# **Chapter 1 Object Relational Tutorial**

The SQLAlchemy Object Relational Mapper presents a method of associating user-defined Python classes with database tables, and instances of those classes (objects) with rows in their corresponding tables. It includes a system that transparently synchronizes all changes in state between objects and their related rows, called a [unit of work](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-unit-of-work), as well as a system for expressing database queries in terms of the user defined classes and their defined relationships between each other.

SQLAlchemy对象关系映射器提供了一种将用户定义的Python类与数据库表相关联的方法，以及在对应表中具有行的那些类(对象)的实例。 它包括一个系统，可透明地同步对象及其相关行之间的状态的所有更改，称为工作单元，以及根据用户定义的类及其彼此之间定义的关系来表达数据库查询的系统。

The ORM is in contrast to the SQLAlchemy Expression Language, upon which the ORM is constructed. Whereas the SQL Expression Language, introduced in [SQL Expression Language Tutorial](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html), presents a system of representing the primitive constructs of the relational database directly without opinion, the ORM presents a high level and abstracted pattern of usage, which itself is an example of applied usage of the Expression Language.

ORM与构建ORM的SQLAlchemy表达式语言形成对比。而在SQL表达语言教程中引入的SQL表达式语言提供了一种直接表示关系数据库的原始构造的系统，而ORM呈现出高级和抽象的使用模式，本身就是应用使用的一个示例表达语言。

While there is overlap among the usage patterns of the ORM and the Expression Language, the similarities are more superficial than they may at first appear. One approaches the structure and content of data from the perspective of a user-defined [domain model](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-domain-model) which is transparently persisted and refreshed from its underlying storage model. The other approaches it from the perspective of literal schema and SQL expression representations which are explicitly composed into messages consumed individually by the database.

虽然ORM和表达语言的使用模式之间存在重叠，但相似之处比起初显示的更为肤浅。从用户定义的域模型的角度来看，数据的结构和内容从透明地持久化并从基础存储模型中刷新。另一种从文字模式和SQL表达式表达的角度来看待，这些表达式被明确地组合成数据库单独消费的消息。

A successful application may be constructed using the Object Relational Mapper exclusively. In advanced situations, an application constructed with the ORM may make occasional usage of the Expression Language directly in certain areas where specific database interactions are required.

可以使用对象关系映射器专门构建成功的应用程序。在高级情况下，使用ORM构建的应用程序可能会在需要特定数据库交互的某些区域中偶尔使用表达式语言。

The following tutorial is in doctest format, meaning each >>> line represents something you can type at a Python command prompt, and the following text represents the expected return value.

以下教程采用doctest格式，这意味着每行>>>行表示可以在Python命令提示符下键入的内容，下面的文本表示预期的返回值。

**1.1 Version Check**

A quick check to verify that we are on at least ****version 1.1**** of SQLAlchemy:

**>>> import sqlalchemy**

**>>>** sqlalchemy.\_\_version\_\_

'1.1.9'

## 1.2 Connecting

For this tutorial we will use an in-memory-only SQLite database. To connect we use  [create\_engine()](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "sqlalchemy.create_engine" \o "sqlalchemy.create_engine):

对于本教程，我们将使用仅内存的SQLite数据库。 要连接我们使用create\_engine()：

**>>> from** **sqlalchemy** **import** create\_engine

**>>>** engine = create\_engine('sqlite:///:memory:', echo=**True**)

The echo flag is a shortcut to setting up SQLAlchemy logging, which is accomplished via Python's standard logging module. With it enabled, we'll see all the generated SQL produced. If you are working through this tutorial and want less output generated, set it to False. This tutorial will format the SQL behind a popup window so it doesn't get in our way; just click the "SQL" links to see what's being generated.

echo标志是设置SQLAlchemy日志的快捷方式，它通过Python的标准logging模块完成。 启用它后，我们将看到生成的所有SQL。 如果您正在完成本教程并希望生成较少的输出，请将其设置为False。 本教程将格式化弹出窗口后面的SQL， 将不会在浏览中出现; 只需点击"SQL"链接即可查看正在生成的内容。

The return value of [create\_engine()](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "sqlalchemy.create_engine" \o "sqlalchemy.create_engine) is an instance of [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine), and it represents the core interface to the database, adapted through a dialect that handles the details of the database and [DBAPI](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-dbapi) in use. In this case the SQLite dialect will interpret instructions to the Python built-in sqlite3 module.

[create\_engine()](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "sqlalchemy.create_engine" \o "sqlalchemy.create_engine)的返回值是[Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine)的一个实例，它表示数据库的核心接口，通过方言进行调整，该方法处理数据库的详细信息和正在使用的DBAPI。 在这种情况下，SQLite方言将解释Python内置sqlite3模块的指令。

**Lazy Connecting/**懒连接

The [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine), when first returned by [create\_engine()](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "sqlalchemy.create_engine" \o "sqlalchemy.create_engine), has not actually tried to connect to the database yet; that happens only the first time it is asked to perform a task against the database.

引擎首次由create\_engine()返回时，尚未实际尝试连接到数据库; 这只会在第一次被要求对数据库执行任务时发生。

The first time a method like [Engine.execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.execute" \o "sqlalchemy.engine.Engine.execute) or [Engine.connect()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.connect" \o "sqlalchemy.engine.Engine.connect) is called, the [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) establishes a real [DBAPI](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-dbapi) connection to the database, which is then used to emit the SQL. When using the ORM, we typically don't use the [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) directly once created; instead, it's used behind the scenes by the ORM as we'll see shortly.

首次调用[Engine.execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.execute" \o "sqlalchemy.engine.Engine.execute)或[Engine.connect()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.connect" \o "sqlalchemy.engine.Engine.connect)方法时，[Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine)会建立与数据库的真实DBAPI连接，然后将其用于发出SQL。 当使用ORM时，我们通常不会直接在已创建就是用[Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine); 相反，它在ORM的幕后使用，我们将很快看到。

**See also**

[Database Urls](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "database-urls) - includes examples of [create\_engine()](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "sqlalchemy.create_engine" \o "sqlalchemy.create_engine) connecting to several kinds of databases with links to more information.

[Database Urls](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "database-urls) - 包含[create\_engine()](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "sqlalchemy.create_engine" \o "sqlalchemy.create_engine)连接到几种数据库的例子，链接到更多的信息。

1.3 Declare a Mapping声明一个映射

When using the ORM, the configurational process starts by describing the database tables we'll be dealing with, and then by defining our own classes which will be mapped to those tables. In modern SQLAlchemy, these two tasks are usually performed together, using a system known as [Declarative](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/index.html), which allows us to create classes that include directives to describe the actual database table they will be mapped to.

当使用ORM时，配置过程首先描述我们将要处理的数据库表，然后通过定义我们自己的类来映射到这些表。 在现代的SQLAlchemy中，这两个任务通常使用一个称为Declarative的系统一起执行，这个系统允许我们创建一些包含指令的类来描述他们将映射到的实际数据库表。

Classes mapped using the Declarative system are defined in terms of a base class which maintains a catalog of classes and tables relative to that base - this is known as the ****declarative base class****. Our application will usually have just one instance of this base in a commonly imported module. We create the base class using the [declarative\_base()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.declarative_base" \o "sqlalchemy.ext.declarative.declarative_base) function, as follows:

使用声明系统映射的类是根据一个基类定义的，该基类维护相对于该基类的类和表的目录 - 这被称为声明性基类。 我们的应用程序通常只有一个这个基础的一个实例在通常导入的模块中。 我们使用declarative\_base()函数创建基类，如下所示：

**>>> from** **sqlalchemy.ext.declarative** **import** declarative\_base

**>>>** Base = declarative\_base()

Now that we have a "base", we can define any number of mapped classes in terms of it. We will start with just a single table called users, which will store records for the end-users using our application. A new class called User will be the class to which we map this table. Within the class, we define details about the table to which we'll be mapping, primarily the table name, and names and datatypes of columns:

现在我们有一个"base"，我们可以根据它定义任意数量的映射类。 我们将从一个名为users的表开始，它将使用我们的应用程序存储最终用户的记录。 一个名为User的新类将是我们映射这个表的类。 在类中，我们定义了我们要映射的表的详细信息，主要是表名，以及列的名称和数据类型：

**>>> from** **sqlalchemy** **import** Column, Integer, String

**>>> class** **User**(Base):

**...**  \_\_tablename\_\_ = 'users'

**......**  id = Column(Integer, primary\_key=**True**)

**...**  name = Column(String)

**...**  fullname = Column(String)

**...**  password = Column(String)

**......**  **def** \_\_repr\_\_(self):

**...**  **return** "<User(name='*%s*', fullname='*%s*', password='*%s*')>" % (

**...**  self.name, self.fullname, self.password)

**Tip/提示**

The User class defines a \_\_repr\_\_() method, but note that is ****optional****; we only implement it in this tutorial so that our examples show nicely formatted User objects.

User类定义了\_\_repr\_\_()方法，但注意是可选的; 我们只在本教程中实现它，以便我们的示例显示格式很好的用户对象。

A class using Declarative at a minimum needs a \_\_tablename\_\_ attribute, and at least one [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) which is part of a primary key [[1]](http://docs.sqlalchemy.org/en/rel_1_1/orm/tutorial.html" \l "id2). SQLAlchemy never makes any assumptions by itself about the table to which a class refers, including that it has no built-in conventions for names, datatypes, or constraints. But this doesn't mean boilerplate is required; instead, you're encouraged to create your own automated conventions using helper functions and mixin classes, which is described in detail at [Mixin and Custom Base Classes](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/mixins.html" \l "declarative-mixins).

一个使用Declarative的类至少需要一个\_\_tablename\_\_属性，至少有一个[Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column)是主键[1]的一部分。 SQLAlchemy本身不会对类引用的表进行任何假设，包括它没有用于名称，数据类型或约束的内置约定。但这并不意味着需要样板;相反，鼓励您使用帮助函数和混合类创建自己的自动约定，这在Mixin和自定义基类中有详细描述。

When our class is constructed, Declarative replaces all the [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects with special Python accessors known as [descriptors](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-descriptors); this is a process known as [instrumentation](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-instrumentation). The "instrumented" mapped class will provide us with the means to refer to our table in a SQL context as well as to persist and load the values of columns from the database.

当我们的类被构造时，Declarative用特殊的Python访问器(称为描述符)替换所有的[Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column)对象;这是一个称为[instrumentation](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-instrumentation)的过程。 "instrumented"映射类将为我们提供在SQL上下文中引用我们表的方法以及从数据库中保留和加载列的值。

Outside of what the mapping process does to our class, the class remains otherwise mostly a normal Python class, to which we can define any number of ordinary attributes and methods needed by our application.

除了映射过程对我们的类之外，该类还是大多数是普通的Python类，我们可以定义任何数量的普通属性和我们的应用程序需要的方法。

|  |  |
| --- | --- |
| [[1]](http://docs.sqlalchemy.org/en/rel_1_1/orm/tutorial.html" \l "id1) | For information on why a primary key is required, see [How do I map a table that has no primary key?](http://docs.sqlalchemy.org/en/rel_1_1/faq/ormconfiguration.html" \l "faq-mapper-primary-key). |

1.4 Create a Schema

With our User class constructed via the Declarative system, we have defined information about our table, known as table metadata. The object used by SQLAlchemy to represent this information for a specific table is called the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object, and here Declarative has made one for us. We can see this object by inspecting the \_\_table\_\_ attribute:

通过使用通过Declarative系统构建的User类，我们定义了有关表的信息，称为表元数据。 SQLAlchemy用于表示特定表的这个信息的对象称为[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)对象，这里Declarative已经为我们提供了一个。 我们可以通过检查\_\_table\_\_属性来看到这个对象：

**>>>** User.\_\_table\_\_ Table('users', MetaData(bind=None), Column('id', Integer(), table=<users>, primary\_key=True, nullable=False), Column('name', String(), table=<users>), Column('fullname', String(), table=<users>),

Column('password', String(), table=<users>), schema=None

)

**Classical Mappings**经典映射

The Declarative system, though highly recommended, is not required in order to use SQLAlchemy's ORM. Outside of Declarative, any plain Python class can be mapped to any [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) using the [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper) function directly; this less common usage is described at [Classical Mappings](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_styles.html" \l "classical-mapping).

为了使用SQLAlchemy的ORM，声明系统虽然被强烈推荐，但不是必需的。 在声明式之外，任何普通Python类都可以直接映射到任何使用mapper()函数的表; 这种较不常见的用法在古典映射中描述。

When we declared our class, Declarative used a Python metaclass in order to perform additional activities once the class declaration was complete; within this phase, it then created a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)object according to our specifications, and associated it with the class by constructing a [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) object. This object is a behind-the-scenes object we normally don't need to deal with directly (though it can provide plenty of information about our mapping when we need it).

当我们宣布我们的类时，Declarative使用一个Python metaclass为了执行额外的活动，一旦类声明完成;在此阶段之前，它根据我们的规范创建了一个Table对象，并通过构造一个Mapper对象与该类相关联。这个对象是一个幕后的对象，我们通常不需要直接处理(尽管在需要时可以提供大量有关我们的映射的信息)。

The [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object is a member of a larger collection known as [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData). When using Declarative, this object is available using the .metadata attribute of our declarative base class.

[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) 对象是一个称为[MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData)的较大集合的成员。使用Declarative时，该对象可以使用我们的声明性基类的.metadata属性。

The [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) is a registry which includes the ability to emit a limited set of schema generation commands to the database. As our SQLite database does not actually have a users table present, we can use [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) to issue CREATE TABLE statements to the database for all tables that don't yet exist. Below, we call the [MetaData.create\_all()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData.create_all" \o "sqlalchemy.schema.MetaData.create_all) method, passing in our [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) as a source of database connectivity. We will see that special commands are first emitted to check for the presence of the users table, and following that the actual CREATE TABLE statement:

MetaData是一个注册表，其中包括向数据库发出有限的模式生成命令集的能力。由于我们的SQLite数据库实际上没有users表，所以我们可以使用[MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData)向所有不存在的表发出CREATE TABLE语句到数据库。下面我们调用[MetaData.create\_all()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData.create_all" \o "sqlalchemy.schema.MetaData.create_all)方法，将我们的引擎作为数据库连接的源头。我们将看到，首先发出特殊命令来检查用户表的存在，并遵循实际的CREATE TABLE语句：

>>> Base.metadata.create\_all(engine)

SELECT

...

PRAGMA table\_info("users")()CREATE TABLE users (

id INTEGER NOT NULL, name VARCHAR,

fullname VARCHAR,

password VARCHAR,

PRIMARY KEY (id))

()

COMMIT

**Minimal Table Descriptions vs. Full Descriptions**

最小表格描述与全面描述

Users familiar with the syntax of CREATE TABLE may notice that the VARCHAR columns were generated without a length; on SQLite and PostgreSQL, this is a valid datatype, but on others, it's not allowed. So if running this tutorial on one of those databases, and you wish to use SQLAlchemy to issue CREATE TABLE, a "length" may be provided to the [String](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.String" \o "sqlalchemy.types.String)type as below:

熟悉CREATE TABLE语法的用户可能会注意到VARCHAR列生成时没有长度; 在SQLite和PostgreSQL上，这是一个有效的数据类型，但在其他数据库，这是不允许的。 因此，如果在其中一个数据库上运行此教程，并且希望使用SQLAlchemy发布CREATE TABLE，则可以向[String](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.String" \o "sqlalchemy.types.String)类型提供"length"，如下所示：

Column(String(50))

The length field on [String](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.String" \o "sqlalchemy.types.String), as well as similar precision/scale fields available on [Integer](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Integer" \o "sqlalchemy.types.Integer), [Numeric](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Numeric" \o "sqlalchemy.types.Numeric), etc. are not referenced by SQLAlchemy other than when creating tables.

[String](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.String" \o "sqlalchemy.types.String)上的长度字段以及[Integer](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Integer" \o "sqlalchemy.types.Integer)，[Numeric](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Numeric" \o "sqlalchemy.types.Numeric)等可用的类似精度/比例字段除了创建表之外，不被SQLAlchemy引用。

Additionally, Firebird and Oracle require sequences to generate new primary key identifiers, and SQLAlchemy doesn't generate or assume these without being instructed. For that, you use the [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) construct:

此外，Firebird和Oracle需要序列来生成新的主键标识符，SQLAlchemy不会在没有指示的情况下生成或假设这些标识符。 为此，您使用[Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence)结构：

**from** **sqlalchemy** **import** Sequence

Column(Integer, Sequence('user\_id\_seq'), primary\_key=**True**)

A full, foolproof [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) generated via our declarative mapping is therefore:

**class** **User**(Base):

\_\_tablename\_\_ = 'users'

id = Column(Integer, Sequence('user\_id\_seq'), primary\_key=**True**)

name = Column(String(50))

fullname = Column(String(50))

password = Column(String(12))

**def** \_\_repr\_\_(self):

**return** "<User(name='*%s*', fullname='*%s*', password='*%s*')>" % (

self.name, self.fullname, self.password)

We include this more verbose table definition separately to highlight the difference between a minimal construct geared primarily towards in-Python usage only, versus one that will be used to emit CREATE TABLE statements on a particular set of backends with more stringent requirements.

我们分别包括这个更详细的表定义，以突出显示主要面向Python内部使用的最小结构之间的差异，而不是用于在具有更严格要求的特定后端集合上发布CREATE TABLE语句。

1.5 Create an Instance of the Mapped Class

With mappings complete, let's now create and inspect a User object:

**>>>** ed\_user = User(name='ed', fullname='Ed Jones', password='edspassword')

**>>>** ed\_user.name

'ed'

**>>>** ed\_user.password

'edspassword'

**>>>** str(ed\_user.id)

'None'

**the**\_\_init\_\_()**method**

Our User class, as defined using the Declarative system, has been provided with a constructor (e.g. \_\_init\_\_() method) which automatically accepts keyword names that match the columns we've mapped. We are free to define any explicit \_\_init\_\_() method we prefer on our class, which will override the default method provided by Declarative.

我们的User类(使用声明系统定义)已经提供了一个构造函数(例如\_\_init\_\_()方法)，它可以自动接受与我们映射的列匹配的关键字名称。 我们可以自由地定义我们喜欢的任何显式的\_\_init\_\_()方法，这将覆盖由Declarative提供的默认方法。

Even though we didn't specify it in the constructor, the id attribute still produces a value of None when we access it (as opposed to Python's usual behavior of raising AttributeError for an undefined attribute). SQLAlchemy's [instrumentation](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-instrumentation) normally produces this default value for column-mapped attributes when first accessed. For those attributes where we've actually assigned a value, the instrumentation system is tracking those assignments for use within an eventual INSERT statement to be emitted to the database.

即使我们没有在构造函数中指定它，但是当我们访问它时，id属性仍然产生一个值None(而不是Python通常为未定义的属性提升AttributeError的行为)。 SQLAlchemy的[instrumentation](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-instrumentation) 通常会在首次访问时为列映射属性生成此默认值。 对于我们实际分配了一个值的属性，仪器系统正在跟踪这些分配，以便在最终的INSERT语句中使用，以将其发送到数据库。

1.6 Creating a Session

We're now ready to start talking to the database. The ORM's "handle" to the database is the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session). When we first set up the application, at the same level as our [create\_engine()](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "sqlalchemy.create_engine" \o "sqlalchemy.create_engine) statement, we define a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) class which will serve as a factory for new [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) objects:

我们现在可以开始与数据库交谈了。 数据库的ORM的“句柄”是Session。 当我们首先将应用程序设置在与我们的create\_engine()语句相同的级别时，我们定义一个Session类，它将作为新的Session对象的工厂：

**>>> from** **sqlalchemy.orm** **import** sessionmaker

**>>>** Session = sessionmaker(bind=engine)

In the case where your application does not yet have an [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) when you define your module-level objects, just set it up like this:

如果您的应用程序在定义模块级对象时还没有引擎，则只需将其设置为：

**>>>** Session = sessionmaker()

Later, when you create your engine with [create\_engine()](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "sqlalchemy.create_engine" \o "sqlalchemy.create_engine), connect it to the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) using [configure()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker.configure" \o "sqlalchemy.orm.session.sessionmaker.configure):

稍后，当您使用create\_engine()创建引擎时，请使用configure()将其连接到会话：

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

**Session Lifecycle Patterns**

会话生命周期模式

The question of when to make a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) depends a lot on what kind of application is being built. Keep in mind, the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) is just a workspace for your objects, local to a particular database connection - if you think of an application thread as a guest at a dinner party, the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) is the guest's plate and the objects it holds are the food (and the database…the kitchen?)! More on this topic available at [When do I construct a Session, when do I commit it, and when do I close it?](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_basics.html" \l "session-faq-whentocreate).

何时进行会话的问题在很大程度上取决于正在构建什么样的应用程序。 请记住，会话只是一个用于您的对象的工作区，特定数据库连接的本地区 - 如果您在晚宴上将应用程序线程视为客人，则会话是客人的板，其所持有的对象是食物 (和数据库...厨房？)！ 有关此主题的更多信息何时构建会话，何时提交会话，什么时候关闭它？

This custom-made [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) class will create new [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) objects which are bound to our database. Other transactional characteristics may be defined when calling [sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker) as well; these are described in a later chapter. Then, whenever you need to have a conversation with the database, you instantiate a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session):

这个定制的Session类将创建绑定到我们的数据库的新Session对象。 当调用sessionmaker时也可以定义其他事务特征; 这些在后面的章节中进行了描述。 然后，当您需要与数据库进行对话时，您将实例化一个会话：

**>>>** session = Session()

The above [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) is associated with our SQLite-enabled [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine), but it hasn't opened any connections yet. When it's first used, it retrieves a connection from a pool of connections maintained by the [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine), and holds onto it until we commit all changes and/or close the session object.

上述会话与我们的启用SQLite-enabled [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine)的[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)相关联，但尚未打开任何连接。 当它首次使用时，它从引擎维护的连接池中检索一个连接，并保持它，直到我们提交所有更改和/或关闭会话对象。

1.7 Adding and Updating Objects

To persist our User object, we [add()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.add" \o "sqlalchemy.orm.session.Session.add) it to our [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session):

要持久我们的User对象，我们将它[add()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.add" \o "sqlalchemy.orm.session.Session.add)到我们的[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)：

**>>>** ed\_user = User(name='ed', fullname='Ed Jones', password='edspassword')

**>>>** session.add(ed\_user)

At this point, we say that the instance is ****pending****; no SQL has yet been issued and the object is not yet represented by a row in the database. The [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) will issue the SQL to persist Ed Jones as soon as is needed, using a process known as a ****flush****. If we query the database for Ed Jones, all pending information will first be flushed, and the query is issued immediately thereafter.

在这一点上，我们说这个实例正在是挂起; 尚未发出SQL，并且该对象尚未由数据库中的一行表示。 [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)将发出SQL，以便在需要的时候使用称为flush的过程来保持Ed Jones。 如果我们为Ed Jones查询数据库，所有待处理的信息将首先被刷新，之后立即发出查询。

For example, below we create a new [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) object which loads instances of User. We "filter by" the name attribute of ed, and indicate that we'd like only the first result in the full list of rows. A User instance is returned which is equivalent to that which we've added:

例如，下面我们创建一个新的[Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query)对象，它加载User的实例。 我们按照ed的name属性过滤，并指出我们只希望在列表的完整列表中显示第一个结果。 返回一个与实例相同的User实例：

>>> our\_user = session.query(User).filter\_by(name='ed').first()

BEGIN (implicit)

INSERT INTO users (name, fullname, password) VALUES (?, ?, ?)

('ed', 'Ed Jones', 'edspassword')

SELECT users.id AS users\_id,

users.name AS users\_name,

users.fullname AS users\_fullname,

users.password AS users\_password

FROM users

WHERE users.name = ?

LIMIT ? OFFSET ?

('ed', 1, 0)

>>> our\_user

<User(name='ed', fullname='Ed Jones', password='edspassword')>

In fact, the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) has identified that the row returned is the ****same**** row as one already represented within its internal map of objects, so we actually got back the identical instance as that which we just added:

事实上，会话已经确定返回的行是与其内部对象映射中已经表示的行相同的行，所以我们实际上得到了刚刚添加的实例：

**>>>** ed\_user **is** our\_user

True

The ORM concept at work here is known as an [identity map](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-identity-map) and ensures that all operations upon a particular row within a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) operate upon the same set of data. Once an object with a particular primary key is present in the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session), all SQL queries on that [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) will always return the same Python object for that particular primary key; it also will raise an error if an attempt is made to place a second, already-persisted object with the same primary key within the session.

这里的ORM概念被称为身份映射，并确保[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)内的特定行上的所有操作都在同一组数据上运行。 一旦具有特定主键的对象存在于[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)中，则该[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)上的所有SQL查询将始终为该特定主键返回相同的Python对象; 如果尝试在会话中放置具有相同主键的第二个已经持久化的对象，它也会引发错误。

We can add more User objects at once using [add\_all()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.add_all" \o "sqlalchemy.orm.session.Session.add_all):

我们可以使用add\_all()一次添加更多的User对象：

>>> session.add\_all([

... User(name='wendy', fullname='Wendy Williams', password='foobar'),

... User(name='mary', fullname='Mary Contrary', password='xxg527'),

... User(name='fred', fullname='Fred Flinstone', password='blah')])

Also, we've decided the password for Ed isn't too secure, so lets change it:

此外，我们已经决定了Ed的密码不是太安全，所以可以改变它：

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

The [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) is paying attention. It knows, for example, that Ed Jones has been modified:

[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)正在关注。 例如，Ed Jones已经被修改了：

>>> session.dirty

IdentitySet([<User(name='ed', fullname='Ed Jones', password='f8s7ccs')>])

and that three new User objects are pending:

并且三个新的用户对象正在等待：

>>> session.new

IdentitySet([

<User(name='wendy', fullname='Wendy Williams', password='foobar')>,

<User(name='mary', fullname='Mary Contrary', password='xxg527')>,

<User(name='fred', fullname='Fred Flinstone', password='blah')>

])

We tell the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) that we'd like to issue all remaining changes to the database and commit the transaction, which has been in progress throughout. We do this via [commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.commit" \o "sqlalchemy.orm.session.Session.commit). The [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) emits the UPDATE statement for the password change on "ed", as well as INSERT statements for the three new User objects we've added:

我们告诉[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)，我们要发布所有剩余的数据库更改并提交交易，这一切都在进行中。 我们通过[commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.commit" \o "sqlalchemy.orm.session.Session.commit)来实现。 [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)发出用于"ed"上的密码更改的UPDATE语句，以及我们添加的三个新User对象的INSERT语句：

>>> session.commit()

[commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.commit" \o "sqlalchemy.orm.session.Session.commit) flushes the remaining changes to the database, and commits the transaction. The connection resources referenced by the session are now returned to the connection pool. Subsequent operations with this session will occur in a ****new**** transaction, which will again re-acquire connection resources when first needed.

[commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.commit" \o "sqlalchemy.orm.session.Session.commit)将剩余的更改刷新到数据库，并提交事务。 会话引用的连接资源现在返回到连接池。 此次会话的后续操作将发生在新的事务中，这将在首次需要时再次重新获取连接资源。

If we look at Ed's id attribute, which earlier was None, it now has a value:

如果我们看看Ed的id属性，它早先是None，它现在有一个值：

>>> ed\_user.id

BEGIN (implicit)

SELECT users.id AS users\_id,

users.name AS users\_name,

users.fullname AS users\_fullname,

users.password AS users\_password

FROM users

WHERE users.id = ?

(1,)

1

After the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) inserts new rows in the database, all newly generated identifiers and database-generated defaults become available on the instance, either immediately or via load-on-first-access. In this case, the entire row was re-loaded on access because a new transaction was begun after we issued [commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.commit" \o "sqlalchemy.orm.session.Session.commit). SQLAlchemy by default refreshes data from a previous transaction the first time it's accessed within a new transaction, so that the most recent state is available. The level of reloading is configurable as is described in [Using the Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session.html).

[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)在数据库中插入新行后，所有新生成的标识符和数据库生成的默认值将立即在实例上或通过首先访问加载。 在这种情况下，由于在发出[commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.commit" \o "sqlalchemy.orm.session.Session.commit)之后开始新的事务，所以整个行都被重新加载。 SQLAlchemy默认情况下刷新来自先前事务的数据，首次在新事务中访问时，以便最近的状态可用。 重新加载的级别是可配置的，如"使用会话"中所述。

**Session Object States/会话对象状态**

As our User object moved from being outside the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session), to inside the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) without a primary key, to actually being inserted, it moved between three out of four available "object states" - ****transient****, ****pending****, and ****persistent****. Being aware of these states and what they mean is always a good idea - be sure to read [Quickie Intro to Object States](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_state_management.html" \l "session-object-states) for a quick overview.

当我们的User对象没有主键盘从Session外部移动到Session内部时，实际上被插入，它在四个可用的"对象状态"中的三个之间移动 - 瞬态，挂起和持久。 意识到这些状态和它们的意思总是会有用 - 请务必阅读[Quickie Intro to Object States](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_state_management.html" \l "session-object-states) ，以便快速了解。

## 1.8 Rolling Back/回滚

Since the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) works within a transaction, we can roll back changes made too. Let's make two changes that we'll revert; ed\_user's user name gets set to Edwardo:

由于[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)在事务中工作，我们可以回滚更改。 让我们进行两个我们将要恢复的变化; ed\_user的用户名设置为Edwardo：

>>> ed\_user.name = 'Edwardo'

and we'll add another erroneous user, fake\_user:

我们将添加另一个错误的用户，fake\_user：

>>> fake\_user = User(name='fakeuser', fullname='Invalid', password='12345')

>>> session.add(fake\_user)

Querying the session, we can see that they're flushed into the current transaction:

查询会话，我们可以看到它们被刷新到当前事务中：

>>> session.query(User).filter(User.name.in\_(['Edwardo', 'fakeuser'])).all()

[<User(name='Edwardo', fullname='Ed Jones', password='f8s7ccs')>, <User(name='fakeuser', fullname='Invalid', password='12345')>]

UPDATE users SET name=? WHERE users.id = ?

