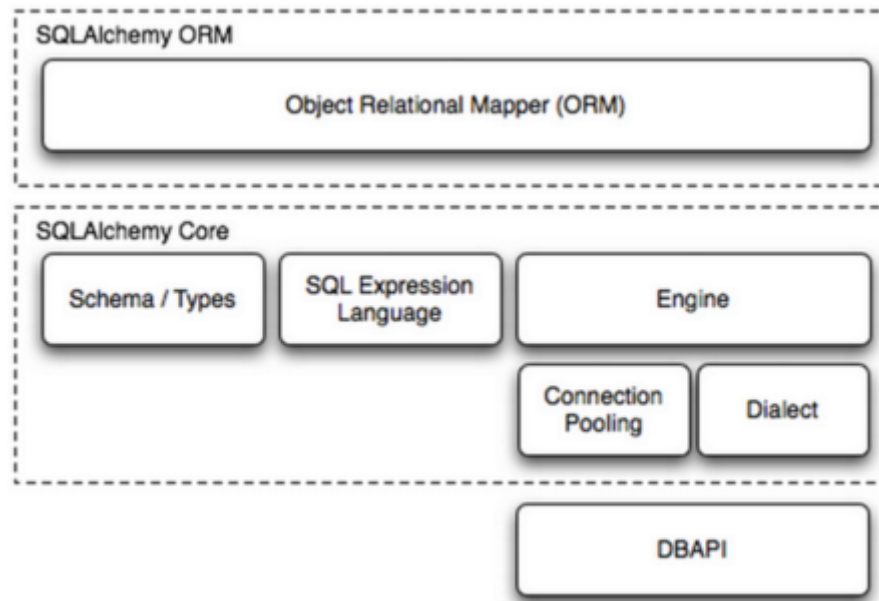


SQLAlchemy模型使用

SQLAlchemy模型使用

简介：

SQLAlchemy是[Python](#)编程语言下的一款ORM框架，该框架建立在数据库API之上，使用关系对象映射进行数据库操作，简言之便是：将对象转换成SQL，然后使用数据库API执行SQL并获取执行结果。



SQLAlchemy本身无法操作数据库，其必须以来pymysql等第三方插件，Dialect用于和数据API进行交流，根据配置文件的不同调用不同的数据库API，从而实现对数据库的操作，如：

```
1 MySQL-Python
2     mysql+mysqldb://<user>:<password>@<host>[:<port>]/<dbname>
3
4 pymysql
5     mysql+pymysql://<username>:<password>@<host>/<dbname>[?<options>]
6
7 MySQL-Connector
8     mysql+mysqlconnector://<user>:<password>@<host>[:<port>]/<dbname>
9
10 cx_Oracle
11     oracle+cx_oracle://user:pass@host:port/dbname[?key=value&key=value...]
12
13 更多详见: http://docs.sqlalchemy.org/en/latest/dialects/index.html
```

1.SQLAlchemy初始化和创建连接



```
# -*- coding: utf-8 -*-

from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from sqlalchemy import Column
from sqlalchemy.types import String, Integer
from sqlalchemy.ext.declarative import declarative_base
#导入相应的模块

engine = create_engine("mysql+pymysql://mysql:123456@10.0.0.8:3306/test", max_overflow=5) #创建数据库连接, max_overflow指定最大连接数
DBSession = sessionmaker(engine) #创建DBSession类型
session = DBSession() #创建session对象

BaseModel = declarative_base() #创建对象的基类

class User(BaseModel): #定义User对象
    __tablename__ = 'user' #创建表, 指定表名称

    #指定表的结构
    id = Column(String, primary_key=True)
    username = Column(String, index=True)

class Session(BaseModel):
    __tablename__ = 'session'

    id = Column(String, primary_key=True)
    user = Column(String, index=True)
    ip = Column(String)

BaseModel.metadata.create_all(engine) #创建表, 执行所有BaseModel类的子类
session.commit() #提交, 必须
```



2. SQLAlchemy创建连接

SQLAlchemy 的连接创建是 Lazy 的方式, 即在需要使用时才会去真正创建. 之前做的工作, 全是"定义". 连接的定义是在 *engine* 中做的.

2.1. Engine

engine 的定义包含了三部分的内容, 一是具体数据库类型的实现, 二是连接池, 三是策略(即*engine* 自己的实现).

所谓的数据库类型即是 MySQL, Postgresql, SQLite 这些不同的数据库.

一般创建 *engine* 是使用 `create_engine` 方法:

```
engine = create_engine("mysql+pymysql://mysql:123456@10.0.0.8:3306/test")
```

参数字符串的各部分的意义:

```
"mysql+pymysql://mysql:123456@10.0.0.8:3306/test"
```

对于这个字符串, SQLAlchemy 提供了工具可用于处理它:



```
1 # -*- coding: utf-8 -*-
2
3 from sqlalchemy import create_engine
4 from sqlalchemy.engine.url import make_url
5 from sqlalchemy.engine.url import URL
6
7 s = 'postgresql://test@localhost:5432/bbcustom'
8 url = make_url(s)
9 s = URL(drivername='postgresql', username='test', password="", host="localhost", port=5432,
database="bbcustom")
10
11 engine = create_engine(url)
```

```
12 engine = create_engine(s)
13
14 print engine.execute('select id from "user"').fetchall()
```



`create_engine` 函数有很多的控制参数, 这个后面再详细说.

2.2. Engine的策略

`create_engine` 的调用, 实际上会变成 `strategy.create` 的调用. 而 *strategy* 就是 *engine* 的实现细节. *strategy* 可以在 `create_engine` 调用时通过 `strategy` 参数指定, 目前官方的支持有三种:

- plain, 默认的
- threadlocal, 连接是线程局部的
- mock, 所有的 SQL 语句的执行会使用指定的函数

mock 这个实现, 会把所有的 SQL 语句的执行交给指定的函数来做, 这个函数是由 `create_engine` 的 `executor` 参数指定:



```
1 #!/usr/bin/env python
2 # time:
3 # Auto: PANpan
4 # func:
5 from sqlalchemy.ext.declarative import declarative_base
6 from sqlalchemy import Column, Integer, String, ForeignKey, UniqueConstraint, Index
7 from sqlalchemy.orm import sessionmaker, relationship
8 from sqlalchemy import create_engine

9 def f(sql, *args, **kwargs):
10     print(sql, args, kwargs)
11 s="mysql+pymysql://mysql:123456@10.0.0.8:3306/test"
12 engin=create_engine(s, strategy='mock', executor=f)
13 print(engin.execute('select id from "user"'))
```



2.3. 各数据库实现

各数据库的实现在 SQLAlchemy 中分成了两个部分, 一是数据库的类型, 二是具体数据库中适配的客户端实现. 比如对于 Postgresql 的访问, 可以使用 `psycopg2`, 也可以使用 `pg8000`:

```
1 s = 'postgresql+psycopg2://test@localhost:5432/bbcustom'
2 s = 'postgresql+pg8000://test@localhost:5432/bbcustom'
3 engine = create_engine(s)
```

具体的适配工作, 是需要在代码中实现一个 `Dialect` 类来完成的. 官方的实现在 `dialects` 目录下.

获取具体的 `Dialect` 的行为, 则是前面提到的 URL 对象的 `get_dialect` 方法. `create_engine` 时你单传一个字符串, SQLAlchemy 自己也会使用 `make_url` 得到一个 URL 的实例).

2.4. 连接池

SQLAlchemy 支持连接池, 在 `create_engine` 时添加相关参数即可使用.

