# [sqlalchemy学习笔记](https://segmentfault.com/a/1190000006949536)

[SQLAlchemy](http://www.sqlalchemy.org/" \t "_blank)是python的一个数据库ORM工具，提供了强大的对象模型间的转换，可以满足绝大多数数据库操作的需求，并且支持多种数据库引擎（sqlite，mysql，postgres, mongodb等），在这里记录基本用法和学习笔记

## 一、安装

通过pip安装

$ pip install SQLAlchemy

## 二、使用

首先是连接到数据库，SQLALchemy支持多个数据库引擎，不同的数据库引擎连接字符串不一样，常用的有

mysql://username:password@hostname/database

postgresql://username:password@hostname/database

sqlite:////absolute/path/to/database

sqlite:///c:/absolute/path/to/database

更多连接字符串的介绍参见[这里](http://docs.sqlalchemy.org/en/latest/core/engines.html?highlight=create_engine#database-urls)

下面是连接和使用sqlite数据库的例子

### 1. connection

使用传统的connection的方式连接和操作数据库

**from** sqlalchemy **import** create\_engine

# 数据库连接字符串

DB\_CONNECT\_STRING = 'sqlite:///:memory:'

# 创建数据库引擎,echo为True,会打印所有的sql语句

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

# 创建一个connection，这里的使用方式与python自带的sqlite的使用方式类似

**with** engine.connect() **as** con:

# 执行sql语句，如果是增删改，则直接生效，不需要commit

rs = con.execute('SELECT 5')

data = rs.fetchone()[0]

**print** "Data: %s" % data

与python自带的sqlite不同，这里不需要Cursor光标，执行sql语句不需要commit

### 2. connection事务

使用事务可以进行批量提交和回滚

**from** sqlalchemy **import** create\_engine

# 数据库连接字符串

DB\_CONNECT\_STRING = 'sqlite:////Users/zhengxiankai/Desktop/Document/db.sqlite'

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

**with** engine.connect() **as** connection:

trans = connection.begin()

**try**:

r1 = connection.execute("select \* from User")

r2 = connection.execute("insert into User(name, age) values(?, ?)", 'bomo', 24)

trans.commit()

**except**:

trans.rollback()

**raise**

### 3. session

connection是一般使用数据库的方式，sqlalchemy还提供了另一种操作数据库的方式，通过session对象，session可以记录和跟踪数据的改变，在适当的时候提交，并且支持强大的ORM的功能，下面是基本使用

**from** sqlalchemy **import** create\_engine

**from** sqlalchemy.orm **import** sessionmaker

# 数据库连接字符串

DB\_CONNECT\_STRING = 'sqlite:////Users/zhengxiankai/Desktop/Document/db.sqlite'

# 创建数据库引擎,echo为True,会打印所有的sql语句

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

# 创建会话类

DB\_Session = sessionmaker(bind=engine)

# 创建会话对象

session = DB\_Session()

# dosomething with session

# 用完记得关闭，也可以用with

session.close()

上面创建了一个session对象，接下来可以操作数据库了，session也支持通过sql语句操作数据库

session.execute('select \* from User')

session.execute("insert into User(name, age) values('bomo', 13)")

session.execute("insert into User(name, age) values(:name, :age)", {'name': 'bomo', 'age':12})

# 如果是增删改，需要commit

session.commit()

注意参数使用dict，并在sql语句中使用:key占位

### 4. ORM

上面简单介绍了sql的简单用法，既然是ORM框架，我们先定义两个模型类User和Role，sqlalchemy的模型类继承自一个由declarative\_base()方法生成的类，我们先定义一个模块Models.py生成Base类

**from** sqlalchemy.ext.declarative **import** declarative\_base

Base = declarative\_base()

User.py

**from** sqlalchemy **import** Column, Integer, String

**from** Models **import** Base

**class** **User**(Base):

\_\_tablename\_\_ = 'User'

id = Column('id', Integer, primary\_key=**True**, autoincrement=**True**)

name = Column('name', String(50))

age = Column('age', Integer)

Role.py

**from** sqlalchemy **import** Column, Integer, String

**from** Models **import** Base

**class** **Role**(Base):

\_\_tablename\_\_ = 'Role'

id = Column('id', Integer, primary\_key=**True**, autoincrement=**True**)

name = Column('name', String(50))

