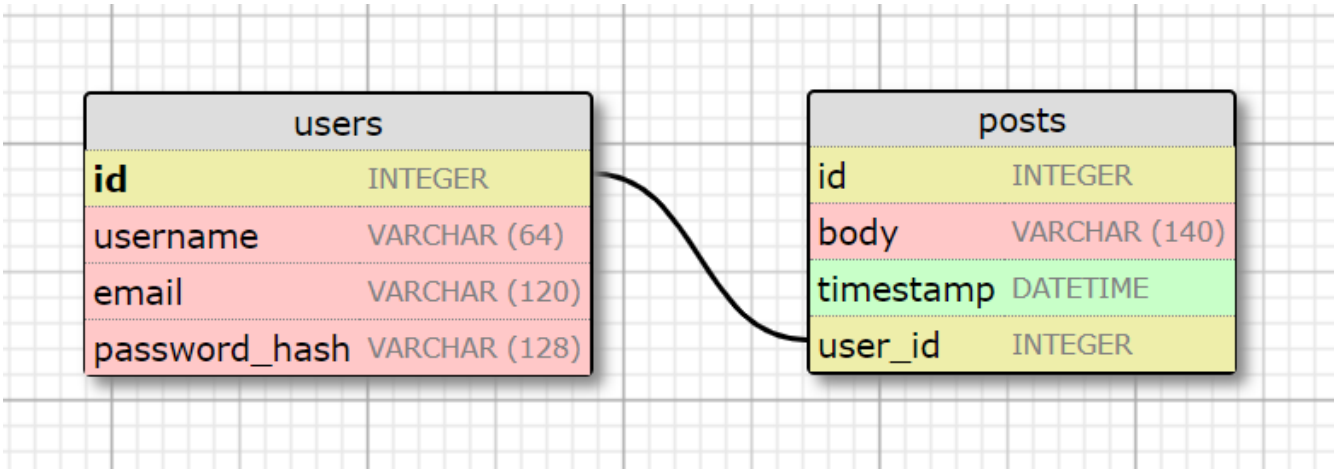
粉丝机制实现

1. 数据库关系分析

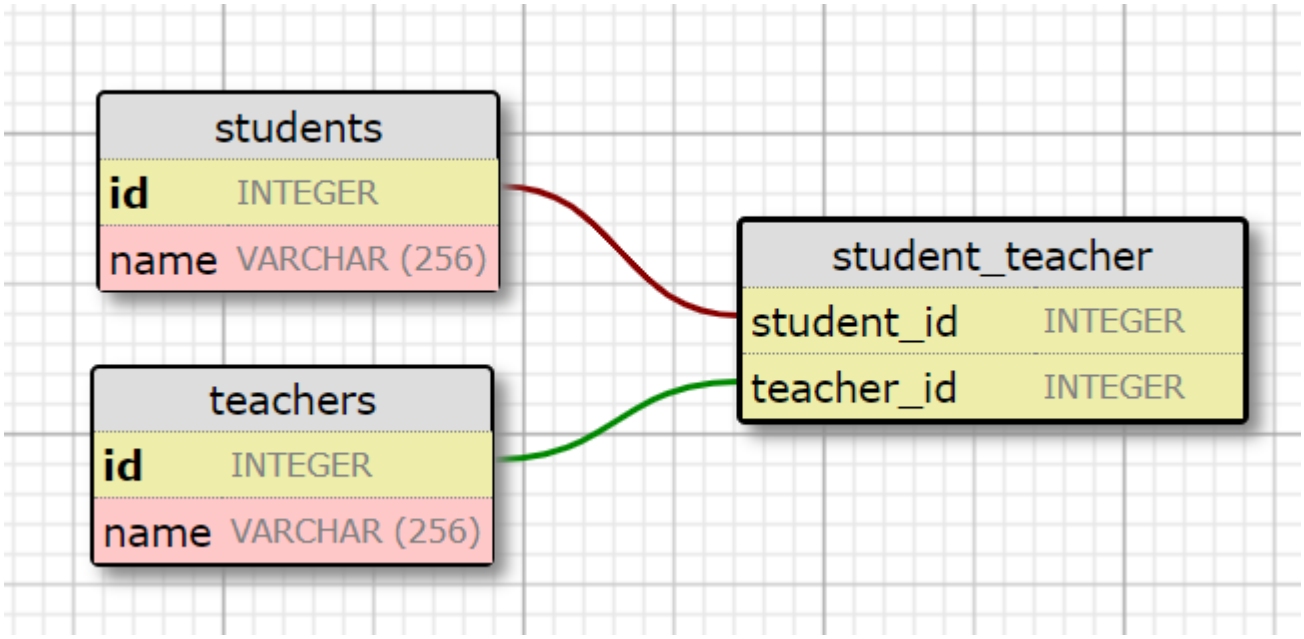
1.1. 一对多

已经实现的用户与帖子的关系即为一对多关系，一个用户对应多个帖子信息。外键是`post`表的`user_id`字段，可以冠梁`user`表中的用户信息，E-R图如下：



1.2. 多对多

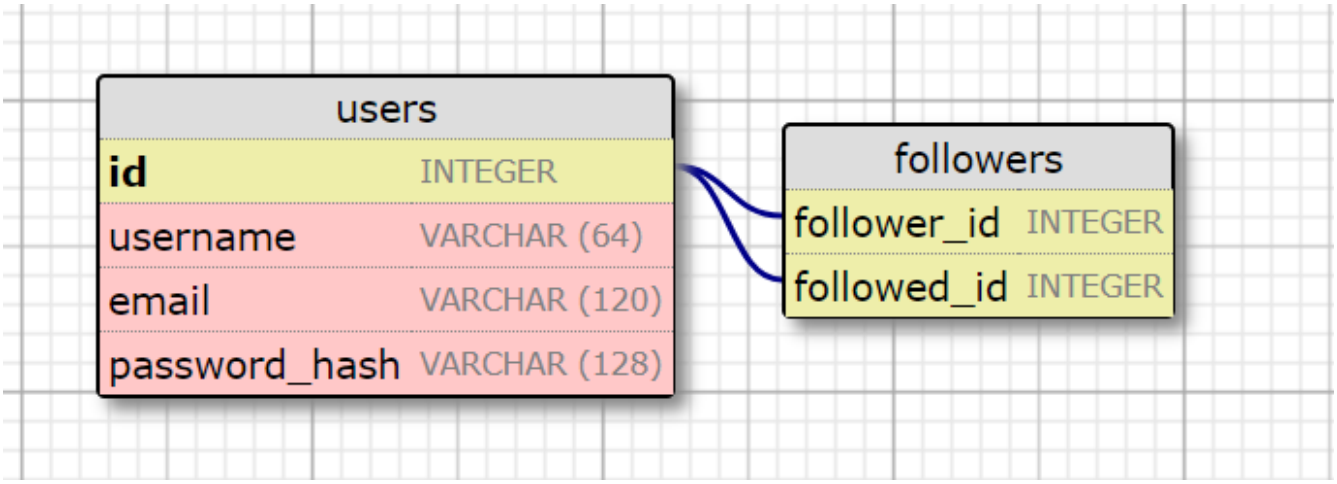
多对多关系类似于老师跟学生的关系，一个老师对应多个学生，一个学生对应多个老师，E-R图如下：



2. 粉丝机制数据库模型改造

2.1. 自引用的多对多关系

关注者和被关注者也属于多对多关系，一个用户可以关注多个用户，一个用户也可以被多个人关注，只不过关注者和被关注者同属于一个用户类的实例的对象。 一个类的实例被关联到同一个类的其他实例的关系就是自引用关系。E-R图如下：



2.2. 数据库模型改造

修改`app/models.py`脚本，增加关注关系表`followers`用于记录关注与被关注的关系。此表只是一个辅助表，没有声明成模型类。

```
followers = db.Table(
    # 表名
    'followers',
    # 关注者（粉丝）
    db.Column('follower_id', db.Integer, db.ForeignKey('users.id')),
    # 被关注者
    db.Column('followed_id', db.Integer, db.ForeignKey('users.id'))
)
```