('Edwardo', 1)

INSERT INTO users (name, fullname, password) VALUES (?, ?, ?)

('fakeuser', 'Invalid', '12345')

SELECT users.id AS users\_id,

users.name AS users\_name,

users.fullname AS users\_fullname,

users.password AS users\_password

FROM users

WHERE users.name IN (?, ?)

('Edwardo', 'fakeuser')

Rolling back, we can see that ed\_user's name is back to ed, and fake\_user has been kicked out of the session:

回滚后，我们可以看到ed\_user的名字已经回到ed，而fake\_user已经被踢出了会话：

>>> session.rollback()

ROLLBACK

>>> ed\_user.name

u'ed'

>>> fake\_user **in** session

False

BEGIN (implicit)

SELECT users.id AS users\_id,

users.name AS users\_name,

users.fullname AS users\_fullname,

users.password AS users\_password

FROM users

WHERE users.id = ?

(1,)

Issuing a SELECT illustrates the changes made to the database:

发出一个SELECT说明了对数据库所做的更改：

>>> session.query(User).filter(User.name.in\_(['ed', 'fakeuser'])).all()

SELECT users.id AS users\_id,

users.name AS users\_name,

users.fullname AS users\_fullname,

users.password AS users\_password

FROM users

WHERE users.name IN (?, ?)

('ed', 'fakeuser')

[<User(name='ed', fullname='Ed Jones', password='f8s7ccs')>]

## 1.9 Querying/查询

A [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) object is created using the [query()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.query" \o "sqlalchemy.orm.session.Session.query) method on [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session). This function takes a variable number of arguments, which can be any combination of classes and class-instrumented descriptors. Below, we indicate a [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) which loads User instances. When evaluated in an iterative context, the list of User objects present is returned:

使用[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)上的[query()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.query" \o "sqlalchemy.orm.session.Session.query)方法创建[Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query)对象。 该函数接受可变数量的参数，它们可以是类和类仪表化描述符的任意组合。 下面我们指出一个加载User实例的[Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query)。 当在迭代上下文中进行评估时，返回出现的User对象列表：

>>> **for** instance **in** session.query(User).order\_by(User.id):

... **print**(instance.name, instance.fullname)

SELECT users.id AS users\_id,

users.name AS users\_name,

users.fullname AS users\_fullname,

users.password AS users\_password

FROM users ORDER BY users.id

()

ed Ed Jones

wendy Wendy Williams

mary Mary Contrary

fred Fred Flinstone

The [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) also accepts ORM-instrumented descriptors as arguments. Any time multiple class entities or column-based entities are expressed as arguments to the [query()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.query" \o "sqlalchemy.orm.session.Session.query) function, the return result is expressed as tuples:

[Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query)还接受ORM-instrumented的描述符作为参数。 任何时候，多个类实体或基于列的实体都被表示为[query()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.query" \o "sqlalchemy.orm.session.Session.query)函数的参数，返回结果表示为元组：

>>> **for** name, fullname **in** session.query(User.name, User.fullname):

... **print**(name, fullname)

SELECT users.name AS users\_name,

users.fullname AS users\_fullname

FROM users

()

ed Ed Jones

wendy Wendy Williams

mary Mary Contrary

fred Fred Flinstone

The tuples returned by [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) are *named* tuples, supplied by the [KeyedTuple](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.util.KeyedTuple" \o "sqlalchemy.util.KeyedTuple) class, and can be treated much like an ordinary Python object. The names are the same as the attribute's name for an attribute, and the class name for a class:

由[Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query)返回的元组被命名为元组，由[KeyedTuple](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.util.KeyedTuple" \o "sqlalchemy.util.KeyedTuple)类提供，并且可以像普通的Python对象那样对待。 名称与属性的属性名称以及类的类名称相同：

>>> **for** row **in** session.query(User, User.name).all():

... **print**(row.User, row.name)

SELECT users.id AS users\_id,

users.name AS users\_name,

users.fullname AS users\_fullname,

users.password AS users\_password

FROM users

()

<User(name='ed', fullname='Ed Jones', password='f8s7ccs')>

ed

<User(name='wendy', fullname='Wendy Williams', password='foobar')>

wendy

<User(name='mary', fullname='Mary Contrary', password='xxg527')>

mary

<User(name='fred', fullname='Fred Flinstone', password='blah')>

fred

You can control the names of individual column expressions using the [label()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement.label" \o "sqlalchemy.sql.expression.ColumnElement.label) construct, which is available from any [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement)-derived object, as well as any class attribute which is mapped to one (such as User.name):

您可以使用可从任何[ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement)派生对象获取的[label()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement.label" \o "sqlalchemy.sql.expression.ColumnElement.label)结构以及映射到其中的任何类属性(例如User.name)来控制各个列表达式的名称：

>>> **for** row **in** session.query(User.name.label('name\_label')).all():

... **print**(row.name\_label)

SELECT users.name AS name\_label

FROM users

()

ed

wendy

mary

fred

The name given to a full entity such as User, assuming that multiple entities are present in the call to [query()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.query" \o "sqlalchemy.orm.session.Session.query), can be controlled using [aliased()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.aliased" \o "sqlalchemy.orm.aliased) :

假设在对[query()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.query" \o "sqlalchemy.orm.session.Session.query)的调用中存在多个实体，给予诸如User的完整实体的名称，可以使用aliased()进行控制：

>>> **from** **sqlalchemy.orm** **import** aliased

>>> user\_alias = aliased(User, name='user\_alias')

>>> **for** row **in** session.query(user\_alias, user\_alias.name).all():

... **print**(row.user\_alias)

SELECT user\_alias.id AS user\_alias\_id,

user\_alias.name AS user\_alias\_name,

user\_alias.fullname AS user\_alias\_fullname,

user\_alias.password AS user\_alias\_password

FROM users AS user\_alias

()

<User(name='ed', fullname='Ed Jones', password='f8s7ccs')>

<User(name='wendy', fullname='Wendy Williams', password='foobar')>

<User(name='mary', fullname='Mary Contrary', password='xxg527')>

<User(name='fred', fullname='Fred Flinstone', password='blah')>

Basic operations with [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) include issuing LIMIT and OFFSET, most conveniently using Python array slices and typically in conjunction with ORDER BY:

[Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query)的基本操作包括发布LIMIT和OFFSET，最方便地使用Python数组切片，通常与ORDER BY结合使用：

>>> **for** u **in** session.query(User).order\_by(User.id)[1:3]:

... **print**(u)

SELECT users.id AS users\_id,

users.name AS users\_name,

users.fullname AS users\_fullname,

users.password AS users\_password

FROM users ORDER BY users.id

LIMIT ? OFFSET ?

(2, 1)

<User(name='wendy', fullname='Wendy Williams', password='foobar')>

<User(name='mary', fullname='Mary Contrary', password='xxg527')>

and filtering results, which is accomplished either with [filter\_by()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.filter_by" \o "sqlalchemy.orm.query.Query.filter_by), which uses keyword arguments:

和过滤结果，这可以通过使用关键字参数的[filter\_by()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.filter_by" \o "sqlalchemy.orm.query.Query.filter_by)来实现：

>>> **for** name, **in** session.query(User.name).\

... filter\_by(fullname='Ed Jones'):

... **print**(name)

SELECT users.name AS users\_name FROM users

WHERE users.fullname = ?

('Ed Jones',)

ed

…or [filter()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.filter" \o "sqlalchemy.orm.query.Query.filter), which uses more flexible SQL expression language constructs. These allow you to use regular Python operators with the class-level attributes on your mapped class:

...或[filter()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.filter" \o "sqlalchemy.orm.query.Query.filter)，它使用更灵活的SQL表达式语言构造。 这些允许您使用常规Python运算符与映射类上的类级属性：

>>> **for** name, **in** session.query(User.name).\

... filter(User.fullname=='Ed Jones'):

... **print**(name)

ed

SELECT users.name AS users\_name FROM users

WHERE users.fullname = ?

('Ed Jones',)

The [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) object is fully ****generative****, meaning that most method calls return a new [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) object upon which further criteria may be added. For example, to query for users named "ed" with a full name of "Ed Jones", you can call [filter()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.filter" \o "sqlalchemy.orm.query.Query.filter) twice, which joins criteria using AND:

[Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query)对象是完全可生成的，这意味着大多数方法调用返回一个新的[Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query)对象，在其上可以添加进一步的准则。 例如，要查询名为"ed"全名为"Ed Jones"的用户，您可以调用[filter()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.filter" \o "sqlalchemy.orm.query.Query.filter)两次，其中使用AND连接条件：

>>> **for** user **in** session.query(User).\

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

... filter(User.fullname=='Ed Jones'):

... **print**(user)

SELECT users.id AS users\_id,

users.name AS users\_name,

users.fullname AS users\_fullname,

users.password AS users\_password

FROM users

WHERE users.name = ? AND users.fullname = ?

('ed', 'Ed Jones')

<User(name='ed', fullname='Ed Jones', password='f8s7ccs')>

### 1.9.1 Common Filter Operators 常用筛选运算符

Here's a rundown of some of the most common operators used in [filter()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.filter" \o "sqlalchemy.orm.query.Query.filter):

以下是[filter()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.filter" \o "sqlalchemy.orm.query.Query.filter)中使用的一些最常用的运算符：

[equals](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.__eq__" \o "sqlalchemy.sql.operators.ColumnOperators.__eq__):

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

[not equals](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.__ne__" \o "sqlalchemy.sql.operators.ColumnOperators.__ne__):

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

[LIKE](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.like" \o "sqlalchemy.sql.operators.ColumnOperators.like):

query.filter(User.name.like('*%e*d%'))

**Note**

[ColumnOperators.like()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.like" \o "sqlalchemy.sql.operators.ColumnOperators.like) renders the LIKE operator, which is case insensitive on some backends, and case sensitive on others. For guaranteed case-insensitive comparisons, use [ColumnOperators.ilike()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.ilike" \o "sqlalchemy.sql.operators.ColumnOperators.ilike).

[ColumnOperators.like()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.like" \o "sqlalchemy.sql.operators.ColumnOperators.like)呈现LIKE运算符，这在一些后端是不区分大小写的，对其他的区分大小写。 对于保证不区分大小写的比较，请使用[ColumnOperators.ilike()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.ilike" \o "sqlalchemy.sql.operators.ColumnOperators.ilike)。

[ILIKE](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.ilike" \o "sqlalchemy.sql.operators.ColumnOperators.ilike) (case-insensitive LIKE):

query.filter(User.name.ilike('*%e*d%'))

**Note**

most backends don't support ILIKE directly. For those, the [ColumnOperators.ilike()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.ilike" \o "sqlalchemy.sql.operators.ColumnOperators.ilike) operator renders an expression combining LIKE with the LOWER SQL function applied to each operand.

大多数后端直接不支持ILIKE。 对于这些，[ColumnOperators.ilike()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.ilike" \o "sqlalchemy.sql.operators.ColumnOperators.ilike)运算符将表达式组合LIKE与应用于每个操作数的LOWER SQL函数。

[IN](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.in_" \o "sqlalchemy.sql.operators.ColumnOperators.in_):

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

*# works with query objects too:*query.filter(User.name.in\_(

session.query(User.name).filter(User.name.like('*%e*d%'))))

[NOT IN](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.notin_" \o "sqlalchemy.sql.operators.ColumnOperators.notin_):

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

[IS NULL](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.is_" \o "sqlalchemy.sql.operators.ColumnOperators.is_):

query.filter(User.name == **None**)

*# alternatively, if pep8/linters are a concern*

query.filter(User.name.is\_(**None**))

[IS NOT NULL](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.isnot" \o "sqlalchemy.sql.operators.ColumnOperators.isnot):

query.filter(User.name != **None**)

*# alternatively, if pep8/linters are a concern*

query.filter(User.name.isnot(**None**))

[AND](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.and_" \o "sqlalchemy.sql.expression.and_):

*# use and\_()*

**from** **sqlalchemy** **import** and\_

query.filter(and\_(User.name == 'ed', User.fullname == 'Ed Jones'))

*# or send multiple expressions to .filter()*

query.filter(User.name == 'ed', User.fullname == 'Ed Jones')

*# or chain multiple filter()/filter\_by() calls*

query.filter(User.name == 'ed').filter(User.fullname == 'Ed Jones')

**Note**

Make sure you use [and\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.and_" \o "sqlalchemy.sql.expression.and_) and ****not**** the Python and operator!

确保你使用了[and\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.and_" \o "sqlalchemy.sql.expression.and_)而不是Python and操作符！

[OR](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.or_" \o "sqlalchemy.sql.expression.or_):

**from** **sqlalchemy** **import** or\_

query.filter(or\_(User.name == 'ed', User.name == 'wendy'))

**Note**

Make sure you use [or\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.or_" \o "sqlalchemy.sql.expression.or_) and ****not**** the Python or operator!

确保你使用了[or\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.or_" \o "sqlalchemy.sql.expression.or_)而不是Python or操作符！

[MATCH](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.match" \o "sqlalchemy.sql.operators.ColumnOperators.match):

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

**Note**

[match()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.match" \o "sqlalchemy.sql.operators.ColumnOperators.match) uses a database-specific MATCH or CONTAINS function; its behavior will vary by backend and is not available on some backends such as SQLite.

[match()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.match" \o "sqlalchemy.sql.operators.ColumnOperators.match)使用数据库特定的MATCH或CONTAINS函数; 它的行为会因后端而有所不同，并且在某些后端（如SQLite）上不可用。

### 1.9.2 Returning Lists and Scalars 返回列表和标量

A number of methods on [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) immediately issue SQL and return a value containing loaded database results. Here's a brief tour:

[Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query)的一些方法立即发出SQL并返回一个包含加载的数据库结果的值。 这是一个简短的旅程：

[all()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.all" \o "sqlalchemy.orm.query.Query.all) returns a list:

[all()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.all" \o "sqlalchemy.orm.query.Query.all)返回一个列表：

>>> query = session.query(User).filter(User.name.like('*%e*d')).order\_by(User.id)

>>> query.all()

SELECT users.id AS users\_id,

users.name AS users\_name,

users.fullname AS users\_fullname,

users.password AS users\_password

FROM users

WHERE users.name LIKE ? ORDER BY users.id

('%ed',)

[<User(name='ed', fullname='Ed Jones', password='f8s7ccs')>,

<User(name='fred', fullname='Fred Flinstone', password='blah')>]

[first()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.first" \o "sqlalchemy.orm.query.Query.first) applies a limit of one and returns the first result as a scalar:

[first()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.first" \o "sqlalchemy.orm.query.Query.first)应用一个限制，并将第一个结果作为标量返回：

>>> query.first()

<User(name='ed', fullname='Ed Jones', password='f8s7ccs')>

SELECT users.id AS users\_id,

users.name AS users\_name,

users.fullname AS users\_fullname,

users.password AS users\_password

FROM users

WHERE users.name LIKE ? ORDER BY users.id

LIMIT ? OFFSET ?

('%ed', 1, 0)

[one()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.one" \o "sqlalchemy.orm.query.Query.one) fully fetches all rows, and if not exactly one object identity or composite row is present in the result, raises an error. With multiple rows found:

[one()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.one" \o "sqlalchemy.orm.query.Query.one)完全获取所有行，如果结果中不存在一个对象标识或复合行，则会引发错误。 找到多个行：

>>> user = query.one()

Traceback (most recent call last):

...

MultipleResultsFound: Multiple rows were found **for** one()

With no rows found:

>>> user = query.filter(User.id == 99).one()

Traceback (most recent call last):

...

NoResultFound: No row was found **for** one()

The [one()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.one" \o "sqlalchemy.orm.query.Query.one) method is great for systems that expect to handle "no items found" versus "multiple items found" differently; such as a RESTful web service, which may want to raise a "404 not found" when no results are found, but raise an application error when multiple results are found.

[one()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.one" \o "sqlalchemy.orm.query.Query.one)方法对于期望处理"没有找到的项目"与"找到的多个项目"的系统是不同的; 例如RESTful Web服务，当没有找到结果时，它可能想要引发"404未找到"，但是当发现多个结果时会引发应用程序错误。

[one\_or\_none()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.one_or_none" \o "sqlalchemy.orm.query.Query.one_or_none) is like [one()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.one" \o "sqlalchemy.orm.query.Query.one), except that if no results are found, it doesn't raise an error; it just returns None. Like [one()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.one" \o "sqlalchemy.orm.query.Query.one), however, it does raise an error if multiple results are found.

[one\_or\_none()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.one_or_none" \o "sqlalchemy.orm.query.Query.one_or_none)就像one()，除了如果没有找到结果，它不会引发错误; 它只返回无。 但是，像[one()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.one" \o "sqlalchemy.orm.query.Query.one)一样，如果找到多个结果，它会引发错误。

[scalar()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.scalar" \o "sqlalchemy.orm.query.Query.scalar) invokes the [one()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.one" \o "sqlalchemy.orm.query.Query.one) method, and upon success returns the first column of the row:

[scalar()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.scalar" \o "sqlalchemy.orm.query.Query.scalar)调用one()方法，成功返回行的第一列：

>>> query = session.query(User.id).filter(User.name == 'ed').\

... order\_by(User.id)

>>> query.scalar()

SELECT users.id AS users\_id

FROM users

WHERE users.name = ? ORDER BY users.id

('ed',)

1

### 1.9.3 Using Textual SQLs使用文本SQL

Literal strings can be used flexibly with [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query), by specifying their use with the [text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text) construct, which is accepted by most applicable methods. For example,[filter()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.filter" \o "sqlalchemy.orm.query.Query.filter) and [order\_by()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.order_by" \o "sqlalchemy.orm.query.Query.order_by):

文字字符串可以与[Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query)灵活使用，通过指定它们与[text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text)结构的使用，这被大多数适用的方法所接受。 例如，[filter()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.filter" \o "sqlalchemy.orm.query.Query.filter)和[order\_by()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.order_by" \o "sqlalchemy.orm.query.Query.order_by)：

>>> **from** **sqlalchemy** **import** text

>>> **for** user **in** session.query(User).\

... filter(text("id<224")).\

... order\_by(text("id")).all():

... **print**(user.name)

SELECT users.id AS users\_id,

users.name AS users\_name,

users.fullname AS users\_fullname,

users.password AS users\_password

FROM users

WHERE id<224 ORDER BY id

()

ed

wendy

mary

fred

Bind parameters can be specified with string-based SQL, using a colon. To specify the values, use the [params()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.params" \o "sqlalchemy.orm.query.Query.params) method:

使用冒号可以使用基于字符串的SQL来指定绑定参数。 要指定值，请使用[params()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.params" \o "sqlalchemy.orm.query.Query.params)方法：

>>> session.query(User).filter(text("id<:value and name=:name")).\

... params(value=224, name='fred').order\_by(User.id).one()

SELECT users.id AS users\_id,

users.name AS users\_name,

users.fullname AS users\_fullname,

users.password AS users\_password

FROM users

WHERE id<? and name=? ORDER BY users.id

(224, 'fred')

<User(name='fred', fullname='Fred Flinstone', password='blah')>

To use an entirely string-based statement, a [text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text) construct representing a complete statement can be passed to [from\_statement()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.from_statement" \o "sqlalchemy.orm.query.Query.from_statement). Without additional specifiers, the columns in the string SQL are matched to the model columns based on name, such as below where we use just an asterisk to represent loading all columns:

要使用完全基于字符串的语句，可以将表示完整语句的[text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text)结构传递给[from\_statement()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.from_statement" \o "sqlalchemy.orm.query.Query.from_statement)。 没有附加的说明符，字符串SQL中的列将根据名称与模型列匹配，如下所示，我们仅使用星号表示加载所有列：

>>> session.query(User).from\_statement(

... text("SELECT \* FROM users where name=:name")).\

... params(name='ed').all()

[<User(name='ed', fullname='Ed Jones', password='f8s7ccs')>]

SELECT \* FROM users where name=?

('ed',)

Matching columns on name works for simple cases but can become unwieldy when dealing with complex statements that contain duplicate column names or when using anonymized ORM constructs that don't easily match to specific names. Additionally, there is typing behavior present in our mapped columns that we might find necessary when handling result rows. For these cases, the [text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text) construct allows us to link its textual SQL to Core or ORM-mapped column expressions positionally; we can achieve this by passing column expressions as positional arguments to the [TextClause.columns()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.TextClause.columns" \o "sqlalchemy.sql.expression.TextClause.columns) method:

名称上的匹配列适用于简单的案例，但在处理包含重复列名称的复杂语句或使用不易与特定名称匹配的匿名ORM结构时，可能变得笨拙。 另外，在处理结果行时，我们可能会发现在映射列中存在打字行为。 对于这些情况，[text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text)结构允许我们将其文本SQL链接到Core或ORM映射列表达式; 我们可以通过将列表达式作为位置参数传递给[TextClause.columns()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.TextClause.columns" \o "sqlalchemy.sql.expression.TextClause.columns)方法来实现：

>>> stmt = text("SELECT name, id, fullname, password "

... "FROM users where name=:name")

>>> stmt = stmt.columns(User.name, User.id, User.fullname, User.password)

>>> session.query(User).from\_statement(stmt).params(name='ed').all()

SELECT name, id, fullname, password FROM users where name=?

('ed',)

[<User(name='ed', fullname='Ed Jones', password='f8s7ccs')>]

*New in version 1.1:*The [TextClause.columns()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.TextClause.columns" \o "sqlalchemy.sql.expression.TextClause.columns) method now accepts column expressions which will be matched positionally to a plain text SQL result set, eliminating the need for column names to match or even be unique in the SQL statement.

版本1.1中的新功能：[TextClause.columns()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.TextClause.columns" \o "sqlalchemy.sql.expression.TextClause.columns)方法现在接受将与纯文本SQL结果集位置匹配的列表达式，从而无需在SQL语句中匹配或甚至是唯一的列名。

When selecting from a [text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text) construct, the [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) may still specify what columns and entities are to be returned; instead of query(User) we can also ask for the columns individually, as in any other case:

当从一个[text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text)结构中进行选择时，[Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query)可能仍然指定要返回哪些列和实体; 而不是query(User)，我们也可以单独得到列，如在任何其他情况下：

>>> stmt = text("SELECT name, id FROM users where name=:name")

>>> stmt = stmt.columns(User.name, User.id)

>>> session.query(User.id, User.name).\

... from\_statement(stmt).params(name='ed').all()

SELECT name, id FROM users where name=?

('ed',)

[(1, u'ed')]

**See also**

[Using Textual SQL](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html" \l "sqlexpression-text) - The [text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text) construct explained from the perspective of Core-only queries.

使用文本SQL - 从纯核查询的角度解释text()构造。

### 1.9.4 Counting

[Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) includes a convenience method for counting called [count()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.count" \o "sqlalchemy.orm.query.Query.count):

查询包括一个方便的计数方法，称为count()：

>>> session.query(User).filter(User.name.like('*%e*d')).count()

SELECT count(\*) AS count\_1

FROM (SELECT users.id AS users\_id,

users.name AS users\_name,

users.fullname AS users\_fullname,

users.password AS users\_password

FROM users

WHERE users.name LIKE ?) AS anon\_1

('%ed',)

2

**Counting on**count()

[Query.count()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.count" \o "sqlalchemy.orm.query.Query.count) used to be a very complicated method when it would try to guess whether or not a subquery was needed around the existing query, and in some exotic cases it wouldn't do the right thing. Now that it uses a simple subquery every time, it's only two lines long and always returns the right answer. Use func.count() if a particular statement absolutely cannot tolerate the subquery being present.

[Query.count()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.count" \o "sqlalchemy.orm.query.Query.count)以前是一个非常复杂的方法，当它尝试猜测是否需要一个子查询在现有的查询周围，而在一些异国情况下，它不会做正确的事情。 现在它每次都使用一个简单的子查询，它只有两行长，总是返回正确的答案。 如果特定的语句绝对不能容忍子查询的存在，请使用func.count()。

The [count()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.count" \o "sqlalchemy.orm.query.Query.count) method is used to determine how many rows the SQL statement would return. Looking at the generated SQL above, SQLAlchemy always places whatever it is we are querying into a subquery, then counts the rows from that. In some cases this can be reduced to a simpler SELECT count(\*) FROM table, however modern versions of SQLAlchemy don't try to guess when this is appropriate, as the exact SQL can be emitted using more explicit means.

[count()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.count" \o "sqlalchemy.orm.query.Query.count)方法用于确定SQL语句将返回多少行。 看看上面生成的SQL，SQLAlchemy总是将我们正在查询的任何东西放在一个子查询中，然后计数那个行。 在某些情况下，这可以减少到更简单的SELECT count(\*) FROM table，但是现代版本的SQLAlchemy不会尝试猜测何时适合，因为可以使用更明确的方式发出确切的SQL。

For situations where the "thing to be counted" needs to be indicated specifically, we can specify the "count" function directly using the expression func.count(), available from the [func](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.func" \o "sqlalchemy.sql.expression.func) construct. Below we use it to return the count of each distinct user name:

对于需要具体说明“待计数的事情”的情况，我们可以使用[func](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.func" \o "sqlalchemy.sql.expression.func)构造中可用的表达式func.count()直接指定“count”函数。 下面我们用它来返回每个不同用户名的计数：

>>> **from** **sqlalchemy** **import** func

>>> session.query(func.count(User.name), User.name).group\_by(User.name).all()

[(1, u'ed'), (1, u'fred'), (1, u'mary'), (1, u'wendy')]

To achieve our simple SELECT count(\*) FROM table, we can apply it as:

>>> session.query(func.count('\*')).select\_from(User).scalar()

SELECT count(?) AS count\_1

FROM users

('\*',)

4

The usage of [select\_from()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.select_from" \o "sqlalchemy.orm.query.Query.select_from) can be removed if we express the count in terms of the User primary key directly:

如果直接User主键表示计数，则可以删除[select\_from()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.select_from" \o "sqlalchemy.orm.query.Query.select_from)的用法：

>>> session.query(func.count(User.id)).scalar()

SELECT count(users.id) AS count\_1

FROM users

()

4

## 1.10 Building a Relationship

Let's consider how a second table, related to User, can be mapped and queried. Users in our system can store any number of email addresses associated with their username. This implies a basic one to many association from the users to a new table which stores email addresses, which we will call addresses. Using declarative, we define this table along with its mapped class, Address:

我们考虑如何映射和查询与User相关的第二个表。 系统中的用户可以存储与其用户名相关联的任意数量的电子邮件地址。 这意味着从users到存储电子邮件地址的新表的基本一对多关联，我们将调用addresses。 使用声明式，我们定义了这个表及其映射类Address：

**>>> from** **sqlalchemy** **import** ForeignKey

**>>> from** **sqlalchemy.orm** **import** relationship

**>>> 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 = relationship("User", back\_populates="addresses")

**...**

**...**  **def** \_\_repr\_\_(self):

**...**  **return** "<Address(email\_address='*%s*')>" % self.email\_address

**>>>** User.addresses = relationship(

**...**  "Address", order\_by=Address.id, back\_populates="user")

The above class introduces the [ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey) construct, which is a directive applied to [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) that indicates that values in this column should be [constrained](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-constrained) to be values present in the named remote column. This is a core feature of relational databases, and is the "glue" that transforms an otherwise unconnected collection of tables to have rich overlapping relationships. The [ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey) above expresses that values in the addresses.user\_id column should be constrained to those values in the users.id column, i.e. its primary key.

上面的类介绍了[ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey)结构，它是一个应用于[Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column)的指令，表明该列中的值应被约束为指定的远程列中存在的值。这是关系数据库的一个核心特征，而且是"粘合"，可以转换另一种未连接的表集合以具有丰富的重叠关系。以上[ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey)表示addresses.user\_id列中的值应限制在users.id列中的值，即其主键。

A second directive, known as [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship), tells the ORM that the Address class itself should be linked to the User class, using the attribute Address.user. [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) uses the foreign key relationships between the two tables to determine the nature of this linkage, determining that Address.user will be [many to one](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-many-to-one). An additional [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) directive is placed on the User mapped class under the attribute User.addresses. In both[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) directives, the parameter [relationship.back\_populates](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.back_populates" \o "sqlalchemy.orm.relationship) is assigned to refer to the complementary attribute names; by doing so, each [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) can make intelligent decision about the same relationship as expressed in reverse; on one side, Address.user refers to a User instance, and on the other side, User.addresses refers to a list of Address instances.

第二个指令，称为[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)，告诉ORM，Address类本身应该使用Address.user属性链接到User类。 [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)使用两个表之间的外键关系来确定此链接的性质，确定Address.user将是多对一的。在User.addresses属性下的User映射类上放置一个附加relationship()指令。在两个relationship()指令中，参数[relationship.back\_populates](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.back_populates" \o "sqlalchemy.orm.relationship)被分配以引用互补属性名称;通过这样做，每个relationship()可以对相反的关系做出明智的决定;一方面，Address.user指的是一个User实例，另一方面，User.addresses是一个Address实例列表。

**Note**

The [relationship.back\_populates](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.back_populates" \o "sqlalchemy.orm.relationship) parameter is a newer version of a very common SQLAlchemy feature called [relationship.backref](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.backref" \o "sqlalchemy.orm.relationship). The [relationship.backref](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.backref" \o "sqlalchemy.orm.relationship) parameter hasn't gone anywhere and will always remain available! The [relationship.back\_populates](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.back_populates" \o "sqlalchemy.orm.relationship) is the same thing, except a little more verbose and easier to manipulate. For an overview of the entire topic, see the section [Linking Relationships with Backref](http://docs.sqlalchemy.org/en/rel_1_1/orm/backref.html" \l "relationships-backref).

