**SQLAlchemy 学习笔记**

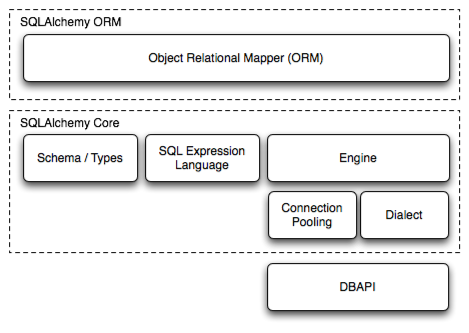
2016年03月04日 01:06:12

阅读数：3890

**一、写在前面**

这篇文章主要介绍了Python的SQLAlchemy框架使用入门,SQLAlchemy框架是Python中用来操作数据库的ORM框架之一,学习过程中主要参考网上现有资料,整理成笔记以便后续自己查阅。   
如果转载，请保留作者信息。   
邮箱地址：jpzhang.ht@gmail.com   
SQLAlchemy: <http://www.sqlalchemy.org/>   
中文参考：<https://github.com/lzjun567/note/blob/master/note/python/sqlalchemy.md>

**二、简介**

SQLAlchemy的是Python的SQL工具包和对象关系映射，给应用程序开发者提供SQL的强大功能和灵活性。它提供了一套完整的企业级的持久性模式，专为高效率和高性能的数据库访问，改编成简单的Python的领域语言。   
SQLAlchemy是Python界的ORM（Object Relational Mapper）框架，它两个主要的组件： SQLAlchemy ORM 和 SQLAlchemy Core 。   


**二、安装**

pip install SQLAlchemy

检查安装是否成功:

>>> import sqlalchemy

>>> sqlalchemy.\_\_version\_\_

>>> '1.0.12'

* 1
* 2
* 3
* 4
* 5

**三、连接**

连接MySQL数据库(需要MySQLdb支持)：

from sqlalchemy import create\_engine

DB\_CONNECT\_STRING = 'mysql+mysqldb://root:password@localhost/test?charset=utf8'

engine = create\_engine(DB\_CONNECT\_STRING,echo=True)

* 1
* 2
* 3

create\_engine方法返回一个Engine实例，Engine实例只有直到触发数据库事件时才真正去连接数据库。   
echo参数是设置 SQLAlchemy 日志记录，这通过 Python 的标准logging模块的快捷方式。启用它，我们会看到产生的所有生成的 SQL，sqlalchemy与数据库通信的命令都将打印出来，例如执行：

engine.execute("select 1").scalar()

* 1

执行打印信息：

2016-02-28 23:55:37,544 INFO sqlalchemy.engine.base.Engine SHOW VARIABLES LIKE 'sql\_mode'

2016-02-28 23:55:37,544 INFO sqlalchemy.engine.base.Engine ()

2016-02-28 23:55:37,545 INFO sqlalchemy.engine.base.Engine SELECT DATABASE()

2016-02-28 23:55:37,546 INFO sqlalchemy.engine.base.Engine ()

......

* 1
* 2
* 3
* 4
* 5

**四、声明一个映射（declare a Mapping)**

declarative\_base类维持了一个从类到表的关系，通常一个应用使用一个base实例，所有实体类都应该继承此类对象

from sqlalchemy.ext.declarative import declarative\_base

Base = declarative\_base()

* 1
* 2

定义映射的表，表名称、 数据类型的列,在这里定义一个User类:

from sqlalchemy.ext.declarative import declarative\_base

from sqlalchemy import Column, Integer, String

from sqlalchemy import create\_engine

DB\_CONNECT\_STRING = 'mysql+mysqldb://root:hengtian@localhost/test?charset=utf8'

engine = create\_engine(DB\_CONNECT\_STRING,echo=True)

Base = declarative\_base()

class User(Base):

\_\_tablename\_\_ = 'users'

id = Column(Integer,primary\_key=True)

name = Column(String(10))

fullname = Column(String(20))

password = Column(String(20)) #可以设定长度

def \_\_init\_\_(self,name,fullname,password):

self.name = name

self.fullname = fullname

self.password = password

def \_\_repr(self):

return "<User('%s','%s','%s')>"%(self.name,self.fullname,self.password)

Base.metadata.create\_all(engine)

* 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

类使用声明式至少需要一个**tablename**属性定义数据库表名字，并至少一Column是主键。User类定义一个**repr**()方法，但请注意，是可选;   
sqlalchemy 就是把Base子类转变为数据库表，定义好User类后，会生成Table和mapper()，分别通过User.**table** 和User.**mapper**返回这两个对象，对于主键，象oracle没有自增长的主键时，要使用：

from sqlalchemy import Sequence

Column(Integer,Sequence('user\_idseq'),prmary\_key=True)

