给Django用户的SQLAlchemy介绍

11-14 15:43
1.08k

SQLAlchemy是什么？

SQLAlchemy的官网上写着它的介绍文字:

```
SQLAlchemy is the Python SQL toolkit and Object Relational Mapper that gives
application developers the full power and flexibility of SQL.
```

SQLAlchemy 是一个非常强大的ORM和数据库工具，但是它庞大的文档和复杂的功能总是让很 多人望而生畏。而Django的ORM相对来说就让很多人觉得简单实用。

事实上，SQLAlchemy其实也没有那么复杂，光使用它一些比较高级的功能其实并没有比 使用Django ORM复杂多少，而它丰富的功能则能让你在遇到更复杂的问题时处理起来得心应手。

写作本文的主要目的在于：

- 通过对比SQLAlchemy ORM和Django ORM的主要使用方法， 尽量简单直观的让Django用户能够快速了解和上手SQLAlchemy这款强大的工具。
- 不牵扯到SQLAlchemy具体的技术细节，包括Engine连接池、Session的具体工作原理等等

SQLAlchemy相对于Django内建的ORM来说，有几处非常明显的优点：

- 可独立使用，任何使用Python的项目都可以用它来操作数据库
- 和直接使用原始的DBAPI相比，提供了非常丰富的特性：连接池、auto-map等等
- 提供了更底层的SQL抽象语言，能用原始sql解决的问题基本上都可以用SQLAlchemy解决

接下来我们针对日常的数据库操作来对比一下Django ORM和SQLAlchemy。

文中使用的 *SQLAlchemy* 版本为 *0.9.8*

Django VS SQLAlchemy

建立数据表

首先，我们需要先建立几个表。

Django

在Django中，如果要建表，就是在models.py中定义你的数据类型：

```
from django.db import models
```

```
class Game(models.Model):
    ... ...

class GameCompany(models.Model):
    ... ...
```

因为文章主要面向有经验的Django用户，所以此处不写出详细的定义代码。定义Model以后 我们还需要在settings.py中DATABASES处设置需要连接的数据库地址。最后，使用syncdb来 完成数据库表的创建。

SQLAlchemy

在SQLAlchemy中，定义表结构的过程和Django类似：

```
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Column, Integer, String, ForeignKey, Date
from sqlalchemy.orm import relationship, backref

Base = declarative_base()

# 定义表结构
class GameCompany(Base):
    __tablename__ = 'game_company'

    id = Column(Integer, primary_key=True)
    name = Column(String(200), nullable=False)
    country = Column(String(50))


class Game(Base):
    __tablename__ = 'game'

    id = Column(Integer, primary_key=True)
    company_id = Column(Integer, ForeignKey('game_company.id'), index=True)
    category = Column(String(10))
    name = Column(String(200), nullable=False)
    release_date = Column(Date)

    # 和Django不同，外键需要显式定义，具体好坏见仁见智
    # 此处的relation可以为lazy加载外键内容时提供一些可配置的选项
    company = relationship('GameCompany', backref=backref('games'))


# 此处定义要使用的数据库
engine = create_engine('mysql://root:root@localhost:5379/sqlalchemy_tutorial?c
# 调用create_all来创建表结构，已经存在的表将被忽略
Base.metadata.create_all(engine)
```

插入一些数据

接下来，我们往表中插入一些数据

Django

Django中比较常用的插入数据方法就是使用 `.save()` 了。

```
nintendo = GameCompany(name="nintendo", country="Japan")
nintendo.save()

game1 = Game(
    company=nintendo,
    category="ACT",
    name="Super Mario Bros",
    release_date='1985-10-18')
game1.save()

# 或者使用create
Game.objects.create(... ...)
```

SQLAlchemy

在SQLAlchemy ORM中，有一个非常关键的对象 `session` ，所有对于数据的操作都是 通过session来进行的，所以要插入数据之前，我们得先初始化一个session：

```
from sqlalchemy.orm import sessionmaker
Session = sessionmaker(bind=engine)
session = Session()
```

之后插入数据的方法也和Django比较相似：

```
# 添加数据
nintendo = GameCompany(name="Nintendo", country="Japan")
capcom = GameCompany(name="Capcom", country="Japan")
game1 = Game(
    company=nintendo,
    category="ACT",
    name="Super Mario Bros",
    release_date='1985-10-18'
)
game2 = Game(
    company=capcom,
    category="ACT",
    name="Devil May Cry 3: Dante's Awakening",
    release_date="2005-03-01",
)
game3 = Game(
    company=nintendo,
```

```
    category="RPG",
    name="Mario & Luigi: Dream Team",
    release_date="2013-08-11",
)

# 使用add_all来让这些objects和session产生关系
session.add_all([nintendo, capcom, game1, game2])
# 在没有开启autocommit的模式下，不要忘了调用commit来让数据写到数据库中
session.commit()
```

除了commit之外，session还有rollback()等方法，你可以把session对象简单看成是一次transaction，所以当你对内容进行修改时，需要调用 `session.commit()` 来提交这些修改。