[relationship.back\_populates](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.back_populates" \o "sqlalchemy.orm.relationship)参数是一个非常常见的SQLAlchemy功能的新版本，名为[relationship.backref](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.backref" \o "sqlalchemy.orm.relationship)。 [relationship.backref](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.backref" \o "sqlalchemy.orm.relationship)参数没有任何地方，并且始终保持可用！ [relationship.back\_populates](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.back_populates" \o "sqlalchemy.orm.relationship)是一样的，除了一点更冗长和更容易操纵。 有关整个主题的概述，请参阅链接与Backref的关系部分。

The reverse side of a many-to-one relationship is always [one to many](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-one-to-many). A full catalog of available [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) configurations is at [Basic Relationship Patterns](http://docs.sqlalchemy.org/en/rel_1_1/orm/basic_relationships.html" \l "relationship-patterns).

多对一关系的反面就是一对多的关系。 可用relationship()配置的完整目录是基本关系模式。

The two complementing relationships Address.user and User.addresses are referred to as a [bidirectional relationship](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-bidirectional-relationship), and is a key feature of the SQLAlchemy ORM. The section [Linking Relationships with Backref](http://docs.sqlalchemy.org/en/rel_1_1/orm/backref.html" \l "relationships-backref) discusses the "backref" feature in detail.

Address.user和User.addresses两个互补关系被称为双向关系，是SQLAlchemy ORM的一个关键特征。 [Linking Relationships with Backref](http://docs.sqlalchemy.org/en/rel_1_1/orm/backref.html" \l "relationships-backref) 的部分详细讨论了"backref"功能。

Arguments to [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) which concern the remote class can be specified using strings, assuming the Declarative system is in use. Once all mappings are complete, these strings are evaluated as Python expressions in order to produce the actual argument, in the above case the User class. The names which are allowed during this evaluation include, among other things, the names of all classes which have been created in terms of the declared base.

可以使用字符串来指定关系到远程类的relationship()的参数，假设声明式系统正在使用。 一旦所有映射完成，这些字符串将被评估为Python表达式，以便产生实际参数，在上述情况下为User类。 在此评估期间允许的名称包括根据声明的基础创建的所有类的名称。

See the docstring for [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) for more detail on argument style.

**Did you know ?**

* a FOREIGN KEY constraint in most (though not all) relational databases can only link to a primary key column, or a column that has a UNIQUE constraint.大多数(尽管并非全部)关系数据库中的FOREIGN KEY约束只能链接到主键列或具有UNIQUE约束的列。
* a FOREIGN KEY constraint that refers to a multiple column primary key, and itself has multiple columns, is known as a "composite foreign key". It can also reference a subset of those columns.一个引用多列主键的FOREIGN KEY约束，它本身有多个列，被称为“复合外键”。 它也可以引用这些列的一个子集。
* FOREIGN KEY columns can automatically update themselves, in response to a change in the referenced column or row. This is known as the CASCADE *referential action*, and is a built in function of the relational database.FOREIGNKEY列可以自动更新自己，以响应引用的列或行的更改。 这被称为CASCADE参考动作，并且是关系数据库的内置函数。
* FOREIGN KEY can refer to its own table. This is referred to as a "self-referential" foreign key.FOREIGN KEY可以参考自己的表。 这被称为“自引用”外键。
* Read more about foreign keys at [Foreign Key - Wikipedia](http://en.wikipedia.org/wiki/Foreign_key).更多关于外键的外键 - 维基百科。

We'll need to create the addresses table in the database, so we will issue another CREATE from our metadata, which will skip over tables which have already been created:

我们需要在数据库中创建地址表，所以我们将从我们的元数据中发出另一个CREATE，它将跳过已经创建的表：

>>> Base.metadata.create\_all(engine)

PRAGMA...

CREATE TABLE addresses (

id INTEGER NOT NULL,

email\_address VARCHAR NOT NULL,

user\_id INTEGER,

PRIMARY KEY (id),

FOREIGN KEY(user\_id) REFERENCES users (id)

)

()

COMMIT

1.11 Working with Related Objects

Now when we create a User, a blank addresses collection will be present. Various collection types, such as sets and dictionaries, are possible here (see [Customizing Collection Access](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "custom-collections) for details), but by default, the collection is a Python list.

现在当我们创建一个用户时，将会出现一个空白的地址集合。 各种集合类型(如集合和字典)都可以在这里(有关详细信息，请参阅自定义集合访问)，但默认情况下，集合是Python列表。

>>> jack = User(name='jack', fullname='Jack Bean', password='gjffdd')

>>> jack.addresses[]

We are free to add Address objects on our User object. In this case we just assign a full list directly:

我们可以在我们的User对象上添加Address对象。 在这种情况下，我们只需直接分配一个完整列表：

>>> jack.addresses = [

... [Address(email\_address='jack@google.com'),](mailto:Address(email_address='jack@google.com'),)

... Address(email\_address='j25@yahoo.com')]

When using a bidirectional relationship, elements added in one direction automatically become visible in the other direction. This behavior occurs based on attribute on-change events and is evaluated in Python, without using any SQL:

当使用双向关系时，沿一个方向添加的元素在另一个方向自动变得可见。 这种行为是基于属性on-change事件发生的，并且在Python中进行评估，而不使用任何SQL：

>>> jack.addresses[1]

<Address(email\_address='j25@yahoo.com')>

>>> jack.addresses[1].user

<User(name='jack', fullname='Jack Bean', password='gjffdd')>

Let's add and commit Jack Bean to the database. jack as well as the two Address members in the corresponding addresses collection are both added to the session at once, using a process known as ****cascading****:

我们来添加并提交Jack Bean到数据库。 jack以及相应地址集合中的两个Address成员都一次添加到会话中，使用称为级联的过程：

>>> session.add(jack)

>>> session.commit()

INSERT INTO users (name, fullname, password) VALUES (?, ?, ?)

('jack', 'Jack Bean', 'gjffdd')

INSERT INTO addresses (email\_address, user\_id) VALUES (?, ?)

('jack@google.com', 5)

INSERT INTO addresses (email\_address, user\_id) VALUES (?, ?)

('j25@yahoo.com', 5)

COMMIT

Querying for Jack, we get just Jack back. No SQL is yet issued for Jack's addresses:

查询Jack，我们得到了Jack。 还没有为杰克的地址发出SQL：

>>> jack = session.query(User).\

... filter\_by(name='jack').one()

BEGIN (implicit)

SELECT users.id AS users\_id,

users.name AS users\_name,

users.fullname AS users\_fullname,

users.password AS users\_password

FROM users

WHERE users.name = ?

('jack',)

>>> jack<User(name='jack', fullname='Jack Bean', password='gjffdd')>

Let's look at the addresses collection. Watch the SQL:

>>> jack.addresses

SELECT addresses.id AS addresses\_id,

addresses.email\_address AS

addresses\_email\_address,

addresses.user\_id AS addresses\_user\_id

FROM addresses

WHERE ? = addresses.user\_id ORDER BY addresses.id

(5,)

[<Address(email\_address='jack@google.com')>, <Address(email\_address='j25@yahoo.com')>]

When we accessed the addresses collection, SQL was suddenly issued. This is an example of a [lazy loading](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-lazy-loading) relationship. The addresses collection is now loaded and behaves just like an ordinary list. We'll cover ways to optimize the loading of this collection in a bit.

当我们访问地址集合时，SQL被突然发出。 这是一个懒加载关系的例子。 地址集合现在被加载，并且行为就像普通列表。 我们将介绍如何优化此集合的加载。

## 1.12 Querying with Joins

Now that we have two tables, we can show some more features of [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query), specifically how to create queries that deal with both tables at the same time. The [Wikipedia page on SQL JOIN](http://en.wikipedia.org/wiki/Join_(SQL)) offers a good introduction to join techniques, several of which we'll illustrate here.

现在我们有两个表，我们可以显示一些Query的更多功能，特别是如何创建同时处理这两个表的查询。 SQL JOIN的维基百科页面提供了一个很好的介绍加入技术，其中有几个将在这里进行说明。

To construct a simple implicit join between User and Address, we can use [Query.filter()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.filter" \o "sqlalchemy.orm.query.Query.filter) to equate their related columns together. Below we load the Userand Address entities at once using this method:

要构建用户和地址之间的简单隐式连接，我们可以使用Query.filter()将其相关列相等。 下面我们使用以下方法一次加载Userand Address实体：

>>> **for** u, a **in** session.query(User, Address).\

... filter(User.id==Address.user\_id).\

... [filter(Address.email\_address=='jack@google.com').\](mailto:filter(Address.email_address=='jack@google.com')./)

... all():

... **print**(u)

... **print**(a)

SELECT users.id AS users\_id,

users.name AS users\_name,

users.fullname AS users\_fullname,

users.password AS users\_password,

addresses.id AS addresses\_id,

addresses.email\_address AS addresses\_email\_address,

addresses.user\_id AS addresses\_user\_id

FROM users, addresses

WHERE users.id = addresses.user\_id

AND addresses.email\_address = ?

('jack@google.com',)

<User(name='jack', fullname='Jack Bean', password='gjffdd')>

<Address(email\_address='jack@google.com')>

The actual SQL JOIN syntax, on the other hand, is most easily achieved using the [Query.join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join) method:

另一方面，实际的SQL JOIN语法是使用[Query.join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join)方法最容易实现的：

>>> session.query(User).join(Address).\

... [filter(Address.email\_address=='jack@google.com').\](mailto:filter(Address.email_address=='jack@google.com')./)

... all()

SELECT users.id AS users\_id,

users.name AS users\_name,

users.fullname AS users\_fullname,

users.password AS users\_password

FROM users JOIN addresses ON users.id = addresses.user\_id

WHERE addresses.email\_address = ?

('jack@google.com',)

[<User(name='jack', fullname='Jack Bean', password='gjffdd')>]

[Query.join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join) knows how to join between User and Address because there's only one foreign key between them. If there were no foreign keys, or several, [Query.join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join) works better when one of the following forms are used:

[Query.join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join)知道如何在User 和Address 之间加入，因为它们之间只有一个外键。 如果没有外键，或者几个，[Query.join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join)可以使用以下表单之一来更好地工作：

query.join(Address, User.id==Address.user\_id) *# explicit condition*

query.join(User.addresses) *# specify relationship from left to right*

query.join(Address, User.addresses) *# same, with explicit target*

query.join('addresses') *# same, using a string*

As you would expect, the same idea is used for "outer" joins, using the [outerjoin()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.outerjoin" \o "sqlalchemy.orm.query.Query.outerjoin) function:

正如你所期望的，使用[outerjoin()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.outerjoin" \o "sqlalchemy.orm.query.Query.outerjoin)函数的“外”连接使用相同的想法：

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

The reference documentation for [join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join) contains detailed information and examples of the calling styles accepted by this method; [join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join) is an important method at the center of usage for any SQL-fluent application.

[join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join)的参考文档包含此方法接受的调用样式的详细信息和示例; [join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join)是任何SQL流畅应用程序使用中心的重要方法。

**What does**[Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query)**select from if there's multiple entities?如果有多个实体，“查询”会选择什么？**

The [Query.join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join) method will ****typically join from the leftmost item**** in the list of entities, when the ON clause is omitted, or if the ON clause is a plain SQL expression. To control the first entity in the list of JOINs, use the [Query.select\_from()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.select_from" \o "sqlalchemy.orm.query.Query.select_from) method:

[Query.join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join)方法通常从实体列表中的最左边的项目中加入，当省略ON子句时，或者ON子句是纯SQL表达式。 要控制JOIN列表中的第一个实体，请使用[Query.select\_from()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.select_from" \o "sqlalchemy.orm.query.Query.select_from)方法：

query = session.query(User, Address).select\_from(Address).join(User)

### 1.12.1 Using Aliases

When querying across multiple tables, if the same table needs to be referenced more than once, SQL typically requires that the table be *aliased* with another name, so that it can be distinguished against other occurrences of that table. The [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) supports this most explicitly using the [aliased](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.aliased" \o "sqlalchemy.orm.aliased) construct. Below we join to the Address entity twice, to locate a user who has two distinct email addresses at the same time:

当跨多个表进行查询时，如果同一个表需要多次引用，则SQL通常需要将该表与其他名称进行别名，以便可以将该表与其他表的出现区分开来。 [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) 使用别名([aliased](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.aliased" \o "sqlalchemy.orm.aliased))构造最明显地支持这一点。 下面我们加入Address实体两次，以查找同时具有两个不同电子邮件地址的用户：

>>> **from** **sqlalchemy.orm** **import** aliased

>>> adalias1 = aliased(Address)

>>> adalias2 = aliased(Address)

>>> **for** username, email1, email2 **in** \

... session.query(User.name, adalias1.email\_address, adalias2.email\_address).\

... join(adalias1, User.addresses).\

... join(adalias2, User.addresses).\

... [filter(adalias1.email\_address=='jack@google.com').\](mailto:filter(adalias1.email_address=='jack@google.com')./)

... [filter(adalias2.email\_address=='j25@yahoo.com'):](mailto:filter(adalias2.email_address=='j25@yahoo.com'):)

... **print**(username, email1, email2)

SELECT users.name AS users\_name,

addresses\_1.email\_address AS addresses\_1\_email\_address,

addresses\_2.email\_address AS addresses\_2\_email\_address

FROM users JOIN addresses AS addresses\_1

ON users.id = addresses\_1.user\_id

JOIN addresses AS addresses\_2

ON users.id = addresses\_2.user\_id

WHERE addresses\_1.email\_address = ?

AND addresses\_2.email\_address = ?

('jack@google.com', 'j25@yahoo.com')

jack jack**@google.com**

j25**@yahoo.com**

### 1.12.2 Using Subqueries

The [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) is suitable for generating statements which can be used as subqueries. Suppose we wanted to load User objects along with a count of how many Address records each user has. The best way to generate SQL like this is to get the count of addresses grouped by user ids, and JOIN to the parent. In this case we use a LEFT OUTER JOIN so that we get rows back for those users who don't have any addresses, e.g.:

该[Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query)适用于生成可用作子查询的语句。 假设我们要加载用户对象以及每个用户拥有多少个地址记录的计数。 生成SQL的最佳方法就是获取按用户ID分组的地址数，并将JOIN添加到父级。 在这种情况下，我们使用LEFT OUTER JOIN，以便我们为没有任何地址的用户获取行，例如：

SELECT users.\*, adr\_count.address\_count

FROM users LEFT OUTER JOIN

(SELECT user\_id, count(\*)

AS address\_count

FROM addresses GROUP BY user\_id)

AS adr\_count

ON users.id=adr\_count.user\_id

Using the [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query), we build a statement like this from the inside out. The statement accessor returns a SQL expression representing the statement generated by a particular [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) - this is an instance of a [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) construct, which are described in [SQL Expression Language Tutorial](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html):

使用Query，我们从内而外构建一个这样的语句。 语句访问器返回一个表达由特定Query生成的语句的SQL表达式 - 这是一个select()结构的实例，它在SQL表达式语言教程中描述：

**>>> from** **sqlalchemy.sql** **import** func

**>>>** stmt = session.query(Address.user\_id, func.count('\*').\

**...**  label('address\_count')).\

**...**  group\_by(Address.user\_id).subquery()

The func keyword generates SQL functions, and the subquery() method on [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) produces a SQL expression construct representing a SELECT statement embedded within an alias (it's actually shorthand for query.statement.alias()).

func关键字生成SQL函数，Query上的子查询()方法生成一个表达嵌入在别名中的SELECT语句的SQL表达式构造(实际上它是query.statement.alias()的缩写)。

Once we have our statement, it behaves like a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) construct, such as the one we created for users at the start of this tutorial. The columns on the statement are accessible through an attribute called c:

一旦我们有了我们的表达式，它的行为就像一个[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)结构，比如我们为本教程开始时为用户创建的结构。 语句上的列可通过名为c的属性访问：

>>> **for** u, count **in** session.query(User, stmt.c.address\_count).\

... outerjoin(stmt, User.id==stmt.c.user\_id).order\_by(User.id):

... **print**(u, count)

SELECT users.id AS users\_id,

users.name AS users\_name,

users.fullname AS users\_fullname,

users.password AS users\_password,

anon\_1.address\_count AS anon\_1\_address\_count

FROM users LEFT OUTER JOIN

(SELECT addresses.user\_id AS user\_id, count(?) AS address\_count

FROM addresses GROUP BY addresses.user\_id) AS anon\_1

ON users.id = anon\_1.user\_id

ORDER BY users.id

('\*',)

<User(name='ed', fullname='Ed Jones', password='f8s7ccs')>

None

<User(name='wendy', fullname='Wendy Williams', password='foobar')>

None

<User(name='mary', fullname='Mary Contrary', password='xxg527')>

None

<User(name='fred', fullname='Fred Flinstone', password='blah')>

None

<User(name='jack', fullname='Jack Bean', password='gjffdd')>

2

### 1.12.3 Selecting Entities from Subqueries

Above, we just selected a result that included a column from a subquery. What if we wanted our subquery to map to an entity ? For this we use aliased() to associate an "alias" of a mapped class to a subquery:

以上，我们刚刚选择了一个包含子查询列的结果。 如果我们希望我们的子查询映射到一个实体呢？ 为此，我们使用aliased()将映射类的"别名"与子查询相关联：

>>> stmt = session.query(Address).\

... filter(Address.email\_address != ['j25@yahoo.com').\](mailto:'j25@yahoo.com')./)

... subquery()

>>> adalias = aliased(Address, stmt)

>>> **for** user, address **in** session.query(User, adalias).\

... join(adalias, User.addresses):

... **print**(user)

... **print**(address)

SELECT users.id AS users\_id,

users.name AS users\_name,

users.fullname AS users\_fullname,

users.password AS users\_password,

anon\_1.id AS anon\_1\_id,

anon\_1.email\_address AS anon\_1\_email\_address,

anon\_1.user\_id AS anon\_1\_user\_id

FROM users JOIN

(SELECT addresses.id AS id,

addresses.email\_address AS email\_address,

addresses.user\_id AS user\_id

FROM addresses

WHERE addresses.email\_address != ?)

AS anon\_1

ON users.id = anon\_1.user\_id

('j25@yahoo.com',)

<User(name='jack', fullname='Jack Bean', password='gjffdd')>

<Address(email\_address='jack@google.com')>

### 1.12.4 Using EXISTS

The EXISTS keyword in SQL is a boolean operator which returns True if the given expression contains any rows. It may be used in many scenarios in place of joins, and is also useful for locating rows which do not have a corresponding row in a related table.

SQL中的EXISTS关键字是一个布尔运算符，如果给定的表达式包含任何行，则返回True。 它可以在许多情况下用于代替连接，并且也用于定位在相关表中没有相应行的行。

There is an explicit EXISTS construct, which looks like this:

有一个明确的EXISTS结构，如下所示：

>>> **from** **sqlalchemy.sql** **import** exists

>>> stmt = exists().where(Address.user\_id==User.id)

>>> **for** name, **in** session.query(User.name).filter(stmt):

... **print**(name)

SELECT users.name AS users\_name

FROM users

WHERE EXISTS (SELECT \*

FROM addresses

WHERE addresses.user\_id = users.id)

()

jack

The [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) features several operators which make usage of EXISTS automatically. Above, the statement can be expressed along the User.addresses relationship using [any()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator.any" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator.any):

>>> **for** name, **in** session.query(User.name).\

... filter(User.addresses.any()):

... **print**(name)

SELECT users.name AS users\_name

FROM users

WHERE EXISTS (SELECT 1

FROM addresses

WHERE users.id = addresses.user\_id)

()

jack

[any()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator.any" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator.any) takes criterion as well, to limit the rows matched:

>>> **for** name, **in** session.query(User.name).\

... filter(User.addresses.any(Address.email\_address.like('*%g*oogle%'))):

... **print**(name)

SELECT users.name AS users\_name

FROM users

WHERE EXISTS (SELECT 1

FROM addresses

WHERE users.id = addresses.user\_id AND addresses.email\_address LIKE ?)

('%google%',)

jack

[has()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator.has" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator.has) is the same operator as [any()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator.any" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator.any) for many-to-one relationships (note the ~ operator here too, which means "NOT"):

>>> session.query(Address).\

... filter(~Address.user.has(User.name=='jack')).all()

SELECT addresses.id AS addresses\_id,

addresses.email\_address AS addresses\_email\_address,

addresses.user\_id AS addresses\_user\_id

FROM addresses

WHERE NOT (EXISTS (SELECT 1

FROM users

WHERE users.id = addresses.user\_id AND users.name = ?))

('jack',)

[]

### 1.12.5 Common Relationship Operators通用关系运算符

Here's all the operators which build on relationships - each one is linked to its API documentation which includes full details on usage and behavior:

这里是所有基于关系的运算符 - 每个都链接到其API文档，其中包含有关使用和行为的完整详细信息：

[\_\_eq\_\_()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator.__eq__" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator.__eq__) (many-to-one "equals" comparison):

query.filter(Address.user == someuser)

[\_\_ne\_\_()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator.__ne__" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator.__ne__) (many-to-one "not equals" comparison):

query.filter(Address.user != someuser)

IS NULL (many-to-one comparison, also uses [\_\_eq\_\_()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator.__eq__" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator.__eq__)):

query.filter(Address.user == **None**)

[contains()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator.contains" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator.contains) (used for one-to-many collections):

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

[any()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator.any" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator.any) (used for collections):

query.filter(User.addresses.any(Address.email\_address == 'bar'))

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

[has()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator.has" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator.has) (used for scalar references):

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

[Query.with\_parent()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.with_parent" \o "sqlalchemy.orm.query.Query.with_parent) (used for any relationship):

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

1.13 Eager Loading预加载

Recall earlier that we illustrated a [lazy loading](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-lazy-loading) operation, when we accessed the User.addresses collection of a User and SQL was emitted. If you want to reduce the number of queries (dramatically, in many cases), we can apply an eager load to the query operation. SQLAlchemy offers three types of eager loading, two of which are automatic, and a third which involves custom criterion. All three are usually invoked via functions known as query options which give additional instructions to the [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) on how we would like various attributes to be loaded, via the [Query.options()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.options" \o "sqlalchemy.orm.query.Query.options) method.

回想起来，当我们访问User的User.addresses集合并发出SQL时，我们展示了一个懒加载操作。 如果要减少查询次数(在很多情况下，会大大减少)，我们可以对查询操作应用一个预加载。 SQLAlchemy提供三种预加载，其中两种是自动加载，另外三种是自定义标准。 所有这三个通常通过称为查询选项的函数来调用，这些函数通过[Query.options()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.options" \o "sqlalchemy.orm.query.Query.options)方法为[Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query)提供了关于如何加载各种属性的其他指令。

### 1.13.1 Subquery Load

In this case we'd like to indicate that User.addresses should load eagerly. A good choice for loading a set of objects as well as their related collections is the [orm.subqueryload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.subqueryload" \o "sqlalchemy.orm.subqueryload) option, which emits a second SELECT statement that fully loads the collections associated with the results just loaded. The name "subquery" originates from the fact that the SELECT statement constructed directly via the [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) is re-used, embedded as a subquery into a SELECT against the related table. This is a little elaborate but very easy to use:

在这种情况下，我们希望指出User.addresses应该加载。 加载一组对象及其相关集合的一个好选择是[orm.subqueryload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.subqueryload" \o "sqlalchemy.orm.subqueryload)选项，它会发出第二个SELECT语句，它完全加载与刚加载的结果相关联的集合。 名称“子查询”源于以下事实：直接通过[Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query)构建的SELECT语句被重新使用，作为子查询嵌入到相对于表的SELECT中。 这是一个有点复杂但很容易使用：

>>> **from** **sqlalchemy.orm** **import** subqueryload

>>> jack = session.query(User).\

... options(subqueryload(User.addresses)).\

... filter\_by(name='jack').one()

SELECT users.id AS users\_id,

users.name AS users\_name,

users.fullname AS users\_fullname,

users.password AS users\_password

FROM users

WHERE users.name = ?

('jack',)

SELECT addresses.id AS addresses\_id,

addresses.email\_address AS addresses\_email\_address,

addresses.user\_id AS addresses\_user\_id,

anon\_1.users\_id AS anon\_1\_users\_id

FROM (SELECT users.id AS users\_id

FROM users WHERE users.name = ?) AS anon\_1

JOIN addresses ON anon\_1.users\_id = addresses.user\_id

ORDER BY anon\_1.users\_id, addresses.id

('jack',)

>>> jack

<User(name='jack', fullname='Jack Bean', password='gjffdd')>

>>> jack.addresses

[<Address(email\_address='jack@google.com')>, <Address(email\_address='j25@yahoo.com')>]

**Note**

[subqueryload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.subqueryload" \o "sqlalchemy.orm.subqueryload) when used in conjunction with limiting such as [Query.first()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.first" \o "sqlalchemy.orm.query.Query.first), [Query.limit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.limit" \o "sqlalchemy.orm.query.Query.limit) or [Query.offset()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.offset" \o "sqlalchemy.orm.query.Query.offset) should also include [Query.order\_by()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.order_by" \o "sqlalchemy.orm.query.Query.order_by) on a unique column in order to ensure correct results. See [The Importance of Ordering](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "subqueryload-ordering).

与[Query.first()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.first" \o "sqlalchemy.orm.query.Query.first), [Query.limit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.limit" \o "sqlalchemy.orm.query.Query.limit) or [Query.offset()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.offset" \o "sqlalchemy.orm.query.Query.offset)一起使用时，subqueryload()也应该在唯一列上包含Query.order\_by()，以确保正确的结果。 参见[The Importance of Ordering](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "subqueryload-ordering)。

### 1.13.2 Joined Load

The other automatic eager loading function is more well known and is called [orm.joinedload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.joinedload" \o "sqlalchemy.orm.joinedload). This style of loading emits a JOIN, by default a LEFT OUTER JOIN, so that the lead object as well as the related object or collection is loaded in one step. We illustrate loading the same addresses collection in this way - note that even though the User.addresses collection on jack is actually populated right now, the query will emit the extra join regardless:

另一种自动加载功能更为人所知，称为[orm.joinedload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.joinedload" \o "sqlalchemy.orm.joinedload)。 这种加载方式在默认情况下会发出一个JOIN，一个LEFT OUTER JOIN，以便引导对象以及相关的对象或集合在一个步骤中被加载。 我们以这种方式说明了加载相同的地址集合 - 请注意，尽管目前实际上已经填充了插孔上的User.addresses集合，但查询将会发出额外的连接：

>>> **from** **sqlalchemy.orm** **import** joinedload

>>> jack = session.query(User).\

... options(joinedload(User.addresses)).\

... filter\_by(name='jack').one()

SELECT users.id AS users\_id,

users.name AS users\_name,

users.fullname AS users\_fullname,

users.password AS users\_password,

addresses\_1.id AS addresses\_1\_id,

addresses\_1.email\_address AS addresses\_1\_email\_address,

addresses\_1.user\_id AS addresses\_1\_user\_id

FROM users

LEFT OUTER JOIN addresses AS addresses\_1 ON users.id = addresses\_1.user\_id

WHERE users.name = ? ORDER BY addresses\_1.id

('jack',)

>>> jack

<User(name='jack', fullname='Jack Bean', password='gjffdd')>

>>> jack.addresses

[<Address(email\_address='jack@google.com')>, <Address(email\_address='j25@yahoo.com')>]

Note that even though the OUTER JOIN resulted in two rows, we still only got one instance of User back. This is because [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) applies a "uniquing" strategy, based on object identity, to the returned entities. This is specifically so that joined eager loading can be applied without affecting the query results.

请注意，即使OUTER JOIN导致两行，我们仍然只有一个User的实例。 这是因为[Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query)将基于对象标识的“唯一”策略应用于返回的实体。 这是特别的，以便可以应用加入的渴望加载而不影响查询结果。

While [joinedload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.joinedload" \o "sqlalchemy.orm.joinedload) has been around for a long time, [subqueryload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.subqueryload" \o "sqlalchemy.orm.subqueryload) is a newer form of eager loading. [subqueryload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.subqueryload" \o "sqlalchemy.orm.subqueryload) tends to be more appropriate for loading related collections while [joinedload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.joinedload" \o "sqlalchemy.orm.joinedload) tends to be better suited for many-to-one relationships, due to the fact that only one row is loaded for both the lead and the related object.

在[joinedload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.joinedload" \o "sqlalchemy.orm.joinedload)已经存在了很长时间的时候，[subqueryload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.subqueryload" \o "sqlalchemy.orm.subqueryload)是一种更加新颖的加载形式。 对于加载相关的集合而言，[subqueryload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.subqueryload" \o "sqlalchemy.orm.subqueryload)往往更适合于加载相关的集合，而[joinedload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.joinedload" \o "sqlalchemy.orm.joinedload)倾向于更适合于多对一的关系，这是因为为铅和相关对象只加载一行。

joinedload()**is not a replacement for**join()

The join created by [joinedload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.joinedload" \o "sqlalchemy.orm.joinedload) is anonymously aliased such that it ****does not affect the query results****. An [Query.order\_by()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.order_by" \o "sqlalchemy.orm.query.Query.order_by) or [Query.filter()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.filter" \o "sqlalchemy.orm.query.Query.filter) call ****cannot**** reference these aliased tables - so-called "user space" joins are constructed using [Query.join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join). The rationale for this is that [joinedload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.joinedload" \o "sqlalchemy.orm.joinedload) is only applied in order to affect how related objects or collections are loaded as an optimizing detail - it can be added or removed with no impact on actual results. See the section [The Zen of Joined Eager Loading](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "zen-of-eager-loading) for a detailed description of how this is used.

