv 3.5.3 blog353
$ pyenv local blog353
$ pip list
$ tar xf blog-1.0.tar.gz
$ mv blog-1.0 blog # Django应用部署目录
$ cd blog

CentOS需要root权限yum安装，mysqlclient依赖
# yum install python-devel mysql-devel

$ pip install -r requirements

$ pip list
Package     Version
----------- -------
bcrypt      3.1.4
cffi        1.11.5
Django      1.11
mysqlclient 1.3.13
pip         10.0.1
pycparser   2.18
```

```
PyJWT       1.6.4
pytz        2018.5
setuptools  28.8.0
simplejson  3.16.0
six         1.11.0

$ sed -i -e 's/DEBUG.*/DEBUG = False/' -e 's/ALLOWED_HOSTS.*/ALLOWED_HOSTS = ["*"]/'
blog/settings.py # 修改Django配置
$ python manage.py runserver 0.0.0.0:8000 # 测试
```

使用`http://192.168.142.135:8000/post?page=2&size=2`成功返回数据，说明Django应用成功

至此，Django应用部署完成。Django带了个开发用Web Server，生成环境不用，需要借助其它Server。

注意：ALLOWED_HOSTS = ["*"] 这是所有都可以访问，生产环境应指定具体可以访问的IP，而不是所有。

WSGI

Web Server Gateway Interface，是Python中定义的Web Server与应用程序的接口定义。

应用程序有WSGI 的 Django框架负责，WSGI Server谁来做?

uWSGI 项目

uWSGI是一个C语言的项目，提供一个WEB服务器，它支持WSGI协议，可以和Python的WSGI应用程序通信。

官方文档 https://uwsgi-docs.readthedocs.io/en/latest/

uWSGI可以直接启动HTTP服务，接收HTTP请求，并调用Django应用。

安装

```
$ pip install uwsgi
$ uwsgi --help
```

uWSGI + Django部署

在Django项目根目录下，运行`$ uwsgi --http :8000 --wsgi-file blog/wsgi.py --stats :8001 --stats-http`，使用下面的链接测试

http://192.168.142.135:8000/
http://192.168.142.135:8000/post/?page=1&size=2

运行正常。

stats能够显示服务器状态值。--stats-http选项可以使用http访问这个值。
安装uwsgitop获取这个stat值。注意使用这个命令不要使用--stats-http选项。

```
$ pip install uwsgitop
$ uwsgitop --frequency 10 127.0.0.1:8001
```

React项目打包

rimraf 递归删除文件，rm -rf

`$ npm install rimraf --save-dev` 或者 `$ yarn add rimraf --dev`

在package.json中替换

`"build": "rimraf dist && webpack -p --config webpack.config.prod.js"`

`$ npm run build` 或者 `$ yarn run build` 编译，成功。查看项目目录中的dist目录。

nginx uwsgi 部署

tengine安装

淘宝提供的nginx

```
# yum install gcc openssl-devel pcre-devel -y

# tar xf tengine-1.2.3
# cd tengine-1.2.3
# ./configure --help | grep wsgi
# ./configure # 第一步
# make && make install # 第二步、第三步
# cd /usr/local/nginx/ # 默认安装位置
```

编译安装过程中，已经看到安装了fastcgi、uwsgi模块。

nginx配置

```
    server {
        listen       80;
        server_name  localhost;

        #charset koi8-r;
        #access_log  logs/host.access.log  main;

        location ^~ /api/ {
            rewrite ^/api(/.*) $1 break;
            proxy_pass http://127.0.0.1:8000;
        }

