数据库orm 学习

**SQLAlchemy ORM教程之一：Create**

2017年04月15日 14:43:39 [krzover](https://me.csdn.net/krzover) 阅读数 767 标签： [python](https://so.csdn.net/so/search/s.do?q=python&t=blog)[orm](https://so.csdn.net/so/search/s.do?q=orm&t=blog)[mysql](https://so.csdn.net/so/search/s.do?q=mysql&t=blog)[sql](https://so.csdn.net/so/search/s.do?q=sql&t=blog)[SQLAlchemy](https://so.csdn.net/so/search/s.do?q=SQLAlchemy&t=blog) 更多

个人分类： [Sever&Database](https://blog.csdn.net/krzover/article/category/6797467)

近日想要学习SQLAlchemy，好像没有发现很好的中文版的Tutorial，所以我这里将看英文版的Tutorial的过程记录下来，权当笔记与复习。最终本文是基于[SQLAlchemy的官方教程](http://docs.sqlalchemy.org/en/rel_1_0/orm/tutorial.html#working-with-related-objects)整理而来。

教程一共分三部分:

[SQLAlchemy ORM教程之一:Create](http://blog.csdn.net/krzover/article/details/70184468)    [SQLAlchemy ORM教程之二:Query](http://blog.csdn.net/krzover/article/details/70184480)   [SQLAlchemy ORM教程之三:Relationship](http://blog.csdn.net/krzover/article/details/70184486)

**Object Relational Tutorial**

所谓ORM（Object Relational Mapping），就是建立其由Python类到数据库表的映射关系：一个Python实例(instance)对应数据库中的一行(row)。这种映射包含两层含义，一是实现对象和与之关联的的行的状态同步，二是将涉及数据库的查询操作，表达为Python类的相互关系。

注意ORM和SQLAlchemy的Expression Language不同。后者可以视为对原始SQL的封装。ORM是基于Expression Language而构建的，其抽象层次要高于Expression Language。很多时候我们都是使用ORM，有时需要一些高度定制化的功能时，就需要使用到Expression Language。

下面就是教程的正文了。

**Version Check**

注意这里的教程都是基于1.0版本的，你可以同下面的命令来查看SQLAlchemy的版本：

1. >>> import sqlalchemy
2. >>> sqlalchemy.\_\_version\_\_\_
3. 1.0.0

**Connecting**

在这个教程中我们使用in-memory的SQLite数据库，你也可以根据自己的需要配置对应的数据库设置。为了建立同数据库的链接，我们需要使用到create\_engine

1. >>> from sqlalchemy import create\_engine
2. >>> engine = create\_engine('sqlite:///:memory:', echo=True)

这里的echo设置为True可以使得后面我们可以在控制台看到操作涉及的SQL语言。如果你觉得麻烦，可以将其设置为False。我们这里就不贴出了。

create\_engine返回的是一个Engine实例，它代表了指向数据库的一些非常核心的接口。他会根据你选择的数据库配置而调用对应的DBAPI。

当第一次如Engine.execute()或者Engine.connect()的方法被调用时，Engine才会真正的建立起到数据库的DBAPI连接。实际上，我们一般并不会直接使用Engine。

**Declaring a Mapping**

当我们使用ORM的时候，其配置过程主要分为两个部分：一是描述我们要处理的数据库表的信息，二是将我们的Python类映射到这些表上。这两个过程在SQLAlchemy中是一起完成的，我们将这个过程称之为**Declarative**。

使用Declarative参与ORM映射的类需要被定义成为一个指定基类的子类，这个基类应当含有ORM映射中相关的类和表的信息。这样的基类我们称之为**declarative base class**。在我们的应用中，我们一般只需要一个这样的基类。这个基类我们可以通过declarative\_base来创建

1. >>> from sqlalchemy.ext.declarative import declarative\_base
3. >>> Base = declarative\_base()

现在我们已经有了一个基类，我们可以基于这个基类来创建我们的自定义类了。我们以建立一个用户类为例子。从Base派生一个名为User的类，在这个类里面我们可以定义将要映射到数据库的表上的属性（主要是表的名字，列的类型和名称等）：

1. >>> from sqlalchemy import Column, Integer, String
2. >>> class User(Base):
3. ... \_\_tablename\_\_ = 'users'
4. ...
5. ... id = Column(Integer, primary\_key=True)
6. ... name = Column(String)
7. ... fullname = Column(String)
8. ... password = Column(String)
9. ...
10. ... def \_\_repr\_\_(self):
11. ... return "<User(name='%s', fullname='%s', password='%s')>" % (
12. ... self.name, self.fullname, self.password)

通过Declarative生成的类至少应该包含一个名为**tablename**的属性来给出目标表的名称，以及至少一个Column来给出表的主键(Primary Key)。SQLAlchemy不会对于类名和表名之间的关联做任何假设，也不会自动涉及数据类型以及约束的转换。一般的你可以自己创建一个模板来建立这些自动转换，这样可以减少你的很多重复劳动。

当我们的类声明完成后，Declarative将会将所有的Column成员替换成为特殊的Python访问器(accessors)，我们称之为**descriptors**。这个过程我们称为**instrumentation**，经过instrumentation的映射类可以让我们能够读写数据库的表和列。

注意除了这些涉及ORM的映射意外，这些mapping类的其他部分仍然是不变的。

**Create a schema**

我们通过Declarative系统构建好我们的User类之后，与之同时的关于表的信息也已经创建好了，我们称之为**table metadata**。描述这些信息的类为Table。我们可以通过\_\_table\_\_这个类变量来查看表信息

1. >>> User.\_\_table\_\_
2. Table('users', MetaData(bind=None),
3. Column('id', Integer(), table=<users>, primary\_key=True, nullable=False),
4. Column('name', String(), table=<users>),
5. Column('fullname', String(), table=<users>),
6. Column('password', String(), table=<users>), schema=None)

当我们完成类声明时，Declarative用一个Python的metaclass来为这个类进行了加工。在这个阶段，它依据我们给出的设置创建了Table对象，然后构造一个Mapper对象来与之关联。这些幕后的对象我们大多都不需要直接与之打交道。

Table对象是一个更大家庭----我们称之为MetaData----的一部分。当我们使用Declarative时，这个对象也可以在Declarative base class的.metadata属性中看到。

MetaData是我们与数据库打交道的一个接口。对于我们的SQLite数据库而言，此时还没有一个名为users的表的存在，我们需要使用MetaData来发出CREATE TABLE的命令。下面我们使用MetaData.create\_all()指令，将我们上面得到的Engine作为参数传入。如果你上面设置了echo为True的话，应该可以看到这一过程中的SQL指令。首先检查了users表的存在性，如果不存在的话会执行表的创建工作。

1. >>> Base.metadata.create\_all(engine)
2. SELECT ...
3. PRAGMA table\_info("users")
4. ()
5. CREATE TABLE users (
6. id INTEGER NOT NULL, name VARCHAR,
7. fullname VARCHAR,
8. password VARCHAR,
9. PRIMARY KEY (id)
10. )
11. ()
12. COMMIT

**Create an Instance of the Mapped Class**

创建User对象十分简单

1. >>> ed\_user = User(name='ed', fullname='Ed Jones', password='edspassword')
2. >>> ed\_user.name
3. 'ed'
4. >>> ed\_user.password
5. 'edspassword'
6. >>> str(ed\_user.id)
7. 'None'