由joinload()创建的连接是匿名别名的，因此不影响查询结果。 Query.order\_by()或Query.filter()调用不能引用这些别名表 - 使用Query.join()构造所谓的"用户空间"连接。 这样做的理由是，仅仅应用了joinload()才能影响相关对象或集合作为优化细节的加载 - 它可以被添加或删除而不影响实际的结果。 请参阅"[The Zen of Joined Eager Loading](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "zen-of-eager-loading)"加载中的详细说明。

### 1.13.3 Explicit Join + Eagerload

A third style of eager loading is when we are constructing a JOIN explicitly in order to locate the primary rows, and would like to additionally apply the extra table to a related object or collection on the primary object. This feature is supplied via the [orm.contains\_eager()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.contains_eager" \o "sqlalchemy.orm.contains_eager) function, and is most typically useful for pre-loading the many-to-one object on a query that needs to filter on that same object. Below we illustrate loading an Address row as well as the related User object, filtering on the User named "jack" and using [orm.contains\_eager()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.contains_eager" \o "sqlalchemy.orm.contains_eager) to apply the "user" columns to the Address.user attribute:

第三种预加载方式是当我们为了定位主行而明确构建一个JOIN，并且想要将额外的表应用到主对象上的相关对象或集合。 此功能通过orm.contains\_eager()函数提供，最常用于在需要在同一对象上过滤的查询上预加载多对一对象。 下面我们将说明如何加载Address行以及相关的User对象，过滤用户名为"jack"并使用orm.contains\_eager()将"user"列应用于Address.user属性：

>>> **from** **sqlalchemy.orm** **import** contains\_eager

>>> jacks\_addresses = session.query(Address).\

... join(Address.user).\

... filter(User.name=='jack').\

... options(contains\_eager(Address.user)).\

... all()

SELECT users.id AS users\_id,

users.name AS users\_name,

users.fullname AS users\_fullname,

users.password AS users\_password,

addresses.id AS addresses\_id,

addresses.email\_address AS addresses\_email\_address,

addresses.user\_id AS addresses\_user\_id

FROM addresses JOIN users ON users.id = addresses.user\_id

WHERE users.name = ?

('jack',)

>>> jacks\_addresses

[<Address(email\_address='jack@google.com')>, <Address(email\_address='j25@yahoo.com')>]

>>> jacks\_addresses[0].user

<User(name='jack', fullname='Jack Bean', password='gjffdd')>

For more information on eager loading, including how to configure various forms of loading by default, see the section [Relationship Loading Techniques](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html).

有关预加载的更多信息，包括默认情况下如何配置各种加载形式，请参阅"[Relationship Loading Techniques](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html)"一节。

1.14 Deleting

Let's try to delete jack and see how that goes. We'll mark the object as deleted in the session, then we'll issue a count query to see that no rows remain:

让我们尝试删除jack，看看怎么回事。 我们将在会话中将对象标记为已删除，然后我们将发出一个计数count，看看没有行保留：

>>> session.delete(jack)

>>> session.query(User).filter\_by(name='jack').count()

UPDATE addresses SET user\_id=? WHERE addresses.id = ?

((None, 1), (None, 2))

DELETE FROM users WHERE users.id = ?

(5,)

SELECT count(\*) AS count\_1

FROM (SELECT users.id AS users\_id,

users.name AS users\_name,

users.fullname AS users\_fullname,

users.password AS users\_password

FROM users

WHERE users.name = ?) AS anon\_1

('jack',)

0

So far, so good. How about Jack's Address objects ?

>>> session.query(Address).filter(

... Address.email\_address.in\_(['jack@google.com', ['j25@yahoo.com'])](mailto:'j25@yahoo.com']))

... ).count()

SELECT count(\*) AS count\_1

FROM (SELECT addresses.id AS addresses\_id,

addresses.email\_address AS addresses\_email\_address,

addresses.user\_id AS addresses\_user\_id

FROM addresses

WHERE addresses.email\_address IN (?, ?)) AS anon\_1

('jack@google.com', 'j25@yahoo.com')

2

Uh oh, they're still there ! Analyzing the flush SQL, we can see that the user\_id column of each address was set to NULL, but the rows weren't deleted. SQLAlchemy doesn't assume that deletes cascade, you have to tell it to do so.

哦，他们还在那里！ 分析刷新SQL，我们可以看到每个地址的user\_id列设置为NULL，但是行没有被删除。 SQLAlchemy不假设删除级联，你必须告诉它这样做。

### 1.14.1 Configuring delete/delete-orphan Cascade

We will configure ****cascade**** options on the User.addresses relationship to change the behavior. While SQLAlchemy allows you to add new attributes and relationships to mappings at any point in time, in this case the existing relationship needs to be removed, so we need to tear down the mappings completely and start again - we'll close the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session):

我们将在User.addresses关系上配置级联选项以更改行为。 虽然SQLAlchemy允许您在任何时间点向映射添加新的属性和关系，但在这种情况下，需要删除现有的关系，因此我们需要完全拆除映射并重新开始 - 我们将关闭会话：

**>>>** session.close()

ROLLBACK

and use a new [declarative\_base()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.declarative_base" \o "sqlalchemy.ext.declarative.declarative_base):

**>>>** Base = declarative\_base()

Next we'll declare the User class, adding in the addresses relationship including the cascade configuration (we'll leave the constructor out too):

接下来，我们将声明User类，添加addresses 关系，包括级联配置(我们将离开构造函数)：

**>>> class** **User**(Base):

**...**  \_\_tablename\_\_ = 'users'

**...**

**...**  id = Column(Integer, primary\_key=**True**)

**...**  name = Column(String)

**...**  fullname = Column(String)

**...**  password = Column(String)

**...**

**...**  addresses = relationship("Address", back\_populates='user',

**...**  cascade="all, delete, delete-orphan")

**...**

**...**  **def** \_\_repr\_\_(self):

**...**  **return** "<User(name='*%s*', fullname='*%s*', password='*%s*')>" % (

**...**  self.name, self.fullname, self.password)

Then we recreate Address, noting that in this case we've created the Address.user relationship via the User class already:

然后我们重新创建地址，注意在这种情况下，我们已经通过User类创建了Address.user关系：

**>>> 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 = relationship("User", back\_populates="addresses")

**...**

**...**  **def** \_\_repr\_\_(self):

**...**  **return** "<Address(email\_address='*%s*')>" % self.email\_address

Now when we load the user jack (below using [get()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.get" \o "sqlalchemy.orm.query.Query.get), which loads by primary key), removing an address from the corresponding addresses collection will result in that Address being deleted:

现在当我们加载用户jack (下面使用[get()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.get" \o "sqlalchemy.orm.query.Query.get)，由主键加载)，从相应的地址集合中删除一个addresses将导致该Address被删除：

*# load Jack by primary key*

>>> jack = session.query(User).get(5)

BEGIN (implicit)

SELECT users.id AS users\_id,

users.name AS users\_name,

users.fullname AS users\_fullname,

users.password AS users\_password

FROM users

WHERE users.id = ?

(5,)

*# remove one Address (lazy load fires off)*

>>> **del** jack.addresses[1]

SELECT addresses.id AS addresses\_id,

addresses.email\_address AS addresses\_email\_address,

addresses.user\_id AS addresses\_user\_id

FROM addresses

WHERE ? = addresses.user\_id

(5,)

*# only one address remains*

>>> session.query(Address).filter(

... Address.email\_address.in\_(['jack@google.com', ['j25@yahoo.com'])](mailto:'j25@yahoo.com']))

... ).count()

1

DELETE FROM addresses WHERE addresses.id = ?

(2,)

SELECT count(\*) AS count\_1

FROM (SELECT addresses.id AS addresses\_id,

addresses.email\_address AS addresses\_email\_address,

addresses.user\_id AS addresses\_user\_id

FROM addresses

WHERE addresses.email\_address IN (?, ?)) AS anon\_1

('jack@google.com', 'j25@yahoo.com')

Deleting Jack will delete both Jack and the remaining Address associated with the user:

>>> session.delete(jack)

>>> session.query(User).filter\_by(name='jack').count()

DELETE FROM addresses WHERE addresses.id = ?

(1,)

DELETE FROM users WHERE users.id = ?

(5,)

SELECT count(\*) AS count\_1

FROM (SELECT users.id AS users\_id,

users.name AS users\_name,

users.fullname AS users\_fullname,

users.password AS users\_password

FROM users

WHERE users.name = ?) AS anon\_1

('jack',)

0

>>> session.query(Address).filter(

... Address.email\_address.in\_(['jack@google.com', ['j25@yahoo.com'])](mailto:'j25@yahoo.com']))

... ).count()

0

SELECT count(\*) AS count\_1

FROM (SELECT addresses.id AS addresses\_id,

addresses.email\_address AS addresses\_email\_address,

addresses.user\_id AS addresses\_user\_id

FROM addresses

WHERE addresses.email\_address IN (?, ?)) AS anon\_1

('jack@google.com', 'j25@yahoo.com')

0

**More on Cascades**

Further detail on configuration of cascades is at [Cascades](http://docs.sqlalchemy.org/en/rel_1_1/orm/cascades.html" \l "unitofwork-cascades). The cascade functionality can also integrate smoothly with the ON DELETE CASCADE functionality of the relational database. See [Using Passive Deletes](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "passive-deletes) for details.

级联配置的更多细节是Cascades。 级联功能还可以顺利地与关系数据库的ON DELETE CASCADE功能集成。 有关详细信息，请参阅使用被动删除。

1.15 Building a Many To Many Relationship

We're moving into the bonus round here, but lets show off a many-to-many relationship. We'll sneak in some other features too, just to take a tour. We'll make our application a blog application, where users can write BlogPost items, which have Keyword items associated with them.

我们正在这里进入奖金，但让我们展现出多对多的关系。 我们也会偷偷摸摸其他的功能，只是为了参观。 我们将使我们的应用程序成为一个博客应用程序，用户可以在其中编写与其相关联的Keyword项的BlogPost项目。

For a plain many-to-many, we need to create an un-mapped [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) construct to serve as the association table. This looks like the following:

对于一个简单的多对多，我们需要创建一个未映射的[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)结构作为关联表。 这看起来如下：

**>>> from sqlalchemy import** Table, Text

**>>>** *# association table*

**>>>** post\_keywords = Table('post\_keywords', Base.metadata

,**...** Column('post\_id', ForeignKey('posts.id'), primary\_key=**True**)

,**...** Column('keyword\_id', ForeignKey('keywords.id'), primary\_key=**True**)

**...** )

Above, we can see declaring a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) directly is a little different than declaring a mapped class. [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) is a constructor function, so each individual [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) argument is separated by a comma. The [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) object is also given its name explicitly, rather than it being taken from an assigned attribute name.

以上，我们可以看到直接声明一个[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)与声明一个映射类有一点不同。 [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)是一个构造函数，所以每个[Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column)参数都用逗号分隔。 [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column)对象也被明确地赋予它的名称，而不是从被赋值的属性名称获取。

Next we define BlogPost and Keyword, using complementary [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) constructs, each referring to the post\_keywords table as an association table:

接下来，我们使用互补[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)构造定义BlogPost和Keyword，每个引用作为关联表的post\_keywords表：

**>>> class** **BlogPost**(Base):

**...**  \_\_tablename\_\_ = 'posts'

**...**

**...**  id = Column(Integer, primary\_key=**True**)

**...**  user\_id = Column(Integer, ForeignKey('users.id'))

**...**  headline = Column(String(255), nullable=**False**)

**...**  body = Column(Text)

**...**

**...**  *# many to many BlogPost<->Keyword*

**...**  keywords = relationship('Keyword',

**...**  secondary=post\_keywords,

**...**  back\_populates='posts')

**...**

**...**  **def** \_\_init\_\_(self, headline, body, author):

**...**  self.author = author

**...**  self.headline = headline

**...**  self.body = body

**...**

**...**  **def** \_\_repr\_\_(self):

**...**  **return** "BlogPost(*%r*, *%r*, *%r*)" % (self.headline, self.body, self.author)

**>>> class** **Keyword**(Base):

**...**  \_\_tablename\_\_ = 'keywords'

**...**

**...**  id = Column(Integer, primary\_key=**True**)

**...**  keyword = Column(String(50), nullable=**False**, unique=**True**)

**...**  posts = relationship('BlogPost',

**...**  secondary=post\_keywords,

**...**  back\_populates='keywords')

**...**

**...**  **def** \_\_init\_\_(self, keyword):

**...**  self.keyword = keyword

**Note**

The above class declarations illustrate explicit

\_\_init\_\_() methods. Remember, when using Declarative, it's optional!

上面的类声明说明了显式的\_\_init\_\_()方法。记住，当使用Declarative时，它是可选的！

Above, the many-to-many relationship is BlogPost.keywords. The defining feature of a many-to-many relationship is the secondary keyword argument which references a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object representing the association table. This table only contains columns which reference the two sides of the relationship; if it has *any* other columns, such as its own primary key, or foreign keys to other tables, SQLAlchemy requires a different usage pattern called the "association object", described at[Association Object](http://docs.sqlalchemy.org/en/rel_1_1/orm/basic_relationships.html" \l "association-pattern).

以上，多对多关系是BlogPost.keywords。多对多关系的定义特征是引用表示关联表的[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)对象的secondary关键字参数。该表仅包含引用双方关系的列;如果它具有任何其他列(例如其自己的主键)或其他表的外键，则SQLAlchemy需要一种称为"关联对象"的不同使用模式，在"协会对象"中描述。

We would also like our BlogPost class to have an author field. We will add this as another bidirectional relationship, except one issue we'll have is that a single user might have lots of blog posts. When we access User.posts, we'd like to be able to filter results further so as not to load the entire collection. For this we use a setting accepted by [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) called lazy='dynamic', which configures an alternate ****loader strategy**** on the attribute:

我们也希望我们的BlogPost类有一个author字段。我们会将其作为另一个双向关系，除了一个问题我们将会有一个用户可能有很多博客文章。当我们访问User.posts时，我们希望能够进一步过滤结果，以便不加载整个集合。为此，我们使用被称为lazy='dynamic'的[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)接受的设置，该属性配置了替代加载器策略：

>>> BlogPost.author = relationship(User, back\_populates="posts")

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

Create new tables:

>>> Base.metadata.create\_all(engine)

PRAGMA...

CREATE TABLE keywords (

id INTEGER NOT NULL,

keyword VARCHAR(50) NOT NULL,

PRIMARY KEY (id),

UNIQUE (keyword)

)

()

COMMIT

CREATE TABLE posts (

id INTEGER NOT NULL,

user\_id INTEGER,

headline VARCHAR(255) NOT NULL,

body TEXT,

PRIMARY KEY (id),

FOREIGN KEY(user\_id) REFERENCES users (id)

)

()

COMMIT

CREATE TABLE post\_keywords (

post\_id INTEGER NOT NULL,

keyword\_id INTEGER NOT NULL,

PRIMARY KEY (post\_id, keyword\_id),

FOREIGN KEY(post\_id) REFERENCES posts (id),

FOREIGN KEY(keyword\_id) REFERENCES keywords (id)

)

()

COMMIT

Usage is not too different from what we've been doing. Let's give Wendy some blog posts:

用法与我们一直在做的不太一样。 让我们给Wendy 一些博文：

>>> wendy = session.query(User).\

... filter\_by(name='wendy').\

... one()

SELECT users.id AS users\_id,

users.name AS users\_name,

users.fullname AS users\_fullname,

users.password AS users\_password

FROM users

WHERE users.name = ?

('wendy',)

>>> post = BlogPost("Wendy's Blog Post", "This is a test", wendy)

>>> session.add(post)

We're storing keywords uniquely in the database, but we know that we don't have any yet, so we can just create them:

我们正在数据库中存储关键字，但是我们知道我们还没有，所以我们可以创建它们：

>>> post.keywords.append(Keyword('wendy'))

>>> post.keywords.append(Keyword('firstpost'))

We can now look up all blog posts with the keyword 'firstpost'. We'll use the any operator to locate "blog posts where any of its keywords has the keyword string 'firstpost'":

>>> session.query(BlogPost).\

... filter(BlogPost.keywords.any(keyword='firstpost')).\

... all()

[BlogPost("Wendy's Blog Post", 'This is a test', <User(name='wendy', fullname='Wendy Williams', password='foobar')>)]

INSERT INTO keywords (keyword) VALUES (?)

('wendy',)

INSERT INTO keywords (keyword) VALUES (?)

('firstpost',)

INSERT INTO posts (user\_id, headline, body) VALUES (?, ?, ?)

(2, "Wendy's Blog Post", 'This is a test')

INSERT INTO post\_keywords (post\_id, keyword\_id) VALUES (?, ?)

(...)

SELECT posts.id AS posts\_id,

posts.user\_id AS posts\_user\_id,

posts.headline AS posts\_headline,

posts.body AS posts\_body

FROM posts

WHERE EXISTS (SELECT 1

FROM post\_keywords, keywords

WHERE posts.id = post\_keywords.post\_id

AND keywords.id = post\_keywords.keyword\_id

AND keywords.keyword = ?)

('firstpost',)

If we want to look up posts owned by the user wendy, we can tell the query to narrow down to that User object as a parent:

>>> session.query(BlogPost).\

... filter(BlogPost.author==wendy).\

... filter(BlogPost.keywords.any(keyword='firstpost')).\

... all()

SELECT posts.id AS posts\_id,

posts.user\_id AS posts\_user\_id,

posts.headline AS posts\_headline,

posts.body AS posts\_body

FROM posts

WHERE ? = posts.user\_id AND (EXISTS (SELECT 1

FROM post\_keywords, keywords

WHERE posts.id = post\_keywords.post\_id

AND keywords.id = post\_keywords.keyword\_id

AND keywords.keyword = ?))

(2, 'firstpost')

[BlogPost("Wendy's Blog Post", 'This is a test', <User(name='wendy', fullname='Wendy Williams', password='foobar')>)]

Or we can use Wendy's own posts relationship, which is a "dynamic" relationship, to query straight from there:

>>> wendy.posts.\

... filter(BlogPost.keywords.any(keyword='firstpost')).\

... all()

SELECT posts.id AS posts\_id,

posts.user\_id AS posts\_user\_id,

posts.headline AS posts\_headline,

posts.body AS posts\_body

FROM posts

WHERE ? = posts.user\_id AND (EXISTS (SELECT 1

FROM post\_keywords, keywords

WHERE posts.id = post\_keywords.post\_id

AND keywords.id = post\_keywords.keyword\_id

AND keywords.keyword = ?))

(2, 'firstpost')

[BlogPost("Wendy's Blog Post", 'This is a test', <User(name='wendy', fullname='Wendy Williams', password='foobar')>)]

## 1.16 Further Reference

Query Reference: query\_api\_toplevel

Mapper Reference: [Mapper Configuration](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapper_config.html)

Relationship Reference: [Relationship Configuration](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationships.html)

Session Reference: [Using the Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session.html)

# 

# **Chapter 2 Mapper Configuration**

This section describes a variety of configurational patterns that are usable with mappers. It assumes you've worked through Object Relational Tutorial and know how to construct and use rudimentary mappers and relationships.

本节介绍可用于映射器的各种配置模式。 它假设您已经通过对象关系教程工作，并且知道如何构建和使用初步的映射器和关系。

## 2.1 Types of Mappings

Modern SQLAlchemy features two distinct styles of mapper configuration. The "Classical" style is SQLAlchemy's original mapping API, whereas "Declarative" is the richer and more succinct system that builds on top of "Classical". Both styles may be used interchangeably, as the end result of each is exactly the same - a user-defined class mapped by the [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper) function onto a selectable unit, typically a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table).

现代SQLAlchemy具有两种截然不同的映射器配置。 "经典"风格是SQLAlchemy的原始映射API，而"声明式"是建立在"经典"之上的更加丰富和更简洁的系统。 两种样式可以互换使用，因为每个样式的最终结果完全相同 - 由[mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper)函数映射到可选单元(通常为[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table))的用户定义类。

### 2.1.1Declarative Mapping

The *Declarative Mapping* is the typical way that mappings are constructed in modern SQLAlchemy. Making use of the [Declarative](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/index.html) system, the components of the user-defined class as well as the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) metadata to which the class is mapped are defined at once:

声明式映射是在现代SQLAlchemy中构建映射的典型方式。 使用Declarative系统，定义类的组件以及类映射到的[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)元数据：

**from sqlalchemy.ext.declarative import** declarative\_base

**from sqlalchemy import** Column, Integer, String, ForeignKey

Base = declarative\_base()

**class User**(Base):

\_\_tablename\_\_ = 'user'

id = Column(Integer, primary\_key=**True**)

name = Column(String)

fullname = Column(String)

password = Column(String)

Above, a basic single-table mapping with four columns. Additional attributes, such as relationships to other mapped classes, are also declared inline within the class definition:

以上，一个基本的单表映射四列。 其他属性，例如与其他映射类的关系，也在类定义内部内联声明：

**class** **User**(Base):

\_\_tablename\_\_ = 'user'

id = Column(Integer, primary\_key=**True**)

name = Column(String)

fullname = Column(String)

password = Column(String)

addresses = relationship("Address", backref="user", order\_by="Address.id")

**class** **Address**(Base):

\_\_tablename\_\_ = 'address'

id = Column(Integer, primary\_key=**True**)

user\_id = Column(ForeignKey('user.id'))

email\_address = Column(String)

The declarative mapping system is introduced in the [Object Relational Tutorial](http://docs.sqlalchemy.org/en/rel_1_1/orm/tutorial.html). For additional details on how this system works, see [Declarative](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/index.html).

声明性映射系统在对象关系教程中引入。 有关此系统如何工作的其他详细信息，请参阅[Declarative](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/index.html)。

### 2.1.2 Classical Mappings

A *Classical Mapping* refers to the configuration of a mapped class using the [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper) function, without using the Declarative system. This is SQLAlchemy's original class mapping API, and is still the base mapping system provided by the ORM.

经典映射是指使用[mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper)函数配置映射类，而不使用Declarative系统。 这是SQLAlchemy的原始类映射API，而且仍然是由ORM提供的基本映射系统。

In "classical" form, the table metadata is created separately with the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) construct, then associated with the User class via the [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper) function:

在“经典”表单中，表元数据是与表结构单独创建的，然后通过mapper()函数与User类相关联：

**from** **sqlalchemy** **import** Table, MetaData, Column, Integer, String, ForeignKey

**from** **sqlalchemy.orm** **import** mapper

metadata = MetaData()

user = Table('user', metadata,

Column('id', Integer, primary\_key=**True**),

Column('name', String(50)),

Column('fullname', String(50)),

Column('password', String(12))

)

**class** **User**(object):

**def** \_\_init\_\_(self, name, fullname, password):

self.name = name

self.fullname = fullname

self.password = password

mapper(User, user)

Information about mapped attributes, such as relationships to other classes, are provided via the properties dictionary. The example below illustrates a second [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object, mapped to a class called Address, then linked to User via [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship):

关于映射属性的信息，如与其他类的关系，通过属性字典提供。 下面的例子说明了第二个[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)对象，映射到一个名为Address的类，然后通过[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)链接到User ：

address = Table('address', metadata,

Column('id', Integer, primary\_key=**True**),

Column('user\_id', Integer, ForeignKey('user.id')),

Column('email\_address', String(50))

)

mapper(User, user, properties={

'addresses' : relationship(Address, backref='user', order\_by=address.c.id)})

mapper(Address, address)

When using classical mappings, classes must be provided directly without the benefit of the "string lookup" system provided by Declarative. SQL expressions are typically specified in terms of the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects, i.e. address.c.id above for the Address relationship, and not Address.id, as Address may not yet be linked to table metadata, nor can we specify a string here.

当使用经典映射时，必须直接提供类，而不需要声明式提供的“字符串查找”系统的好处。 SQL表达式通常按照[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)对象进行指定，例如Address对应的address.c.id，而不是Address.id，因为Address可能还没有链接到表元数据，也不能在这里指定一个字符串。

Some examples in the documentation still use the classical approach, but note that the classical as well as Declarative approaches are ****fully interchangeable****. Both systems ultimately create the same configuration, consisting of a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table), user-defined class, linked together with a [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper). When we talk about "the behavior of [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper)", this includes when using the Declarative system as well - it's still used, just behind the scenes.

文档中的一些例子仍然使用经典方法，但请注意，经典和声明方法是完全可互换的。 两个系统最终都会创建一个由[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)，用户定义的类，与[mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper)连接在一起的相同配置。 当我们谈论“[mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper)的行为”时，这包括当使用声明系统时，它仍然在后面使用。

### 2.1.3 Runtime Introspection of Mappings, Objects映射，对象的运行时内在检查

The [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) object is available from any mapped class, regardless of method, using the [Runtime Inspection API](http://docs.sqlalchemy.org/en/rel_1_1/core/inspection.html) system. Using the [inspect()](http://docs.sqlalchemy.org/en/rel_1_1/core/inspection.html" \l "sqlalchemy.inspection.inspect" \o "sqlalchemy.inspection.inspect) function, one can acquire the [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) from a mapped class:

使用[Runtime Inspection API](http://docs.sqlalchemy.org/en/rel_1_1/core/inspection.html)系统，无论方法如何，[Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper)对象都可以从任何映射类获得。 使用[inspect()](http://docs.sqlalchemy.org/en/rel_1_1/core/inspection.html" \l "sqlalchemy.inspection.inspect" \o "sqlalchemy.inspection.inspect)函数，可以从映射类获取[Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper)：

**>>> from** **sqlalchemy** **import** inspect

**>>>** insp = inspect(User)

Detailed information is available including [Mapper.columns](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper.columns" \o "sqlalchemy.orm.mapper.Mapper.columns):

详细信息包括Mapper.columns：

**>>>**insp.columns

<sqlalchemy.util.\_collections.OrderedProperties object at 0x102f407f8>

This is a namespace that can be viewed in a list format or via individual names:

这是一个可以列表格式或通过个别名称查看的命名空间：

**>>>** list(insp.columns)

[Column('id', Integer(), table=<user>, primary\_key=True, nullable=False),

Column('name', String(length=50), table=<user>),

Column('fullname', String(length=50), table=<user>), Column('password', String(length=12), table=<user>)]

**>>>** insp.columns.name

Column('name', String(length=50), table=<user>)

Other namespaces include [Mapper.all\_orm\_descriptors](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper.all_orm_descriptors" \o "sqlalchemy.orm.mapper.Mapper.all_orm_descriptors), which includes all mapped attributes as well as hybrids, association proxies:

其他命名空间包括Mapper.all\_orm\_descriptors，其中包括所有映射属性以及混合，关联代理：

**>>>**insp.all\_orm\_descriptors

<sqlalchemy.util.\_collections.ImmutableProperties object at 0x1040e2c68>

**>>>** insp.all\_orm\_descriptors.keys()

['fullname', 'password', 'name', 'id']

As well as [Mapper.column\_attrs](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper.column_attrs" \o "sqlalchemy.orm.mapper.Mapper.column_attrs):

**>>>** list(insp.column\_attrs)

[<ColumnProperty at 0x10403fde0; id>,

<ColumnProperty at 0x10403fce8; name>,

<ColumnProperty at 0x1040e9050; fullname>,

<ColumnProperty at 0x1040e9148; password>]

**>>>** insp.column\_attrs.name

<ColumnProperty at 0x10403fce8; name>

**>>>** insp.column\_attrs.name.expression

Column('name', String(length=50), table=<user>)

**See also**

[Runtime Inspection API](http://docs.sqlalchemy.org/en/rel_1_1/core/inspection.html)

[Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper)

[InstanceState](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState" \o "sqlalchemy.orm.state.InstanceState)

## 2.2 Mapping Columns and Expressions

The following sections discuss how table columns and SQL expressions are mapped to individual object attributes.

以下部分讨论如何将表的列和SQL表达式映射到各个对象属性。

### 2.2.1 Mapping Table Columns

The default behavior of [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper) is to assemble all the columns in the mapped [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) into mapped object attributes, each of which are named according to the name of the column itself (specifically, the key attribute of [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column)). This behavior can be modified in several ways.

[mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper)的默认行为是将映射[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)中的所有列组合到映射的对象属性中，每个列都根据列本身的名称(具体来说是[Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column)的key属性)命名。 可以通过多种方式修改此行为。

**Naming Columns Distinctly from Attribute Names**

A mapping by default shares the same name for a [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) as that of the mapped attribute - specifically it matches the Column.key attribute on [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column), which by default is the same as the Column.name.

默认情况下，映射将与[Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column)映射属性具有相同的名称，特别是与[Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) 的Column.key属性匹配，该属性默认与Column.name相同。

The name assigned to the Python attribute which maps to [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) can be different from either Column.name or Column.key just by assigning it that way, as we illustrate here in a Declarative mapping:

分配给映射到Column的Python属性的名称可以与Column.name或Column.key不同，只需通过以这种方式分配即可，如我们在声明映射中所示：

**class** **User**(Base):

\_\_tablename\_\_ = 'user'

id = Column('user\_id', Integer, primary\_key=**True**)

name = Column('user\_name', String(50))

Where above User.id resolves to a column named user\_id and User.name resolves to a column named user\_name.