从上面很容易看出来，这里的模型对应数据库中的表，模型支持的类型有Integer, String, Boolean, Date, DateTime, Float，更多类型包括类型对应的Python的类型参见：[这里](http://docs.sqlalchemy.org/en/latest/core/type_basics.html?highlight=column%20type#generic-types)

Column构造函数相关设置

* name：名称
* type\_：列类型
* autoincrement：自增
* default：默认值
* index：索引
* nullable：可空
* primary\_key：外键

更多介绍参见[这里](http://docs.sqlalchemy.org/en/latest/core/metadata.html?highlight=column%20autoincrement#sqlalchemy.schema.Column.__init__)

接下来通过session进行增删改查

**from** sqlalchemy **import** create\_engine

**from** sqlalchemy.orm **import** sessionmaker

**from** User **import** User

**from** Role **import** Role

**from** Models **import** Base

DB\_CONNECT\_STRING = 'sqlite:////Users/zhengxiankai/Desktop/Document/db.sqlite'

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

DB\_Session = sessionmaker(bind=engine)

session = DB\_Session()

# 1. 创建表（如果表已经存在，则不会创建）

Base.metadata.create\_all(engine)

# 2. 插入数据

u = User(name = 'tobi', age = 200)

r = Role(name = 'user')

# 2.1 使用add，如果已经存在，会报错

session.add(u)

session.add(r)

session.commit()

**print** r.id

# 3 修改数据

# 3.1 使用merge方法，如果存在则修改，如果不存在则插入（只判断主键，不判断unique列）

r.name = 'admin'

session.merge(r)

# 3.2 也可以通过这种方式修改

session.query(Role).filter(Role.id == 1).update({'name': 'admin'})

# 4. 删除数据

session.query(Role).filter(Role.id == 1).delete()

# 5. 查询数据

# 5.1 返回结果集的第二项

user = session.query(User).get(2)

# 5.2 返回结果集中的第2-3项

users = session.query(User)[1:3]

# 5.3 查询条件

user = session.query(User).filter(User.id < 6).first()

# 5.4 排序

users = session.query(User).order\_by(User.name)

# 5.5 降序（需要导入desc方法）

**from** sqlalchemy **import** desc

users = session.query(User).order\_by(desc(User.name))

# 5.6 只查询部分属性

users = session.query(User.name).order\_by(desc(User.name))

**for** user **in** users:

**print** user.name

# 5.7 给结果集的列取别名

users = session.query(User.name.label('user\_name')).all()

**for** user **in** users:

**print** user.user\_name

# 5.8 去重查询（需要导入distinct方法）

**from** sqlalchemy **import** distinct

users = session.query(distinct(User.name).label('name')).all()

# 5.9 统计查询

user\_count = session.query(User.name).order\_by(User.name).count()

age\_avg = session.query(func.avg(User.age)).first()

age\_sum = session.query(func.sum(User.age)).first()

# 5.10 分组查询

users = session.query(func.count(User.name).label('count'), User.age).group\_by(User.age)

**for** user **in** users:

**print** 'age:{0}, count:{1}'.format(user.age, user.count)

# 6.1 exists查询(不存在则为~exists())

**from** sqlalchemy.sql **import** exists

session.query(User.name).filter(~exists().where(User.role\_id == Role.id))

# SELECT name AS users\_name FROM users WHERE NOT EXISTS (SELECT \* FROM roles WHERE users.role\_id = roles.id)

# 6.2 除了exists，any也可以表示EXISTS

session.query(Role).filter(Role.users.any())

# 7 random

**from** sqlalchemy.sql.functions **import** random

user = session.query(User).order\_by(random()).first()

session.close()

参考链接：

* [any](http://docs.sqlalchemy.org/en/latest/orm/internals.html?highlight=any#sqlalchemy.orm.properties.RelationshipProperty.Comparator.any)

### 5. 多表关系

上面的所有操作都是基于单个表的操作，下面是多表以及关系的使用，我们修改上面两个表，添加外键关联（一对多和多对一）

User模型

**from** sqlalchemy **import** Column, Integer, String

**from** sqlalchemy **import** ForeignKey

**from** sqlalchemy.orm **import** relationship

**from** Models **import** Base

**class** **User**(Base):

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

id = Column('id', Integer, primary\_key=**True**, autoincrement=**True**)

name = Column('name', String(50))

age = Column('age', Integer)