**Create a Session**

Session是一个非常重要的概念，类似于iOS中的NSManagedContext的概念，我也在尝试进一步去理解它。

我们现在可以和数据库对话了。ORM对数据库的入口即是Session，当我们构建应用时，和create\_engine的同一级别下，我们定义一个Session类来作为生成新的Session的Factory类

1. >>> from sqlalchemy.orm import sessionmaker
2. >>> Session = sessionmaker(bind=engine)

当你试图在定义Engine之前定义Sesssion的话，这里的bind可以不设置

>>> Session = sessionmaker()

后续你定义好Engine后可以通过configure()来将其连接到Session

>>> Session.configure(bind=engine) *# once engine is available*

这个我们自定义的工厂类就可以拿来我们构造新的Session了。

session = Session()

上面的Session已经和我们的SQLite的数据库的Engine关联起来了，但是我们可以发现它还没有打开任何到数据库的连接(connection)。当一个Session被首次使用时，它会从Engine所维护的连接池中取出一个连接来操作数据库。这个连接在我们应用有所更改或者关闭Session时会被释放。

**Adding and Update Objects**

为了将User对象存入数据库，我们调用Sesson的add()函数

1. >>> ed\_user = User(name='ed', fullname='Ed Jones', password='edspassword')
2. >>> session.add(ed\_user)

当这个操作完成之后，我们成这个User实例的状态为**pending**。目前实际上还没有执行SQL操作，也就是说数据库中还没有产生和这个User实例对应的行。Session将会在需要的时候执行相应的SQL命令，这个过程我们称之为**flush**。如果我们试图查询Ed Jones，所有处于pending状态的信息将会首先被**flush**，然后负责进行查询的SQL语言在此之后立即被执行。

例如，我们创建一个查询来获取刚刚我们创建的用户（涉及查询的部分我们后续会详细介绍）。这个查询会返回一个和我们之前添加的用户相同的用户实例。

1. >>> our\_user = session.query(User).filter\_by(name='ed').first() BEGIN (implicit)
2. INSERT INTO users (name, fullname, password) VALUES (?, ?, ?)
3. ('ed', 'Ed Jones', 'edspassword')
4. SELECT users.id AS users\_id,
5. users.name AS users\_name,
6. users.fullname AS users\_fullname,
7. users.password AS users\_password
8. FROM users
9. WHERE users.name = ?
10. LIMIT ? OFFSET ?
11. ('ed', 1, 0)
12. >>> our\_user
13. <User(name='ed', fullname='Ed Jones', password='edspassword')>

事实上这里的Session判断出来了需要返回的行和已经存在内存中的一个映射实例应当是同一个，所以我们会得到一个和之前完全相同的实例

1. >>> ed\_user is our\_user
2. True

这里ORM所表现的理念，我们称之为[identity map](http://docs.sqlalchemy.org/en/rel_1_0/glossary.html#term-identity-map)。这个设计理念保证了在一个Session对于一个制定行的操作，作用于同一个内存实例上。当一个拥有特定主键的对象出现在Session中时，所有的查询操作对这个主键都会返回一个相同的Python对象。并且，如果你试图引入重复了主键的新的对象时，系统会产生一个错误来阻止你的操作。

我们可以通过add\_all()来一次加入多个对象

1. >>> session.add\_all([
2. ... User(name='wendy', fullname='Wendy Williams', password='foobar'),
3. ... User(name='mary', fullname='Mary Contrary', password='xxg527'),
4. ... User(name='fred', fullname='Fred Flinstone', password='blah')])

并且，如果我们希望改变Ed的密码，可以直接修改之：

>>> ed\_user.password = 'f8s7ccs'

这个修改会被Session记录下来

1. >>> session.dirty
2. IdentitySet([<User(name='ed', fullname='Ed Jones', password='f8s7ccs')>])

当然，上面的插入操作也被记录了

1. >>> session.new
2. IdentitySet([<User(name='wendy', fullname='Wendy Williams', password='foobar')>,
3. <User(name='mary', fullname='Mary Contrary', password='xxg527')>,
4. <User(name='fred', fullname='Fred Flinstone', password='blah')>])

我们可以使用commit()命令来将这些更改**flush**到数据库中。

>>> session.commit()

**SQLAlchemy ORM教程之二：Query**

2017年04月15日 14:45:31 [krzover](https://me.csdn.net/krzover) 阅读数 3836

教程一共分三部分:

[SQLAlchemy ORM教程之一:Create](http://blog.csdn.net/krzover/article/details/70184468)    [SQLAlchemy ORM教程之二:Query](http://blog.csdn.net/krzover/article/details/70184480)   [SQLAlchemy ORM教程之三:Relationship](http://blog.csdn.net/krzover/article/details/70184486)

在上一篇中我们主要是解决了如何配置ORM系统，建立从类到表的映射的过程，以及如何插入和修改记录。在这个教程中我们主要解决使用的问题。

**Query**

Session的query函数会返回一个Query对象。query函数可以接受多种参数类型。可以是类，或者是类的instrumented **descriptor**。下面的这个例子取出了所有的User记录。

1. >>> for instance in session.query(User).order\_by(User.id):
2. ... print(instance.name, instance.fullname)
3. ed Ed Jones
4. wendy Wendy Williams
5. mary Mary Contrary
6. fred Fred Flinstone

Query也接受ORM-instrumented descriptors作为参数。当多个参数传入时，返回结果为以同样顺序排列的tuples

1. >>> for name, fullname in session.query(User.name, User.fullname):
2. ... print(name, fullname)
3. ed Ed Jones
4. wendy Wendy Williams
5. mary Mary Contrary
6. fred Fred Flinstone

Query返回的tuples由KeyedTuple这个类提供，其成员除了用下标访问意外，还可以视为实例变量来获取。对应的变量的名称与被查询的类变量名称一样，如下例：

1. >>> for row in session.query(User, User.name).all():
2. ... print(row.User, row.name)
3. <User(name='ed', fullname='Ed Jones', password='f8s7ccs')> ed
4. <User(name='wendy', fullname='Wendy Williams', password='foobar')> wendy
5. <User(name='mary', fullname='Mary Contrary', password='xxg527')> mary
6. <User(name='fred', fullname='Fred Flinstone', password='blah')> fred

你可以通过label()来制定descriptor对应实例变量的名称

1. >>> for row in session.query(User.name.label('name\_label')).all():
2. ... print(row.name\_label)
3. ed
4. wendy
5. mary
6. fred

而对于类参数而言，要实现同样的定制需要使用aliased

1. >>> from sqlalchemy.orm import aliased
2. >>> user\_alias = aliased(User, name='user\_alias')
4. SQL>>> for row in session.query(user\_alias, user\_alias.name).all():
5. ... print(row.user\_alias)
6. <User(name='ed', fullname='Ed Jones', password='f8s7ccs')>
7. <User(name='wendy', fullname='Wendy Williams', password='foobar')>
8. <User(name='mary', fullname='Mary Contrary', password='xxg527')>
9. <User(name='fred', fullname='Fred Flinstone', password='blah')>