上面的User.id解析为一个名为user\_id的列，User.name解析为一个名为user\_name的列。

When mapping to an existing table, the [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) object can be referenced directly:

当映射到现有的表时，可以直接引用Column对象：

**class** **User**(Base):

\_\_table\_\_ = user\_table

id = user\_table.c.user\_id

name = user\_table.c.user\_name

Or in a classical mapping, placed in the properties dictionary with the desired key:

或者在经典的映射中，放置在具有所需键的属性字典中：

mapper(User, user\_table, properties={

'id': user\_table.c.user\_id,

'name': user\_table.c.user\_name,})

In the next section we'll examine the usage of .key more closely.

在下一节中，我们将更仔细地研究.key的用法。

**Automating Column Naming Schemes from Reflected Tables**

In the previous section [Naming Columns Distinctly from Attribute Names](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_columns.html" \l "mapper-column-distinct-names), we showed how a [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) explicitly mapped to a class can have a different attribute name than the column. But what if we aren't listing out [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects explicitly, and instead are automating the production of [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects using reflection (e.g. as described in [Reflecting Database Objects](http://docs.sqlalchemy.org/en/rel_1_1/core/reflection.html))? In this case we can make use of the [DDLEvents.column\_reflect()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.DDLEvents.column_reflect" \o "sqlalchemy.events.DDLEvents.column_reflect) event to intercept the production of [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects and provide them with the Column.key of our choice:

在上一节中，命名列与属性名称不同，我们展示了如何将[Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column)显式映射到类可以具有与列不同的属性名称。 但是，如果我们没有明确列出[Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column)对象，而是使用反射自动生成[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)对象(例如反映数据库对象中所述)？ 在这种情况下，我们可以利用[DDLEvents.column\_reflect()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.DDLEvents.column_reflect" \o "sqlalchemy.events.DDLEvents.column_reflect)事件拦截[Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column)对象的生成，并为它们提供我们选择的Column.key：

**@event**.listens\_for(Table, "column\_reflect")

**def** column\_reflect(inspector, table, column\_info):

*# set column.key = "attr\_<lower\_case\_name>"*

column\_info['key'] = "attr\_*%s*" % column\_info['name'].lower()

With the above event, the reflection of [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects will be intercepted with our event that adds a new ".key" element, such as in a mapping as below:

使用上述事件，[Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column)对象的反射将被拦截，我们的事件会添加一个新的“.key”元素，例如映射如下：

**class** **MyClass**(Base):

\_\_table\_\_ = Table("some\_table", Base.metadata,

autoload=**True**, autoload\_with=some\_engine)

If we want to qualify our event to only react for the specific [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) object above, we can check for it in our event:

如果我们要限定我们的事件只对上述特定的MetaData对象做出反应，我们可以在我们的事件中检查它：

**@event**.listens\_for(Table, "column\_reflect")

**def** column\_reflect(inspector, table, column\_info):

**if** table.metadata **is** Base.metadata:

*# set column.key = "attr\_<lower\_case\_name>"*

column\_info['key'] = "attr\_*%s*" % column\_info['name'].lower()

**Naming All Columns with a Prefix**

A quick approach to prefix column names, typically when mapping to an existing [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object, is to use column\_prefix:

通常在映射到现有[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)对象时，使用column\_prefix来快速前缀列名称：

**class** **User**(Base):

\_\_table\_\_ = user\_table

\_\_mapper\_args\_\_ = {'column\_prefix':'\_'}

The above will place attribute names such as \_user\_id, \_user\_name, \_password etc. on the mapped User class.

以上将在映射的User类上放置诸如\_user\_id, \_user\_name, \_password等属性名称。

This approach is uncommon in modern usage. For dealing with reflected tables, a more flexible approach is to use that described in [Automating Column Naming Schemes from Reflected Tables](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_columns.html" \l "mapper-automated-reflection-schemes).

这种方法在现代使用中是不常见的。 为了处理反映的表，更灵活的方法是使用从反射表自动列命名方案中描述的方法。

**Using column\_property for column level options**

Options can be specified when mapping a [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) using the [column\_property()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_columns.html" \l "sqlalchemy.orm.column_property" \o "sqlalchemy.orm.column_property) function. This function explicitly creates the [ColumnProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.ColumnProperty" \o "sqlalchemy.orm.properties.ColumnProperty) used by the [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper) to keep track of the [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column); normally, the [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper) creates this automatically. Using [column\_property()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_columns.html" \l "sqlalchemy.orm.column_property" \o "sqlalchemy.orm.column_property), we can pass additional arguments about how we'd like the [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) to be mapped. Below, we pass an option active\_history, which specifies that a change to this column's value should result in the former value being loaded first:

使用[column\_property()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_columns.html" \l "sqlalchemy.orm.column_property" \o "sqlalchemy.orm.column_property)函数映射[Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) 时，可以指定选项。 此函数显式创建由[mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper)使用的[ColumnProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.ColumnProperty" \o "sqlalchemy.orm.properties.ColumnProperty)来跟踪[Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column); 通常，[mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper)会自动创建。 使用[column\_property()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_columns.html" \l "sqlalchemy.orm.column_property" \o "sqlalchemy.orm.column_property)，我们可以传递关于我们如何映射[Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column)的其他参数。 下面，我们传递一个选项active\_history，它指定对此列的值的更改应该导致此前的值先被加载：

**from** **sqlalchemy.orm** **import** column\_property

**class** **User**(Base):

\_\_tablename\_\_ = 'user'

id = Column(Integer, primary\_key=**True**)

name = column\_property(Column(String(50)), active\_history=**True**)

[column\_property()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_columns.html" \l "sqlalchemy.orm.column_property" \o "sqlalchemy.orm.column_property) is also used to map a single attribute to multiple columns. This use case arises when mapping to a [join()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.join" \o "sqlalchemy.sql.expression.join) which has attributes which are equated to each other:

[column\_property()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_columns.html" \l "sqlalchemy.orm.column_property" \o "sqlalchemy.orm.column_property)也用于将单个属性映射到多个列。 当映射到具有彼此等价的属性的[join()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.join" \o "sqlalchemy.sql.expression.join)时，会出现此用例：

**class** **User**(Base):

\_\_table\_\_ = user.join(address)

*# assign "user.id", "address.user\_id" to the*

*# "id" attribute*

id = column\_property(user\_table.c.id, address\_table.c.user\_id)

For more examples featuring this usage, see [Mapping a Class against Multiple Tables](http://docs.sqlalchemy.org/en/rel_1_1/orm/nonstandard_mappings.html" \l "maptojoin).

有关此用法的更多示例，请参阅对多个表映射类。

Another place where [column\_property()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_columns.html" \l "sqlalchemy.orm.column_property" \o "sqlalchemy.orm.column_property) is needed is to specify SQL expressions as mapped attributes, such as below where we create an attribute fullname that is the string concatenation of the firstname and lastname columns:

需要[column\_property()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_columns.html" \l "sqlalchemy.orm.column_property" \o "sqlalchemy.orm.column_property)的另一个地方是将SQL表达式指定为映射属性，例如下面我们创建一个属性fullname是firstname 和lastname列的字符串连接：

**class** **User**(Base):

\_\_tablename\_\_ = 'user'

id = Column(Integer, primary\_key=**True**)

firstname = Column(String(50))

lastname = Column(String(50))

fullname = column\_property(firstname + " " + lastname)

See examples of this usage at [SQL Expressions as Mapped Attributes](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapped_sql_expr.html" \l "mapper-sql-expressions).

sqlalchemy.orm.**column\_property**(*\*columns*, *\*\*kwargs*)

Provide a column-level property for use with a Mapper.

提供用于Mapper的列级属性。

Column-based properties can normally be applied to the mapper's properties dictionary using the [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) element directly. Use this function when the given column is not directly present within the mapper's selectable; examples include SQL expressions, functions, and scalar SELECT queries.

基于列的属性通常可以直接应用于使用[Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column)元素的映射properties字典。 当给定的列不直接存在于映射器的可选择内容中时使用此功能; 示例包括SQL表达式，函数和标量SELECT查询。

Columns that aren't present in the mapper's selectable won't be persisted by the mapper and are effectively "read-only" attributes.

映射器可选择中不存在的列将不会被映射器持久化，并且实际上是"只读"属性。

|  |  |
| --- | --- |
| **Parameters:** | * ****\*cols**** – list of Column objects to be mapped.要映射的Column对象的列表。 * ****active\_history=False**** –When True, indicates that the "previous" value for a scalar attribute should be loaded when replaced, if not already loaded. Normally, history tracking logic for simple non-primary-key scalar values only needs to be aware of the "new" value in order to perform a flush. This flag is available for applications that make use of [attributes.get\_history()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.attributes.get_history" \o "sqlalchemy.orm.attributes.get_history) or [Session.is\_modified()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.is_modified" \o "sqlalchemy.orm.session.Session.is_modified) which also need to know the "previous" value of the attribute.当为True时，表示在替换时应加载标量属性的“上一个”值，如果尚未加载。 通常，简单非主键标量值的历史跟踪逻辑仅需要注意“新”值才能执行刷新。 该标志适用于使用[attributes.get\_history()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.attributes.get_history" \o "sqlalchemy.orm.attributes.get_history)或[Session.is\_modified()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.is_modified" \o "sqlalchemy.orm.session.Session.is_modified)的应用程序，该对象也需要知道属性的“previous”值。   *New in version 0.6.6.*   * ****comparator\_factory**** – a class which extends [ColumnProperty.Comparator](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.ColumnProperty.Comparator" \o "sqlalchemy.orm.properties.ColumnProperty.Comparator) which provides custom SQL clause generation for comparison operations.一个扩展[ColumnProperty.Comparator](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.ColumnProperty.Comparator" \o "sqlalchemy.orm.properties.ColumnProperty.Comparator)的类，它为比较操作提供了自定义SQL子句生成。 * ****group**** – a group name for this property when marked as deferred.标记为延期的此属性的组名称。 * ****deferred**** – when True, the column property is "deferred", meaning that it does not load immediately, and is instead loaded when the attribute is first accessed on an instance. See also [deferred()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "sqlalchemy.orm.deferred" \o "sqlalchemy.orm.deferred).当为True时，列属性为“延迟”，这意味着它不会立即加载，而是在实例上首次访问该属性时加载该属性。 另请参见[deferred()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "sqlalchemy.orm.deferred" \o "sqlalchemy.orm.deferred)。 * ****doc**** – optional string that will be applied as the doc on the class-bound descriptor.将用作类绑定描述符上的文档的可选字符串。 * ****expire\_on\_flush=True**** –Disable expiry on flush. A column\_property() which refers to a SQL expression (and not a single table-bound column) is considered to be a "read only" property; populating it has no effect on the state of data, and it can only return database state. For this reason a column\_property()'s value is expired whenever the parent object is involved in a flush, that is, has any kind of "dirty" state within a flush. Setting this parameter to False will have the effect of leaving any existing value present after the flush proceeds. Note however that the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) with default expiration settings still expires all attributes after a [Session.commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.commit" \o "sqlalchemy.orm.session.Session.commit) call, however.禁用flush过期。 引用SQL表达式(而不是单个表格列)的column\_property()被认为是“只读”属性; 填充它对数据的状态没有影响，它只能返回数据库状态。 由于这个原因，只要父对象涉及到一个flush，column\_property()的值就会到期，也就是在flush中有任何类型的“脏”状态。 将此参数设置为False将具有在刷新进行后留下任何现有值的效果。 但请注意，具有默认过期设置的[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)仍然会在[Session.commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.commit" \o "sqlalchemy.orm.session.Session.commit)调用之后过期所有属性。   *New in version 0.7.3.*   * ****info –****Optional data dictionary which will be populated into the [MapperProperty.info](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "MapperProperty.info" \o "MapperProperty.info) attribute of this object.可选的数据字典，其将被填充到该对象的[MapperProperty.info](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "MapperProperty.info" \o "MapperProperty.info)属性中。   *New in version 0.8.*   * ****~~extension~~****~~– an [AttributeExtension](http://docs.sqlalchemy.org/en/rel_1_1/orm/deprecated.html" \l "sqlalchemy.orm.interfaces.AttributeExtension" \o "sqlalchemy.orm.interfaces.AttributeExtension) instance, or list of extensions, which will be prepended to the list of attribute listeners for the resulting descriptor placed on the class.~~****~~Deprecated.~~****~~Please see [AttributeEvents](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.AttributeEvents" \o "sqlalchemy.orm.events.AttributeEvents).一个AttributeExtension实例或扩展名列表，它将被添加到属性侦听器的列表中，用于在类上生成的描述符。已过时。 请参阅AttributeEvents。~~ |

**Mapping a Subset of Table Columns**

Sometimes, a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object was made available using the reflection process described at [Reflecting Database Objects](http://docs.sqlalchemy.org/en/rel_1_1/core/reflection.html" \l "metadata-reflection) to load the table's structure from the database. For such a table that has lots of columns that don't need to be referenced in the application, the include\_properties or exclude\_properties arguments can specify that only a subset of columns should be mapped. For example:

有时，使用“反射数据库对象”中描述的反射过程可以使用Table对象从数据库加载表的结构。 对于具有不需要在应用程序中引用的列的这样一个表，include\_properties或exclude\_properties参数可以指定只应该映射列的一个子集。 例如：

**class** **User**(Base):

\_\_table\_\_ = user\_table

\_\_mapper\_args\_\_ = {

'include\_properties' :['user\_id', 'user\_name']

}

…will map the User class to the user\_table table, only including the user\_id and user\_name columns - the rest are not referenced. Similarly:

...将User类映射到user\_table表，只包括user\_id和user\_name列 - 其余不被引用。同理：

**class** **Address**(Base):

\_\_table\_\_ = address\_table

\_\_mapper\_args\_\_ = {

'exclude\_properties' : ['street', 'city', 'state', 'zip']

}

…will map the Address class to the address\_table table, including all columns present except street, city, state, and zip.

...将地址类映射到address\_table表，包括除街道，城市，州和邮政编码之外的所有列。

When this mapping is used, the columns that are not included will not be referenced in any SELECT statements emitted by [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query), nor will there be any mapped attribute on the mapped class which represents the column; assigning an attribute of that name will have no effect beyond that of a normal Python attribute assignment.

当使用此映射时，不包括的列将不会在Query发出的任何SELECT语句中引用，也不会在表示列的映射类上存在任何映射属性; 分配该名称的属性将不会超出普通Python属性分配的效果。

In some cases, multiple columns may have the same name, such as when mapping to a join of two or more tables that share some column name.include\_properties and exclude\_properties can also accommodate [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects to more accurately describe which columns should be included or excluded:

在某些情况下，多个列可能具有相同的名称，例如映射到共享一些列name.include\_properties的两个或多个表的连接时，exclude\_properties还可以容纳Column对象以更准确地描述应包含或排除哪些列：

**class** **UserAddress**(Base):

\_\_table\_\_ = user\_table.join(addresses\_table)

\_\_mapper\_args\_\_ = {

'exclude\_properties' :[address\_table.c.id],

'primary\_key' : [user\_table.c.id]

}

**Note**

insert and update defaults configured on individual [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects, i.e. those described at [Column Insert/Update Defaults](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "metadata-defaults) including those configured by the default, update,server\_default and server\_onupdate arguments, will continue to function normally even if those [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects are not mapped. This is because in the case of default and update, the [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) object is still present on the underlying [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table), thus allowing the default functions to take place when the ORM emits an INSERT or UPDATE, and in the case of server\_default and server\_onupdate, the relational database itself maintains these functions.

插入和更新在单个[Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column)对象上配置的默认值，即列插入/更新默认值(包括由default, update,server\_default和server\_onupdate参数配置的默认值)中描述的默认值，将继续正常运行，即使这些Column对象未映射。 这是因为在default和update的情况下，[Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column)对象仍然存在于底层表上，从而允许在ORM发出INSERT或UPDATE时发生默认功能，而在server\_default和server\_onupdate的情况下，关系型数据库本身维护这些功能

### 2.2.2 SQL Expressions as Mapped Attributes

Attributes on a mapped class can be linked to SQL expressions, which can be used in queries.

映射类上的属性可以链接到SQL表达式，可以在查询中使用。

Using a Hybrid

The easiest and most flexible way to link relatively simple SQL expressions to a class is to use a so-called "hybrid attribute", described in the section [Hybrid Attributes](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/hybrid.html). The hybrid provides for an expression that works at both the Python level as well as at the SQL expression level. For example, below we map a class User, containing attributes firstname and lastname, and include a hybrid that will provide for us the fullname, which is the string concatenation of the two:

将相对简单的SQL表达式链接到类的最简单和最灵活的方法是使用所谓的"混合属性"，如"混合属性"一节中所述。 该混合提供了一种在Python级别以及SQL表达式级别都可以工作的表达式。 例如，下面我们映射一个类User，其中包含firstname和lastname属性，并且包含一个将为我们提供fullname的混合，这是两个字符串的连接：

**from** **sqlalchemy.ext.hybrid** **import** hybrid\_property

**class** **User**(Base):

\_\_tablename\_\_ = 'user'

id = Column(Integer, primary\_key=**True**)

firstname = Column(String(50))

lastname = Column(String(50))

**@hybrid\_property**

**def** fullname(self):

**return** self.firstname + " " + self.lastname

Above, the fullname attribute is interpreted at both the instance and class level, so that it is available from an instance:

some\_user = session.query(User).first()

print(some\_user.fullname)

as well as usable within queries:

some\_user = session.query(User).filter(User.fullname == "John Smith").first()

The string concatenation example is a simple one, where the Python expression can be dual purposed at the instance and class level. Often, the SQL expression must be distinguished from the Python expression, which can be achieved using [hybrid\_property.expression()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/hybrid.html" \l "sqlalchemy.ext.hybrid.hybrid_property.expression" \o "sqlalchemy.ext.hybrid.hybrid_property.expression). Below we illustrate the case where a conditional needs to be present inside the hybrid, using the if statement in Python and the [sql.expression.case()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.case" \o "sqlalchemy.sql.expression.case) construct for SQL expressions:

字符串连接示例是一个简单的例子，其中Python表达式可以在实例和类级别双重选择。 通常，SQL表达式必须与Python表达式区分开来，这可以使用[hybrid\_property.expression()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/hybrid.html" \l "sqlalchemy.ext.hybrid.hybrid_property.expression" \o "sqlalchemy.ext.hybrid.hybrid_property.expression)来实现。 下面我们说明一个条件需要在混合中存在的情况，使用Python中的if语句和SQL表达式的[sql.expression.case()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.case" \o "sqlalchemy.sql.expression.case)构造：

**from** **sqlalchemy.ext.hybrid** **import** hybrid\_property

**from** **sqlalchemy.sql** **import** case

**class** **User**(Base):

\_\_tablename\_\_ = 'user'

id = Column(Integer, primary\_key=**True**)

firstname = Column(String(50))

lastname = Column(String(50))

**@hybrid\_property**

**def** fullname(self):

**if** self.firstname **is** **not** **None**:

**return** self.firstname + " " + self.lastname

**else**:

**return** self.lastname

**@fullname**.expression

**def** fullname(cls):

**return** case([

(cls.firstname != **None**, cls.firstname + " " + cls.lastname),

], else\_ = cls.lastname)

**Using column\_property**

The [orm.column\_property()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_columns.html" \l "sqlalchemy.orm.column_property" \o "sqlalchemy.orm.column_property) function can be used to map a SQL expression in a manner similar to a regularly mapped [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column). With this technique, the attribute is loaded along with all other column-mapped attributes at load time. This is in some cases an advantage over the usage of hybrids, as the value can be loaded up front at the same time as the parent row of the object, particularly if the expression is one which links to other tables (typically as a correlated subquery) to access data that wouldn't normally be available on an already loaded object.

[orm.column\_property()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_columns.html" \l "sqlalchemy.orm.column_property" \o "sqlalchemy.orm.column_property)函数可用于以类似于定期映射的列的方式映射SQL表达式。 使用这种技术，属性在加载时与所有其他列映射属性一起加载。 这在某些情况下比使用混合动画更有优势，因为该值可以与对象的父行同时加载，特别是如果表达式是链接到其他表的表达式(通常作为相关子查询)访问在已加载的对象上通常不可用的数据。

Disadvantages to using [orm.column\_property()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_columns.html" \l "sqlalchemy.orm.column_property" \o "sqlalchemy.orm.column_property) for SQL expressions include that the expression must be compatible with the SELECT statement emitted for the class as a whole, and there are also some configurational quirks which can occur when using [orm.column\_property()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_columns.html" \l "sqlalchemy.orm.column_property" \o "sqlalchemy.orm.column_property) from declarative mixins.

对于SQL表达式使用[orm.column\_property()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_columns.html" \l "sqlalchemy.orm.column_property" \o "sqlalchemy.orm.column_property)的缺点在于表达式必须与为整个类发布的SELECT语句兼容，并且还有一些配置问题，当使用声明性混合的[orm.column\_property()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_columns.html" \l "sqlalchemy.orm.column_property" \o "sqlalchemy.orm.column_property)。

Our "fullname" example can be expressed using [orm.column\_property()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_columns.html" \l "sqlalchemy.orm.column_property" \o "sqlalchemy.orm.column_property) as follows:

我们的"全名"示例可以使用orm.column\_property()表示如下：

**from** **sqlalchemy.orm** **import** column\_property

**class** **User**(Base):

\_\_tablename\_\_ = 'user'

id = Column(Integer, primary\_key=**True**)

firstname = Column(String(50))

lastname = Column(String(50))

fullname = column\_property(firstname + " " + lastname)

Correlated subqueries may be used as well. Below we use the [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) construct to create a SELECT that links together the count of Address objects available for a particular User:

也可以使用相关的子查询。 下面我们使用[select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select)结构来创建一个将特定User可用的Address对象的数量链接在一起的SELECT：

**from** **sqlalchemy.orm** **import** column\_property

**from** **sqlalchemy** **import** select, func

**from** **sqlalchemy** **import** Column, Integer, String, ForeignKey

**from** **sqlalchemy.ext.declarative** **import** declarative\_base

Base = declarative\_base()

**class** **Address**(Base):

\_\_tablename\_\_ = 'address'

id = Column(Integer, primary\_key=**True**)

user\_id = Column(Integer, ForeignKey('user.id'))

**class** **User**(Base):

\_\_tablename\_\_ = 'user'

id = Column(Integer, primary\_key=**True**)

address\_count = column\_property(

select([func.count(Address.id)]).\

where(Address.user\_id==id).\

correlate\_except(Address)

)

In the above example, we define a [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) construct like the following:

select([func.count(Address.id)]).\

where(Address.user\_id==id).\

correlate\_except(Address)

The meaning of the above statement is, select the count of Address.id rows where the Address.user\_id column is equated to id, which in the context of the User class is the [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) named id (note that id is also the name of a Python built in function, which is not what we want to use here - if we were outside of the User class definition, we'd use User.id).

上述语句的意思是，选择Address.id行中的Address.user\_id列等于id的计数，在User类的上下文中，它是名为id的[Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column)(请注意，id也是 Python内置函数，这不是我们想要使用的 - 如果我们在User类定义之外，我们将使用User.id)。

The select.correlate\_except() directive indicates that each element in the FROM clause of this [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) may be omitted from the FROM list (that is, correlated to the enclosing SELECT statement against User) except for the one corresponding to Address. This isn't strictly necessary, but prevents Address from being inadvertently omitted from the FROM list in the case of a long string of joins between User and Address tables where SELECT statements against Address are nested.

select.correlate\_except()指令指出，除了对应于Address的对应的FROM列表中，这个select()的FROM子句中的每一个元素都可以从FROM列表中省略(即与用户的封闭SELECT语句相关)。 这并不是绝对必要的，但是在用户和地址表之间的长串连接的情况下，阻止来自FROM列表的地址被无意中忽略，其中针对Address的SELECT语句是嵌套的。

If import issues prevent the [column\_property()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_columns.html" \l "sqlalchemy.orm.column_property" \o "sqlalchemy.orm.column_property) from being defined inline with the class, it can be assigned to the class after both are configured. In Declarative this has the effect of calling [Mapper.add\_property()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper.add_property" \o "sqlalchemy.orm.mapper.Mapper.add_property) to add an additional property after the fact:

如果导入问题阻止了[column\_property()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_columns.html" \l "sqlalchemy.orm.column_property" \o "sqlalchemy.orm.column_property)与类内联定义，则可以在配置之后将其分配给该类。 在声明式中，这具有调用[Mapper.add\_property()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper.add_property" \o "sqlalchemy.orm.mapper.Mapper.add_property)在事实之后添加一个附加属性的效果：

User.address\_count = column\_property(

select([func.count(Address.id)]).\

where(Address.user\_id==User.id)

)

For many-to-many relationships, use [and\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.and_" \o "sqlalchemy.sql.expression.and_) to join the fields of the association table to both tables in a relation, illustrated here with a classical mapping:

对于多对多关系，使用and\_()将关联表的字段连接到一个关系中的两个表，这里用经典映射说明：

**from** **sqlalchemy** **import** and\_

mapper(Author, authors, properties={

'book\_count': column\_property(

select([func.count(books.c.id)],

and\_(

book\_authors.c.author\_id==authors.c.id,

book\_authors.c.book\_id==books.c.id

)))

})

**Using a plain descriptor**

In cases where a SQL query more elaborate than what [orm.column\_property()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_columns.html" \l "sqlalchemy.orm.column_property" \o "sqlalchemy.orm.column_property) or [hybrid\_property](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/hybrid.html" \l "sqlalchemy.ext.hybrid.hybrid_property" \o "sqlalchemy.ext.hybrid.hybrid_property) can provide must be emitted, a regular Python function accessed as an attribute can be used, assuming the expression only needs to be available on an already-loaded instance. The function is decorated with Python's own @property decorator to mark it as a read-only attribute. Within the function, [object\_session()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.object_session" \o "sqlalchemy.orm.session.object_session) is used to locate the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) corresponding to the current object, which is then used to emit a query:

在SQL查询比[orm.column\_property()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_columns.html" \l "sqlalchemy.orm.column_property" \o "sqlalchemy.orm.column_property)或[hybrid\_property](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/hybrid.html" \l "sqlalchemy.ext.hybrid.hybrid_property" \o "sqlalchemy.ext.hybrid.hybrid_property)可以提供的更详细的SQL查询必须被发出的情况下，可以使用作为属性访问的常规Python函数，假设该表达式仅需要在已加载的实例上可用。 该功能使用Python自己的@property装饰器进行装饰，将其标记为只读属性。 在函数中，[object\_session()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.object_session" \o "sqlalchemy.orm.session.object_session)用于定位与当前对象相对应的[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)，然后用于发出查询：

**from** **sqlalchemy.orm** **import** object\_session

**from** **sqlalchemy** **import** select, func

**class** **User**(Base):

\_\_tablename\_\_ = 'user'

id = Column(Integer, primary\_key=**True**)

firstname = Column(String(50))

lastname = Column(String(50))

**@property**

**def** address\_count(self):

**return** object\_session(self).\

scalar(

select([func.count(Address.id)]).\

where(Address.user\_id==self.id)

)

The plain descriptor approach is useful as a last resort, but is less performant in the usual case than both the hybrid and column property approaches, in that it needs to emit a SQL query upon each access.

简单描述方法作为最后的手段是有用的，但是在通常的情况下比hybrid和column property方法性能较差，因为它需要在每次访问时发出SQL查询。

### 2.2.3 Changing Attribute Behavior

### Simple Validators

A quick way to add a "validation" routine to an attribute is to use the [validates()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapped_attributes.html" \l "sqlalchemy.orm.validates" \o "sqlalchemy.orm.validates) decorator. An attribute validator can raise an exception, halting the process of mutating the attribute's value, or can change the given value into something different. Validators, like all attribute extensions, are only called by normal userland code; they are not issued when the ORM is populating the object:

在属性中添加"验证"例程的一种快速方法是使用[validates()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapped_attributes.html" \l "sqlalchemy.orm.validates" \o "sqlalchemy.orm.validates)装饰器。 属性验证器可以引发异常，停止突变属性值的过程，或者可以将给定值更改为不同的值。 验证器与所有属性扩展一样，只能由普通用户地址代码调用; 当ORM填充对象时，它们不会被发出：

**from** **sqlalchemy.orm** **import** validates

**class** **EmailAddress**(Base):

\_\_tablename\_\_ = 'address'

id = Column(Integer, primary\_key=**True**)

email = Column(String)

**@validates**('email')

**def** validate\_email(self, key, address):

**assert** '@' **in** address

**return** address

*Changed in version 1.0.0:*- validators are no longer triggered within the flush process when the newly fetched values for primary key columns as well as some python- or server-side defaults are fetched. Prior to 1.0, validators may be triggered in those cases as well.

在版本1.0.0中更改： - 在获取主键列的新获取值以及一些python或服务器端默认值时，验证器不会在刷新过程中触发。 在1.0之前，也可以在这些情况下触发验证器。

Validators also receive collection append events, when items are added to a collection:

当项目被添加到集合中时，验证器还会收到收集附加事件：

**from** **sqlalchemy.orm** **import** validates

**class** **User**(Base):

*# ...*

addresses = relationship("Address")

**@validates**('addresses')

**def** validate\_address(self, key, address):

**assert** '@' **in** address.email

**return** address

The validation function by default does not get emitted for collection remove events, as the typical expectation is that a value being discarded doesn't require validation.