        location / {
            root   html;
            index  index.html index.htm;
        }
    }
```

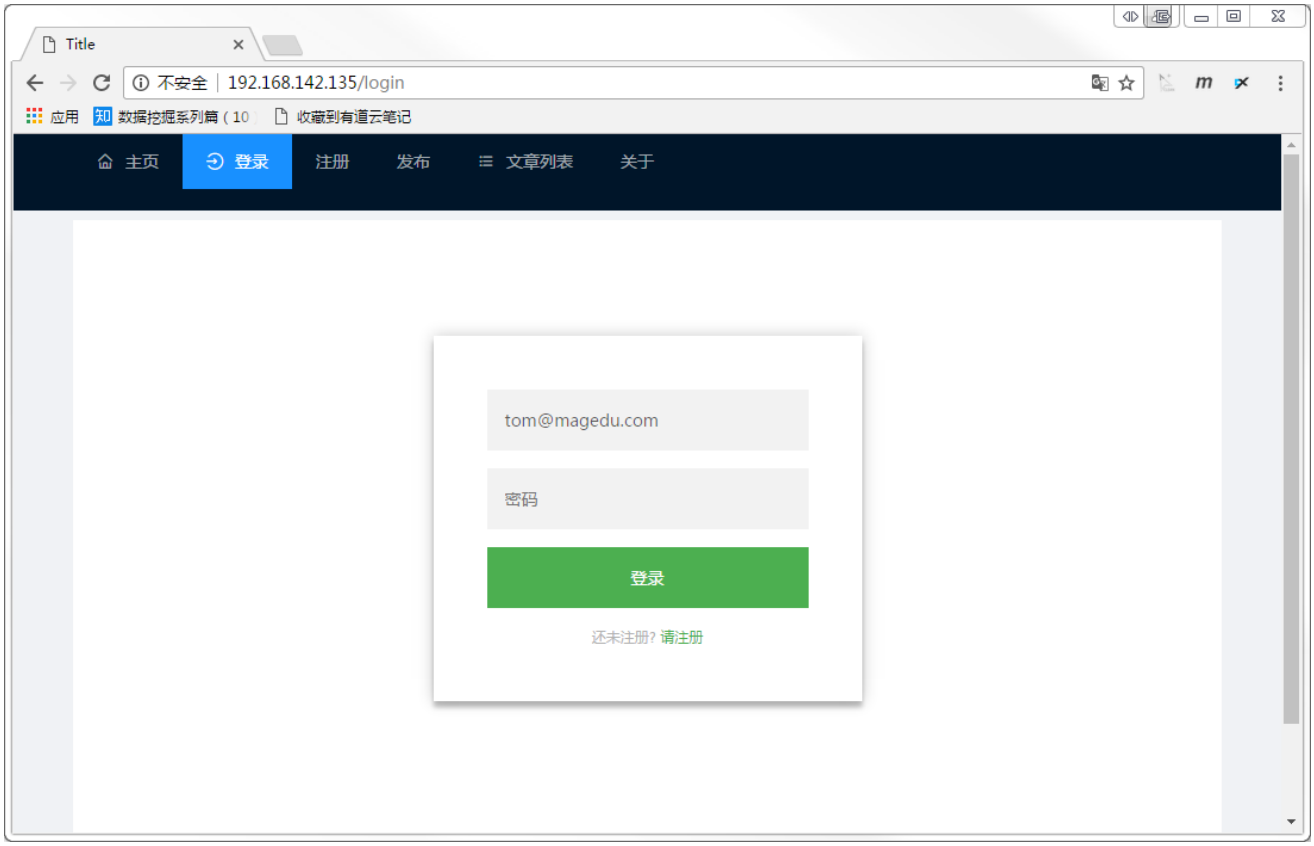
`^~ /api/` 左前缀匹配
`rewrite ^/api(/.*) $1 break;` 重写请求的path

```
# pwd
/usr/local/nginx

# sbin/nginx
```

访问`http://192.168.142.135/`，可以看到nginx的首页了。

部署博客前端系统

将编译好的前端项目文件index.htm复制到/usr/local/nginx/html， app-xxx.js复制到/usr/local/nginx/html/assets。刷新首页，一切从此开始。



uwsgi部署

目前nginx和uWSGI直接使用的是HTTP通信，效率低。改为使用uwsgi通信。

使用uwsgi协议的命令行写法如下

`$ uwsgi --socket :9000 --wsgi-file blog/wsgi.py`

本次pyenv的虚拟目录是/home/python/magedu/projects/web，将Django项目所有项目文件和目录放在这个目录下面。
uWSGI的配置文件blog.ini也放在这个目录中。