修改`app/models.py`脚本中的`User`模型，增加多对多关系。

```
class User(UserMixin, db.Model):
......
    # 最后访问时间
    last_seen = db.Column(db.DateTime, default=datetime.datetime.utcnow)
    # 当前用户的关注关系
    followed = db.relationship(
        # 关系当中的右侧实体（将左侧实体看成是上级类）
        'User',
        # 指定用于该关系的关联表
        secondary=followers,
        # 指明了通过关系表关联到左侧实体（关注者=粉丝）的条件
        # 关系中的左侧的join条件是关系表中的`follower_id`字段与这个关注者的用户ID匹配
        primaryjoin=(followers.c.follower_id == id),
        # 指明了通过关系表关联到右侧实体（被关注者）的条件
        secondaryjoin=(followers.c.followed_id == id),
        # 定义了右侧实体访问该关系的方式
        # 在左侧，关系被命名为followed，所以在右侧我将使用followers来表示所有左侧用户的列表，即粉丝列表
        # 附加的lazy参数表示这个查询的执行模式，设置为动态模式的查询不会立即执行，直到被调用
        backref=db.backref('followers', lazy='dynamic'),
        # 和backref中的lazy类似，只不过当前的这个是应用于左侧实体，backref中的是应用于右侧实体
        lazy='dynamic'
    )
```

2.3. 数据库变更迁移

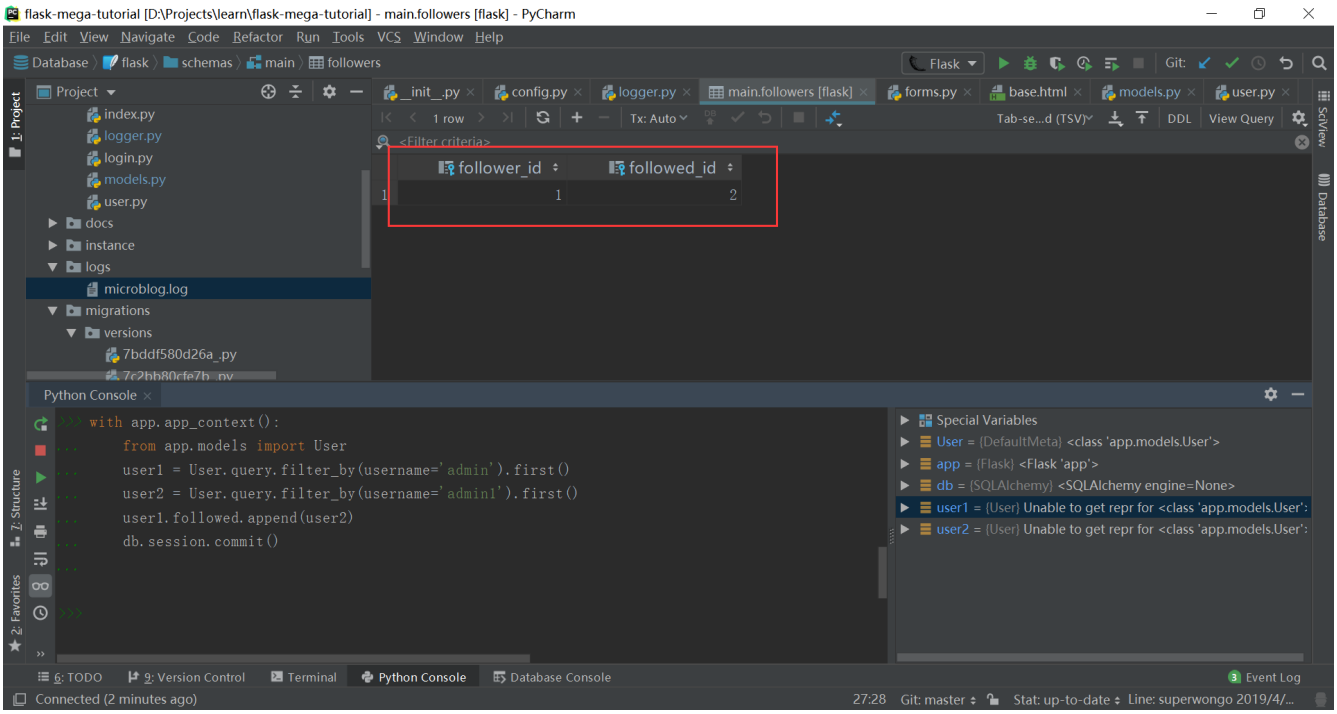
通过`flask db migrate`、`flask db upgrade`命令进行数据库变更迁移。

```
(venv) D:\Projects\learn\flask-mega-tutorial>flask db migrate -m 'followers'
[2019-04-29 16:38:02,892] INFO in logger: 微博已启动
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.autogenerate.compare] Detected added table 'followers'
Generating D:\Projects\learn\flask-mega-
tutorial\migrations\versions\251e4854ef4c_followers.py ... done

(venv) D:\Projects\learn\flask-mega-tutorial>flask db upgrade
[2019-04-29 16:38:13,832] INFO in logger: 微博已启动
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.runtime.migration] Running upgrade 99a18f68c13a -> 251e4854ef4c, 'followers'
```

