2017-07-14，田宇伟

在Flask中使用ORM进行数据库操作

目录

在Flask中使用ORM进行数据库操作

Web 程序最常用基于关系模型的数据库，这种数据库也称为 SQL 数据库，因为它们使用结构化查询语言。文档数据库和键值对数据库，这两种数据库合称 NoSQL 数据库。

Python数据库框架

大多数的数据库引擎都有对应的 Python 包，包括开源包和商业包。还有一些数据库抽象层代码包供选择，比如SQLAlchemy，Peewee和MongoEngine等，可以使用这些抽象包直接处理高等级的 Python 对象，而不用处理如表、文档或查询语言此类的数据库实体。

选择数据库框架时，要考虑的因素：

- 易用性

  如果直接比较数据库引擎和数据库抽象层，显然后者取胜。抽象层，也称为对象关系映射（Object-Relational Mapper，ORM）或对象文档映射（Object-Document Mapper，ODM），在用户不知觉的情况下把高层的面向对象操作转换成低层的数据库指令。

- 性能

  ORM 和 ODM 把对象业务转换成数据库业务会有一定的损耗。大多数情况下，这种性能的降低微不足道，但也不一定都是如此。一般情况下，ORM 和 ODM 对生产率的提升远远超过了这一丁点儿的性能降低，所以性能降低这个理由不足以说服用户完全放弃 ORM 和 ODM。 **真正的关键点在于如何选择一个能直接操作低层数据库的抽象层， 以防特定的操作需要直接使用数据库原生指令优化。**

- 可移植性

选择数据库时，必须考虑其是否能在你的开发平台和生产平台中使用。例如，如果你打算利用云平台托管程序，就要知道这个云服务提供了哪些数据库可供选择。

可移植性还针对 ORM 和 ODM。尽管有些框架只为一种数据库引擎提供抽象层，但其他框架可能做了更高层的抽象，它们支持不同的数据库引擎，而且都使用相同的面向对象接口。

- Flask 集成度

  选择框架时，不一定非得选择已经集成了 Flask 的框架，但选择这些框架可以节省编写集成代码的时间。使用集成了 Flask 的框架可以简化配置和操作。

基于以上因素，我们最终选择Peewee。

Peewee

Peewee是个简单轻量的ORM（对象关系映射），易于使用，默认支持SQLite、MySQL和PostgreSQL。由于其功能比较简单，所以效率比**SQLAlchemy**要略高一些。

使用pip安装Peewee

```
(flask_env3) $ pip install peewee
```

安装完之后就可以在ipython中尝试使用Peewee了。

Model定义

Model类，字段（fields）和Model实例和数据库概念的对应关系：

名称	对应于
Model类	数据库表
字段实例	数据表中的列
Model实例	数据表中的行

下面的代码展示了使用Peewee定义Model，连接数据库并创建数据表：

```
import peewee as pw

db = pw.SqliteDatabase('people.db')

class Person(pw.Model):
    name = pw.CharField()
    birthday = pw.DateField()
    is_relative = pw.BooleanField()

    class Meta:
        database = db  # This model uses the "people.db" database


class Pet(pw.Model):
    owner = pw.ForeignKeyField(Person, related_name='pets')
    name = pw.CharField()
    animal_type = pw.CharField()

    class Meta:
        database = db


db.connect()
db.create_tables([Person, Pet])
```

存储数据

```
from datetime import date

uncle_bob = Person(name='Bob', birthday=date(1960, 1, 15), is_relative=True)
uncle_bob.save()  # 返回修改的行数
```

或者通过Model的 `create()` 方法来创建Model实例：

```
grandma = Person.create(name='Grandma', birthday=date(1935, 3, 1), is_relative=True)
herb = Person.create(name='Herb', birthday=date(1950, 5, 5), is_relative=False)
```

为了更新某一行数据，修改Model实例并调用 `save()` 方法来持久化修改：

```
grandma.name = 'Grandma L.'
grandma.save()
```

现在我们在数据库中存储了3个人，让我们给他们一些pets。

```
bob_kitty = Pet.create(owner=uncle_bob, name='Kitty', animal_type='cat')
herb_fido = Pet.create(owner=herb, name='Fido', animal_type='dog')
herb_mittens = Pet.create(owner=herb, name='Mittens', animal_type='cat')
herb_mittens_jr = Pet.create(owner=herb, name='Mittens Jr', animal_type='cat')
```

后来，Mittens生病死了，我们需要在数据库中删除它：

```
herb_mittens.delete_instance()  # 返回从数据库中删除的行数
```

Bob叔叔收养了Herb的Fido：

```
herb_fido.owner = uncle_bob
herb_fido.save()
bob_fido = herb_fido
```

