SQLAlchemy参考

2014-08-23 18:02 更新

邹业盛

1. 基本流程

```
# -*- 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('postgresql://test@localhost:5432/test')
DBSession = sessionmaker(engine)
session = DBSession()

BaseModel = declarative_base()

class User(BaseModel):
    __tablename__ = 'user'

    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)

query = session.query(Session, User.username).join(User, User.id == Session.user)
for i in query:
    print dir(i)
```

2. 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('postgresql+psycopg2://scott:tiger@localhost/mydatabase')
```

参数字符串的各部分的意义:

```
dialect+driver://username:password@host:port/database
```

对于这个字符串, SQLAlchemy 提供了工具可用于处理它:

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

from sqlalchemy import create_engine
from sqlalchemy.engine.url import make_url
from sqlalchemy.engine.url import URL

s = 'postgresql://test@localhost:5432/bbcustom'
url = make_url(s)
s = URL(drivername='postgresql', username='test', password="",
        host="localhost", port=5432, database="bbcustom")

engine = create_engine(url)
engine = create_engine(s)

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:

```
def f(sql, *args, **kargs):
    print sql, args, kargs

s = 'postgresql://test@localhost:5432/bbcustom'
engine = create_engine(s, strategy='mock', executor=f)

print engine.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` :

```
s = 'postgresql+psycopg2://test@localhost:5432/bbcustom'
s = 'postgresql+pg8000://test@localhost:5432/bbcustom'
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.engine.url import make_url
from sqlalchemy.engine.url import URL

s = 'postgresql://test@localhost:5432/bbcustom'
engine = create_engine(s, pool_size=2, max_overflow=0)


from threading import Thread

def f():
    print engine.execute('select pg_sleep(5)').fetchall()


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

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

连接池的实现, 在 `create_engine` 调用时也可以指定:

```
from sqlalchemy.pool import QueuePool
engine = create_engine('sqlite:///file.db', poolclass=QueuePool)
```

还有:

```
from sqlalchemy.pool import NullPool
engine = create_engine(
         'postgresql+psycopg2://scott:tiger@localhost/test',
          poolclass=NullPool)
```

或者仅仅是获取连接的方法:

```
import sqlalchemy.pool as pool
import psycopg2

def getconn():
    c = psycopg2.connect(username='ed', host='127.0.0.1', dbname='test')
    # do things with 'c' to set up
    return c
```

```
engine = create_engine('postgresql+psycopg2://', creator=getconn)
```

连接池可以被单独使用:

```
import sqlalchemy.pool as pool
import psycopg2

def getconn():
    c = psycopg2.connect(username='ed', host='127.0.0.1', dbname='test')
    return c

mypool = pool.QueuePool(getconn, max_overflow=10, pool_size=5)

conn = mypool.connect()
cursor = conn.cursor()
cursor.execute("select foo")
```

连接池可以被多个 engine 共享使用:

```
e = create_engine('postgresql://', pool=mypool)
```

3. 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', metadata,
    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)
)
```

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

from sqlalchemy import create_engine
from sqlalchemy import Column
from sqlalchemy.types import String, Integer, CHAR, BIGINT
from sqlalchemy.ext.declarative import declarative_base
BaseModel = declarative_base()
Engine = create_engine('postgresql://test@localhost:5432/test', echo=True)


class Blog(BaseModel):
    __tablename__ = 'blog'

    id = Column(CHAR(32), primary_key=True)
    title = Column(String(64), server_default='', nullable=False)
    text = Column(String, server_default='', nullable=False)
    user = Column(CHAR(32), index=True, server_default='', nullable=False)
    create = Column(BIGINT, index=True, server_default='0', nullable=False)
```

```
class User(BaseModel):
    __tablename__ = 'user'

    id = Column(CHAR(32), primary_key=True)
    name = Column(String(32), server_default='', nullable=False)
    username = Column(String(32), index=True, server_default='', nullable=False)
    password = Column(String(64), server_default='', nullable=False)


def init_db():
    BaseModel.metadata.create_all(Engine)

def drop_db():
    BaseModel.metadata.drop_all(Engine)


if __name__ == '__main__':
    #init_db()
    drop_db()
    #BaseModel.metadata.tables['user'].create(Engine, checkfirst=True)
    #BaseModel.metadata.tables['user'].drop(Engine, checkfirst=False)
    pass
```

3.2. 创建