* 1
* 2

数据表字段长度定义：

Column(String(50))

* 1

Base.metadata返回sqlalchemy.schema.MetaData对象，它是所有Table对象的集合，调用create\_all()该对象会触发CREATE TABLE语句，如果数据库还不存在这些表的话。

>>> Base.metadata.create\_all(engine)

>>> CREATE TABLE users (

id INTEGER NOT NULL AUTO\_INCREMENT,

name VARCHAR(10),

fullname VARCHAR(20),

password VARCHAR(20),

PRIMARY KEY (id)

)

2016-02-29 22:52:03,260 INFO sqlalchemy.engine.base.Engine ()

2016-02-29 22:52:03,350 INFO sqlalchemy.engine.base.Engine COMMIT

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

创建一个User类：

ed\_user = User(name='ed', fullname='Ed Jones', password='edspassword')

print ed\_user.name

print ed\_user.password

print str(ed\_user.id)

输出：

>>>ed

>>>edspassword

>>>None

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

**五、创建Session**

Session 使用 connection发送query，把返回的result row 填充到一个object中，该对象同时还会保存在Session中，Session内部有一个叫 Identity Map的数据结构，为每一个对象维持了唯一的副本。primary key 作为 key ，value就是该object。session刚开始无状态，直到有query发起时。对象的变化会被session的跟踪维持着，在数据库做下一次查询后者当前的事务已经提交了时，it fushed all pendings changes to the database.   
这就是传说中的 Unit of work 模式   
例如：

def unit\_of\_work():

session = Session()

album = session.query(Album).get(4)

album.name = "jun"

#这里不会修改album的name属性，不会触发update语句

def unit\_of\_work():

session = Session()

album = session.query(Album).get(4)

album.name = "jun"

#这里修改了album的name属性，会触发一个update语句

session.query(Artist).get(11)

session.commit()

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

**构造了session，何时commit，何时close:**

规则：始终保持session与function和objecct分离

**对象的四种状态:**

⚠注意：对象实例有四种状态，分别是：   
Transient（瞬时的）：实例还不在session中，还没有保存到数据库中去，没有数据库身份，像刚创建出来的对象比如User()，仅仅只有mapper()与之关联.   
Pending（挂起的）：调用session.add()后，Transient对象就会变成Pending，这个时候它还是不会保存到数据库中，只有等到触发了flush动作才会存在数据库，比如query操作就可以出发flush。同样这个时候的实例的主键一样为None   
Persistent（持久的）：session中，数据库中都有对应的一条记录存在，主键有值了。   
Detached（游离的）：数据库中有记录，但是session中不存在，对这个状态的对象进行操作时，不会触发任何SQL语句。

**Session什么时候清理缓存:**

commit()方法调用的时候   
查询时会清理缓存，保证查询结果能反映对象的最新状态   
显示调用session的flush方法   
Session是真正与数据库通信的handler，你还可以把他理解一个容器，add就是往容器中添加对象   
执行完add方法后，ed\_user对象处于pending状态，不会触发INSERT语句，当然ed\_uesr.id也为None，如果在add方后有查询(session.query)，那么会flush一下，把数据刷一遍，把所有的pending信息先flush再执行query。

from sqlalchemy.orm import sessionmaker

Session = sessionmaker(bind=engine)

* 1
* 2

这个定制的Session类将创建新的Session对象绑定到我们的数据库。   
无论何时你需要与数据库连接，你实例化一个Session:

session = Session()

* 1

上述Session是与我们Mysql连接启用的Engine，但它还没有打开任何连接。它第一次使用时，由维护的连接池检索连接，并获取不放，直到我们提交所有更改和/或关闭会话对象才会释放掉。

添加User对象到我们定义的session上，

ed\_user = User(name='ed', fullname='Ed Jones', password='edspassword')

session.add(ed\_user)

* 1
* 2

添加多个的User对象，使用add\_all() :

session.add\_all([

User(name='wendy', fullname='Wendy Williams', password='foobar'),

User(name='mary', fullname='Mary Contrary', password='xxg527'),

User(name='fred', fullname='Fred Flinstone', password='blah')])

* 1
* 2
* 3
* 4

**提交事务：**

session.commit()

* 1

commit()刷新任何剩余的变化保持到数据库中，并将提交的事务。

**回滚：**

由于Session工作在一个事务内，我们可以回滚所做过的更改。

session.rollback()

