基于Flask开发企业级API应用系列文章

> 关于我 编程界的一名小小程序猿，目前在一个创业团队任team lead，技术栈涉及Android、Python、Java和Go，这个也是我们团队的主要技术栈。 Github：https://github.com/hylinux1024 微信公众号：angrycode

前面对`Python WEB`框架`Flask`的源码进行走读，对服务的启动流程、路由原理和模板渲染有了一个宏观的认识。不过说了那么多理论，接下来就利用`Flask`开发一个企业级的`API`应用。

第一篇

我选用团队最近开发的一个企业应用作为案例。这是一个恋爱交友应用，本来是使用`Java`的`SpringBoot`框架进行开发的，不过为了避免不必要的麻烦，我会使用`Flask`进行改造，当然这个案例我还会精简一下，保持核心业务的同时，重点关注其中涉及到的**技术和工具库**的使用，最大限度的还原项目开发的完整流程。

0x00 技术栈

这里我们使用`Python`版本为3.7，`WEB`框架当然就是`Flask`，数据库使用`MySql`，`ORM`使用`SqlAlchemy`，使用`Redis`作为缓存，可能还会使用到序列化工具库`marshmallow`。

开发环境使用`venv`，部署服务环境会使用`nginx+gunicorn+supervisord`

因此整个技术栈为

```
# 开发技术栈
Python3.7+venv+Flask+MySql+SqlAlchemy+Redis+marshmallow
# 部署技术栈
Python3.7+venv+nginx+gunicorn+supervisord
```

当然企业实际开发中还需要编写接口文档，用于各端同学的交互。我们可以使用`postman`或者淘宝的API文档服务。

0x01 项目设计

技术选型做好之后，先不急于写代码，而是先把项目前期的设计做好，根据业务需求理清功能模块、数据库表结构、接口文档等。

我们的需求是做一个**恋爱交友**的应用，那么它**主要功能模块**就应该有

- **登录注册** 这里使用用户手机号进行登录注册
- **用户列表** 用户登录后，可以查看当前热门推荐的用户
- **联系人列表** 联系过的用户，会出现在联系人列表中
- **聊天模块** 给用户发送消息，消息类型包括文本、语音等
- **附近的人** 根据用户登录的地理位置，查看附近的人
- **谁看过我** 查看谁看过我，这个可以作为`VIP`功能
- **个人信息** 包括用户基本信息、用户相册和用户标签等
- **VIP模块** 当用户充值为`VIP`后可以解锁一些功能，比如查看**谁看过我的列表**等