- `pool_size` 连接数
- `max_overflow` 最多多几个连接
- `pool_recycle` 连接重置周期
- `pool_timeout` 连接超时时间

连接池效果:



```
# -*- coding: utf-8 -*-

from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from sqlalchemy import Column
from sqlalchemy.types import String, Integer
from sqlalchemy.ext.declarative import declarative_base

engine = create_engine("mysql+pymysql://mysql:123456@10.0.0.8:3306/test", poolsize=2, max_overflow=0)
```

```
DBSession = sessionmaker(engine)
session = DBSession()
BaseModel = declarative_base()

from threading import Thread
def f():
    print (engine.execute('show databases').fetchall())

p = []
for i in range(3):
    p.append(Thread(target=f))

for t in p:
    t.start()
```



3.SQLAlchemy 模型使用

3.1. 模型定义

对于 *Table* 的定义, 本来是直接的实例化调用, 通过 *declarative* 的包装, 可以像"定义类"这样的更直观的方式来完成.

```
user = Table('user',
    Column('user_id', Integer, primary_key = True),
    Column('user_name', String(16), nullable = False),
    Column('email_address', String(60)),
    Column('password', String(20), nullable = False)
```

)



```
1 #!/usr/bin/env python
2 # -*- coding:utf-8 -*-
3 from sqlalchemy.ext.declarative import declarative_base
4 from sqlalchemy import Column, Integer, String, ForeignKey, UniqueConstraint, Index
5 from sqlalchemy.orm import sessionmaker, relationship
6 from sqlalchemy import create_engine
7
8 engine = create_engine("mysql+pymysql://mysql:123456@10.0.0.8:3306/test", max_overflow=5) #创建数据库
连接
9 DBSession=sessionmaker(engine)#创建了一个自定义了的 Session类
10 session=DBSession()
11 Base = declarative_base()#创建对象的基类
12
13
14 class Blog(Base):
15     __tablename__ = 'blog'
16
17     id = Column(Integer, primary_key=True)
18     title = Column(String(64), server_default='', nullable=False)
19     text = Column(String, server_default='', nullable=False)
20     user = Column(String(32), index=True, server_default='', nullable=False)
21     create = Column(String(32), index=True, server_default='0', nullable=False)
22
23
24 class User(Base):
25     __tablename__ = 'user'
26
27     id = Column(Integer, primary_key=True)
28     name = Column(String(32), server_default='', nullable=False)
29     username = Column(String(32), index=True, server_default='', nullable=False)
30     password = Column(String(64), server_default='', nullable=False)
31
32
```



```
33 def init_db():
34     Base.metadata.create_all(engine)
35
36 def drop_db():
37     Base.metadata.drop_all(engine)
38
39 if __name__ == '__main__':
40     init_db()
41     #drop_db()
42     #Base.metadata.tables['user'].create(engine, checkfirst=True)
43     #Base.metadata.tables['user'].drop(engine, checkfirst=False)
44     #pass
```



3.2. 创建

```
1 session = Session()
2 session.add(User(id=1)) #创建一个
3 session.add(Blog(id=1))
4 session.add_all([ User(id=2), Blog(id=2) ]) #创建多个
5 session.commit()
```

执行的顺序并不一定会和代码顺序一致, SQLAlchemy 自己会整合逻辑再执行.

3.3. 查询

SQLAlchemy 实现的查询非常强大, 写起来有一种随心所欲的感觉.

查询的结果, 有几种不同的类型, 这个需要注意, 像是:

- instance
- instance of list
- keyed tuple of list
- value of list

基本查询:



```
1 session.query(User).filter_by(username='abc').all() #查询用户User创建表中username字段等于abc的内容
2 session.query(User).filter(User.username=='abc').all()
   #平时使用的时候，两者区别主要就是当使用filter的时候条件之间是使用"=="，filter_by使用的是"="

3 session.query(Blog).filter(Blog.create >= 0).all() #查询Blog对象中创建表中create字段大于等于0的所有内容
4 session.query(Blog).filter(Blog.create >= 0).first()
5 session.query(Blog).filter(Blog.create >= 0 | Blog.title == 'A').first() #条件或
6 session.query(Blog).filter(Blog.create >= 0 & Blog.title == 'A').first() #条件与
7 session.query(Blog).filter(Blog.create >= 0).offset(1).limit(1).scalar() # offset(N) 从第N条开始返回
limit(N) 最多返回 N 条记录 scalar() 如果有记录
8 session.query(User).filter(User.username == 'abc').scalar()
9 session.query(User.id).filter(User.username == 'abc').scalar()
10 session.query(Blog.id).filter(Blog.create >= 0).all()
11 session.query(Blog.id).filter(Blog.create >= 0).all()[0].id
12 dict(session.query(Blog.id, Blog.title).filter(Blog.create >= 0).all())
13 session.query(Blog.id, Blog.title).filter(Blog.create >= 0).first().title #记录不存在时，first() 会返回
None
14 session.query(User.id).order_by('id desc').all() #对结果集进行排序
15 session.query(User.id).order_by('id').first()
16 session.query(User.id).order_by(User.id).first()
17 session.query(User.id).order_by(-User.id).first()
18 session.query('id', 'username').select_from(User).all() #与
19 session.query(User).get('16e19a64d5874c308421e1a835b01c69') # 以主键获取
```



多表查询：

```
session.query(Blog, User).filter(Blog.user == User.id).first().User.username
session.query(Blog, User.id, User.username).filter(Blog.user == User.id).first().id
session.query(Blog.id, User.id, User.username).filter(Blog.user == User.id).first().keys()
```

条件查询：



```
from sqlalchemy import or_, not_

session.query(User).filter(or_(User.id == '',
                                User.id == '1')).all()

session.query(User).filter(not_(User.id == '1')).all()

session.query(User).filter(User.id.in_(['1'])).all()

session.query(User).filter(User.id.like('1%')).all()

session.query(User).filter(User.id.startswith('1')).all()

dir(User.id)
```



函数:



```
from sqlalchemy import func

session.query(func.count('1')).select_from(User).scalar()

session.query(func.count('1'), func.max(User.username)).select_from(User).first()

session.query(func.count('1')).select_from(User).scalar()

session.query(func.md5(User.username)).select_from(User).all()

session.query(func.current_timestamp()).scalar()

session.query(User).count()
```



3.4. 修改

还是通常的两种方式:

```
1 session.query(User).filter(User.username == 'abc').update({'name': '123'})
2 session.commit()
3
4 user=session.query(User).filter_by(username='abc').scalar()
5 user.name = '223'
6 session.commit()
```

如果涉及对属性原值的引用, 则要考虑 `synchronize_session` 这个参数.

- 'evaluate' 默认值, 会同时修改当前 session 中的对象属性.
- 'fetch' 修改前, 会先通过 select 查询条目的值.
- False 不修改当前 session 中的对象属性.

在默认情况下, 因为会有修改当前会话中的对象属性, 所以如果语句中有 SQL 函数, 或者"原值引用", 那是无法完成的操作, 自然也会报错, 比如:

```
from sqlalchemy import func
session.query(User).update({User.name: func.trim('123 ')})
session.query(User).update({User.name: User.name + 'x'})
```

这种情况下, 就不能要求 SQLAlchemy 修改当前 session 的对象属性了, 而是直接进行数据库的交互, 不管当前会话值:

```
session.query(User).update({User.name: User.name + 'x'}, synchronize_session=False)
```

是否修改当前会话的对象属性, 涉及到当前会话的状态. 如果当前会话过期, 那么在获取相关对象的属性值时, SQLAlchemy 会自动作一次数据库查询, 以便获取正确的值:

```
user = session.query(User).filter_by(username='abc').scalar()
print user.name
session.query(User).update({User.name: 'new'}, synchronize_session=False)
print (user.name)
session.commit()
print (user.name)
```

执行了 update 之后, 虽然相关对象的实际的属性值已变更, 但是当前会话中的对象属性值并没有改变. 直到 `session.commit()` 之后, 当前会话变成"过期"状态, 再次获取 `user.name` 时, SQLAlchemy 通过 `user` 的 `id` 属性, 重新去数据库查询了新值. (如果 `user` 的 `id` 变了呢? 那就会出事了啊.)

`synchronize_session` 设置成 'fetch' 不会有这样的问题, 因为在做 update 时已经修改了当前会话中的对象了.

不管 `synchronize_session` 的行为如何, commit 之后 session 都会过期, 再次获取相关对象值时, 都会重新作一次查询.

3.5. 删除

```
session.query(User).filter_by(username='abc').delete()

user = session.query(User).filter_by(username='abc').first()
session.delete(user)
```

删除同样有像修改一样的 `synchronize_session` 参数的问题, 影响当前会话的状态.