2.4. 关注与取消关注

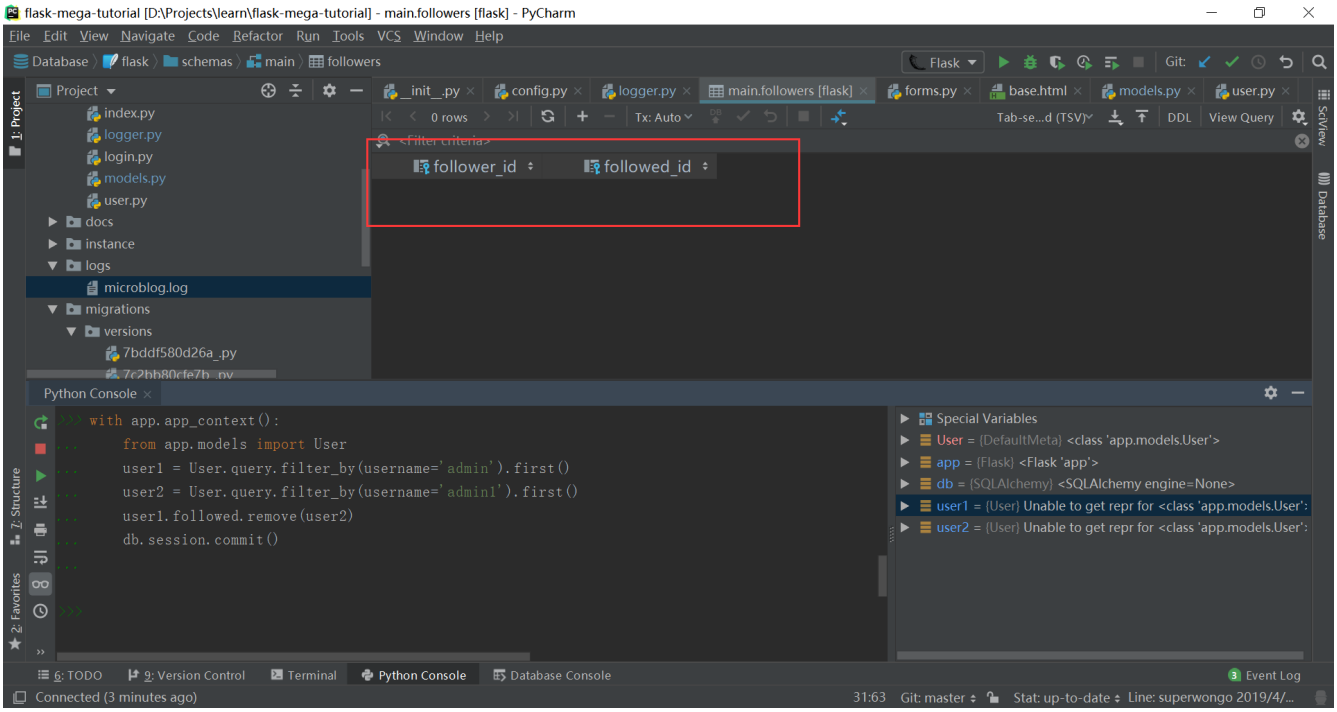
添加关注操作如下：

```
>>> from app import create_app, db
>>> app = create_app()
[2019-04-29 16:47:05,493] INFO in logger: □□□□□□□□□□
>>> app.app_context().push()
>>> with app.app_context():
...     from app.models import User
...     user1 = User.query.filter_by(username='admin').first()
...     user2 = User.query.filter_by(username='admin1').first()
...     user1.followed.append(user2)
...     db.session.commit()
...
```



取消关注操作如下：

```
>>> with app.app_context():
...     from app.models import User
...     user1 = User.query.filter_by(username='admin').first()
...     user2 = User.query.filter_by(username='admin1').first()
...     user1.followed.remove(user2)
...     db.session.commit()
...
```



