SQLAlchemy 最新权威详细教程

前言:最近开始学习 SQLAlchemy,本教程是其官方文档以及在读英文版<Essential SQLAlchemy>的翻译加一些自己的理解和总结

1 什么是 SQLAlchemy?

它是给 mysql, oracle,sqlite 等关系型数据库的 python 接口,不需要大幅修改原有的 python 代码,它已经包含了 SQL 表达式语言和 ORM,看一些例子:

```
sql=”INSERT INTO user(user_name, password) VALUES (%s, %s)”
cursor = conn.cursor()
cursor.execute(sql, (‘dongwm’, ‘testpass’))
```

以上是一个常用的mysql的SQL语句,但是冗长也容易出错,并且可能导致安全问题(因为是字符串的语句,会存在 SQL 注入),并且代码不跨平台,在不同数据库软件的语句不同(以下是一个 Oracle 例子),不具备客移植性:

```
sql=”INSERT INTO user(user_name, password) VALUES (:1, :2)”
cursor = conn.cursor()
cursor.execute(sql, ‘dongwm’, ‘testpass’)
```

而在 SQLAlchemy 里只需要这样写:

```
statement = user_table.insert(user_name=’rick’, password=’parrot’)
statement.execute()   #护略是什么数据库环境
```

SQLAlchemy 还能让你写出很 pythonic 的语句:

```
statement = user_table.select(and_(
user_table.c.created >= date(2007,1,1),
user_table.c.created < date(2008,1,1))
result = statement.execute()   #检索所有在 2007 年创建的用户

metadata=MetaData(‘sqlite://’) # 告诉它你设置的数据库类型是基于内存的 sqlite
user_table = Table(   #创建一个表
 ‘tf_user’, metadata,
Column(‘id’, Integer, primary_key=True),   #一些字段,假设你懂 SQL,那么以下的字段很好理解
Column(‘user_name’, Unicode(16), unique=True, nullable=False),
Column(‘email_address’, Unicode(255), unique=True, nullable=False),
Column(‘password’, Unicode(40), nullable=False),
Column(‘first_name’, Unicode(255), default=”),
```

Column(‘last_name’ , Unicode(255), default=”),
Column(‘created’ , DateTime, default=datetime.now))

users_table = Table(‘users’ , metadata, autoload=True) #假设 table 已经存在.就不需要指定字段,只是加个 autoload=True

class User(object): pass #虽然 SQLAlchemy 强大,但是插入更新还是需要手动指定,可以使用 ORM,方法就是:设定一个类,定义一个表,把表映射到类里面
mapper(User, user_table)

下面是一个完整 ORM 的例子:

Source code

```
from sqlalchemy.orm import mapper, sessionmaker

#sessionmaker() 函数是最常使用的创建最顶层可用于整个应用 Session

的方法,Session 管理着所有与数据库之间的会话

from datetime import datetime
from sqlalchemy import Table, MetaData, Column, ForeignKey,

Integer, String, Unicode, DateTime #会 SQL 的人能理解这些函数吧?

engine = create_engine("sqlite:///tutorial.db", echo=True)

#创建到数据库的连接,echo=True 表示用 logging 输出调试结果

metadata = MetaData() #跟踪表属性

user_table = Table( #创建一个表所需的信息:字段,表名等

'tf_user', metadata,
Column('id', Integer, primary_key=True),
Column('user_name', Unicode(16), unique=True,
nullable=False),
Column('email_address', Unicode(255), unique=True,
nullable=False),
Column('password', Unicode(40), nullable=False),
Column('first_name', Unicode(255), default=''),
Column('last_name', Unicode(255), default=''),
Column('created', DateTime, default=datetime.now))

metadata.create_all(engine)  #在数据库中生成表

class User(object): pass #创建一个映射类

mapper(User, user_table) #把表映射到类
```

```
Session = sessionmaker() #创建了一个自定义了的 Session 类

Session.configure(bind=engine)  #将创建的数据库连接关联到这个
session
session = Session()
u = User()
u.user_name='dongwm'
u.email_address='dongwm@dongwm.com'
u.password='testpass'  #给映射类添加以下必要的属性,因为上面创建
表指定这几个字段不能为空
session.add(u)  #在 session 中添加内容


session.flush() #保存数据

session.commit() #数据库事务的提交,sisson 自动过期而不需要关闭


query = session.query(User) #query() 简单的理解就是 select() 的
支持 ORM 的替代方法,可以接受任意组合的 class/column 表达式

print list(query) #列出所有 user

print query.get(1) #根据主键显示

print query.filter_by(user_name='dongwm').first()  #类似于
SQL 的 where,打印其中的第一个
u = query.filter_by(user_name='dongwm').first()
u.password = 'newpass' #修改其密码字段

session.commit() #提交事务

print query.get(1).password #打印会出现新密码

for instance in session.query(User).order_by(User.id):  #
根据 id 字段排序,打印其中的用户名和邮箱地址
    print instance.user_name, instance.email_address
```

既然是ORM框架,我们来一个更复杂的包含关系的例子,先看sql语句:

```
CREATE TABLE tf_user (
id INTEGER NOT NULL,
user_name VARCHAR(16) NOT NULL,
email_address VARCHAR(255) NOT NULL,
password VARCHAR(40) NOT NULL,
first_name VARCHAR(255),
last_name VARCHAR(255),
created TIMESTAMP,
PRIMARY KEY (id),
UNIQUE (user_name),
UNIQUE (email_address));
CREATE TABLE tf_group (
id INTEGER NOT NULL,
group_name VARCHAR(16) NOT NULL,
PRIMARY KEY (id),
UNIQUE (group_name));
CREATE TABLE tf_permission (
id INTEGER NOT NULL,
permission_name VARCHAR(16) NOT NULL,
PRIMARY KEY (id),
UNIQUE (permission_name));
CREATE TABLE user_group (
user_id INTEGER,
group_id INTEGER,
PRIMARY KEY(user_id, group_id),
FOREIGN KEY(user_id) REFERENCES tf_user (id), #user_group的user_id关联了tf_user的id字段
FOREIGN KEY(group_id) REFERENCES tf_group (id));   #group_id关联了tf_group 的id字段

CREATE TABLE group_permission (
group_id INTEGER,
permission_id INTEGER,
PRIMARY KEY(group_id, permission_id),
FOREIGN KEY(group_id) REFERENCES tf_group (id),   #group_permission的id关联 tf_group的id字段
FOREIGN KEY(permission_id) REFERENCES tf_permission (id));
#permission_id关联了tf_permission 的id字段
```

这是一个复杂的多对多的关系,比如检查用户是否有admin权限,sql需要这样:

```
SELECT COUNT(*) FROM tf_user, tf_group, tf_permission WHERE
tf_user.user_name=’dongwm’ AND tf_user.id=user_group.user_id
```

```
AND user_group.group_id = group_permission.group_id
AND group_permission.permission_id = tf_permission.id
AND permission_name=’admin’;   看起来太复杂并且繁长了
```

在面向对象的世界里,是这样的:

```
class User(object):
groups=[]
class Group(object):
users=[]
permissions=[]
class Permission(object):
groups=[]
```

Source code

```
print 'Summary for %s' % user.user_name
for g in user.groups:
    print ' Member of group %s' % g.group_name
    for p in g.permissions:
        print '... which has permission %s' % p.permission_name
```

Source code

```
def user_has_permission(user, permission_name):  #检查用户是
```

否有 permission_name 的权限的函数

```
    for g in user.groups:
        for p in g.permissions: #可以看出来使用了 for 循环
            if p.permission_name == 'admin':
                return True
    return False
```

而在 SQLAlchemy 中,这样做:

```
mapper(User, user_table, properties=dict(
groups=relation(Group, secondary=user_group, backref=’users’)))
#properties 是一个字典值。增加了一个 groups 值，它又是一个 relation 对象，这个对象实现
```

#了 Group 类与 user_group 的 映射。这样我通过 user_table 的 groups 属性就可以反映出 RssFeed 的值来,

```
#中间表对象(user_group)传给 secondary 参数，backref 为自己的表(users)
mapper(Group, group_table, properties=dict(
permissions=relation(Permission, secondary=group_permission,
backref=’ groups’ )))
mapper(Permission, permission_table)



q = session.query(Permission)
dongwm_is_admin =
q.count_by(permission_name=’ admin’ ,user_name=’ dongwm’ )
```

假如计算组里用户数(不包含忘记删除但是重复的)

```
for p in permissions:
users = set()
for g in p.groups:
for u in g.users:
users.add(u)
print ‘Permission %s has %d users’ % (p.permission_name, len(users))
```

在 SQLAlchemy 可以这样:

```
q=select([Permission.c.permission_name,
func.count(user_group.c.user_id)],
and_(Permission.c.id==group_permission.c.permission_id,
Group.c.id==group_permission.c.group_id,
Group.c.id==user_group.c.group_id),
group_by=[Permission.c.permission_name],
distinct=True)
rs=q.execute()
for permission_name, num_users in q.execute():
print ‘Permission %s has %d users’ % (permission_name, num_users) #
虽然也长,但是减少了数据库查询次数,也就是让简单事情简单化,复杂事情可能
简单解决
```

看一个综合的例子:

```
class User(object):   #这些类设计数据库的模型

def __init__(self, group_name=None, users=None, permissions=None):
if users is None: users = []
if permissions is None: permissions = []
self.group_name = group_name
```

```
self._users = users
self._permissions = permissions

def add_user(self, user):
self._users.append(user)

def del_user(self, user):
self._users.remove(user)

def add_permission(self, permission):
self._permissions.append(permission)

def del_permission(self, permission):
self._permissions.remove(permission)

class Permission(object):

def __init__(self, permission_name=None, groups=None):
self.permission_name = permission_name
self._groups = groups

def join_group(self, group):
self._groups.append(group)

def leave_group(self, group):
self._groups.remove(group)
```

用 sqlalchemy 的效果是这样的:

```
user_table = Table(
 ‘tf_user’, metadata,
Column(‘id’, Integer, primary_key=True),
Column(‘user_name’, Unicode(16), unique=True, nullable=False),
Column(‘password’, Unicode(40), nullable=False))

group_table = Table(
 ‘tf_group’, metadata,
Column(‘id’, Integer, primary_key=True),
Column(‘group_name’, Unicode(16), unique=True, nullable=False))

permission_table = Table(
 ‘tf_permission’, metadata,
Column(‘id’, Integer, primary_key=True),
Column(‘permission_name’, Unicode(16), unique=True,
nullable=False))
```

```
user_group = Table(
 ‘user_group’,   metadata,
Column(‘user_id’, None, ForeignKey(‘tf_user.id’),
primary_key=True),
Column(‘group_id’, None, ForeignKey(‘tf_group.id’),
primary_key=True))

group_permission = Table(
 ‘group_permission’,   metadata,
Column(‘group_id’, None, ForeignKey(‘tf_group.id’),
primary_key=True),
Column(‘permission_id’, None, ForeignKey(‘tf_permission.id’),
primary_key=True))

mapper(User, user_table, properties=dict(
_groups=relation(Group, secondary=user_group, backref=’_users’)))
mapper(Group, group_table, properties=dict(
_permissions=relation(Permission, secondary=group_permission,
backref=_’groups’)))
mapper(Permission, permission_table)
```

这里没有修改对象,而 join_group,leave_group 这样的函数依然可用,sqlalchemy 会跟踪变化,并且自动刷新数据库

上面介绍了一个完整的例子,连接数据库嗨可以这样:

```
engine = create_engine(‘sqlite://’)
connection = engine.connect()   #使用 connect
result = connection.execute(“select user_name from tf_user”)
for row in result:
print  ‘user name: %s’  % row['user_name']
result.close()

engine = create_engine(‘sqlite://’,
strategy=’threadlocal’)   #,strategy=’threadlocal’表示重用其它本地线程减少对数据库的访问

from sqlalchemy.databases.mysql import MSEnum, MSBigInteger   #这个 sqlalchemy.databases 是某数据库软件的’方言’集合,只支持特定平台
user_table = Table(‘tf_user’, meta,
Column(‘id’, MSBigInteger),
Column(‘honorific’, MSEnum(‘Mr’, ‘Mrs’, ‘Ms’, ‘Miss’, ‘Dr’,
 ‘Prof’)))
```

以下是几个 MetaData 的应用:

```
unbound_meta = MetaData()   #这个 metadata 没有绑定
db1 = create_engine('sqlite://')
unbound_meta.bind = db1   #关联引擎

db2 = MetaData('sqlite:///test1.db')   #直接设置引擎
bound_meta1 = MetaData(db2)

# Create a bound MetaData with an implicitly created engine
bound_meta2 = MetaData('sqlite:///test2.db')   #隐式绑定引擎
meta = MetaData('sqlite://') #直接绑定引擎可以让源数据直接访问数据库

user_table = Table(
 'tf_user', meta,
Column('id', Integer, primary_key=True),
Column('user_name', Unicode(16), unique=True, nullable=False),
Column('password', Unicode(40), nullable=False))

group_table = Table(
 'tf_group', meta,
Column('id', Integer, primary_key=True),
Column('group_name', Unicode(16), unique=True, nullable=False))

meta.create_all() #创建所有的数据库(以上 2 个),函数无参数

result_set = group_table.select().execute() #选取 group_table 的所有表数据
```

以下看一个关联多引擎的例子:

```
meta = MetaData()   #这里不能直接关联了
engine1 = create_engine('sqlite:///test1.db')   #2 个引擎
engine2 = create_engine('sqlite:///test2.db')

# Use the engine parameter to load tables from the first engineuser_table
= Table(
 'tf_user', meta, autoload=True, autoload_with=engine1)   #从第一个引擎加载这些表
group_table = Table(
 'tf_group', meta, autoload=True, autoload_with=engine1)
permission_table = Table(
 'tf_permission', meta, autoload=True, autoload_with=engine1)
user_group_table = Table(
 'user_group', meta, autoload=True, autoload_with=engine1)
```

```
group_permission_table = Table(
 ‘group_permission’, meta, autoload=True, autoload_with=engine1)

meta.create_all(engine2) #在第二个引擎里面创建表

class ImageType(sqlalchemy.types.Binary):   #自定义我们的 table 的类
def convert_bind_param(self, value, engine):
sfp = StringIO()
value.save(sfp, ‘JPEG’)
return sfp.getvalue()
def convert_result_value(self, value, engine):
sfp = StringIO(value)
image = PIL.Image.open(sfp)
return image   #这里我们定义了一个图形处理的类型
```

当定义了 metadata 后,会自定生成一个 table.c object:

```
q = user_table.select(   #查询创建在 2007 年 6 月 1 号之前的用户,并且第一个字母是’r’
user_table.c.user_name.like(‘r%’)   #这里的 c 就是那个特殊的类,当使用 sql 表达式会用到
& user_table.c.created < datetime(2007,6,1))
```

或者替代这样:

```
q = user_table.select(and_(
user_table.c.user_name.like(‘r%’),
user_table.c.created < datetime(2007,6,1)))
```