```
[python@node web]$ pwd
/home/python/magedu/projects/web
[python@node web]$ tree -d
.
├── blog
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
├── blog.ini
├── post
├── requirements
├── setup.py
├── static
│   ├── jquery.js
│   └── t.html
├── templates
│   ├── index.html
└── user
```

blog.ini配置如下

```
[uwsgi]
socket = 127.0.0.1:9000
chdir = /home/python/magedu/projects/web
wsgi-file = blog/wsgi.py
```

配置项	说明
socket = 127.0.0.1:9000	使用uwsgi协议通信
chdir = /home/python/magedu/projects/web	Django项目根目录
wsgi-file = blog/wsgi.py	指定App文件，blog下wsgi.py

```
(blog353) [python@node web]$ pwd
/home/python/magedu/projects/web
(blog353) [python@node web]$ vim blog.ini
(blog353) [python@node web]$ cat blog.ini
[uwsgi]
socket = 127.0.0.1:9000
chdir = /home/python/magedu/projects/web
wsgi-file = blog/wsgi.py
$ uwsgi blog.ini
```

在nginx中配置uwsgi

http://nginx.org/en/docs/http/ngx_http_uwsgi_module.html

```
    server {
        listen       80;
        server_name  localhost;

        #charset koi8-r;
        #access_log  logs/host.access.log  main;

        location ^~ /api/ {
            rewrite ^/api(/.*) $1 break;
            #proxy_pass http://127.0.0.1:8000
            include    uwsgi_params;
            uwsgi_pass 127.0.0.1:9000;
        }