基本的查询操作除了上面这些之外，还包括OFFSET和LIMIT，这个可以通过Python的array slice来完成。

1. >>> for u in session.query(User).order\_by(User.id)[1:3]:
2. ... print(u)
3. <User(name='wendy', fullname='Wendy Williams', password='foobar')>
4. <User(name='mary', fullname='Mary Contrary', password='xxg527')>

上述过程实际上只涉及了整体取出的操作，而没有进行筛选，筛选常用的函数是filter\_by和filter。其中后者比起前者要更灵活一些，你可以在后者的参数中使用python的运算符。

1. >>> for name, in session.query(User.name).\
2. ... filter\_by(fullname='Ed Jones'):
3. ... print(name)
4. ed
5. >>> for name, in session.query(User.name).\
6. ... filter(User.fullname=='Ed Jones'):
7. ... print(name)
8. ed

注意Query对象是**generative**的，这意味你可以把他们串接起来调用，如下：

1. >>> for user in session.query(User).\
2. ... filter(User.name=='ed').\
3. ... filter(User.fullname=='Ed Jones'):
4. ... print(user)
5. <User(name='ed', fullname='Ed Jones', password='f8s7ccs')>

串接的filter之间是**与**的关系。

**常用的filter操作符**

下面的这些操作符可以应用在filter函数中

* equals:

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

* not equals:

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

* LIKE:

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

* IN:

1. query.filter(User.name.in\_(['ed', 'wendy', 'jack']))
3. *# works with query objects too:*
4. query.filter(User.name.in\_(
5. session.query(User.name).filter(User.name.like('%ed%'))
6. ))

* NOT IN:

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

* IS NULL:

1. query.filter(User.name == None)
3. *# alternatively, if pep8/linters are a concern*
4. query.filter(User.name.is\_(None))

* IS NOT NULL:

1. query.filter(User.name != None)
3. *# alternatively, if pep8/linters are a concern*
4. query.filter(User.name.isnot(None))

* AND:

1. *# use and\_()*
2. from sqlalchemy import and\_
3. query.filter(and\_(User.name == 'ed', User.fullname == 'Ed Jones'))
5. *# or send multiple expressions to .filter()*
6. query.filter(User.name == 'ed', User.fullname == 'Ed Jones')
8. *# or chain multiple filter()/filter\_by() calls*
9. query.filter(User.name == 'ed').filter(User.fullname == 'Ed Jones')

* OR:

1. from sqlalchemy import or\_
2. query.filter(or\_(User.name == 'ed', User.name == 'wendy'))

* MATCH:

query.filter(User.name.match('wendy'))

**返回列表(List)和单项(Scalar)**

很多Query的方法执行了SQL命令并返回了取出的数据库结果。

* all()返回一个列表:
  1. >>> query = session.query(User).filter(User.name.like('%ed')).order\_by(User.id)
  2. SQL>>> query.all()
  3. [<User(name='ed', fullname='Ed Jones', password='f8s7ccs')>,
  4. <User(name='fred', fullname='Fred Flinstone', password='blah')>]
* first()返回至多一个结果，而且以单项形式，而不是只有一个元素的tuple形式返回这个结果.

1. >>> query.first()
2. <User(name='ed', fullname='Ed Jones', password='f8s7ccs')>

* one()返回且仅返回一个查询结果。当结果的数量不足一个或者多于一个时会报错。

1. >>> user = query.one()
2. Traceback (most recent call last):
3. ...
4. MultipleResultsFound: Multiple rows were found for one()

没有查找到结果时：

1. >>> user = query.filter(User.id == 99).one()
2. Traceback (most recent call last):
3. ...
4. NoResultFound: No row was found for one()

* one\_or\_none()：从名称可以看出，当结果数量为0时返回None， 多于1个时报错
* scalar()和one()类似，但是返回单项而不是tuple

**嵌入使用SQL**

你可以在Query中通过text()使用SQL语句。例如：

1. >>> from sqlalchemy import text
2. >>> for user in session.query(User).\
3. ... filter(text("id<224")).\
4. ... order\_by(text("id")).all():
5. ... print(user.name)
6. ed
7. wendy
8. mary
9. fred

除了上面这种直接将参数写进字符串的方式外，你还可以通过params()方法来传递参数

1. >>> session.query(User).filter(text("id<:value and name=:name")).\
2. ... params(value=224, name='fred').order\_by(User.id).one()
3. <User(name='fred', fullname='Fred Flinstone', password='blah')>

并且，你可以直接使用完整的SQL语句，但是要注意将表名和列明写正确。

1. >>> session.query(User).from\_statement(
2. ... text("SELECT \* FROM users where name=:name")).\
3. ... params(name='ed').all()
4. [<User(name='ed', fullname='Ed Jones', password='f8s7ccs')>]

**计数**

Query定义了一个很方便的计数函数count()

1. >>> session.query(User).filter(User.name.like('%ed')).count()
2. SELECT count(\*) AS count\_1
3. FROM (SELECT users.id AS users\_id,
4. users.name AS users\_name,
5. users.fullname AS users\_fullname,
6. users.password AS users\_password
7. FROM users
8. WHERE users.name LIKE ?) AS anon\_1
9. ('%ed',)
10. 2

注意上面我们同时列出了实际的SQL指令。在SQLAlchemy中，我们总是将被计数的查询打包成一个子查询，然后对这个子查询进行计数。即便是最简单的SELECT count(\*) FROM table，也会如此处理。为了更精细的控制计数过程，我们可以采用func.count()这个函数。

1. >>> from sqlalchemy import func
2. SQL>>> session.query(func.count(User.name), User.name).group\_by(User.name).all()
3. SELECT count(users.name) AS count\_1, users.name AS users\_name
4. FROM users GROUP BY users.name
5. ()
6. [(1, u'ed'), (1, u'fred'), (1, u'mary'), (1, u'wendy')]

为了实现最简单的SELECT count(\*) FROM table，我们可以如下调用

1. >>> session.query(func.count('\*')).select\_from(User).scalar()
2. SELECT count(?) AS count\_1
3. FROM users
4. ('\*',)
5. 4

如果我们对User的主键进行计数，那么select\_from也可以省略。

1. >>> session.query(func.count(User.id)).scalar()
2. SELECT count(users.id) AS count\_1
3. FROM users
4. ()
5. 4

在下一篇教程里面我们将会介绍SQLAlchemy对于『关系』的处理方式，以及针对关系的更加复杂的查询。

# SQLAlchemy ORM教程之三：Relationship

2017年04月15日 14:46:57 [krzover](https://me.csdn.net/krzover) 阅读数 1947

教程一共分三部分:

[SQLAlchemy ORM教程之一:Create](http://blog.csdn.net/krzover/article/details/70184468)    [SQLAlchemy ORM教程之二:Query](http://blog.csdn.net/krzover/article/details/70184480)   [SQLAlchemy ORM教程之三:Relationship](http://blog.csdn.net/krzover/article/details/70184486)

『关系』是关系型数据库的一大特色，也是我们在建模过程中的一个重要的抽象过程。在前面的两个教程中，我们分别回顾了使用SQLAlchemy在数据库中进行[创建](http://blog.csdn.net/krzover/article/details/70184468)和[简单查询](http://blog.csdn.net/krzover/article/details/70184480)的方法，今天我们来深入到更为复杂和抽象部分。