注意为了避免项目开发周期过长我们主要关注前台api的开发，对于后台管理功能暂时不考虑。

根据这些功能模块，我们对项目中的**实体进行抽象**主要有

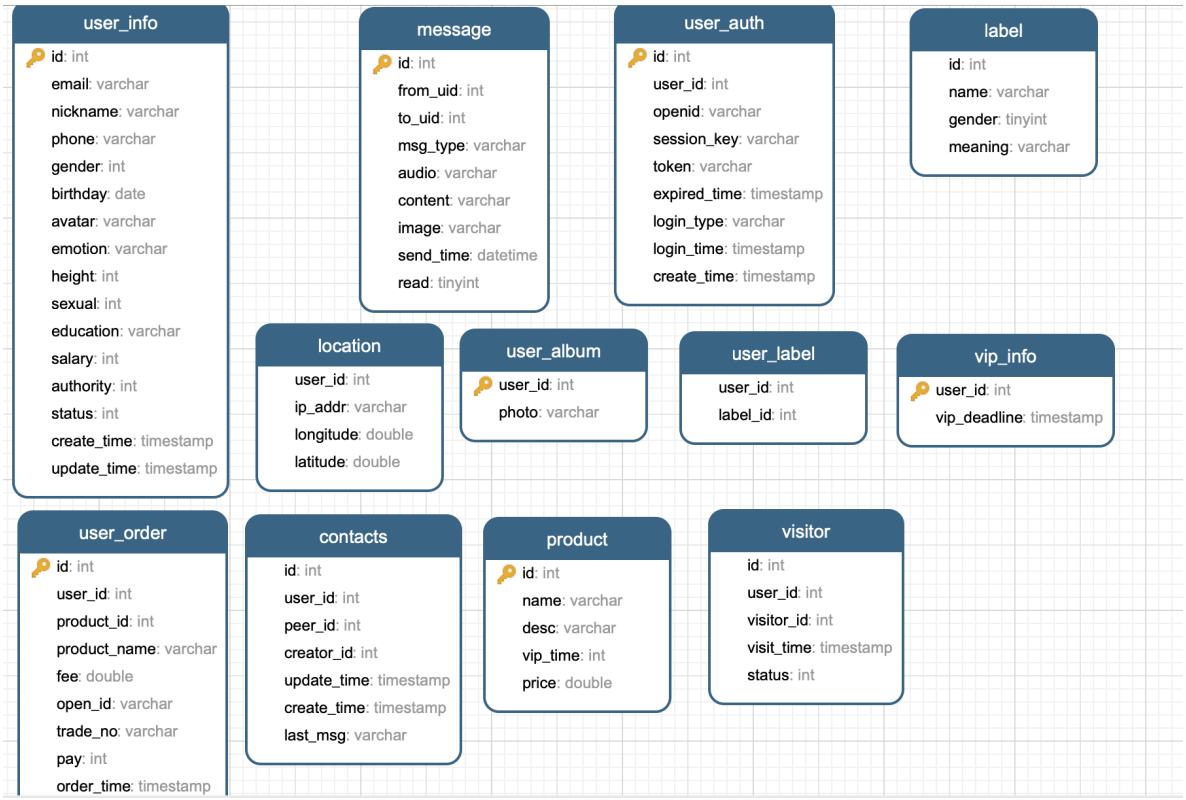
- **登录授权user_auth**
- **用户基本信息user_info**
- **用户位置location**
- **用户相册user_album**
- **用户标签user_label**
- **标签label**
- **联系人contacts**
- **消息message**
- **访问足迹visitor**
- **充值VIP的商品product** 有月度`VIP`、季度`VIP`和年度`VIP`三种
- **订单user_order**
- **用户VIP信息vip_info**

这些实体在**数据库建模**中分别对应各自的表。避免代码篇幅太长，这里就不再贴出各表脚本代码。关于`sql`表结构会在后面的项目地址中给出。

0x02 数据库

我这里使用的是腾讯云的数据库，当然使用本地的数据库也是可以的。

各表的字段如下图



注意这些表我都没有加外键约束。

0x03 项目框架搭建

我使用`PyCharm`作为开发环境的`IDE`，创建了一个名为`DatingToday`项目，结构如下

```
(venv) ➜  DatingToday tree -L 1
.
├── app.py
├── datingtoday.sql
├── requirements.txt
├── static
├── templates
└── venv
```

注意到我已经把数据库脚本文件放在项目根目录了。`venv`环境安装了以下依赖库

```
(venv) ➜  DatingToday pip list
Package                Version
---------------------- -------
Click                  7.0
Flask                  1.1.1
flask-marshmallow      0.10.1
Flask-SQLAlchemy       2.4.0
itsdangerous           1.1.0
Jinja2                 2.10.1
MarkupSafe             1.1.1
marshmallow            2.19.5
marshmallow-sqlalchemy 0.17.0
pip                    10.0.1
setuptools             39.1.0
six                    1.12.0
SQLAlchemy             1.3.6
Werkzeug               0.15.5
```

可以使用命令

```
(venv) ➜ pip freeze > requirements.txt
```

生成`requirements.tx`文件。

使用命令

```
(venv) ➜ pip install -r requirements.txt
```

还原虚拟环境中的依赖。

0x04 总结

本篇是基于Flask开发企业级API应用的第一篇，主要是对项目开发前期的准备工作，包括项目设计、数据库设计以及项目结构搭建，当然实际工作中可能还会先出`API`文档，让前端的同学可以先动起来，但我这里因为已经是在*写文档*了，所以`API`文档就省略了。磨刀不误砍柴工，这些工作都是必需的。

0x05 项目地址

https://github.com/hylinux1024/datingtoday

0x06 学习资料

- https://palletsprojects.com/p/flask/

- https://realpython.com/flask-connexion-rest-api-part-2/

第二篇

本节开始项目的编码实现。首先我们来实现**登录注册**模块的相关`API`。本项目我们是使用**前后端分离**的模式，在实现登录注册功能之前，假设我们的接口是开放的，那么需要确定**接口校验方案**。