        location / {
            root   html;
            index  index.html index.htm;
        }
    }
```

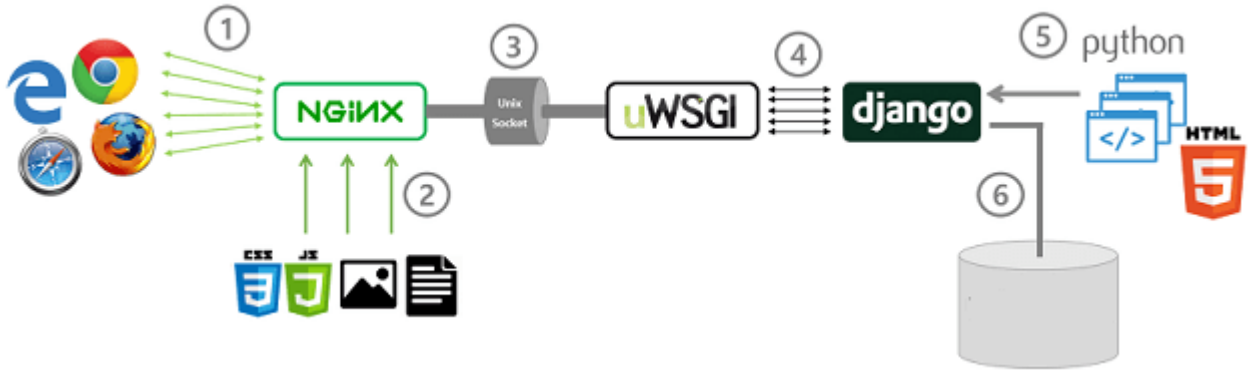
重新装载nginx配置文件，成功运行。

至此，前后端分离的开发、动静分离的部署的博客项目大功告成。

参看 https://uwsgi-docs.readthedocs.io/en/latest/WSGIquickstart.html

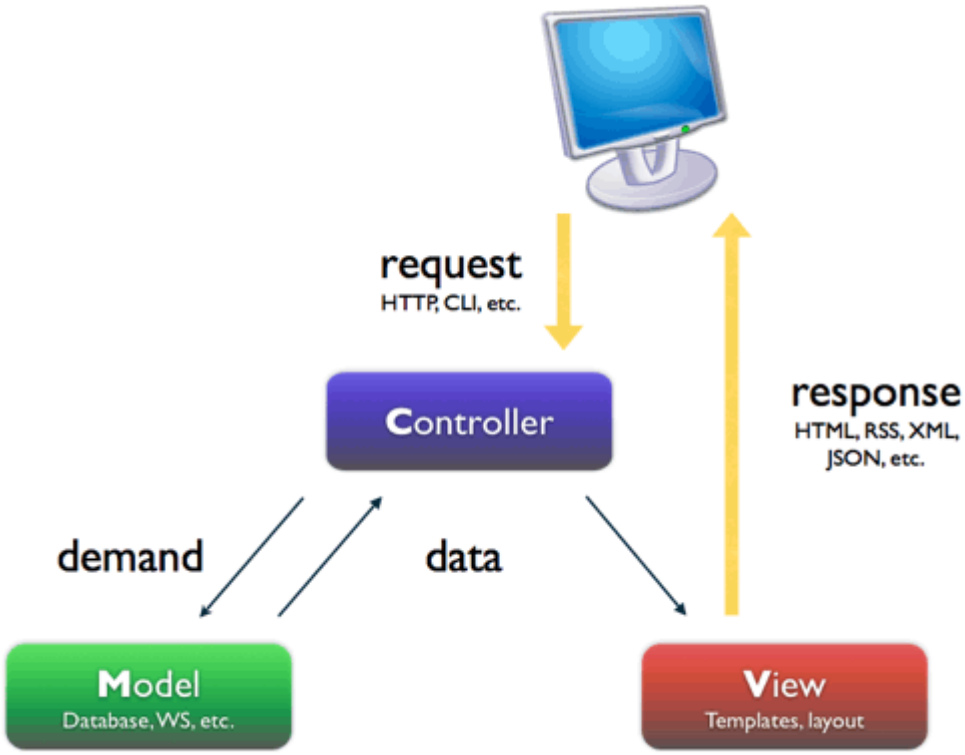
uwsgi协议 https://uwsgi-docs.readthedocs.io/en/latest/Protocol.html

部署图



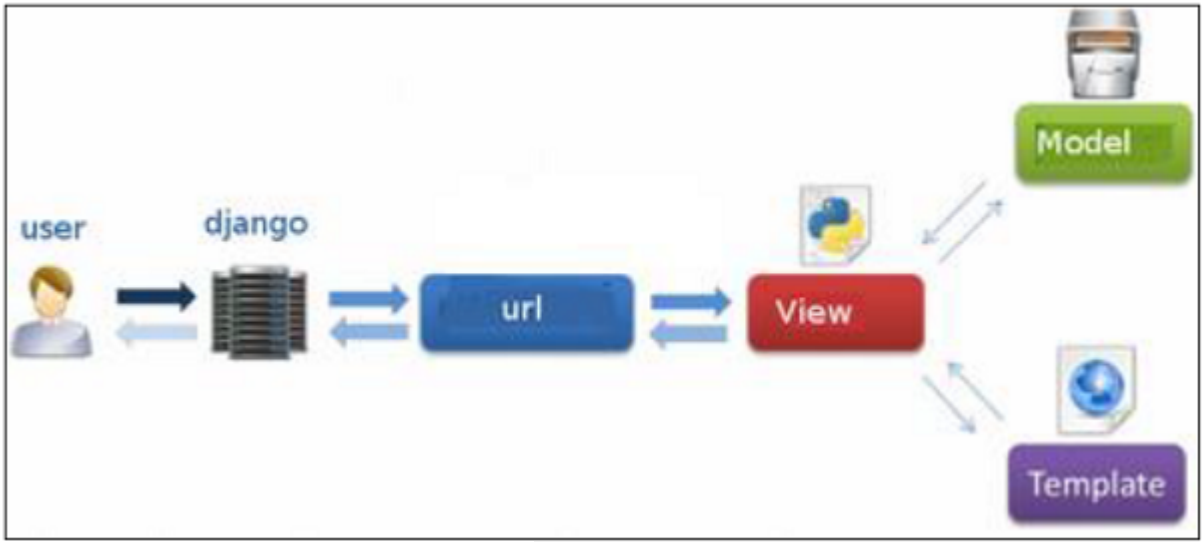
1. 浏览器通过互联网HTTP协议访问NGINX
2. 静态内容（图片、JS、CSS、文件）都由Nginx负责提供WEB服务
3. Nginx配置代理。可以是Http和Socket通信。本次使用uwsgi协议
4. uWSGI程序提供uwsgi协议的支持，将从Nginx发来的请求封装后调用WSGI的Application。这个Application可能很复杂，有可能是基于Django框架编写。这个程序将获得请求信息。
5. 通过Django的路由，将请求交给视图函数处理，可能需要访问数据库的数据。最终数据返回给浏览器。

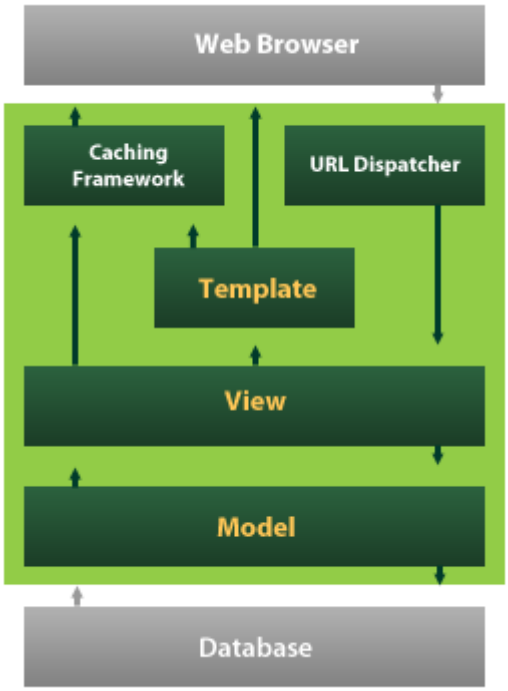
Django



MVC设计模式

Controller控制器：负责接收用户请求，调用Model完成数据，调用view完成对用户的响应