也可以使用 rom 映射:

```
q = session.query(User)
q = q.filter(User.c.user_name.like(‘r%’)
& User.c.created > datetime(2007,6,1))
```

还是一个 ORM 的例子:

```
user_table = Table(
 ‘tf_user’, metadata,
Column(‘id’, Integer, primary_key=True),
Column(‘user_name’, Unicode(16), unique=True, nullable=False),
Column(‘email_address’, Unicode(255), unique=True, nullable=False),
Column(‘password’, Unicode(40), nullable=False),
Column(‘first_name’, Unicode(255), default=”),
Column(‘last_name’, Unicode(255), default=”),
Column(‘created’, DateTime, default=datetime.now))   #这是一个定义的表类型

group_table = Table(
 ‘tf_group’, metadata,
```

```
Column( ‘id’ , Integer, primary_key=True),
Column( ‘group_name’ , Unicode(16), unique=True, nullable=False))

user_group = Table(
 ‘user_group’ ,   metadata,
Column( ‘user_id’ , None, ForeignKey( ‘tf_user.id’ ),
primary_key=True),
Column( ‘group_id’ , None, ForeignKey( ‘tf_group.id’ ),
… primary_key=True))

import sha
class User(object):   #映射类

def _get_password(self):
return self._password
def _set_password(self, value):
self._password = sha.new(value).hexdigest() #只存储用户的哈希密码
password=property(_get_password, _set_password)

def password_matches(self, password):
return sha.new(password).hexdigest() == self._password

mapper(User, user_table, properties=dict( #映射将创建 id, user_name,
email_address, password, first_name, last_name, created 等字段
_password=user_table.c.password)) #使用哈希后的密码替换真实密码,数据
库只保存哈希后的,这里在 orm 上修改

mapper(User, user_table, properties=dict(
_password=user_table.c.password,
groups=relation(Group, secondary=user_group, backref=’ users’ ))) #这
里表示可以访问所有的组，用户只需访问一个成员团体属性,user_group 映射类
添加 group 和 Group 关联,

# User 类添加 users 访问 group 属性,看效果:
group1.users.append(user1)    #给 group1 添加用户 user1,自动更新
user2.groups.append(group2) #把 user2 添加到 group2 组,自动更新
```

对于 SQLAlchemy 的一些总结:

1 metadata.create_all()

创建多个 table 可以这样使用,但是他还有个功能,它添加了”IF NOT EXISTS”,
就是在数据库存在的时候,他还是安全的

2 交互模式下的一个全过程:

Source code

```
dongwm@localhost ~ $ python
Python 2.7.3 (default, Jul 11 2012, 10:10:17)
[GCC 4.5.3] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>> from sqlalchemy import create_engine
>>> from sqlalchemy import Table, MetaData, Column,
ForeignKey, Integer, String, Unicode, DateTime
>>> from datetime import datetime
>>> metadata = MetaData('sqlite:///tutorial.db')
>>> user_table = Table(
...     'tf_user', metadata,
...     Column('id', Integer, primary_key=True),
...     Column('user_name', Unicode(16),
...           unique=True, nullable=False),
...     Column('password', Unicode(40), nullable=False),
...     Column('display_name', Unicode(255), default=''),
...     Column('created', DateTime, default=datetime.now))
__main__:7: SAWarning: Unicode column received non-unicode
default value.

>>> stmt = user_table.insert()  #插入数据

>>> stmt.execute(user_name='dongwm1',
password='secret',display_name='testdongwm1')
/usr/lib/python2.7/site-packages/SQLAlchemy-0.7.8-py2.7-l
inux-i686.egg/sqlalchemy/engine/default.py:463: SAWarning:
Unicode type received non-unicode bind param value.
  param.append(processors[key](compiled_params[key]))
<sqlalchemy.engine.base.ResultProxy object at 0x8377fcc>
>>> stmt.execute(user_name='dongwm2',

password='secret',display_name='testdongwm2') #这个实例可以

多次插入,和 sql 区别很大

<sqlalchemy.engine.base.ResultProxy object at 0x837e4ec>

>>> stmt = user_table.select() #select 查询

>>> result = stmt.execute()
>>> for row in result:
...     print row
...
(1, u'dongwm1', u'secret', u'testdongwm1',
datetime.datetime(2012, 7, 17, 11, 57, 48, 515953))
```

```
(2, u'dongwm2', u'secret', u'testdongwm2',
datetime.datetime(2012, 7, 17, 11, 58, 5, 226977))
>>> result = stmt.execute()
```

>>> row =result.fetchone() #只获取符合要求的第一项

```
>>> print  row['user_name']
dongwm1
>>> print row.password
secret
>>> print row.items()
[(u'id', 1), (u'user_name', u'dongwm1'), (u'password',
u'secret'), (u'display_name', u'testdongwm1'), (u'created',
datetime.datetime(2012, 7, 17, 11, 57, 48, 515953))]
>>> stmt =
```

user_table.select(user_table.c.user_name=='dongwm1') #过滤

留下 user_name=='dongwm1 的项

>>> print stmt.execute().fetchall() #获取所有符合项

```
[(1, u'dongwm1', u'secret', u'testdongwm1',
datetime.datetime(2012, 7, 17, 11, 57, 48, 515953))]
>>> stmt =
```

user_table.update(user_table.c.user_name=='dongwm1') #更新

数据

>>> stmt.execute(password='secret123') #修改密码

```
<sqlalchemy.engine.base.ResultProxy object at 0x8377f6c>
>>> stmt = user_table.delete(user_table.c.user_name !=
```

'dongwm1') #删除 user_name 不是 dongwm1 的条目

```
>>> stmt.execute()
<sqlalchemy.engine.base.ResultProxy object at 0x837f3ac>
```

>>> user_table.select().execute().fetchall() #查询发现就剩一

条了

```
[(1, u'dongwm1', u'secret123', u'testdongwm1',
datetime.datetime(2012, 7, 17, 11, 57, 48, 515953))]
```

3 sission 上面已经说过了,补充一些:

session.delete(u) #把映射类从会话中删除

4 关于引擎

引擎就是根据不同的数据库方言连接数据库的方法

以下是一些例子(方法 driver://username:password@host:port/database):

```
engine = create_engine(‘sqlite://’)   #连接基于内存的 sqlite
engine = create_engine(‘sqlite:///data.sqlite’)   #连接基于硬盘文件的 sqlite
engine =
create_engine(‘postgres://dongwm:foo@localhost:5432/pg_db’)   #连接 postgresql
engine = create_engine(‘mysql://localhost/mysql_db’)   #连接 mysql
engine = create_engine(‘oracle://dongwm:foo@oracle_tns’) #连接基于 TNS 协议的 Oracle
engine
=create_engine(‘oracle://dongwm:foo@localhost:1521/oracle_sid’) #连接没有 TNS 名字的 Oracle
```

也可以带一些参数:

```
url=’ postgres://dongwm:foo@localhost/pg_db?arg1=foo&arg2=bar’
engine = create_engine(url)
```

或者:

```
engine = create_engine(‘postgres://dongwm:foo@localhost/pg_db’,
connect_args=dict(arg1=’ foo’, arg2=’ bar’))
```

还可以通过函数完全控制连接:

```
import psycopg
def connect_pg():
return psycopg.connect(user=’ rick’, host=’ localhost’)
engine = create_engine(‘postgres://’, creator=connect_pg)
```

```
import logging
handler = logging.FileHandler(‘sqlalchemy.engine.log’)   #可以给它添加一个日志文件处理类
handler.level = logging.DEBUG
logging.getLogger(‘sqlalchemy.engine’).addHandler(handler)
```

上面说的操作表,也可以直接操作数据库:

```
conn = engine.connect()
result = conn.execute(‘select user_name, email_address from tf_user’)
#结果是一个 sqlalchemy.engine.ResultProxy 的实例
for row in result:
```

```
print ‘User name: %s Email address: %s’ % (
row['user_name'], row['email_address'])
conn.close()

from sqlalchemy import pool #本来它已经自动通过数据库连接管理数据池,但是也可以手动管理
import psycopg2
psycopg = pool.manage(psycopg2) #结果是一个 sqlalchemy.pool.DBProxy 实例
connection = psycopg.connect(database=’ mydb’ ,
username=’ rick’ , password=’ foo’ )
```

5 关于元数据 metadata

它收集了描述 table 对象等的元数据类,当使用 ORM 等时必须使用 metadata

如果他被绑定了,那么使用 table.create()就会生成表,没有绑定需要:table.create(bind=some_engine_or_connection),其中 table.create

包含一些函数:

autoload:默认是 false,当数据库已经存在这个 table 会自动加载覆盖

autoload_with:默认是 false,是否自动加载引擎的字段结构

reflect:默认是 false,是否体现源表结构

```
brand_table = Table( ‘brand’ , metadata,
Column( ‘name’ , Unicode(255)), # 覆盖类型
autoload=True)
```

6 关于表结构:

设置表主键可以这样:

```
Column( ‘brand_id’ , Integer,
ForeignKey( ‘brand.id’ ),primary_key=True), #通过 primary_key=True
Column( ‘sku’ , Unicode(80), primary_key=True))
也可以这样:
product_table = Table(
 ‘product’ , metadata,
Column( ‘brand_id’ , Integer, ForeignKey( ‘brand.id’ )),
Column( ‘sku’ , Unicode(80)),
PrimaryKeyConstraint( ‘brand_id’ , ‘sku’ , name=’ prikey’ ))  #通过 PrimaryKeyConstraint
```

```
style_table = Table(
 ‘style’, metadata,
Column(‘brand_id’, Integer, primary_key=True),
Column(‘sku’, Unicode(80), primary_key=True),
Column(‘code’, Unicode(80), primary_key=True),
ForeignKeyConstraint(   #使用复合键,关联外部表的字段
['brand_id', 'sku'],
['product.brand_id', 'product.sku']))

product_table = Table(
 ‘product’, metadata,
Column(‘id’, Integer, primary_key=True),
Column(‘brand_id’, Integer, ForeignKey(‘brand.id’)), #他的 brand_id 关联 brand 的让 id
Column(‘sku’, Unicode(80)),
UniqueConstraint(‘brand_id’, ‘sku’)) #约束唯一标识数据库表中的每条记录

payment_table = Table(
 ‘payment’, metadata,
Column(‘amount’, Numeric(10,2), CheckConstraint(‘amount > 0′ ))) #验证 amount 大于 0
user_table = Table(
 ‘tf_user’, MetaData(),
Column(‘id’, Integer, primary_key=True),
Column(‘user_name’, Unicode(16), unique=True, nullable=False),
Column(‘password’, Unicode(40), nullable=False),
Column(‘first_name’, Unicode(255), default=”),
Column(‘last_name’, Unicode(255), default=”),
Column(‘created_apptime’, DateTime, default=datetime.now), #default 表示当不舍定具体值时设定一个默认值
Column(‘created_dbtime’, DateTime, PassiveDefault(‘sysdate’)), #PassiveDefault 是数据库级别的默认值,
Column(‘modified’, DateTime, onupdate=datetime.now)) #单设置 onupdate 这个属性，这是不应用到数据库的设计中的.只是存在于映射类中.

#它是活跃更新的,因为每次执行的时间都不同

user_table = Table(
 ‘tf_user’, MetaData(),
Column(‘id’, Integer, primary_key=True),
Column(‘user_name’, Unicode(16), unique=True, nullable=False,
index=True), #一旦数据库增长到一定规模时，可能需要考虑增加表的索引,以加快某些操作
Column(‘password’, Unicode(40), nullable=False),
```

```
Column( ‘first_name’, Unicode(255), default=”),
Column( ‘last_name’, Unicode(255), default=”, index=True))
```

其中指定索引也可以这样:

```
i = Index( ‘idx_name’,
user_table.c.first_name,user_table.c.last_name,unique=True)
i.create(bind=e)

brand_table = Table(
 ‘brand’, metadata,
Column( ‘id’, Integer, Sequence( ‘brand_id_seq’),
primary_key=True),   #需要通过序列化方式来创建新主键标识符的数据库,

#SQLAlchemy 并不会自动为其生成。可以指定 Sequence 生成
Column( ‘name’, Unicode(255), unique=True, nullable=False))
```

7 元数据操作

```
meta1 = MetaData( ‘postgres://postgres:password@localhost/test’,
… reflect=True)
meta2 = MetaData( ‘sqlite://’ )
for table in meta1.table_iterator():
table.tometadata(meta2) #通过这个方法让 meta1 的元数据被 meta2 使用
meta2.create_all()
```

2 假如想放弃绑定使用 drop_all( )或者 drop(e)

1 自定义表结构类型:

```
from sqlalchemy import types

class MyCustomEnum(types.TypeDecorator):   #自定义的类型继承至
types.TypeDecorator

impl=types.Integer   #实现指定的类型 int

def __init__(self, enum_values, *l, **kw):
types.TypeDecorator.__init__(self, *l, **kw)
self._enum_values = enum_values

def convert_bind_param(self, value, engine):   #必须含有这个方法,转换
python 语言为 SQL
result = self.impl.convert_bind_param(value, engine)
if result not in self._enum_values:
raise TypeError, (
```

```
“Value %s must be one of %s” % (result, self._enum_values))
return result

def convert_result_value(self, value, engine):  #必须含有这个方法,通过 db 的 api 把 SQL 转换成 python 语言
‘Do nothing here’
return self.impl.convert_result_value(value, engine)
```

看一个例子:

Source code

```
from sqlalchemy import types
from sqlalchemy.databases import sqlite
class MyCustomEnum(types.TypeDecorator):
    impl = types.Integer
    def __init__(self, enum_values, *l, **kw):
        types.TypeDecorator.__init__(self, *l, **kw)
        self._enum_values = enum_values

    def bind_processor(self, dialect): #如果提供这个方法会替代

convert_bind_param( )和 convert_result_value( )

        impl_processor = self.impl.bind_processor(dialect)
        if impl_processor:
            def processor(value):
                result = impl_processor(value)
                assert value in self._enum_values, \
                    "Value %s must be one of %s" % (result,
                    self._enum_values)
                return result
        else:
            def processor(value):
                assert value in self._enum_values, \
                    "Value %s must be one of %s" % (value,
                     self._enum_values)
                return value
        return processor
mce=MyCustomEnum([1,2,3])
processor = mce.bind_processor(sqlite.dialect())

print processor(1) #返回 1

print processor(5) #返回错误,因为不是 1,2,3 中的数据
```

你甚至可以直接定义自定的 TypeDecorator

```
class NewType(types.TypeEngine): #TypeDecorator 继承自 types.TypeEngine

def __init__(self, *args):
self._args = args

def get_col_spec(self):   #create_table( )会用到这个方法
return 'NEWTYPE(%s)' % ','.join(self._args)

def convert_bind_param(self, value, engine):   #这个必须设置
return value

def convert_result_value(self, value, engine):   #这个也必须设置
return value
```