3.6. JOIN

SQLAlchemy 可以很直观地作 join 的支持:

```
1 r = session.query(Blog, User).join(User, Blog.user == User.id).all()
2 for blog, user in r:
3     print (blog.id, blog.user, user.id)

4 r = session.query(Blog, User.name, User.username).join(User, Blog.user == User.id).all()
5 print (r)
```

表操作总结:




```
1 #!/usr/bin/env python
2 # -*- coding:utf-8 -*-
3 from sqlalchemy.ext.declarative import declarative_base
4 from sqlalchemy import Column, Integer, String, ForeignKey, UniqueConstraint, Index
5 from sqlalchemy.orm import sessionmaker, relationship
6 from sqlalchemy import create_engine
7
8 engine = create_engine("mysql+pymysql://root:123@127.0.0.1:3306/t1", max_overflow=5)
9
10 Base = declarative_base()
11
12 # 创建单表
13 class Users(Base):
```

```
14     __tablename__ = 'users'
15     id = Column(Integer, primary_key=True)
16     name = Column(String(32))
17     extra = Column(String(16))
18
19     __table_args__ = (
20         UniqueConstraint('id', 'name', name='uix_id_name'),
21         Index('ix_id_name', 'name', 'extra'),
22     )
23
24     def __repr__(self):
25         return "%s-%s" %(self.id, self.name)
26
27 # 一对多
28 class Favor(Base):
29     __tablename__ = 'favor'
30     nid = Column(Integer, primary_key=True)
31     caption = Column(String(50), default='red', unique=True)
32
33     def __repr__(self):
34         return "%s-%s" %(self.nid, self.caption)
35
36 class Person(Base):
37     __tablename__ = 'person'
38     nid = Column(Integer, primary_key=True)
39     name = Column(String(32), index=True, nullable=True)
40     favor_id = Column(Integer, ForeignKey("favor.nid"))
41     # 与生成表结构无关, 仅用于查询方便
42     favor = relationship("Favor", backref='pers')
43
44 # 多对多
45 class ServerToGroup(Base):
46     __tablename__ = 'servertogroup'
47     nid = Column(Integer, primary_key=True, autoincrement=True)
48     server_id = Column(Integer, ForeignKey('server.id'))
49     group_id = Column(Integer, ForeignKey('group.id'))
50     group = relationship("Group", backref='s2g')
```

```
51     server = relationship("Server", backref='s2g')
52
53 class Group(Base):
54     __tablename__ = 'group'
55     id = Column(Integer, primary_key=True)
56     name = Column(String(64), unique=True, nullable=False)
57     port = Column(Integer, default=22)
58     # group = relationship('Group',secondary=ServerToGroup,backref='host_list')
59
60
61 class Server(Base):
62     __tablename__ = 'server'
63
64     id = Column(Integer, primary_key=True, autoincrement=True)
65     hostname = Column(String(64), unique=True, nullable=False)
66
67
68
69
70 def init_db():
71     Base.metadata.create_all(engine)
72
73
74 def drop_db():
75     Base.metadata.drop_all(engine)
76
77
78 Session = sessionmaker(bind=engine)
79 session = Session()
```



1.增



```
1 obj = Users(name="alex0", extra='sb')
2 session.add(obj)
3 session.add_all([
4     Users(name="alex1", extra='sb'),
5     Users(name="alex2", extra='sb'),
6 ])
7 session.commit()
```



2.删

```
session.query(Users).filter(Users.id > 2).delete()
session.commit()
```

3.改

```
1 session.query(Users).filter(Users.id > 2).update({"name" : "099"})
2 session.query(Users).filter(Users.id > 2).update({Users.name: Users.name + "099"},
synchronize_session=False)
3 session.query(Users).filter(Users.id > 2).update({"num": Users.num + 1},
synchronize_session="evaluate")
4 session.commit()
```

4.查

```
1 ret = session.query(Users).all()
2 ret = session.query(Users.name, Users.extra).all()
3 ret = session.query(Users).filter_by(name='alex').all()
4 ret = session.query(Users).filter_by(name='alex').first()
```

5.其他



```
1 # 条件
2 ret = session.query(Users).filter_by(name='alex').all()
```



```
3 ret = session.query(Users).filter(Users.id > 1, Users.name == 'eric').all()
4 ret = session.query(Users).filter(Users.id.between(1, 3), Users.name == 'eric').all()
5 ret = session.query(Users).filter(Users.id.in_([1,3,4])).all()
6 ret = session.query(Users).filter(~Users.id.in_([1,3,4])).all()
7 ret =
session.query(Users).filter(Users.id.in_(session.query(Users.id).filter_by(name='eric'))).all()
8 from sqlalchemy import and_, or_
9 ret = session.query(Users).filter(and_(Users.id > 3, Users.name == 'eric')).all()
10 ret = session.query(Users).filter(or_(Users.id < 2, Users.name == 'eric')).all()
11 ret = session.query(Users).filter(
12     or_(
13         Users.id < 2,
14         and_(Users.name == 'eric', Users.id > 3),
15         Users.extra != ""
16     )).all()
17
18
19 # 通配符
20 ret = session.query(Users).filter(Users.name.like('e%')).all()
21 ret = session.query(Users).filter(~Users.name.like('e%')).all()
22
23 # 限制
24 ret = session.query(Users)[1:2]
25
26 # 排序
27 ret = session.query(Users).order_by(Users.name.desc()).all()
28 ret = session.query(Users).order_by(Users.name.desc(), Users.id.asc()).all()
29
30 # 分组
31 from sqlalchemy.sql import func
32
33 ret = session.query(Users).group_by(Users.extra).all()
34 ret = session.query(
35     func.max(Users.id),
36     func.sum(Users.id),
37     func.min(Users.id)).group_by(Users.name).all()
38
```

```
39 ret = session.query(
40     func.max(Users.id),
41     func.sum(Users.id),
42     func.min(Users.id)).group_by(Users.name).having(func.min(Users.id) >2).all()
43
44 # 连表
45
46 ret = session.query(Users, Favor).filter(Users.id == Favor.nid).all()
47
48 ret = session.query(Person).join(Favor).all()
49
50 ret = session.query(Person).join(Favor, isouter=True).all()
51
52
53 # 组合
54 q1 = session.query(Users.name).filter(Users.id > 2)
55 q2 = session.query(Favor.caption).filter(Favor.nid < 2)
56 ret = q1.union(q2).all()
57
58 q1 = session.query(Users.name).filter(Users.id > 2)
59 q2 = session.query(Favor.caption).filter(Favor.nid < 2)
60 ret = q1.union_all(q2).all()
```



4. SQLAlchemy外键和关系

4.1. 外键约束

使用 *ForeignKey* 来定义一个外键约定:



```
1 #!/usr/bin/env python
2 # -*- coding:utf-8 -*-
3 from sqlalchemy.ext.declarative import declarative_base
4 from sqlalchemy import Column, Integer, String, CHAR, BIGINT, ForeignKey, UniqueConstraint, Index
5 from sqlalchemy.orm import sessionmaker, relationship
6 from sqlalchemy import create_engine
7
8 engine = create_engine("mysql+pymysql://mysql:123456@10.0.0.8:3306/test", max_overflow=5) #创建数据库
连接
9 DBSession=sessionmaker(engine) #创建了一个自定义了的 Session类
10 session=DBSession()
11 BaseModel = declarative_base() #创建对象的基类
12 class Blog(BaseModel):
13     __tablename__='blog'
14
15     id=Column(BIGINT,primary_key=True,autoincrement=True)
16     title=Column(String(64),server_default='',nullable=False)
17     text=Column(String(256),server_default='',nullable=False)
18     user=Column(BIGINT,ForeignKey('user.id'),index=True,nullable=False) #将user表中id字段作为外键,使用
ForeignKey指定
19
20 class User(BaseModel):
21     __tablename__='user'
22
23     id=Column(BIGINT,primary_key=True,autoincrement=True)
24     name=Column(String(32),server_default='',nullable=False)
25     username=Column(String(32),index=True,server_default='',nullable=True)
26     passwd=Column(String(64),server_default='',nullable=False)
```



创建时:



```
1 BaseModel.metadata.create_all(engine)
2 user=User(name='first',username='pan',passwd='123456')
3 session.add(user)
4 session.flush()
5 blog = Blog(title='frist', user=user.id)
6 session.add(blog)
7 session.commit()
```



`session.flush()` 是进行数据库交互, 但是事务并没有提交. 进行数据库交互之后, `user.id` 才有值.