2.5. 用户模型增加关注与取消关注方法

修改`app/models.py`脚本中的`User`模型，增加关注与取消关注的方法。

```
class User(UserMixin, db.Model):
......
    def is_following(self, user):
        """是否存在关注关系"""
        return self.followed.filter(followers.c.followed_id == user.id).count() > 0

    def follow(self, user):
        """添加关注"""
        if not self.is_following(user):
            self.followed.append(user)

    def unfollow(self, user):
        """取消关注"""
        if self.is_following(user):
            self.followed.remove(user)
......
```

2.6. 用户模型增加查询已关注用户帖子的方法

修改`app/models.py`脚本中的`User`模型，增加查询已关注用户帖子的方法。

```
 class User(UserMixin, db.Model):
......
    def followed_posts(self):
        """已关注用户帖子查询"""
        return Post.query.join(
            followers, (followers.c.followed_id == Post.user_id)
        ).filter(
            followers.c.follower_id == self.id
        ).order_by(Post.timestamp.desc())
......
```

以下通过表格的方式表现以下查询方法的实现步骤：

假设`User`表中存在以下用户信息：

id	username
1	john
2	susan
3	mary
4	david

假设`followers`关系表中数据表达的是用户`john`关注用户`susan`和`david`，用户`susan`关注用户`mary`，用户`mary`关注用户`david`。这些的数据如下表所示：

follower_id（粉丝）	followed_id（作者）
1	2
1	4
2	3
3	4

用户帖子信息表中包含每个用户的帖子信息：

id	text	user_id
1	post from susan	2
2	post from mary	3
3	post from david	4
4	post from john	1

通过`Post.query.join(followers, (followers.c.followed_id == Post.user_id))`进行联合查询，查询条件是作者为被关注用户的帖子信息，类似于`Post`表左连接`User`表。查询结果如下表所示：

id	text	user_id	followed_id（作者）	follower_id（粉丝）
1	post from susan	2	2	1
2	post from mary	3	3	2
3	post from david	4	4	1
3	post from david	4	4	3
~~4~~	~~post from john~~	~~1~~	~~1~~	

通过`filter(followers.c.follower_id == self.id)`语句过滤关注者（粉丝）为本用户的帖子信息。

id	text	user_id	followed_id（作者）	follower_id（粉丝）
1	post from susan	2	2	1
3	post from david	4	4	1

通过`order_by(Post.timestamp.desc())`进行排序处理。

2.7. 组合自身动态和关注的用户动态

修改`app/models.py`脚本中的`User`模型中的`followed_posts`函数，增加本用户帖子查询，并`union`到已关注用户帖子中。

```
class User(UserMixin, db.Model):
......
    def followed_posts(self):
        """已关注用户帖子查询"""
        followed = Post.query.join(
            followers, (followers.c.followed_id == Post.user_id)
        ).filter(
            followers.c.follower_id == self.id
        )
        own = Post.query.filter_by(user_id=self.id)
        return followed.union(own).order_by(Post.timestamp.desc())
......
```

3. 单元测试脚本编写

3.1. 安装测试用三方库

测试需要安装`pytest`、`coverage`三方库来进行测试和衡量代码。

```
pip install pytest coverage
```

3.2. 测试配置与固件

在应用目录`app`同级下创建`tests`模块，用于放置测试脚本。新建`tests/conftest.py`文件，内部包含名为 *fixtures* （固件）的配置 函数，每个测试都会用到这个函数。测试位于 Python 模块中，以 `test_` 开头， 并且模块中的每个测试函数也以 `test_` 开头。创建`tests/data.sql`文件，用于配置测试临时数据库中初始化数据的SQL语句。

`tests/conftest.py` 文件:

```
import os
import tempfile

import pytest

from app import create_app, db

# 加载初始化SQL语句文件
with open(os.path.join(os.path.dirname(__file__), 'data.sql'), 'rb') as f:
    _data_sql = f.readlines()


@pytest.fixture
def app():
    """初始化应用实例、数据库"""
    # tempfile.mkstemp() 创建并打开一个临时文件，返回该文件对象和路径。
    # DATABASE 路径被重载，这样它会指向临时路径，而不是实例文件夹。
    db_fd, db_path = tempfile.mkstemp()

    # TESTING: 设置 Flask 应用处在测试模式下。
    app = create_app({
        'TESTING': True,
        # 数据库文件存放路径
        'SQLALCHEMY_DATABASE_URI': 'sqlite:///' + db_path
    })

    # 加载应用环境
    with app.app_context():
        # 初始化数据库模型
        db.create_all()
        # 循环执行初始化SQL
        for sql in _data_sql:
            if sql:
                db.engine.execute(sql.decode('utf8'))

    yield app

    # 再次创建APP时，会删除原临时数据库文件，进行数据库重新初始化
    os.close(db_fd)
    os.unlink(db_path)


@pytest.fixture
def client(app):
    """用于调用由 app 固件创建的应用 对象"""
    return app.test_client()
```

```
@pytest.fixture
def runner(app):
    """创建一个运行器，用于调用应用注册的 Click 命令"""
    return app.test_cli_runner()
```