0x00 接口校验方案

我们的目标是接口**不能被抓包重复访问**，并且要对**客户端的可靠性**进行验证。

- **防重放攻击**可使用参数有`timestamp`、`nonce`、`token`和`sign`
- **支持可信任的客户端**的请求则可以考虑添加`appkey`和`appsecret`参数

公共参数

1、timestamp 时间戳

单位毫秒，也可以是秒，与服务端保持一致； 时间戳是无关时区的，所以客户端与服务端的时间戳是可以用来比较的； 如果客户端与服务端时间戳相差比较大，则可以考虑使用服务端时间进行校准； 时间戳的作用是，**保证这个请求在一定时间内（例如60秒内）是有效的。有效期内的校验就需要nonce参数**

2、nonce 随机数

由客户端产生的随机数，客户端每次接口请求时需要保证它是不一样的。 `nonce`的作用是**保证timestamp有效期内的请求是否是合法的。** 服务端接收到这个参数后，会将其保存在某个集合中。 服务端会检测这个`nonce`是否在该集合中出现过，如果出现过说明该请求是不合法的。 每个`nonce`的有效期设置跟`timestamp`参数有关，例如可以设置为60秒。

3、token 登录态

需要登录的接口则需要一个`token`参数。 服务端生成的`token`在有效期内有效，如果`token`过期则需要提示客户端重新登录。 `token`的生成规则可使用随机数

```
token = md5(1024位的随机数)
```

4、sign 签名或校验参数

```
msg = 除了timestamp、nonce、token、sign参数之外的其它排序后的参数列表和值列表 = sort(参数1=值1&参数2=值2&参数3=值3...)

sign = md5(msg+token+timestamp+nonce+salt)

salt = 客户端与服务端约定字符串
```

5、appkey 和 appsecret

服务端为可信任的客户端分配`appkey`和`appsecret`参数。可由随机数或自定义的规则生成，要保证`appkey`和`appsecret`是对应的。 客户端需保证`appsecret`不被泄露。 客户端接口请求时只需带上`appkey`参数。 `appsecret`则添加到`sign`校验参数的计算中

```
sign = md5(token+msg+timestamp+nonce+appsecret)
```

结合上面的参数，一个接口请求应该类似这样

```
http://api.example.com/v1/login?
phone=13499990000&timestamp=1564486841415&nonce=34C2AF&sign=e10adc3949ba59abbe56
e057f20f883e&appkey=A23CE80D
```

服务端程序接收到请求后**验证流程**应该是这样的

1. 通过`appkey`查询到`appsecret`，如果查不到则返回出错信息，否则继续；
2. 通过`timestamp`检查`nonce`是否在有效时间内是的重复请求，如果是多次重复请求，则返回出错信息，否则继续；
3. 通过请求参数构造`msg`并计算`sign`，将此参数与请求中获取到的参数进行对比，验证成功后才开始我们的业务逻辑。

这样我们的一个简单实用的接口验证方案就出来了，当然可能还有其它一些好的想法，欢迎留言一起探讨学习。

0x01 show me the code

现在开始实现登录注册功能，相信这个模块走通了，之后其它模块也是依样画葫芦。

先看下模块

```
├── api
|   ├── __init__.py
|   └── auth.py
├── app.py
├── config.ini
├── datingtoday.sql
├── models.py
├── requirements.txt
├── test
└── venv
```

增加了一个`api`相关的文件包。 还有一个`config.ini`，主要用于配置数据库等信息，而`models.py`文件是定义实体类的地方。

`api/__init__.py`

```python
from flask import jsonify

def make_response_ok(data=None):
    resp = {'code': 0, 'msg': 'success'}
    if data:
        resp['data'] = data
    return jsonify(resp)

def make_response_error(code, msg):
    resp = {'code': code, 'msg': msg}
    return jsonify(resp)

def validsign(func):
    """
    验证签名
    :param func:
    :return:
    """
```

```
    def decorator():
        params = request.form
        appkey = params.get('appkey')
        sign = params.get('sign')
        csign = signature(params)
        if not appkey:
            return make_response_error(300, 'appkey is none.')
        if csign != sign:
            return make_response_error(500, 'signature is error.')
        return func()

    return decorator
```