```
session = Session()
session.add(User(id=uuid.uuid4().hex))
session.add(Blog(id=uuid.uuid4().hex))
session.add_all([
    User(id=uuid.uuid4().hex),
    Blog(id=uuid.uuid4().hex)
])
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

```
session.query(User).filter_by(username='abc').all()
session.query(User).filter(User.username=='abc').all()
session.query(Blog).filter(Blog.create >= 0).all()
session.query(Blog).filter(Blog.create >= 0).first()
session.query(Blog).filter(Blog.create >= 0 | Blog.title == 'A').first()
session.query(Blog).filter(Blog.create >= 0 & Blog.title == 'A').first()
session.query(Blog).filter(Blog.create >= 0).offset(1).limit(1).scalar()
session.query(User).filter(User.username ==  'abc').scalar()
```

```
session.query(User.id).filter(User.username ==  'abc').scalar()
session.query(Blog.id).filter(Blog.create >= 0).all()
session.query(Blog.id).filter(Blog.create >= 0).all()[0].id
dict(session.query(Blog.id, Blog.title).filter(Blog.create >= 0).all())
session.query(Blog.id, Blog.title).filter(Blog.create >= 0).first().title
session.query(User.id).order_by('id desc').all()
session.query(User.id).order_by('id').first()
session.query(User.id).order_by(User.id).first()
session.query(User.id).order_by(-User.id).first()
session.query('id', 'username').select_from(User).all()
session.query(User).get('16e19a64d5874c308421e1a835b01c69')
```

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 == '16e19a64d5874c308421e1a835b01c69')).all()
session.query(User).filter(not_(User.id == '16e19a64d5874c308421e1a835b01c69')).all()
session.query(User).filter(User.id.in_(['16e19a64d5874c308421e1a835b01c69'])).all()
session.query(User).filter(User.id.like('16e19a%')).all()
session.query(User).filter(User.id.startswith('16e19a')).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:

```
session.query(User).filter(User.username == 'abc').update({'name': '123'})
session.commit()

user = session.query(User).filter_by(username='abc').scalar()
user.name = '223'
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:

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


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

4. 外键和关系定义

4.1. 外键约束

使用 ForeignKey 来定义一个外键约定:

```
from sqlalchemy import Column, ForeignKey
from sqlalchemy.types import String, Integer, CHAR, BIGINT

class Blog(BaseModel):
    __tablename__ = 'blog'

    id = Column(BIGINT, primary_key=True, autoincrement=True)
    title = Column(String(64), server_default='', nullable=False)
    text = Column(String, server_default='', nullable=False)
    user = Column(BIGINT, ForeignKey('user.id'), index=True, nullable=False)
    create = Column(BIGINT, index=True, server_default='0', nullable=False)


class User(BaseModel):
    __tablename__ = 'user'

    id = Column(BIGINT, primary_key=True, autoincrement=True)
    name = Column(String(32), server_default='', nullable=False)
    username = Column(String(32), index=True, server_default='', nullable=True)
    password = Column(String(64), server_default='', nullable=False)
```

创建时:

```
session = Session()
user = User(name='first', username=u'新的')
session.add(user)
session.flush()
blog = Blog(title=u'第一个', user=user.id)
session.add(blog)
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.

```
from sqlalchemy import Column, ForeignKey
from sqlalchemy.types import String, Integer, CHAR, BIGINT
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker, relationship


class Blog(BaseModel):
    __tablename__ = 'blog'

    id = Column(BIGINT, primary_key=True, autoincrement=True)
    title = Column(String(64), server_default='', nullable=False)
    user = Column(BIGINT, ForeignKey('user.id'), index=True, nullable=False)
    create = Column(BIGINT, index=True, server_default='0', nullable=False)

    user_obj = relationship('User')


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', order_by='Blog.create')
```

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:

```
session = Session()
print session.query(Blog).get(1).user_obj
print session.query(User).get(1).blog_list
```

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：

```
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', order_by='Blog.create', lazy="dynamic")
```

这样在获取实例时就可以自由控制了:

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

```
class Blog(BaseModel):
    __tablename__ = 'blog'

    id = Column(Integer, autoincrement=True, primary_key=True)
    title = Column(Unicode(32), server_default='')
    user = Column(Integer, ForeignKey('user.id'), index=True)

    user_obj = relationship('User')


class User(BaseModel):
    __tablename__ = 'user'

    id = Column(Integer, autoincrement=True, primary_key=True)
    name = Column(Unicode(32), server_default='')

    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 == u'A')).first()
```

