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

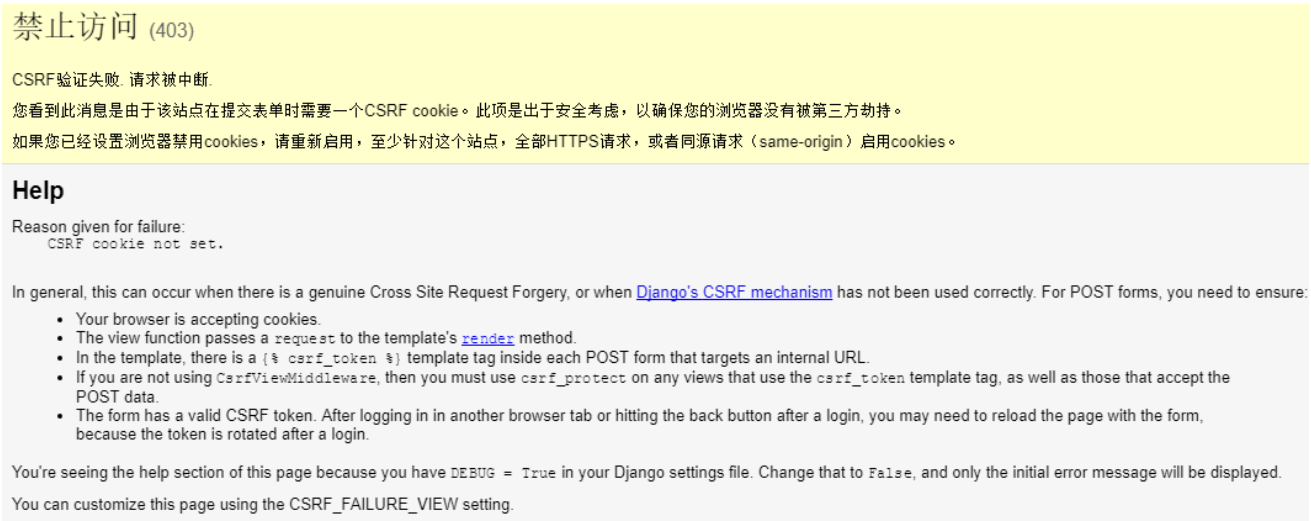
Django中有很多异常类，定义在django.http下，这些类都继承自HttpResponse。

```
# 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

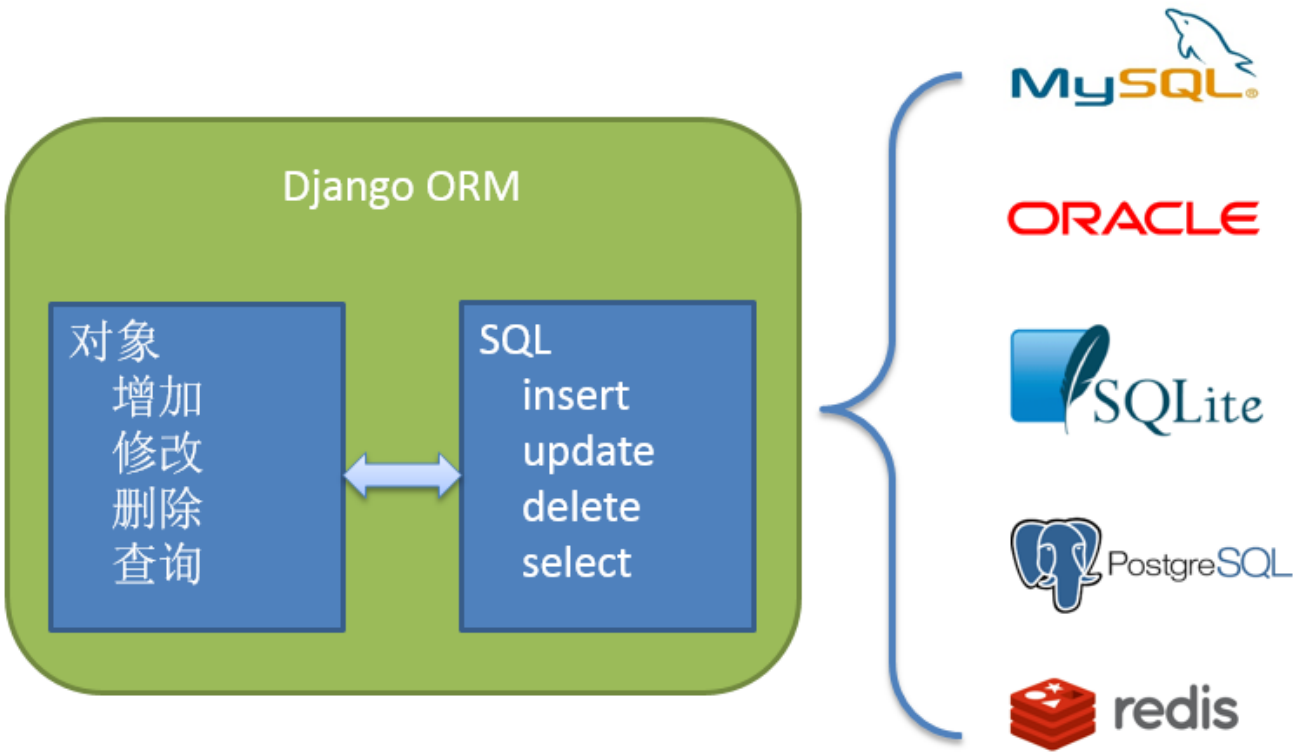
管理器对象

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
```