在`__init__.py`中首先定义了两个封装统一的`json`数据结构的的方法，主要是用到`flask`中的`jsonify`函数，它可以把一个对象转成`json`。

在前面我们讲了接口的验证逻辑，这一部分对参数的校验功能其实是可以通用的，所以对这个逻辑也进行了封装成`validsign`方法。

不错，这是一个装饰器的定义。我们希望在接口访问的方法使用装饰器，就可以进行通用的接口校验。

`auth.py`

这一节的重点是实现登录注册和发短信接口，因此创建一个`auth.py`的文件来写跟授权登录相关的接口，这样有利于我们组织代码。 我们知道要实现接口的访问路径的定义与方法直接的对应，是使用`@route`这个装饰器。这里我们在一个新文件中定义我们的接口，就需要用到`Blueprint`

> A blueprint is an object that allows defining application functions without requiring an application object ahead of time. It uses the same decorators as Flask, but defers the need for an application by recording them for later registration.

说白了，它的作用跟`@route`差不多。

由于我们把登录注册当作一个接口来实现，即用户通过短信进行登录，后端会判断该用户是否为新用户，如果是新用户则自动注册。

0x02 短信接口

首先定义接口的访问路径为

```
{host:port}/api/auth/sendsms

请求方法：POST
参数：phone
请求成功
{
    "code": 0,
    "data": {
        "code": "97532",
        "phone": "18922986865"
    },
    "msg": "success"
}
```

根据接口定义我们会在`auth.py`中定义一个`Blueprint`对象用来映射我们的访问路径和方法。

```
bp = Blueprint("auth", __name__, url_prefix='/api/auth')
```

短信接口的实现这里会使用到`redis`，将请求到的短信验证码保存在`redis`中，并设置过期时间。然后登录时，再进行验证。

```
@bp.route("/sendsms", methods=['POST'], endpoint="sendsms")
@validsign
def send_sms():
    phone = request.form.get('phone')
    m = re.match(pattern_phone, phone)
    if not m:
        return make_response_error(300, 'phone number format error.')
    # 这里需要修改为对接短信服务
    code = '97532'
    key = f'{phone}-{code}'
    r.set(key, code, 60)
    return make_response_ok({'phone': phone, 'code': code})
```

注意这里的`endpoint="sendsms"`是必需设置，因为`@validsign`会修饰我们的方法，每个方法都是用一个通用的校验，方法名称会变成一样的，所以如果不设置`endpoint`会导致`url`映射失败。

0x03 登录注册接口

首先定义接口的访问路径为

```
{host:port}/api/auth/login

请求方法：POST
参数：phone
参数：code
请求成功
{
    "code": 0,
    "data": {
        "expire_time": "2019-08-10 07:34:20",
        "token": "5bea89727e7553284f162d35c9926414",
        "user_id": 100784
    },
    "msg": "success"
}
```

执行登录接口时，会先验证`redis`中的验证码，然后查一下授权表`user_auth`看看是否是新用户，最后返回用户的登录授权信息。

```
@bp.route("/login", methods=['POST'], endpoint='login')
@validsign
def login():
    phone = request.form.get('phone')
    code = request.form.get('code')
    key = f'{phone}-{code}'
    sms_code = r.get(key)
    if sms_code:
        sms_code = sms_code.decode()
    if code != sms_code:
        return make_response_error(503, 'sms code error')
```

```
    auth_info = UserAuth.query.filter_by(open_id=phone).first()
    if not auth_info:
        auth_info = register_by_phone(phone)
    else:
        auth_info = login_by_phone(auth_info)

    data = {'token': auth_info.token,
            'expired_time': auth_info.expired_time.strftime("%Y-%m-%d
%H:%M:%S"),
            'user_id': auth_info.user_basic.id}

    r.set(f'auth_info_{auth_info.user_id}', str(data))
    return make_response_ok(data)
```