2 SQL 语句在交互模式下的例子:

```
dongwm@localhost ~ $ python
Python 2.7.3 (default, Jul 11 2012, 10:10:17)
[GCC 4.5.3] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>> from sqlalchemy import Table, MetaData, Column, ForeignKey, Integer,
String, Unicode, DateTime
>>> metadata=MetaData()
>>> simple_table = Table(   #一个简单的表结构
...          'simple', metadata,
...         Column('id', Integer, primary_key=True),
...         Column('col1', Unicode(20)))
>>>
>>> stmt = simple_table.insert()   #插入数据操作的实例
>>> print stmt #打印这个实例
INSERT INTO simple (id, col1) VALUES (:id, :col1) #里面包含需要替换的
变量
>>> compiled_stmt = stmt.compile()   #编译语句
>>> print compiled_stmt.params #转成了字典得方式
{'id': None, 'col1': None}
>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite://')
>>> simple_table.create(bind=engine)   #创建 table
>>> engine.execute(stmt, col1="Foo") #给语句添加值
/usr/lib/python2.7/site-packages/SQLAlchemy-0.7.8-py2.7-linux-i686.eg
g/sqlalchemy/engine/default.py:463: SAWarning: Unicode type received
```

```
non-unicode bind param value.
param.append(processors[key](compiled_params[key]))
<sqlalchemy.engine.base.ResultProxy object at 0x8376c8c>
>>> metadata.bind = engine   #和上面效果一样,给语句添加值
>>> stmt.execute(col1=” Bar” )
<sqlalchemy.engine.base.ResultProxy object at 0x8376f4c>
>>> stmt = simple_table.insert(values=dict(col1=” Initial value”)) #
这次插入已经设置了值
>>> print stmt
INSERT INTO simple (col1) VALUES (?)
>>> compiled_stmt = stmt.compile()
>>> print compiled_stmt.params
{ ‘col1′ : ‘Initial value’ }
>>> stmt = simple_table.insert()
>>> stmt.execute(col1=” First value” )
<sqlalchemy.engine.base.ResultProxy object at 0x838832c>
>>>
>>> stmt.execute(col1=” Second value” )
<sqlalchemy.engine.base.ResultProxy object at 0x838844c>
>>> stmt.execute(col1=” Third value” ) #这样一行一行插入真是费劲
<sqlalchemy.engine.base.ResultProxy object at 0x838856c>
>>> stmt.execute([dict(col1="Fourth Value"), #可以一次插入多行
...          dict(col1="Fifth Value"),
...          dict(col1="Sixth Value")])
<sqlalchemy.engine.base.ResultProxy object at 0x83886ac>
>>> from sqlalchemy import text
>>> stmt = simple_table.update(
…          whereclause=text(“col1=’ First value’ ”),
…          values=dict(col1=’ 1st Value’))   #执行 col1 是 First value
的条目设置值为 1st Value
>>> stmt.execute()
<sqlalchemy.engine.base.ResultProxy object at 0x838878c>
>>> stmt = simple_table.update(text(“col1=’ Second value’ ”)) #寻找
col1 是 Second value 的条目
>>> stmt.execute(col1=’ 2nd Value’ ) #执行更新时,设置其值,想过和上面的
一样
<sqlalchemy.engine.base.ResultProxy object at 0x8376f4c>
>>> stmt = simple_table.update(text(“col1=’ Third value’ ”))
>>> print stmt
UPDATE simple SET id=?, col1=? WHERE col1=’ Third value’
>>> engine.echo = True #设置打印调试日志
>>> stmt.execute(col1=’ 3rd value’ )
2012-07-17 15:16:59,231 INFO sqlalchemy.engine.base.Engine UPDATE
simple SET col1=? WHERE col1=’ Third value’
```

```
2012-07-17 15:16:59,245 INFO sqlalchemy.engine.base.Engine (’ 3rd
value’,)
2012-07-17 15:16:59,245 INFO sqlalchemy.engine.base.Engine COMMIT
<sqlalchemy.engine.base.ResultProxy object at 0x83767ec>

>>> stmt = simple_table.delete(   #删除
…         text(“col1=’ Second value’ ”))
>>> stmt.execute()
2012-07-17 15:21:03,806 INFO sqlalchemy.engine.base.Engine DELETE FROM
simple WHERE col1=’ Second value’
2012-07-17 15:21:03,806 INFO sqlalchemy.engine.base.Engine ()
2012-07-17 15:21:03,806 INFO sqlalchemy.engine.base.Engine COMMIT
<sqlalchemy.engine.base.ResultProxy object at 0x8376a0c>
>>> from sqlalchemy import select
>>> stmt = select([simple_table.c.col1])   #查询 col1 这个字段
>>> for row in stmt.execute():
…         print row
(u’ Foo’,)
(u’ Bar’,)
(u’ 1st Value’,)
(u’ 2nd Value’,)
(u’ 3rd value’,)
(u’ Fourth Value’,)
(u’ Fifth Value’,)
(u’ Sixth Value’,)

>>> stmt = simple_table.select() #和上面的区别是这是条目全部显示
>>> for row in stmt.execute():   #这 2 句也可以这样表示 stmt =
select( simple_table])
…         print row
…
(1, u’ Foo’ )
(2, u’ Bar’ )
(3, u’ 1st Value’ )
(4, u’ 2nd Value’ )
(5, u’ 3rd value’ )
(6, u’ Fourth Value’ )
(7, u’ Fifth Value’ )
(8, u’ Sixth Value’ )
>>> x = simple_table.c.col1==” Foo”
>>> print type(x)
<class ‘sqlalchemy.sql.expression._BinaryExpression’ >
>>> print x
simple.col1 = :col1_1
```

```
>>> expr = simple_table.c.col1 + “-col1”    #它还支持运算符
>>> print expr
simple.col1 || :col1_1
>>> from sqlalchemy.databases import mysql
>>> print expr.compile(dialect=mysql.MySQLDialect())
concat(simple.col1, %s) #在不同的数据库软件,效果不同

>>> from sqlalchemy import func
>>> print func.now()
now()
>>> print func.current_timestamp
<sqlalchemy.sql.expression._FunctionGenerator object at 0x83888cc>
>>> print func._(text( ‘a=b’ ))
(a=b)
```

注:sqlalchemy 支持 in,op,startwith,endwith,between,like 等运算

```
>>> from sqlalchemy import bindparam   #自定义绑定的词
>>> stmt = select([simple_table.c.col1],
…          whereclause=simple_table.c.col1==bindparam( ‘test’ ))   #
用 test 替换原来的 col1
>>> print stmt
SELECT simple.col1
FROM simple
WHERE simple.col1 = ? #这里依然是 col1
>>> print stmt.execute(test=’ Foo’ ).fetchall()
[(u'Foo',)]

>>> stmt = simple_table.select(order_by=[simple_table.c.col1])   #更具
col1,升序排序
>>> print stmt
SELECT simple.id, simple.col1
FROM simple ORDER BY simple.col1
>>> print stmt.execute().fetchall()
[(3, u'1st Value'), (4, u'2nd Value'), (5, u'3rd value'), (2, u'Bar'),
(7, u'Fifth Value'), (1, u'Foo'), (6, u'Fourth Value'), (8, u'Sixth
Value')]
>>> from sqlalchemy import desc
>>> stmt = simple_table.select(order_by=[desc(simple_table.c.col1)]) #
根据 col1,降序排序
>>> print stmt
SELECT simple.id, simple.col1
FROM simple ORDER BY simple.col1 DESC
>>> print stmt.execute().fetchall()
[(8, u'Sixth Value'), (6, u'Fourth Value'), (1, u'Foo'), (7, u'Fifth
```

Value'), (2, u'Bar'), (5, u'3rd value'), (4, u'2nd Value'), (3, u'1st Value')]
注:distinct=True 去重复,效果类似于 SELECT DISTINCT

>>> stmt = simple_table.select(offset=1, limit=1) #offset 设置偏移,这里就是略过第一个,返回第二个.limit 设置返回多少个条目
>>> print stmt
SELECT simple.id, simple.col1
FROM simple
LIMIT ? OFFSET ?
>>> print stmt.execute().fetchall()
[(2, u'Bar')]
看下面的例子:

“Persons” 表:

Id_P	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Changan Street	Beijing

“Orders” 表:

Id_O	OrderNo	Id_P
1	77895	3
2	44678	3
3	22456	1
4	24562	1
5	34764	65

现在，我们希望列出所有的人，以及他们的定购号码:

```
SELECT Persons.LastName, Persons.FirstName, Orders.OrderNo
FROM Persons
LEFT JOIN Orders #将 orders 表 join 进来
ON Persons.Id_P=Orders.Id_P #关系联系
ORDER BY Persons.LastName #排序
```

书中的例子是这样的:

```
SELECT store.name
FROM store
JOIN product_price ON store.id=product_price.store_id
JOIN product ON product_price.sku=product.sku
WHERE product.msrp != product_price.price;
```

转换成 sqlalchemy 语句:

```
>>>from_obj =
store_table.join(product_price_table).join(product_table)
>>> query = store_table.select()
>>> query = query.select_from(from_obj)
>>> query = query.where(product_table.c.msrp !=
product_price_table.c.price)
>>> print query
SELECT store.id, store.name
FROM store JOIN product_price ON store.id = product_price.store_id JOIN
product ON product.sku = product_price.sku
WHERE product.msrp != product_price.price

>>> print query.column( 'product.sku' )
SELECT store.id, store.name, product.sku
FROM store JOIN product_price ON store.id =
product_price.store_id   JOIN product ON product.sku =
product_price.sku
WHERE product.msrp != product_price.price
>>> query2 = select([store_table,
product_table.c.sku],from_obj=[from_obj],whereclause=(product_table.c.
msrp !=product_price_table.c.price))
>>> print query2
SELECT store.id, store.name, product.sku
FROM store JOIN product_price ON store.id = product_price.store_id JOIN
product ON product.sku = product_price.sku
WHERE product.msrp != product_price.price
>>> query = product_table.select(and_(product_table.c.msrp >
10.00 ,product_table.c.msrp < 20.00)) #范围查询
>>> print query
SELECT product.sku, product.msrp
FROM product
WHERE product.msrp > ? AND product.msrp < ?
>>> for r in query.execute():
…print r
(u' 123' , Decimal(" 12.34" ))

>>> from sqlalchemy import intersect

>>> query0 = product_table.select(product_table.c.msrp > 10.00)
>>> query1 = product_table.select(product_table.c.msrp < 20.00)
>>> query = intersect(query0, query1) #使用 intersect 添加多 query
>>> print query
SELECT product.sku, product.msrp
```

```
employee_table = Table(
 ‘employee’, metadata,
Column(‘id’, Integer, primary_key=True),
Column(‘manager’, None, ForeignKey(‘employee.id’)),
Column(‘name’, String(255)))
```

给设定 alias:

比如想实现以下 SQL

```
SELECT employee.name
FROM employee, employee AS manager
WHERE employee.manager_id = manager.id
AND manager.name =  ‘Fred’

>>> manager = employee_table.alias(‘mgr’)
>>> stmt = select([employee_table.c.name],
…
and_(employee_table.c.manager_id==manager.c.id,
…
manager.c.name==’Fred’))
>>> print stmt
SELECT employee.name
FROM employee, employee AS mgr
WHERE employee.manager_id = mgr.id AND mgr.name = ?

>>> manager = employee_table.alias()   #自动 alias
>>> stmt = select([employee_table.c.name],
…and_(employee_table.c.manager_id==manager.c.id,
…manager.c.name==’Fred’))
>>> print stmt
SELECT employee.name
FROM employee, employee AS employee_1
WHERE employee.manager_id = employee_1.id AND employee_1.name = ?
```

```
from sqlalchemy import types
class MyCustomEnum(types.TypeDecorator):
impl=types.Integer
def __init__(self, enum_values, *l, **kw):
types.TypeDecorator.__init__(self, *l, **kw)
self._enum_values = enum_values
```

```
def convert_bind_param(self, value, engine):
result = self.impl.convert_bind_param(value, engine)
if result not in self._enum_values:
raise TypeError, (
“Value %s must be one of %s” % (result, self._enum_values))
Application-Specific Custom Types | 63return result
def convert_result_value(self, value, engine):
‘Do nothing here’
return self.impl.convert_result_value(value, engine)
```

1　ORM 模型的简单性简化了数据库查询过程。使用 ORM 查询工具，用户可以访问期望数据，而不必理解数据库的底层结构

以下是 SQL 语句:

```
region_table = Table(
‘region’, metadata,
Column(‘id’, Integer, primary_key=True),
Column(‘name’, Unicode(255)))
```

相应的类:

```
class Region(object):

def __init__(self, name):
self.name = name

def __repr__(self):
return ‘<Region %s>’ % self.name
```

