##### SQLAlchemy 简介#####

SQLAlchemy 分为两个部分，一共用于 ORM 的对象映射，另外一个是核心的 SQL expression 。第一个很好理解，纯粹的ORM，后面这个不是 ORM，而是DBAPI的封装，当然也提供了很多方法，避免了直接写sql，而是通过一些sql表达式。使用 SQLAlchemy 则可以分为三种方式。

* 使用 sql expression ，通过 SQLAlchemy 的方法写sql表达式，简介的写sql
* 使用 raw sql， 直接书写 sql
* 使用 ORM 避开直接书写 sql

**Python学习总结笔记（10）-- MySQL数据库操作之SQLAlchemy使用总结**

2016年12月13日 22:47:02

阅读数：5382

SQLAlchemy是一个著名的ORM框架，使用ORM操作数据库，不用去关注SQL语句本身，这样可以提高开发的效率。同时使用ORM框架虽然可以减少代码编写的消耗，但是可能也会执行很多冗余的数据库操作，降低程序的运行效率。不过总的来说，合理利用ORM框架与数据库交互还是一个比较不错的选择。

**0x01 安装SQLAlchemy**

在Python中，有很多ORM框架，SQLAlchemy是其中非常著名的一个框架。可以通过pip来完成安装：

pip install sqlalchemy

* 1

SQLAlchemy提供对很多数据库的支持。下面我们就以MySQL为例，学习下SQLAlchemy的基本用法。

**0x02 连接数据库**

#!/usr/bin/env python

#coding:utf-8

\_\_author\_\_ = 'kikay'

#导入相关库

from sqlalchemy.orm import sessionmaker

from sqlalchemy import create\_engine

#MySQL数据库连接字符串

CONSTR='mysql+mysqlconnector://root:root@localhost:3306/test?charset=utf8'

#初始化数据库连接对象

engine=create\_engine(CONSTR,echo=True)

db\_session=sessionmaker(bind=engine)

session=db\_session()

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16

上面CONSTR定义了数据库的连接信息，其中mysql+mysqlconnector指明了调用的是mysql-connector模块完成数据库连接，需要注意charset指定了连接的字符集，可以缺省，但是这个参数很重要，设置不当可能导致查询结果出现乱码。

sessionmaker()生成数据库会话类。我们可以把session当成的一个数据库连接对象。SQLAlchemy维护了一个数据库连接池（默认为5个连接对象），因此初始化一个会话开销不会太大。

完成了session对象实例化后，就可以执行SQL语句了。

**0x03 execute方式**

#获取当前MySQL中的全部数据库

dbs=session.execute('show databases;').fetchall()

for db in dbs:

print db

#切换当前数据库

session.execute('use test;')

row1=session.execute('select \* from user where Id>1;').first()

#利用占位符的方式

row2= session.execute('select \* from user where Id>:id',{'id':1}).first()

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9

上面的row1和row2实现的效果是等价的，其中row2使用的是占位符的方式。其实这种方式就是直接执行SQL语句，和前面讲的mysql-connector模板方式类似，不符合ORM思想，这里就不继续探讨了。

**0x04 ORM方式插入**

ORM方式很重要的一点就是实现了数据表与类实例的对应关系。这里我们首先引入一个模块：

from sqlalchemy.ext.declarative import declarative\_base

* 1

定义实体类的基类：

Base=declarative\_base()

* 1

然后实例化我们数据库中的user数据表，完整代码如下：

#!/usr/bin/env python

#coding:utf-8

\_\_author\_\_ = 'kikay'

#导入相关库

from sqlalchemy.orm import sessionmaker

from sqlalchemy import create\_engine

from sqlalchemy.ext.declarative import declarative\_base

from sqlalchemy import Column,String,Integer,PrimaryKeyConstraint

#MySQL数据库连接字符串

CONSTR='mysql+mysqlconnector://root:root@localhost:3306/test?charset=utf8'

#定义基类

Base=declarative\_base()

class User(Base):

#显示声明关联的数据表名称

\_\_tablename\_\_='user'

#表的结构

#主键Id

id=Column(Integer,name='Id',primary\_key=True)

#name：非空

name=Column(String(20),nullable=False)

age=Column(Integer,nullable=False)

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27

添加数据：

#初始化数据库连接对象

engine=create\_engine(CONSTR,echo=True)

db\_session=sessionmaker(bind=engine)

session=db\_session()

user=User(name='kikay',age=20)

#添加

session.add(user)