去文档可以了解更多session相关内容：http://docs.sqlalchemy.org/en/rel_0_9/orm/session.html

常用操作

简单查询

批量查询

```
# -- Django --
Game.objects.filter(category="RPG")

# -- SQLAlchemy --
# 使用filter_by是和django ORM比较接近的方式
session.query(Game).filter_by(category="RPG")
session.query(Game).filter(Game.category == "RPG")
```

查询单个对象

```
# -- Django --
Game.objects.get(name="Super Mario Bros")

# -- SQLAlchemy --
session.query(Game).filter_by(name="Super Mario Bros").one()
# `get_objects_or_None()`
session.query(Game).filter_by(name="Super Mario Bros").scalar()
```

Django中得各种 > 、< 都是使用在字段名称后面追加 "__gt"、"__lt" 来实现的，在SQLAlchemy 中这样的查询还要更直观一些

```
# -- Django --
Game.objects.filter(release_date__gte='1999-01-01')
# 取反
Game.objects.exclude(release_date__gte='1999-01-01')

# -- SQLAlchemy --
session.query(Game).filter(Game.release_date >= '1999-01-01').count()
```

```
# 取反使用 ~ 运算符
session.query(Game).filter(~Game.release_date >= '1999-01-01').count()
```

通过外键组合查询

```
# -- Django --
Game.objecs.filter(company__name="Nintendo")

# -- SQLAlchemy --
session.query(Game).join(GameCompany).filter(GameCompany.name == "Nintendo")
```

多条件或查询

```
# -- Django --
from django.db.models import Q
Game.objects.filter(Q(category="RPG") | Q(category="ACT"))

# -- SQLAlchemy --
from sqlalchemy import or_
session.query(Game).filter(or_(Game.category == "RPG", Game.category == "ACT")
session.query(Game).filter((Game.category == "RPG") | (Game.category == "ACT")
```

in查询

```
# -- Django --
Game.objects.filter(category__in=["GAL", "ACT"])

# -- SQLAlchemy --
session.query(Game).filter(Game.category.in_(["GAL", "ACT"]))
```

like查询

```
# -- Django --
Game.objects.filter(name__contains="Mario")

# -- SQLAlchemy --
session.query(Game.name.contains('Mario'))
```

统计个数

简单统计总数

```
# -- Django --
Game.objects.filter(category="RPG").count()

# -- SQLAlchemy --
session.query(Game).filter_by(category="RPG").count()
```

分组统计个数

```
# -- Django --
from django.db.models import Count
Game.objects.values_list('category').annotate(Count('pk')).order_by()

# -- SQLAlchemy --
from sqlalchemy import func
session.query(Game.category, func.count(Game.category)).group_by(Game.category
```

结果排序

对查询结果进行排序

```
# -- Django --
Game.objects.all().order_by('release_date')
Game.objects.all().order_by('-release_date')
# 多字段排序
Game.objects.all().order_by('-release_date', 'category')

# -- SQLAlchemy --
session.query(Game).order_by(Game.release_date)
session.query(Game).order_by(Game.release_date.desc())
# 多字段排序
session.query(Game).order_by(Game.release_date.desc(), Game.category)
```

修改数据

```
# -- Django --
game =  Game.objects.get(pk=1)
game.name = 'Super Mario Brothers'
game.save()

# -- SQLAlchemy --
game = session.query(Game).get(1)
game.name = 'Super Mario Brothers'
session.commit()
```

批量修改

```
# -- Django --
Game.objects.filter(category="RPG").update(category="ARPG")

# -- SQLAlchemy --
session.query(Game).filter_by(category="RPG").update({"category": "ARPG"})
```

批量删除

```
# -- Django --
Game.objects.filter(category="ARPG").delete()

# -- SQLAlchemy --
session.query(Game).filter_by(category="ARPG").delete()
```

SQLAlchemy其他一些值得关注的功能

上面简单列了一些SQLAlchemy ORM和Django ORM的使用方法对比，SQLAlchemy同时还提供了一些 其他非常有用的功能。

Automap

假如你有一个Django项目，通过ORM创建了一大堆Model。这时来了一个新项目，需要操作 这些表，应该怎么办？拷贝这些Models？使用原始的DB-API加上sql来操作？

其实使用SQLAlchemy的Automap可以让你的工作变得非常的方便，你只要在新项目连接到旧数据库，然后 稍微配置一下Automap，就可以使用SQLAlchemy的ORM操作那些通过别的系统创建的表了。

就像这样：

```
from sqlalchemy.ext.automap import automap_base
from sqlalchemy.orm import Session
from sqlalchemy import create_engine

Base = automap_base()
engine = create_engine("sqlite:///mydatabase.db")
Base.prepare(engine, reflect=True)

# user和address就是表明，通过这样的语句就可以把他们分别映射到User和Address类
User = Base.classes.user
Address = Base.classes.address
```

更多信息可以参考详细文档：http://docs.sqlalchemy.org/en/rel_0_9/orm/extensions/automap.html

Alembic

Django有south可以方便做表结构修改？SQLAlchemy当然也可以，甚至比south更为强大。 自动migrate？手动migrate？统统不是问题。

更多信息可参考文档：http://alembic.readthedocs.org/en/latest/index.html