检索数据

数据库的强项就是其允许我们通过查询检索数据。关系数据库特别适合检索查询。

获取单条记录

使用 `SelectQuery.get()` 从数据库中获取单条记录：

```
grandma_query = Person.select().where(Person.name == 'Grandma L.')
print(grandma_query.sql())
grandma = grandma_query.get()
type(grandma)
```

我们还可以使用等价的简写方式 `Model.get()` ：

```
grandma = Person.get(Person.name == 'Grandma L.')
```

获取多条记录

列举数据中的存储的人：

```
for person in Person.select():
    print(person.name, person.is_relative)
```

列出所有的猫和它们的主人：

```
query = Pet.select().where(Pet.animal_type == 'cat')
for pet in query:
    print(pet.name, pet.owner.name)
```

上面的查询中有一个大的问题：当我们访问 `pet.owner.name` 时并没有在原来的查询中选择查询它， Peewee将执行一次额外的查询来获取宠物的主人。

我们可以通过 `join` 同时查询宠物和人：

```
query = (Pet
         .select(Pet, Person)
         .join(Person)
         .where(Pet.animal_type == 'cat'))

for pet in query:
    print(pet.name, pet.owner.name)
```

获取Bob的所有宠物：

```
for pet in Pet.select().join(Person).where(Person.name == 'Bob'):
    print(pet.name)

for pet in Pet.select().where(Pet.owner == uncle_bob):
    print(pet.name)
```

通过 `order_by()` 语句确保上面查询到的宠物按照字母表序进行排序：

```
for pet in Pet.select().where(Pet.owner == uncle_bob).order_by(Pet.name):
    print(pet.name)
```

列出所有人，根据年龄，从小到大：

```
for person in Person.select().order_by(Person.birthday.desc()):
    print(person.name, person.birthday)
```

列出所有人以及他们宠物的一些信息：

```
for person in Person.select():
    print(person.name, person.pets.count(), 'pets')
    for pet in person.pets:
        print('    ', pet.name, pet.animal_type)
```

上面的代码又会导致额外查询问题，我们通过 `JOIN` 查询和聚合查询记录来避免这个问题：

```
subquery = Pet.select(pw.fn.COUNT(Pet.id)).where(Pet.owner == Person.id)
query = (Person
         .select(Person, Pet, subquery.alias('pet_count'))
         .join(Pet, pw.JOIN.LEFT_OUTER)
         .order_by(Person.name))
print(query.sql())

for person in query.aggregate_rows():
    print(person.name, person.pet_count, 'pets')
    for pet in person.pets:
        print('    ', pet.name, pet.animal_type)
```

尽管我们创建了一个子查询，但是只有一个查询真正地被执行了。

最后，让我们做一个复杂点的。获取所有人，他们的生日符合下面的要求之一：

- 晚于1940（grandma）
- 早于1960（bob）

```
d1940 = date(1940, 1, 1)
d1960 = date(1960, 1, 1)

query = (Person
         .select()
         .where((Person.birthday < d1940) | (Person.birthday > d1960)))

for person in query:
    print(person.name, person.birthday)
```

现在做相反的查询，某人的生日在1940和1960之间：

```
query = (Person
         .select()
         .where((Person.birthday > d1940) & (Person.birthday < d1960)))
for person in query:
    print(person.name, person.birthday)
```

最后一个查询。这次将使用一个SQL函数来查找名字是以大写或小写的 G 开头的人：

```
expression = (pw.fn.Lower(pw.fn.Substr(Person.name, 1, 1)) == 'g')
for person in Person.select().where(expression):
    print(person.name)
```

想了解更多的sqlite3核心函数，点击这里查看。数据库操作完了之后，关闭连接：

```
db.close()
```

上面的代码示例展示的仅仅是一些基础，你可以执行自己想要的更加复杂的查询。

所有其他的SQL语句也是支持的，比如：

- group_by()
- having()
- limit() and offset()

在Flask中使用Peewee

定义模型

在 hello.py 中定义Role和User模型：

```
import peewee as pw


db = pw.SqliteDatabase("flaskr.db")


class BaseModel(pw.Model):
    class Meta:
        database = db


class Role(BaseModel):
    name = pw.CharField(64, unique=True)

    def __repr__(self):
        return '<Role %r>' % self.name

    class Meta:
```

```
        db_table = 'roles'


class User(BaseModel):
    username = pw.CharField(64, unique=True, index=True)

    def __repr__(self):
        return '<User %r>' % self.username

    class Meta:
        db_table = 'users'
```

