**8.3权限拦截·Flask实现Token认证**

上文只是简单的实现了权限拦截的逻辑，但是就整个权限认证体系来说，还是不够的，我们缺少了权限认证的入口。权限认证，存储会话信息有多种方式，比如Flask自带的session。但是随着系统的发展，session是不太能满足需求了，本文采用的是Token认证的方式。整体流程为：

* 登录成功->返回认证token
* 访问需要认证的接口时携带认证的token
* 权限装饰器判断是否携带认证token来进行权限拦截

# 关于Token存储策略

当我们生成token凭证后，后端也是需要存储token信息的，这样客户端请求时携带token凭证,后端才能验证该凭证是否存在且是否在有效期内。在这里我们可以提供三种常用的存储方案。我们可以根据业务的需要，进行配置使用。

## CacheTokenStrategy

缓存存储策略,一般开发阶段常用的方案，服务重启后，token数据会丢失。

## MysqlTokenStrategy

Mysql存储策略，不想多安装一个redis中间件的的另一种token持久化方案

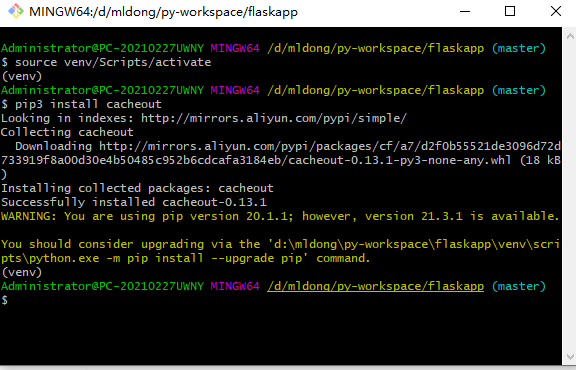
## RedisTokenStrategy

redis存储策略，建议使用的方案

# 安装依赖

## 安装缓存依赖库

# 安装最新版本  
pip3 install cacheout  
# 或指定安装版本  
pip3 install cacheout==0.13.1



## 安装flask-redis依赖库

# 安装最新版本  
pip3 install flask-redis  
# 或指定安装版本  
pip3 install flask-redis==0.4.0

# 新增目录mauth/mtoken

mkdir -p mauth/mtoken

# 新增mauth/mtoken/\_\_init\_\_.py文件

三个token存储实现类

import datetime  
import json  
import uuid  
  
from flask import request, current\_app, g  
from flask\_redis import FlaskRedis  
  
from models import BaseModel, db  
  
redis\_client = FlaskRedis()  
  
  
class AbstractTokenStrategy:  
 """  
 token存储策略抽象类  
 """  
 # token 过期时间- 单位是s # 2小时  
 TOKEN\_EXPIRE\_TIMEOUT = 60 \* 60 \* 2  
 # token-key  
 TOKEN\_KEY = "token"  
 # token 前辍  
 TOKEN\_KEY\_PREFIX = "flaskapp:"  
 # 超级管理员id  
 SUPER\_ADMIN\_ID = 1  
  
 def \_\_init\_\_(self, app=None):  
 self.app = app  
 if app:  
 self.init\_app(app)  
 else:  
 self.init\_app(current\_app)  
  
 def init\_app(self, app):  
 if app.config.get('TOKEN\_EXPIRE\_TIMEOUT'):  
 self.TOKEN\_EXPIRE\_TIMEOUT = app.config.get('TOKEN\_EXPIRE\_TIMEOUT')  
 if app.config.get('TOKEN\_KEY'):  
 self.TOKEN\_KEY = app.config.get('TOKEN\_KEY')  
 if app.config.get('TOKEN\_KEY\_PREFIX'):  
 self.TOKEN\_KEY\_PREFIX = app.config.get('TOKEN\_KEY\_PREFIX')  
 if app.config.get('SUPER\_ADMIN\_ID'):  
 self.SUPER\_ADMIN\_ID = app.config.get('SUPER\_ADMIN\_ID')  
  
 def \_\_new\_\_(cls, \*args, \*\*kwargs):  
 """  
 设置成单例模式  
 :param args:  
 :param kwargs:  
 """  
 if not hasattr(cls, '\_instance'):  
 orig = super(AbstractTokenStrategy, cls)  
 cls.\_instance = orig.\_\_new\_\_(cls, \*args, \*\*kwargs)  
 return cls.\_instance  
  
 def set(self, data):  
 """  
 存储token  
 :param data:  
 :return:  
 """  
 pass  
  
 def get(self, token):  
 """  
 获取token  
 :param token:  
 :return:  
 """  
 pass  
  
 def remove(self, token):  
 """  
 删除token  
 :param token:  
 :return:  
 """  
 pass  
  
  
class TokenStrategyFactory:  
  