定义了外键, 对查询来说, 并没有影响. 外键只是单纯的一条约束而已. 当然, 可以在外键上定义一些关联的事件操作, 比如当外键条目被删除时, 字段置成 `null`, 或者关联条目也被删除等.

4.2. 关系定义

要定义关系, 必有使用 *ForeignKey* 约束. 当然, 这里说的只是在定义模型时必有要有, 至于数据库中是否真有外键约定, 这并不重要.

接下来我们来了解几个关于外键(Foreign Key)的小知识:

1. FOREIGN KEY 约束是大多数(但不是所有)的关系型数据库中可以链接到主键列, 或者拥有UNIQUE约束的列.
2. FOREIGN KEY 能够引用多重列主键, 并且其自身拥有多重列, 被称为“复合外键”(composite foreign key). 其也能够引用这些列的子集(subset). (注: 这地方不太明白)
3. FOREIGN KEY 列作为对于其引用的列或者行的变化的响应能够自动更新其自身, 比如CASCADE引用操作, 这些都是内置于关系型数据库的功能之一。
4. FOREIGN KEY 能够引用其自身的表, 这个就涉及到“自引用”(self-referential)的外键了。
5. 更多关于外键的资料可以参考[Foreign Key – Wikipedia](#)。



```
1 #!/usr/bin/env python
2 # -*- coding:utf-8 -*-
3 from sqlalchemy.ext.declarative import declarative_base
4 from sqlalchemy import Column, Integer, String, CHAR, BIGINT, ForeignKey, UniqueConstraint, Index
5 from sqlalchemy.orm import sessionmaker, relationship
6 from sqlalchemy import create_engine
7
8 engine = create_engine("mysql+pymysql://mysql:123456@10.0.0.8:3306/test", max_overflow=5) #创建数据库
连接
9 DBSession=sessionmaker(engine) #创建了一个自定义了的 Session类
10 session=DBSession()
11 BaseModel = declarative_base() #创建对象的基类
12 class Blog(BaseModel):
13     __tablename__='blog'
14
15     id=Column(BIGINT,primary_key=True,autoincrement=True)
16     title=Column(String(64),server_default='',nullable=False)
17     text=Column(String(256),server_default='',nullable=False)
18     user=Column(BIGINT,ForeignKey('user.id'),index=True,nullable=False)
19     user_obj=relationship('User')
20
21 class User(BaseModel):
22     __tablename__='user'
23
24     id=Column(BIGINT,primary_key=True,autoincrement=True)
25     name=Column(String(32),server_default='',nullable=False)
26     username=Column(String(32),index=True,server_default='',nullable=True)
27     passwd=Column(String(64),server_default='',nullable=False)
28     blog_list=relationship('Blog',backref='Blog.user')
29     #relationship函数是sqlalchemy对关系之间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。
30     #大致原理应该就是在sqlalchemy在运行时对Blog对象动态的设置了一个指向所属User对象的属性, 这样就能在实际开发中使逻辑
关系更加清晰, 代码更加简洁了。
31
32 BaseModel.metadata.create_all(engine)
```

```

33 user=User(name='first',username='pan',passwd='123456')
34 session.add(user)
35 session.flush()
36 blog = Blog(title='frist', user=user.id)
37 session.add(blog)
38 session.commit()

```



关系只是 SQLAlchemy 提供的工具, 与数据库无关, 所以任何时候添加都是可以的.

上面的 *User-Blog* 是一个"一对多"关系, 通过 *Blog* 的 *user* 这个 *ForeignKey*, SQLAlchemy 可以自动处理关系的定义. 在查询时, 返回的结果自然也是, 一个是列表, 一个是单个对象:



```

1 ret=session.query(Blog).get(1).user_obj
2 ret1=session.query(User).get(1).blog_list
3 print(ret.passwd,ret1[0].id)#ret.passwd=User.passwd  ret1[0].id=Blog.user列表中第一个值

```

```

#blog_list=relationship('Blog',backref='Blog.user')
#user_obj=relationship('User')
#relationship指定对象不同引用方法不同

```



这种关系的定义, 并不影响查询并获取对象的行为, 不会添加额外的 *join* 操作. 在对象上取一个 *user_obj* 或者取 *blog_list* 都是发生了一个新的查询操作.

上面的关系定义, 对应的属性是实际查询出的实例列表, 当条目数多的时候, 这样可能会有问题. 比如用户名下有成千上万的文章, 一次全取出就太暴力了. 关系对应的属性可以定义成一个 *Query* :



```

1 class User(BaseModel):
2     __tablename__ = 'user'
3
4     id = Column(BIGINT, primary_key=True, autoincrement=True)
5     name = Column(String(32), server_default='', nullable=False)

```

```
6
7     blog_list = relationship('Blog', order_by='Blog.user', lazy="dynamic")
```



这样在获取实例时就可以自由控制了:

```
1 session.query(User).get(1).blog_list.all()
2 session.query(User).get(1).blog_list.filter(Blog.title == '1').first()
```

4.3. 关系的查询

关系定义之后, 除了在查询时会有自动关联的效果, 在作查询时, 也可以对定义的关系做操作:



```
1 class Blog(BaseModel):
2     __tablename__ = 'blog'
3
4     id = Column(Integer, autoincrement=True, primary_key=True)
5     title = Column(Unicode(32), server_default='')
6     user = Column(Integer, ForeignKey('user.id'), index=True)
7
8     user_obj = relationship('User')
9
10
11 class User(BaseModel):
12     __tablename__ = 'user'
13
14     id = Column(Integer, autoincrement=True, primary_key=True)
15     name = Column(Unicode(32), server_default='')
16
17     blogs = relationship('Blog')
```



对于 一对多的关系, 使用 `any()` 函数查询:

```
user = session.query(User).filter(User.blogs.any(Blog.title == 'first')).first()
```

SQLAlchemy 会使用 `exists` 条件, 类似于:

```
1 SELECT *FROM user WHERE EXISTS
2     (SELECT 1 FROM blog WHERE user.id = blog.user AND blog.title = ?) LIMIT ? OFFSET ?
```

反之, 如果是 多对一 的关系, 则使用 `has()` 函数查询:

```
blog = session.query(Blog).filter(Blog.user_obj.has(User.name == 'pan')).first()
```

最后的 SQL 语句都是一样的.

4.4. 关系的获取形式

前面介绍的关系定义中, 提到了两种关系的获取形式, 一种是:

```
user_obj = relationship('User')
```

这种是在对象上获取关系对象时, 再去查询.

另一种是:

```
blog_list = relationship('Blog', lazy="dynamic")
```

这种的结果, 是在对象上获取关系对象时, 只返回 *Query*, 而查询的细节由人为来控制.

总的来说, 关系的获取分成两种, *Lazy* 或 *Eager*. 在直接查询层面, 上面两种都属于 *Lazy* 的方式, 而 *Eager* 的一种, 就是在获取对象时的查询语句, 是直接带 `join` 的, 这样关系对象的数据在一个查询语句中就直接获取到了:



```
1 class Blog(BaseModel):
2     __tablename__ = 'blog'
3
4     id = Column(BIGINT, primary_key=True, autoincrement=True)
5     title = Column(String(64), server_default='', nullable=False)
```



```

6     user = Column(BIGINT, ForeignKey('user.id'), index=True, nullable=False)
7
8     user_obj = relationship('User', lazy='joined', cascade='all')
9
10
11 class User(BaseModel):
12     __tablename__ = 'user'
13
14     id = Column(BIGINT, primary_key=True, autoincrement=True)
15     name = Column(String(32), server_default='', nullable=False)

```



这样在查询时:

```

blog = session.query(Blog).first()
print( blog.user_obj)

```

便会多出 LEFT OUTER JOIN 的语句, 结果中直接获取到对应的 User 实例对象.

也可以把 joined 换成子查询, subquery:



```

class User(BaseModel):
    __tablename__ = 'user'

    id = Column(BIGINT, primary_key=True, autoincrement=True)
    name = Column(String(32), server_default='', nullable=False)

    blog_list = relationship('Blog', cascade='all', lazy='subquery')

if __name__ == '__main__':

    session = Session()
    user = session.query(User).first()
    session.commit()

```



子查询会用到临时表.