* 1

**查询：**

在Session上使用query()方法，创建了一个Query对象。此函数接受数目可变的参数，可以是任意组合的类和类表的描述符   
Query对象通过Session.query获取，query接收类或属性参数，以及多个类

>>> for instance in session.query(User).order\_by(User.id):

... print(instance.name, instance.fullname)

ed Ed Jones

wendy Wendy Williams

mary Mary Contrary

fred Fred Flinstone

Query还接受 ORM 检测描述符作为参数,参数返回的结果被表示为元组:

>>> for name, fullname in session.query(User.name, User.fullname):

... print(name, fullname)

ed Ed Jones

wendy Wendy Williams

mary Mary Contrary

fred Fred Flinstone

由Query返回的元组是命名的元组，提供的KeyedTuple类，并可以像普通的 Python 对象多处理。名称是相同的属性，该属性的名称和类的类名:

>>> for row in session.query(User, User.name).all():

... print(row.User, row.name)

<User(name='ed', fullname='Ed Jones', password='f8s7ccs')> ed

<User(name='wendy', fullname='Wendy Williams', password='foobar')> wendy

<User(name='mary', fullname='Mary Contrary', password='xxg527')> mary

<User(name='fred', fullname='Fred Flinstone', password='blah')> fred

* 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

**筛选:**

filter\_by() 它使用关键字参数:

>>> for name, in session.query(User.name).\

... filter\_by(fullname='Ed Jones'):

... print(name)

ed

* 1
* 2
* 3
* 4

filter\_by接收的参数形式是关键字参数，而filter接收的参数是更加灵活的SQL表达式结构：

sqlalchemy源码对filter\_by的定义

def filter\_by(self, \*\*kwargs):

举例：

for user in session.query(User).filter\_by(name=’ed’).all():

print user

for user in session.query(User).filter(User.name==”ed”).all():

print user

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

**常用过滤操作：**

* equals:

query.filter(User.name == 'ed')

* 1
* 2
* not equal:

query.filter(User.name !='ed')

* 1
* LIKE:

query.filter(User.name.like('%d%')

* 1
* IN:

query.filter(User.name.in\_(['a','b','c'])

* 1
* NOT IN:

query.filter(~User.name.in\_(['ed','x'])

* 1
* IS NULL:

filter(User.name==None)

* 1
* IS NOT NULL:

filter(User.name!=None)

* 1
* AND:

from sqlalchemy import and\_

filter(and\_(User.name == 'ed',User.fullname=='xxx'))

* 1
* 2

或者多次调用filter或filter\_by

filter(User.name =='ed').filter(User.fullname=='xx')

* 1

还可以是：

query.filter(User.name == ‘ed’, User.fullname == ‘Ed Jones’)

* 1
* OR:

from sqlalchemy import or\_

query.filter(or\_(User.name == ‘ed’, User.name == ‘wendy’))

* 1
* 2

**查询返回结果:**

* query.all(): all()返回列表
* query.first(): 返回第一个元素
* query.one(): 有且只有一个元素时才正确返回。

此外，filter函数还可以接收text对象，text是SQL查询语句的字面对象，比如：

for user in session.query(User).filter(text(“id<224”)).order\_by(text(“id”)).all():

print user.name

* 1
* 2

**count:**

有两种count，第一种是纯粹是执行SQL语句后返回有多少行，对应的函数count()，第二个是func.count()，适用在分组统计，比如按性别分组时，男的有多少，女的多少：

session.query(User).filter(User.name==’ed’).count()

session.query(func.count(), User.name).group\_by(User.name).all( )

* 1
* 2

**六、Building a Relationship**

SQLAlchemy中的映射关系有四种,分别是一对多,多对一,一对一,多对多

**一对多(one to many）:**

因为外键(ForeignKey)始终定义在多的一方.如果relationship定义在多的一方,那就是多对一,一对多与多对一的区别在于其关联(relationship)的属性在多的一方还是一的一方，如果relationship定义在一的一方那就是一对多.   
这里的例子中,一指的是Parent,一个parent有多个child:

class Parent(Base):

\_\_tablename\_\_ = 'parent'

id = Column(Integer,primary\_key = True)

children = relationship("Child",backref='parent')

class Child(Base):

\_\_tablename\_\_ = 'child'

id = Column(Integer,primary\_key = True)

parent\_id = Column(Integer,ForeignKey('parent.id'))

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

**多对一(many to one):**

这个例子中many是指parent了,意思是一个child可能有多个parent(父亲和母亲),这里的外键(child\_id)和relationship(child)都定义在多(parent)的一方:

class Parent(Base):