3.3. 工厂函数增加测试配置的传入及重载

修改`app/__init__.py`脚本，工厂函数添加测试配置参数，用于测试配置的重载。

```
......
def create_app(test_config=None):
    """应用工厂函数"""
    application = Flask(__name__)
    # 加载config配置
    # 使用 config.py 中的值来重载缺省配置
    application.config.from_pyfile('config.py', silent=True)

    # test_config：单独设置配置参数，替代实例配置。这样可以实现 测试和开发的配置分离，相互独立。
    if test_config:
        application.config.from_mapping(test_config)
......
```

3.4. 编写测试脚本

新增`app/test_db.py`测试脚本，用户编写数据库模型测试函数。

```
from datetime import datetime, timedelta

from app import db
from app.models import User, Post


def test_password_hashing():
    """测试密码加密"""
    u = User(username='susan')
    u.set_password('cat')
    assert u.check_password('dog') is False
    assert u.check_password('cat') is True


def test_follow(app):
    """测试关注函数"""
    with app.app_context():
        u1 = User(username='john', email='john@example.com')
        u2 = User(username='susan', email='susan@example.com')

        db.session.add(u1)
        db.session.add(u2)
        db.session.commit()
        assert u1.followed.all() == []
```

```
        assert u1.followers.all() == []

        # john关注了susan
        u1.follow(u2)
        db.session.commit()
        assert u1.is_following(u2) is True
        # john用户信息的被关注者是susan
        assert u1.followed.count() == 1
        assert u1.followed.first().username == 'susan'
        # susan用户信息的关注者是john
        assert u2.followers.count() == 1
        assert u2.followers.first().username == 'john'

        # 解除关注
        u1.unfollow(u2)
        db.session.commit()
        assert u1.is_following(u2) is False
        assert u1.followed.count() == 0
        assert u2.followers.count() == 0


def test_follow_posts(app):
    """测试关注帖子获取函数"""
    with app.app_context():
        u1 = User(username='john', email='john@example.com')
        u2 = User(username='susan', email='susan@example.com')
        u3 = User(username='mary', email='mary@example.com')
        u4 = User(username='david', email='david@example.com')
        db.session.add_all([u1, u2, u3, u4])

        now = datetime.utcnow()
        p1 = Post(body="post from john", author=u1, timestamp=now + timedelta(seconds=1))
        p2 = Post(body="post from susan", author=u2, timestamp=now + timedelta(seconds=4))
        p3 = Post(body="post from mary", author=u3, timestamp=now + timedelta(seconds=3))
        p4 = Post(body="post from david", author=u4, timestamp=now + timedelta(seconds=2))
        db.session.add_all([p1, p2, p3, p4])
        db.session.commit()

        # john关注了susan
        u1.follow(u2)
        # john关注了david
        u1.follow(u4)
        # susan关注了mary
        u2.follow(u3)
        # mary关注了david
        u3.follow(u4)
        db.session.commit()

        f1 = u1.followed_posts().all()
        f2 = u2.followed_posts().all()
        f3 = u3.followed_posts().all()
        f4 = u4.followed_posts().all()
        # john获取的帖子包括susan的、david的、自己的
```

```
        assert f1 == [p2, p4, p1]
        # susan获取的帖子包括mary的、自己的
        assert f2 == [p2, p3]
        # mary获取的帖子包括david的、自己的
        assert f3 == [p3, p4]
        # david获取的帖子只有自己的
        assert f4 == [p4]
```

3.5. 创建测试配置文件setup.cfg

与应用`app`同级，新建`setup.cfg`脚本，用于配置测试参数。其中`filterwarnings`用于忽略废弃提示，`testpaths`用于定义测试脚本所在目录，`source`设置`coverage`测试时的应用名称。

```
[tool:pytest]
filterwarnings =
    error
    ignore::DeprecationWarning
testpaths = tests

[coverage:run]
branch = True
source =
    app
```

