**7.3业务逻辑层·事务装饰器**

为了处理事务更优雅一些，我们可以增加一个事务装饰器，让装饰器帮我们自动管理事务。

装饰器本质上是一个Python函数，它可以让其他函数在不需要做任何代码变动的前提下增加额外功能，装饰器的返回值也是一个函数对象。它经常用于有切面需求的场景，比如：插入日志、性能测试、事务处理、缓存、权限校验等场景。装饰器是解决这类问题的绝佳设计，有了装饰器，我们就可以抽离出大量与函数功能本身无关的雷同代码并继续重用。概括的讲，装饰器的作用就是为已经存在的对象添加额外的功能。

# 创建目录decorators

mkdir decorators

# 新增文件decorators/\_\_init\_\_.py

from functools import wraps  
  
  
class Transactional(ｏｂｊｅｃｔ):  
 """  
 事务装饰器类  
 """  
 def \_\_init\_\_(self, db=None):  
 if db is None:  
 from models import db  
 self.db = db  
 else:  
 self.db = db  
  
 def \_\_call\_\_(self, func):  
 @wraps(func)  
 def wrapped\_function(\*args, \*\*kwargs):  
 # 开始事务，with上下文管理器，会自动commit和rollback  
 with self.db.session.begin():  
 # 调用具体的方法  
 res = func(\*args, \*\*kwargs)  
 return res  
  
 return wrapped\_function

# 重构services/\_\_init\_\_.py

新增装饰器注解，去掉手动commit

from decorators import Transactional  
from tools.db\_tool import DbTool  
  
  
class BaseService(ｏｂｊｅｃｔ):  
 """  
 业务逻辑怪基类  
 """  
  
 def \_\_init\_\_(self, db=None, model=None):  
 if db is not None:  
 self.db = db  
 else:  
 from models import db  
 self.db = db  
 if model is not None:  
 self.model = model  
 else:  
 raise Exception("model不能为空")  
  
 def get(self, form):  
 """  
 通过id获取用户信息  
 :param form:  
 :return:  
 """  
 model = self.db.session.query(self.model).get(form.id.data)  
 return model  
  
 def list(self, form):  
 """  
 分页查询用户列表  
 :param form:  
 :return:  
 """  
 # 可通过form.data获取所有提交参数  
 # 可通过form.pageNum.data获取pageNum  
 # 可通过form.pageSize.data获取pageSize  
 # page=self.db.query(User).filter().paginate(form.pageNum.data, form.pageSize.data,False)  
 page = DbTool.filter\_by\_custom(self.model).paginate(form.pageNum.data, form.pageSize.data, False)  
 return self.model.to\_page(page)  
  
 @Transactional()  
 def save(self, form):  
 """  
 添加用户  
 :param form:  
 :return:  
 """  
 model = self.model(\*\*form.data)  
 self.db.session.add(model)  
  
 @Transactional()  
 def update(self, form):  
 """  
 修改用户  
 :param form:  
 :return:  
 """  
 model = self.model(\*\*form.data)  
 self.db.session.query(self.model).filter\_by(id=form.id.data).update(model.to\_dict(camel=False))  
  
 @Transactional()  
 def delete(self, form):  
 """  
 删除用户  
 :param form:  
 :return:  
 """  
 self.db.session.query(self.model).filter(self.model.id.in\_(form.ids.data)).delete()

# 修改services/user\_service.py

from decorators import Transactional  
from models.user import User  
from services import BaseService  
  
  
class UserService(BaseService):  
 """  
 用户模块业务处理类  
 """  
 def login(self, user\_name, password):  
 """  
 用户密码登录  
 :param user\_name:  
 :param password:  
 :return:  
 """  
 u = self.db.session.query(User).filter(User.user\_name == user\_name).first()  
 if u is None:  
 raise Exception("用户名或密码错误")  
 if u.password != password:  
 raise Exception("用户名或密码错误")  
 res = u.to\_dict(camel=True)  
 del res['password']  
 del res['createTime']  
 del res['updateTime']  
 del res['isDeleted']  
 return res  
  