总体上逻辑还是比较清晰的，最后我们看一下`app.py`

```
from flask import Flask

from api import auth, config
from models import db

app = Flask(__name__)
# 将blueprint注册到app中
app.register_blueprint(auth.bp)
# 配置app的config，将数据库信息配置好
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
app.config["SQLALCHEMY_DATABASE_URI"] = config['DATABASE']['uri']
# 最好生成一个secret_key
app.secret_key = '8c2c0b555e6e6cb01a5fd36dd981bcee'

db.init_app(app)

@app.route('/')
def hello_world():
    return 'Hello World!'


if __name__ == '__main__':
    app.run()
```

配置文件`config.ini`

```
# 配置数据库链接
[DATABASE]
uri = mysql+pymysql://user:password@127.0.0.1:3306/datingtoday

# 配置appkey和secret
[APP]
appkey = 432ABZ
appsecret = 1AE32B09224
```

0x04 单元测试

由于接口都需要动态计算校验码，所以单元测试是必需的。这里我使用最简单的方式，直接使用`unittest`模块。

例如测试发短信的业务接口，首先生成一个随机数`nonce`，然后计算校验码`sign`参数，最后调用`flask`中的`post`方法模拟接口请求。

```
def test_sendsms(self):
    import math
    nonce = math.floor(random.uniform(100000, 1000000))
    params = {'phone': '18922986865', 'appkey': '432ABZ', 'timestamp':
datetime.now().timestamp(),
              'nonce': nonce}
    sign = signature(params)
    params['sign'] = sign

    respdata = self.app.post("/api/auth/sendsms", data=params)
    resp = respdata.json
    self.assertEqual(resp['code'], 0, respdata.data)
```

如果请求成功，就认为通过测试。当然这里的逻辑还是比较简单，希望小伙伴们留言讨论。

0x05 项目地址

源码地址： github.com/hylinux1024...

Flask官方地址： palletsprojects.com/p/flask/

第三篇

前两篇把程序的结构以及`API`的协议基本上搭建起来了。本文开始不打算对每个模块接口都进行实现，因为基本上都是业务逻辑代码，而且整篇文章都把代码贴出来，那将是一个灾难。

《上一篇》对**登录授权模块**的接口进行了实现，在写本篇文字的时候，我也把**用户模块**的用户列表、用户信息查询、更新用户信息等接口进行了实现。写到这里的时候我发现，有很多重复的逻辑。比如说，登录参数校验、错误信息处理等这些逻辑，其实这些逻辑可以进行统一处理。

0x00 统一错误处理

客户端如果访问了以下这个**没有定义**的接口

```
http://127.0.0.1:5000/api/auth/something
```

将返回以下信息

```
Not Found

The requested URL was not found on the server. If you entered the URL manually
please check your spelling and try again.
```

或者有一些**数据库操作出错**，也会导致服务器的内部错误

```
Internal Server Error

The server encountered an internal error and was unable to complete your
request. Either the server is overloaded or there is an error in the
application.
```

这些信息对使用这个系统`API`的客户端来说不是很友好，我们希望通过结构化的`json`数据进行返回。

要对这种`http`协议的错误信息请求统一处理或者实现自定义的错误页面，就需要用到`@errorhandler`这个装饰器。

在`app.py`中，增加以下两个方法

```
@app.errorhandler(404)
def not_found_error(error):
    return make_response_error(404, error.description)


@app.errorhandler(500)
def internal_error(error):
    db.session.rollback()
    return make_response_error(500, error.description)
```

当请求一个不存在的`url`时，我们的系统应该返回类似以下的信息

```
{
"code": 404,
"msg": "The requested URL was not found on the server. If you entered the URL
manually please check your spelling and try again."
}
```