### 建立关系

之前我们已经建立了一个用户(User)表，现在我们来考虑增加一个与用户关联的新的表。在我们的系统里面，用户可以存储多个与之相关的email地址。这是一种基本的一对多的关系。我们把这个新增加的存储email地址的表称为addresses。应用Declarative，我们按照如下方式定义这个新表：

1. >>> from sqlalchemy import ForeignKey
2. >>> from sqlalchemy.orm import relationship
4. >>> class Address(Base):
5. ... \_\_tablename\_\_ = 'addresses'
6. ... id = Column(Integer, primary\_key=True)
7. ... email\_address = Column(String, nullable=False)
8. ... user\_id = Column(Integer, ForeignKey('users.id'))
9. ...
10. ... user = relationship("User", back\_populates="addresses")
11. ...
12. ... def \_\_repr\_\_(self):
13. ... return "<Address(email\_address='%s')>" % self.email\_address
15. >>> User.addresses = relationship(
16. ... "Address", order\_by=Address.id, back\_populates="user")

上面的代码中我们使用了一个新的名为ForeignKey的构造。其含义为，其所在的列的值域应当被限制在另一个表的指定列的取值范围之类。这一特性是关系型数据库的核心特性之一。就上例而言，addresses.user\_id这一列的取值范围，应当包含在users.id的取值范围之内。

除了ForeignKey之外，我们还引入了一个relationship，来告诉ORM，Address类需要被连接到User类。relationship和ForeignKey这个两个属性决定了表之间关系的属性，决定了这个关系是多对一的。

在完成对Address类的声明之后，我们还定义另一个relationship，将其赋值给了User.addresses。在两个relationship中，我们都有传入了一个relationship.back\_populates的属性来为反向关系所对应的属性进行命名。（作者：到这里为止，看来SQLAlchemy中定义关系要比Django的ORM要麻烦许多。Django中只需要一行就可以了。而且这里的两个relationship的定义明显是冗余的）

多对一的关系的反向永远都是一对多的关系。关于更多的relationship()的配置方法，可以参见这个链接[Basic Relationship Patterns](http://docs.sqlalchemy.org/en/rel_1_0/orm/basic_relationships.html#relationship-patterns)。

上述我们定义的两个互补的关系Address.user和User.addresses被称为双向关系([bidirectional relationship](http://docs.sqlalchemy.org/en/rel_1_0/glossary.html#term-bidirectional-relationship))，这是SQLAlchemy的核心特性这一。

relationship()的参数配置中指向被连接的类的字符串，可以指向工程中任何位置所定义的，基于declarative base的类，而无先后之分。Declarative会在完成所有的映射以后的将这些字符串转换为适当的、实际使用的参数形式。

### 使用关联对象

现在，当我们创建一个User实例的时候，会同时创建一个空的addresses的collection。这个collection可能是多种类型，如list, set, 或是dictionary。默认情况下，其应当为一个Python列表。

1. >>> jack = User(name='jack', fullname='Jack Bean', password='gjffdd')
2. >>> jack.addresses
3. []

此时你可以自由的向这个列表里面插入User对象。

1. >>> jack.addresses = [
2. ... Address(email\_address='jack@google.com'),
3. ... Address(email\_address='j25@yahoo.com')]

当使用bidirectional relationship时，通过其中一个方向的关系（如上例）会自动出现在另一个方向的关系上。

1. >>> jack.addresses[1]
2. <Address(email\_address='j25@yahoo.com')>
4. >>> jack.addresses[1].user
5. <User(name='jack', fullname='Jack Bean', password='gjffdd')>

让我们把jack添加进入Session。

1. >>> session.add(jack)
2. >>> session.commit()
3. INSERT INTO users (name, fullname, password) VALUES (?, ?, ?)
4. ('jack', 'Jack Bean', 'gjffdd')
5. INSERT INTO addresses (email\_address, user\_id) VALUES (?, ?)
6. ('jack@google.com', 5)
7. INSERT INTO addresses (email\_address, user\_id) VALUES (?, ?)
8. ('j25@yahoo.com', 5)
9. COMMIT

可以发现上面执行了三个INSERT命令，也就是说与jack关联的两个Address对象也被提交了。现在我们通过查询来取出jack。

1. >>> jack = session.query(User).\
2. ... filter\_by(name='jack').one()
3. BEGIN (implicit)
4. SELECT users.id AS users\_id,
5. users.name AS users\_name,
6. users.fullname AS users\_fullname,
7. users.password AS users\_password
8. FROM users
9. WHERE users.name = ?
10. ('jack',)
12. >>> jack
13. <User(name='jack', fullname='Jack Bean', password='gjffdd')>

可以发现目前只有针对User表的查询，而没有对Address表的查询。此时访问addresses属性，相关的SQL才会执行

1. >>> jack.addresses
2. SELECT addresses.id AS addresses\_id,
3. addresses.email\_address AS
4. addresses\_email\_address,
5. addresses.user\_id AS addresses\_user\_id
6. FROM addresses
7. WHERE ? = addresses.user\_id ORDER BY addresses.id
8. (5,)
9. [<Address(email\_address=*'jack@google.com')>, <Address(email\_address='j25@yahoo.com')>]*

上面这种方式我们称之为[lazy loading](http://docs.sqlalchemy.org/en/rel_1_0/glossary.html#term-lazy-loading)。

### 使用join进行查询

现在我们有了两会在那个彼此关联的数据表了，相比与上一篇教程中的简单查询情况，此时试图对这两张表进行联合查询就更加复杂一些了。关于join技术，读者可以自行阅读[我的前一篇文章](http://www.jianshu.com/p/9e1d3793cba6)。

为了在User和Address之间构造一个简单的join，我们可以通过Query.filter()来连接其相关列（本质是隐式写法的JOIN）。下面是一个简单的例子：

1. >>> for u, a in session.query(User, Address).\
2. ... filter(User.id==Address.user\_id).\
3. ... filter(Address.email\_address=='jack@google.com').\
4. ... all():
5. ... print(u)
6. ... print(a)
7. <User(name='jack', fullname='Jack Bean', password='gjffdd')>
8. <Address(email\_address='jack@google.com')>

而实际的SQL JOIN语法，可以通过Query.join()来想实现

1. >>> session.query(User).join(Address).\
2. ... filter(Address.email\_address=='jack@google.com').\
3. ... all()
4. users.id AS users\_id,
5. users.name AS users\_name,
6. users.fullname AS users\_fullname,
7. users.password AS users\_password
8. FROM users JOIN addresses ON users.id = addresses.user\_id
9. WHERE addresses.email\_address = ?
10. ('jack@google.com',)
11. [<User(name='jack', fullname='Jack Bean', password='gjffdd')>]

在上面的例子中由于只存在一个ForeignKey，Query.join知道如何选取合适的列进行JOIN。如果没有定义ForeignKey，或者存在多个，此时你需要手动指明你参与JOIN的列。Query.join()以如下方式进行：

1. query.join(Address, User.id==Address.user\_id) *# explicit condition*
2. query.join(User.addresses) *# specify relationship from left to right*
3. query.join(Address, User.addresses) *# same, with explicit target*
4. query.join('addresses')

对于OUTER JOIN，只需要使用Query.outerjoin()就可以了。

query.outerjoin(User.addresses) *# LEFT OUTER JOIN*

关于join()更为详细的用法，还是请参考官方的文档[join](http://docs.sqlalchemy.org/en/rel_1_0/orm/query.html#sqlalchemy.orm.query.Query.join)