 @Transactional()  
 def save\_batch(self, form\_list):  
 """  
 批量插入用户-开启事务  
 :param form\_list:  
 :return:  
 """  
 for form in form\_list:  
 model = User(\*\*form)  
 self.db.session.add(model)  
 # flush会将session中的数据刷到数据库中，使数据库主键自增；但不会写到磁盘里  
 self.db.session.flush()  
  
 def save\_batch\_sub\_trans(self, form\_list):  
 """  
 批量插入用户-未开启事务  
 :param form\_list:  
 :return:  
 """  
 for form in form\_list:  
 model = User(\*\*form)  
 self.db.session.add(model)  
 self.db.session.commit()

# 运行Flask服务

flask run

# 测试前重新建表

drop table if exists t\_user;  
CREATE TABLE `t\_user` (  
 `id` bigint(20) NOT NULL AUTO\_INCREMENT COMMENT '主键',  
 `user\_name` varchar(32) NOT NULL COMMENT '用户名',  
 `real\_name` varchar(32) NOT NULL COMMENT '姓名',  
 `password` varchar(64) NOT NULL COMMENT '密码',  
 `create\_time` datetime DEFAULT NULL COMMENT '创建时间',  
 `update\_time` datetime DEFAULT NULL COMMENT '更新时间',  
 `is\_deleted` tinyint(1) DEFAULT NULL COMMENT '逻辑删除:0=未删除,1=删除',  
 PRIMARY KEY (`id`),  
 UNIQUE KEY `user\_name` (`user\_name`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COMMENT='用户';

# 使用Postman接口测试工具访问

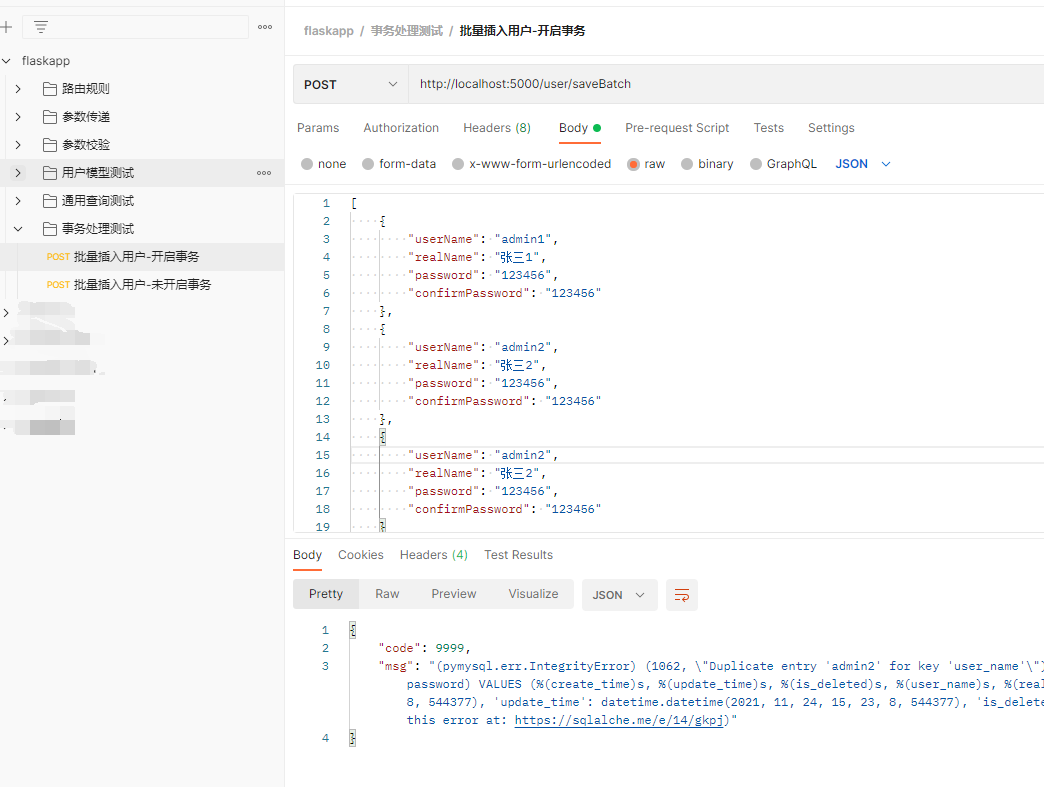
## 访问/user/saveBatch

因为user\_name设置为唯一，所以可以人为造一些用户名重复的数据

[  
 {  
 "userName": "admin1",  
 "realName": "张三1",  
 "password": "123456",  
 "confirmPassword": "123456"  
 },  
 {  
 "userName": "admin2",  
 "realName": "张三2",  
 "password": "123456",  
 "confirmPassword": "123456"  
 },  
 {  
 "userName": "admin2",  
 "realName": "张三2",  
 "password": "123456",  
 "confirmPassword": "123456"  
 }  
]