下表列出了一些可用的字段类型以及对应的数据库字段类型：

字段类型	Sqlite	Postgresql	MySQL
CharField	varchar	varchar	varchar
FixedCharField	char	char	char
TextField	text	text	longtext
DateTimeField	datetime	timestamp	datetime
IntegerField	integer	integer	integer
BooleanField	integer	boolean	bool
FloatField	real	real	real
DoubleField	real	double precision	double precision
BigIntegerField	integer	bigint	bigint
SmallIntegerField	integer	smallint	smallint
DecimalField	decimal	numeric	numeric
PrimaryKeyField	integer	serial	integer
ForeignKeyField	integer	integer	integer
DateField	date	date	date
TimeField	time	time	time
TimestampField	integer	integer	integer
BlobField	blob	bytea	blob
UUIDField	text	uuid	varchar(40)
BareField	untyped	not supported	not supported

字段初始化参数及默认值：

- `null = False` – 布尔值，是否允许储存 `null` 值。
- `index = False` – 布尔值，是否为这一列添加索引。
- `unique = False` – 布尔值，是否为这一列添加唯一性索引。另外可参考如何添加复合索引。
- `verbose_name = None` – 字符串，为模型字段添加用户友好的自定义标注。
- `help_text = None` – 字符串，为此字段添加帮助文本信息。
- `db_column = None` – 字符串，用于底层存储的列名，有助于遗留数据库的兼容性。
- `default = None` – 任意类型，用于初始化的默认值，如果是可调用对象，则调用生成相应的值。
- `choices = None` – 可迭代的二元元组，对应 `value` 和 `display` 。
- `primary_key = False` – 布尔值，是否时此数据表的主键。
- `sequence = None` – 字符串，序列名字，如果后端数据库支持的话。
- `constraints = None` - 一个或多个约束条件列表，例如 `[Check('price > 0')]` 。
- `schema = None` – 字符串，schema的可选名字，如果后端数据库支持的话。

一些列类型接收特定的参数：

字段类型	特殊参数
CharField	max_length
FixedCharField	max_length
DateTimeField	formats
DateField	formats
TimeField	formats
TimestampField	resolution , utc
DecimalField	max_digits , decimal_places , auto_round , rounding
ForeignKeyField	rel_model , related_name , to_field , on_delete , on_update , extra
BareField	coerce

虽然没有强制要求，但这两个模型都定义了 __repr()__ 方法，返回一个具有可读性的字符串表示模型，可在调试和测试时使用。

关系

关系型数据库使用关系把不同表中的行联系起来。

角色到用户是**一对多**关系，因为一个角色可属于多个用户，而每个用户都只能有一个角色。

```
class User(BaseModel):
    # ...
    role = pw.ForeignKeyField(Role, related_name='users', null=True)
```

数据库操作

学习如何使用模型的最好方法是在 Python shell 中实际操作。

创建表

首先，要根据模型类来创建数据库。

定义一个函数 create_tables ，用来创建数据表。

```
def create_tables():
    db.connect()
    db.create_tables([Role, User])
```

接下来在开启Flask shell 并进行实际操作：

```
(flask_env3) $ export FLASK_APP=hello.py
(flask_env3) $ export FLASK_DEBUG=1
(flask_env3) $ flask shell
```

```
...
App: hello [debug]
Instance: ...
```

```
>>> from hello import create_tables
>>> create_tables()
```

插入行

```
>>> from hello import Role, User
>>> admin_role = Role(name='Admin')
>>> mod_role = Role(name='Moderator')
>>> user_role = Role(name='User')
>>> user_john = User(username='john', role=admin_role)
>>> user_susan = User(username='susan', role=user_role)
>>> user_david = User(username='david', role=user_role)
>>> print(admin_role.id)
None
>>> admin_role.save()
>>> mod_role.save()
>>> user_role.save()
>>> user_john.save()
>>> user_susan.save()
>>> user_david.save()
```

再查看 id 属性，现在已经赋值了：

```
>>> print(admin_role.id)
1
>>> print(mod_role.id)
2
>>> print(user_role.id)
3
```

修改行

```
>>> admin_role.name = 'Administrator'
>>> admin_role.save()
```

删除行

```
>>> mod_role.delete_instance()
1
```

查询行

```
>>> list(Role.select())
[<Role 'Administrator'>, <Role 'User'>]
>>> list(User.select())
[<User 'john'>, <User 'susan'>, <User 'david'>]
```

查找角色为 "User" 的所有用户：

```
>>> list(User.select().where(User.role==user_role))
[<User 'susan'>, <User 'david'>]
```

查看Peewee为查询生成的原生SQL查询语句：

```
>>> print(User.select().where(User.role==user_role).sql())
('SELECT "t1"."id", "t1"."username", "t1"."role_id" FROM "users" AS t1 WHERE ("t1"."role_id" =
```

