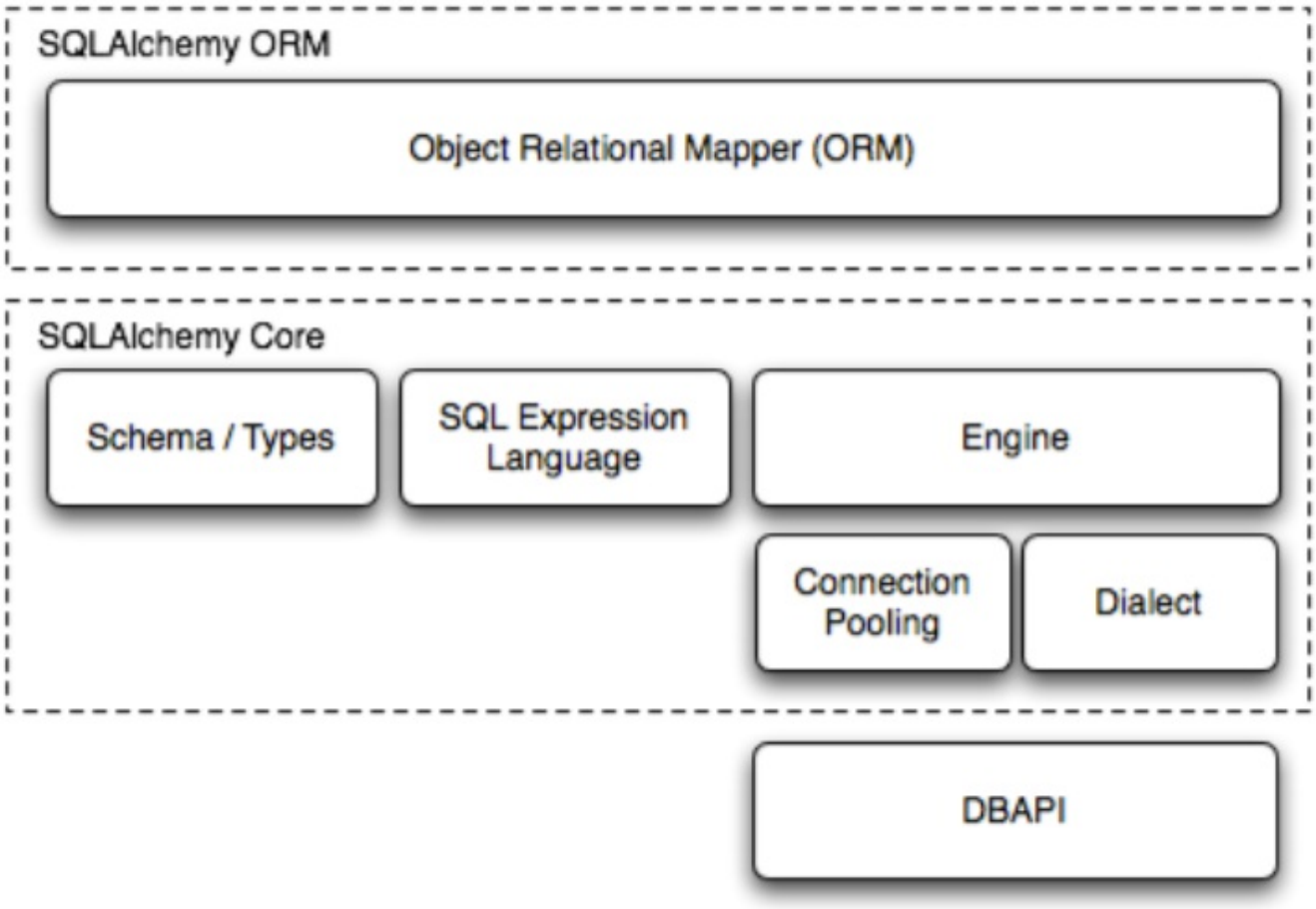
sqlalchemy 笔记

sqlalchemy介绍

sqlalchemy是一个使Python应用程序和数据库之间建立通信的一个python库，大多数时候,这个库是作为一个对象关系映射器(ORM)工具。

sqlalchemy结构图



1.DBAPI：创建指定Python模块与数据库之间如何暴露接口，尽管我们不是直接与这些底层API打交道，例如它存在的api：connect，close，commit，和rollback等，但是需要很好的理解它的行为以及效果。

2.SQLAlchemy Engines：当我们想使用sqlalchemy与数据库进行通信，我们需要创建一个数据库驱动引擎，在sqlalchemy中，engine引擎用来管理线程池pool和 dialect。