# 添加角色id外键(关联到Role表的id属性)

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

# 添加同表外键

second\_role\_id = Column('second\_role\_id', Integer, ForeignKey('roles.id'))

# 添加关系属性，关联到role\_id外键上

role = relationship('Role', foreign\_keys='User.role\_id', backref='User\_role\_id')

# 添加关系属性，关联到second\_role\_id外键上

second\_role = relationship('Role', foreign\_keys='User.second\_role\_id', backref='User\_second\_role\_id')

Role模型

**from** sqlalchemy **import** Column, Integer, String

**from** sqlalchemy.orm **import** relationship

**from** Models **import** Base

**class** **Role**(Base):

\_\_tablename\_\_ = 'roles'

id = Column('id', Integer, primary\_key=**True**, autoincrement=**True**)

name = Column('name', String(50))

# 添加关系属性，关联到User.role\_id属性上

users = relationship("User", foreign\_keys='User.role\_id', backref="Role\_users")

# 添加关系属性，关联到User.second\_role\_id属性上

second\_users = relationship("User", foreign\_keys='User.second\_role\_id', backref="Role\_second\_users")

这里有一点需要注意的是，设置外键的时候ForeignKey('roles.id')这里面使用的是表名和表列，在设置关联属性的时候relationship('Role', foreign\_keys='User.role\_id', backref='User\_role\_id')，这里的foreign\_keys使用的时候类名和属性名

接下来就可以使用了

u = User(name='tobi', age=200)

r1 = Role(name='admin')

r2 = Role(name='user')

u.role = r1

u.second\_role = r2

session.add(u)

session.commit()

# 查询（对于外键关联的关系属性可以直接访问，在需要用到的时候session会到数据库查询）

roles = session.query(Role).all()

**for** role **in** roles:

**print** 'role:{0} users'

**for** user **in** role.users:

**print** '\t{0}'.format(user.name)

**print** 'role:{0} second\_users'

**for** user **in** role.second\_users:

**print** '\t{0}'.format(user.name)

上面表示的是一对多（多对一）的关系，还有一对一，多对多，如果要表示一对一的关系，在定义relationship的时候设置uselist为False（默认为True），如在Role中

**class** **Role**(Base):

...

user = relationship("User", uselist=**False**, foreign\_keys='User.role\_id', backref="Role\_user")

### 6. 多表查询

多表查询通常使用join进行表连接，第一个参数为表名，第二个参数为条件，例如

users = db.session.query(User).join(Role, Role.id == User.role\_id)

**for** u **in** users:

**print** u.name

join为内连接，还有左连接outerjoin，用法与join类似，右连接和全外链接在1.0版本上不支持，通常来说有这两个结合查询的方法基本够用了，1.1版本貌似添加了右连接和全外连接的支持，但是目前只是预览版

还可以直接查询多个表，如下

result = db.session.query(User, Role).filter(User.role\_id = Role.id)

# 这里选择的是两个表，使用元组获取数据

**for** u, r **in** result:

**print** u.name

## 三、数据库迁移

sqlalchemy的数据库迁移/升级有两个库支持[alembic](http://alembic.zzzcomputing.com/en/latest/)和[sqlalchemy-migrate](https://sqlalchemy-migrate.readthedocs.io/en/latest/)

由于sqlalchemy-migrate在2011年发布了0.7.2版本后，就已经停止更新了，并且已经不维护了，也积累了很多bug，而alembic是较后来才出现，而且是sqlalchemy的作者开发的，有良好的社区支持，所以在这里只学习alembic这个库

alembic实现了类似git/svn的版本管理的控制，我们可以通过alembic维护每次升级数据库的版本

### 1. 安装

通过pip安装，pip会自动安装相关的依赖

$ pip install alembic

### 2. 初始化

安装完成后再项目根目录运行

$ alembic init YOUR\_ALEMBIC\_DIR

alembic会在根目录创建YOUR\_ALEMBIC\_DIR目录和alembic.ini文件，如下

yourproject/

alembic.ini

YOUR\_ALEMBIC\_DIR/

env.py

README

script.py.mako

versions/

3512b954651e\_add\_account.py

2b1ae634e5cd\_add\_order\_id.py

3adcc9a56557\_rename\_username\_field.py

其中

* alembic.ini 提供了一些基本的配置
* env.py 每次执行Alembic都会加载这个模块，主要提供项目Sqlalchemy Model 的连接
* script.py.mako 迁移脚本生成模版
* versions 存放生成的迁移脚本目录