上面定义的:

```
1 blog_list = relationship('Blog', lazy="dynamic")
2 user_obj = relationship('User', lazy='joined')
3 blog_list = relationship('Blog', lazy='subquery')
```

都算是一种默认方式. 在具体使用查询时, 还可以通过 `options()` 方法定义关联的获取方式:

```
from sqlalchemy.orm import lazyload, joinedload, subqueryload
user = session.query(User).options(lazyload('blog_list')).first()
print (user.blog_list)
```

更多的用法:



```
1 session.query(Parent).options(
2     joinedload('foo').joinedload('bar').joinedload('bat')
3 ).all()
4
5 session.query(A).options(
6     defaultload("atob").joinedload("btoc")
7 ).all()
8
9 session.query(MyClass).options(lazyload('*'))
10
11 session.query(MyClass).options(
12     lazyload('*'), joinedload(MyClass.widget)
13 )
14
15 session.query(User, Address).options(Load(Address).lazyload('*'))
```



如果关联的定义之前是 *Lazy* 的, 但是实际使用中, 希望在手工 *join* 之后, 把关联对象直接包含进结果实例, 可以使用 `contains_eager()` 来包装一下:

```
1 from sqlalchemy.orm import contains_eager
2
3 blog = session.query(Blog).join(Blog.user_obj)\
4     .options(contains_eager(Blog.user_obj)).first()
5 print blog.user_obj
```

4.5. 关系的表现形式

关系在对象属性中的表现, 默认是列表, 但是, 这不是唯一的形式. 根据需要, 可以作成 dictionary, set 或者其它你需要的对象.



```
1 class Blog(BaseModel):
2     __tablename__ = 'blog'
3
4     id = Column(Integer, autoincrement=True, primary_key=True)
5     title = Column(Unicode(32), server_default='')
6     user = Column(Integer, ForeignKey('user.id'), index=True)
7
8     user_obj = relationship('User')
9
10
11 class User(BaseModel):
12     __tablename__ = 'user'
13
14     id = Column(Integer, autoincrement=True, primary_key=True)
15     name = Column(Unicode(32), server_default='')
16
17     blogs = relationship('Blog')
```



对于上面的两个模型:

```
user = session.query(User).first()
print (user.blogs)
```

现在 `user.blogs` 是一个列表. 我们可以在 `relationship()` 调用时通过 `collection_class` 参数指定一个类, 来重新定义关系的表现形式:

```
1 user = User(name='XXX')
2 session.add_all([Blog(title='A', user_obj=user), Blog(title='B', user_obj=user)])
3 session.commit()
4
5 user = session.query(User).first()
6 print (user.blogs)
```

set, 集合:

```
blogs = relationship('Blog', collection_class=set)

#InstrumentedSet([<__main__.Blog object at 0x1a58710>, <__main__.Blog object at 0x1a587d0>])
```

attribute_mapped_collection, 字典, 键值从属性取:

```
1 from sqlalchemy.orm.collections import attribute_mapped_collection
2
3 blogs = relationship('Blog', collection_class=attribute_mapped_collection('title'))
4
5 #{'A': <__main__.Blog object at 0x20ed810>, 'B': <__main__.Blog object at 0x20ed8d0>}
```

如果 `title` 重复的话, 结果会覆盖.

mapped_collection, 字典, 键值自定义:

```
1 from sqlalchemy.orm.collections import mapped_collection
2
3 blogs = relationship('Blog', collection_class=mapped_collection(lambda blog: blog.title.lower()))
```

4.6. 多对多关系

先考虑典型的多对多关系结构:



```
1 class Blog(BaseModel):
2     __tablename__ = 'blog'
3
4     id = Column(BIGINT, primary_key=True, autoincrement=True)
5     title = Column(String(64), server_default='', nullable=False)
6
7     tag_list = relationship('Tag')
8     tag_list = relationship('BlogAndTag')
9
10
11 class Tag(BaseModel):
12     __tablename__ = 'tag'
13
14     id = Column(BIGINT, primary_key=True, autoincrement=True)
15     name = Column(String(16), server_default='', nullable=False)
16
17
18 class BlogAndTag(BaseModel):
19     __tablename__ = 'blog_and_tag'
20
21     id = Column(BIGINT, primary_key=True, autoincrement=True)
22     blog = Column(BIGINT, ForeignKey('blog.id'), index=True)
23     tag = Column(BIGINT, ForeignKey('tag.id'), index=True)
24     create = Column(BIGINT, index=True, server_default='0')
```



在 Blog 中的:

```
tag_list = relationship('Tag')
```

显示是错误的, 因为在 Tag 中并没有外键. 而:

```
tag_list = relationship('BlogAndTag')
```

这样虽然正确, 但是 `tag_list` 的关系只是到达 `BlogAndTag` 这一层, 并没有到达我们需要的 `Tag`.

这种情况下, 一个多对多关系是有三张表来表示的, 在定义 *relationship* 时, 就需要一个 `secondary` 参数来指明关系表:

```
1 class Blog(BaseModel):
2     __tablename__ = 'blog'
3
4     id = Column(BIGINT, primary_key=True, autoincrement=True)
5     title = Column(String(64), server_default='', nullable=False)
6
7     tag_list = relationship('Tag', secondary=lambda: BlogAndTag.__table__)
8     #是用lambda可以使后面跟的类无需提前定义, 如果直接secondary=classname, 则class需提前定义
9
10 class Tag(BaseModel):
11     __tablename__ = 'tag'
12
13     id = Column(BIGINT, primary_key=True, autoincrement=True)
14     name = Column(String(16), server_default='', nullable=False)
15
16
17 class BlogAndTag(BaseModel):
18     __tablename__ = 'blog_and_tag'
19
20     id = Column(BIGINT, primary_key=True, autoincrement=True)
21     blog = Column(BIGINT, ForeignKey('blog.id'), index=True)
22     tag = Column(BIGINT, ForeignKey('tag.id'), index=True)
23     create = Column(BIGINT, index=True, server_default='0')
```

这样, 在操作时可以直接获取到对应的实例列表:

```
blog = session.query(Blog).filter(Blog.title == 'a').one()
print (blog.tag_list)
```


访问 `tag_list` 时, SQLAlchemy 做的是个普通的多表查询.

`tag_list` 属性同时支持赋值操作:

```
session = Session()
blog = session.query(Blog).filter(Blog.title == 'a').one()
blog.tag_list = [Tag(name='t1')]
session.commit()
```

提交时, SQLAlchemy 总是会创建 `Tag`, 及对应的关系 `BlogAndTag`.


而如果是:



```
1 session = Session()
2 blog = session.query(Blog).filter(Blog.title == 'a').one()
3 blog.tag_list = []
4 session.commit()
5
6 tag = session.query(Tag).filter(Tag.name == 'x').one()
7 blog.tag_list.remove(tag)
8 session.commit()
```

那么 SQLAlchemy 只会删除对应的关系 `BlogAndTag`, 不会删除实体 `Tag`.

如果你直接删除实体, 那么对应的关系是不会自动删除的:



```
1 session = Session()
2 blog = session.query(Blog).filter(Blog.title == 'a').one()
3 tag = Tag(name='ok')
4 blog.tag_list = [tag]
5 session.commit()
6
7 tag = session.query(Tag).filter(Tag.name == 'ok').one()
```

```
8 session.delete(tag)
9 session.commit()
```



4.7. Cascades 自动关系处理

前面提到的, 当操作关系, 实体时, 与其相关联的关系, 实体是否会被自动处理的问题, 在 SQLAlchemy 中是通过 *Cascades* 机制来定义和解决的. (*Cascades* 这个词是来源于 *Hibernate* .)

cascade 是一个 *relationship* 的参数, 其值是逗号分割的多个字符串, 以表示不同的行为. 默认值是 "*save-update, merge*", 稍后会介绍每个词项的作用.