#提交

session.commit()

#关闭

session.close()

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12

上面演示了一个完整的添加记录过程。对于SQLAlchemy而言，用户并没有与SQL语句直接打交道。

SQLAlchemy添加批量记录：

#批量添加

for i in xrange(100):

user=User(name='Tom',age=20+i)

#添加

session.add(user)

#提交

session.commit()

#关闭

session.close()

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9

**0x05 ORM方式查询**

下面完整定义两个数据表：

数据表user:

| **序号** | **字段名称** | **数据类型** |
| --- | --- | --- |
| 1 | Id | Int 自增主键 |
| 2 | name | nvarchar(20) 非空 |
| 3 | age | Int 非空 |

数据表job:

| **序号** | **字段名称** | **数据类型** |
| --- | --- | --- |
| 1 | Id | Int 自增主键 |
| 2 | jobname | nvarchar(200) |
| 3 | userid | 外键(user.id) |

对应的实体类如下：

#!/usr/bin/env python

#coding:utf-8

\_\_author\_\_ = 'kikay'

#导入相关库

from sqlalchemy.orm import sessionmaker,relationship

from sqlalchemy import create\_engine

from sqlalchemy.ext.declarative import declarative\_base

from sqlalchemy import Column,String,Integer

from sqlalchemy import text,or\_,not\_,update

from sqlalchemy.dialects.mysql import INTEGER

from sqlalchemy import ForeignKey

#MySQL数据库连接字符串

CONSTR='mysql+mysqlconnector://root:root@localhost:3306/test?charset=utf8'

#定义基类

Base=declarative\_base()

class User(Base):

#显示声明关联的数据表名称

\_\_tablename\_\_='user'

#修改数据表配置

\_\_table\_args\_\_={

'mysql\_engine':'InnoDB',

'mysql\_charset':'utf8'

}

#主键Id

id=Column(INTEGER(unsigned=True),name='Id',primary\_key=True)

#name：非空

name=Column(String(20),nullable=False)

#age:非空

age=Column(Integer,nullable=False)

#外键对象

#jobs=relationship('job')

class Job(Base):

\_\_tablename\_\_='job'

id=Column(INTEGER(unsigned=True),name='Id',primary\_key=True)

jobname=Column(String(200))

#声明外键

userid=Column(INTEGER,ForeignKey('user.Id',

ndelete='CASCADE',onupdate='CASCADE'))

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46

需要强调的是，在声明外键时，加上了ondelete=’CASCADE’,onupdate=’CASCADE’，这是因为删除 user 表的数据，可能会导致 job的外键不指向一个真实存在的记录，在MySQL数据库中默认禁止这样的操作，在InnoDB模式下， 允许指定 ON DELETE 为 CASCADE 和 SET NULL，前者会删除 job 中无效的记录，后者会将这些记录的外键设为 NULL。   
除了删除，还有可能更改主键，这也会导致 job 的外键失效。于是相应的就有 ON UPDATE 了。其中 CASCADE 变成了更新相应的外键，而不是删除。

下面讲下基本的查询、更新、删除的使用方法：

#查询对象

query1=session.query(User)

query2=session.query(Job)

* 1
* 2
* 3

查询全部对象（select \* …）：

for u in query1.all():

print 'Id:\t',u.id,'\tname:\t',u.name,'\tage:\t',u.age

结果：

Id: 80 name: Tom age: 30

Id: 81 name: Tom3 age: 29

Id: 82 name: Andy age: 25

Id: 88 name: 小花 age: 19

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9

取第1条记录：

u=query1.first()

print 'Id:\t',u.id,'\tname:\t',u.name,'\tage:\t',u.age

结果：

Id: 80 name: Tom age: 30

* 1
* 2
* 3
* 4
* 5
* 6

条件过滤：

for u in query1.filter(User.id>=82).all():

print 'Id:\t',u.id,'\tname:\t',u.name,'\tage:\t',u.age

* 1
* 2

等价于（sql语句中的where部分）：

for u in query1.filter('id>=82').all():

print 'Id:\t',u.id,'\tname:\t',u.name,'\tage:\t',u.age

* 1
* 2

根据主键值获取对应的记录：

u=query1.get(88)

print 'Id:\t',u.id,'\tname:\t',u.name,'\tage:\t',u.age

结果：

Id: 88 name: 小花 age: 19

* 1
* 2
* 3
* 4
* 5
* 6

设置排序：

for u in query1.order\_by(User.name.desc()).all():