默认情况下创建的是基于单个数据库的，如果需要支持多个数据库或其他，可以通过alembic list\_templates查看支持的模板

$ alembic list\_templates

Available templates:

generic - Generic single-database configuration.

multidb - Rudimentary multi-database configuration.

pylons - Configuration that reads from a Pylons project environment.

Templates are used via the 'init' command, e.g.:

alembic init --template generic ./scripts

### 3. 配置

使用之前，需要配置一下链接字符串，打开alembic.ini文件，设置sqlalchemy.url连接字符串，例如

sqlalchemy.url = sqlite:////Users/zhengxiankai/Desktop/database.db

其他参数可以参见官网说明：<http://alembic.zzzcomputing.com/en/latest/tutorial.html>

### 4. 创建数据库版本

接下来我们创建一个数据库版本，并新建两个表

$ alembic revision -m 'create table'

创建一个版本（会在yourproject/YOUR\_ALEMBIC\_DIR/versions/文件夹中创建一个python文件1a8a0d799b33\_create\_table.py）

该python模块包含upgrade和downgrade两个方法，在这里添加一些新增表的逻辑

"""create table

Revision ID: 4fd533a56b34

Revises:

Create Date: 2016-09-18 17:20:27.667100

"""

**from** alembic **import** op

**import** sqlalchemy **as** sa

# revision identifiers, used by Alembic.

revision = '4fd533a56b34'

down\_revision = **None**

branch\_labels = **None**

depends\_on = **None**

**def** **upgrade**():

# 添加表

op.create\_table(

'account',

sa.Column('id', sa.Integer, primary\_key=**True**),

sa.Column('name', sa.String(50), nullable=**False**),

sa.Column('description', sa.Unicode(200)),

)

# 添加列

# op.add\_column('account', sa.Column('last\_transaction\_date', sa.DateTime))

**def** **downgrade**():

# 删除表

op.drop\_table('account')

# 删除列

# op.drop\_column('account', 'last\_transaction\_date')

这里使用到了了op对象，关于op对象的更多API使用，参见[这里](http://alembic.zzzcomputing.com/en/latest/ops.html#ops)

这里生成的文件名是依照在alembic.ini文件声明的模板来的，默认为版本号+名字，可以加上一些日期信息，否则不好排序，更多参数参见[这里](http://alembic.zzzcomputing.com/en/latest/tutorial.html?highlight=file_template)

file\_template = %%(year)d\_%%(month).2d\_%%(day).2d\_%%(hour).2d\_%%(minute).2d\_%%(rev)s\_%%(slug)s

另外通常我们也改一下生成模板script.py.mako，加上编码信息，否则在升级脚本中如果有中文会报错

#!/usr/bin/python

# -\*- coding:utf-8 -\*-

### 5. 升级数据库

刚刚实现了升级和降级的方法，通过下面命令升级数据库到最新版本

$ alembic upgrade head

这时候可以看到数据库多了两个表alembic\_version和account，alembic\_version存放数据库版本

关于升级和降级的其他命令还有下面这些

# 升到最高版本

$ alembic upgrade head

# 降到最初版本

$ alembic downgrade base

# 升两级

$ alembic upgrade +2

# 降一级

$ alembic downgrade -1

# 升级到制定版本

$ alembic upgrade e93b8d488143

# 查看当前版本

$ alembic current

# 查看历史版本详情

$ alembic history --verbose

# 查看历史版本（-r参数）类似切片

$ alembic history -r1975ea:ae1027

$ alembic history -r-3:current

$ alembic history -r1975ea:

### 6. 通过元数据升级数据库

上面我们是通过API升级和降级，我们也可以直接通过元数据更新数据库，也就是自动生成升级代码，先定义两个Model（User, Role），这里我定义成三个文件

yourproject/

YOUR\_ALEMBIC\_DIR/

tutorial/Db

Models.py

User.py

Role.py

代码就放在一起了

**from** sqlalchemy.ext.declarative **import** declarative\_base

**from** sqlalchemy **import** Column, Integer, String

Base = declarative\_base()

**class** **User**(Base):

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

id = Column('id', Integer, primary\_key=**True**, autoincrement=**True**)

name = Column('name', String)

**class** **Role**(Base):

\_\_tablename\_\_ = 'roles'

id = Column('id', Integer, primary\_key=**True**, autoincrement=**True**)

name = Column('name', String)

在YOUR\_ALEMBIC\_DIR/env.py配置元数据

target\_metadata = **None**

改为

**import** os

**import** sys

# 这里需要添加相对路径到sys.path，否则会引用失败，尝试过使用相对路径，但各种不好使，还是使用这种方法靠谱些

sys.path.append(os.path.abspath(os.path.join(os.getcwd(), "../yourproject/tutorial/Db")))

**from** User **import** User

**from** Role **import** Role

**from** Models **import** Base

target\_metadata = Base.metadata

os.path.join(os.getcwd()这个获取到的地址不是env.py的路径，而是根目录

在创建数据库版本的时候添加--autogenerate参数，就会从Base.metadata元数据中生成脚本

$ alembic revision --autogenerate -m "add user table"

这时候会在生成升级代码

"""add user table

Revision ID: 97de1533584a

Revises: 8678ab6d48c1

Create Date: 2016-09-19 21:58:00.758410

"""

**from** alembic **import** op

**import** sqlalchemy **as** sa

# revision identifiers, used by Alembic.

revision = '97de1533584a'

down\_revision = '8678ab6d48c1'

branch\_labels = **None**

depends\_on = **None**

**def** **upgrade**():

### commands auto generated by Alembic - please adjust! ###

op.create\_table('roles',

sa.Column('id', sa.Integer(), nullable=**False**),

sa.Column('name', sa.String(), nullable=**True**),

sa.PrimaryKeyConstraint('id')

)

op.create\_table('users',

sa.Column('id', sa.Integer(), nullable=**False**),

sa.Column('name', sa.String(), nullable=**True**),

sa.PrimaryKeyConstraint('id')

)

op.drop\_table('account')

### end Alembic commands ###

**def** **downgrade**():

### commands auto generated by Alembic - please adjust! ###

op.create\_table('account',

sa.Column('id', sa.INTEGER(), nullable=**False**),

sa.Column('name', sa.VARCHAR(length=50), nullable=**False**),

sa.Column('description', sa.VARCHAR(length=200), nullable=**True**),

sa.Column('last\_transaction\_date', sa.DATETIME(), nullable=**True**),

sa.PrimaryKeyConstraint('id')

)

op.drop\_table('users')

op.drop\_table('roles')

### end Alembic commands ###

由于我没有定义account模型，会被识别为删除，如果删除了model的列的声明，则会被识别为删除列，自动生成的版本我们也可以自己修改，然后执行升级命令即可升级alembic upgrade head

**需要注意的是**

1. Base.metadata声明的类必须以数据库中的一一对应，如果数据库中有的表，而在元数据中没有，会识别成删除表
2. revision创建版本之前执行之前需要升级到最新版本
3. 配置Base之前，需要保证所有的Model都已经执行（即导入）过一次了，否则无法读取到，也就是需要把所有Model都import进来

数据库升级有风险，升级前最好先检查一遍upgrade函数，可以的话做好备份哈

## 四、常见问题

### 1. String长度问题

如果使用mysql数据库，String类型对应的是VARCHAR类型，需要指定长度，否则会报下面错误，而在sqlite不会出现

(**in** table 'user', column 'name'): VARCHAR requires a length on dialect mysql

如有问题欢迎到[我的博客](http://zhengbomo.github.com/2016-09-11/sqlalchemy-start/)留言

## 五、参考链接

* [Auto Generating Migrations](http://alembic.zzzcomputing.com/en/latest/autogenerate.html)
* [tutorial](http://alembic.zzzcomputing.com/en/latest/tutorial.html)

最后安利一下自己的博客：[http://zhengbo](http://zhengbomo.github.com/)