SQLAlchemy 会使用 `exists` 条件，类似于：

```
SELECT *
FROM user
WHERE EXISTS
    (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 == u'XX')).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:

```
class Blog(BaseModel):
    __tablename__ = 'blog'

    id = Column(BIGINT, primary_key=True, autoincrement=True)
    title = Column(String(64), server_default='', nullable=False)
    user = Column(BIGINT, ForeignKey('user.id'), index=True, nullable=False)

    user_obj = relationship('User', lazy='joined', cascade='all')


class User(BaseModel):
    __tablename__ = 'user'

    id = Column(BIGINT, primary_key=True, autoincrement=True)
    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:

```
blog_list = relationship('Blog', lazy="dynamic")
user_obj = relationship('User', lazy='joined')
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:

```
session.query(Parent).options(
    joinedload('foo').joinedload('bar').joinedload('bat')
    ).all()

session.query(A).options(
    defaultload("atob").joinedload("btoc")
    ).all()

session.query(MyClass).options(lazyload('*'))

session.query(MyClass).options(
    lazyload('*'), joinedload(MyClass.widget)
    )

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:

```
from sqlalchemy.orm import contains_eager

blog = session.query(Blog).join(Blog.user_obj)\
              .options(contains_eager(Blog.user_obj)).first()
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.

```
class Blog(BaseModel):
    __tablename__ = 'blog'

    id = Column(Integer, autoincrement=True, primary_key=True)
    title = Column(Unicode(32), server_default='')
    user = Column(Integer, ForeignKey('user.id'), index=True)

    user_obj = relationship('User')


class User(BaseModel):
    __tablename__ = 'user'

    id = Column(Integer, autoincrement=True, primary_key=True)
    name = Column(Unicode(32), server_default='')

    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:

```
user = User(name=u'XX')
session.add_all([Blog(title=u'A', user_obj=user), Blog(title=u'B', user_obj=user)])
session.commit()

user = session.query(User).first()
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:

```
from sqlalchemy.orm.collections import attribute_mapped_collection

blogs = relationship('Blog', collection_class=attribute_mapped_collection('title'))

#{u'A': <__main__.Blog object at 0x20ed810>, u'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:

```
from sqlalchemy.orm.collections import mapped_collection

blogs = relationship('Blog', collection_class=mapped_collection(lambda blog: blog.title.lower()))

#{u'a': <__main__.Blog object at 0x1de4890>, u'b': <__main__.Blog object at 0x1de4950>}
```

4.6. 多对多关系

先考虑典型的多对多关系结构:

```
class Blog(BaseModel):
    __tablename__ = 'blog'

    id = Column(BIGINT, primary_key=True, autoincrement=True)
    title = Column(String(64), server_default='', nullable=False)

    tag_list = relationship('Tag')
```

```
    tag_list = relationship('BlogAndTag')


class Tag(BaseModel):
    __tablename__ = 'tag'

    id = Column(BIGINT, primary_key=True, autoincrement=True)
    name = Column(String(16), server_default='', nullable=False)


class BlogAndTag(BaseModel):
    __tablename__ = 'blog_and_tag'

    id = Column(BIGINT, primary_key=True, autoincrement=True)
    blog = Column(BIGINT, ForeignKey('blog.id'), index=True)
    tag = Column(BIGINT, ForeignKey('tag.id'), index=True)
    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:

```
class Blog(BaseModel):
    __tablename__ = 'blog'

    id = Column(BIGINT, primary_key=True, autoincrement=True)
    title = Column(String(64), server_default='', nullable=False)

    tag_list = relationship('Tag', secondary=lambda: BlogAndTag.__table__)


class Tag(BaseModel):
    __tablename__ = 'tag'

    id = Column(BIGINT, primary_key=True, autoincrement=True)
    name = Column(String(16), server_default='', nullable=False)


class BlogAndTag(BaseModel):
    __tablename__ = 'blog_and_tag'

    id = Column(BIGINT, primary_key=True, autoincrement=True)
    blog = Column(BIGINT, ForeignKey('blog.id'), index=True)
    tag = Column(BIGINT, ForeignKey('tag.id'), index=True)
    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 做的是一个普通的多表查询.