 @staticmethod  
 def get\_instance():  
 token\_strategy = current\_app.config.get("TOKEN\_STRATEGY", "cache")  
 if token\_strategy == 'redis':  
 return RedisTokenStrategy()  
 elif token\_strategy == 'mysql':  
 return MysqlTokenStrategy()  
 else:  
 return CacheTokenStrategy()  
  
 @staticmethod  
 def create\_token(data=None):  
 uid = str(uuid.uuid4())  
 suid = ''.join(uid.split('-'))  
 return suid  
  
 """  
 校验token是否存在  
 """  
  
 @staticmethod  
 def check\_token(access=None):  
 """  
 核实token-如果权限标识存在，则校验是否存在权限标识  
 :param access: 权限标识  
 :return:  
 """  
 token\_strategy = TokenStrategyFactory.get\_instance()  
 token = request.headers.get(token\_strategy.TOKEN\_KEY, "")  
 data = token\_strategy.get(token)  
 if data is not None:  
 # 将当前用户信息注入到这一次请求中，方便全局使用  
 g.current\_user = data  
 if data.get("userId") is not token\_strategy.SUPER\_ADMIN\_ID and access is not None:  
 return access in data.get("perms", "").split(",")  
 return True  
 else:  
 return False  
  
  
class CacheTokenStrategy(AbstractTokenStrategy):  
 """  
 缓存Token存储策略  
 """  
 from cacheout import LFUCache # 最小频率使用机制  
 cache = LFUCache(maxsize=1000)  
  
 def set(self, data):  
 token = data.get("token", TokenStrategyFactory.create\_token(data))  
 data['token'] = token  
 self.cache.set(token, json.dumps(data, ensure\_ascii=False), ttl=self.TOKEN\_EXPIRE\_TIMEOUT)  
 return data  
  
 def get(self, token):  
 data = self.cache.get(token)  
 if data is None:  
 return None  
 # 重新设置，延长存活时间  
 self.cache.set(token, data, ttl=self.TOKEN\_EXPIRE\_TIMEOUT)  
 return json.loads(data)  
  
 def remove(self, token):  
 if token is None:  
 token = request.headers.get(self.TOKEN\_KEY, "")  
 self.cache.delete(token)  
  
  
class AccessToken(BaseModel):  
 """  
 access\_token存储表  
 """  
 \_\_tablename\_\_ = "t\_access\_token"  
 \_\_table\_args\_\_ = ({"comment": "Token"})  
 id = db.Column(db.Integer, primary\_key=True, comment="主键")  
 user\_id = db.Column(db.Integer, name="user\_id", unique=False, nullable=False, comment="用户id")  
 user\_name = db.Column(db.String(32), name="user\_name", unique=False, nullable=False, comment="用户名")  
 token = db.Column(db.String(40), name="token", unique=True, nullable=False, comment="token")  
 perms = db.Column(db.String(1000), name="perms", unique=False, nullable=True, comment="权限集合")  
  
  
class MysqlTokenStrategy(AbstractTokenStrategy):  
 """  
 mysql Token存储策略  
 """  
  
 def set(self, data):  
 accessToken = AccessToken()  
 accessToken.token = data.get("token", TokenStrategyFactory.create\_token(data))  
 accessToken.user\_id = data.get("userId", data.get("id"))  
 accessToken.user\_name = data.get("userName", "")  
 accessToken.perms = data.get("perms", "")  
 db.session.add(accessToken)  
 db.session.commit()  
 data['token'] = accessToken.token  
 return data  
  
 def get(self, token):  
 now = datetime.datetime.now()  
 delta = datetime.timedelta(seconds=self.TOKEN\_EXPIRE\_TIMEOUT)  
 expireTime = now - delta  
 accessToken = AccessToken.query.filter\_by(token=token).filter(AccessToken.update\_time > expireTime).first()  
 if accessToken is None:  
 return None  
 else:  
 # 更新时间,基类已经做了自动更新更新时间，这里只修改一个字段触发更新操作即可  
 AccessToken.query.filter\_by(id=accessToken.id).update({  
 AccessToken.is\_deleted: accessToken.is\_deleted  
 })  
 # 提交事务  
 db.session.commit()  
 return AccessToken.to\_dict(accessToken)  
  
 def remove(self, token):  
 if token is None:  
 token = request.headers.get(self.TOKEN\_KEY, "")  
 AccessToken.query.filter\_by(token=token).delete()  
  
  
class RedisTokenStrategy(AbstractTokenStrategy):  
 """  
 Redis Token存储策略  
 """  
  
 def set(self, data):  
 token = data.get('token', TokenStrategyFactory.create\_token(data))  
 data['token'] = token  
 token = self.TOKEN\_KEY\_PREFIX + token  
 p = redis\_client.pipeline()  
 p.set(token, json.dumps(data, ensure\_ascii=False))  
 p.expire(token, self.TOKEN\_EXPIRE\_TIMEOUT)  
 p.execute()  
 return data  
  