3.6. 执行测试

通过`pytest`执行所有测试。如果有测试失败， pytest 会显示引发的错误。可以使用 `pytest -v` 得到每个测试的列表，而不是一串点。

```
(venv) D:\Projects\learn\flask-mega-tutorial>pytest
================================================================= test session starts
=================================================================
platform win32 -- Python 3.6.6, pytest-4.4.1, py-1.8.0, pluggy-0.9.0
rootdir: D:\Projects\learn\flask-mega-tutorial, inifile: setup.cfg, testpaths: tests
collected 3 items


tests\test_db.py ...
                                                      [100%]

============================================================== 3 passed in 1.12 seconds
===============================================================

(venv) D:\Projects\learn\flask-mega-tutorial>pytest -v
================================================================= test session starts
=================================================================
platform win32 -- Python 3.6.6, pytest-4.4.1, py-1.8.0, pluggy-0.9.0 --
d:\projects\learn\flask-mega-tutorial\venv\scripts\python.exe
cachedir: .pytest_cache
rootdir: D:\Projects\learn\flask-mega-tutorial, inifile: setup.cfg, testpaths: tests
collected 3 items
```

```
tests/test_db.py::test_password_hashing PASSED
                                                         [ 33%]
tests/test_db.py::test_follow PASSED
                                                         [ 66%]
tests/test_db.py::test_follow_posts PASSED
                                                         [100%]

============================================================ 3 passed in 1.14 seconds
============================================================
```

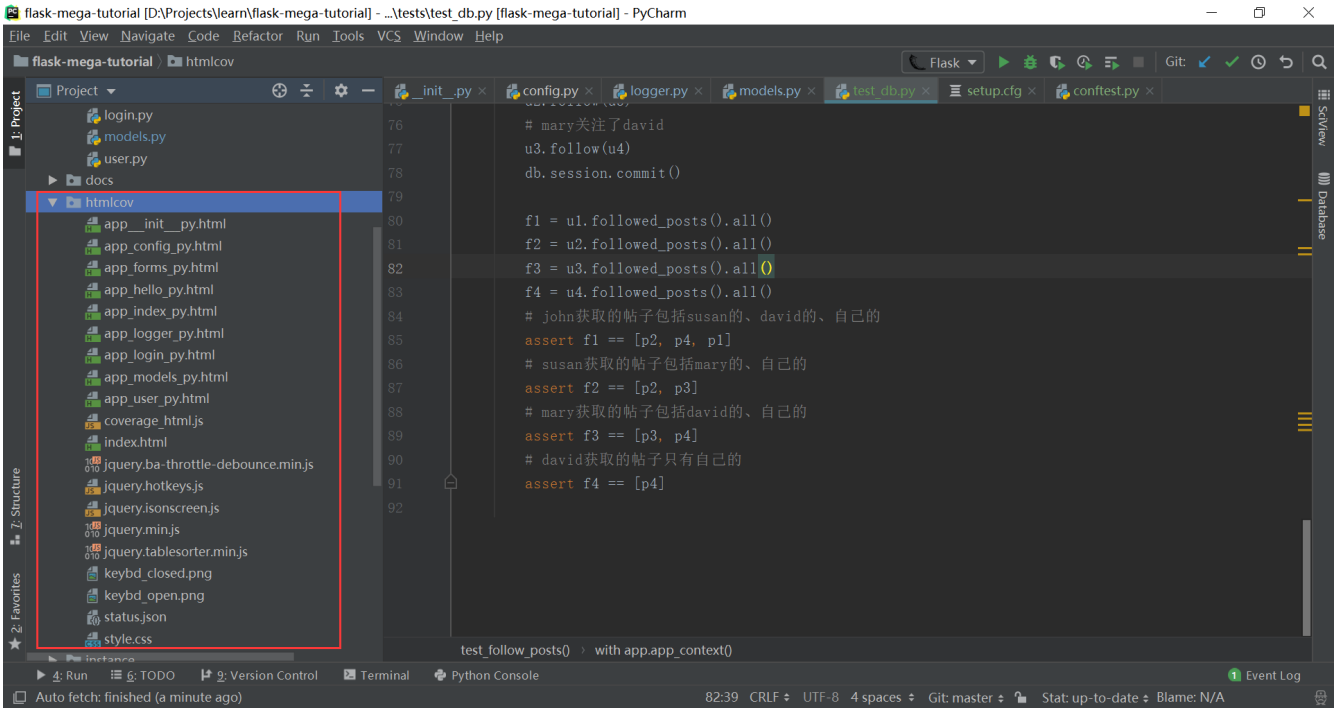
可以使用 `coverage` 命令代替直接使用 pytest 来运行测试，这样可以衡量测试 覆盖率。