这里的所有规则介绍, 只涉及从 *Parent* 到 *Child* , *Parent* 即定义 *relationship* 的类. 不涉及 *backref* .

cascade 所有的可选字符串项是:

- *all* , 所有操作都会自动处理到关联对象上.
- *save-update* , 关联对象自动添加到会话.
- *delete* , 关联对象自动从会话中删除.
- *delete-orphan* , 属性中去掉关联对象, 则会话中会自动删除关联对象.
- *merge* , *session.merge()* 时会处理关联对象.
- *refresh-expire* , *session.expire()* 时会处理关联对象.
- *expunge* , *session.expunge()* 时会处理关联对象.

save-update

当一个对象被添加进 *session* 后, 此对象标记为 *save-update* 的 *relationship* 关系对象也会同时添加进这个 *session* .



```
1 class Blog(BaseModel):
2     __tablename__ = 'blog'
3
4     id = Column(BIGINT, primary_key=True, autoincrement=True)
5     title = Column(String(64), server_default='', nullable=False)
```



```

6     user = Column(BIGINT, ForeignKey('user.id'), index=True, nullable=False)
7
8
9 class User(BaseModel):
10     __tablename__ = 'user'
11
12     id = Column(BIGINT, primary_key=True, autoincrement=True)
13     name = Column(String(32), server_default='', nullable=False)
14
15     blog_list = relationship('Blog', cascade='')
16     blog_list_auto = relationship('Blog', cascade='save-update')
17
18
19 if __name__ == '__main__':
20
21     session = Session()
22
23     user = User(name=u'哈哈')
24     blog = Blog(title=u'第一个')
25     user.blog_list = [blog]
26     #user.blog_list_auto = [blog]
27     session.add(user)
28     print (blog in session)
29     session.commit()

```



delete

当一个对象在 session 中被标记为删除时, 其属性中 *relationship* 关联的对象也会被标记成删除, 否则, 关联对象中的对应外键字段会被改成 NULL, 不能为 NULL 则报错.



```

1 class Blog(BaseModel):
2     __tablename__ = 'blog'
3

```

```

4     id = Column(BIGINT, primary_key=True, autoincrement=True)
5     title = Column(String(64), server_default='', nullable=False)
6     user = Column(BIGINT, ForeignKey('user.id'), index=True, nullable=False)
7
8
9 class User(BaseModel):
10     __tablename__ = 'user'
11
12     id = Column(BIGINT, primary_key=True, autoincrement=True)
13     name = Column(String(32), server_default='', nullable=False)
14
15     blog_list = relationship('Blog', cascade='save-update, delete')
16
17
18 if __name__ == '__main__':
19     session = Session()
20
21     #user = User(name=u'用户')
22     #user.blog_list = [Blog(title=u'哈哈')]
23     #session.add(user)
24     user = session.query(User).first()
25     session.delete(user)
26     session.commit()

```



delete-orphan

当 *relationship* 属性变化时, 被 "去掉" 的对象会被自动删除. 比如之前是:

```
user.blog_list = [blog, blog2]
```

现在变成:

```
user.blog_list = [blog2]
```

那么 blog 这个关联实体是会自动删除的. 这各机制只适用于 "一对多" 的关系中, "多对多" 和反过来的 "多对一" 都不适用. 在 relationship 定义时, 可以添加 single_parent = True 参数来强约束. 当然, 在实现上 SQLAlchemy 是会先查出所有关联实体, 然后计算差集确认哪些需要被删除.



```
1 class Blog(BaseModel):
2     __tablename__ = 'blog'
3
4     id = Column(BIGINT, primary_key=True, autoincrement=True)
5     title = Column(String(64), server_default='', nullable=False)
6     user = Column(BIGINT, ForeignKey('user.id'), index=True, nullable=False)
7
8
9 class User(BaseModel):
10     __tablename__ = 'user'
11
12     id = Column(BIGINT, primary_key=True, autoincrement=True)
13     name = Column(String(32), server_default='', nullable=False)
14
15     blog_list = relationship('Blog', cascade='save-update, delete-orphan')
16
17
18 if __name__ == '__main__':
19
20     session = Session()
21
22     #user = User(name=u'用户')
23     #blog = Blog(title=u'一')
24     #blog2 = Blog(title=u'二')
25     #user.blog_list = [blog, blog2]
26     #session.add(user)
27     user = session.query(User).first()
28     blog2 = session.query(Blog).filter(Blog.title == u'二').first()
29     user.blog_list = [blog2]
30     #session.delete(user)
31     session.commit()
```



merge

这个选项是标识在 `session.merge()` 时处理关联对象. `session.merge()` 的作用, 是把一个会话外的实例, "整合"进会话, 比如 "有则修改, 无则创建" 就是典型的一种 "整合":



```
1 user = User(id=1, name="1")
2 session.add(user)
3 session.commit()
4
5 user = User(id=1)
6 user = session.merge(user)
7 print user.name
8
9 user = User(id=1, name="2")
10 user = session.merge(user)
11 session.commit()
```



cascade 中的 merge 作用:



```
1 class Blog(BaseModel):
2     __tablename__ = 'blog'
3
4     id = Column(BIGINT, primary_key=True, autoincrement=True)
5     title = Column(String(64), server_default='', nullable=False)
6     user = Column(BIGINT, ForeignKey('user.id'), index=True, nullable=False)
7
8
```

```

9 class User(BaseModel):
10     __tablename__ = 'user'
11
12     id = Column(BIGINT, primary_key=True, autoincrement=True)
13     name = Column(String(32), server_default='', nullable=False)
14
15     blog_list = relationship('Blog',
16                             cascade='save-update, delete, delete-orphan, merge')
17
18
19 if __name__ == '__main__':
20
21     session = Session()
22
23     user = User(id=1, name='1')
24     session.add(user)
25     session.commit(user)
26
27     user = User(id=1, blog_list=[Blog(title='哈哈')])
28     session.merge(user)
29
30     session.commit()

```



refresh-expire



1 当使用 `session.expire()` 标识一个对象过期时，此对象的关联对象是否也被标识为过期（访问属性会重新查询数据库）。

2

```

3 class Blog(BaseModel):
4     __tablename__ = 'blog'
5
6     id = Column(BIGINT, primary_key=True, autoincrement=True)
7     title = Column(String(64), server_default='', nullable=False)

```

```

8     user = Column(BIGINT, ForeignKey('user.id'), index=True, nullable=False)
9
10
11 class User(BaseModel):
12     __tablename__ = 'user'
13
14     id = Column(BIGINT, primary_key=True, autoincrement=True)
15     name = Column(String(32), server_default='', nullable=False)
16
17     blog_list = relationship('Blog',
18                             cascade='save-update, delete, delete-orphan, merge, refresh-expire')
19
20
21 if __name__ == '__main__':
22
23     session = Session()
24
25     #user = User(id=1, name='1')
26     #blog = Blog(title="abc")
27     #user.blog_list = [blog]
28     #session.add(user)
29
30     user = session.query(User).first()
31     blog = user.blog_list[0]
32     print (user.name)
33     print (blog.title)
34     session.expire(user)
35     print ('EXPIRE')
36     print (user.name)
37     print (blog.title)
38
39     session.commit()

```



与 *merge* 相反, 当 *session.expunge()* 把对象从会话中去除的时候, 此对象的关联对象也同时从会话中消失.



```
1 class Blog(BaseModel):
2     __tablename__ = 'blog'
3
4     id = Column(BIGINT, primary_key=True, autoincrement=True)
5     title = Column(String(64), server_default='', nullable=False)
6     user = Column(BIGINT, ForeignKey('user.id'), index=True, nullable=False)
7
8
9 class User(BaseModel):
10     __tablename__ = 'user'
11
12     id = Column(BIGINT, primary_key=True, autoincrement=True)
13     name = Column(String(32), server_default='', nullable=False)
14
15     blog_list = relationship('Blog', cascade='delete, delete-orphan, expunge')
16
17
18 if __name__ == '__main__':
19
20     session = Session()
21     user = User(name=u'用户')
22     blog = Blog(title=u'第一个')
23     user.blog_list = [blog]
24
25     session.add(user)
26     session.add(blog)
27
28     session.expunge(user)
29     print (blog in session)
30
31     #session.commit()
```



4.8. 属性代理

考虑这样的情况, 关系是关联的整个模型对象的, 但是, 有时我们对于这个关系, 并不关心整个对象, 只关心其中的某个属性. 考虑下面的场景:



```
1 from sqlalchemy.ext.associationproxy import association_proxy
2
3 class Blog(BaseModel):
4     __tablename__ = 'blog'
5
6     id = Column(Integer, autoincrement=True, primary_key=True)
7     title = Column(Unicode(32), nullable=False, server_default='')
8     user = Column(Integer, ForeignKey('user.id'), index=True)
9
10
11 class User(BaseModel):
12     __tablename__ = 'user'
13
14     id = Column(Integer, autoincrement=True, primary_key=True)
15     name = Column(Unicode(32), nullable=False, server_default='')
16
17     blog_list = relationship('Blog')
18     blog_title_list = association_proxy('blog_list', 'title')
```