看一下在交互模式下:

```
>>> dir(Region)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__',
'__getattribute__', '__hash__', '__init__', '__module__', '__new__',
'__reduce__','__reduce_ex__','__repr__','__setattr__','__sizeof__',
'__str__', '__subclasshook__', '__weakref__']
>>> mapper(Region,region_table)   #ORM 映射
<Mapper at 0x84bdb2c; Region>
>>> dir(Region)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__',
'__getattribute__', '__hash__', '__init__', '__module__', '__new__',
'__reduce__','__reduce_ex__','__repr__','__setattr__','__sizeof__',
'__str__','__subclasshook__','__weakref__','_sa_class_manager','id',
'name'] #增加了很多属性
>>> Region.id
<sqlalchemy.orm.attributes.InstrumentedAttribute object at 0x84c238c>
```

```
>>> Region.name
<sqlalchemy.orm.attributes.InstrumentedAttribute object at 0x84c254c>

>>> r0 = Region(name=”Northeast”)
>>> r1 = Region(name=”Southwest”)
>>> r0
<Region Northeast>   #类能显示这样的数据是因为类定义了__repr__方法
>>> r1
<Region Southwest>
>>> from sqlalchemy.orm import clear_mappers
>>> clear_mappers() #取消映射
>>> Region.name #不再有这个属性
Traceback (most recent call last):
File “<stdin>”, line 1, in <module>
AttributeError: type object ‘Region’ has no attribute ‘name’
>>> dir(Region)   #回到了原来的只有类属性
['__class__', '__delattr__', '__dict__', '__doc__', '__format__',
'__getattribute__', '__hash__', '__init__', '__module__', '__new__',
'__reduce__','__reduce_ex__','__repr__','__setattr__','__sizeof__',
'__str__', '__subclasshook__', '__weakref__']

>>> r0 = Region(name=”Northeast”)   #从这里开始理解 ORM 做了什么
>>> r1 = Region(name=”Southwest”) #实现了 2 个类的实例

>>> metadata.create_all(engine) #创建 table

>>> Session = sessionmaker()   #通过 sessionmaker 产生一个会话
>>> Session.configure(bind=engine) #绑定到数据库连接
>>> session = Session()   #产生会话实例,让对象可以被载入或保存到数据库,
而只需要访问类却不用直接访问数据库
>>> session.bind.echo = True #显示打印信息

>>> session.add(r1) #把 r0,r12 个实例加到会话中
>>> session.add(r0)
>>> print r0.id   #因为还没有保存,数据为空
None
>>> session.flush() #提交数据到数据库
2012-07-18 10:24:07,116 INFO sqlalchemy.engine.base.Engine BEGIN
(implicit)
2012-07-18 10:24:07,116 INFO sqlalchemy.engine.base.Engine INSERT INTO
region (name) VALUES (?)
2012-07-18 10:24:07,116 INFO sqlalchemy.engine.base.Engine
(‘Southwest’,)
2012-07-18 10:24:07,117 INFO sqlalchemy.engine.base.Engine INSERT INTO
region (name) VALUES (?)
```

```
2012-07-18 10:24:07,117 INFO sqlalchemy.engine.base.Engine
(‘Northeast’,)
>>> print r0.id #id 因为子增长,出现了
2
>>> r0.name = ‘Northwest’
>>> session.flush() #修改提交
2012-07-18 10:24:50,644 INFO sqlalchemy.engine.base.Engine UPDATE
region SET name=? WHERE region.id = ?
2012-07-18 10:24:50,644 INFO sqlalchemy.engine.base.Engine
(‘Northwest’, 2)
>>> print r0.name #数据库中的数据被 update 成了新值
Northwest
>>> dir(Region)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__',
'__getattribute__', '__hash__', '__init__', '__module__', '__new__',
'__reduce__','__reduce_ex__','__repr__','__setattr__','__sizeof__',
'__str__', '__subclasshook__', '__weakref__']
>>> mapper(Region, region_table, include_properties=['id']) #使用
include_properties 只映射某些字段,同样还有 exclude_properties
<Mapper at 0x84c26cc; Region>
>>> dir(Region)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__',
'__getattribute__', '__hash__', '__init__', '__module__', '__new__',
'__reduce__','__reduce_ex__','__repr__','__setattr__','__sizeof__',
'__str__', '__subclasshook__', '__weakref__', '_sa_class_manager',
'id']  #只多了一个”id”

>>> clear_mappers()
>>> dir(Region)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__',
'__getattribute__', '__hash__', '__init__', '__module__', '__new__',
'__reduce__','__reduce_ex__','__repr__','__setattr__','__sizeof__',
'__str__', '__subclasshook__', '__weakref__']
>>> mapper(Region, region_table, column_prefix=’_’)  #映射后自定义修
改新属性的前缀
<Mapper at 0x84f73ac; Region>
>>> dir(Region)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__',
'__getattribute__', '__hash__', '__init__', '__module__', '__new__',
'__reduce__','__reduce_ex__','__repr__','__setattr__','__sizeof__',
'__str__', '__subclasshook__', '__weakref__', '_id', '_name',
'_sa_class_manager'] #id 和 name 等前面都有了”_”
```

```
>>> clear_mappers()
>>> dir(Region)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__',
'__getattribute__', '__hash__', '__init__', '__module__', '__new__',
'__reduce__','__reduce_ex__','__repr__','__setattr__','__sizeof__',
'__str__', '__subclasshook__', '__weakref__']
>>> mapper(Region, region_table, properties=dict(
…         region_name=region_table.c.name,    #想把 name 的属性定义为
region_name,因为 c.name 就是用 Table 创建的表结构的特定实例的 name 属性
…         region_id=region_table.c.id))
<Mapper at 0x8509d2c; Region>
>>> dir(Region)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__',
'__getattribute__', '__hash__', '__init__', '__module__', '__new__',
'__reduce__','__reduce_ex__','__repr__','__setattr__','__sizeof__',
'__str__', '__subclasshook__', '__weakref__', '_sa_class_manager',
'region_id', 'region_name']   #id 改名为 region_id

>>> class Region(object):   #重新定义类
…         def __init__(self, name):
…                 self.name = name
…         def __repr__(self):
…                 return  '<Region %s>'  % self.name
…         def _get_name(self):#这个_get 和_set 是为了让内置的 property
调用
…                 return self._name
…         def _set_name(self, value):
…                 assert value.endswith( 'Region' ), \
…                         'Region names must end in "Region" '
…                 self._name = value
…         name=property(_get_name, _set_name) #通过 property 的定义，
当获取成员 x 的值时，就会调用_get_name 函数(第一个参数)，当给成员 x 赋值
时，就会调用_set_name 函数(第二个参数)，当删除 x 时，就会调用 delx 函数(这
里没有设置)
…
>>> from sqlalchemy.orm import synonym
>>> mapper(Region, region_table, column_prefix='_', properties=dict(
…         name=synonym( '_name' ))) #首先检验_name 的属性是否满足
<Mapper at 0x84f7acc; Region>
>>> s0 = Region( 'Southeast' )   #没有正确结尾
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<string>", line 4, in __init__
File
```

“/usr/lib/python2.7/site-packages/SQLAlchemy-0.7.8-py2.7-linux-i686.egg/sqlalchemy/orm/state.py”, line 98, in initialize_instance
return manager.original_init(*mixed[1:], **kwargs)
File “<stdin>”, line 3, in __init__
File “<string>”, line 1, in __set__
File “<stdin>”, line 10, in _set_name
AssertionError: Region names must end in “Region”
>>> s0 = Region(‘Southeast Region’) #正常

```
>>> segment_table = Table(
…          ‘segment’, metadata,
…         Column(‘id’, Integer, primary_key=True),
…         Column(‘lat0′, Float),
…         Column(‘long0′, Float),
…         Column(‘lat1′, Float),
…         Column(‘long1′, Float))

>>> metadata.create_all(engine) #创建表
>>> class RouteSegment(object): #一个含有 begin 和 end 的类
…         def __init__(self, begin, end):
…                 self.begin = begin
…                 self.end = end
…         def __repr__(self):
…                 return ‘<Route %s to %s>’ % (self.begin, self.end)
…
>>> class MapPoint(object):
…         def __init__(self, lat, long):
…                 self.coords = lat, long
…         def __composite_values__(self): #返回比较后的列表或者元祖
…                 return self.coords
…         def __eq__(self, other):
…                 return self.coords == other.coords
…         def __ne__(self, other):
…                 return self.coords != other.coords
…         def __repr__(self):
…                 return ‘(%s lat, %s long)’ % self.coords
…
…

>>> from sqlalchemy.orm import composite
>>> mapper(RouteSegment, segment_table, properties=dict(
…         begin=composite(MapPoint,   #创建多个属性
…                 segment_table.c.lat0,
…                 segment_table.c.long0),
…         end=composite(MapPoint,
```

```
…                 segment_table.c.lat1, segment_table.c.long1)))
<Mapper at 0x86203cc; RouteSegment>
>>> work=MapPoint(33.775562,-84.29478)
>>> library=MapPoint(34.004313,-84.452062)
>>> park=MapPoint(33.776868,-84.389785)
>>> routes = [
...          RouteSegment(work, library),
...          RouteSegment(work, park),
...          RouteSegment(library, work),
...          RouteSegment(library, park),
...          RouteSegment(park, library),
...          RouteSegment(park, work)]

>>> for rs in routes:
…          session.add(rs)
…
>>> session.flush()
>>> q = session.query(RouteSegment)
>>> print RouteSegment.begin==work
segment.lat0 = :lat0_1 AND segment.long0 = :long0_1
>>> q = q.filter(RouteSegment.begin==work)
>>> for rs in q:
…          print rs
…
2012-07-18 11:12:29,360 INFO sqlalchemy.engine.base.Engine SELECT
segment.id AS segment_id, segment.lat0 AS segment_lat0, segment.long0 AS
segment_long0, segment.lat1 AS segment_lat1, segment.long1 AS
segment_long1
FROM segment
WHERE segment.lat0 = ? AND segment.long0 = ?
2012-07-18 11:12:29,360 INFO sqlalchemy.engine.base.Engine (33.775562,
-84.29478)
<Route (33.775562 lat, -84.29478 long) to (34.004313 lat, -84.452062
long)>
<Route (33.775562 lat, -84.29478 long) to (33.776868 lat, -84.389785
long)>

>>> from sqlalchemy.orm import PropComparator
>>> class MapPointComparator(PropComparator): #自定义运算符继承
PropComparator 类
…          def __lt__(self, other):   #自定义小于运算结果
…                  return and_(*[a<b for a, b in
...                           zip(self.prop.columns,
...                                   other.__composite_values__())])
```

```
…
>>> mapper(RouteSegment, segment_table, properties=dict(
…         begin=composite(MapPoint,
…                                         segment_table.c.lat0,
segment_table.c.long0,
…                                         comparator=MapPointCompara
tor),   #定义使用自定义的运算类
…         end=composite(MapPoint,
…                                       segment_table.c.lat1,
segment_table.c.long1,
…                                       comparator=MapPointComparator))
)
<Mapper at 0x85b2bac; RouteSegment>
>>> product_table = Table(
…  'product', metadata,
… Column( 'sku', String(20), primary_key=True),
… Column( 'msrp', Numeric),
… Column( 'image', BLOB))
>>> from sqlalchemy.orm import deferred
>>> mapper(Product, product_table, properties=dict(
…         image=deferred(product_table.c.image)))   #deferred 意思是
延迟,就是在实现 mapper 时，可以指定某些字段是 Deferred 装入的，这样象
通常一样取出数据时，这些字段并不真正的从数据库中取出，只有在你真正需要
时才取出，这样可以减少资源的占用和提高效率,只有在读取 image 时才会取出
相应的数据
<Mapper at 0x862a40c; Product>

>>> metadata.remove(product_table)   #因为已经常见了表,先删除
>>> product_table = Table(
…          'product', metadata,
…         Column( 'sku', String(20), primary_key=True),
…         Column( 'msrp', Numeric),
…         Column( 'image1' , Binary),
…         Column( 'image2' , Binary),
…         Column( 'image3' , Binary))

>>> clear_mappers() #已经映射,先取消
>>> mapper(Product, product_table, properties=dict(
…         image1=deferred(product_table.c.image1, group=' images' ),
…         image2=deferred(product_table.c.image2, group=' images' ),
…         image3=deferred(product_table.c.image3, group=' images' )))
#Deferred 字段可以通过在 properties 中指定 group 参数来表示编组情况。这
样当一个组的某个
```

#字段被取出时， 同组的其它字段均被取出
<Mapper at 0x85b8c4c; Product>

>>> q = product_table.join(被映射的是 join 了 product_summary_table 到 product_table 的结果
… product_summary_table,
…
product_table.c.sku==product_summary_table.c.sku).alias(‘full_product’)
>>> class FullProduct(object): pass
…
>>> mapper(FullProduct, q)
<Mapper at 0x86709cc; FullProduct>

mapper 函数的一些参数:

always_refresh =False:返回查询旧会修改内存中的值,但是 populate_existing 优先级高

allow_column_override =False:允许关系属性将具有相同的名称定义为一个映射列,否则名称冲突,产生异常

2 ORM 的关系

1 1:N relations (1 对多)

>>> mapper(Store, store_table)
<Mapper at 0x84fba4c; Store>
>>> from sqlalchemy.orm import relation
>>> mapper(Region, region_table, properties=dict(
…   stores=relation(Store))) #让 2 个表关联,给 Region 添加一个属性 stores,通过它联系 Store 来修改 Store
<Mapper at 0x84f76ac; Region>

>>> r0 = Region(‘test’)

>>> session.add(r0) #先生成一条数据
>>> session.commit()
2012-07-18 13:56:26,858 INFO sqlalchemy.engine.base.Engine BEGIN (implicit)
2012-07-18 13:56:26,859 INFO sqlalchemy.engine.base.Engine INSERT INTO region (name) VALUES (?)
2012-07-18 13:56:26,859 INFO sqlalchemy.engine.base.Engine (‘test’ ,)
2012-07-18 13:56:26,859 INFO sqlalchemy.engine.base.Engine COMMIT
>>> rgn = session.query(Region).get(1) #获取这条数据

```
2012-07-18 13:56:37,250 INFO sqlalchemy.engine.base.Engine BEGIN
(implicit)
2012-07-18 13:56:37,251 INFO sqlalchemy.engine.base.Engine SELECT
region.id AS region_id, region.name AS region_name
FROM region
WHERE region.id = ?
2012-07-18 13:56:37,251 INFO sqlalchemy.engine.base.Engine (1,)
>>> s0 = Store(name='3rd and Juniper') #创建一个实例
>>> rgn.stores.append(s0) #通过 Region 的依赖建立新的 Store(其中的一个
字段 region_id 值来着 region 的 id 字段)
2012-07-18 13:56:51,611 INFO sqlalchemy.engine.base.Engine SELECT
store.id AS store_id, store.region_id AS store_region_id, store.name AS
store_name
FROM store
WHERE ? = store.region_id
2012-07-18 13:56:51,611 INFO sqlalchemy.engine.base.Engine (1,)
>>> session.flush() #保存数据库
2012-07-18 13:57:02,131 INFO sqlalchemy.engine.base.Engine INSERT INTO
store (region_id, name) VALUES (?, ?)
2012-07-18 13:57:02,131 INFO sqlalchemy.engine.base.Engine (1, '3rd and
Juniper')
```

注:假如 2 个表之间有多个外部依赖关系,需要使用 primaryjoin 指定:

```
mapper(Region, region_table, properties=dict(
stores=relation(Store,
primaryjoin=(store_table.c.region_id    #判断关系来着 region_id 和
region 的 id
==region_table.c.id))))
```

2 M:N relations(多对多)

上面有 SQL 语句:我复制过来:

```
category_table = Table(
 'category', metadata,
Column('id', Integer, primary_key=True),
Column('level_id', None, ForeignKey('level.id')),
Column('parent_id', None, ForeignKey('category.id')),
Column('name', String(20)))
product_table = Table(
 'product', metadata,
Column('sku', String(20), primary_key=True),
Column('msrp', Numeric))
product_category_table = Table(
 'product_category', metadata,
```

```
Column( ‘product_id’ , None, ForeignKey( ‘product.sku’ ),
primary_key=True),
Column( ‘category_id’ , None, ForeignKey( ‘category.id’ ),
primary_key=True))
```

可以看出来 product_category_table 和 category_table 是多对多的关系.

```
>>> mapper(Category, category_table, properties=dict(
…         products=relation(Product,
…         secondary=product_category_table)))
<Mapper at 0x859c8cc; Category>
>>> mapper(Product, product_table, properties=dict(
…         categories=relation(Category,
…         secondary=product_category_table)))
<Mapper at 0x859c5cc; Product>

>>> r0=Product(’ 123′ ,’ 234′ )

>>> session.add(r0)
>>> session.flush()
2012-07-18 14:18:06,599 INFO sqlalchemy.engine.base.Engine BEGIN
(implicit)
2012-07-18 14:18:06,618 INFO sqlalchemy.engine.base.Engine INSERT INTO
product (sku, msrp) VALUES (?, ?)
2012-07-18 14:18:06,618 INFO sqlalchemy.engine.base.Engine (’ 123′ ,
234.0)
>>> session.query(Product).get(’ 123′ ).categories

>>> clear_mappers()
>>> mapper(Category, category_table, properties=dict(
…         products=relation(Product,
secondary=product_category_table,
… primaryjoin=(product_category_table.c.category_id   #primaryjoin 是
要被映射的表和连接表的条件
…
     == category_table.c.id),
…
secondaryjoin=(product_category_table.c.product_id   #secondaryjoin
是连接表和想加入的表的条件
…
         == product_table.c.sku))))
<Mapper at 0x84ff7cc; Category>
>>> mapper(Product, product_table, properties=dict(
…         categories=relation(Category,
secondary=product_category_table,
```

```
… primaryjoin=(product_category_table.c.product_id
…
         == product_table.c.sku),
… secondaryjoin=(product_category_table.c.category_id
…
             == category_table.c.id))))
<Mapper at 0x859cb8c; Product>
```