Model模型：负责业务数据的处理
View视图：负责用户的交互界面





Model层
ORM 建立对象关系映射，提供数据库操作

Template层
负责数据的可视化，使用HTML、CSS等构成模板，将数据应用到模板中，并返回给浏览器。

View层
Django完成URL映射后，把请求交给view层的视图函数处理，调用Model层完成数据，如有必要调用Template层响应客户端，如果不需要，直接返回数据。

Celery

Celery是一个使用Python开发的分布式任务调度模块，因此对于大量使用Python构建的系统，使用起来方便。

Celery目前版本4.x，仅支持Django 1.8以上版本。Celery 3.1只可以支持Django1.8以下版本。

优点

- 简单：调用接口简单易用
- 高可用：客户端、Worker之间连接自动重试，Broker自身支持HA
- 快速：单个Celery进程每分钟可以数以百万计的任务
- 灵活：Celery的每一个部分都能扩展，或直接使用，或用户自己定义。

常见应用

Celery可以支持实时实时异步任务处理，也支持任务的定时调度。

1. 异步发邮件
   celery执行队列
2. 间隔半小时同步天气信息等
   celery定时操作

角色

- 任务Task：对应一个Python函数
- 队列Queue：待执行任务的队列
- 工人Worker：一个新的进程，负责执行任务
- 代理人Broker：负责调度，在任务环境中使用RabbitMQ、Redis等

Celery需要依靠RabbitMQ等作为消息代理，同时也支持Redis甚至是Mysql、Mongo等，当然，官方默认推荐的是RabbitMQ，如果使用Redis需要配置。

本次采用Redis来作为Broker，也是Redis存储任务结果。

安装