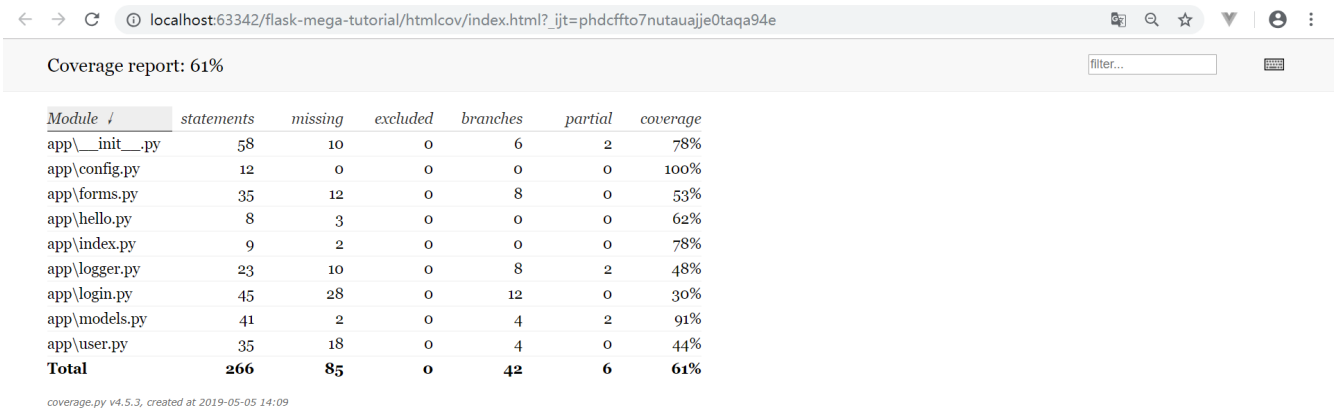
```
(venv) D:\Projects\learn\flask-mega-tutorial>coverage run -m pytest
============================================================================ test session starts
============================================================================
platform win32 -- Python 3.6.6, pytest-4.4.1, py-1.8.0, pluggy-0.9.0
rootdir: D:\Projects\learn\flask-mega-tutorial, inifile: setup.cfg, testpaths: tests
collected 3 items


tests\test_db.py ...
                                                                        [100%]

============================================================================ 3 passed in 1.22 seconds
============================================================================

(venv) D:\Projects\learn\flask-mega-tutorial>coverage report
Name              Stmts   Miss Branch BrPart  Cover
-------------------------------------------------------
app\__init__.py      58     10      6      2    78%
app\config.py        12      0      0      0   100%
app\forms.py         35     12      8      0    53%
app\hello.py          8      3      0      0    62%
app\index.py          9      2      0      0    78%
app\logger.py        23     10      8      2    48%
app\login.py         45     28     12      0    30%
app\models.py        41      2      4      2    91%
app\user.py          35     18      4      0    44%
-------------------------------------------------------
TOTAL               266     85     42      6    61%
```

还可以生成 HTML 报告，可以看到每个文件中测试覆盖了哪些行：

```
coverage html
```

这个命令在 `htmlcov` 文件夹中生成测试报告，然后在浏览器中打开 `htmlcov/index.html` 查看。





Module	statements	missing	excluded	branches	partial	coverage
app__init__.py	58	10	0	6	2	78%
app\config.py	12	0	0	0	0	100%
app\forms.py	35	12	0	8	0	53%
app\hello.py	8	3	0	0	0	62%
app\index.py	9	2	0	0	0	78%
app\logger.py	23	10	0	8	2	48%
app\login.py	45	28	0	12	0	30%
app\models.py	41	2	0	4	2	91%
app\user.py	35	18	0	4	0	44%
Total	**266**	**85**	**0**	**42**	**6**	**61%**

coverage.py v4.5.3, created at 2019-05-05 14:09