例如：

```
from sqlalchemy import create_engine
engine = create_engine('postgresql://usr:pass@localhost:5432/sqlalchemy')
```

这个例子中创建了一个postgresql数据库实例，但是需要注意的是，创建一个数据库引擎并不会立即连接数据库，这个过程被推迟到真正需要的适合，像我们提交submit一个query，或者创建或者更新一条表中的数据。

3.sqlalchemy数据库连接池： 连接池是对象池模式的一种比较传统的实现方式，通过使用对象池作为预缓存对象直接拿来使用，而不是花费时间来创建经常需要的对象，像数据库的连接。例如，通过create_engine()函数创建一个engine经常会自动创建一个QueuePool连接池，这种池配置了一些合理的默认值,如池的最大连接数为5.

表示案例

映射表orm映射

1.**Declare a Mapping**：定义类到数据库表table的映射，declarative_base函数定义了和table对应的关系映射。下面我们定义了User类,Base类是模型的基类

```
from sqlalchemy.ext.declarative import declarative_base
Base = declarative_base()
from sqlalchemy import Column, Integer, String
   class User(Base):
         __tablename__ = 'users'
        id = Column(Integer, primary_key=True)
        ...
```

2. **Base.metadata.create_all(bind=engine)** 和**Base.metadata.drop_all(bind=engine)**:从base中由于定义了User类，我们可以拿到metadata元数据，然后create_all以后即可连接数据库，数据库中即可生成相关的表。

多种模型关系

定义关系(一对一，多对一/一对多)

Building a Relationship:

一对多关系定义，在这个例子中，parent的id放在childern类定义中，然后在parent中定义了relationship关联children。

```
class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    children = relationship("Child")
class Children(Base):
     __tablename__ = 'child'
    id = Column(Integer, primary_key=True)
    parent_id = Column(Integer, ForeignKey('parent.id'))
```

一对一关系定义，只需要在一对多关系基础上的parent类中使用 uselist =False来表示,然后在则可表示为一对一。

```
     class Parent(Base):
        __tablename__ = 'parent'
        id = Column(Integer, primary_key=True)
        child = relationship("Child", uselist=False, back_populates="parent")
    class Child(Base):
        __tablename__ = 'child'
```

```
        id = Column(Integer, primary_key=True)
        parent_id = Column(Integer, ForeignKey('parent.id'))
        parent = relationship("Parent", back_populates="child")
```

- Many To Many
  多对多关系会在两个类之间增加一个关联的表

```
association_table = Table('association', Base.metadata,Column('left_id', Integer, Fo
reignKey('left.id')),Column('right_id', Integer, ForeignKey('right.id'))
class Parent(Base):
    __tablename__ = 'left'
    id = Column(Integer, primary_key=True)
    children = relationship("Child",secondary=association_table)
    # 在父表中的 relationship() 方法传入 secondary 参数，其值为关联表的表名
class Child(Base):
    __tablename__ = 'right'
    id = Column(Integer, primary_key=True)
```

3.**backref**作用
提供了动态指向表与表方向性的引用，即在relationship关联表这种一对多属性的单向引用到双向引用，即可以双方均可以得到关联的属性值。

```
u = User()
u.addresses#[]
a = Address()
a.user#Traceback AttributeError:'Address' object has no attribute 'user'
```

但是当我们有从Address对象获取所属用户的需求时，backref参数就派上用场了。

```
addresses = relationship('Address', backref='user')
```

example

An example with departments and employees:**A department has many employees** while an employee belongs to at most one department. Therefore, the database could be designed as follows:

```
>>> from sqlalchemy import Column, String, Integer, ForeignKey
>>> from sqlalchemy.orm import relationship, backref
>>> from sqlalchemy.ext.declarative import declarative_base
>>>
>>>
>>> Base = declarative_base()
>>>
>>>
>>> class Department(Base):
...     __tablename__ = 'department'
...     id = Column(Integer, primary_key=True)
...     name = Column(String)
...
>>>
>>> class Employee(Base):
...     __tablename__ = 'employee'
...     id = Column(Integer, primary_key=True)
...     name = Column(String)
...     department_id = Column(Integer, ForeignKey('department.id'))
...     department = relationship(Department, backref=backref('employees', uselist=True))
...
```

```
>>>
>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite:///')
>>>
>>> from sqlalchemy.orm import sessionmaker
>>> session = sessionmaker()
>>> session.configure(bind=engine)
>>> Base.metadata.create_all(engine)
```