##### 使用Aliases

当你的查询涉及多个表，而其中同一个表出现了多次时，你需要的为重复的表aliase一个新的名字来避免冲突。这个功能其实我们在上一篇文章里面也提到过，下面是关于aliased的一个例子：

1. >>> from sqlalchemy.orm import aliased
2. >>> adalias1 = aliased(Address)
3. >>> adalias2 = aliased(Address)
4. >>> for username, email1, email2 in \
5. ... session.query(User.name, adalias1.email\_address, adalias2.email\_address).\
6. ... join(adalias1, User.addresses).\
7. ... join(adalias2, User.addresses).\
8. ... filter(adalias1.email\_address=='jack@google.com').\
9. ... filter(adalias2.email\_address=='j25@yahoo.com'):
10. ... print(username, email1, email2)
11. SELECT users.name AS users\_name,
12. addresses\_1.email\_address AS addresses\_1\_email\_address,
13. addresses\_2.email\_address AS addresses\_2\_email\_address
14. FROM users JOIN addresses AS addresses\_1
15. ON users.id = addresses\_1.user\_id
16. JOIN addresses AS addresses\_2
17. ON users.id = addresses\_2.user\_id
18. WHERE addresses\_1.email\_address = ?
19. AND addresses\_2.email\_address = ?
20. ('jack@google.com', 'j25@yahoo.com')
21. jack jack@google.com j25@yahoo.com

###### 使用子查询(Subqueries)

Query适合于用来构造子查询。假如我们想要取出User记录，并且同时计算各个用户的Address的数量。产生这种功能的SQL指令最好的办法是按照user的id分组统计地址的数量，然后join到外层查询。此时我们需要LEFT JOIN，这样可以使得没有地址的用户也会出现在查询结果中（地址数量为0）。 我们期望的SQL命令是这样的：

1. SELECT users.\*, adr\_count.address\_count FROM users LEFT OUTER JOIN
2. (SELECT user\_id, count(\*) AS address\_count
3. FROM addresses GROUP BY user\_id) AS adr\_count
4. ON users.id=adr\_count.user\_id

使用Query，我们可以从内到外来构造上面的语句。

1. >>> from sqlalchemy.sql import func
2. >>> stmt = session.query(Address.user\_id, func.count('\*').\
3. ... label('address\_count')).\
4. ... group\_by(Address.user\_id).subquery()

func我们已经在之前的教程中认识过了。subquery()可以产生一个内嵌了alias（是一个query.statement.alias()）的查询(SELECT)语句的表达。

当我们生成了statement之后，其完全可以视为一个Table来使用。你可以通过c来访问它的属性。

1. >>> for u, count in session.query(User, stmt.c.address\_count).\
2. ... outerjoin(stmt, User.id==stmt.c.user\_id).order\_by(User.id):
3. ... print(u, count)
4. SELECT users.id AS users\_id,
5. users.name AS users\_name,
6. users.fullname AS users\_fullname,
7. users.password AS users\_password,
8. anon\_1.address\_count AS anon\_1\_address\_count
9. FROM users LEFT OUTER JOIN
10. (SELECT addresses.user\_id AS user\_id, count(?) AS address\_count
11. FROM addresses GROUP BY addresses.user\_id) AS anon\_1
12. ON users.id = anon\_1.user\_id
13. ORDER BY users.id
14. ('\*',)
15. <User(name='ed', fullname='Ed Jones', password='f8s7ccs')> None
16. <User(name='wendy', fullname='Wendy Williams', password='foobar')> None
17. <User(name='mary', fullname='Mary Contrary', password='xxg527')> None
18. <User(name='fred', fullname='Fred Flinstone', password='blah')> None
19. <User(name='jack', fullname='Jack Bean', password='gjffdd')> 2

##### 从子查询中取出Entity

在前一个例子中，我们从子查询活着的是一个临时性的JOIN后的表，但是这个表并未定义我们在ORM中定义的Entity。如果我们想将这个临时表映射到ORM中的类呢？此时我们可以使用aliased这个函数来完成这个映射。

1. >>> stmt = session.query(Address).\
2. ... filter(Address.email\_address != 'j25@yahoo.com').\
3. ... subquery()
4. >>> adalias = aliased(Address, stmt)
5. >>> for user, address in session.query(User, adalias).\
6. ... join(adalias, User.addresses):
7. ... print(user)
8. ... print(address)
9. SELECT users.id AS users\_id,
10. users.name AS users\_name,
11. users.fullname AS users\_fullname,
12. users.password AS users\_password,
13. anon\_1.id AS anon\_1\_id,
14. anon\_1.email\_address AS anon\_1\_email\_address,
15. anon\_1.user\_id AS anon\_1\_user\_id
16. FROM users JOIN
17. (SELECT addresses.id AS id,
18. addresses.email\_address AS email\_address,
19. addresses.user\_id AS user\_id
20. FROM addresses
21. WHERE addresses.email\_address != ?) AS anon\_1
22. ON users.id = anon\_1.user\_id
23. ('j25@yahoo.com',)
24. <User(name='jack', fullname='Jack Bean', password='gjffdd')>
25. <Address(email\_address='jack@google.com')>

##### 使用EXISTS

EXISTS关键字是一个BOOL型操作符。当查询结果存在至少一行时返回True。EXISTS可以常常和JOIN搭配使用。

下面是一个显式的EXISTS构造方法：

1. >>> from sqlalchemy.sql import exists
2. >>> stmt = exists().where(Address.user\_id==User.id)
3. >>> for name, in session.query(User.name).filter(stmt):
4. ... print(name)
5. SELECT users.name AS users\_name
6. FROM users
7. WHERE EXISTS (SELECT \*
8. FROM addresses
9. WHERE addresses.user\_id = users.id)
10. ()
11. jack

Query还定义了若干个自动使用了EXISTS的操作。上面的例子可以用any()来完成：

1. >>> for name, in session.query(User.name).\
2. ... filter(User.addresses.any()):
3. ... print(name)
4. SELECT users.name AS users\_name
5. FROM users
6. WHERE EXISTS (SELECT 1
7. FROM addresses
8. WHERE users.id = addresses.user\_id)
9. ()
10. jack

any()也接受筛选条件来限制匹配的行：

1. >>> for name, in session.query(User.name).\
2. ... filter(User.addresses.any(Address.email\_address.like('%google%'))):
3. ... print(name)
4. jack

has()对于的many-to-one的关系，起到的是和any()同样的作用（注意这里~表示NOT）：

1. >>> session.query(Address).\
2. ... filter(~Address.user.has(User.name=='jack')).all()
3. []

##### 常用的关系操作

下面只是简单的列出了一些常用的操作。想要更为详细的了解这些功能，还是推荐去官网的相关文档。

* **eq**() (many-to-one “equals” comparison):

query.filter(Address.user == someuser)

* **ne**() (many-to-one “not equals” comparison):

query.filter(Address.user != someuser)

* IS NULL (many-to-one comparison, also uses **eq**()):

query.filter(Address.user == None)

* contains() (used for one-to-many collections):

query.filter(User.addresses.contains(someaddress))

* any() (used for collections):