默认情况下，验证函数不会被收集移除事件，因为典型的期望是被丢弃的值不需要验证。 However, [validates()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapped_attributes.html" \l "sqlalchemy.orm.validates" \o "sqlalchemy.orm.validates) supports reception of these events by specifying include\_removes=True to the decorator. When this flag is set, the validation function must receive an additional boolean argument which if True indicates that the operation is a removal:

但是，[validates()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapped_attributes.html" \l "sqlalchemy.orm.validates" \o "sqlalchemy.orm.validates)支持通过为装饰器指定include\_removes=True来接收这些事件。 当设置此标志时，验证函数必须接收一个额外的布尔参数，如果True表示操作是一个删除：

**from** **sqlalchemy.orm** **import** validates

**class** **User**(Base):

*# ...*

addresses = relationship("Address")

**@validates**('addresses', include\_removes=**True**)

**def** validate\_address(self, key, address, is\_remove):

**if** is\_remove:

**raise** ValueError(

"not allowed to remove items from the collection")

**else**:

**assert** '@' **in** address.email

**return** address

The case where mutually dependent validators are linked via a backref can also be tailored, using the include\_backrefs=False option; this option, when set to False, prevents a validation function from emitting if the event occurs as a result of a backref:

通过backref链接相互依赖的验证器的情况也可以使用include\_backrefs=False选项进行调整; 此选项设置为False时，如果由于backref导致事件发生，则会阻止验证功能发出：

**from** **sqlalchemy.orm** **import** validates

**class** **User**(Base):

*# ...*

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

**@validates**('addresses', include\_backrefs=**False**)

**def** validate\_address(self, key, address):

**assert** '@' **in** address.email

**return** address

Above, if we were to assign to Address.user as in some\_address.user = some\_user, the validate\_address() function would *not* be emitted, even though an append occurs to some\_user.addresses - the event is caused by a backref.

以上，如果我们要像some\_address.user = some\_user那样分配给Address.user，即使附加发生到some\_user.addresses，也不会发出validate\_address()函数 - 事件是由backref引起的。

Note that the [validates()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapped_attributes.html" \l "sqlalchemy.orm.validates" \o "sqlalchemy.orm.validates) decorator is a convenience function built on top of attribute events. An application that requires more control over configuration of attribute change behavior can make use of this system, described at [AttributeEvents](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.AttributeEvents" \o "sqlalchemy.orm.events.AttributeEvents).

请注意，[validates()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapped_attributes.html" \l "sqlalchemy.orm.validates" \o "sqlalchemy.orm.validates)装饰器是建立在属性事件之上的方便函数。 需要更多控制属性更改行为配置的应用程序可以利用[AttributeEvents](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.AttributeEvents" \o "sqlalchemy.orm.events.AttributeEvents)中描述的此系统。

sqlalchemy.orm.**validates**(*\*names*, *\*\*kw*)

Decorate a method as a 'validator' for one or more named properties.

将一个方法作为一个或多个命名属性的“验证器”进行装饰。

Designates a method as a validator, a method which receives the name of the attribute as well as a value to be assigned, or in the case of a collection, the value to be added to the collection. The function can then raise validation exceptions to halt the process from continuing (where Python's built-in ValueError and AssertionError exceptions are reasonable choices), or can modify or replace the value before proceeding. The function should otherwise return the given value.

指定方法作为验证器，接收属性名称以及要分配的值的方法，或者在集合的情况下，将要添加到集合的值。 然后，该函数可以提高验证异常以停止进程(Python的内置ValueError和AssertionError异常是合理的选择)，或者可以在继续之前修改或替换该值。 该函数否则返回给定的值。

Note that a validator for a collection ****cannot**** issue a load of that collection within the validation routine - this usage raises an assertion to avoid recursion overflows. This is a reentrant condition which is not supported.

请注意，集合的验证器无法在验证例程中发出该集合的负载 - 此使用会引发断言以避免递归溢出。 这是不支持的折返条件。

|  |  |
| --- | --- |
| **Parameters:** | * ****\*names**** – list of attribute names to be validated.要验证的属性名称列表。 * ****include\_removes –****if True, "remove" events will be sent as well - the validation function must accept an additional argument "is\_remove" which will be a boolean.如果为True，“remove”事件也将被发送 - 验证函数必须接受一个额外的参数“is\_remove”，它将是一个布尔值。   *New in version 0.7.7.*   * ****include\_backrefs –****defaults to True; if False, the validation function will not emit if the originator is an attribute event related via a backref. This can be used for bi-directional[validates()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapped_attributes.html" \l "sqlalchemy.orm.validates" \o "sqlalchemy.orm.validates) usage where only one validator should emit per attribute operation.默认为True; 如果False，如果始发者是通过backref相关的属性事件，验证函数将不会发出。 这可以用于双向[validates()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapped_attributes.html" \l "sqlalchemy.orm.validates" \o "sqlalchemy.orm.validates))使用，其中只有一个验证器应发送每个属性操作。   *New in version 0.9.0.* |

**See also**

[Simple Validators](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapped_attributes.html" \l "simple-validators) - usage examples for [validates()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapped_attributes.html" \l "sqlalchemy.orm.validates" \o "sqlalchemy.orm.validates)

Using Descriptors and Hybrids

A more comprehensive way to produce modified behavior for an attribute is to use [descriptors](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-descriptors). These are commonly used in Python using the property() function. The standard SQLAlchemy technique for descriptors is to create a plain descriptor, and to have it read/write from a mapped attribute with a different name. Below we illustrate this using Python 2.6-style properties:

为属性生成修改行为的更全面的方法是使用描述符。 这些在Python中常用于使用property()函数。 用于描述符的标准SQLAlchemy技术是创建一个简单的描述符，并使其具有不同名称的映射属性进行读/写。 下面我们使用Python 2.6风格的属性来说明这一点：

**class** **EmailAddress**(Base):

\_\_tablename\_\_ = 'email\_address'

id = Column(Integer, primary\_key=**True**)

*# name the attribute with an underscore,*

*# different from the column name*

\_email = Column("email", String)

*# then create an ".email" attribute*

*# to get/set ".\_email"*

**@property**

**def** email(self):

**return** self.\_email

**@email**.setter

**def** email(self, email):

self.\_email = email

The approach above will work, but there's more we can add. While our EmailAddress object will shuttle the value through the email descriptor and into the \_email mapped attribute, the class level EmailAddress.email attribute does not have the usual expression semantics usable with [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query). To provide these, we instead use the [hybrid](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/hybrid.html" \l "module-sqlalchemy.ext.hybrid" \o "sqlalchemy.ext.hybrid) extension as follows:

上面的方法将会起作用，但是我们还可以添加更多。 虽然我们的EmailAddress对象将通过电子email描述符跳过该值并传入\_email映射属性，但类级别EmailAddress.email属性没有通用的可用于[Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query)的表达式语义。 为了提供这些，我们使用[hybrid](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/hybrid.html" \l "module-sqlalchemy.ext.hybrid" \o "sqlalchemy.ext.hybrid)扩展如下：

**from** **sqlalchemy.ext.hybrid** **import** hybrid\_property

**class** **EmailAddress**(Base):

\_\_tablename\_\_ = 'email\_address'

id = Column(Integer, primary\_key=**True**)

\_email = Column("email", String)

**@hybrid\_property**

**def** email(self):

**return** self.\_email

**@email**.setter

**def** email(self, email):

self.\_email = email

The .email attribute, in addition to providing getter/setter behavior when we have an instance of EmailAddress, also provides a SQL expression when used at the class level, that is, from the EmailAddress class directly:

除了在具有EmailAddress实例的情况下提供getter / setter行为之外，.email属性还可以在类级别(即EmailAddress类)直接使用时提供SQL表达式：

**from** **sqlalchemy.orm** **import** Session

session = Session()

address = session.query(EmailAddress).\

filter(EmailAddress.email == 'address@example.com').\

one()

SELECT address.email AS address\_email, address.id AS address\_id

FROM address

WHERE address.email = ?

('address@example.com',)

address.email = ['otheraddress@example.com'](mailto:'otheraddress@example.com')

session.commit()

UPDATE address SET email=? WHERE address.id = ?

('otheraddress@example.com', 1)

COMMIT

The [hybrid\_property](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/hybrid.html" \l "sqlalchemy.ext.hybrid.hybrid_property" \o "sqlalchemy.ext.hybrid.hybrid_property) also allows us to change the behavior of the attribute, including defining separate behaviors when the attribute is accessed at the instance level versus at the class/expression level, using the [hybrid\_property.expression()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/hybrid.html" \l "sqlalchemy.ext.hybrid.hybrid_property.expression" \o "sqlalchemy.ext.hybrid.hybrid_property.expression) modifier. Such as, if we wanted to add a host name automatically, we might define two sets of string manipulation logic:

hybrid\_property还允许我们改变属性的行为，包括使用hybrid\_property.expression()修饰符在属性被访问时在实例级与类/表达式级别上定义单独的行为。 例如，如果我们要自动添加主机名，我们可以定义两组字符串操作逻辑：

**class** **EmailAddress**(Base):

\_\_tablename\_\_ = 'email\_address'

id = Column(Integer, primary\_key=**True**)

\_email = Column("email", String)

**@hybrid\_property**

**def** email(self):

*"""Return the value of \_email up until the last twelve characters."""*

**return** self.\_email[:-12]

**@email**.setter

**def** email(self, email):

*"""Set the value of \_email, tacking on the twelve character value @example.com."""*

self.\_email = email + "@example.com"

**@email**.expression

**def** email(cls):

*"""Produce a SQL expression that represents the value of the \_email column, minus the last twelve characters."""*

**return** func.substr(cls.\_email, 0, func.length(cls.\_email) - 12)

Above, accessing the email property of an instance of EmailAddress will return the value of the \_email attribute, removing or adding the hostname @example.com from the value. When we query against the email attribute, a SQL function is rendered which produces the same effect:

以上，访问EmailAddress实例的电子邮件属性将返回\_email属性的值，从该值中删除或添加主机名@ example.com。 当我们查询电子邮件属性时，会渲染一个产生相同效果的SQL函数：

address = session.query(EmailAddress).filter(EmailAddress.email == 'address').one()

SELECT address.email AS address\_email, address.id AS address\_id

FROM address

WHERE substr(address.email, ?, length(address.email) - ?) = ?

(0, 12, 'address')

Read more about Hybrids at [Hybrid Attributes](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/hybrid.html).

Synonyms

Synonyms are a mapper-level construct that allow any attribute on a class to "mirror" another attribute that is mapped.

同义词是一个映射器级构造，允许类上的任何属性“映射”另一个映射的属性。

In the most basic sense, the synonym is an easy way to make a certain attribute available by an additional name:

在最基本的意义上，同义词是一种简单的方法，可以通过附加名称使某个属性可用：

**class** **MyClass**(Base):

\_\_tablename\_\_ = 'my\_table'

id = Column(Integer, primary\_key=**True**)

job\_status = Column(String(50))

status = synonym("job\_status")

The above class MyClass has two attributes, .job\_status and .status that will behave as one attribute, both at the expression level:

上述类MyClass有两个属性：.job\_status和.status，它们将在表达式级别上表现为一个属性：

**>>>** print(MyClass.job\_status == 'some\_status')

my\_table.job\_status = :job\_status\_1

**>>>** print(MyClass.status == 'some\_status')

my\_table.job\_status = :job\_status\_1

and at the instance level:

**>>>** m1 = MyClass(status='x')

**>>>** m1.status, m1.job\_status('x', 'x')

**>>>** m1.job\_status = 'y'

**>>>** m1.status, m1.job\_status('y', 'y')

The [synonym()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapped_attributes.html" \l "sqlalchemy.orm.synonym" \o "sqlalchemy.orm.synonym) can be used for any kind of mapped attribute that subclasses [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty), including mapped columns and relationships, as well as synonyms themselves.

同义词()可以用于任何类型的MapperProperty映射属性，包括映射列和关系，以及同义词本身。

Beyond a simple mirror, [synonym()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapped_attributes.html" \l "sqlalchemy.orm.synonym" \o "sqlalchemy.orm.synonym) can also be made to reference a user-defined [descriptor](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-descriptor). We can supply our status synonym with a @property:

除了简单的镜像之外，同义词()也可以引用用户定义的描述符。 我们可以提供一个@property的状态同义词：

**class** **MyClass**(Base):

\_\_tablename\_\_ = 'my\_table'

id = Column(Integer, primary\_key=**True**)

status = Column(String(50))

**@property**

**def** job\_status(self):

**return** "Status: " + self.status

job\_status = synonym("status", descriptor=job\_status)

When using Declarative, the above pattern can be expressed more succinctly using the [synonym\_for()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.synonym_for" \o "sqlalchemy.ext.declarative.synonym_for) decorator:

当使用Declarative时，可以使用synonym\_for()装饰器更简洁地表达上述模式：

**from** **sqlalchemy.ext.declarative** **import** synonym\_for

**class** **MyClass**(Base):

\_\_tablename\_\_ = 'my\_table'

id = Column(Integer, primary\_key=**True**)

status = Column(String(50))

**@synonym\_for**("status")

**@property**

**def** job\_status(self):

**return** "Status: " + self.status

While the [synonym()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapped_attributes.html" \l "sqlalchemy.orm.synonym" \o "sqlalchemy.orm.synonym) is useful for simple mirroring, the use case of augmenting attribute behavior with descriptors is better handled in modern usage using the [hybrid attribute](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapped_attributes.html" \l "mapper-hybrids) feature, which is more oriented towards Python descriptors. Technically, a [synonym()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapped_attributes.html" \l "sqlalchemy.orm.synonym" \o "sqlalchemy.orm.synonym) can do everything that a [hybrid\_property](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/hybrid.html" \l "sqlalchemy.ext.hybrid.hybrid_property" \o "sqlalchemy.ext.hybrid.hybrid_property) can do, as it also supports injection of custom SQL capabilities, but the hybrid is more straightforward to use in more complex situations.

虽然synonym()对于简单的镜像是有用的，但使用描述符来增强属性行为的用例在使用混合属性特征的现代使用中更好地处理，该特性更适用于Python描述符。 在技术上，同义词()可以做一个hybrid\_property可以做的一切，因为它也支持注入自定义SQL功能，但是混合在更复杂的情况下更容易使用。

sqlalchemy.orm.**synonym**(*name*, *map\_column=None*, *descriptor=None*, *comparator\_factory=None*, *doc=None*, *info=None*)

Denote an attribute name as a synonym to a mapped property, in that the attribute will mirror the value and expression behavior of another attribute.

将属性名称表示为映射属性的同义词，因为该属性将镜像另一个属性的值和表达式行为。

|  |  |
| --- | --- |
| **Parameters:** | * ****name**** – the name of the existing mapped property. This can refer to the string name of any [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty) configured on the class, including column-bound attributes and relationships.现有映射属性的名称。 这可以引用在类上配置的任何MapperProperty的字符串名称，包括列绑定的属性和关系。 * ****descriptor**** – a Python [descriptor](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-descriptor) that will be used as a getter (and potentially a setter) when this attribute is accessed at the instance level.一个Python描述符，当在实例级访问此属性时，它将被用作一个getter(并且可能是一个setter)。 * ****map\_column –****if True, the [synonym()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapped_attributes.html" \l "sqlalchemy.orm.synonym" \o "sqlalchemy.orm.synonym) construct will locate the existing named [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty) based on the attribute name of this [synonym()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapped_attributes.html" \l "sqlalchemy.orm.synonym" \o "sqlalchemy.orm.synonym), and assign it to a new attribute linked to the name of this [synonym()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapped_attributes.html" \l "sqlalchemy.orm.synonym" \o "sqlalchemy.orm.synonym). That is, given a mapping like:如果为True，则synonym()构造将基于该同义词()的属性名称来定位现有的命名的MapperProperty，并将其分配给与该同义词()的名称链接的新属性。 也就是说，给出一个映射，如：   **class** **MyClass**(Base):  \_\_tablename\_\_ = 'my\_table'  id = Column(Integer, primary\_key=**True**)  job\_status = Column(String(50))  job\_status = synonym("\_job\_status", map\_column=**True**)  The above class MyClass will now have the job\_status [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) object mapped to the attribute named \_job\_status, and the attribute named job\_status will refer to the synonym itself. This feature is typically used in conjunction with the descriptor argument in order to link a user-defined descriptor as a "wrapper" for an existing column.上面的MyClass类现在将job\_status Column对象映射到名为\_job\_status的属性，名为job\_status的属性将引用同义词本身。 该特征通常与描述符参数一起使用，以将用户定义的描述符链接为现有列的“包装器”。   * ****info –****Optional data dictionary which will be populated into the InspectionAttr.info attribute of this object.可选的数据字典将被填充到该对象的InspectionAttr.info属性中。   *New in version 1.0.0.*   * ****comparator\_factory –****A subclass of [PropComparator](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator" \o "sqlalchemy.orm.interfaces.PropComparator) that will provide custom comparison behavior at the SQL expression level.PropComparator的子类将在SQL表达式级别提供自定义的比较行为。   **Note**  For the use case of providing an attribute which redefines both Python-level and SQL-expression level behavior of an attribute, please refer to the Hybrid attribute introduced at [Using Descriptors and Hybrids](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapped_attributes.html" \l "mapper-hybrids) for a more effective technique.  对于提供重新定义属性的Python级别和SQL表达式级别行为的属性的用例，请参考使用描述符和混合器引入的混合属性，以获得更有效的技术 |

**See also**

[Synonyms](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapped_attributes.html" \l "synonyms) - examples of functionality.

[Using Descriptors and Hybrids](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapped_attributes.html" \l "mapper-hybrids) - Hybrids provide a better approach for more complicated attribute-wrapping schemes than synonyms.

使用描述符和杂交 - 杂交为比较复杂的属性包装方案提供了一种比同义词更好的方法。

Operator Customization

The "operators" used by the SQLAlchemy ORM and Core expression language are fully customizable. For example, the comparison expression User.name == 'ed'makes usage of an operator built into Python itself called operator.eq - the actual SQL construct which SQLAlchemy associates with such an operator can be modified. New operations can be associated with column expressions as well. The operators which take place for column expressions are most directly redefined at the type level - see the section [Redefining and Creating New Operators](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "types-operators) for a description.

SQLAlchemy ORM和Core表达式语言使用的“运算符”是完全可定制的。 例如，比较表达式User.name =='ed'使用内置于Python本身的操作符(称为operator.eq) - SQLAlchemy与此类运算符关联的实际SQL构造可以修改。 新操作也可以与列表达式相关联。 列表达式的操作符在类型级别最为直接重新定义 - 有关描述，请参见“重新定义和创建新操作符”一节。

ORM level functions like [column\_property()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_columns.html" \l "sqlalchemy.orm.column_property" \o "sqlalchemy.orm.column_property), [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship), and [composite()](http://docs.sqlalchemy.org/en/rel_1_1/orm/composites.html" \l "sqlalchemy.orm.composite" \o "sqlalchemy.orm.composite) also provide for operator redefinition at the ORM level, by passing a [PropComparator](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator" \o "sqlalchemy.orm.interfaces.PropComparator) subclass to the comparator\_factory argument of each function. Customization of operators at this level is a rare use case. See the documentation at [PropComparator](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator" \o "sqlalchemy.orm.interfaces.PropComparator) for an overview.

通过将PropComparator子类传递给每个函数的comparator\_factory参数，ORM级别函数(如column\_property()，relationship()和composite()也提供ORM级别的操作符重新定义。 这个级别的操作员的定制是一个罕见的用例。 有关概述，请参阅PropComparator上的文档。

### 2.2.4 Composite Column Types

Sets of columns can be associated with a single user-defined datatype. The ORM provides a single attribute which represents the group of columns using the class you provide.

一组列可以与单个用户定义的数据类型相关联。 ORM提供了一个单一属性，它表示使用您提供的类的列组。

*Changed in version 0.7:*Composites have been simplified such that they no longer "conceal" the underlying column based attributes. Additionally, in-place mutation is no longer automatic; see the section below on enabling mutability to support tracking of in-place changes.

版本0.7更改：复合材料已被简化，使得它们不再“隐藏”基于列的属性。 此外，就地突变不再是自动的; 请参阅下面的有关启用可变性以支持就地更改跟踪的部分。

*Changed in version 0.9:*Composites will return their object-form, rather than as individual columns, when used in a column-oriented [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) construct. See [Composite attributes are now returned as their object form when queried on a per-attribute basis](http://docs.sqlalchemy.org/en/rel_1_1/changelog/migration_09.html" \l "migration-2824).

在版本0.9中更改：当在面向列的Query构造中使用时，Composites将返回其对象形式，而不是单独的列。 在每个属性查询时，现在返回复合属性作为其对象形式。

A simple example represents pairs of columns as a Point object. Point represents such a pair as .x and .y:

一个简单的例子表示一列作为Point对象。 Point表示这样一对.x和.y：

**class** **Point**(object):

**def** \_\_init\_\_(self, x, y):

self.x = x

self.y = y

**def** \_\_composite\_values\_\_(self):

**return** self.x, self.y

**def** \_\_repr\_\_(self):

**return** "Point(x=*%r*, y=*%r*)" % (self.x, self.y)

**def** \_\_eq\_\_(self, other):

**return** isinstance(other, Point) **and** \

other.x == self.x **and** \

other.y == self.y

**def** \_\_ne\_\_(self, other):

**return** **not** self.\_\_eq\_\_(other)

The requirements for the custom datatype class are that it have a constructor which accepts positional arguments corresponding to its column format, and also provides a method \_\_composite\_values\_\_() which returns the state of the object as a list or tuple, in order of its column-based attributes. It also should supply adequate \_\_eq\_\_() and \_\_ne\_\_() methods which test the equality of two instances.

自定义数据类类的要求是它有一个构造函数，它接受与其列格式相对应的位置参数，并且还提供了一个方法\_\_composite\_values \_\_()，它以对列的列表或元组的顺序返回对象的状态， 基于属性。 它还应提供足够的\_\_eq \_\_()和\_\_ne \_\_()方法来测试两个实例的相等性。

We will create a mapping to a table vertices, which represents two points as x1/y1 and x2/y2. These are created normally as [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects. Then, the [composite()](http://docs.sqlalchemy.org/en/rel_1_1/orm/composites.html" \l "sqlalchemy.orm.composite" \o "sqlalchemy.orm.composite) function is used to assign new attributes that will represent sets of columns via the Point class:

我们将创建一个映射到一个表顶点，它表示两个点作为x1 / y1和x2 / y2。 它们通常作为Column对象创建。 然后，composite()函数用于分配通过Point类表示列的新属性：

**from** **sqlalchemy** **import** Column, Integer

**from** **sqlalchemy.orm** **import** composite

**from** **sqlalchemy.ext.declarative** **import** declarative\_base

Base = declarative\_base()

**class** **Vertex**(Base):

\_\_tablename\_\_ = 'vertices'

id = Column(Integer, primary\_key=**True**)

x1 = Column(Integer)

y1 = Column(Integer)

x2 = Column(Integer)

y2 = Column(Integer)

start = composite(Point, x1, y1)

end = composite(Point, x2, y2)

A classical mapping above would define each [composite()](http://docs.sqlalchemy.org/en/rel_1_1/orm/composites.html" \l "sqlalchemy.orm.composite" \o "sqlalchemy.orm.composite) against the existing table:

上面的经典映射将针对现有表定义每个composite()

mapper(Vertex, vertices\_table, properties={

'start':composite(Point, vertices\_table.c.x1, vertices\_table.c.y1),

'end':composite(Point, vertices\_table.c.x2, vertices\_table.c.y2),})

We can now persist and use Vertex instances, as well as query for them, using the .start and .end attributes against ad-hoc Point instances:

现在我们可以使用针对ad-hoc Point实例的.start和.end属性来保留和使用顶点实例以及查询它们。

>>> v = Vertex(start=Point(3, 4), end=Point(5, 6))

>>> session.add(v)

>>> q = session.query(Vertex).filter(Vertex.start == Point(3, 4))

>>> **print**(q.first().start)

Point(x=3, y=4)

sqlalchemy.orm.**composite**(*class\_*, *\*attrs*, *\*\*kwargs*)

Return a composite column-based property for use with a Mapper.

返回与Mapper一起使用的基于列的复合属性。

See the mapping documentation section [Composite Column Types](http://docs.sqlalchemy.org/en/rel_1_1/orm/composites.html" \l "mapper-composite) for a full usage example.

有关完整的使用示例，请参阅映射文档部分“复合列类型”。

The [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty) returned by [composite()](http://docs.sqlalchemy.org/en/rel_1_1/orm/composites.html" \l "sqlalchemy.orm.composite" \o "sqlalchemy.orm.composite) is the [CompositeProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.descriptor_props.CompositeProperty" \o "sqlalchemy.orm.descriptor_props.CompositeProperty).

Composite()返回的MapperProperty是CompositeProperty。

|  |  |
| --- | --- |
| **Parameters:** | * ****class\_**** – The "composite type" class. * ****\*cols**** – List of Column objects to be mapped. * ****active\_history=False**** –When True, indicates that the "previous" value for a scalar attribute should be loaded when replaced, if not already loaded. See the same flag on [column\_property()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_columns.html" \l "sqlalchemy.orm.column_property" \o "sqlalchemy.orm.column_property).   *Changed in version 0.7:*This flag specifically becomes meaningful - previously it was a placeholder.   * ****group**** – A group name for this property when marked as deferred. * ****deferred**** – When True, the column property is "deferred", meaning that it does not load immediately, and is instead loaded when the attribute is first accessed on an instance. See also [deferred()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "sqlalchemy.orm.deferred" \o "sqlalchemy.orm.deferred). * ****comparator\_factory**** – a class which extends [CompositeProperty.Comparator](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.descriptor_props.CompositeProperty.Comparator" \o "sqlalchemy.orm.descriptor_props.CompositeProperty.Comparator) which provides custom SQL clause generation for comparison operations. * ****doc**** – optional string that will be applied as the doc on the class-bound descriptor. * ****info**** –Optional data dictionary which will be populated into the [MapperProperty.info](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "MapperProperty.info" \o "MapperProperty.info) attribute of this object.   *New in version 0.8.*   * ****extension**** – an [AttributeExtension](http://docs.sqlalchemy.org/en/rel_1_1/orm/deprecated.html" \l "sqlalchemy.orm.interfaces.AttributeExtension" \o "sqlalchemy.orm.interfaces.AttributeExtension) instance, or list of extensions, which will be prepended to the list of attribute listeners for the resulting descriptor placed on the class. ****Deprecated.**** Please see [AttributeEvents](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.AttributeEvents" \o "sqlalchemy.orm.events.AttributeEvents). |

Tracking In-Place Mutations on Composites

In-place changes to an existing composite value are not tracked automatically. Instead, the composite class needs to provide events to its parent object explicitly. This task is largely automated via the usage of the [MutableComposite](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableComposite" \o "sqlalchemy.ext.mutable.MutableComposite) mixin, which uses events to associate each user-defined composite object with all parent associations. Please see the example in [Establishing Mutability on Composites](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "mutable-composites).

对现有复合值的就地更改不会自动跟踪。 相反，复合类需要明确地向其父对象提供事件。 通过使用MutableComposite mixin来实现此任务，该变量使用事件将每个用户定义的组合对象与所有父关联关联。 请参见建立复合材料的可变性的例子。

*Changed in version 0.7:*In-place changes to an existing composite value are no longer tracked automatically; the functionality is superseded by the [MutableComposite](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableComposite" \o "sqlalchemy.ext.mutable.MutableComposite) class.

0.7版本更改：不再自动跟踪现有复合值的就地更改; 该功能被MutableComposite类取代。

Redefining Comparison Operations for Composites

The "equals" comparison operation by default produces an AND of all corresponding columns equated to one another. This can be changed using the comparator\_factory argument to [composite()](http://docs.sqlalchemy.org/en/rel_1_1/orm/composites.html" \l "sqlalchemy.orm.composite" \o "sqlalchemy.orm.composite), where we specify a custom [CompositeProperty.Comparator](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.descriptor_props.CompositeProperty.Comparator" \o "sqlalchemy.orm.descriptor_props.CompositeProperty.Comparator) class to define existing or new operations. Below we illustrate the "greater than" operator, implementing the same expression that the base "greater than" does:

默认情况下的“等于”比较操作将产生所有相应列的“等于”彼此。 这可以使用comparator\_factory参数更改为composite()，其中我们指定一个自定义的CompositeProperty.Comparator类来定义现有的或新的操作。 下面我们来说明“大于”运算符，实现与“大于”基数相同的表达式：

**from** **sqlalchemy.orm.properties** **import** CompositeProperty

**from** **sqlalchemy** **import** sql

**class** **PointComparator**(CompositeProperty.Comparator):

**def** \_\_gt\_\_(self, other):

*"""redefine the 'greater than' operation"""*

**return** sql.and\_(\*[a>b **for** a, b **in**

zip(self.\_\_clause\_element\_\_().clauses,

other.\_\_composite\_values\_\_())])

**class** **Vertex**(Base):

\_\_\_tablename\_\_ = 'vertices'

id = Column(Integer, primary\_key=**True**)

x1 = Column(Integer)

y1 = Column(Integer)

x2 = Column(Integer)

y2 = Column(Integer)

start = composite(Point, x1, y1,

comparator\_factory=PointComparator)

end = composite(Point, x2, y2,

comparator\_factory=PointComparator)

## 2.3 Mapping Class Inheritance Hierarchies

SQLAlchemy supports three forms of inheritance: ****single table inheritance****, where several types of classes are represented by a single table, ****concrete table inheritance****, where each type of class is represented by independent tables, and ****joined table inheritance****, where the class hierarchy is broken up among dependent tables, each class represented by its own table that only includes those attributes local to that class.