1:1 relations(一对一):特殊的(1:N)

还是上面的 SQL:

```
product_table = Table(
 'product', metadata,
Column('sku', String(20), primary_key=True),
Column('msrp', Numeric))
product_summary_table = Table(
 'product_summary', metadata,
Column('sku', None, ForeignKey('product.sku'), primary_key=True), #
只有一个外联到 product
Column('name', Unicode(255)),
Column('description', Unicode))

>>> mapper(Product, product_table, properties=dict(
…         summary=relation(ProductSummary)))
KeyboardInterrupt
>>> mapper(ProductSummary, product_summary_table)
<Mapper at 0x84fbe6c; ProductSummary>
>>> mapper(Product, product_table, properties=dict(
…         summary=relation(ProductSummary)))
<Mapper at 0x85bee6c; Product>
>>> prod = session.query(Product).get('123')
[]   #product_summary_table 因为 product_table 儿存在,浪费了

>>> mapper(ProductSummary, product_summary_table)
<Mapper at 0x84f7dec; ProductSummary>
>>> mapper(Product, product_table, properties=dict(
…         summary=relation(ProductSummary,uselist=False)))   #使用
uselist=False 就不会这样了
<Mapper at 0x860584c; Product>
>>> prod = session.query(Product).get('123')
>>> print prod.summary
None
>>> mapper(ProductSummary, product_summary_table)
<Mapper at 0x859ca0c; ProductSummary>
>>> mapper(Product, product_table, properties=dict(
…         summary=relation(ProductSummary, uselist=False,
```

```
…         backref=’product’))) #自定义自己表的函数
<Mapper at 0x860e90c; Product>
>>> prod = session.query(Product).get(’123′)
>>> prod.summary = ProductSummary(name=”Fruit”, description=”Some
… Fruit”)
>>> print prod.summary
<ProductSummary Fruit>
>>> print prod.summary.product #他的属性就是prod,可就是表本身
<Product 123>
>>> print prod.summary.product is prod
True

>>> mapper(Level, level_table, properties=dict(
…         categories=relation(Category, backref=’level’)))
<Mapper at 0x860590c; Level>
>>> mapper(Category, category_table, properties=dict(
…         products=relation(Product,
…                 secondary=product_category_table)))
<Mapper at 0x860ec8c; Category>
>>> mapper(Product, product_table, properties=dict(
…         categories=relation(Category,
…                 secondary=product_category_table)))
<Mapper at 0x860e7ec; Product>
>>> lvl = Level(name=’Department’)
>>> cat = Category(name=’Produce’, level=lvl)
>>> session.add(lvl)
>>> session.flush()
2012-07-18 14:44:02,005 INFO sqlalchemy.engine.base.Engine INSERT INTO
level (parent_id, name) VALUES (?, ?)
2012-07-18 14:44:02,005 INFO sqlalchemy.engine.base.Engine (None,
‘Department’)
2012-07-18 14:44:02,020 INFO sqlalchemy.engine.base.Engine INSERT INTO
category (level_id, parent_id, name) VALUES (?, ?, ?)
2012-07-18 14:44:02,020 INFO sqlalchemy.engine.base.Engine (1, None,
‘Produce’)
>>> prod = session.query(Product).get(’123′)
>>> print prod.categories
[]
>>> print cat.products
2012-07-18 14:44:25,517 INFO sqlalchemy.engine.base.Engine SELECT
product.sku AS product_sku, product.msrp AS product_msrp
FROM product, product_category
WHERE ? = product_category.category_id AND product.sku =
product_category.product_id
```

```
2012-07-18 14:44:25,517 INFO sqlalchemy.engine.base.Engine (1,)
[]
>>> prod.categories.append(cat)
>>> print prod.categories
[<Category Department.Produce>]
>>> print cat.products   #backref 自动更新,在多对多的情况,可以使用
relation 函数两次,但是 2 个属性没有保持同步
[]   #解决方法:

>>> mapper(Level, level_table, properties=dict(
…categories=relation(Category, backref=’ level’ )))
>>> mapper(Category, category_table, properties=dict(
…products=relation(Product, secondary=product_category_table,
… backref=’ categories’ )))   #在 Product 也设置 backref,就会保持同步
>>> mapper(Product, product_table)
>>> lvl = Level(name=’ Department’ )
>>> cat = Category(name=’ Produce’ , level=lvl)
>>> session.save(lvl)
>>> prod = session.query(Product).get(’ 123′ )
>>> print prod.categories
[]
>>> print cat.products
[]
>>> prod.categories.append(cat)
>>> print prod.categories
[<Category Department.Produce>]
>>>print cat.products
[<Product 123>]

>>> from sqlalchemy.orm import backref
>>> clear_mappers()
>>> mapper(ProductSummary, product_summary_table, properties=dict(
… product=relation(Product,
… backref=backref( ‘summary’ , uselist=False))))   #还可以使用
backref 函数做一样的事情
<Mapper at 0x860aaec; ProductSummary>
>>> mapper(Product, product_table)
<Mapper at 0x85bee6c; Product>
```

4 Self-Referential 自我参照映射

```
level_table = Table(
 ‘level’ , metadata,
Column( ‘id’ , Integer, primary_key=True),
Column( ‘parent_id’ , None, ForeignKey( ‘level.id’ )), #这个外联其实还
是这个类的 id,也就是映射了自己的对象
```

```
Column( ‘name’ , String(20)))
>>> mapper(Level, level_table, properties=dict(
… children=relation(Level))) #不同层次之间的父子关系,我这里指定得到”子”的属性
<Mapper at 0x860a66c; Level>
>>> mapper(Level, level_table, properties=dict(
…          children=relation(Level,
…          backref=backref( ‘parent’ ,
…          remote_side=[level_table.c.id]))))   #remote_side 指定’子’的 id,local side”就是字段 parent_id
<Mapper at 0x860e42c; Level>
>>> l0 = Level( ‘Gender’ )
>>> l1 = Level( ‘Department’ , parent=l0)
>>> session.add(l0)
>>> session.flush()
2012-07-18 15:07:55,810 INFO sqlalchemy.engine.base.Engine INSERT INTO level (parent_id, name) VALUES (?, ?)
2012-07-18 15:07:55,810 INFO sqlalchemy.engine.base.Engine (None, ‘Gender’ ) #插入 l0,他没有父级
2012-07-18 15:07:55,810 INFO sqlalchemy.engine.base.Engine INSERT INTO level (parent_id, name) VALUES (?, ?)
2012-07-18 15:07:55,810 INFO sqlalchemy.engine.base.Engine (2, ‘Department’ )
```

注 我们还能反过来用:

```
mapper(Level, level_table, properties=dict(
parent=relation(Level, remote_side=[level_table.c.parent_id],
backref=’ children’ )))
```

我们创建一个多引擎的例子:

Source code

```
from sqlalchemy import create_engine
from sqlalchemy.orm import mapper, sessionmaker
from sqlalchemy import Numeric,Table, MetaData, Column, ForeignKey, Integer, String
engine1 = create_engine('sqlite://')
engine2 = create_engine('sqlite://')
metadata = MetaData()
product_table = Table(
'product', metadata,
Column('sku', String(20), primary_key=True),
Column('msrp', Numeric))
```

```
product_summary_table = Table(
'product_summary', metadata,
Column('sku', String(20), ForeignKey('product.sku'),
primary_key=True),
Column('name', Unicode(255)),
Column('description', Unicode))
product_table.create(bind=engine1)
product_summary_table.create(bind=engine2)
stmt = product_table.insert()
engine1.execute(
stmt,
[dict(sku="123", msrp=12.34),
dict(sku="456", msrp=22.12),
dict(sku="789", msrp=41.44)])
stmt = product_summary_table.insert()
engine2.execute(
stmt,
[dict(sku="123", name="Shoes", description="Some Shoes"),
dict(sku="456", name="Pants", description="Some Pants"),
dict(sku="789", name="Shirts", description="Some Shirts")])
```

这样就创建了表并且插入了一些数据

```
dongwm@localhost ~ $ python
Python 2.7.3 (default, Jul 11 2012, 10:10:17)
[GCC 4.5.3] on linux2
Type “help”, “copyright”, “credits” or “license” for more
information.
>>> from sqlalchemy import create_engine
>>> from sqlalchemy.orm import mapper, sessionmaker
>>> from sqlalchemy import Numeric,Table, MetaData, Column, ForeignKey,
Integer, String,Unicode
>>> engine1 = create_engine( ‘sqlite://’ )
>>> engine2 = create_engine( ‘sqlite://’ )   #创建多个引擎
>>> metadata = MetaData()
>>> product_table = Table(
…  ‘product’, metadata,
… Column( ‘sku’, String(20), primary_key=True),
… Column( ‘msrp’, Numeric))
>>> product_summary_table = Table(
…  ‘product_summary’, metadata,
… Column( ‘sku’, String(20), ForeignKey( ‘product.sku’ ),
primary_key=True),
```

```
… Column( ‘name’ , Unicode(255)),
… Column( ‘description’ , Unicode))
>>> product_table.create(bind=engine1)
>>> product_summary_table.create(bind=engine2)
>>> stmt = product_table.insert()
>>> engine1.execute(
… stmt,
… [dict(sku="123", msrp=12.34),
... dict(sku="456", msrp=22.12),
... dict(sku="789", msrp=41.44)])
<sqlalchemy.engine.base.ResultProxy object at 0x84ef9ec>
>>> stmt = product_summary_table.insert()
>>> engine2.execute(   #用引擎 2 插入数据,那么 product_summary 的数据就在这个引擎
… stmt,
… [dict(sku="123", name="Shoes", description="Some Shoes"),
... dict(sku="456", name="Pants", description="Some Pants"),
... dict(sku="789", name="Shirts", description="Some Shirts")])
/usr/lib/python2.7/site-packages/SQLAlchemy-0.7.8-py2.7-linux-i686.egg/sqlalchemy/engine/default.py:463: SAWarning: Unicode type received non-unicode bind param value.
param.append(processors[key](compiled_params[key]))
<sqlalchemy.engine.base.ResultProxy object at 0x84e896c>
>>> class Product(object):
…         def __init__(self, sku, msrp, summary=None):
…                 self.sku = sku
…                 self.msrp = msrp
…                 self.summary = summary
…         def __repr__(self):
…                 return  ‘<Product %s>’  % self.sku
…
>>> class ProductSummary(object):
…         def __init__(self, name, description):
…                 self.name = name
…                 self.description = description
…         def __repr__(self):
…                 return  ‘<ProductSummary %s>’  % self.name
…
>>> from sqlalchemy.orm import clear_mappers,backref,relation
>>> clear_mappers()
>>> mapper(ProductSummary, product_summary_table, properties=dict(
…         product=relation(Product,
…                                         backref=backref( ‘summary’ , uselist=False))))
```

```
<Mapper at 0x84efa4c; ProductSummary>
>>> mapper(Product, product_table)
<Mapper at 0x84efd0c; Product>
>>> Session = sessionmaker(binds={Product:engine1,    #这里绑定了 2 个引擎,不同 orm 的引擎不同
…          ProductSummary:engine2})
>>> session = Session()
>>> engine1.echo = engine2.echo = True
>>> session.query(Product).all() #查询 product 的数据
2012-07-18 19:00:59,514 INFO sqlalchemy.engine.base.Engine BEGIN (implicit)
2012-07-18 19:00:59,514 INFO sqlalchemy.engine.base.Engine SELECT product.sku AS product_sku, product.msrp AS product_msrp
FROM product
2012-07-18 19:00:59,514 INFO sqlalchemy.engine.base.Engine ()
/usr/lib/python2.7/site-packages/SQLAlchemy-0.7.8-py2.7-linux-i686.egg/sqlalchemy/types.py:215: SAWarning: Dialect sqlite+pysqlite does *not* support Decimal objects natively, and SQLAlchemy must convert from floating point  -  rounding errors and other issues may occur. Please consider storing Decimal numbers as strings or integers on this platform for lossless storage.
d[coltype] = rp = d['impl'].result_processor(dialect, coltype)
[<Product 123>, <Product 456>, <Product 789>]
>>> session.query(ProductSummary).all() #查询 ProductSummary
2012-07-18 19:01:07,510 INFO sqlalchemy.engine.base.Engine BEGIN (implicit)
2012-07-18 19:01:07,510 INFO sqlalchemy.engine.base.Engine SELECT product_summary.sku AS product_summary_sku, product_summary.name AS product_summary_name, product_summary.description AS product_summary_description
FROM product_summary
2012-07-18 19:01:07,510 INFO sqlalchemy.engine.base.Engine ()
[<ProductSummary Shoes>, <ProductSummary Pants>, <ProductSummary Shirts>]

>>> from sqlalchemy.orm.shard import ShardedSession #使用 ShardedSession 对会话水平分区,根据需求把数据分开
>>> product_table = Table(
…          'product', metadata,
…          Column( 'sku' , String(20), primary_key=True),
…          Column( 'msrp' , Numeric))
>>> metadata.create_all(bind=engine1)
>>> metadata.create_all(bind=engine2)
>>> class Product(object):
```

```
…         def __init__(self, sku, msrp):
…                 self.sku = sku
…                 self.msrp = msrp
…         def __repr__(self):
…                 return  ‘<Product %s>’  % self.sku
…
>>> clear_mappers()
>>> product_mapper = mapper(Product, product_table)
>>> def shard_chooser(mapper, instance, clause=None):   #返回包含映射和实例的行的分区 ID
…         if mapper is not product_mapper: #非设定的 orm 映射叫做 odd
…                 return  ‘odd’
…         if (instance.sku   #数据为偶数也叫做 even
…                 and instance.sku[0].isdigit()
…                 and int(instance.sku[0]) % 2 == 0):
…                 return  ‘even’
…         else:
…                 return  ‘odd’  #否则叫做 odd
…

>>> def id_chooser(query, ident):   根据查询和映射类的主键返回对象想通过查询驻留的 shard ID 列表
…         if query.mapper is not product_mapper:
…                 return ['odd']
…         if (ident \
…                 and ident[0].isdigit()
…                 and int(ident[0]) % 2 == 0):
…                 return ['even']
…         return ['odd']
…
>>> def query_chooser(query): #返回可选的 shard ID 列表
…         return ['even', 'odd']
…
>>> Session = sessionmaker(class_=ShardedSession)
>>> session = Session(
…         shard_chooser=shard_chooser,
…         id_chooser=id_chooser,
…         query_chooser=query_chooser,
…         shards=dict(even=engine1,
…                                 odd=engine2))
>>> products = [ Product('%d%d%d' % (i,i,i), 0.0)
...          for i in range(10) ]
>>> for p in products:
…         session.add(p)
```

```
…
>>> session.flush()
>>> for row in engine1.execute(product_table.select()):
…         print row
…
2012-07-18 19:11:19,811 INFO sqlalchemy.engine.base.Engine SELECT
product.sku, product.msrp
FROM product
2012-07-18 19:11:19,811 INFO sqlalchemy.engine.base.Engine ()
(u'000', Decimal('0E-10')) #偶数数据写在 engine1
(u'222', Decimal('0E-10'))
(u'444', Decimal('0E-10'))
(u'666', Decimal('0E-10'))
(u'888', Decimal('0E-10'))
>>> for row in engine2.execute(product_table.select()):
…         print row
…
2012-07-18 19:11:40,098 INFO sqlalchemy.engine.base.Engine SELECT
product.sku, product.msrp
FROM product
2012-07-18 19:11:40,099 INFO sqlalchemy.engine.base.Engine ()
(u'111', Decimal('0E-10')) #奇数数据写在 engine1
(u'333', Decimal('0E-10'))
(u'555', Decimal('0E-10'))
(u'777', Decimal('0E-10'))
(u'999', Decimal('0E-10'))
>>> session.query(Product).all()
2012-07-18 19:12:36,130 INFO sqlalchemy.engine.base.Engine SELECT
product.sku AS product_sku, product.msrp AS product_msrp
FROM product
2012-07-18 19:12:36,130 INFO sqlalchemy.engine.base.Engine ()
2012-07-18 19:12:36,131 INFO sqlalchemy.engine.base.Engine SELECT
product.sku AS product_sku, product.msrp AS product_msrp
FROM product
2012-07-18 19:12:36,131 INFO sqlalchemy.engine.base.Engine ()
[<Product 123>, <Product 456>, <Product 789>, <Product 000>, <Product
222>, <Product 444>, <Product 666>, <Product 888>, <Product 111>,
<Product 333>, <Product 555>, <Product 777>, <Product 999>]

from sqlalchemy import create_engine
from sqlalchemy.orm import mapper, sessionmaker
from datetime import datetime
from sqlalchemy import Numeric,Table, MetaData, Column, ForeignKey,
Integer, String, Unicode, DateTime
```

```
from sqlalchemy import types
from sqlalchemy.databases import sqlite
engine1 = create_engine( ‘sqlite://’ )
engine2 = create_engine( ‘sqlite://’ )
metadata = MetaData()
product_table = Table(
 ‘product’, metadata,
Column( ‘sku’, String(20), primary_key=True),
Column( ‘msrp’, Numeric))
product_summary_table = Table(
 ‘product_summary’, metadata,
Column( ‘sku’, String(20), ForeignKey( ‘product.sku’ ),
primary_key=True),
Column( ‘name’, Unicode(255)),
Column( ‘description’, Unicode))
product_table.create(bind=engine1)
product_summary_table.create(bind=engine2)
stmt = product_table.insert()
engine1.execute(
stmt,
[dict(sku="123", msrp=12.34),
dict(sku="456", msrp=22.12),
dict(sku="789", msrp=41.44)])
stmt = product_summary_table.insert()
engine2.execute(
stmt,
[dict(sku="123", name="Shoes", description="Some Shoes"),
dict(sku="456", name="Pants", description="Some Pants"),
dict(sku="789", name="Shirts", description="Some Shirts")])
```