`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:

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

tag = session.query(Tag).filter(Tag.name == 'x').one()
blog.tag_list.remove(tag)
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:

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

tag = session.query(Tag).filter(Tag.name == 'ok').one()
session.delete(tag)
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 .

```
class Blog(BaseModel):
    __tablename__ = 'blog'

    id = Column(BIGINT, primary_key=True, autoincrement=True)
    title = Column(String(64), server_default='', nullable=False)
    user = Column(BIGINT, ForeignKey('user.id'), index=True, nullable=False)


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='')
    blog_list_auto = relationship('Blog', cascade='save-update')


if __name__ == '__main__':

    session = Session()

    user = User(name=u'哈哈')
    blog = Blog(title=u'第一个')
    user.blog_list = [blog]
    #user.blog_list_auto = [blog]
    session.add(user)
    print blog in session
    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.

```
class Blog(BaseModel):
    __tablename__ = 'blog'

    id = Column(BIGINT, primary_key=True, autoincrement=True)
    title = Column(String(64), server_default='', nullable=False)
    user = Column(BIGINT, ForeignKey('user.id'), index=True, nullable=False)


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='save-update, delete')
```

```
if __name__ == '__main__':
    session = Session()

    #user = User(name=u'用户')
    #user.blog_list = [Blog(title=u'哈哈')]
    #session.add(user)
    user = session.query(User).first()
    session.delete(user)
    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` 参数来强制约束. 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.

```
class Blog(BaseModel):
    __tablename__ = 'blog'

    id = Column(BIGINT, primary_key=True, autoincrement=True)
    title = Column(String(64), server_default='', nullable=False)
    user = Column(BIGINT, ForeignKey('user.id'), index=True, nullable=False)


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='save-update, delete-orphan')


if __name__ == '__main__':

    session = Session()

    #user = User(name=u'用户')
    #blog = Blog(title=u'一')
    #blog2 = Blog(title=u'二')
    #user.blog_list = [blog, blog2]
    #session.add(user)
    user = session.query(User).first()
    blog2 =  session.query(Blog).filter(Blog.title == u'二').first()
    user.blog_list = [blog2]
    #session.delete(user)
    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":

```
user = User(id=1, name="1")
session.add(user)
session.commit()

user = User(id=1)
user = session.merge(user)
print user.name

user = User(id=1, name="2")
user = session.merge(user)
session.commit()
```

`cascade` 中的 `merge` 作用:

```
class Blog(BaseModel):
    __tablename__ = 'blog'

    id = Column(BIGINT, primary_key=True, autoincrement=True)
    title = Column(String(64), server_default='', nullable=False)
    user = Column(BIGINT, ForeignKey('user.id'), index=True, nullable=False)


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='save-update, delete, delete-orphan, merge')


if __name__ == '__main__':

    session = Session()

    user = User(id=1, name='1')
    session.add(user)
    session.commit(user)

    user = User(id=1, blog_list=[Blog(title='哈哈')])
    session.merge(user)

    session.commit()
```

refresh-expire

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).

```
class Blog(BaseModel):
    __tablename__ = 'blog'

    id = Column(BIGINT, primary_key=True, autoincrement=True)
    title = Column(String(64), server_default='', nullable=False)
    user = Column(BIGINT, ForeignKey('user.id'), index=True, nullable=False)
```

```
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='save-update, delete, delete-orphan, merge, refresh-expire')


if __name__ == '__main__':

    session = Session()

    #user = User(id=1, name='1')
    #blog = Blog(title="abc")
    #user.blog_list = [blog]
    #session.add(user)

    user = session.query(User).first()
    blog = user.blog_list[0]
    print user.name
    print blog.title
    session.expire(user)
    print 'EXPIRE'
    print user.name
    print blog.title

    session.commit()
```

expunge

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.