1. query.filter(User.addresses.any(Address.email\_address == 'bar'))
3. *# also takes keyword arguments:*
4. query.filter(User.addresses.any(email\_address='bar'))

* has() (used for scalar references):

query.filter(Address.user.has(name='ed'))

* Query.with\_parent() (used for any relationship):

session.query(Address).with\_parent(someuser, 'addresses')

### Eager Loading（找不到合适的翻译）

前面的教程中我们有提及到lazing loading的机制。当我们通过查询取出用户时，与之关联的地址并没有取出来。当我们试图获取User.addresses时，相关的针对地址的SQL查询才起作用。如果你想要减少query的次数的话，就需要使用Eager Loading了。SQLAlchemy提供了三种Eager Loading的方式，其中两种是自动的，而第三种涉及到自定义的筛选条件。所有的这三种Eager Loading方式都会通过调用Query.options()来影响查询的过程，促使Query生成需要的额外配置来取出期望的内容。

##### Subquery Loading

在上面的例子中，我们希望在 取出用户的时候就同步取出对应的地址。此时你们可以此采用orm.subqueryload()。这个函数可以发起第二个SELECT查询来取出与结果相关的另一个表的信息。这里取名为"subquery"的原因是，此处的Query在发起第二个查询时作为子查询而被复用了。详细过程参加下面的程序：

1. >>> from sqlalchemy.orm import subqueryload
2. >>> jack = session.query(User).\
3. ... options(subqueryload(User.addresses)).\
4. ... filter\_by(name='jack').one()
5. SELECT users.id AS users\_id,
6. users.name AS users\_name,
7. users.fullname AS users\_fullname,
8. users.password AS users\_password
9. FROM users
10. WHERE users.name = ?
11. ('jack',)
12. SELECT addresses.id AS addresses\_id,
13. addresses.email\_address AS addresses\_email\_address,
14. addresses.user\_id AS addresses\_user\_id,
15. anon\_1.users\_id AS anon\_1\_users\_id
16. FROM (SELECT users.id AS users\_id
17. FROM users WHERE users.name = ?) AS anon\_1
18. JOIN addresses ON anon\_1.users\_id = addresses.user\_id
19. ORDER BY anon\_1.users\_id, addresses.id
20. ('jack',)
21. >>> jack
22. <User(name='jack', fullname='Jack Bean', password='gjffdd')>
24. >>> jack.addresses
25. [<Address(email\_address='jack@google.com')>, <Address(email\_address='j25@yahoo.com')>]

**注意**：当subqueryload()和涉及limiting的函数一起使用的时候（如Query.first(), Query.limit(), Query.offset()等），应当加上一个以Unique的行作为参数的Query.order\_by()`来确保结果的正确性。详情参见[The importance of Ordering](http://docs.sqlalchemy.org/en/rel_1_0/orm/loading_relationships.html#subqueryload-ordering)

##### Joined Load

这种自动Eager Loading的方式要更为常用一些。Joined Loading发起了一个JOIN（默认是LEFT OUTER JOIN），故而查询结果和制定的与之关联的行可以被同时取出。我们这里以和上面的Subquery Loading中同样的查询目的为例。

1. >>> from sqlalchemy.orm import joinedload
3. >>> jack = session.query(User).\
4. ... options(joinedload(User.addresses)).\
5. ... filter\_by(name='jack').one()
6. SELECT users.id AS users\_id,
7. users.name AS users\_name,
8. users.fullname AS users\_fullname,
9. users.password AS users\_password,
10. addresses\_1.id AS addresses\_1\_id,
11. addresses\_1.email\_address AS addresses\_1\_email\_address,
12. addresses\_1.user\_id AS addresses\_1\_user\_id
13. FROM users
14. LEFT OUTER JOIN addresses AS addresses\_1 ON users.id = addresses\_1.user\_id
15. WHERE users.name = ? ORDER BY addresses\_1.id
16. ('jack',)
18. >>> jack
19. <User(name='jack', fullname='Jack Bean', password='gjffdd')>
21. >>> jack.addresses
22. [<Address(email\_address='jack@google.com')>, <Address(email\_address='j25@yahoo.com')>]

注意到，如果你是在命令行运行了前一个Subquery Loading的例子的话，在这里jack的addresses实际上已经填充了的，但是这里的Joined Load仍然是会发起JOIN。另外，LEFT OUTER JOIN指令实际上有可能导致重复的User出现，但是在结果中实际得到的User却不会重复。这是因为Query实际上是基于Object Identity采用了一种"uniquing"的策略。

历史上来看joinedload()出现的更早一些。joinedloading()更加适合于处理Many-to-one的关系。

##### 显式的Join + EagerLoad

第三种方式我们是我们自己显式的调用join来定位JOIN连接主键，并接着关联表的信息填充到查询结果中对应对象或者列表中。这个特性需要使用到orm.contains\_eager()函数。这个机制最典型的用途是pre-loading many-to-one关系，同时添加对这个关系的筛选。我们用下面的这个例子来阐述说明上面这些比较绕的话。假设我们需要筛选出用户的名字为jack的邮件地址，进行这个查询的方法如下：

1. >>> from sqlalchemy.orm import contains\_eager
2. >>> jacks\_addresses = session.query(Address).\
3. ... join(Address.user).\
4. ... filter(User.name=='jack').\
5. ... options(contains\_eager(Address.user)).\
6. ... all()
7. SELECT users.id AS users\_id,
8. users.name AS users\_name,
9. users.fullname AS users\_fullname,
10. users.password AS users\_password,
11. addresses.id AS addresses\_id,
12. addresses.email\_address AS addresses\_email\_address,
13. addresses.user\_id AS addresses\_user\_id
14. FROM addresses JOIN users ON users.id = addresses.user\_id
15. WHERE users.name = ?
16. ('jack',)
18. >>> jacks\_addresses
19. [<Address(email\_address='jack@google.com')>, <Address(email\_address='j25@yahoo.com')>]
21. >>> jacks\_addresses[0].user
22. <User(name='jack', fullname='Jack Bean', password='gjffdd')>

### 关系中的删除问题

沃恩尝试删除jack，来看结果：

1. >>> session.delete(jack)
2. >>> session.query(User).filter\_by(name='jack').count()
3. UPDATE addresses SET user\_id=? WHERE addresses.id = ?
4. ((None, 1), (None, 2))
5. DELETE FROM users WHERE users.id = ?
6. (5,)
7. SELECT count(\*) AS count\_1
8. FROM (SELECT users.id AS users\_id,
9. users.name AS users\_name,
10. users.fullname AS users\_fullname,
11. users.password AS users\_password
12. FROM users
13. WHERE users.name = ?) AS anon\_1
14. ('jack',)
15. 0

那么与jack关联的地址呢？

1. >>> session.query(Address).filter(
2. ... Address.email\_address.in\_(['jack@google.com', 'j25@yahoo.com'])
3. ... ).count()
4. 2

地址记录仍然在这里。如果我们commit的话，我们可以从上面的SQL语句中发现，相关的Address的user\_id属性被设置成了NULL。这不符合我们的要求。那么我们需要自己来设置关系的删除规则。

##### 配置delete/delete-orphan Cascade

我们通过配置User.addresses关系的**cascade\***选项来控制删除行为。尽管SQLAlchemy允许你在任何时候给ORM添加属性或者关系。此时我们还是需要移除现存的关系并且重新开始（作者：django的ORM包含）。让我们首先关闭当前的session