本文主要说删除

```
metadata.drop_all(engine) #删除某引擎的全部表

metadata.remove(test_table)   #删除某一个 table

clear_mappers() #取消所有的映射
```

在 relation 中有一个参数 cascade,它是基于 session 的操作,包括把对象放入 session,从 session 删除对象等,如果 指定 cascade=”all”表示做的任何 session 操作给映射类都能很好的工作,默认包含 save-update, merge

```
mapper(ParentClass, parent, properties=dict(
children=relation(ChildClass, backref=’parent’,
cascade=’all,delete-orphan’) )) #delete-orphan
```

表示如果曾经是子类(childclass)实例但是却没有和父类连接的情况下,假如要删除这个子类,而不

想挂空父类引用了的实例,
额看个例子就懂了:

```
photo = Table(
… ‘photo’, metadata,
… Column(‘id’, Integer, primary_key=True))
tag = Table(
… ‘tag’, metadata,
… Column(‘id’, Integer, primary_key=True),
… Column(‘photo_id’, None, ForeignKey(‘photo.id’)),
… Column(‘tag’, String(80)))
class Photo(object):
… pass
…
class Tag(object):
… def __init__(self, tag):
… self.tag = tag
…
mapper(Photo, photo, properties=dict(
… tags=relation(Tag, backref=’photo’, cascade=”all”)))
<Mapper at 0x851504c; Photo>
>>> mapper(Tag, tag)
<Mapper at 0x8515dac; Tag>
>>> p1 = Photo()
>>> p2 = Photo()
>>> p1.tags = [Tag(tag='foo'),Tag(tag='bar'),Tag(tag='baz')]
>>> p2.tags = [Tag(tag='foo'),Tag(tag='bar'),Tag(tag='baz')]
>>> session.add(p1)
>>> session.add(p2)
>>> session.flush()
>>> for t in session.query(Tag):
… print t.id,t.photo_id, t.tag
…
1 1 foo #出现以下关联数据
2 1 bar
3 1 baz
4 2 foo
5 2 bar
6 2 baz
>>> session.delete(session.query(Photo).get(1)) #删除一个 tag 的数据
>>> session.flush()
>>> for t in session.query(Tag):
… print t.id, t.photo_id, t.tag
…
4 2 foo #他会删除关联所有 t.photo_id 为 1 的数据,在这里
```

relation(ChildClass, backref=’parent’, cascade=’all,delete-orphan’)
指定 delete-orphan 没什么,关键看下面
5 2 bar
6 2 baz
>>> p3 = session.query(Photo).get(2)
>>> del p3.tags[0] #如果我只是删除关联点…
>>> session.flush()
>>> for t in session.query(Tag):
… print t.id, t.photo_id, t.tag
…
4 None foo #关联点 photo_id 成了 none,但是条目存在 -他不会影响其它关联表
5 2 bar
6 2 baz

>>> p3 = session.query(Photo).get(2) #假如没有设置 delete-orphan
>>> del p3.tags[0]
>>> session.flush()
>>> for t in session.query(Tag):
… print t.id, t.photo_id, t.tag
5 2 bar #自动删除了关联的其它表的项
6 2 baz
注:可用的 cascade 参数包含:

- save-update -我的理解是调用 session.add()会自动将项目添加到相应级联关系上,也适用于已经从关系中删除的项目嗨没有来得及刷新的情况
- merge - 它是 session.merge 的实现,复制状态到具有相同标识符的持久化实例的实例,如果没有持久的实例和当前 session 相关联，返回的持久化实例。如果给定的实例未保存，他会保存一个副本，并返回这个副本作为一个新的持久化实例
- expunge - 从 session 中删除实例
- delete - 标记一个实例被删除,执行 flush()会执行删除操作
- delete-orphan-如果子类从母类删除,标记之,但是不影响母类
- refresh-expire - 定期刷新在给定的实例的属性,查询并刷新数据库
- all - 以上全部属性的集合:“save-update,merge, refresh-expire, expunge, delete
- **本文主要是 ORM 的 sission 查询和更新**
- session 负责执行内存中的对象和数据库表之间的同步工作,创建 session 可以这样:
- Session = sessionmaker(bind=engine)
  #sqlalchemy.orm.session.Session 类有很多参数,使用 sessionmaker 是为了简化这个过程

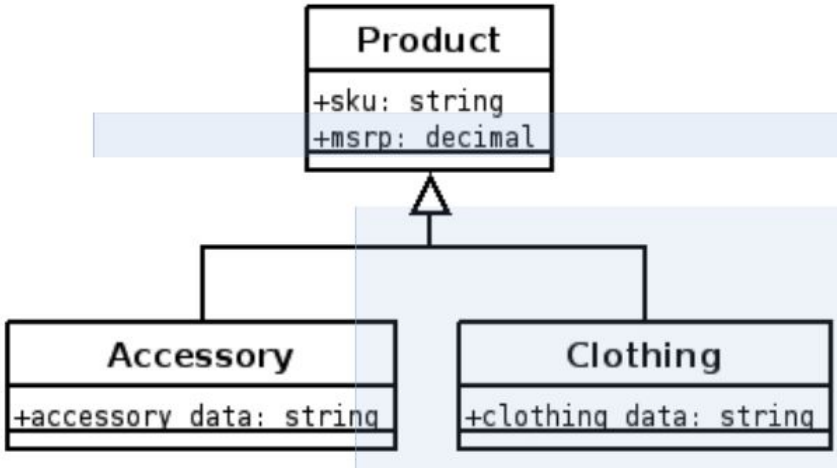
- 或者:
  Session = sessionmaker()
  Session.configure(bind=engine)
- 注:sessionmaker 的参数:
  autoflush=True　#为 True 时，session 将在执行 session 的任何查询前自动调用 flush()。这将确保返回的结果
- transactional=False #为 True 时，session 将自动使用事务 commit
  twophase=False #当处理多个数据库实例,当使用 flush()但是没有提交事务 commit 时,给每个数据库一个标识,使整个事务回滚
- 创建 session,添加数据的例子(以前也出现过很多次了)
  dongwm@localhost ~ $ python
  Python 2.7.3 (default, Jul 11 2012, 10:10:17)
  [GCC 4.5.3] on linux2
  Type “help”, “copyright”, “credits” or “license” for more information.
  >>> from sqlalchemy import *
  >>> from sqlalchemy.orm import *
  >>> engine = create_engine(‘sqlite://’)
  >>> metadata = MetaData(engine)
  >>> account_table = Table(
  … ‘account’, metadata,
  … Column(‘id’, Integer, primary_key=True),
  … Column(‘balance’, Numeric))
  >>> class Account(object): pass
  …
  >>> mapper(Account, account_table)
  <Mapper at 0x84e6f2c; Account>
  >>> account_table.create()
  >>> a = Account()
  >>> a.balance = 100.00
  >>> Session = sessionmaker(bind=engine)
  >>> session = Session()
  >>> session.add(a)
  >>> session.flush()
  >>> session.delete(a) #自动删除 account_table 相应条目,但是在 1:N 和 M:N 关系中不会自动删除它的级联关系
  >>> session.flush()
- **注:session 的对象状态:**
- Transient:短暂的,主要指内存中的对象
- Pending:挂起的,这样的对象准备插入数据库,等执行了 flush 就会插入
- Persistent:持久的
- Detached:对象在数据库里面有记录,但是不属于 session
- >>> make_transient(a)　#因为标识了已删除,恢复 a 的状态
  >>> session.add(a) #重新添加

```
>>> session.flush()
>>> query = session.query(Account)
>>> print query
SELECT account.id AS account_id, account.balance AS
account_balance
FROM account
>>> for obj in query:
…         print obj
…
<__main__.Account object at 0x84eef0c>
```

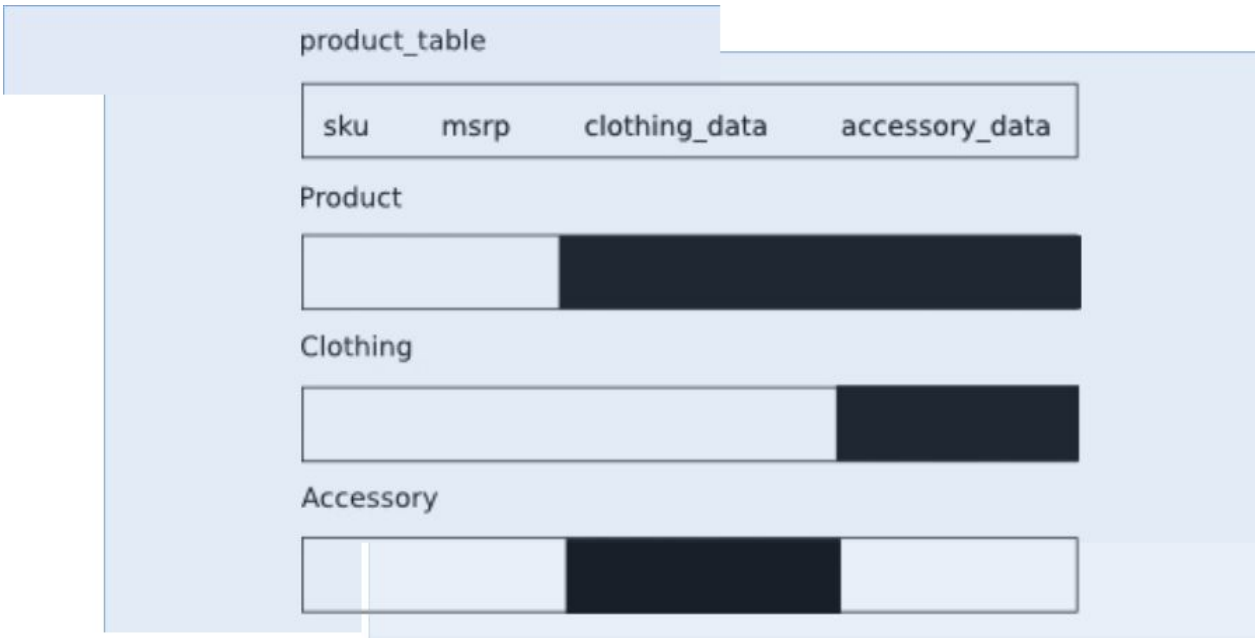
- `>>> query.all()  #查询所有`
  `[<__main__.Account object at 0x84eef0c>]`
  `>>> query = query.filter(Account.balance > 10.00)  #filter 过滤`
  `>>> for obj in query:`
  `…         print obj.balance`
  `…`
- 100.00
- `>>> for i in session.query(Account).filter_by(balance=100.00 ):  #通过条件过滤`
  `…         print i`
  `>>> query = session.query(Account)`
  `>>> query = query.from_statement( ‘select *from account where balance=:bac’ ) #通过带通配符的 SQL 语句其中:bac 标识这个参数是 bac`
  `>>> query = query.params(bac=’ 100.00′ ) #根据 bac 指定值寻找`
  `>>> print query.all()`
  `[<__main__.Account object at 0x84eef0c>]`
- **本地 session**
- `>>> Session = scoped_session(sessionmaker(  #设置一个本地的共享 session`
  `…         bind=engine, autoflush=True))`
  `>>> session = Session()`
  `>>> session2 = Session()`
  `>>> session is session2  #他们是同一个`
  `True`
- `>>> a = Account()`
  `>>> a.balance = 100.00`
  `>>> Session.add(a) #注意 这是的’ S’是大写`
  `>>> Session.flush()`
  `>>> b = Account()`
  `>>> a.balance = 200.00`
  `>>> session.add(a)  #其实他们是一个共享的 session 名字都可以`
  `>>> session.flush()`