In this example, we created a in-memory sqlite database with two tables 'department' and 'employee'. The column 'employee.department_id' is a foreign key to the column 'department.id' and the relationship 'department.employees' include all the employees in that department. To test our setup, we can simply insert several example records and query them using SQLAlchemy's ORM:

```
>>> john = Employee(name='john')
>>> it_department = Department(name='IT')
>>> john.department = it_department
>>> s = session()
>>> s.add(john)
>>> s.add(it_department)
>>> s.commit()
>>> it = s.query(Department).filter(Department.name == 'IT').one()
>>> it.employees
[]
>>> it.employees[0].name
u'john'
```

As you can see, we inserted one employee, john, into the IT department.

```
>>> from sqlalchemy import select
>>> find_it = select([Department.id]).where(Department.name == 'IT')
>>> rs = s.execute(find_it)
>>> rs

>>> rs.fetchone()
(1,)
>>> rs.fetchone()  # Only one result is returned from the query, so getting one more retu
rns None.
>>> rs.fetchone()  # Since the previous fetchone() returned None, fetching more would lea
d to a result-closed exception
Traceback (most recent call last):
  File "", line 1, in
  File "/Users/xiaonuogantan/python2-workspace/lib/python2.7/site-packages/sqlalchemy/eng
ine/result.py", line 790, in fetchone
    self.cursor, self.context)
  File "/Users/xiaonuogantan/python2-workspace/lib/python2.7/site-packages/sqlalchemy/eng
ine/base.py", line 1027, in _handle_dbapi_exception
    util.reraise(*exc_info)
  File "/Users/xiaonuogantan/python2-workspace/lib/python2.7/site-packages/sqlalchemy/eng
ine/result.py", line 781, in fetchone
    row = self._fetchone_impl()
  File "/Users/xiaonuogantan/python2-workspace/lib/python2.7/site-packages/sqlalchemy/eng
ine/result.py", line 700, in _fetchone_impl
    self._non_result()
  File "/Users/xiaonuogantan/python2-workspace/lib/python2.7/site-packages/sqlalchemy/eng
ine/result.py", line 724, in _non_result
    raise exc.ResourceClosedError("This result object is closed.")
sqlalchemy.exc.ResourceClosedError: This result object is closed.
>>> find_john = select([Employee.id]).where(Employee.department_id == 1)
```

```
>>> rs = s.execute(find_john)

>>> rs.fetchone()  # Employee John's ID
(1,)
>>> rs.fetchone()
```

Many-to-many between the departments and the employees

In our previous example, it’s simple that one employee belongs to at most one department. What if an employee could belong to multiple departments? Isn’t one foreign key not enough to represent this kind of relationship?
Yes, one foreign key is not enough. To model a many-to-many relationship between department and employee, we create a new association table with two foreign keys, one to ‘department.id’ and another to ‘employee.id’.

```
>>> from sqlalchemy import Column, String, Integer, ForeignKey
>>> from sqlalchemy.orm import relationship, backref
>>> from sqlalchemy.ext.declarative import declarative_base
>>>
>>>
>>> Base = declarative_base()
>>>
>>>
>>> class Department(Base):
...     __tablename__ = 'department'
...     id = Column(Integer, primary_key=True)
...     name = Column(String)
...     employees = relationship('Employee', secondary='department_employee')
...
>>>
>>> class Employee(Base):
...     __tablename__ = 'employee'
...     id = Column(Integer, primary_key=True)
...     name = Column(String)
...     departments = relationship('Department', secondary='department_employee')
...
>>>
>>> class DepartmentEmployee(Base):
...     __tablename__ = 'department_employee'
...     department_id = Column(Integer, ForeignKey('department.id'), primary_key=True)
...     employee_id = Column(Integer, ForeignKey('employee.id'), primary_key=True)
...
>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite:///')
>>> from sqlalchemy.orm import sessionmaker
>>> session = sessionmaker()
>>> session.configure(bind=engine)
>>> Base.metadata.create_all(engine)
>>>
>>> s = session()
>>> john = Employee(name='john')
>>> s.add(john)
>>> it_department = Department(name='IT')
>>> it_department.employees.append(john)
>>> s.add(it_department)
>>> s.commit()
```

now change:

```
from sqlalchemy import Column, DateTime, String, Integer, ForeignKey, func
from sqlalchemy.orm import relationship, backref
from sqlalchemy.ext.declarative import declarative_base
Base = declarative_base()
class Department(Base):
    __tablename__ = 'department'
    id = Column(Integer, primary_key=True)
    name = Column(String)


class Employee(Base):
    __tablename__ = 'employee'
    id = Column(Integer, primary_key=True)
    name = Column(String)
    # Use default=func.now() to set the default hiring time
    # of an Employee to be the current time when an
    # Employee record was created
    hired_on = Column(DateTime, default=func.now())
    department_id = Column(Integer, ForeignKey('department.id'))
    # Use cascade='delete,all' to propagate the deletion of a Department onto its Employe
es
    department = relationship(
        Department,
        backref=backref('employees',
                        uselist=True,
                        cascade='delete,all'))


from sqlalchemy import create_engine
engine = create_engine('sqlite:///orm_in_detail.sqlite')

from sqlalchemy.orm import sessionmaker
session = sessionmaker()
session.configure(bind=engine)
Base.metadata.create_all(engine)
```