>>> session.close()

并且使用一个新的declarative\_base():

>>> Base = declarative\_base()

下面我们重新声明User类，注意addresses中的配置：

1. >>> class User(Base):
2. ... \_\_tablename\_\_ = 'users'
3. ...
4. ... id = Column(Integer, primary\_key=True)
5. ... name = Column(String)
6. ... fullname = Column(String)
7. ... password = Column(String)
8. ...
9. ... addresses = relationship("Address", back\_populates='user',
10. ... cascade="all, delete, delete-orphan")
11. ...
12. ... def \_\_repr\_\_(self):
13. ... return "<User(name='%s', fullname='%s', password='%s')>" % (
14. ... self.name, self.fullname, self.password)

接下来重新声明Address。

1. >>> class Address(Base):
2. ... \_\_tablename\_\_ = 'addresses'
3. ... id = Column(Integer, primary\_key=True)
4. ... email\_address = Column(String, nullable=False)
5. ... user\_id = Column(Integer, ForeignKey('users.id'))
6. ... user = relationship("User", back\_populates="addresses")
7. ...
8. ... def \_\_repr\_\_(self):
9. ... return "<Address(email\_address='%s')>" % self.email\_address

现在让我们取出jack(下面我们使用了一个之前没有提到的函数get()，其参数为查询目标的主键)，现在从addresses中删除一个地址的话，会导致这个Address被删除。

1. *# load Jack by primary key*
2. SQL>>> jack = session.query(User).get(5)
4. *# remove one Address (lazy load fires off)*
5. SQL>>> del jack.addresses[1]
7. *# only one address remains*
8. SQL>>> session.query(Address).filter(
9. ... Address.email\_address.in\_(['jack@google.com', 'j25@yahoo.com'])
10. ... ).count()
11. 1

删除jack也会导致剩下jack以及其所有的Address都会被删除:

1. >> session.delete(jack)
3. SQL>>> session.query(User).filter\_by(name='jack').count()
4. 0
6. SQL>>> session.query(Address).filter(
7. ... Address.email\_address.in\_(['jack@google.com', 'j25@yahoo.com'])
8. ... ).count()
9. 0

关于更多的Cascade配置请参见官方文档。

### 建立多对多关系ManyToMany Relationship

现在我们需要引入一个新的模型来阐述多对多的关系了。假设我们需要完成一个博客应用。在这个应用里面我们可以书写BlogPost，每个博客都有若干Keyword。

对于一个多对多的关系，我们需要建立一个未映射的（也就是没有一个Python类与之对应的）表Table来作为中间联系的表。

1. >>> from sqlalchemy import Table, Text
2. >>> *# association table*
3. >>> post\_keywords = Table('post\_keywords', Base.metadata,
4. ... Column('post\_id', ForeignKey('posts.id'), primary\_key=True),
5. ... Column('keyword\_id', ForeignKey('keywords.id'), primary\_key=True)
6. ... )

不同于我们之前的典型的ORM方法，在上面的代码中我们直接声明了一个Table，而没有制定与之对应的Python类。Table是一个构造函数，其参数中的每个Colomn以逗号分隔。

下面我们来定义BlogPost和Keyword。我们这里需要使用relationship()在这两个类中定义一对互补的关系，其中每个关系的都指向post\_keyword这个表。

1. >>> class BlogPost(Base):
2. ... \_\_tablename\_\_ = 'posts'
3. ...
4. ... id = Column(Integer, primary\_key=True)
5. ... user\_id = Column(Integer, ForeignKey('users.id'))
6. ... headline = Column(String(255), nullable=False)
7. ... body = Column(Text)
8. ...
9. ... *# many to many BlogPost<->Keyword*
10. ... keywords = relationship('Keyword',
11. ... secondary=post\_keywords,
12. ... back\_populates='posts')
13. ...
14. ... def \_\_init\_\_(self, headline, body, author):
15. ... self.author = author
16. ... self.headline = headline
17. ... self.body = body
18. ...
19. ... def \_\_repr\_\_(self):
20. ... return "BlogPost(%r, %r, %r)" % (self.headline, self.body, self.author)

23. >>> class Keyword(Base):
24. ... \_\_tablename\_\_ = 'keywords'
25. ...
26. ... id = Column(Integer, primary\_key=True)
27. ... keyword = Column(String(50), nullable=False, unique=True)
28. ... posts = relationship('BlogPost',
29. ... secondary=post\_keywords,
30. ... back\_populates='keywords')
31. ...
32. ... def \_\_init\_\_(self, keyword):
33. ... self.keyword = keyword

在上面的定义中，我们可以发现和OneToMany关系不同，relationship()中多了一个secondary的参数，这个参数指向了中间表(原文为associated table)。这个中间表只包含了指向多对多关系两侧的表的主键的列。如果这个表包含了其他属性，甚至是自身的主键，SQLAlchemy需要你使用另一种，称为association object的机制来处理。

我们还希望我们的BlogPost能够拥有一个author属性，这个属性指向我们先前定义的User。此时我们需要再定义一个双向关系。由于一个作者可能拥有很多文章，我们希望访问User.posts的时候可以加以筛选而不是载入全部的相关文章。为此我们在定义User.posts中的时候，设置lazy='dynamic'，来控制载入策略。

1. >>> BlogPost.author = relationship(User, back\_populates="posts")
2. >>> User.posts = relationship(BlogPost, back\_populates="author", lazy="dynamic")

然后让我们来创建数据库中对应的表

1. >>> Base.metadata.create\_all(engine)
2. PRAGMA...
3. CREATE TABLE keywords (
4. id INTEGER NOT NULL,
5. keyword VARCHAR(50) NOT NULL,
6. PRIMARY KEY (id),
7. UNIQUE (keyword)
8. )
9. ()
10. COMMIT
11. CREATE TABLE posts (
12. id INTEGER NOT NULL,
13. user\_id INTEGER,
14. headline VARCHAR(255) NOT NULL,
15. body TEXT,
16. PRIMARY KEY (id),
17. FOREIGN KEY(user\_id) REFERENCES users (id)
18. )
19. ()
20. COMMIT
21. CREATE TABLE post\_keywords (
22. post\_id INTEGER NOT NULL,
23. keyword\_id INTEGER NOT NULL,
24. PRIMARY KEY (post\_id, keyword\_id),
25. FOREIGN KEY(post\_id) REFERENCES posts (id),
26. FOREIGN KEY(keyword\_id) REFERENCES keywords (id)
27. )
28. ()
29. COMMIT

多对多关系的使用方法道也没有太大的不同之处。让我们先来给windy添加博文。

1. >>> wendy = session.query(User).\
2. ... filter\_by(name='wendy').\
3. ... one()
4. >>> post = BlogPost("Wendy's Blog Post", "This is a test", wendy)
5. >>> session.add(post)

给博文添加一些关键字。目前数据库里面还没有关键字存在，我们创建一些：

1. >>> post.keywords.append(Keyword('wendy'))
2. >>> post.keywords.append(Keyword('firstpost'))