 def get(self, token):  
 data = redis\_client.get(self.TOKEN\_KEY\_PREFIX + token)  
 if data is None:  
 return None  
 p = redis\_client.pipeline()  
 p.expire(self.TOKEN\_KEY\_PREFIX + token, self.TOKEN\_EXPIRE\_TIMEOUT)  
 p.execute()  
 return json.loads(data.decode("utf-8"))  
  
 def remove(self, token):  
 if token is None:  
 token = request.headers.get(self.TOKEN\_KEY, "")  
 p = redis\_client.pipeline()  
 p.delete(self.TOKEN\_KEY\_PREFIX + token)  
 p.execute()

# 修改mauth/\_\_init\_\_.py

新增token与权限的判断

from functools import wraps  
from flask import abort  
  
from mauth.mtoken import TokenStrategyFactory  
  
  
class HasPerm(ｏｂｊｅｃｔ):  
 """  
 权限装饰器类  
 """  
  
 def \_\_init\_\_(self, access=None, name=None):  
 self.access = access  
 self.name = name  
  
 def \_\_call\_\_(self, func):  
 @wraps(func)  
 def wrapped\_function(\*args, \*\*kwargs):  
 # 这里处理权限拦截的逻辑  
 if not TokenStrategyFactory.check\_token(self.access):  
 abort(403)  
 else:  
 return func(\*args, \*\*kwargs)  
  
 return wrapped\_function

# 修改controllers/user\_controller.py

新增登录与退出处理逻辑

from flask import Blueprint, request  
  
from controllers import R  
from mauth import HasPerm, TokenStrategyFactory  
from models.user import User  
from services.user\_service import UserService  
from validators import BasePageForm  
from validators.id\_validator import IdForm, IdsForm  
from validators.user\_validator import UserForm, LoginForm  
  
# 声明一个蓝图  
user = Blueprint('user', \_\_name\_\_, url\_prefix="/user")  
  
# 声明一个用户业务服务  
user\_service = UserService(model=User)  
  
  
@user.route("/get", methods=['POST'])  
@HasPerm(access="user:get", name="通过id获取用户信息")  
def user\_get():  
 """  
 通过id获取用户信息  
 :return:  
 """  
 form = IdForm()  
 form.validate\_for\_api()  
 # 可通过form.data获取所有提交参数  
 # 或者直接拿id值 id=form.id.data  
 # u = User.query.filter\_by(id=form.id.data).first()  
 # 通过主键查询  
 u = user\_service.get(form)  
 if u is not None:  
 return R.data(u.to\_dict(camel=True))  
 else:  
 return R.fail("该记录不存在")  
  
  
@user.route("/list", methods=['POST'])  
@HasPerm(access="user:list", name="分页查询用户列表")  
def user\_list():  
 """  
 分页查询用户列表  
 :return:  
 """  
 form = BasePageForm()  
 form.validate\_for\_api()  
 return R.data(user\_service.list(form))  
  
  
@user.route("/save", methods=['POST'])  
@HasPerm(access="user:save", name="添加用户")  
def user\_save():  
 """  
 添加用户  
 :return:  
 """  
 form = UserForm()  
 form.validate\_for\_api()  
 # 可通过form.data获取所有提交参数  
 # print(form.data)  
 user\_service.save(form)  
 return R.success("添加用户成功")  
  
  
@user.route("/update", methods=['POST'])  
@HasPerm(access="user:update", name="修改用户")  
def user\_update():  
 """  
 修改用户  
 :return:  
 """  
 form = UserForm()  
 form.validate\_for\_api()  
 # 可通过form.data获取所有提交参数  
 # print(form.data)  
 user\_service.update(form)  
 return R.success("修改用户成功")  
  
  
@user.route("/delete", methods=['POST'])  
@HasPerm(access="user:delete", name="删除用户")  
def user\_delete():  
 """  
 删除用户  
 :return:  
 """  
 form = IdsForm()  
 form.validate\_for\_api()  
 # 可通过form.data获取所有提交参数  
 # print(form.data)  
 user\_service.delete(form)  
 return R.success("删除用户成功")  
  
  
@user.route("/login", methods=['POST'])  
def user\_login():  
 """  
 登录  
 :return:  
 """  
 form = LoginForm()  
 form.validate\_for\_api()  
 # 可通过form.data获取所有提交参数  
 # print(form.data)  
 res = user\_service.login(form.userName.data, form.password.data)  
 token\_strategy = TokenStrategyFactory.get\_instance()  
 return R.data(token\_strategy.set({  
 "userId": res.get("id"),  
 "userName": res.get("userName")  
 }))  
  