\_\_tablename\_\_ = 'parent'

id = Column(Integer, primary\_key=True)

child\_id = Column(Integer, ForeignKey('child.id'))

child = relationship("Child", backref="parents")

class Child(Base):

\_\_tablename\_\_ = 'child'

id = Column(Integer, primary\_key=True)

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

为了建立双向关系,可以在relationship()中设置backref,Child对象就有parents属性.设置 cascade= ‘all’，可以级联删除:

class Parent(Base):

\_\_tablename\_\_ = 'parent'

id = Column(Integer,primary\_key = True)

children = relationship("Child",cascade='all',backref='parent')

def delete\_parent():

session = Session()

parent = session.query(Parent).get(2)

session.delete(parent)

session.commit()

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

不过不设置cascade，删除parent时，其关联的chilren不会删除，只会把chilren关联的parent.id置为空，设置cascade后就可以级联删除children

**一对一(one to one):**

一对一就是多对一和一对多的一个特例,只需在relationship加上一个参数uselist=False替换多的一端就是一对一:   
从一对多转换到一对一:

class Parent(Base):

\_\_tablename\_\_ = 'parent'

id = Column(Integer, primary\_key=True)

child = relationship("Child", uselist=False, backref="parent")

class Child(Base):

\_\_tablename\_\_ = 'child'

id = Column(Integer, primary\_key=True)

parent\_id = Column(Integer, ForeignKey('parent.id'))

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

从多对一转换到一对一:

class Parent(Base):

\_\_tablename\_\_ = 'parent'

id = Column(Integer, primary\_key=True)

child\_id = Column(Integer, ForeignKey('child.id'))

child = relationship("Child", backref=backref("parent", uselist=False))

class Child(Base):

\_\_tablename\_\_ = 'child'

id = Column(Integer, primary\_key=True)

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

**多对多(many to many):**

多对多关系需要一个中间关联表,通过参数secondary来指定,

from sqlalchemy import Table,Text

post\_keywords = Table('post\_keywords',Base.metadata,

Column('post\_id',Integer,ForeignKey('posts.id')),

Column('keyword\_id',Integer,ForeignKey('keywords.id'))

)

class BlogPost(Base):

\_\_tablename\_\_ = 'posts'

id = Column(Integer,primary\_key=True)

body = Column(Text)

keywords = relationship('Keyword',secondary=post\_keywords,backref='posts')

class Keyword(Base):

\_\_tablename\_\_ = 'keywords'

id = Column(Integer,primary\_key = True)

keyword = Column(String(50),nullable=False,unique=True)

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

**七、关联查询（query with join）**

简单地可以使用：

>>> for u, a in session.query(User, Address).\

... filter(User.id==Address.user\_id).\

... filter(Address.email\_address=='jack@google.com').\

... all():

... print(u)

... print(a)

<User(name='jack', fullname='Jack Bean', password='gjffdd')>

<Address(email\_address='jack@google.com')>

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

如果是使用真正的关联SQL语法来查询可以使用：

>>> session.query(User).join(Address).\

... filter(Address.email\_address=='jack@google.com').\

... all()

[<User(name='jack', fullname='Jack Bean', password='gjffdd')>]

* 1
* 2
* 3
* 4

因为这里的外键就一个，系统知道如何去关联

**八、常用参数解释:**

relationship():函数接收的参数非常多，比如：backref，secondary，primaryjoin，等等。列举一下我用到的参数:   
backref:在一对多或多对一之间建立双向关系,比如:

class Parent(Base):

\_\_tablename\_\_ = 'parent'

id = Column(Integer, primary\_key=True)

children = relationship("Child", backref="parent")

class Child(Base):

\_\_tablename\_\_ = 'child'

id = Column(Integer, primary\_key=True)

parent\_id = Column(Integer, ForeignKey('parent.id'))

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

Prarent对象获取children,parent.children,反过来Child对象可以获取parent:child.parent.

lazy:默认值是True,说明关联对象只有到真正访问的时候才会去查询数据库,比如有parent对象,只有知道访问parent.children的时候才做关联查询.

remote\_side:表中的外键引用的是自身时,如Node类,如果想表示多对一的关系,那么就可以使用remote\_side

class Node(Base):

\_\_tablename\_\_ = 'node'

id = Column(Integer, primary\_key=True)

parent\_id = Column(Integer, ForeignKey('node.id'))

data = Column(String(50))

parent = relationship("Node", remote\_side=[id])

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

如果是想建立一种双向的关系,那么还是结合backref:

class Node(Base):