```
$ pip install celery==4.2.0

安装对redis的支持，并自动升级相关依赖。安装Redis作为Broker，通过配置把结果也放到Redis中
$ pip install -U "celery[redis]"
```

测试

Celery库使用前，必须初始化，所得实例叫做"应用application或app"。应用是线程安全的。不同应用在同一进程中，可以使用不同配置、不同组件、不同结果

```
from celery import Celery

app = Celery('mytask')
```

```
print(app)

@app.task
def add(x, y):
    return x + y

print(add) # <@task: mytask.add of mytask at 0x1a30c4ad400>
print(app.tasks) # {'mytask.add': <@task: mytask.add of mytask at 0x1a30c4ad400>, .....省略}

print(add.name) # mytask.add
print(app.conf)
print(*list(app.conf.items()), sep='\n')
```

```
默认使用amqp连接到本地amqp://guest:**@localhost:5672//
```

本次使用Redis

Redis安装配置

使用Epel源的rpm安装

```
redis安装，使用提供的rpm安装，redis依赖jemalloc
# yum install jemalloc-3.6.0-1.el7.x86_64.rpm redis-3.2.12-2.el7.x86_64.rpm

# rpm -pql redis-3.2.12-2.el7.x86_64.rpm
/etc/logrotate.d/redis
/etc/redis-sentinel.conf
/etc/redis.conf
/usr/bin/redis-cli
/usr/bin/redis-sentinel
/usr/bin/redis-server
/usr/lib/systemd/system/redis-sentinel.service
/usr/lib/systemd/system/redis.service

编辑redis配置文件
# vi /etc/redis.conf
bind 192.168.142.131
protected-mode no
```

启动、停止redis服务

```
# systemctl start redis
# systemctl stop redis
```

broker配置使用

redis连接字符串格式 `redis://:password@hostname:port/db_number`

```
app.conf.broker_url = 'redis://192.168.142.131:6379/0'
# 意思是，指定服务器的redis端口6379，使用0号库
```

Celery使用

生成任务

```
# test1.py 注意模块名，后面命令中用
from celery import Celery
import time

app = Celery('mytask')
app.conf.broker_url = 'redis://192.168.142.131:6379/0' # 0号库存执行任务队列
# 重复执行问题解决
# 如果超过visibility_timeout，Celery会认为此任务失败
# 会重分配其他worker执行该任务，这样会造成重复执行。visibility_timeout这个值大一些
# 注意，如果慢任务执行时长超过visibility_timeout依然会多执行
app.conf.broker_transport_options = {'visibility_timeout': 43200} # 12 hours
app.conf.result_backend = 'redis://192.168.142.131:6379/1' # 1号库存执行结果

app.conf.update(
    enable_utc = True,
    timezone = 'Asia/Shanghai'
)

#@app.task
@app.task(name="firsttask")
#@app.task(ignore_result=True) # 不关心执行的结果
def add(x, y):
    print('in add. ~~~~~~')
    time.sleep(5)
    print('in add, timeout 5s. ~~~~~~')
    return x + y


if __name__ == '__main__':
    # 添加任务到Broker中
    print('in main. Send task')
    add.delay(4, 5)
    add.apply_async((10, 30), countdown=5)  # 5秒后执行
    print('end ~~~~~')
```

注意，上面代码执行，使用add.delay等加入任务到Redis中。在启动celery命令消费Redis的任务，执行并返回结果到Redis中。

```
# 增加任务的常用方法
T.delay(arg, kwarg=value)
always a shortcut to .apply_async.