  
@user.route('/logout')  
def user\_logout():  
 """  
 退出  
 :return:  
 """  
 token\_strategy = TokenStrategyFactory.get\_instance()  
 token\_strategy.remove(None)  
 return R.success()  
  
  
@user.route("/saveBatch", methods=['GET', 'POST'])  
def user\_save\_batch():  
 """  
 批量插入用户-开启事务  
 :return:  
 """  
 # 复杂的表单校验-wtforms支持不是很好，这里先不校验  
 user\_service.save\_batch(request.get\_json())  
 return R.success("添加成功")  
  
  
@user.route("/saveBatchNoTrans", methods=['POST'])  
def user\_save\_batch\_no\_trans():  
 """  
 批量插入用户-未开启事务  
 :return:  
 """  
 # 复杂的表单校验-wtforms支持不是很好，这里先不校验  
 user\_service.save\_batch\_no\_trans(request.get\_json())  
 return R.success("添加成功")

# 修改config.py

新增redis与token相关配置

import os  
from urllib import parse  
  
  
class BaseConfig(ｏｂｊｅｃｔ):  
 """  
 基础配置  
 """  
 DEBUG = True # 调试模式  
 APP\_AUTHOR = "mldong" # 作者  
 WHITE\_LIST = ['/user/get', '/user/list', '/ex/test', '/ex/other', "/db/test", "/config/test"] # 权限白名单  
 DB\_HOST = os.getenv("DB\_HOST", "localhost") # 数据库ip  
 DB\_PORT = os.getenv("DB\_PORT", "3306") # 数据库端口  
 DB\_NAME = os.getenv("DB\_NAME", "flaskapp") # 数据库名称  
 DB\_USER = os.getenv("DB\_USER", "root") # 数据库用户  
 DB\_PASSWORD = os.getenv("DB\_PASSWORD", "root") # 数据库密码  
 # 数据库相关配置结束  
 # 防止密码中有特殊字符，需要使用parse.quote\_plus进行转义  
 SQLALCHEMY\_DATABASE\_URI = f'mysql+pymysql://{DB\_USER}:{parse.quote\_plus(DB\_PASSWORD)}@{DB\_HOST}:{DB\_PORT}/{DB\_NAME}?charset=utf8mb4'  
 SQLALCHEMY\_TRACK\_MODIFICATIONS = True  
 SQLALCHEMY\_RECORD\_QUERIES = True  
 SQLALCHEMY\_ENGINE\_OPTIONS = {  
 'pool\_size': 5,  
 'pool\_timeout': 90,  
 'pool\_recycle': 7200,  
 'max\_overflow': 1024  
 }  
 # 数据库相关配置结束  
 JSON\_AS\_ASCII = False # 禁止中文转义  
 # ------redis相关配置------  
 # 环境变量中获取数据库REDIS\_HOST  
 REDIS\_HOST = os.getenv("REDIS\_HOST", "localhost")  
 # 环境变量中获取数据库REDIS\_PORT  
 REDIS\_PORT = os.getenv("REDIS\_PORT", "6379")  
 # 环境变量中获取数据库REDIS\_PASSWORD  
 REDIS\_PASSWORD = os.getenv("REDIS\_PASSWORD", "")  
 # REDIS\_URL = "redis://:password@localhost:6379/0" --->flask-redis需要该配置  
 REDIS\_URL = f'redis://:{REDIS\_PASSWORD}@{REDIS\_HOST}:{REDIS\_PORT}/0'  
 # ------redis相关配置------  
 # ------token相关配置------  
 # token 过期时间- 单位是s # 2小时  
 TOKEN\_EXPIRE\_TIMEOUT = int(os.getenv("TOKEN\_EXPIRE\_TIMEOUT", 60 \* 60 \* 2))  
 # token-key  
 TOKEN\_KEY = os.getenv("TOKEN\_KEY", "token")  
 # token 前辍  
 TOKEN\_KEY\_PREFIX = os.getenv("TOKEN\_KEY", "flaskapp:")  
 # 超级管理员id  
 SUPER\_ADMIN\_ID = os.getenv("SUPER\_ADMIN\_ID", 1)  
 # Token存储策略  
 TOKEN\_STRATEGY = os.getenv("TOKEN\_STRATEGY", "redis")  
 # ------token相关配置------  
  
  
class Development(BaseConfig):  
 PORT = 5000  
 DEBUG = True  
 ENV = "dev"  
 TESTING = True  
 SQLALCHEMY\_ECHO = True # 打印SQL  
  
  
class Test(BaseConfig):  
 PORT = 5000  
 DEBUG = True  
 ENV = "test"  
 TESTING = True  
  
  
class Production(BaseConfig):  
 PORT = 5000  
 DEBUG = False  
 ENV = "prod"  
 TESTING = False  
  
  
def getConfig():  
 # 从环境变量中加载ENV  
 env = os.getenv("ENV", "dev")  
 if env == "test":  
 return Test  
 elif env == "prod" or env == "production":  
 return Production  
 else:  
 return Development