\_\_tablename\_\_ = 'node'

id = Column(Integer, primary\_key=True)

parent\_id = Column(Integer, ForeignKey('node.id'))

data = Column(String(50))

children = relationship("Node",

backref=backref('parent', remote\_side=[id])

)

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

primaryjoin:用在一对多或者多对一的关系中,默认情况连接条件就是主键与另一端的外键,用primaryjoin参数可以用来指定连接条件 ,比如:下面user的address必须现address是一’tony’开头:

class User(Base):

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

id = Column(Integer, primary\_key=True)

name = Column(String)

addresses = relationship("Address",

primaryjoin="and\_(User.id==Address.user\_id, "

"Address.email.startswith('tony'))",

backref="user")

class Address(Base):

\_\_tablename\_\_ = 'address'

id = Column(Integer, primary\_key=True)

email = Column(String)

user\_id = Column(Integer, ForeignKey('user.id'))

secondary:

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

order\_by:   
在一对多的关系中,如下代码:

class User(Base):

....

addresses = relationship("Address",

order\_by="desc(Address.email)",

primaryjoin="Address.user\_id==User.id")

* 1
* 2
* 3
* 4
* 5

如果user的address要按照email排序,那么就可以在relationship中添加参数order\_by.这里的参数是一字符串形式表示的,不过它等同于python表达式,其实还有另一种基于lambda的方式:

class User(Base):

...

addresses = relationship(lambda: Address,

order\_by=lambda: desc(Address.email),

primaryjoin=lambda: Address.user\_id==User.id)

* 1
* 2
* 3
* 4
* 5

**九、简单的例子**

#!/usr/bin/env python

# -\*- coding: UTF-8 -\*-

from sqlalchemy.orm import mapper, sessionmaker

\_\_author\_\_ = 'jpzhang'

from sqlalchemy import create\_engine, Table, Column, Integer, String, MetaData

from sqlalchemy.sql.expression import Cast

from sqlalchemy.ext.compiler import compiles

from sqlalchemy.dialects.mysql import \

BIGINT, BINARY, BIT, BLOB, BOOLEAN, CHAR, DATE, \

DATETIME, DECIMAL, DECIMAL, DOUBLE, ENUM, FLOAT, INTEGER, \

LONGBLOB, LONGTEXT, MEDIUMBLOB, MEDIUMINT, MEDIUMTEXT, NCHAR, \

NUMERIC, NVARCHAR, REAL, SET, SMALLINT, TEXT, TIME, TIMESTAMP, \

TINYBLOB, TINYINT, TINYTEXT, VARBINARY, VARCHAR, YEAR

#表的属性描述对象

metadata = MetaData()

userTable = Table(

"wzp\_user",metadata,

Column('user\_id', Integer, primary\_key=True),

Column('user\_name', VARCHAR(50), unique=True, nullable=False),

Column('password', VARCHAR(40), nullable=True)

)

#创建数据库连接,MySQLdb连接方式

mysql\_db = create\_engine('mysql://用户名:密码@ip:port/dbname')

#创建数据库连接，使用 mysql-connector-python连接方式

#mysql\_db = create\_engine("mysql+mysqlconnector://用户名:密码@ip:port/dbname")

#生成表

metadata.create\_all(mysql\_db)

#创建一个映射类

class User(object):

pass

#把表映射到类

mapper(User, userTable)

#创建了一个自定义了的 Session类

Session = sessionmaker()

#将创建的数据库连接关联到这个session

Session.configure(bind=mysql\_db)

session = Session()

def main():

u = User()

#给映射类添加以下必要的属性,因为上面创建表指定这个字段不能为空,且唯一

u.user\_name='tan9le测试'

#按照上面创建表的相关代码，这个字段允许为空

u.password='123456'

#在session中添加内容

session.add(u)

#保存数据

session.flush()

#数据库事务的提交,sisson自动过期而不需要关闭

session.commit()

#query() 简单的理解就是select() 的支持 ORM 的替代方法,可以接受任意组合的 class/column 表达式

query = session.query(User)

#列出所有user

print list(query)

#根据主键显示

print query.get(1)

#类似于SQL的where,打印其中的第一个

print query.filter\_by(user\_name='tan9le测试').first()

u = query.filter\_by(user\_name='tan9le测试').first()

#修改其密码字段

u.password = '654321'

#提交事务

session.commit()

#打印会出现新密码

print query.get(1).password

#根据id字段排序,打印其中的用户名和密码

for instance in session.query(User).order\_by(User.user\_id):

print instance.user\_name, instance.password

#释放资源

session.close()

if \_\_name\_\_ == '\_\_main\_\_':

main()