这样就跟我们的定义的数据结构接口协议保持一致。

0x01 统一验证用户token

由于系统中有很多接口是需要用户登录`token`才能访问的，所以每个接口都进行登录`token`的验证。打开`users.py`模块，以下接口都有`token`验证的逻辑

```
@bp.route('/show', endpoint='show')
def show_user_info():
    uid = request.args.get('userId')
    peer_id = request.args.get('peerId')
    token = request.args.get('token', '')
    if not UserInfo.check_token(uid, token):
        return make_response_error(504, 'no operation permission')

    ...
    # 省略不必要的代码
    return make_response_ok(data)
@bp.route('/hot/list', endpoint='list')
def list_hot_user():
    uid = request.args.get('userId')
    token = request.args.get('token', '')
    if not UserInfo.check_token(uid, token):
        return make_response_error(504, 'no operation permission')
    ...
    # 省略不必要的代码
    return make_response_ok(obj)

@bp.route('/update', methods=["POST"], endpoint="update")
def update_user():
    uid = request.form.get('userId', '')
```

```
    token = request.form.get('token', '')

    if not UserInfo.check_token(uid, token):
        return make_response_error(504, 'no operation permission')

    ...
    # 省略不必要的代码
    return make_response_ok(data={"data": user.id})
```

上面三个接口都有相同的验证`token`的逻辑

```
if not UserInfo.check_token(uid, token):
    return make_response_error(504, 'no operation permission')
```

而这个系统的接口远不止这些，如果每个接口都写相同的逻辑代码，看起来也不怎么优雅。

是不是可以跟前面定义的`@validsign`装饰器一样，定义一个`@require_token`的装饰器呢？ 答案是肯定的。

但这里我想直接修改`@validsign`这个装饰器函数，给它添加一个参数`@validsign(require_token=True)`这种方式，使用起来应该会更加简洁。

```
def validsign(require_token=False, require_sign=True):
    """
    验证签名,token信息
    :param require_token: 是否验证token
    :param require_sign: 是否验证签名
    :return:
    """
    def decorator(func):
        def wrapper():
            params = _get_request_params()
            if require_sign:
                appkey = params.get('appkey')
                sign = params.get('sign')
                csign = signature(params)
                if not appkey:
                    return make_response_error(300, 'appkey is none.')
                if csign != sign:
                    return make_response_error(500, 'signature is error.')
            if require_token:
                token = params.get('token')
                uid = params.get('userId')
                if not UserInfo.check_token(uid, token):
                    return make_response_error(504, 'no operation permission')
            return func()
        return wrapper
    return decorator
```

通过参数`require_token`和`require_sign`可以比较灵活的控制接口的验证逻辑，对开发过程中调试也是很有帮助的。

这里把`token`对验证逻辑封装在`UserInfo`里面了，这是一个静态方法

```
@staticmethod
def check_token(uid, token):
    if not token or not uid:
        return False
    user = UserInfo.query.filter_by(id=uid).first()
    if not user:
        return False
    if not user.user_auth:
        return False
    return user.user_auth.token == token
```

0x02 单元测试

由于对之前的`@validsign`装饰器函数进行修改了，单元测试可以验证我们的修改不会影响到具体的业务逻辑，可以保证在原来的基础上进行修改，这是一种保守主义的做事方法。 同样地新添加的模块`users.py`也需要相应的单元测试功能。

```
def test_hotlist(self):
    import math
    nonce = math.floor(random.uniform(100000, 1000000))
    params = {'phone': '18922986865', 'userId': '100784', 'appkey': '432ABZ',
              'token': '575f680ddbd0d494a1b5fad8497293d2',
              'timestamp': datetime.now().timestamp(),
              'nonce': nonce}
    sign = signature(params)
    params['sign'] = sign

    respdata = self.app.get("/api/user/hot/list", data=params)

    self.assertEqual(200, respdata.status_code)

    resp = respdata.json
    self.assertEqual(0, resp['code'], respdata.data)
    self.assertIsNotNone(resp['data'], respdata.data)
```

这个是对首页列表的加载的测试，比较简单。

0x03 小结一下

在项目开发过程中，对于重复的逻辑应该要**抽象封装**

```
Don't repeat yourself
```

而如何封装就要看个人功力了，我觉得除了多学习，多看源码，几乎没有其它捷径。

0x04 学习资料

- palletsprojects.com/p/flask/ `flask`官方文档
- docs.sqlalchemy.org/en/13/orm/b... `models`关系映射相关文档
- github.com/hylinux1024... 本文项目源码

第四篇

0x00 配置缓存服务

几乎现在所有应用都会用到缓存技术，而在服务器端`redis`是很多实现缓存的首选技术。

对于我们这个应用也是需要使用缓存技术提高接口访问速度。

首先安装`redis`，并启动`redis`服务。

下载并编译安装

```
wget http://download.redis.io/releases/redis-5.0.5.tar.gz
tar xzf redis-5.0.5.tar.gz
cd redis-5.0.5
make
```