# 修改app.py

注册flask\_redis

from flask import Flask, request  
  
from config import getConfig  
from controllers import R  
from controllers.role\_controller import role  
from controllers.user\_controller import user  
from mauth.mtoken import redis\_client  
from models import db  
  
app = Flask(\_\_name\_\_)  
  
# 注册用户模块  
app.register\_blueprint(user)  
# 注册角色模块  
app.register\_blueprint(role)  
# 从配置对象中加载  
app.config.from\_ｏｂｊｅｃｔ(getConfig())  
  
# 初始化db  
db.init\_app(app)  
# 初始化redis  
redis\_client.init\_app(app)  
  
  
# @app.before\_request  
def auth():  
 """  
 权限拦截  
 :return:  
 """  
 # 简单处理一下，非白名单路由，提示登录  
 if request.path not in app.config['WHITE\_LIST']:  
 return R.fail("请先登录")  
  
  
@app.route("/ex/test")  
def ex\_test():  
 """  
 测试其他异常  
 :return:  
 """  
 a = 3 / 0  
 return a  
  
  
@app.route("/db/test")  
def db\_test():  
 """  
 数据库连接测试  
 :return:  
 """  
 cursor = db.session.execute('select \* from t\_user')  
 result = cursor.fetchall()  
 if len(result) > 0:  
 u = result[0]  
 return R.data({  
 "id": u.id,  
 "userName": u.user\_name  
 })  
 return R.fail("无记录")  
  
  
@app.route("/config/test")  
def config\_test():  
 """  
 配置测试  
 :return:  
 """  
 return R.data({  
 'APP\_AUTHOR': app.config['APP\_AUTHOR'],  
 "ENV": app.config['ENV']  
 })  
  
  
@app.errorhandler(404)  
def error\_404(e):  
 """  
 404异常处理  
 :param e:  
 :return:  
 """  
 return R.fail("请求地址不存在")  
  
  
@app.errorhandler(Exception)  
def error(e):  
 """  
 其他异常处理  
 :param e: 异常  
 :return:  
 """  
 return R.fail(str(e))  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 app.run(host="0.0.0.0")