>>> print session.query(Account).all() #查询到了 2 个
[<__main__.Account object at 0x851be0c>, <__main__.Account object at 0x84f7d6c>]

- 注:这样的映射 mapper 也可以这样是用:
- mapper(Product, product_table, properties=dict(
categories=relation(Category, secondary=product_category_table, backref=’products’)))
- Session.mapper(Product, product_table, properties=dict(
categories=relation(Category, secondary=product_category_table, backref=’products’))) #它的优点是可以初始化参数
- **本文主要是面向对象的继承映射到关系数据库表的方法**
- >>> class Product(object):
… def __init__(self, sku, msrp):
… self.sku = sku
… self.msrp = msrp
… def __repr__(self):
… return ‘<%s %s>’ % (
… self.__class__.__name__, self.sku)
…
>>> class Clothing(Product):
… def __init__(self, sku, msrp, clothing_info):
… Product.__init__(self, sku, msrp) #继承了 Product
… self.clothing_info = clothing_info
…
>>> class Accessory(Product):
… def __init__(self, sku, msrp, accessory_info):
… Product.__init__(self, sku, msrp) #继承了 Product
… self.accessory_info = accessory_info
也就是这样的意思:



- 这个单表继承中(如下图,黑色的表示没有被映射):



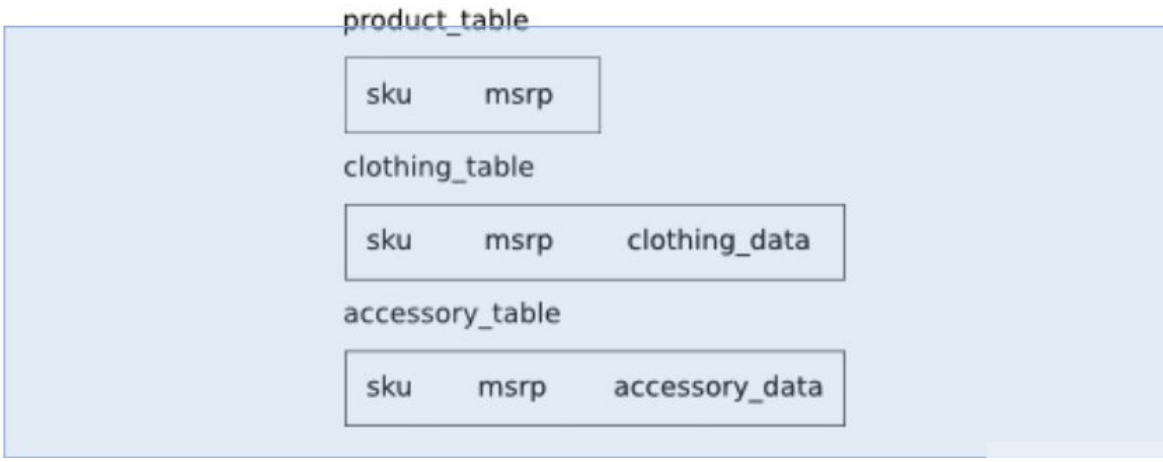
- 从创建表结构是这样:
- >>> product_table = Table(
  … ‘product’, metadata,
  … Column(‘sku’, String(20), primary_key=True),
  … Column(‘msrp’, Numeric),
  … Column(‘clothing_info’, String),
  … Column(‘accessory_info’, String),
  … Column(‘product_type’, String(1), nullable=False))
  #一个新的字段
  >>> mapper(
  … Product, product_table,
  … polymorphic_on=product_table.c.product_type, #映射继承层次结构使用 polymorphic_on 表示继承在 product_type 字段,值是 polymorphic_identity 指定的标识
  … polymorphic_identity=’P’) #标识继承 Product ,父类
  <Mapper at 0x85833ec; Product>
  >>> mapper(Clothing, inherits=Product,
  … polymorphic_identity=’C’) #标识继承 Clothing product
  <Mapper at 0x858362c; Clothing>
  >>>
  >>> mapper(Accessory, inherits=Product, #继承至 Product
  … polymorphic_identity=’A’) #标识继承 Accessory
  <Mapper at 0x8587d8c; Accessory>
  >>> products = [#创建一些产品
  ... Product('123', 11.22),
  ... Product('456', 33.44),

```
...         Clothing('789', 123.45, "Nice Pants"),
...         Clothing('111', 125.45, "Nicer Pants"),
...         Accessory('222', 24.99, "Wallet"),
...         Accessory('333', 14.99, "Belt") ]
>>> Session = sessionmaker()
>>> session = Session()
>>> for p in products:
…         session.add(p)
…
>>> session.flush()
>>> print session.query(Product).all() #全部都有
[<Product 123>, <Product 456>, <Clothing 789>, <Clothing 111>,
<Accessory 222>, <Accessory 333>]
>>> print session.query(Clothing).all()   #只显示 2 个
[<Clothing 789>, <Clothing 111>]
>>> print session.query(Accessory).all() #只显示 2 个,是不是上面
的映射效果和创建 3 个类而分别 orm 好的多呢?
[<Accessory 222>, <Accessory 333>]
```

- >>> for row in product_table.select().execute(): #从父类库查询,所有数据都有,只是 product_type 不同

```
…         print row
…
(u' 123' , Decimal(' 11.2200000000' ), None, None, u' P')
(u' 456' , Decimal(' 33.4400000000' ), None, None, u' P')
(u' 789' , Decimal(' 123.4500000000' ), u' Nice Pants' , None,
u' C')
(u' 111' , Decimal(' 125.4500000000' ), u' Nicer Pants' , None,
u' C')
(u' 222' , Decimal(' 24.9900000000' ), None, u' Wallet' , u' A')
(u' 333' , Decimal(' 14.9900000000' ), None, u' Belt' , u' A')
```

- 具体的映射见下图:



- 查询一个没有的不存在的映射:

- >>> print session.query(Accessory)[0].clothing_info
  None
- **具体表的继承**
- 每个表包含的数据量,需要实现它的类;没有浪费的空间
- >>> metadata.remove(product_table)
  >>> product_table = Table(
  … 'product', metadata,
  … Column('sku', String(20), primary_key=True),
  … Column('msrp', Numeric))
  >>> clothing_table = Table(
  … 'clothing', metadata,
  … Column('sku', String(20), primary_key=True),
  … Column('msrp', Numeric),
  … Column('clothing_info', String))
  >>>
  >>> accessory_table = Table(
  … 'accessory', metadata,
  … Column('sku', String(20), primary_key=True),
  … Column('msrp', Numeric),
  … Column('accessory_info', String))
  >>>
  摄像我们想要获取Product'sku'是222的数据(没有其他额外的工作),我们不得不层次型的查询每个类,请看这个例子:
- >>> punion = polymorphic_union(
  … dict(P=product_table,
  … C=clothing_table,
  … A=accessory_table),
  … 'type_')
  >>>
  >>> print punion
  SELECT accessory.sku, accessory.msrp, accessory.accessory_info,
  CAST(NULL AS VARCHAR) AS clothing_info, 'A' AS type_
  FROM accessory UNION ALL SELECT product.sku, product.msrp,
  CAST(NULL AS VARCHAR) AS accessory_info, CAST(NULL AS VARCHAR) AS
  clothing_info, 'P' AS type_
  FROM product UNION ALL SELECT clothing.sku, clothing.msrp,
  CAST(NULL AS VARCHAR) AS accessory_info, clothing.clothing_info,
  'C' AS type_
  FROM clothing
  现在我们就有了一个很好的标记了(C,A,P)
- >>> mapper(
  … Product, product_table, with_polymorphic=('*',
  punion), #使用with_polymorphic=('*', punion)的方式映射父类,指定不同表选择,实现多态,并且提高了性能(只select了一次)

```
… polymorphic_on=punion.c.type_,
… polymorphic_identity=’ P’)
<Mapper at 0x8605b6c; Product>
>>> mapper(Clothing, clothing_table, inherits=Product,
… polymorphic_identity=’ C’,
… concrete=True)
<Mapper at 0x84f1bac; Clothing>
>>> mapper(Accessory, accessory_table, inherits=Product,
… polymorphic_identity=’ A’,
… concrete=True)
<Mapper at 0x858770c; Accessory>
```

- >>> session.query(Product).get(’ 222′)
  <Accessory 222>
- **本文主要是讲关于 sqlalchemy 的扩展**
- 扩展其实就是一些外部的插件,比如 sqlsoup,associationproxy,declarative,horizontal_shard 等等
- **1 declarative**
- 假如想要数据映射,以前的做法是:

Source code

```
from sqlalchemy import create_engine
from sqlalchemy import Column, MetaData, Table
from sqlalchemy import Integer, String, ForeignKey
from sqlalchemy.orm import mapper, sessionmaker

class User(object): #简单类

    def __init__(self, name, fullname, password):
        self.name = name
        self.fullname = fullname
        self.password = password
    def __repr__(self):
        return "<User('%s','%s', '%s')>" % (self.name,
self.fullname, self.password)
metadata = MetaData()
users_table = Table('users', metadata,
Column('user_id', Integer, primary_key=True),
Column('name', String),
Column('fullname', String),
Column('password', String)
)
email_table = Table('email', metadata,
Column('email_id', Integer, primary_key=True),
Column('email_address', String),
```

- Column('user_id', Integer, ForeignKey('users.user_id'))
-)
- metadata.create_all(engine)
- mapper(User, users_table) #映射
- 但是我们可以该换风格,可以用这样的方法:

Source code

- from sqlalchemy import Column, Integer, String, ForeignKey
- from sqlalchemy import create_engine
- from sqlalchemy.ext.declarative import declarative_base
- from sqlalchemy.orm import backref, mapper, relation, sessionmaker
- Base = declarative_base()
- class User(Base):
- __tablename__ = "users" #设定接收映射的表名
- id = Column(Integer, primary_key=True) #将表结构写到类里面
- name = Column(String)
- fullname = Column(String)
- password = Column(String)
- def __init__(self, name, fullname, password):
- self.name = name
- self.fullname = fullname
- self.password = password
- def __repr__(self):
- return "<User('%s','%s', '%s')>" % (self.name, self.fullname, self.password)
- class Address(Base):
- __tablename__ = "addresses"
- id = Column(Integer, primary_key=True)
- email_address = Column(String, nullable=False)
- user_id = Column(Integer, ForeignKey('users.id'))
- user = relation(User, backref=backref('addresses', order_by=id)) #创建双向关系,标识以 user 的 id 为连接,也就是说:Address 到 User 是多对一,User 到 Address 是一对多
- def __init__(self, email_address):

- self.email_address = email_address
- def __repr__(self):
- return "<Address('%s')>" % self.email_address
- engine = create_engine("sqlite:///tutorial.db", echo=True)
- users_table = User.__table__ #获取 User 表对象句柄
- metadata = Base.metadata #获取 metadata 句柄
- metadata.create_all(engine)
- 下面具体说:
- engine = create_engine(‘sqlite://’) #创建引擎
  Base.metadata.create_all(engine) #常见表
  Base.metadata.bind = create_engine(‘sqlite://’) #绑定
  Base = declarative_base(bind=create_engine(‘sqlite://’)) #绑定引擎
  mymetadata = MetaData()
  Base = declarative_base(metadata=mymetadata) #设定元数据**设定简单关系:**
  class User(Base):
  __tablename__ = ‘users’ id = Column(Integer, primary_key=True)
  name = Column(String(50))
  addresses = relationship(“Address”, backref=”user”)
  #relationship 其实就是 relation 的全称
- class Address(Base):
  __tablename__ = ‘addresses’
- id = Column(Integer, primary_key=True)
  email = Column(String(50))
  user_id = Column(Integer, ForeignKey(‘users.id’))
  设定多对多关系:
  keywords = Table(
  ‘keywords’, Base.metadata,
  Column(‘author_id’, Integer, ForeignKey(‘authors.id’)),
  Column(‘keyword_id’, Integer, ForeignKey(‘keywords.id’))
)
  class Author(Base):
  __tablename__ = ‘authors’
  id = Column(Integer, primary_key=True)
  keywords = relationship(“Keyword”, secondary=keywords)
  定义 SQL 表达式:
  class MyClass(Base):
  __tablename__ = ‘sometable’
  __table_args__ = {‘mysql_engine’:’InnoDB’} #名字,映射类,元数据之外的指定需要使用__table_args__

- 或者:
  class MyClass(Base):
  __tablename__ = ‘sometable’
  __table_args__ = (
  ForeignKeyConstraint([’id’], [’remote_table.id’]), #元组方式
  UniqueConstraint(‘foo’),
)
- 或者:
  class MyClass(Base):
  __tablename__ = ‘sometable’
  __table_args__ = (
  ForeignKeyConstraint([’id’], [’remote_table.id’]),
  UniqueConstraint(‘foo’),
  {‘autoload’:True} #最后的参数可以用字典 想想*argsand **kwargs
)

  使用混合式:

  class MyClass(Base):
  __table__ = Table(‘my_table’, Base.metadata, #在__table__里指定表结构
  Column(‘id’, Integer, primary_key=True),
  Column(‘name’, String(50))
)

  2 sqlsoup(在 sqlalchemy0.8 版本后他变成了一个独立的项目,http://pypi.python.org/pypi/sqlsoup,
- **而我使用 gentoo 提供的 0.7.8 版本,以下的程序 import 部分可能不适用更高版本,而需要 import sqlsoup)**
- sqlsoup 提供一个方便的访问数据库的接方式,而无需创建类,映射数据库
- 还是看例子的对比:
- 用以前的方式创建一个数据库并且插入一些数据:
- >>> from sqlalchemy import *
  >>> engine = create_engine(‘sqlite:///dongwm.db’)
  >>> metadata = MetaData(engine)
  >>> product_table = Table(
  … ‘product’, metadata,
  … Column(‘sku’, String(20), primary_key=True),
  … Column(‘msrp’, Numeric))
  >>> store_table = Table(
  … ‘store’, metadata,
  … Column(‘id’, Integer, primary_key=True),
  … Column(‘name’, Unicode(255)))
  >>> product_price_table = Table(
  … ‘product_price’, metadata,
  … Column(‘sku2′, None, ForeignKey(‘product.sku’),

```
primary_key=True),
… Column( ‘store_id’ , None, ForeignKey( ‘store.id’ ),
primary_key=True),
…          Column( ‘price’ , Numeric, default=0))
>>> metadata.create_all()
>>> stmt = product_table.insert()
>>> stmt.execute([dict(sku="123", msrp=12.34),
...                               dict(sku="456", msrp=22.12),
...                               dict(sku="789", msrp=41.44)])
<sqlalchemy.engine.base.ResultProxy object at 0x84fbdcc>
>>> stmt = store_table.insert()
>>> stmt.execute([dict(name="Main Store"),
...                               dict(name="Secondary Store")])
<sqlalchemy.engine.base.ResultProxy object at 0x850068c>
>>> stmt = product_price_table.insert()
>>> stmt.execute([dict(store_id=1, sku="123"),
...                               dict(store_id=1, sku2="456"),
...                               dict(store_id=1, sku2="789"),
...                               dict(store_id=2, sku2="123"),
...                               dict(store_id=2, sku2="456"),
...                               dict(store_id=2, sku2="789")])
<sqlalchemy.engine.base.ResultProxy object at 0x85008cc>
```