```
class Blog(BaseModel):
    __tablename__ = 'blog'

    id = Column(BIGINT, primary_key=True, autoincrement=True)
    title = Column(String(64), server_default='', nullable=False)
    user = Column(BIGINT, ForeignKey('user.id'), index=True, nullable=False)


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='delete, delete-orphan, expunge')


if __name__ == '__main__':

    session = Session()
    user = User(name=u'用户')
    blog = Blog(title=u'第一个')
    user.blog_list = [blog]

    session.add(user)
    session.add(blog)

    session.expunge(user)
    print blog in session
```

```
        #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:

```
from sqlalchemy.ext.associationproxy import association_proxy

class Blog(BaseModel):
    __tablename__ = 'blog'

    id = Column(Integer, autoincrement=True, primary_key=True)
    title = Column(Unicode(32), nullable=False, server_default='')
    user = Column(Integer, ForeignKey('user.id'), index=True)


class User(BaseModel):
    __tablename__ = 'user'

    id = Column(Integer, autoincrement=True, primary_key=True)
    name = Column(Unicode(32), nullable=False, server_default='')

    blog_list = relationship('Blog')
    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.

```
session = Session()

user = User(name='xxx')
user.blog_list = [Blog(title='ABC')]
session.add(user)
session.commit()

user = session.query(User).first()
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:

```
user = session.query(User).first()
user.blog_title_list = ['NEW']
session.add(user)
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:

```
class Blog(BaseModel):
    __tablename__ = 'blog'

    id = Column(Integer, autoincrement=True, primary_key=True)
    title = Column(Unicode(32), nullable=False, server_default='')
    user = Column(Integer, ForeignKey('user.id'), index=True)

    def __init__(self, title):
        self.title = title
```

这样调用就没有问题了.

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:

```
class User(BaseModel):
    __tablename__ = 'user'

    id = Column(Integer, autoincrement=True, primary_key=True)
    name = Column(Unicode(32), nullable=False, server_default='')

    blog_list = relationship('Blog')
    blog_title_list = association_proxy('blog_list', 'title',
                                        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:

```
class Blog(BaseModel):
    __tablename__ = 'blog'

    id = Column(Integer, autoincrement=True, primary_key=True)
    title = Column(Unicode(32), server_default='')
    user = Column(Integer, ForeignKey('user.id'), index=True)

    user_obj = relationship('User')
    user_name = association_proxy('user_obj', 'name')
```

查询:

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

```
user = User(name=u'名字')
session.add(user)
session.commit()
```

```
try:
    user = session.Query(User).first()
    user.name = u'改名字
    session.commit()
except:
    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"都希望是"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 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：

```
session.add(u1)
session.add(u2)

session.begin_nested()
session.add(u3)
session.rollback() # rolls back u3, keeps u1 and u2

session.commit()
```

或者使用上下文对象:

```
for record in records:
    try:
        with session.begin_nested():
            session.merge(record)
    except:
        print "Skipped record %s" % record
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.

```
user = User(name='2')

session.begin_nested()
session.add(user)
session.commit()

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:

```
conn = Engine.connect()
session = Session(bind=conn)
trans = conn.begin()

user = User(name='2')
session.begin_nested()
session.add(user)
session.commit()

session.commit()

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:

```
Engine = create_engine('postgresql://test@localhost:5432/test', echo=True)
Engine2 = create_engine('postgresql://test@localhost:5432/test2', echo=True)

Session = sessionmaker(twophase=True)

Session.configure(binds={User: Engine, Blog: Engine2})
session = Session()

user = User(name=u'名字')
session.add(user)
session.commit()
```

对应的 SQL 大概就是:

```
begin;
insert into "user" (name) values (?);
prepare transaction 'xx';
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. 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`，`BLOG` 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. 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.

```
class Interval(BaseModel):
    __tablename__ = 'interval'

    id = Column(Integer, autoincrement=True, primary_key=True)
    start = Column(Integer)
    end = Column(Integer)

session.add(Interval(start=0, end=100))
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:

```
from sqlalchemy.ext.hybrid import hybrid_property, hybrid_method

class Interval(BaseModel):
    __tablename__ = 'interval'

    id = Column(Integer, autoincrement=True, primary_key=True)
    start = Column(Integer)
    end = Column(Integer)

    @hybrid_property
    def length(self):
        return self.end - self.start
```

```
    @hybrid_method
    def bigger(self, i):
        return self.length > i


session.add(Interval(start=0, end=100))
session.commit()

ins = session.query(Interval).filter(Interval.length > 10).first()
ins = session.query(Interval).filter(Interval.bigger(10)).first()
print ins.bigger(1)
```

setter 的定义同样使用对应的装饰器即可:

```
class Interval(BaseModel):
    __tablename__ = 'interval'

    id = Column(Integer, autoincrement=True, primary_key=True)
    start = Column(Integer)
    end = Column(Integer)

    @hybrid_property
    def length(self):
        return abs(self.end - self.start)

    @length.setter
    def length(self, l):
        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.