> Notice we made two changes to the employee table: 1. we inserted a new column 'hired_on' which is a DateTime column that stores when the employee was hired and, 2. **we inserted a keyword argument 'cascade' with a value 'delete,all' to the backref of the relationship Employee.department. The cascade allows SQLAlchemy to automatically delete a department's employees when the department itself is deleted.**

SQLAlchemy in batches: Generating a top playlist

we have been using SQLAlchemy for quite a period of time and appreciated the flexibility and elegance it provides over the Data Mapper abstraction. No doubt, it works very well for modern web applications but what about long-running background jobs? Would the abstraction get in your ways?
(在跑长期后台任务的时候)
We built a popular iOS application with a song recommendation system at the backend. The system suggests a top list for 20 popular songs.
（e.g例如展示用户推荐前topN列表的时候）

- Get the top 20 songs (across 100k+ analyzed songs)
- Create a new playlist and insert it into the database
- Run the above script weekly

```
  def gen_autotrend_playlist(playlistdate):
        r = get_redis()
        song_ids = get_trending_song_id(r, playlistdate)

        if not song_ids:
            return

        new_playlist = Playlist(name='Top Charts',
                                ordering=1,
                                available=0,
                                admin_available=1,
                                slug='topchart-week-' + playlistdate)
        db.session.add(new_playlist)
        db.session.commit()

        for i, song_id in enumerate(song_ids):
            new_entry = ListedEntry(song_id=song_id,
                                    playlist_id=new_playlist.id,
                                    ordering=i,
                                    is_top=1)
            db.session.add(new_entry)
            db.session.commit()
```

The code speaks for itself very well: crawl the most trendy songs from Redis and store them into Database one-by-one.
If the operation crashes in the middle of the loop, say, the 5th song, the script will throw an exception and exit.
（存在的问题：万一在第五首哥的时候 挂了。。。）
Boom! Your supposedly TopK-song playlist now gets only 4 songs.

```
def gen_autotrend_playlist(playlistdate):
    r = get_redis()
    song_ids = get_trending_song_id(r, playlistdate)

if not song_ids:
    return

new_playlist = Playlist(name='Top Charts',
                        ordering=1,
                        available=0,
                        admin_available=1,
                        slug='topchart-week-' + playlistdate)
db.session.add(new_playlist)
db.session.flush()

for i, song_id in enumerate(song_ids):
    new_entry = ListedEntry(song_id=song_id,
                            playlist_id=new_playlist.id,
                            ordering=i,
                            is_top=1)
    db.session.add(new_entry)

db.session.commit() # Only commit when everything is done correctly
```

也就是说在所有的new_entry都创建成功的时候才进行commit

SQLAlchemy commit(), flush(), expire(), refresh(), merge() - what’s the difference?

A Session object is basically an ongoing transaction of changes to a database (update, insert, delete).These operations aren't persisted to the database until they are committed,The session object registers transaction operations with session.add(), but doesn't yet communicate them to the database until session.flush() is called.

When you use a Session object to query the database, the query will return results both from the database and from the flushed parts of the uncommitted transaction it holds. By default, Session objects autoflush their operations, but this can be disabled.

```
#---
s = Session()

s.add(Foo('A')) # The Foo('A') object has been added to the session.
                # It has not been committed to the database yet,
                #   but is returned as part of a query.
print 1, s.query(Foo).all()
s.commit()

#---
s2 = Session()
s2.autoflush = False

s2.add(Foo('B'))
print 2, s2.query(Foo).all() # The Foo('B') object is *not* returned
                             #   as part of this query because it hasn't
                             #   been flushed yet.
s2.flush()                   # Now, Foo('B') is in the same state as
                             #   Foo('A') was above.
print 3, s2.query(Foo).all()
s2.rollback()                # Foo('B') has not been committed, and rolling
                             #   back the session's transaction removes it
                             #   from the session.
print 4, s2.query(Foo).all()

#---
Output:
1 [<Foo('A')>]
2 [<Foo('A')>]
3 [<Foo('A')>, <Foo('B')>]
4 [<Foo('A')>]
```