创建插入完毕,然后我们用 sqlsoup 连接操作:

- ```
  >>> from sqlalchemy.ext.sqlsoup import SqlSoup
  >>> db = SqlSoup( ‘sqlite:///dongwm.db’ )   #连接一个存在的数据库
  >>> print db.product.all() #打印结果
  [MappedProduct(sku=u'123',msrp=Decimal('12.3400000000')),
  MappedProduct(sku=u'456',msrp=Decimal('22.1200000000')),
  MappedProduct(sku=u'789',msrp=Decimal('41.4400000000'))]
  >>> print db.product.get(’ 123′ ) #是不是比 session.query(Product) 简单呢?
  MappedProduct(sku=u’ 123′ ,msrp=Decimal(’ 12.3400000000′ ))
  ```
- 注:假如想创建一个数据库: db = SqlSoup(‘sqlite:///:memory:’)
- ```
  >>> newprod = db.product.insert(sku=’ 111′ , msrp=22.44) #没有使用数据映射的插入
  >>> db.flush()
  >>> db.clear() #调用底层,清除所有 session 实例,它是 session.expunge_all 的别名
  >>> db.product.all()
  [MappedProduct(sku=u'123',msrp=Decimal('12.3400000000')),
  MappedProduct(sku=u'456',msrp=Decimal('22.1200000000')),
  MappedProduct(sku=u'789',msrp=Decimal('41.4400000000')),
  MappedProduct(sku=u'111',msrp=Decimal('22.4400000000'))] #新条
  ```

目已经存在了
#MappedProduct 使用__getattr__将无法识别的属性和访问方法转发到它的 query 属性,它还提供了一些数据处理功能用于更新

- >>> from sqlalchemy import or_, and_, desc
  >>> where = or_(db.product.sku==' 123' , db.product.sku==' 111')
  >>>
  db.product.filter(where).order_by(desc(db.product.msrp)).all()
  #这样使用多条件过滤,降序排练
  [MappedProduct(sku='111',msrp=22.44),
  MappedProduct(sku=u'123',msrp=Decimal('12.3400000000'))]
- >>> join1 = db.join(db.product, db.product_price, isouter=True) #关联 2 个表, isouter=True 确保 LEFT OUTER(还没理解)
  >>> join1.all()
  [MappedJoin(sku=u'123',msrp=Decimal('12.3400000000'),sku2=u'123',store_id=1,price=Decimal('0E-10')), #这个字段包含了 2 个表的相应字段
  MappedJoin(sku=u'123',msrp=Decimal('12.3400000000'),sku2=u'123',store_id=2,price=Decimal('0E-10')),
  MappedJoin(sku=u'456',msrp=Decimal('22.1200000000'),sku2=u'456',store_id=1,price=Decimal('0E-10')),
  MappedJoin(sku=u'456',msrp=Decimal('22.1200000000'),sku2=u'456',store_id=2,price=Decimal('0E-10')),
  MappedJoin(sku=u'789',msrp=Decimal('41.4400000000'),sku2=u'789',store_id=1,price=Decimal('0E-10')),
  MappedJoin(sku=u'789',msrp=Decimal('41.4400000000'),sku2=u'789',store_id=2,price=Decimal('0E-10')),
  MappedJoin(sku=u'111',msrp=Decimal('22.4400000000'),sku2=None,store_id=None,price=None)]
  >>> join2 = db.join(join1, db.store, isouter=True) #将 store 表也关联进来(因为也有一个外键),就是关联三个表
  >>> join2.all()
  [MappedJoin(sku=u'123',msrp=Decimal('12.3400000000'),sku2=u'123',store_id=1,price=Decimal('0E-10'),id=1,name=u'Main Store'),
  MappedJoin(sku=u'123',msrp=Decimal('12.3400000000'),sku2=u'123',store_id=2,price=Decimal('0E-10'),id=2,name=u'Secondary Store'),
  MappedJoin(sku=u'456',msrp=Decimal('22.1200000000'),sku2=u'456',store_id=1,price=Decimal('0E-10'),id=1,name=u'Main Store'),
  MappedJoin(sku=u'456',msrp=Decimal('22.1200000000'),sku2=u'456',store_id=2,price=Decimal('0E-10'),id=2,name=u'Secondary Store'),
  MappedJoin(sku=u'789',msrp=Decimal('41.4400000000'),sku2=u'789',store_id=1,price=Decimal('0E-10'),id=1,name=u'Main Store'),
  MappedJoin(sku=u'789',msrp=Decimal('41.4400000000'),sku2=u'789',store_id=2,price=Decimal('0E-10'),id=2,name=u'Secondary Store'),
  MappedJoin(sku=u'111',msrp=Decimal('22.4400000000'),sku2=None,s

tore_id=None,price=None,id=None,name=None)]
>>> join3 = db.with_labels(join1) #根据原籍标记,比如 sku 会说出:product_sku,告诉你它来着 product 表,但是指定了 jion1,就不会标识关于 store 的表
>>> join3.first()
MappedJoin(product_sku=u’123′,product_msrp=Decimal(’12.3400000′),product_price_sku2=u’123′,product_price_store_id=1,product_price_price=Decimal(’0E-10′))
>>> db.with_labels(join2).first()
MappedJoin(product_sku=u’123′,product_msrp=Decimal(’12.3400000′),product_price_sku2=u’123′,product_price_store_id=1,product_price_price=Decimal(’0E-10′),store_id=1,store_name=u’Main Store’)
>>> labelled_product = db.with_labels(db.product)
>>> join4 = db.join(labelled_product,
db.product_price, isouter=True)
>>> join4.first()
MappedJoin(product_sku=u’123′,product_msrp=Decimal(’12.3400000′),sku2=u’123′,store_id=1,price=Decimal(’0E-10′))

- >>> db.clear()
>>> join5 = db.join(db.product, db.product_price)
>>> s = select([db.product._table,
... func.avg(join5.c.price).label('avg_price')], #添加一个字段计算产品(product)的 price 平均值,字段名为 avg_price
… from_obj=[join5._table],
… group_by=[join5.c.sku])
>>> s = s.alias(‘products_with_avg_price’) #它是 from sqlalchemy import alias; a = alias(self, name=name)的简写
>>> products_with_avg_price = db.map(s, primary_key=[join5.c.sku]) #因为没有映射到表或者 join,需要指定如何找到主键
>>> products_with_avg_price.all()
[MappedJoin(sku=u'123',msrp=Decimal('12.3400000000'),avg_price=0.0),
MappedJoin(sku=u'456',msrp=Decimal('22.1200000000'),avg_price=0.0),
MappedJoin(sku=u'789',msrp=Decimal('41.4400000000'),avg_price=0.0)]
>>> db.product_price.first().price = 50.00
>>> db.flush()
>>> products_with_avg_price.all()
[MappedJoin(sku=u'123',msrp=Decimal('12.3400000000'),avg_price=0.0),
MappedJoin(sku=u'456',msrp=Decimal('22.1200000000'),avg_price=0.0),

MappedJoin(sku=u'789',msrp=Decimal('41.4400000000'),avg_price=0.0)]

>>> db.products_with_avg_price = products_with_avg_price #保存映射到 db,方便重用

>>> msrp=select([db.product.c.msrp],

… db.product.sku==db.product_price.sku2) #获取 sku 和 sku2 相等时候 msrp 的值

>>> db.product_price.update(#更新数据

… values=dict(price=msrp),synchronize_session=False) #设置 price 这个字段值为上面对应的 msrp

- 6

>>> db.product_price.all()

[MappedProduct_price(sku2=u'123',store_id=1,price=Decimal('12.340000000')),
MappedProduct_price(sku2=u'456',store_id=1,price=Decimal('22.1200000000')),
MappedProduct_price(sku2=u'789',store_id=1,price=Decimal('41.4400000000')),
MappedProduct_price(sku2=u'123',store_id=2,price=Decimal('12.3400000000')),
MappedProduct_price(sku2=u'456',store_id=2,price=Decimal('22.1200000000')),
MappedProduct_price(sku2=u'789',store_id=2,price=Decimal('41.4400000000'))]

- 3 associationproxy
- associationproxy 用于创建一个读/写整个关系的目标属性
- 看一个例子就懂了:
- >>> user_table = Table(

… ‘user’, metadata,

… Column(‘id’, Integer, primary_key=True),

… Column(‘user_name’, String(255), unique=True),

… Column(‘password’, String(255)))

>>> brand_table = Table(

… ‘brand’, metadata,

… Column(‘id’, Integer, primary_key=True),

… Column(‘name’, String(255)))

>>> sales_rep_table = Table(

… ‘sales_rep’, metadata,

… Column(‘brand_id’, None, ForeignKey(‘brand.id’), primary_key=True),

… Column(‘user_id’, None, ForeignKey(‘user.id’), primary_key=True),

… Column(‘commission_pct’, Integer, default=0))

>>> class User(object): pass

```
…
>>> class Brand(object): pass
…
>>> class SalesRep(object): pass
…
>>> mapper(User, user_table, properties=dict(
…         sales_rep=relation(SalesRep, backref=’user’,
uselist=False)))
<Mapper at 0x87472ec; User>
>>> mapper(Brand, brand_table, properties=dict(
…         sales_reps=relation(SalesRep, backref=’brand’)))
<Mapper at 0x874770c; Brand>
>>> mapper(SalesRep, sales_rep_table)
<Mapper at 0x874768c; SalesRep>
```

- ORM 完成,但是假如我们想要 brand(品牌)类对象的一个所有 SalesReps for Brand(品牌的销售代表)的 User 列表属性,可以这样:
- class Brand(object):
  @property
  def users(self):
  return [sr.user for sr in self.sales_reps]
- 但是不方便增加删除,而使用 association_proxy:
- >>> from sqlalchemy.ext.associationproxy import association_proxy
  >>> class Brand(object):
  … users=association_proxy(‘sales_reps’, ‘user’)
  …
- 或者:
- mapper(Brand, brand_table, properties=dict(
  sales_reps=relation(SalesRep, backref=’brand’)))
  Brand.users=association_proxy(‘sales_reps’, ‘user’)#优点是维持了域对象
- 我们需要修改类,增加属性:
- class User(object):
  def __init__(self, user_name=None, password=None):
  self.user_name=user_name
  self.password=password
- class Brand(object):
  def __init__(self, name=None):
  self.name = name
- class SalesRep(object):
  def __init__(self, user=None, brand=None, commission_pct=0):
  self.user = user
  self.brand = brand
  self.commission_pct=commission_pct

- 看下面的效果:
- >>> b = Brand(‘Cool Clothing’)
  >>> session.add(b)
  >>> u = User(‘rick’ , ‘foo’)
  >>> session.add(u)
  >>> session.flush()
  2012-07-20 12:22:33,191 INFO sqlalchemy.engine.base.Engine INSERT INTO user (user_name, password) VALUES (?, ?)
  2012-07-20 12:22:33,191 INFO sqlalchemy.engine.base.Engine (‘rick’ , ‘foo’)
  2012-07-20 12:22:33,191 INFO sqlalchemy.engine.base.Engine INSERT INTO brand (name) VALUES (?)
  2012-07-20 12:22:33,191 INFO sqlalchemy.engine.base.Engine (‘Cool Clothing’ ,)
  >>> b.users
  2012-07-20 12:22:42,135 INFO sqlalchemy.engine.base.Engine SELECT sales_rep.brand_id AS sales_rep_brand_id, sales_rep.user_id AS sales_rep_user_id, sales_rep.commission_pct AS sales_rep_commission_pct
  FROM sales_rep
  WHERE ? = sales_rep.brand_id
  2012-07-20 12:22:42,135 INFO sqlalchemy.engine.base.Engine (2,)
  []
  >>> b.users.append(u) #自动创建一个单一的位置参数调用其中介(SalesRep)对象
  2012-07-20 12:22:46,782 INFO sqlalchemy.engine.base.Engine SELECT sales_rep.brand_id AS sales_rep_brand_id, sales_rep.user_id AS sales_rep_user_id, sales_rep.commission_pct AS sales_rep_commission_pct
  FROM sales_rep
  WHERE ? = sales_rep.user_id
  2012-07-20 12:22:46,782 INFO sqlalchemy.engine.base.Engine (2,)
  >>> b.users
  [<__main__.User object at 0x87d7b6c>]
  >>> b.sales_reps
  [<__main__.SalesRep object at 0x87d7c4c>]
  >>> b.sales_reps[0].commission_pct
  0
  >>> session.flush()
  2012-07-20 12:23:14,215 INFO sqlalchemy.engine.base.Engine INSERT INTO sales_rep (brand_id, user_id, commission_pct) VALUES (?, ?, ?)
  2012-07-20 12:23:14,215 INFO sqlalchemy.engine.base.Engine (2, 2, 0)
- 更复杂的想法给销售人员一个 10%的提成:

- Brand.users=association_proxy(
  ‘sales_reps’, ‘user’,
  creator=lambda u:SalesRep(user=u, commission_pct=10))
- 假设我们想要的品牌属性是一个附带 User 和佣金 commission_pct 的字典:
- from sqlalchemy.orm.collections import
  attribute_mapped_collection
  >>> from sqlalchemy.orm.collections import
  attribute_mapped_collection
  >>> reps_by_user_class=attribute_mapped_collection(‘user’)
  >>> clear_mappers()
  >>> mapper(Brand, brand_table, properties=dict(
  … sales_reps_by_user=relation(
  … SalesRep, backref=’ brand’ ,
  … collection_class=reps_by_user_class)))
  <Mapper at 0x862c5ec; Brand>
- >>> Brand.commissions=association_proxy(
  … ‘sales_reps_by_user’, ‘commission_pct’,
  … creator=lambda key,value: SalesRep(user=key,
  commission_pct=value))
  >>> mapper(User, user_table, properties=dict(
  … sales_rep=relation(SalesRep, backref=’ user’ ,
  uselist=False)))
  <Mapper at 0x8764b2c; User>
  >>> mapper(SalesRep, sales_rep_table)
  <Mapper at 0x87bb4cc; SalesRep>
  >>> b = session.query(Brand).get(1)
  >>> u = session.query(User).get(1)
  >>> b.commissions[u] = 20
  >>> session.bind.echo = False
  >>> session.flush()
  >>> b = session.query(Brand).get(1)
  >>> u = session.query(User).get(1)
  >>> u.user_name
  u’ dongwm’
  >>> print b.commissions[u]
  20
  >>> print b.sales_reps_by_user[u] #代理和原来的关系是自动同步的
  <__main__.SalesRep object at 0x87e3dcc>
  >>> print b.sales_reps_by_user[u].commission_pct
  20