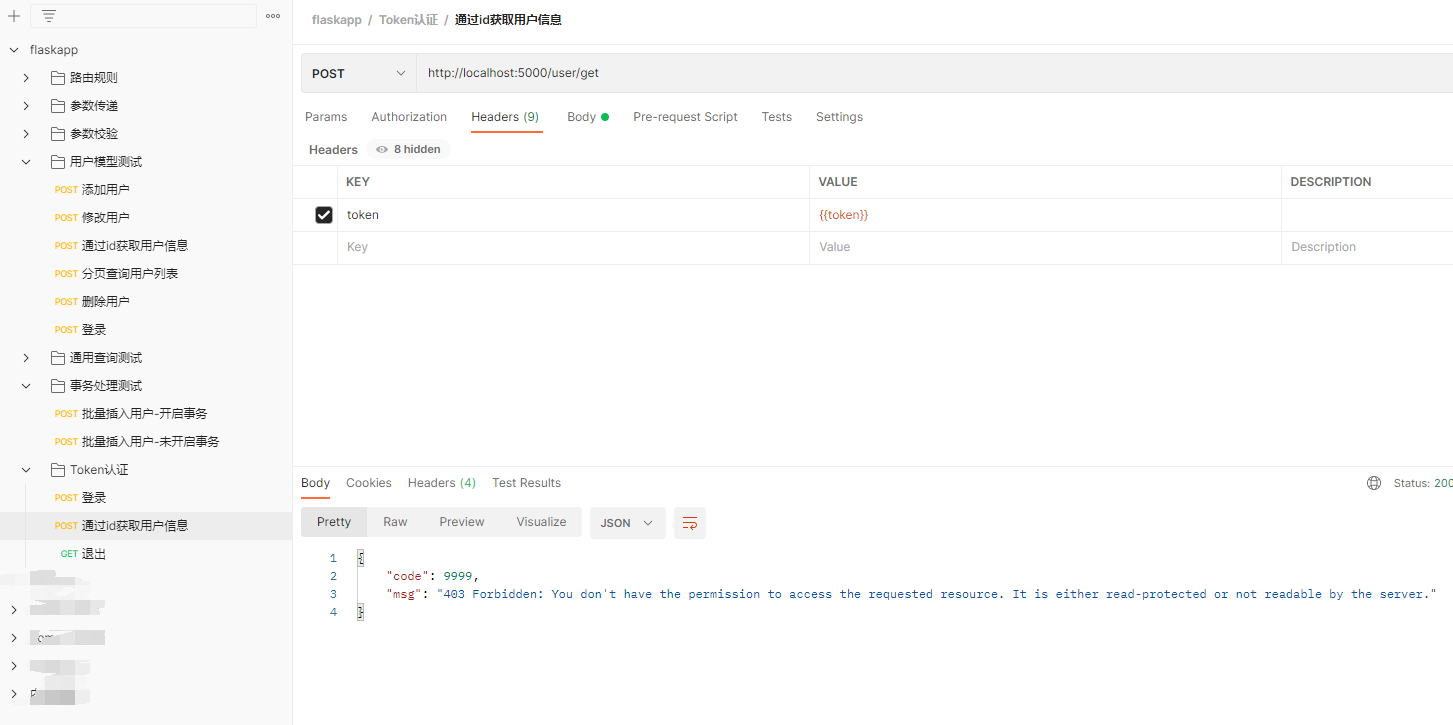
# 运行Flask服务

默认为CacheTokenStrategy存储策略

flask run

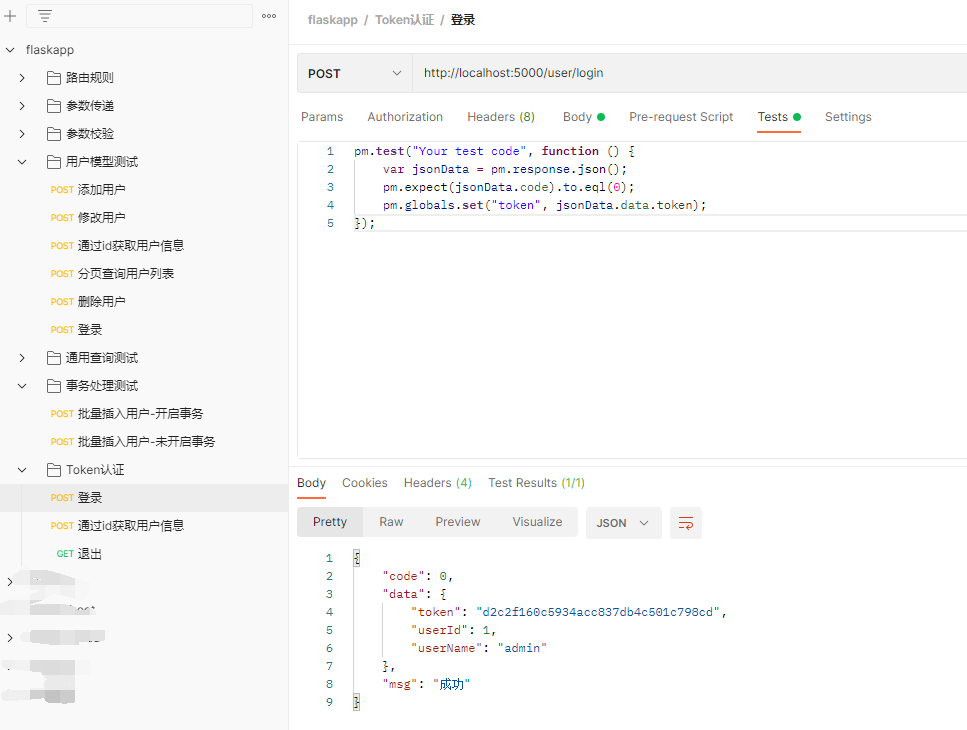
# 使用Postman接口测试工具访问

## 访问/user/get



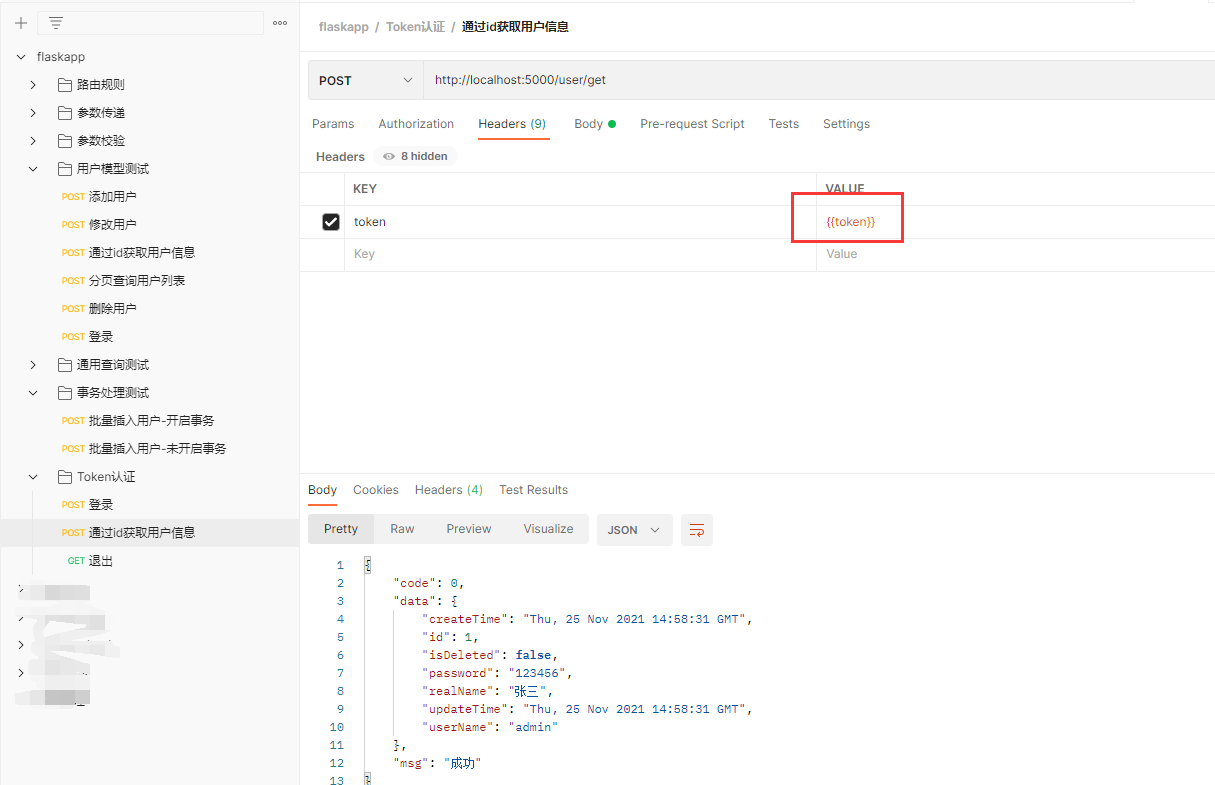
## 访问/user/login

登录后，使用tests设置全局token,方便下个接口使用。

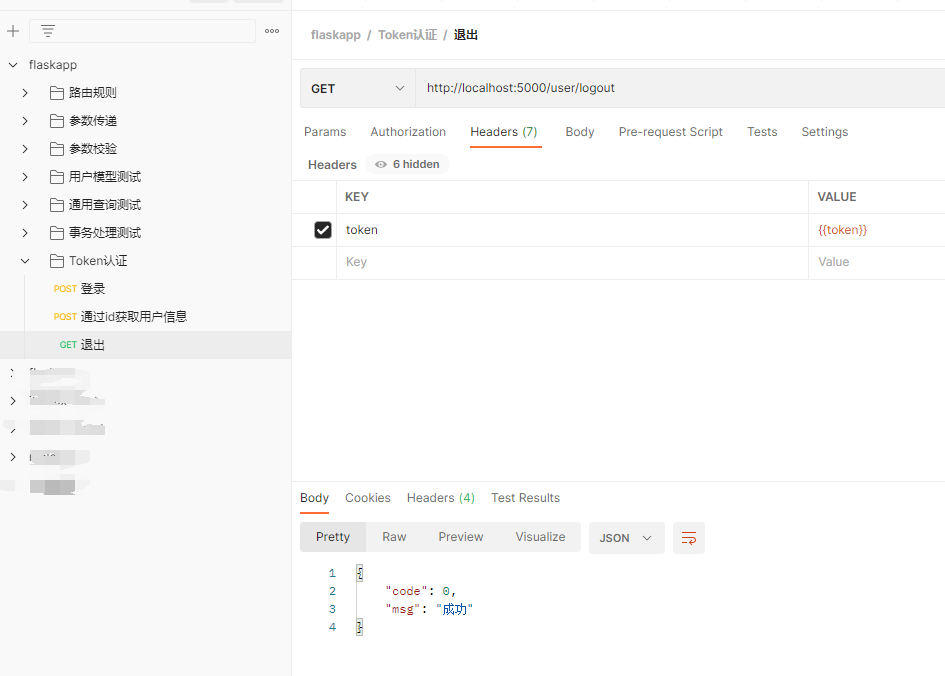


## 再次访问/user/get

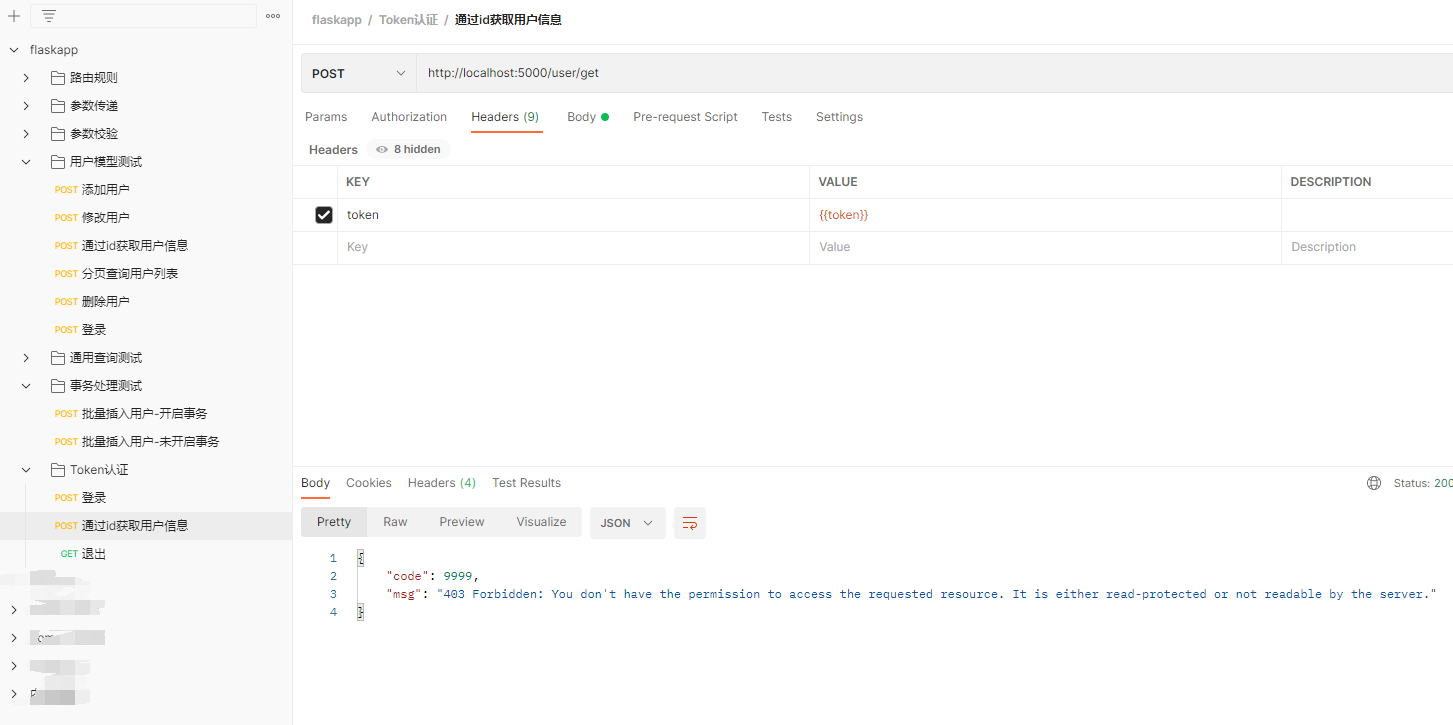
使用上文登录的token