`blog_list` 是一个正确的一对多关系. 下面的 `blog_title_list` 就是这个关系上的一个属性代理. `blog_title_list` 只处理 `blog_list` 这个关系中对应的对象的 `title` 属性, 包括获取和设置两个方向.



```
1 session = Session()
2
3 user = User(name='xxx')
4 user.blog_list = [Blog(title='ABC')]
```



```
5 session.add(user)
6 session.commit()
7
8 user = session.query(User).first()
9 print (user.blog_title_list)
```



上面是获取属性的示例. 在"设置", 或者说"创建"时, 直接操作是有错的:

```
1 user = session.query(User).first()
2 user.blog_title_list = ['NEW']
3 session.add(user)
4 session.commit()
```

原因在于, 对于类 Blog 的初始化形式. `association_proxy('blog_list', 'title')` 中的 `title` 只是获取时的属性定义, 而在上面的设置过程中, 实际上的调用形式为:

```
Blog('NEW')
```

Blog 类没有明确定义 `__init__()` 方法, 所有这种形式的调用会报错. 可以把 `__init__()` 方法补上: 这样调用就没有问题了.



```
1 class Blog(BaseModel):
2     __tablename__ = 'blog'
3
4     id = Column(Integer, autoincrement=True, primary_key=True)
5     title = Column(Unicode(32), nullable=False, server_default='')
6     user = Column(Integer, ForeignKey('user.id'), index=True)
7
8     def __init__(self, title):
9         self.title = title
```



另一个方法, 是在调用 `association_proxy()` 时使用 `creator` 参数明确定义"值"和"实例"的关系:



```
1 class User(BaseModel):
2     __tablename__ = 'user'
3
4     id = Column(Integer, autoincrement=True, primary_key=True)
5     name = Column(Unicode(32), nullable=False, server_default='')
6
7     blog_list = relationship('Blog')
8     blog_title_list = association_proxy('blog_list', 'title',
9                                         creator=lambda t: User(title=t))
```



`creator` 定义的方法, 返回的对象可以被对应的 `blog_list` 关系接收即可.
在查询方面, 多对一 的关系代理上, 可以直接使用属性:



```
1 class Blog(BaseModel):
2     __tablename__ = 'blog'
3
4     id = Column(Integer, autoincrement=True, primary_key=True)
5     title = Column(Unicode(32), server_default='')
6     user = Column(Integer, ForeignKey('user.id'), index=True)
7
8     user_obj = relationship('User')
9     user_name = association_proxy('user_obj', 'name')
```



查询:

```
blog = session.query(Blog).filter(Blog.user_name == 'XX').first()
```

反过来的一对多 关系代理上, 可以使用 `contains()` 函数:

```
user = session.query(User).filter(User.blogs_title.contains('A')).first()
```

5. SQLAlchemy会话与事务控制

5.1. 基本使用

SQLAlchemy 的 *session* 是用于管理数据库操作的一个像容器一样的东西. 模型实例对象本身独立存在, 而要让其修改(创建)生效, 则需要把它们加入某个 *session*. 同时你也可以把模型实例对象从 *session* 中去除. 被 *session* 管理的实例对象, 在 *session.commit()* 时被提交到数据库. 同时 *session.rollback()* 是回滚变更.

session.flush() 的作用是在事务管理内与数据库发生交互, 对应的实例状态被反映到数据库. 比如自增 ID 被填充上值.

```
1 user = User(name='名字')
2 session.add(user)
3 session.commit()
```

```
1 try:
2     user = session.Query(User).first()
3     user.name = u'改名字'
4     session.commit()
5 except:
6     session.rollback()
```

5.2. for update

SQLAlchemy 的 *Query* 支持 `select ... for update / share`.

```
session.Query(User).with_for_update().first()
session.Query(User).with_for_update(read=True).first()
```

完整形式是:

```
with_for_update(read=False, nowait=False, of=None)
```

read

是标识加互斥锁还是共享锁. 当为 True 时, 即 `for share` 的语句, 是共享锁. 多个事务可以获取共享锁, 互斥锁只能一个事务获取. 有"多个地方"都希望能"这段时间我获取的数据不能被修改, 我也不会改", 那么只能使用共享锁.

nowait

其它事务碰到锁, 是否不等待直接"报错".

of

指明上锁的表, 如果不指明, 则查询中涉及的所有表(行)都会加锁.

5.3. 事务嵌套

SQLAlchemy 中的事务嵌套有两种情况. 一是在 session 中管理的事务, 本身有层次性. 二是 session 和原始的连接 connection 之间, 是一种层次关系, 在这 session, connection 两个概念之中的事务同样具有这样的层次.

session 中的事务, 可能通过 `begin_nested()` 方法做 *savepoint* :



```
1 session.add(u1)
2 session.add(u2)
3
4 session.begin_nested()
5 session.add(u3)
6 session.rollback() # rolls back u3, keeps u1 and u2
7
8 session.commit()
```



或者使用上下文对象:



```
1 for record in records:
2     try:
3         with session.begin_nested():
4             session.merge(record)
```

```
5     except:
6         print "Skipped record %s" % record
7 session.commit()
```



嵌套的事务的一个效果,是最外层事务提交整个变更才会生效.



```
1 user = User(name='2')
2
3 session.begin_nested()
4 session.add(user)
5 session.commit()
6
7 session.rollback()
```



于是,前面说的第二种情况有一种应用方式,就是在 connection 上做一个事务,最终也在 connection 上回滚这个事务,如果 session 是 bind 到这个连接上的,那么 session 上所做的更改全部不会生效:



```
1 conn = Engine.connect()
2 session = Session(bind=conn)
3 trans = conn.begin()
4
5 user = User(name='2')
6 session.begin_nested()
7 session.add(user)
8 session.commit()
9
10 session.commit()
11
12 trans.rollback()
```



在测试中这种方式可能会有用.

5.4. 二段式提交

二段式提交, Two-Phase, 是为解决分布式环境下多点事务控制的一套协议.

与一般事务控制的不同是, 一般事务是 `begin`, 之后 `commit` 结束.

而二段式提交的流程上, `begin` 之后, 是 `prepare transaction 'transaction_id'`, 这时相关事务数据已经持久化了. 之后, 再在任何时候(哪怕重启服务), 作 `commit prepared 'transaction_id'` 或者 `rollback prepared 'transaction_id'`.

从多点事务的控制来看, 应用层要做的事是, 先把任务分发出去, 然后收集"事务准备"的状态(`prepare transaction` 的结果). 根据收集的结果决定最后是 `commit` 还是 `rollback`.

简单来说, 就是事务先保存, 再说提交的事.

SQLAlchemy 中对这个机制的支持, 是在构建会话类是加入 `twophase` 参数:

```
Session = sessionmaker(twophase=True)
```

然后会话类可以根据一些策略, 绑定多个 *Engine*, 可以是多个数据库连接, 比如:

```
Session = sessionmaker(twophase=True)
Session.configure(binds={User: Engine, Blog: Engine2})
```

这样, 在获取一个会话实例之后, 就处在二段式提交机制的支持之下, SQLAlchemy 自己会作多点的协调了. 完整的流程:



```
1 Engine = create_engine('postgresql://test@localhost:5432/test', echo=True)
2 Engine2 = create_engine('postgresql://test@localhost:5432/test2', echo=True)
3
4 Session = sessionmaker(twophase=True)
5
6 Session.configure(binds={User: Engine, Blog: Engine2})
7 session = Session()
8
```

```
9 user = User(name='名字')
10 session.add(user)
11 session.commit()
```



对应的 SQL 大概就是:

```
1 begin;
2 insert into "user" (name) values (?);
3 prepare transaction 'xx';
4 commit prepared 'xx';
```

使用时, Postgresql 数据库需要把 `max_prepared_transactions` 这个配置项的值改成大于 0

6. SQLAlchemy 字段类型

6.1. 基本类型

字段类型是在定义模型时, 对每个 Column 的类型约定. 不同类型的字段类型在输入输出上, 及支持的操作方面, 有所区别.