如果你退出了 shell 会话，前面这些例子中创建的对象就不会以 Python 对象的形式存在，而是作为各自数据库表中的行。如果你打开了一个新的 shell 会话，就要从数据库中读取行，再重新创建 Python 对象。

```
>>> user_role = Role.select().where(Role.name=='User').get()
```

关系和查询的处理方式类似。下面这个例子分别从关系的两端查询角色和用户之间的一对多关系：

```
>>> users = user_role.users
>>> list(users)
[<User 'susan'>, <User 'david'>]
>>> users[0].role
<Role 'User'>
```

执行 `git checkout 4a` 签出程序的这个版本。

在视图函数中操作数据库

前一节介绍的数据库操作可以直接在视图函数中进行。

下面的代码示例展示了首页路由的新版本，把用户输入的名字写入数据库。

```
@app.route('/', methods=['GET', 'POST'])
def index():
    form = NameForm()
    if form.validate_on_submit():
        user = User.select().where(User.username == form.name.data).first()
        if user is None:
            user = User(username=form.name.data)
            user.save()
            session['known'] = False
        else:
            session['known'] = True
        session['name'] = form.name.data
        form.name.data = ''
        return redirect(url_for('index'))
    return render_template('index.html', form=form,
                           name=session.get('name'),
                           known=session.get('known', False))
```

变量 known 被写入用户会话中，因此重定向之后，可以把数据传给模板，用来显示自定义的欢迎消息。

对应的模板新版本如下所示，这个模板使用know参数在欢迎消息中加入了第二行，从而对已知用户和新用户显示不同的内容。

```
{# templates/index.html #}

{% extends "base.html" %}
{% import "bootstrap/wtf.html" as wtf %}

{% block page_content %}
    <div class="page-header">
        <h1>Hello, {% if name %}{{ name }}{% else %}Stranger{% endif %}!</h1>
        {% if not known %}
            <p>Pleased to meet you!</p>
        {% else %}
            <p>Happy to see you again!</p>
        {% endif %}
    </div>
    {{ wtf.quick_form(form) }}
{% endblock %}
```

执行 `git checkout 4b` 签出程序的这个版本。

数据库迁移

目前，Peewee尚未支持自动数据库迁移，但是可以使用其提供的 `playhouse.migrate` 模块来创建简单的迁移脚本。

> Peewee's migrations do not handle introspection and database "versioning". Rather, peewee provides a number of helper functions for generating and running schema-altering statements. This engine provides the basis on which a more sophisticated tool could some day be built.
>
> Migrations can be written as simple python scripts and executed from the command-line. Since the migrations only depend on your applications Database object, it should be easy to manage changing your model definitions and maintaining a set of migration scripts without introducing dependencies.

下面是一个例子：

```
from playhouse.migrate import SqliteMigrator, migrate
import peewee as pw


my_db = pw.SqliteDatabase('my_database.db')
migrator = SqliteMigrator(my_db)

title_field = pw.CharField(default='')
status_field = pw.IntegerField(null=True)

# run migrations inside a transaction
with my_db.transaction():
    migrate(
        migrator.add_column('some_table', 'title', title_field),
        migrator.add_column('some_table', 'status', status_field),
        migrator.drop_column('some_table', 'old_column'),
    )
```

支持的操作：

- 为已有模型添加新字段

```
# Create your field instances. For non-null fields you must specify a
# default value.
pubdate_field = DateTimeField(null=True)
comment_field = TextField(default='')

# Run the migration, specifying the database table, field name and field.
migrate(
    migrator.add_column('comment_tbl', 'pub_date', pubdate_field),
    migrator.add_column('comment_tbl', 'comment', comment_field),
)
```

- 重命名字段

```
# Specify the table, original name of the column, and its new name.
migrate(
    migrator.rename_column('story', 'pub_date', 'publish_date'),
    migrator.rename_column('story', 'mod_date', 'modified_date'),
)
```

- 删除字段

```
migrate(
    migrator.drop_column('story', 'some_old_field'),
)
```

- 设置字段 nullable 或者 not nullable

```
# Note that when making a field not null that field must not have any
# NULL values present.
migrate(
    # Make `pub_date` allow NULL values.
    migrator.drop_not_null('story', 'pub_date'),

    # Prevent `modified_date` from containing NULL values.
    migrator.add_not_null('story', 'modified_date'),
)
```

- 添加索引

```
# Specify the table, column names, and whether the index should be
# UNIQUE or not.
migrate(
```

```
    # Create an index on the `pub_date` column.
    migrator.add_index('story', ('pub_date',), False),

    # Create a multi-column index on the `pub_date` and `status` fields.
    migrator.add_index('story', ('pub_date', 'status'), False),

    # Create a unique index on the category and title fields.
    migrator.add_index('story', ('category_id', 'title'), True),
)
```

日期：2017-07-14

作者：田宇伟