## 访问/user/logout



## 再次访问/user/get



# 更换MysqlTokenStrategy存储策略

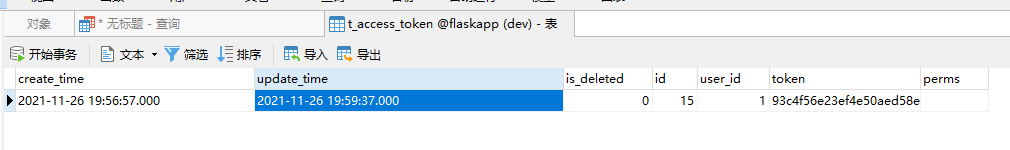
要先创建表

再次测试-略

CREATE TABLE `t\_access\_token` (  
 `create\_time` datetime(3) DEFAULT NULL COMMENT '创建时间',  
 `update\_time` datetime(3) DEFAULT NULL COMMENT '更新时间',  
 `is\_deleted` int(11) DEFAULT NULL COMMENT '逻辑删除:0=未删除,1=删除',  
 `id` int(11) NOT NULL AUTO\_INCREMENT COMMENT '主键',  
 `user\_id` int(11) NOT NULL COMMENT '用户id',  
 `token` varchar(40) NOT NULL COMMENT 'token',  
 `perms` varchar(1000) DEFAULT NULL COMMENT '权限集合',  
 PRIMARY KEY (`id`),  
 UNIQUE KEY `token` (`token`)  
) ENGINE=InnoDB AUTO\_INCREMENT=15 DEFAULT CHARSET=utf8mb4;

export TOKEN\_STRATEGY=mysql  
 flask run

登录成功，数据库会插入一条记录：



# 更换RedisTokenStrategy存储策略

需要安装redis

export TOKEN\_STRATEGY=redis  
 flask run

再次测试-略

登录成功，效果如下：



# 最后

目前只实现了登录权限的拦截，具体的接口权限拦截因为涉及到RBAC相关的权限分配等，这里先不展开，后续做到完整的权限管理的时候，再实现。

# postman导出文件