这里只介绍 `sqlalchemy.types.*` 中的类型, SQL 标准类型方面, 是写什么最后生成的 DDL 语句就是什么, 比如 BIGINT, BLOB 这些, 但是这些类型并不一定在所有数据库中都有支持. 除此而外, SQLAlchemy 也支持一些特定数据库的特定类型, 这些需要从具体的 dialects 实现里导入.

Integer/BigInteger/SmallInteger

整形.

Boolean

布尔类型. Python 中表现为 True/False, 数据库根据支持情况, 表现为 BOOLEAN 或 SMALLINT. 实例化时可以指定是否创建约束(默认创建).

Date/DateTime/Time (timezone=False)

日期类型, Time 和 DateTime 实例化时可以指定是否带时区信息.

Interval

时间偏差类型. 在 Python 中表现为 `datetime.timedelta()`, 数据库不支持此类型则存为日期.

*Enum (*enums, **kw)*

枚举类型, 根据数据库支持情况, SQLAlchemy 会使用原生支持或者使用 `VARCHAR` 类型附加约束的方式实现. 原生支持中涉及新类型创建, 细节在实例化时控制.

Float

浮点小数.

Numeric (precision=None, scale=None, decimal_return_scale=None, ...)

定点小数, Python 中表现为 `Decimal`.

LargeBinary (length=None)

字节数据. 根据数据库实现, 在实例化时可能需要指定大小.

PickleType

Python 对象的序列化类型.

String (length=None, collation=None, ...)

字符串类型, Python 中表现为 `Unicode`, 数据库表现为 `VARCHAR`, 通常都需要指定长度.

Unicode

类似与字符串类型, 在某些数据库实现下, 会明确表示支持非 ASCII 字符. 同时输入输出也强制是 `Unicode` 类型.

Text

长文本类型, Python 表现为 `Unicode`, 数据库表现为 `TEXT`.

UnicodeText

参考 *Unicode*.

7. SQLAlchemy混合属性机制

7.1. 直接行为

混合属性, 官方文档中称之为 *Hybrid Attributes*. 这种机制表现为, 一个属性, 在类和层面, 和实例的层面, 其行为是不同的. 之所以需要关注这部分的差异, 原因源于 Python 上下文和 SQL 上下文的差异.

类 层面经常是作为 SQL 查询时的一部分, 它面向的是 SQL 上下文. 而实例是已经得到或者创建的结果, 它面向的是 Python 上下文.

定义模型的 *Column()* 就是一个典型的混合属性. 作为实例属性时, 是具体的对象值访问, 而作为类属性时, 则有构成 SQL 语句表达式的功能.



```
1 class Interval(BaseModel):
2     __tablename__ = 'interval'
3
4     id = Column(Integer, autoincrement=True, primary_key=True)
5     start = Column(Integer)
6     end = Column(Integer)
7
8 session.add(Interval(start=0, end=100))
9 session.commit()
```



实例行为:

```
ins = session.query(Interval).first()
print (ins.end - ins.start)
```

类行为:

```
ins = session.query(Interval).filter(Interval.end - Interval.start > 10).first()
```

这种机制其实一直在被使用, 但是可能大家都没有留意一个属性在类和实例上的区别.

如果属性需要被进一步封装, 那么就需要明确声明 *Hybrid Attributes* 了:



```
1 from sqlalchemy.ext.hybrid import hybrid_property, hybrid_method
2
3 class Interval(BaseModel):
4     __tablename__ = 'interval'
```

```

5
6     id = Column(Integer, autoincrement=True, primary_key=True)
7     start = Column(Integer)
8     end = Column(Integer)
9
10    @hybrid_property
11    def length(self):
12        return self.end - self.start
13
14    @hybrid_method
15    def bigger(self, i):
16        return self.length > i
17
18
19 session.add(Interval(start=0, end=100))
20 session.commit()
21
22 ins = session.query(Interval).filter(Interval.length > 10).first()
23 ins = session.query(Interval).filter(Interval.bigger(10)).first()
24 print( ins.bigger(1))

```



setter 的定义同样使用对应的装饰器即可:



```

1 class Interval(BaseModel):
2     __tablename__ = 'interval'
3
4     id = Column(Integer, autoincrement=True, primary_key=True)
5     start = Column(Integer)
6     end = Column(Integer)
7
8     @hybrid_property
9     def length(self):
10        return abs(self.end - self.start)
11

```

```
12     @length.setter
13     def length(self, l):
14         self.end = self.start + l
```



7.2. 表达式行为

前面说的属性, 在类和实例上有不同行为, 可以看到, 在类上的行为, 其实就是生成 SQL 表达式时的行为. 上面的例子只是简单的运算, SQLAlchemy 可以自动处理好 Python 函数和 SQL 函数的区别. 但是如果是一些特性更强的 SQL 函数, 就需要手动指定了. 于时, 这时的情况变成, 实例行为是 Python 范畴的调用行为, 而类行为则是生成 SQL 函数的相关表达式.

同时是前面的例子, 对于 `length` 的定义, 更严格上来说, 应该是取绝对值的.



```
1 class Interval(BaseModel):
2     __tablename__ = 'interval'
3
4     id = Column(Integer, autoincrement=True, primary_key=True)
5     start = Column(Integer)
6     end = Column(Integer)
7
8     @hybrid_property
9     def length(self):
10         return abs(self.end - self.start)
```



但是, 如果使用了 Python 的 `abs()` 函数, 在生成 SQL 表达式时显示有无法处理了. 所以, 需要手动定义:



```
1 from sqlalchemy import func
2
3 class Interval(BaseModel):
4     __tablename__ = 'interval'
```

```

5
6     id = Column(Integer, autoincrement=True, primary_key=True)
7     start = Column(Integer)
8     end = Column(Integer)
9
10    @hybrid_property
11    def length(self):
12        return abs(self.end - self.start)
13
14    @length.expression
15    def length(self):
16        return func.abs(self.end - self.start)

```

这样查询时就可以直接使用:

```
ins = session.query(Interval).filter(Interval.length > 1).first()
```

7.3. 应用于关系

总体上没有特别之处:

```

1 class Account(BaseModel):
2     __tablename__ = 'account'
3
4     id = Column(Integer, autoincrement=True, primary_key=True)
5     user = Column(Integer, ForeignKey('user.id'), index=True)
6     balance = Column(Integer, server_default='0')
7
8
9 class User(BaseModel):
10    __tablename__ = 'user'
11
12    id = Column(Integer, autoincrement=True, primary_key=True)
13    name = Column(Unicode(32), nullable=False, server_default='')

```

```

14
15     accounts = relationship('Account')
16     #balance = association_proxy('accounts', 'balance')
17
18     @hybrid_property
19     def balance(self):
20         return sum(x.balance for x in self.accounts)

```



查询时:

```

user = session.query(User).first()
print (user.balance)

```

这里涉及的东西都是 Python 自己的, 包括那个 `sum()` 函数, 和 SQL 没有关系.

如果想实现的是, 使用 SQL 的 `sum()` 函数, 取出指定用户的总账户金额数, 那么就要考虑把 `balance` 作成表达式的形式:

```

1 from sqlalchemy import select
2
3 @hybrid_property
4 def balance(self):
5     return select([func.sum(Account.balance)]).where(Account.user == self.id).label('balance_v')
6     #return func.sum(Account.balance)

```

这样的话, `User.balance` 只是单纯的一个表达式了, 查询时指定字段:

```

user = session.query(User, User.balance).first()
print (user.balance_v)

```

注意, 如果写成:

```

session.query(User.balance).first()

```

意义就不再是"获取第一个用户的总金额", 而变成"获取总金额的第一个". 这里很坑吧.

像上面这样改, 实例层面就无法使用 `balance` 属性. 所以, 还是先前介绍的, 表达式可以单独处理:



```
1 @hybrid_property
2 def balance(self):
3     return sum(x.balance for x in self.accounts)
4
5 @balance.expression
6 def balance(self):
7     return select([func.sum(Account.balance)]).where(Account.user == self.id).label('balance_v')
```



定义了表达式的 `balance` , 这部分作为查询条件上当然也是可以的:

```
user = session.query(User).filter(User.balance > 1).first()
```