print 'Id:\t',u.id,'\tname:\t',u.name,'\tage:\t',u.age

结果：

Id: 88 name: 小花 age: 19

Id: 81 name: Tom3 age: 29

Id: 80 name: Tom age: 30

Id: 82 name: Andy age: 25

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9

限制返回的条数：

for u in query1.limit(2).all():

print 'Id:\t',u.id,'\tname:\t',u.name,'\tage:\t',u.age

结果：

Id: 80 name: Tom age: 30

Id: 81 name: Tom3 age: 29

* 1
* 2
* 3
* 4
* 5
* 6
* 7

设置查询的偏移量：

for u in query1.offset(3).all():

print 'Id:\t',u.id,'\tname:\t',u.name,'\tage:\t',u.age

结果：

Id: 88 name: 小花 age: 19

* 1
* 2
* 3
* 4
* 5
* 6

结合偏移量、限制条数以及排序功能，可以方便的实现分页查询：

#每页2条记录，打印第2页的内容

for u in query1.order\_by(User.id).offset(2).limit(2):

print 'Id:\t',u.id,'\tname:\t',u.name,'\tage:\t',u.age

结果：

Id: 82 name: Andy age: 25

Id: 88 name: 小花 age: 19

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8

返回第一行的第一个字段：

print session.query(User.name,User.age).filter(User.id==88).scalar()

* 1

条件查询and、or、in、not in、not

#and

print 'And:'

for u in query1.filter(User.id>80,User.age>25).all():

print 'Id:\t',u.id,'\tname:\t',u.name,'\tage:\t',u.age

#or

print 'Or:'

for u in query1.filter(or\_(User.id>88,User.age>29)).all():

print 'Id:\t',u.id,'\tname:\t',u.name,'\tage:\t',u.age

#in

print 'in:'

for u in query1.filter(User.id.in\_((79,81,90))).all():

print 'Id:\t',u.id,'\tname:\t',u.name,'\tage:\t',u.age

#not in

print 'not in:'

for u in query1.filter(User.id.notin\_((81,82,88))).all():

print 'Id:\t',u.id,'\tname:\t',u.name,'\tage:\t',u.age

#not

print 'not:'

for u in query1.filter(not\_(or\_(User.name=='Tom',User.name==u'小花'))).all():

print 'Id:\t',u.id,'\tname:\t',u.name,'\tage:\t',u.age

结果：

And:

Id: 81 name: Tom3 age: 29

Or:

Id: 80 name: Tom age: 30

in:

Id: 81 name: Tom3 age: 29

not in:

Id: 80 name: Tom age: 30

not:

Id: 81 name: Tom3 age: 29

Id: 82 name: Andy age: 25

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38

返回记录的总条数：

print query1.filter(User.id>=81).count()

* 1

模糊查询：

for u in query1.filter(User.name.like('%t%')).all():

print 'Id:\t',u.id,'\tname:\t',u.name,'\tage:\t',u.age

结果：

Id: 80 name: Tom age: 30

Id: 81 name: Tom3 age: 29

* 1
* 2
* 3
* 4
* 5
* 6
* 7

**0x06 ORM方式更新与删除**

更新：

print 'old name:',query1.filter(User.id==80).one().name

query1.filter(User.id==80).update({User.name:'Tom2'})

session.commit()

print 'new name:',query1.filter(User.id==80).one().name

结果：

old name: Tom

new name: Tom2

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9

删除：

query1.filter(User.id<=80).delete()

* 1

需要注意的是，update和delete在做批量操作的时候（使用 where…in(…)）操作，需要指定synchronize\_session的值。

query1.filter(User.id.in\_((1,2,3))).update({User.name:'Test'})

query1.filter(User.id.in\_((1,2,3))).delete()

* 1
* 2

报错：

sqlalchemy.exc.InvalidRequestError: Could not evaluate current criteria in Python. Specify 'fetch' or False for the synchronize\_session parameter.

* 1

修改如下：

query1.filter(User.id.in\_((1,2,3))).update({User.name:'Test'},synchronize\_session=False)

query1.filter(User.id.in\_((1,2,3))).delete(synchronize\_session=False)

* 1
* 2

**0x07 后记**

上面简单介绍了下SQLAlchemy的使用方法，主要还是关注在增删改查方面，其实还有很多知识点没有讲到，有兴趣的可以自己接着研究下。ORM框架使用起来很方便，但是要想正确理解ORM，前提还是要对关系型数据库的基本原理有所了解。