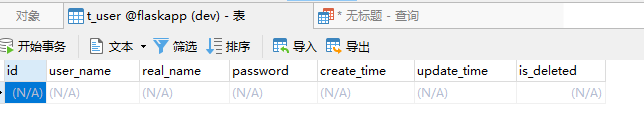
执行结果如下：

存在用户名唯一冲突



去数据库查看如下：

数据插入失败-事务生效-回滚成功



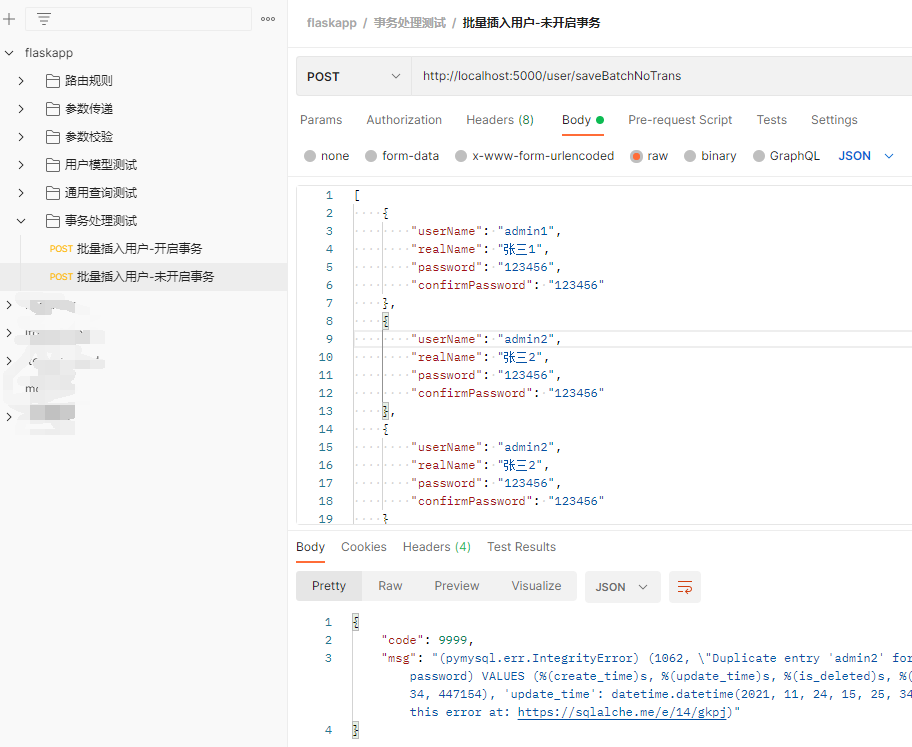
## 访问/user/saveBatchNoTrans

使用相同的数据

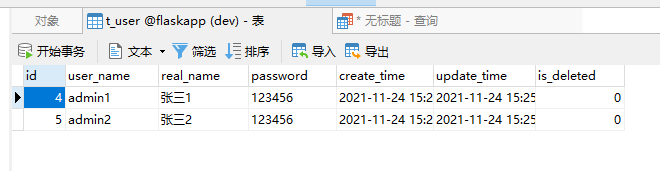
[  
 {  
 "userName": "admin1",  
 "realName": "张三1",  
 "password": "123456",  
 "confirmPassword": "123456"  
 },  
 {  
 "userName": "admin2",  
 "realName": "张三2",  
 "password": "123456",  
 "confirmPassword": "123456"  
 },  
 {  
 "userName": "admin2",  
 "realName": "张三2",  
 "password": "123456",  
 "confirmPassword": "123456"  
 }  
]

执行结果如下：

一样存在用户名冲突



但是前两条记录插入成功了-没回滚



至此，事务验证结束。

# 关于嵌套事务

如上述的已经由装饰器类自动管理事务后，如果有个方法test\_method要如下方式调用：

@Transactional()  
def test\_method(self):  
 self.save(...)  
 self.update(...)

因存在多次开启事务的问题，会报错，这里目前尚未有优雅的处理方案，所以暂时未处理，只能大家在开发的时候尽量避免该类操作。

# postman导出文件