我们可以开始查询了。先以'firstpost'为关键字来检索所有的博文。我们使用any来查询拥有关键词'firstpost'的博文：

1. >>> session.query(BlogPost).\
2. ... filter(BlogPost.keywords.any(keyword='firstpost')).\
3. ... all()
4. [BlogPost("Wendy's Blog Post", 'This is a test', <User(name='wendy', fullname='Wendy Williams', password='foobar')>)]

如果我们希望将查询范围限制在wendy用户所拥有的博文之内，

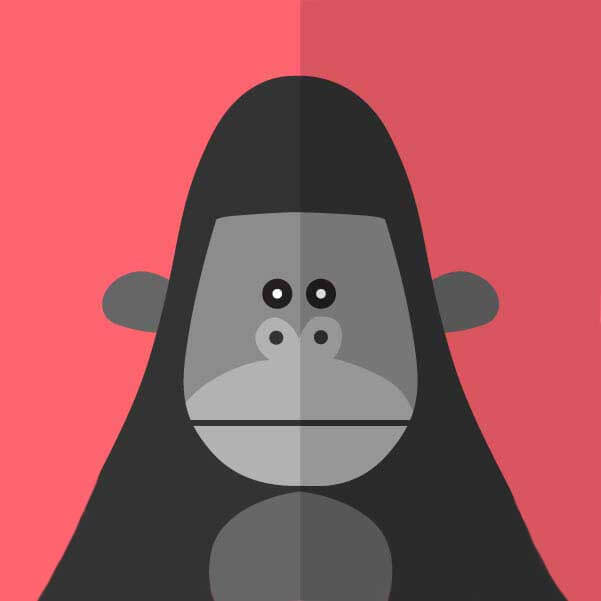
1. >>> session.query(BlogPost).\
2. ... filter(BlogPost.author==wendy).\
3. ... filter(BlogPost.keywords.any(keyword='firstpost')).\
4. ... all()
5. SELECT posts.id AS posts\_id,
6. posts.user\_id AS posts\_user\_id,
7. posts.headline AS posts\_headline,
8. posts.body AS posts\_body
9. FROM posts
10. WHERE ? = posts.user\_id AND (EXISTS (SELECT 1
11. FROM post\_keywords, keywords
12. WHERE posts.id = post\_keywords.post\_id
13. AND keywords.id = post\_keywords.keyword\_id
14. AND keywords.keyword = ?))
15. (2, 'firstpost')
16. [BlogPost("Wendy's Blog Post", 'This is a test', <User(name='wendy', fullname='Wendy Williams', password='foobar')>)]

或者我们可以直接在wendy的posts属性上进行查询：

1. >>> wendy.posts.\
2. ... filter(BlogPost.keywords.any(keyword='firstpost')).\
3. ... all()
4. [BlogPost("Wendy's Blog Post", 'This is a test', <User(name='wendy', fullname='Wendy Williams', password='foobar')>)]

**SQLAlchemy系列的教程到此告了一个小段落了，但是我觉得还是有很多说的不详细的地方。并且，教程中给出的用法和我在一些较为成熟的工程中所看到的SQLAlchemy的用法不是很一样，之后我会采用专题的形式来分话题详细阐述这些内容，喜欢的话like一下吧。**

# 2018-06-17 深入理解SQLAlchemy中的backref 是什么，和旧版back-populate的区别

[](https://www.jianshu.com/u/7fd6f81c5fb2)

[aoaocool](https://www.jianshu.com/u/7fd6f81c5fb2) 关注

2018.06.18 04:44\* 字数 933 阅读 242评论 0喜欢 0

深入理解SQLAlchemy中的backref 是什么，和旧版back-populate的区别：

看了 网上相关材料如下 :

https://www.zhihu.com/question/38456789

https://l.jifangcheng.com/p/46

基本 自悟的总结如下：

1） backref  是  back-populate的更高级，更简化版本，是新对旧的替代关系。

because of  旧版的 back-populate 必须 双向的，（啰嗦的back-populate 在 有relationship的2个类结构下都声明）

而 backref ，可以 单向的声明，在任何一个 类下，都可以，

2） ，backref ，一旦使用了，还具备支持 具体嵌套功能的功能：

看下面的 明明是 1对多关系的 （restaurant -对 history ，即 同一个 restaurant可以 对应 多个 它的history 记录）

但有了 backref的描绘后，就可以 用 history['restaurants'] （这个写法 == 使用restaurant对象,注意这里的history是histories类的一个实例表） 的方式  直接 取 restaurants 类的表结构里的col 值

下面为转载： 说的更清楚。

https://l.jifangcheng.com/p/46

SQLAlchemy 中的 backref 和 back\_populates

SQLAlchemy 中的 relationship 函数为模型之间的关系提供了一种便利的调用方式，backref 参数则对关系提供反向引用的声明。

下面的例子定义了一个简单的一对多的关系

class Parent(Base):\_\_tablename\_\_ ='parent'id = Column(Integer, primary\_key=True)    children = relationship("Child")class Child(Base):\_\_tablename\_\_ ='child'id = Column(Integer, primary\_key=True)    parent\_id = Column(Integer, ForeignKey('parent.id'))

一个 Parent 可以有多个 Child，一个 Child 只有一个 Parent

如果我们需要查询一个 Parent 的 Child，很简单，因为我们用了 relationship 函数将 Parent 的 children 字段和 Child表关联了起来，所以直接取 Parent 实例的 children 属性就可以了：

parent.children

但是如果我们要查询一个 Child 的 Parent 要怎么操作呢？Child 模型里面有一个 parent\_id 字段，这是一个关联到其对应 Parent 的外键，我们可以拿到这个 Child 的 Parent 的 id，然后用这个 id 去 Parent 表里面查询对应的 Parent 对象：

parent\_id = child.parent\_id

parent = session.query(Parent).filter\_by(id=parent\_id).first()

这也是一个方法，不过太麻烦了，SQLAlchemy 为关系的反向引用提供了两种更为简单、易用的方法：backref 和 back\_populates

backref 参数提供了对关系反向引用的声明：

class Parent(Base):\_\_tablename\_\_ ='parent'id = Column(Integer, primary\_key=True)    children = relationship("Child", backref="parent")

在定义关系的时候使用 backref 参数，该参数使得 Child 对象可以直接使用 backref 参数定义的 parent 值来访问其对应的 Parent

parent = child.parent

back\_populates 可以实现和 backref 同样的效果，不过在模型的定义上要麻烦一点：

class Parent(Base):\_\_tablename\_\_ ='parent'id = Column(Integer, primary\_key=True)    children = relationship("Child", 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="children")

在 Parent 和 Child 中都定义一个相互关联的 relationship，Parent 可以通过 children 字段获取其对应的 Child，Child也可以通过 parent 字段获取其对应的 Parent

back\_populates 和 backref 达到的效果都是一样的，对于它们的区别，我的理解是 backref 是隐式地声明了关系反向引用的字段，back\_populates 则是显式地定义了关系反向引用的字段。相比于 backref，back\_populates 使得模型之间的关系理清晰，所有的关系一目了然，不过也比 backref 麻烦一点，各有各的好处。

参考链接

[Basic Relationship Patterns — SQLAlchemy 1.2 Documentation](http://docs.sqlalchemy.org/en/latest/orm/basic_relationships.html)

[SQLAlchemy浅析](http://www.liuliqiang.info/post/sqlalchemy-guide/)