T.apply_async((arg, ), {'kwarg': value})

T.apply_async(countdown=10)
executes 10 seconds from now.
```

执行任务

如果在Linux下可能出现下面的问题，可如下配置

```
from celery import platforms
# Linux下，默认不允许root用户启动celery，可使用下面的配置
platforms.C_FORCE_ROOT = True
```

使用命令执行Redis中的任务

```
-A APP, --app APP 指定app名称，APP是模块名
worker 指定worker工作
--loglevel 指定日志级别
-n 名称，%n指主机名
--concurrency 指定并发多进程数，缺省CPU数

$ celery -A test1 worker --loglevel=INFO --concurrency=5 -n worker1@%n
```

windows下可能下面问题

```
[ERROR/MainProcess] Task handler raised error: ValueError('not enough values to unpack (expected
3, got 0)',)
Traceback (most recent call last):
  File "e:\classprojects\venvs\p18test\lib\site-packages\billiard\pool.py", line 358, in
workloop
    result = (True, prepare_result(fun(*args, **kwargs)))
  File "e:\classprojects\venvs\p18test\lib\site-packages\celery\app\trace.py", line 537, in
_fast_trace_task
    tasks, accept, hostname = _loc
ValueError: not enough values to unpack (expected 3, got 0)
```

安装eventlet解决问题

```
$ pip install eventlet
```

重新执行任务

```
-P, --pool 指定进程池实现，默认prefork，windows下使用eventlet

$ celery -A test1 worker -P eventlet --loglevel=INFO --concurrency=5 -n worker1@%n

2任务，运行日志如下
[2018-03-27 10:08:23,598: INFO/MainProcess] Connected to redis://192.168.142.131:6379/0
[2018-03-27 10:08:23,607: INFO/MainProcess] mingle: searching for neighbors
[2018-03-27 10:08:24,661: INFO/MainProcess] mingle: all alone
[2018-03-27 10:08:24,676: INFO/MainProcess] worker@DESKTOP-D34H5HF ready.
[2018-03-27 10:08:24,689: INFO/MainProcess] pidbox: Connected to redis://192.168.142.131:6379/0.
[2018-03-27 10:08:25,102: INFO/MainProcess] Received task: firsttask[b642b80a-1180-4525-b3c7-
d4a89474d4de]
[2018-03-27 10:08:25,103: WARNING/MainProcess] in add. ~~~~~~
[2018-03-27 10:08:25,104: INFO/MainProcess] Received task: firsttask[9e624d2d-8116-46ce-af03-
4dd00ab93466]  ETA:[2018-03-27 10:07:50.606347+08:00]
[2018-03-27 10:08:25,105: WARNING/MainProcess] in add. ~~~~~~
[2018-03-27 10:08:30,101: WARNING/MainProcess] in add, timeout 5s. ~~~~~~
[2018-03-27 10:08:30,105: WARNING/MainProcess] in add, timeout 5s. ~~~~~~
[2018-03-27 10:08:30,106: INFO/MainProcess] Task firsttask[b642b80a-1180-4525-b3c7-d4a89474d4de]
succeeded in 5.014999999999418s: 9
[2018-03-27 10:08:30,109: INFO/MainProcess] Task firsttask[9e624d2d-8116-46ce-af03-4dd00ab93466]
succeeded in 5.0s: 40
在redis的1号库当中也能看到运行的结果
```

发邮件

在用户注册完激活时，或修改了用户信息，或遇到故障等情况时，都会发送邮件或发送短信息，这些业务场景不需要一直阻塞等待这些发送任务完成，一般都会采用异步执行。也就是说，都会向队列中添加一个任务后，直接返回。

Django中发送邮件需要在settings.py中配置，如下

```
# settings.py
# magedu.com设置
# SMTP
EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
EMAIL_HOST = "smtp.exmail.qq.com"
EMAIL_PORT = 465 #缺省25，SSL的一般465
EMAIL_USE_SSL = True #缺省False
EMAIL_HOST_USER = "magetest@magedu.com"
EMAIL_HOST_PASSWORD = "Python123"
EMAIL_USE_TLS = False #缺省False
```

注意，不同邮箱服务器配置不太一样

邮件发送测试代码如下

```
#https://docs.djangoproject.com/en/1.11/topics/email/
# 使用 SMTP
# 这个函数测试时，可以写在任意模块中，需要时被调用就可以了
from django.core.mail import send_mail
```