```
class Interval(BaseModel):
    __tablename__ = 'interval'

    id = Column(Integer, autoincrement=True, primary_key=True)
    start = Column(Integer)
    end = Column(Integer)

    @hybrid_property
    def length(self):
        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:

```
from sqlalchemy import func

class Interval(BaseModel):
    __tablename__ = 'interval'
```

```
    id = Column(Integer, autoincrement=True, primary_key=True)
    start = Column(Integer)
    end = Column(Integer)

    @hybrid_property
    def length(self):
        return abs(self.end - self.start)

    @length.expression
    def length(self):
        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()
```

对应的 SQL :

```
SELECT *
FROM interval
WHERE abs(interval."end" - interval.start) > ?
 LIMIT ? OFFSET ?
```

7.3. 应用于关系

总体上没有特别之处:

```
class Account(BaseModel):
    __tablename__ = 'account'

    id = Column(Integer, autoincrement=True, primary_key=True)
    user = Column(Integer, ForeignKey('user.id'), index=True)
    balance = Column(Integer, server_default='0')


class User(BaseModel):
    __tablename__ = 'user'

    id = Column(Integer, autoincrement=True, primary_key=True)
    name = Column(Unicode(32), nullable=False, server_default='')

    accounts = relationship('Account')
    #balance = association_proxy('accounts', 'balance')

    @hybrid_property
    def balance(self):
        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：

```
from sqlalchemy import select

@hybrid_property
def balance(self):
    return select([func.sum(Account.balance)]).where(Account.user == self.id).label('balance_v')
    #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：

```
@hybrid_property
def balance(self):
    return sum(x.balance for x in self.accounts)

@balance.expression
def balance(self):
    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()
```

8. 示例: AdjacencyList, 单向链接列表

这里说的 AdjacencyList ，就是最常用来在关系数据库中表示树结构的，parent 方式：

id	name	parent
1	一	null
2	二	1
3	三	2

上面的数据，表示的结构就是：

```
一
  |- 二
    |- 三
```

模型定义很好做:

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

from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Column, ForeignKey
from sqlalchemy.types import Integer, Unicode
from sqlalchemy.orm import relationship, sessionmaker, joinedload

BaseModel = declarative_base()
Engine = create_engine('sqlite://', echo=True)
Session = sessionmaker(Engine)

class Node(BaseModel):
    __tablename__ = 'node'

    id = Column(Integer, autoincrement=True, primary_key=True)
    name = Column(Unicode(32), nullable=False, server_default='')
    parent = Column(Integer, ForeignKey('node.id'), index=True,
                    nullable=False, server_default='0')
```

这里不让 `parent` 字段有 `null` , 而使用 `0` 代替.

这个例子在关系上, 有一个纠结的地方, 因为 node 这个表, 它是自关联的, 所以如果想要 `children` 和 `parent_obj` 这两个关系时:

```
children = relationship('Node')
parent_obj = relationship('Node')
```

呃, 尴尬了.

如果是两个表, 那么 SQLAlchemy 可以通过外键在哪张表这个信息, 来确定关系的方向:

```
class Blog(BaseModel):
    ...
    user = Column(Integer, ForeignKey('user.id'))
    user_obj = relationship('User')

class User(BaseModel):
    ...
    blog_list = relationship('Blog')
```

因为外键在 `Blog` 中, 所以 Blog -> User 的 `user_obj` 是一个 N -> 1 关系.

反之, User -> Blog 的 `blog_list` 则是一个 1 -> N 的关系.

而自相关的 `Node` 无法直接判断方向, 所以 SQLAlchemy 会按 1 -> N 处理, 那么:

```
children = relationship('Node')
parent_obj = relationship('Node')
```

这两条之中，`children` 是正确的，是我们想要的. 要定义 `parent_obj` 则需要在 `relationship` 中通过参数明确表示方向:

```
parent_obj = relationship('Node', remote_side=[id])
```

这种方式就定义了一个，"到 id" 的 N -> 1 关系.

现在完整的模型定义是:

```
class Node(BaseModel):
    __tablename__ = 'node'

    id = Column(Integer, autoincrement=True, primary_key=True)
    name = Column(Unicode(32), nullable=False, server_default='')
    parent = Column(Integer, ForeignKey('node.id'), index=True,
                    nullable=False, server_default='0')

    children = relationship('Node') # 1 -> N
    parent_obj = relationship('Node', remote_side=[id])
```

查询方面没什么特殊的了，不过我发现在自相关的模型关系，`lazy` 选项不起作用:

```
children = relationship('Node', lazy="joined")
parent_obj = relationship('Node', remote_side=[id], lazy="joined")
```

都是无效的，只有在查询时，手动使用 `options()` 定义:

```
n = session.query(Node).filter(Node.name==u'一')\
            .options(joinedload('parent_obj')).first()
```

如果要一次查出多级的子节点:

```
n = session.query(Node).filter(Node.name==u'一')\
            .options(joinedload('children').joinedload('children')).first()
print n.name, n.children, n.children[0].children
```

多个 `joinedload()` 串连的话，可以使用 `joinedload_all()` 来整合:

```
from sqlalchemy.orm import joinedload_all

n = session.query(Node).filter(Node.name==u'一')\
            .options(joinedload_all('children', 'children')).first()
```

在修改方面，删除的话，配置了 `cascade` ，删除父节点，则子节点也会自动删除:

```
children = relationship('Node', lazy='joined', cascade='all') # 1 -> N
node = session.query(Node).filter(Node.name == u'一').first()
session.delete(node)
session.commit()
```

如果只删除子节点，那么 delete-orphan 选项就很好用了：

```
children = relationship('Node', lazy='joined', cascade='all, delete-orphan') # 1 -> N
node = session.query(Node).filter(Node.name == u'一').first()
node.children = []
session.commit()
```

9. 示例：属性实体化建模

假设有这样的场景，某实体在具体条目上，其属性是不定的，或者其属性是充分稀疏的：

id	name	attr_0	attr_1	attr_2	...	attr_n
1	foo	1	abc	33	...	any

这种情况下，把属性看成是单独的实体，是一个更好的建模方式：

id	name
1	foo

id	entity_id	attr_name	attr_value
1	1	attr_0	1
2	1	attr_1	33
...	...	...	...
n	1	attr_n	any

这种模型下，ORM 层面我们考虑封装一个对操作更友好的上层操作接口，比如：

```
obj = Entity()
obj['attr_0'] = '1'
obj['attr_1'] = '33'
session.add(obj)
session.commit()
```

实现上，就是把对象的方法，包装成 SQLAlchemy 的 ORM 中的对应的关系操作.

```
class BaseModel(declarative_base()):
    __abstract__ = True

class Entity(BaseModel):
    __tablename__ = 'entity'

    id = Column(Integer, autoincrement=True, primary_key=True)
    name = Column(Unicode(32), server_default='')

    _attributes = relationship('Attribute',
                               collection_class=attribute_mapped_collection('key'),
                               lazy='joined', cascade="all, delete-orphan")
    attributes = association_proxy('_attributes', 'value',
                                   creator=lambda k, v: Attribute(key=k, value=v))
```

```
class Attribute(BaseModel):
    __tablename__ = 'attribute'

    id = Column(Integer, autoincrement=True, primary_key=True)
    entity = Column(Integer, ForeignKey('entity.id'), index=True)
    key = Column(Unicode(32), server_default='')
    value = Column(UnicodeText, server_default='')


if __name__ == '__main__':
    BaseModel.metadata.create_all(Engine)

    session = Session()

    entity = Entity(name=u'哈哈')
    entity.attributes[u'first'] = u'abc'
    entity.attributes[u'sec'] = u'hoho'
    session.add(entity)
    session.commit()

    entity = session.query(Entity).first()
    print entity.attributes
    del entity.attributes['first']
    session.commit()

    entity = session.query(Entity).first()
    print entity.attributes
```

实现上就两点:

- `_attributes` 关系中, 指定 `collection_class` , 于是就可以得到一个像 `dict` 的属性对象了.
- `association_proxy` 从 `dict` 的属性对象中只抽出我们关心的 `value` 属性值.

这个场景中, 还可以再进一步, 在 `Entity` 类上实现 `dict` 的一些方法, 直接操作其 `attributes` 属性, `association_proxy` 就直接返回 `Entity` 的实例, 这样代码可以变成这样:

```
entity = Entity(name=u'ABC')
entity[u'first'] = u'a'
entity[u'sec'] = u'hoho'
```

评论