启动服务

```
src/redis-server
```

使用`redis`命令行客户端连接测试

```
src/redis-cli
redis> set foo bar
OK
redis> get foo
"bar"
```

建议详细的配置可以参阅相关官方文档。本文的重点还是关注在项目中的使用。

安装依赖

安装

```
pip install redis
```

(当然，你也可以使用`Flask-Cache`这个插件，使用起来也挺方便，不过本文延续之前的一些历史代码，就是直接使用`redis`这个库)

然后对`redis`做了一个简单的封装，分别是创建实例，设置缓存和获取缓存共三个静态方法。

```
import redis
import json
import datetime

# 创建连接池
pool = redis.ConnectionPool(host='127.0.0.1', port=6379)

class Redis:
    @staticmethod
    def connect(db=0):
        """默认使用0号库"""
        r = redis.Redis(connection_pool=pool, db=db)
        return r

    # 将内存数据二进制通过序列号转为文本流，再存入redis
    @staticmethod
    def set(r, key: str, data, ex=None):
        r.set(key, json.dumps(data, cls=CJsonEncoder), ex=ex)
```

```
    # 将文本流从redis中读取并反序列化，返回
    @staticmethod
    def get(r, key: str):
        data = r.get(key)
        if data is None:
            return None

        return json.loads(data)
```

在需要使用缓存的模块中通过以下方式进行获取`redis`实例

```
r_cache = redis_helper.Redis.connect(db=5)
```

由于我本机中还有其它的服务在开发，所以我选择5号库作为缓存，避免与其它服务发生冲突。

0x01 SQLAlchemy中实体关系的表示

前面几讲对模型中的关系在`SQLAlchemy`中的表示没有详细的说明，今天来拆解一下。

我们以用户表(`UserInfo`)与授权表(`UserAuth`)进行说明。

```
class UserInfo(db.Model):
    """用户基本信息"""
    __tablename__ = 'user_info'

    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    email = db.Column(db.String(64))  # email
    nickname = db.Column(db.String(64))
    phone = db.Column(db.String(16))
    gender = db.Column(db.Integer)  # 1男2女0未知
    ...

class UserAuth(db.Model):
    """授权登录表"""
    __tablename__ = 'user_auth'

    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    user_id = db.Column(db.Integer, db.ForeignKey('user_info.id'))
    user_basic = db.relationship(UserInfo, backref=db.backref('user_auth',
uselist=False))
    ...
```

`UserInfo`与`UserAuth`是一对一的关系。即一个用户对应一个授权信息，在`SQLAlchemy`中表示关系是使用`db.relationship()`接口

在`UserAuth`中定义外键`user_id`时需要指定`db.ForeignKey('user_info.id'))`参数，说明是关联的是`user_info`表中的`id`字段。通过外键关联了这两张表，但**在实际的开发使用中，我们希望在查询到UserAuth实例的时候，希望能够直接就能够得到对应的用户信息UserInfo的实例**，这时候就可以用`db.relationship()`接口。在`UserAuth`就指定了这个关系

```
user_basic = db.relationship(UserInfo, backref=db.backref('user_auth',
uselist=False))
```

说明在`UserAuth`的实体中定义了一个`user_basic`的字段，当查询到`UserAuth`实例时，可以直接得到`UserInfo`的信息，而不需要程序猿再去通过外键`user_id`去数据库中查询用户信息。

`db.relationship()`中的第一个参数表示要关联的哪张表，可以传类名或表名称的字符串；第二个参数`backref`的意思是在`UserInfo`中也定义一个`user_auth`的属性，方便查询到用户信息时，可以通过这个属性得到对应的授权信息。`uselist=False`的意思是`user_auth`与`user_info`是一对一的关系，如果是一对多的关系，这里`uselist=True`，不传参数时默认`uselist`是`True`。

0x03 总结

本文对前面几讲中的涉及到的模型定义中关系的表示和缓存的使用做一个补充说明。

0x04 引用

- flask-sqlalchemy.palletsprojects.com/en/2.x/mode...
- redis.io/download