```
def email():

    send_mail(
        'first test email',
        'Right here waiting',
        settings.EMAIL_HOST_USER,
        ['wei.xu@magedu.com'],
        fail_silently=False,
        html_message="<h1>test title<a href='http://www.magedu.com'
target='_blank'>magedu.com</a></h1>"
    )
    print('++++++++++++++++++')


# 测试用的视图函数
def test1(request):
    try:
        email()
    except Exception as e:
        print(e)
        return HttpResponseBadRequest()

    return HttpResponse('test1 ok')
```

Celery集成

新版Celery集成到Django方式改变了。

目录结构

```
blogpro
    blog
        __init__.py
        settings.py
        celery.py
        urls.py
    app1
        __init__.py
        tasks.py
        view.py
        models.py
```

在Django全局目录中（settings.py所在目录）

1、定义一个celery.py

```
from __future__ import absolute_import, unicode_literals
import os
from celery import Celery
```

```
# set the default Django settings module for the 'celery' program.
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'blog.settings')

app = Celery('blog')

# Using a string here means the worker doesn't have to serialize
# the configuration object to child processes.
# - namespace='CELERY' means all celery-related configuration keys
#   should have a `CELERY_` prefix.
app.config_from_object('django.conf:settings', namespace='CELERY')

# Load task modules from all registered Django app configs.
app.autodiscover_tasks()

# 增加以下内容配置
app.conf.broker_url = 'redis://192.168.142.131:6379/0'
# 如果超过visibility_timeout，Celery会认为此任务失败
# 会重分配其他worker执行该任务，这样会造成重复执行。visibility_timeout这个值大一些
# 注意，如果慢任务执行时长超过visibility_timeout依然会多执行
app.conf.broker_transport_options = {'visibility_timeout': 43200} # 12 hours
app.conf.result_backend = 'redis://192.168.142.131:6379/1' #  执行结果存储

app.conf.update(
    enable_utc = True,
    timezone = 'Asia/Shanghai'
)
```

2、修改__init__.py

```
from __future__ import absolute_import, unicode_literals
# This will make sure the app is always imported when
# Django starts so that shared_task will use this app.
from .celery import app as celery_app

__all__ = ('celery_app',)
```

在user应用下

1、urls.py

```
from django.conf.urls import url
from .views import reg, login, test, logout, test1, testsendmail

urlpatterns = [
    url(r'^reg$', reg),
    url(r'^login$', login),
    url(r'^test$', test),
    url(r'^test1$', test1),
    url(r'^logout$', logout),
    url(r'^mail$', testsendmail) # 测试发邮件
]
```

2、views.py

```
from .tasks import sendmail

def testsendmail(request):
    try:
        # 用户注册了，注册信息保存了，然后发邮件给他，里面写
        sendmail.delay() # 阻塞效果 => 非阻塞的异步调用
    except Exception as e:
        print(e, '~~~~~~~~~~~~~~~~')
        return HttpResponseBadRequest()
    return HttpResponse('邮件已发送，请等待5分钟查收')
```

3、tasks.py

```
# Create your tasks here
from __future__ import absolute_import, unicode_literals
from blog.celery import app

from django.core.mail import send_mail
from django.conf import settings
import datetime

# celery -A blog worker -P eventlet -l INFO -c 5 -n worker@%n
@app.task(name='sendemail')
def sendmail():
    send_mail(
        '发邮件测试',
        '测试用',
        settings.EMAIL_HOST_USER, #'from@example.com', # 谁发的
        ['wei.xu@magedu.com'], # 发给谁们
        fail_silently=False,
        html_message="<p>这是一封测试邮件 {:%Y%m%d-%H:%M:%S}<br /><a href='{}' target='_blank'>官
网</a></p>".format(
            datetime.datetime.now(), 'http://www.magedu.com')
    )
    print('++++++++++++++++++')
```

访问http://127.0.0.1:8000/user/mail 会调用test1视图函数，会执行email.delay()，会在redis中增加任务。

执行任务

```
$ celery -A blog -P eventlet worker --loglevel=INFO --concurrency=5 -n worker@%n
```