SQLAlchemy支持三种继承形式：**单表继承**，其中几种类型由单个表表示，**具体表继承**，每种类型的类由独立表表示，**连接表继承**，其中类层次结构被分解 在依赖表中，每个由其自己的表表示的类只包含该类的本地属性。

The most common forms of inheritance are single and joined table, while concrete inheritance presents more configurational challenges.

最常见的继承形式是单一和连接的表，而具体的继承呈现出更多的配置挑战。

When mappers are configured in an inheritance relationship, SQLAlchemy has the ability to load elements [polymorphically](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-polymorphically), meaning that a single query can return objects of multiple types.

当mapper配置为继承关系时，SQLAlchemy能够以多态方式加载元素，这意味着单个查询可以返回多种类型的对象。

**See also**

[Inheritance Mapping Recipes](http://docs.sqlalchemy.org/en/rel_1_1/orm/examples.html" \l "examples-inheritance) - complete exampes of joined, single and concrete inheritance

### 2.3.1 Joined Table Inheritance

In joined table inheritance, each class along a hierarchy of classes is represented by a distinct table. Querying for a particular subclass in the hierarchy will render as a SQL JOIN along all tables in its inheritance path - if the class is the base class, the default behavior is to include only the base table in the SELECT. In all cases, the ultimate class to instantiate for a given row is determined by a discriminator column or expression that works against the base table. A subclass loaded against the base table only will have only base attributes populated at first; the additional attributes will [lazy load](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-lazy-load) when they are accessed. Options also exist to query for all columns across multiple tables/subclasses up front.

在连接表继承中，沿类的层次结构的每个类由不同的表表示。 查询层次结构中的特定子类将以其继承路径中的所有表格形式显示为SQL JOIN - 如果该类是基类，则默认行为是仅在SELECT中包含基表。 在所有情况下，要为给定行实例化的最终类由通过对基表进行操作的标识符列或表达式确定。 基本加载的子类首先只会填充基本属性; 额外的属性在访问时会延迟加载。 还可以使用选项查询前面的多个表/子类中的所有列。

The base class in a joined inheritance hierarchy is configured with additional arguments that will refer to the polymorphic discriminator column as well as the identifier for the base class:

连接继承层次结构中的基类配置了附加参数，这些参数将引用多态标识符列以及基类的标识符：

**class** **Employee**(Base):

\_\_tablename\_\_ = 'employee'

id = Column(Integer, primary\_key=**True**)

name = Column(String(50))

type = Column(String(50))

\_\_mapper\_args\_\_ = {

'polymorphic\_identity':'employee',

'polymorphic\_on':type

}

Above, an additional column type is established to act as the ****discriminator****, configured as such using the [mapper.polymorphic\_on](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.params.polymorphic_on" \o "sqlalchemy.orm.mapper) parameter. This column will store a value which indicates the type of object represented within the row. The column may be of any datatype, though string and integer are the most common.

以上，建立一个额外的列type作为鉴别器，使用[mapper.polymorphic\_on](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.params.polymorphic_on" \o "sqlalchemy.orm.mapper)参数进行配置。 该列将存储一个值，该值指示行中表示的对象的类型。 列可能是任何数据类型，尽管字符串和整数是最常见的。

While a polymorphic discriminator expression is not strictly necessary, it is required if polymorphic loading is desired. Establishing a simple column on the base table is the easiest way to achieve this, however very sophisticated inheritance mappings may even configure a SQL expression such as a CASE statement as the polymorphic discriminator.

虽然多态鉴别器表达式不是严格必需的，但是如果需要多态加载则是必需的。 在基表上建立一个简单的列是实现此目的的最简单方法，然而非常复杂的继承映射甚至可以配置SQL表达式，如CASE语句作为多态鉴别器。

**Note**

Currently, ****only one discriminator column or SQL expression may be configured for the entire inheritance hierarchy****, typically on the base- most class in the hierarchy. "Cascading" polymorphic discriminator expressions are not yet supported.

目前，只能为整个继承层次结构配置一个鉴别列或SQL表达式，通常在层次结构中的最基类上。 尚未支持“级联”多态鉴别器表达式。

We next define Engineer and Manager subclasses of Employee. Each contains columns that represent the attributes unique to the subclass they represent. Each table also must contain a primary key column (or columns), as well as a foreign key reference to the parent table:

我们接下来定义Employee的Engineer和经理Manager。 每个都包含表示它们所代表的子类唯一的属性的列。 每个表还必须包含主键列(或列)，以及对父表的外键引用：

**class** **Engineer**(Employee):

\_\_tablename\_\_ = 'engineer'

id = Column(Integer, ForeignKey('employee.id'), primary\_key=**True**)

engineer\_name = Column(String(30))

\_\_mapper\_args\_\_ = {

'polymorphic\_identity':'engineer',

}

**class** **Manager**(Employee):

\_\_tablename\_\_ = 'manager'

id = Column(Integer, ForeignKey('employee.id'), primary\_key=**True**)

manager\_name = Column(String(30))

\_\_mapper\_args\_\_ = {

'polymorphic\_identity':'manager',

}

It is most common that the foreign key constraint is established on the same column or columns as the primary key itself, however this is not required; a column distinct from the primary key may also be made to refer to the parent via foreign key. The way that a JOIN is constructed from the base table to subclasses is also directly customizable, however this is rarely necessary.

最常见的是，外键约束与主键本身在同一列或列上建立，但这不是必需的; 与主键不同的列也可以通过外键来指代父代。 从基表到子类构建JOIN的方式也可以直接自定义，但这很少是必要的。

**Joined inheritance primary keys**

One natural effect of the joined table inheritance configuration is that the identity of any mapped object can be determined entirely from rows in the base table alone. This has obvious advantages, so SQLAlchemy always considers the primary key columns of a joined inheritance class to be those of the base table only. In other words, the id columns of both the engineer and manager tables are not used to locate Engineer or Manager objects - only the value in employee.id is considered. engineer.id and manager.id are still of course critical to the proper operation of the pattern overall as they are used to locate the joined row, once the parent row has been determined within a statement.

连接表继承配置的一个自然效果是任何映射对象的身份可以完全由基表中的行确定。这具有明显的优势，因此SQLAlchemy始终将连接继承类的主键列视为基表的主键列。换句话说，工程师和管理器表的id列不用于查找工程师或管理器对象 - 只考虑了employee.id中的值。一旦父语句行在语句中被确定，那么工程师.id和manager.id仍然对于整体模式的正确操作至关重要，因为它们被用来定位连接的行。

With the joined inheritance mapping complete, querying against Employee will return a combination of Employee, Engineer and Manager objects. Newly saved Engineer, Manager, and Employee objects will automatically populate the employee.type column with the correct "discriminator" value in this case "engineer", "manager", or "employee", as appropriate.

通过连接继承映射完成，对Employee的查询将返回Employee，Engineer和Manager对象的组合。新保存的工程师，经理和员工对象将根据需要自动填充employee.type列中具有正确“鉴别器”值的“工程师”，“管理者”或“员工”。

### Relationships with Joined Inheritance

Relationships are fully supported with joined table inheritance. The relationship involving a joined-inheritance class should target the class in the hierarchy that also corresponds to the foreign key constraint; below, as the employee table has a foreign key constraint back to the company table, the relationships are set up between Company and Employee:

关系通过连接的表继承完全支持。 涉及加入继承类的关系应该对应于也与外键约束对应的层次结构中的类; 以下，由于员工表有一个外键约束回到公司表，公司与员工之间建立了关系：

**class** **Company**(Base):

\_\_tablename\_\_ = 'company'

id = Column(Integer, primary\_key=**True**)

name = Column(String(50))

employees = relationship("Employee", back\_populates="company")

**class** **Employee**(Base):

\_\_tablename\_\_ = 'employee'

id = Column(Integer, primary\_key=**True**)

name = Column(String(50))

type = Column(String(50))

company\_id = Column(ForeignKey('company.id'))

company = relationship("Company", back\_populates="employees")

\_\_mapper\_args\_\_ = {

'polymorphic\_identity':'employee',

'polymorphic\_on':type

}

**class** **Manager**(Employee):

*# ...*

**class** **Engineer**(Employee):

*# ...*

If the foreign key constraint is on a table corresponding to a subclass, the relationship should target that subclass instead. In the example below, there is a foreign key constraint from manager to company, so the relationships are established between the Manager and Company classes:

如果外键约束位于与子类对应的表上，则该关系应该针对该子类。 在下面的例子中，从经理到公司有一个外键约束，所以在经理和公司类之间建立了关系：

**class** **Company**(Base):

\_\_tablename\_\_ = 'company'

id = Column(Integer, primary\_key=**True**)

name = Column(String(50))

managers = relationship("Manager", back\_populates="company")

**class** **Employee**(Base):

\_\_tablename\_\_ = 'employee'

id = Column(Integer, primary\_key=**True**)

name = Column(String(50))

type = Column(String(50))

\_\_mapper\_args\_\_ = {

'polymorphic\_identity':'employee',

'polymorphic\_on':type

}

**class** **Manager**(Employee):

\_\_tablename\_\_ = 'manager'

id = Column(Integer, ForeignKey('employee.id'), primary\_key=**True**)

manager\_name = Column(String(30))

company\_id = Column(ForeignKey('company.id'))

company = relationship("Company", back\_populates="managers")

\_\_mapper\_args\_\_ = {

'polymorphic\_identity':'manager',

}

**class** **Engineer**(Employee):

*# ...*

Above, the Manager class will have a Manager.company attribute; Company will have a Company.managers attribute that always loads against a join of the employee and manager tables together.

以上，Manager类将具有Manager.company属性; 公司将有一个Company.managers属性，它们总是加载在员工和管理员表的加入。

### Loading Joined Inheritance Mappings

See the sections [Loading Inheritance Hierarchies](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html) and [Loading objects with joined table inheritance](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "loading-joined-inheritance) for background on inheritnce loading techniques, including configuration of tables to be queried both at mapper configuration time as well as query time.

请参阅“加载继承层次结构并加载具有连接表继承的对象的部分”，以了解继承加载技术的背景，包括在映射器配置时以及查询时间时要查询的表的配置。

2.3.2 Single Table Inheritance

Single table inheritance represents all attributes of all subclasses within a single table. A particular subclass that has attributes unique to that class will persist them within columns in the table that are otherwise NULL if the row refers to a different kind of object.

单表继承表示单个表中所有子类的所有属性。具有该类唯一的属性的特定子类将在表中的列中保留，否则为NULL，如果该行引用另一种类型的对象。

Querying for a particular subclass in the hierarchy will render as a SELECT against the base table, which will include a WHERE clause that limits rows to those with a particular value or values present in the discriminator column or expression.

查询层次结构中的特定子类将以基于表的形式显示为SELECT，其中将包含一个WHERE子句，该行将子行限制为具有特定值或者存在于鉴别符列或表达式中的值。

Single table inheritance has the advantage of simplicity compared to joined table inheritance; queries are much more efficient as only one table needs to be involved in order to load objects of every represented class.

与连接表继承相比，单表继承具有简单的优点;查询效率更高，因为只需要一个表来加载每个代表类的对象。

Single-table inheritance configuration looks much like joined-table inheritance, except only the base class specifies \_\_tablename\_\_. A discriminator column is also required on the base table so that classes can be differentiated from each other.

单表继承配置看起来很像连接表继承，只有基类指定\_\_tablename\_\_。基表上也需要一个鉴别符列，这样可以将类彼此区分开来。

Even though subclasses share the base table for all of their attributes, when using Declarative, [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects may still be specified on subclasses, indicating that the column is to be mapped only to that subclass; the [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) will be applied to the same base [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object:

即使子类共享其所有属性的基表，当使用Declarative时，仍可以在子类上指定Column对象，指示该列仅映射到该子类;列将被应用于相同的基础表对象：

**class** **Employee**(Base):

\_\_tablename\_\_ = 'employee'

id = Column(Integer, primary\_key=**True**)

name = Column(String(50))

type = Column(String(20))

\_\_mapper\_args\_\_ = {

'polymorphic\_on':type,

'polymorphic\_identity':'employee'

}

**class** **Manager**(Employee):

manager\_data = Column(String(50))

\_\_mapper\_args\_\_ = {

'polymorphic\_identity':'manager'

}

**class** **Engineer**(Employee):

engineer\_info = Column(String(50))

\_\_mapper\_args\_\_ = {

'polymorphic\_identity':'engineer'

}

Note that the mappers for the derived classes Manager and Engineer omit the \_\_tablename\_\_, indicating they do not have a mapped table of their own.

请注意，派生类管理器和工程师的映射器忽略\_\_tablename\_\_，表示它们没有自己的映射表。

### Relationships with Single Table Inheritance

Relationships are fully supported with single table inheritance. Configuration is done in the same manner as that of joined inheritance; a foreign key attribute should be on the same class that's the "foreign" side of the relationship:

单表继承完全支持关系。 配置的方式与加入继承相同; 一个外键属性应该在同一个类中，就是这个关系的“外部”方面：

**class** **Company**(Base):

\_\_tablename\_\_ = 'company'

id = Column(Integer, primary\_key=**True**)

name = Column(String(50))

employees = relationship("Employee", back\_populates="company")

**class** **Employee**(Base):

\_\_tablename\_\_ = 'employee'

id = Column(Integer, primary\_key=**True**)

name = Column(String(50))

type = Column(String(50))

company\_id = Column(ForeignKey('company.id'))

company = relationship("Company", back\_populates="employees")

\_\_mapper\_args\_\_ = {

'polymorphic\_identity':'employee',

'polymorphic\_on':type

}

**class** **Manager**(Employee):

manager\_data = Column(String(50))

\_\_mapper\_args\_\_ = {

'polymorphic\_identity':'manager'

}

**class** **Engineer**(Employee):

engineer\_info = Column(String(50))

\_\_mapper\_args\_\_ = {

'polymorphic\_identity':'engineer'

}

Also, like the case of joined inheritance, we can create relationships that involve a specific subclass. When queried, the SELECT statement will include a WHERE clause that limits the class selection to that subclass or subclasses:

另外，像加入继承的情况一样，我们可以创建涉及特定子类的关系。 当查询时，SELECT语句将包括一个将类选择限制为该子类或子类的WHERE子句：

**class** **Company**(Base):

\_\_tablename\_\_ = 'company'

id = Column(Integer, primary\_key=**True**)

name = Column(String(50))

managers = relationship("Manager", back\_populates="company")

**class** **Employee**(Base):

\_\_tablename\_\_ = 'employee'

id = Column(Integer, primary\_key=**True**)

name = Column(String(50))

type = Column(String(50))

\_\_mapper\_args\_\_ = {

'polymorphic\_identity':'employee',

'polymorphic\_on':type

}

**class** **Manager**(Employee):

manager\_name = Column(String(30))

company\_id = Column(ForeignKey('company.id'))

company = relationship("Company", back\_populates="managers")

\_\_mapper\_args\_\_ = {

'polymorphic\_identity':'manager',

}

**class** **Engineer**(Employee):

engineer\_info = Column(String(50))

\_\_mapper\_args\_\_ = {

'polymorphic\_identity':'engineer'

}

Above, the Manager class will have a Manager.company attribute; Company will have a Company.managers attribute that always loads against the employeewith an additional WHERE clause that limits rows to those with type = 'manager'.

以上，Manager类将具有Manager.company属性; 公司将拥有一个Company.managers属性，它们总是加载一个额外的WHERE子句，用于将行限制为type ='manager'的行。

### Loading Single Inheritance Mappings

The loading techniques for single-table inheritance are mostly identical to those used for joined-table inheritance, and a high degree of abstraction is provided between these two mapping types such that it is easy to switch between them as well as to intermix them in a single hierarchy (just omit \_\_tablename\_\_ from whichever subclasses are to be single-inheriting). See the sections [Loading Inheritance Hierarchies](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html) and [Loading objects with single table inheritance](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "loading-single-inheritance) for documentation on inheritance loading techniques, including configuration of classes to be queried both at mapper configuration time as well as query time.

单表继承的加载技术大多与用于连接表继承的加载技术相同，并且在这两种映射类型之间提供了高度的抽象，使得它们之间的切换容易并且将它们混合在一起 单个层次结构(仅从不考虑子类的单个继承中省略\_\_tablename\_\_)。 有关继承加载技术的文档，请参阅载入继承层次结构和加载对象的单表继承，包括在映射器配置时查询的类的配置以及查询时间。

2.3.3 Concrete Table Inheritance

Concrete inheritance maps each subclass to its own distinct table, each of which contains all columns necessary to produce an instance of that class. A concrete inheritance configuration by default queries non-polymorphically; a query for a particular class will only query that class' table and only return instances of that class. Polymorphic loading of concrete classes is enabled by configuring within the mapper a special SELECT that typically is produced as a UNION of all the tables.

具体的继承将每个子类映射到其自己的不同表，每个表都包含生成该类实例所需的所有列。 默认查询具体的继承配置非多态; 对特定类的查询将仅查询该类的表，并仅返回该类的实例。 通过在映射器内部配置一个特殊的SELECT，通常将其作为所有表的UNION生成，可以实现具体类的多态加载。

Whereas joined and single table inheritance are fluent in "polymorphic" loading, it is a more awkward affair in concrete inheritance. For this reason, concrete inheritance is more appropriate when polymorphic loading is not required. Establishing relationships that involve concrete inheritance classes is also more awkward.

而加入和单表继承流利的“多态”加载，这在具体继承中是一个更尴尬的事情。 因此，当不需要多态加载时，具体的继承更合适。 建立涉及具体继承类的关系也更加尴尬。

To establish a class as using concrete inheritance, add the [mapper.concrete](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.params.concrete" \o "sqlalchemy.orm.mapper) parameter within the \_\_mapper\_args\_\_. This indicates to Declarative as well as the mapping that the superclass table should not be considered as part of the mapping:

要建立一个类使用具体的继承，请在\_\_mapper\_args\_\_中添加mapper.concrete参数。 这表示声明性以及超类表不应被视为映射的一部分的映射：

**class** **Employee**(Base):

\_\_tablename\_\_ = 'employee'

id = Column(Integer, primary\_key=**True**)

name = Column(String(50))

**class** **Manager**(Employee):

\_\_tablename\_\_ = 'manager'

id = Column(Integer, primary\_key=**True**)

name = Column(String(50))

manager\_data = Column(String(50))

\_\_mapper\_args\_\_ = {

'concrete': **True**

}

**class** **Engineer**(Employee):

\_\_tablename\_\_ = 'engineer'

id = Column(Integer, primary\_key=**True**)

name = Column(String(50))

engineer\_info = Column(String(50))

\_\_mapper\_args\_\_ = {

'concrete': **True**

}

Two critical points should be noted:

* We must ****define all columns explicitly**** on each subclass, even those of the same name. A column such as Employee.name here is ****not**** copied out to the tables mapped by Manager or Engineer for us.我们必须在每个子类上显式地定义所有列，即使是同名的列。 这里的一个列(如Employee.name)不会被复制到由Manager或Engineer为我们映射的表中。
* while the Engineer and Manager classes are mapped in an inheritance relationship with Employee, they still ****do not include polymorphic loading****. Meaning, if we query for Employee objects, the manager and engineer tables are not queried at all.工程师和经理课程与Employee的遗传关系映射时，它们仍然不包括多态加载。 意思是，如果我们查询Employee对象，则不会查询经理和工程师表。

### Concrete Polymorphic Loading Configuration

Polymorphic loading with concrete inheritance requires that a specialized SELECT is configured against each base class that should have polymorphic loading. This SELECT needs to be capable of accessing all the mapped tables individually, and is typically a UNION statement that is constructed using a SQLAlchemy helper [polymorphic\_union()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.util.polymorphic_union" \o "sqlalchemy.orm.util.polymorphic_union).

具体具体继承的多态加载需要针对应该具有多态加载的每个基类配置专门的SELECT。 此SELECT需要能够单独访问所有映射表，并且通常是使用SQLAlchemy帮助器[polymorphic\_union()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.util.polymorphic_union" \o "sqlalchemy.orm.util.polymorphic_union)构造的UNION语句。

As discussed in [Loading Inheritance Hierarchies](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html), mapper inheritance configurations of any type can be configured to load from a special selectable by default using the [mapper.with\_polymorphic](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.params.with_polymorphic" \o "sqlalchemy.orm.mapper) argument. Current public API requires that this argument is set on a [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) when it is first constructed.

如加载继承层次结构中所讨论的，任何类型的映射器继承配置都可以配置为通过默认使用[mapper.with\_polymorphic](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.params.with_polymorphic" \o "sqlalchemy.orm.mapper)参数从特殊可选中加载。 当前的公共API要求在首次构建Mapper时设置此参数。

However, in the case of Declarative, both the mapper and the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) that is mapped are created at once, the moment the mapped class is defined. This means that the [mapper.with\_polymorphic](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.params.with_polymorphic" \o "sqlalchemy.orm.mapper) argument cannot be provided yet, since the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects that correspond to the subclasses haven't yet been defined.

There are a few strategies available to resolve this cycle, however Declarative provides helper classes [ConcreteBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.ConcreteBase" \o "sqlalchemy.ext.declarative.ConcreteBase) and [AbstractConcreteBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.AbstractConcreteBase" \o "sqlalchemy.ext.declarative.AbstractConcreteBase) which handle this issue behind the scenes.

然而，在Declarative的情况下，映射器和映射的表都将立即创建映射类定义的时刻。 这意味着mapper.with\_polymorphic参数不能提供，因为尚未定义与子类对应的Table对象。

有一些策略可用于解决此循环，但Declarative提供了帮助类ConcreteBase和AbstractConcreteBase，可以在幕后处理此问题。

Using [ConcreteBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.ConcreteBase" \o "sqlalchemy.ext.declarative.ConcreteBase), we can set up our concrete mapping in almost the same way as we do other forms of inheritance mappings:

使用ConcreteBase，我们可以用与其他形式的继承映射几乎相同的方式来设置我们的具体映射：

**from** **sqlalchemy.ext.declarative** **import** ConcreteBase

**class** **Employee**(ConcreteBase, Base):

\_\_tablename\_\_ = 'employee'

id = Column(Integer, primary\_key=**True**)

name = Column(String(50))

\_\_mapper\_args\_\_ = {

'polymorphic\_identity': 'employee',

'concrete': **True**

}

**class** **Manager**(Employee):

\_\_tablename\_\_ = 'manager'

id = Column(Integer, primary\_key=**True**)

name = Column(String(50))

manager\_data = Column(String(40))

\_\_mapper\_args\_\_ = {

'polymorphic\_identity': 'manager',

'concrete': **True**

}

**class** **Engineer**(Employee):

\_\_tablename\_\_ = 'engineer'

id = Column(Integer, primary\_key=**True**)

name = Column(String(50))

engineer\_info = Column(String(40))

\_\_mapper\_args\_\_ = {

'polymorphic\_identity': 'engineer',

'concrete': **True**

}

Above, Declarative sets up the polymorphic selectable for the Employee class at mapper "initialization" time; this is the late-configuration step for mappers that resolves other dependent mappers. The [ConcreteBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.ConcreteBase" \o "sqlalchemy.ext.declarative.ConcreteBase) helper uses the [polymorphic\_union()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.util.polymorphic_union" \o "sqlalchemy.orm.util.polymorphic_union) function to create a UNION of all concrete-mapped tables after all the other classes are set up, and then configures this statement with the already existing base-class mapper.

以上，声明式设置了在mapper“初始化”时间为Employee类可选择的多态， 这是解析其他依赖映射器的映射器的后期配置步骤。 在所有其他类设置完成之后，ConcreteBase帮助程序使用polymorphic\_union()函数创建所有具体映射表的UNION，然后使用已存在的基类映射器配置此语句。

Upon select, the polymorphic union produces a query like this:

选择后，多态联合会产生如下所示的查询：

session.query(Employee).all()

SELECT

pjoin.id AS pjoin\_id,

pjoin.name AS pjoin\_name,

pjoin.type AS pjoin\_type,

pjoin.manager\_data AS pjoin\_manager\_data,

pjoin.engineer\_info AS pjoin\_engineer\_info

FROM (

SELECT

employee.id AS id,

employee.name AS name,

CAST(NULL AS VARCHAR(50)) AS manager\_data,

CAST(NULL AS VARCHAR(50)) AS engineer\_info,

'employee' AS type

FROM employee

UNION ALL

SELECT

manager.id AS id,

manager.name AS name,

manager.manager\_data AS manager\_data,

CAST(NULL AS VARCHAR(50)) AS engineer\_info,

'manager' AS type

FROM manager

UNION ALL

SELECT

engineer.id AS id,

engineer.name AS name,

CAST(NULL AS VARCHAR(50)) AS manager\_data,

engineer.engineer\_info AS engineer\_info,

'engineer' AS type

FROM engineer

) AS pjoin

The above UNION query needs to manufacture "NULL" columns for each subtable in order to accommodate for those columns that aren't members of that particular subclass.

上述UNION查询需要为每个子表生成“NULL”列，以适应那些不是该特定子类成员的列。

### Abstract Concrete Classes

The concrete mappings illustrated thus far show both the subclasses as well as the base class mapped to individual tables. In the concrete inheritance use case, it is common that the base class is not represented within the database, only the subclasses. In other words, the base class is "abstract".

到目前为止所示的具体映射显示了子类以及映射到单个表的基类。 在具体的继承用例中，通常基数据库中没有表示基类，只有子类。 换句话说，基类是“抽象的”。

Normally, when one would like to map two different subclasses to individual tables, and leave the base class unmapped, this can be achieved very easily. When using Declarative, just declare the base class with the \_\_abstract\_\_ indicator:

通常，当想要将两个不同的子类映射到单独的表，并且离开基类时未映射，这可以非常容易地实现。 当使用Declarative时，只需使用\_\_abstract\_\_指示符声明基类：

**class** **Employee**(Base):

\_\_abstract\_\_ = **True**

**class** **Manager**(Employee):

\_\_tablename\_\_ = 'manager'

id = Column(Integer, primary\_key=**True**)

name = Column(String(50))

manager\_data = Column(String(40))

\_\_mapper\_args\_\_ = {

'polymorphic\_identity': 'manager',

}

**class** **Engineer**(Employee):

\_\_tablename\_\_ = 'engineer'

id = Column(Integer, primary\_key=**True**)

name = Column(String(50))

engineer\_info = Column(String(40))

\_\_mapper\_args\_\_ = {

'polymorphic\_identity': 'engineer',

}

Above, we are not actually making use of SQLAlchemy's inheritance mapping facilities; we can load and persist instances of Manager and Engineer normally. The situation changes however when we need to ****query polymorphically****, that is, we'd like to emit session.query(Employee) and get back a collection of Managerand Engineer instances. This brings us back into the domain of concrete inheritance, and we must build a special mapper against Employee in order to achieve this.

以上，我们实际上并没有使用SQLAlchemy的继承映射工具; 我们可以正常加载和保存经理和工程师的实例。 情况发生变化，但是当我们需要查询时，就是要发送session.query(Employee)并返回一个Manager和Engineer实例的集合。 这使我们回到具体的继承领域，为了实现这一点，我们必须建立一个专门的员工映射器。

**Mappers can always SELECT**

**Mappers** 可以随时**SELECT**

In SQLAlchemy, a mapper for a class always has to refer to some "selectable", which is normally a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) but may also refer to any [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) object as well. While it may appear that a "single table inheritance" mapper does not map to a table, these mappers in fact implicitly refer to the table that is mapped by a superclass.

在SQLAlchemy中，类的映射器总是必须引用一些“可选择”，通常是一个表，但也可以引用任何select()对象。 虽然似乎“单表继承”映射器不映射到表，但这些映射器实际上是隐含地引用由超类映射的表。

To modify our concrete inheritance example to illustrate an "abstract" base that is capable of polymorphic loading, we will have only an engineer and a managertable and no employee table, however the Employee mapper will be mapped directly to the "polymorphic union", rather than specifying it locally to the[mapper.with\_polymorphic](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.params.with_polymorphic" \o "sqlalchemy.orm.mapper) parameter.

为了修改我们具体的继承示例来说明一个能够进行多态加载的“抽象”基础，我们将只有一个工程师和一个可管理的，没有员工表，但是Employee映射器将被直接映射到“多态联盟” 而不是将它本地指定给themapper.with\_polymorphic参数。

To help with this, Declarative offers a variant of the [ConcreteBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.ConcreteBase" \o "sqlalchemy.ext.declarative.ConcreteBase) class called [AbstractConcreteBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.AbstractConcreteBase" \o "sqlalchemy.ext.declarative.AbstractConcreteBase) which achieves this automatically:

为了帮助这一点，Declarative提供了一个名为AbstractConcreteBase的ConcreteBase类的变体，它自动实现：

**from** **sqlalchemy.ext.declarative** **import** AbstractConcreteBase

**class** **Employee**(AbstractConcreteBase, Base):

**pass**

**class** **Manager**(Employee):

\_\_tablename\_\_ = 'manager'

id = Column(Integer, primary\_key=**True**)

name = Column(String(50))

manager\_data = Column(String(40))

\_\_mapper\_args\_\_ = {

'polymorphic\_identity': 'manager',

'concrete': **True**

}

**class** **Engineer**(Employee):

\_\_tablename\_\_ = 'engineer'

id = Column(Integer, primary\_key=**True**)

name = Column(String(50))

engineer\_info = Column(String(40))

\_\_mapper\_args\_\_ = {

'polymorphic\_identity': 'engineer',

'concrete': **True**

}

The [AbstractConcreteBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.AbstractConcreteBase" \o "sqlalchemy.ext.declarative.AbstractConcreteBase) helper class has a more complex internal process than that of [ConcreteBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.ConcreteBase" \o "sqlalchemy.ext.declarative.ConcreteBase), in that the entire mapping of the base class must be delayed until all the subclasses have been declared. With a mapping like the above, only instances of Manager and Engineer may be persisted; querying against the Employee class will always produce Manager and Engineer objects.

[AbstractConcreteBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.AbstractConcreteBase" \o "sqlalchemy.ext.declarative.AbstractConcreteBase)辅助类具有比[ConcreteBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.ConcreteBase" \o "sqlalchemy.ext.declarative.ConcreteBase)更复杂的内部进程，因为必须延迟基类的整个映射，直到所有的子类都被声明为止。 通过如上所述的映射，只有Manager和Engineer的实例可以被持久化; 对Employee类进行查询将始终生成Manage和Engineer对象。

**See also**

[Concrete Table Inheritance](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/inheritance.html" \l "declarative-concrete-table) - in the Declarative reference documentation

也可以看看

具体表继承 - 在声明式参考文档中

### Classical and Semi-Classical Concrete Polymorphic Configuration

The Declarative configurations illustrated with [ConcreteBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.ConcreteBase" \o "sqlalchemy.ext.declarative.ConcreteBase) and [AbstractConcreteBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.AbstractConcreteBase" \o "sqlalchemy.ext.declarative.AbstractConcreteBase) are equivalent to two other forms of configuration that make use of [polymorphic\_union()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.util.polymorphic_union" \o "sqlalchemy.orm.util.polymorphic_union) explicitly. These configurational forms make use of the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object explicitly so that the "polymorphic union" can be created first, then applied to the mappings. These are illustrated here to clarify the role of the [polymorphic\_union()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.util.polymorphic_union" \o "sqlalchemy.orm.util.polymorphic_union) function in terms of mapping.

A ****semi-classical mapping**** for example makes use of Declarative, but establishes the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects separately:

使用[ConcreteBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.ConcreteBase" \o "sqlalchemy.ext.declarative.ConcreteBase)和[AbstractConcreteBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.AbstractConcreteBase" \o "sqlalchemy.ext.declarative.AbstractConcreteBase)说明的声明式配置等同于使用[polymorphic\_union()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.util.polymorphic_union" \o "sqlalchemy.orm.util.polymorphic_union)的另外两种配置形式。 这些配置形式明确地使用了[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)对象，以便可以先创建“多态联合”，然后应用于映射。 这些在这里进行说明，以阐明[polymorphic\_union()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.util.polymorphic_union" \o "sqlalchemy.orm.util.polymorphic_union)函数在映射方面的作用。

例如，半经典映射使用Declarative，但分别建立[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)对象：

metadata = Base.metadata

employees\_table = Table(

'employee', metadata,

Column('id', Integer, primary\_key=**True**),

Column('name', String(50)),)

managers\_table = Table(

'manager', metadata,

Column('id', Integer, primary\_key=**True**),

Column('name', String(50)),

Column('manager\_data', String(50)),)

engineers\_table = Table(

'engineer', metadata,

Column('id', Integer, primary\_key=**True**),

Column('name', String(50)),

Column('engineer\_info', String(50)),)

Next, the UNION is produced using [polymorphic\_union()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.util.polymorphic_union" \o "sqlalchemy.orm.util.polymorphic_union):

**from** **sqlalchemy.orm** **import** polymorphic\_union

pjoin = polymorphic\_union({

'employee': employees\_table,

'manager': managers\_table,

'engineer': engineers\_table}, 'type', 'pjoin')

With the above [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects, the mappings can be produced using "semi-classical" style, where we use Declarative in conjunction with the \_\_table\_\_ argument; our polymorphic union above is passed via \_\_mapper\_args\_\_ to the [mapper.with\_polymorphic](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.params.with_polymorphic" \o "sqlalchemy.orm.mapper) parameter:

使用上述[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)对象，可以使用“半经典”样式生成映射，我们使用Declarative结合\_\_table\_\_参数; 我们上面的多态联合通过\_\_mapper\_args\_\_传递给[mapper.with\_polymorphic](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.params.with_polymorphic" \o "sqlalchemy.orm.mapper)参数：

**class** **Employee**(Base):

\_\_table\_\_ = employee\_table

\_\_mapper\_args\_\_ = {

'polymorphic\_on': pjoin.c.type,

'with\_polymorphic': ('\*', pjoin),

'polymorphic\_identity': 'employee'

}

**class** **Engineer**(Employee):

\_\_table\_\_ = engineer\_table

\_\_mapper\_args\_\_ = {

'polymorphic\_identity': 'engineer',

'concrete': **True**}

**class** **Manager**(Employee):

\_\_table\_\_ = manager\_table

\_\_mapper\_args\_\_ = {

'polymorphic\_identity': 'manager',

'concrete': **True**}

Alternatvely, the same [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects can be used in fully "classical" style, without using Declarative at all. A constructor similar to that supplied by Declarative is illustrated:

相反，相同的[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)对象可以完全用于“古典”样式，而不使用Declarative。 说明了与Declarative类似的构造函数：

**class** **Employee**(object):

**def** \_\_init\_\_(self, \*\*kw):

**for** k **in** kw:

setattr(self, k, kw[k])

**class** **Manager**(Employee):

**pass**

**class** **Engineer**(Employee):

**pass**

employee\_mapper = mapper(Employee, pjoin,

with\_polymorphic=('\*', pjoin),

polymorphic\_on=pjoin.c.type)manager\_mapper = mapper(Manager, managers\_table,

inherits=employee\_mapper,

concrete=**True**,

polymorphic\_identity='manager')engineer\_mapper = mapper(Engineer, engineers\_table,

inherits=employee\_mapper,

concrete=**True**,

polymorphic\_identity='engineer')

The "abstract" example can also be mapped using "semi-classical" or "classical" style. The difference is that instead of applying the "polymorphic union" to the [mapper.with\_polymorphic](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.params.with_polymorphic" \o "sqlalchemy.orm.mapper) parameter, we apply it directly as the mapped selectable on our basemost mapper. The semi-classical mapping is illustrated below:

“抽象”的例子也可以使用“半古典”或“古典”的风格进行映射。 不同之处在于，我们将“多态联合”应用于[mapper.with\_polymorphic](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.params.with_polymorphic" \o "sqlalchemy.orm.mapper)参数，而不是将其直接应用于我们的basemost映射器上的映射可选。 半经典映射如下所示：

**from** **sqlalchemy.orm** **import** polymorphic\_union

pjoin = polymorphic\_union({

'manager': managers\_table,

'engineer': engineers\_table}, 'type', 'pjoin')

**class** **Employee**(Base):

\_\_table\_\_ = pjoin

\_\_mapper\_args\_\_ = {

'polymorphic\_on': pjoin.c.type,

'with\_polymorphic': '\*',

'polymorphic\_identity': 'employee'

}

**class** **Engineer**(Employee):

\_\_table\_\_ = engineer\_table

\_\_mapper\_args\_\_ = {

'polymorphic\_identity': 'engineer',

'concrete': **True**}

**class** **Manager**(Employee):

\_\_table\_\_ = manager\_table

\_\_mapper\_args\_\_ = {

'polymorphic\_identity': 'manager',

'concrete': **True**}

Above, we use [polymorphic\_union()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.util.polymorphic_union" \o "sqlalchemy.orm.util.polymorphic_union) in the same manner as before, except that we omit the employee table.

**See also**

[Classical Mappings](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_styles.html" \l "classical-mapping) - background information on "classical" mappings

### Relationships with Concrete Inheritance

In a concrete inheritance scenario, mapping relationships is challenging since the distinct classes do not share a table. If the relationships only involve specific classes, such as a relationship between Company in our previous examples and Manager, special steps aren't needed as these are just two related tables.

在具体的继承方案中，映射关系是具有挑战性的，因为不同的类不共享表。 如果关系仅涉及特定的类，例如我们之前的例子中的公司与经理之间的关系，则不需要特殊的步骤，因为它们只是两个相关的表。

However, if Company is to have a one-to-many relationship to Employee, indicating that the collection may include both Engineer and Manager objects, that implies that Employee must have polymorphic loading capabilities and also that each table to be related must have a foreign key back to the company table. An example of such a configuration is as follows:

然而，如果Company要与Employee有一对多的关系，表明该集合可能包括Engineer和Manager，这意味着Employee必须具有多态加载能力，并且每个相关表必须有一个外键到company表。 这样的配置的示例如下：

**from** **sqlalchemy.ext.declarative** **import** ConcreteBase

**class** **Company**(Base):

\_\_tablename\_\_ = 'company'

id = Column(Integer, primary\_key=**True**)

name = Column(String(50))

employees = relationship("Employee")

**class** **Employee**(ConcreteBase, Base):

\_\_tablename\_\_ = 'employee'

id = Column(Integer, primary\_key=**True**)

name = Column(String(50))

company\_id = Column(ForeignKey('company.id'))

\_\_mapper\_args\_\_ = {

'polymorphic\_identity': 'employee',

'concrete': **True**

}

**class** **Manager**(Employee):

\_\_tablename\_\_ = 'manager'

id = Column(Integer, primary\_key=**True**)

name = Column(String(50))

manager\_data = Column(String(40))

company\_id = Column(ForeignKey('company.id'))

\_\_mapper\_args\_\_ = {

'polymorphic\_identity': 'manager',

'concrete': **True**

}

**class** **Engineer**(Employee):

\_\_tablename\_\_ = 'engineer'

id = Column(Integer, primary\_key=**True**)

name = Column(String(50))

engineer\_info = Column(String(40))

company\_id = Column(ForeignKey('company.id'))

\_\_mapper\_args\_\_ = {

'polymorphic\_identity': 'engineer',

'concrete': **True**

}

The next complexity with concrete inheritance and relationships involves when we'd like one or all of Employee, Manager and Engineer to themselves refer back to Company. For this case, SQLAlchemy has special behavior in that a [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) placed on Employee which links to Company ****does not work**** against the Manager and Engineer classes, when exercised at the instance level. Instead, a distinct [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) must be applied to each class. In order to achieve bi-directional behavior in terms of three separate relationships which serve as the opposite of Company.employees, the [relationship.back\_populates](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.back_populates" \o "sqlalchemy.orm.relationship)parameter is used between each of the relationships:

下一个具体遗产和关系的复杂性涉及当我们希望员工，经理和工程师中的一个或全部自己回到公司时。 对于这种情况，SQLAlchemy具有特殊的行为，因为在实例级别执行时，连接到Company的Employee上的relationship()对于Manager和Engineer类不起作用。 相反，必须对每个类应用一个明确的关系()。 为了实现与用作Company.employees相反的三个单独关系的双向行为，在每个关系之间使用relationship.back\_popplesparameter：

**from** **sqlalchemy.ext.declarative** **import** ConcreteBase

**class** **Company**(Base):

\_\_tablename\_\_ = 'company'

id = Column(Integer, primary\_key=**True**)

name = Column(String(50))

employees = relationship("Employee", back\_populates="company")

**class** **Employee**(ConcreteBase, Base):

\_\_tablename\_\_ = 'employee'

id = Column(Integer, primary\_key=**True**)

name = Column(String(50))

company\_id = Column(ForeignKey('company.id'))

company = relationship("Company", back\_populates="employees")

\_\_mapper\_args\_\_ = {

'polymorphic\_identity': 'employee',

'concrete': **True**

}

**class** **Manager**(Employee):

\_\_tablename\_\_ = 'manager'

id = Column(Integer, primary\_key=**True**)

name = Column(String(50))

manager\_data = Column(String(40))

company\_id = Column(ForeignKey('company.id'))

company = relationship("Company", back\_populates="employees")

\_\_mapper\_args\_\_ = {

'polymorphic\_identity': 'manager',

'concrete': **True**

}

**class** **Engineer**(Employee):

\_\_tablename\_\_ = 'engineer'

id = Column(Integer, primary\_key=**True**)

name = Column(String(50))

engineer\_info = Column(String(40))

company\_id = Column(ForeignKey('company.id'))

company = relationship("Company", back\_populates="employees")

\_\_mapper\_args\_\_ = {

'polymorphic\_identity': 'engineer',

'concrete': **True**

}

The above limitation is related to the current implementation, including that concrete inheriting classes do not share any of the attributes of the superclass and therefore need distinct relationships to be set up.

上述限制与当前实现有关，包括具体的继承类不共享超类的任何属性，因此需要建立不同的关系。

### Loading Concrete Inheritance Mappings

The options for loading with concrete inheritance are limited; generally, if polymorphic loading is configured on the mapper using one of the declarative concrete mixins, it can't be modified at query time in current SQLAlchemy versions. Normally, the [orm.with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic" \o "sqlalchemy.orm.with_polymorphic) function would be able to override the style of loading used by concrete, however due to current limitations this is not yet supported.

装载具体继承的选项有限; 通常，如果在映射器上使用一个声明性混合混合器配置多态加载，则在当前SQLAlchemy版本中无法在查询时修改它。 通常，[orm.with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic" \o "sqlalchemy.orm.with_polymorphic)函数将能够覆盖具体使用的加载样式，但是由于目前的限制，这还不被支持。

## 2.4 Non-Traditional Mappings

# **2.4.1Mapping a Class against Multiple Tables**

Mappers can be constructed against arbitrary relational units (called *selectables*) in addition to plain tables. For example, the [join()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.join" \o "sqlalchemy.sql.expression.join) function creates a selectable unit comprised of multiple tables, complete with its own composite primary key, which can be mapped in the same way as a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table):

除了普通表之外，还可以针对任意关系单元(称为可选项)来构造映射器。 例如，join()函数创建一个由多个表组成的可选单元，其中包含自己的复合主键，可以以与表相同的方式进行映射：

**from** **sqlalchemy** **import** Table, Column, Integer, \

String, MetaData, join, ForeignKey

**from** **sqlalchemy.ext.declarative** **import** declarative\_base

**from** **sqlalchemy.orm** **import** column\_property

metadata = MetaData()

*# define two Table objects*

user\_table = Table('user', metadata,

Column('id', Integer, primary\_key=**True**),

Column('name', String),

)

address\_table = Table('address', metadata,

Column('id', Integer, primary\_key=**True**),

Column('user\_id', Integer, ForeignKey('user.id')),

Column('email\_address', String)

)

*# define a join between them. This# takes place across the user.id and address.user\_id# columns.*

user\_address\_join = join(user\_table, address\_table)

Base = declarative\_base()

*# map to it*

**class** **AddressUser**(Base):

\_\_table\_\_ = user\_address\_join

id = column\_property(user\_table.c.id, address\_table.c.user\_id)

address\_id = address\_table.c.id

In the example above, the join expresses columns for both the user and the address table. The user.id and address.user\_id columns are equated by foreign key, so in the mapping they are defined as one attribute, AddressUser.id, using [column\_property()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_columns.html" \l "sqlalchemy.orm.column_property" \o "sqlalchemy.orm.column_property) to indicate a specialized column mapping. Based on this part of the configuration, the mapping will copy new primary key values from user.id into the address.user\_id column when a flush occurs.

在上面的示例中，连接表示user和address表的列。 user.id和address.user\_id列等价于外键，因此在映射中，它们被定义为一个属性AddressUser.id，使用[column\_property()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_columns.html" \l "sqlalchemy.orm.column_property" \o "sqlalchemy.orm.column_property)表示专门的列映射。基于这部分配置，当发生冲突时，映射将将新的主键值从user.id复制到address.user\_id列中。

Additionally, the address.id column is mapped explicitly to an attribute named address\_id. This is to ****disambiguate**** the mapping of the address.id column from the same-named AddressUser.id attribute, which here has been assigned to refer to the user table combined with the address.user\_id foreign key.

另外，将address.id列显式映射到一个名为address\_id的属性。这是为了消除address.id列与同名的AddressUser.id属性的映射，这个属性在这里被分配为引用与address.user\_id外键组合的用户表。

The natural primary key of the above mapping is the composite of (user.id, address.id), as these are the primary key columns of the user and address table combined together. The identity of an AddressUser object will be in terms of these two values, and is represented from an AddressUser object as (AddressUser.id, AddressUser.address\_id).

上述映射的自然主键是(user.id, address.id)的组合，因为这些是user的主键列，并将address 表组合在一起。 AddressUser对象的身份将按照这两个值来表示，并且由AddressUser对象表示为(AddressUser.id, AddressUser.address\_id)。

Mapping a Class against Arbitrary Selects

Similar to mapping against a join, a plain [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) object can be used with a mapper as well. The example fragment below illustrates mapping a class called Customer to a [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) which includes a join to a subquery:

类似于对连接的映射，普通的[select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select)对象也可以与映射器一起使用。 下面的示例片段说明将名为Customer的类映射到包含与子查询的连接的[select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select)：

**from** **sqlalchemy** **import** select, func

subq = select([

func.count(orders.c.id).label('order\_count'),

func.max(orders.c.price).label('highest\_order'),

orders.c.customer\_id

]).group\_by(orders.c.customer\_id).alias()

customer\_select = select([customers, subq]).\

select\_from(

join(customers, subq,

customers.c.id == subq.c.customer\_id)

).alias()

**class** **Customer**(Base):

\_\_table\_\_ = customer\_select

Above, the full row represented by customer\_select will be all the columns of the customers table, in addition to those columns exposed by the subq subquery, which are order\_count, highest\_order, and customer\_id. Mapping the Customer class to this selectable then creates a class which will contain those attributes.

在上面，由customer\_select表示的全部行将是customers表的所有列，除了由subq子查询公开的那些列之外，它们是order\_count，highest\_order和customer\_id。 将Customer类映射到此可选项，然后创建一个将包含这些属性的类。

When the ORM persists new instances of Customer, only the customers table will actually receive an INSERT. This is because the primary key of the orders table is not represented in the mapping; the ORM will only emit an INSERT into a table for which it has mapped the primary key.

当ORM保持Customer的新实例时，只有customers表实际上会收到一个INSERT。 这是因为在映射中没有表示orders表的主键; ORM只会将INSERT发送到映射主键的表中。

**Note**

The practice of mapping to arbitrary SELECT statements, especially complex ones as above, is almost never needed; it necessarily tends to produce complex queries which are often less efficient than that which would be produced by direct query construction. The practice is to some degree based on the very early history of SQLAlchemy where the [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper) construct was meant to represent the primary querying interface; in modern usage, the [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) object can be used to construct virtually any SELECT statement, including complex composites, and should be favored over the "map-to-selectable" approach.

映射到任意SELECT语句的做法，特别是上述的复杂语句几乎是不需要的; 它往往会产生复杂的查询，这些查询通常比通过直接查询构建产生的更为有效。 这种做法在某种程度上是基于SQLAlchemy的早期历史，其中[mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper)结构意在表示主要查询界面; 在现代使用中，[Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query)对象可以用于构建几乎任何SELECT语句，包括复杂的复合，并且应该通过“map-to-selectable”的方法。

Multiple Mappers for One Class

In modern SQLAlchemy, a particular class is mapped by only one so-called ****primary**** mapper at a time. This mapper is involved in three main areas of functionality: querying, persistence, and instrumentation of the mapped class. The rationale of the primary mapper relates to the fact that the [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper) modifies the class itself, not only persisting it towards a particular [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table), but also [instrumenting](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-instrumenting) attributes upon the class which are structured specifically according to the table metadata. It's not possible for more than one mapper to be associated with a class in equal measure, since only one mapper can actually instrument the class.

在现代的SQLAlchemy中，一个特定的类只能被一个所谓的主映射器映射。这个映射器涉及功能的三个主要领域：查询，持久化和检测映射类。主映射器的原理涉及到这样一个事实：[mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper)修改类本身，不仅将其持久化到特定的[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)，而且还对根据表元数据特定结构进行类的属性进行检测。由于只有一个映射器实际上可以对该类进行调试，所以不能将多个映射器与等级相关联。

However, there is a class of mapper known as the ****non primary**** mapper with allows additional mappers to be associated with a class, but with a limited scope of use. This scope typically applies to being able to load rows from an alternate table or selectable unit, but still producing classes which are ultimately persisted using the primary mapping. The non-primary mapper is created using the classical style of mapping against a class that is already mapped with a primary mapper, and involves the use of the [non\_primary](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.params.non_primary" \o "sqlalchemy.orm.mapper) flag.

然而，有一类称为非主映射器的映射器，允许其他映射器与一个类相关联，但使用范围有限。此范围通常适用于能够从备用表或可选单元加载行，但仍生成最终使用主映射持久化的类。非主要映射器是使用与已经与主映射器映射的类的映射的古典风格来创建的，并且涉及使用[non\_primary](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.params.non_primary" \o "sqlalchemy.orm.mapper)标志。

The non primary mapper is of very limited use in modern SQLAlchemy, as the task of being able to load classes from subqueries or other compound statements can be now accomplished using the [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) object directly.

非主要映射器在现代SQLAlchemy中的用途非常有限，因为可以直接使用Query对象从子查询或其他复合语句加载类的任务。

There is really only one use case for the non-primary mapper, which is that we wish to build a [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) to such a mapper; this is useful in the rare and advanced case that our relationship is attempting to join two classes together using many tables and/or joins in between. An example of this pattern is at [Relationship to Non Primary Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/join_conditions.html" \l "relationship-non-primary-mapper).

非主要映射器真的只有一个用例，我们希望与这样的映射器建立一个关系();这在我们的关系试图将两个类加入到一起使用许多表和/或连接之间的罕见和高级情况中是有用的。这种模式的一个例子是与非主映射关系。

As far as the use case of a class that can actually be fully persisted to different tables under different scenarios, very early versions of SQLAlchemy offered a feature for this adapted from Hibernate, known as the "entity name" feature. However, this use case became in feasable within SQLAlchemy once the mapped class itself became the source of SQL expression construction; that is, the class' attributes themselves link directly to mapped table columns. The feature was removed and replaced with a simple recipe-oriented approach to accomplishing this task without any ambiguity of instrumentation - to create new subclasses, each mapped individually. This pattern is now available as a recipe at [Entity Name](http://www.sqlalchemy.org/trac/wiki/UsageRecipes/EntityName).

就在不同场景下实际可以完全持久化到不同表的类的用例来说，非常早期的SQLAlchemy版本为Hibernate提供了一个称为"实体名称"功能的功能。 但是，一旦映射类本身成为SQL表达式构建的源头，这个用例在SQLAlchemy中成为可行的; 也就是说，类的属性本身直接链接到映射表列。 该功能被删除，并用简单的面向对象的方法替代，以完成此任务，而不会有任何歧义的模块 - 创建新的子类，每个映射单独。 此模式现在可作为实体名称的配方使用。

## **2.5 Configuring a Version Counter**

The [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) supports management of a version id column, which is a single table column that increments or otherwise updates its value each time an UPDATE to the mapped table occurs. This value is checked each time the ORM emits an UPDATE or DELETE against the row to ensure that the value held in memory matches the database value.

Mapper支持对版本ID列的管理，该列是单个表列，每次发生对映射表的UPDATE时增加或以其他方式更新其值。 每次ORM针对该行发出UPDATE或DELETE时，将检查此值，以确保内存中保存的值与数据库值匹配。

**Warning**

Because the versioning feature relies upon comparison of the ****in memory**** record of an object, the feature only applies to the [Session.flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.flush" \o "sqlalchemy.orm.session.Session.flush) process, where the ORM flushes individual in-memory rows to the database. It does ****not**** take effect when performing a multirow UPDATE or DELETE using [Query.update()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.update" \o "sqlalchemy.orm.query.Query.update) or [Query.delete()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.delete" \o "sqlalchemy.orm.query.Query.delete) methods, as these methods only emit an UPDATE or DELETE statement but otherwise do not have direct access to the contents of those rows being affected.

因为版本控制功能依赖于对象的内存记录的比较，所以该功能仅适用于[Session.flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.flush" \o "sqlalchemy.orm.session.Session.flush)进程，ORM会将单个内存中的行刷新到数据库。 当使用[Query.update()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.update" \o "sqlalchemy.orm.query.Query.update)或[Query.delete()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.delete" \o "sqlalchemy.orm.query.Query.delete)方法执行多重UPDATE或DELETE时，它不起作用，因为这些方法只发出UPDATE或DELETE语句，但是不能直接访问这些行的内容影响。

The purpose of this feature is to detect when two concurrent transactions are modifying the same row at roughly the same time, or alternatively to provide a guard against the usage of a "stale" row in a system that might be re-using data from a previous transaction without refreshing (e.g. if one sets expire\_on\_commit=False with a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session), it is possible to re-use the data from a previous transaction).

此功能的目的是检测两个并发事务是否在大致相同的时间修改同一行，或者替代地提供防止系统中"过时"行的使用的保护，这可能是重新使用来自 以前的事务没有刷新(例如，如果一个会话使expire\_on\_commit=False，则可以重新使用先前事务中的数据)。

**Concurrent transaction updates**

When detecting concurrent updates within transactions, it is typically the case that the database's transaction isolation level is below the level of repeatable read; otherwise, the transaction will not be exposed to a new row value created by a concurrent update which conflicts with the locally updated value. In this case, the SQLAlchemy versioning feature will typically not be useful for in-transaction conflict detection, though it still can be used for cross-transaction staleness detection.

当检测事务中的并发更新时，通常情况下数据库的事务隔离级别低于可重复读取级别; 否则，事务不会暴露于与本地更新值冲突的并发更新创建的新行值。 在这种情况下，SQLAlchemy版本控制功能通常不会对事务内冲突检测有用，尽管它仍然可以用于交叉事务的初始化检测。

The database that enforces repeatable reads will typically either have locked the target row against a concurrent update, or is employing some form of multi version concurrency control such that it will emit an error when the transaction is committed. SQLAlchemy's version\_id\_col is an alternative which allows version tracking to occur for specific tables within a transaction that otherwise might not have this isolation level set.

执行可重复读取的数据库通常会将目标行锁定在并发更新中，或者采用某种形式的多版本并发控制，以便在事务提交时发出错误。 SQLAlchemy的version\_id\_col是一种替代方法，允许对事务中的特定表进行版本跟踪，否则可能不会设置此隔离级别。

**See also**

[Repeatable Read Isolation Level](http://www.postgresql.org/docs/9.1/static/transaction-iso.html" \l "XACT-REPEATABLE-READ) - PostgreSQL's implementation of repeatable read, including a description of the error condition.

[Repeatable Read Isolation Level](http://www.postgresql.org/docs/9.1/static/transaction-iso.html" \l "XACT-REPEATABLE-READ) - PostgreSQL执行可重复读取，包括错误状态的描述。

Simple Version Counting

The most straightforward way to track versions is to add an integer column to the mapped table, then establish it as the version\_id\_col within the mapper options:

跟踪版本的最简单的方法是向映射表添加一个整数列，然后在mapper选项中将其设置为version\_id\_col：

**class** **User**(Base):

\_\_tablename\_\_ = 'user'

id = Column(Integer, primary\_key=**True**)

version\_id = Column(Integer, nullable=**False**)

name = Column(String(50), nullable=**False**)

\_\_mapper\_args\_\_ = {

"version\_id\_col": version\_id

}

**Note**

It is ****strongly recommended**** that the version\_id column be made NOT NULL. The versioning feature ****does not support**** a NULL value in the versioning column.

the table later on will always be emitted in a manner similar to the following:

强烈建议将version\_id列设为NOT NULL。 版本控制功能在版本控制列中不支持NULL值。

Above, the User mapping tracks integer versions using the column version\_id. When an object of type User is first flushed, the version\_id column will be given a value of "1". Then, an UPDATE of

以上，User映射使用列version\_id跟踪整数版本。 当首先刷新User类型的对象时，version\_id列将被赋予值"1"。 然后，表的更新将始终以类似于以下的方式发出：

UPDATE user SET version\_id=:version\_id, name=:nameWHERE user.id = :user\_id AND user.version\_id = :user\_version\_id

{"name": "new name", "version\_id": 2, "user\_id": 1, "user\_version\_id": 1}

The above UPDATE statement is updating the row that not only matches user.id = 1, it also is requiring that user.version\_id = 1, where "1" is the last version identifier we've been known to use on this object. If a transaction elsewhere has modified the row independently, this version id will no longer match, and the UPDATE statement will report that no rows matched; this is the condition that SQLAlchemy tests, that exactly one row matched our UPDATE (or DELETE) statement. If zero rows match, that indicates our version of the data is stale, and a [StaleDataError](http://docs.sqlalchemy.org/en/rel_1_1/orm/exceptions.html" \l "sqlalchemy.orm.exc.StaleDataError" \o "sqlalchemy.orm.exc.StaleDataError) is raised.

上述UPDATE语句正在更新不仅匹配user.id = 1的行，它还要求user.version\_id = 1，其中"1"是我们已知在此对象上使用的最后一个版本标识符。 如果其他地方的交易独立地修改了该行，则该版本号将不再匹配，UPDATE语句将报告没有行匹配; 这是SQLAlchemy测试的条件，正好一行与我们的UPDATE(或DELETE)语句相匹配。 如果零行匹配，则表示我们的数据版本过时，并引发[StaleDataError](http://docs.sqlalchemy.org/en/rel_1_1/orm/exceptions.html" \l "sqlalchemy.orm.exc.StaleDataError" \o "sqlalchemy.orm.exc.StaleDataError)。

Custom Version Counters / Types

Other kinds of values or counters can be used for versioning. Common types include dates and GUIDs. When using an alternate type or counter scheme, SQLAlchemy provides a hook for this scheme using the version\_id\_generator argument, which accepts a version generation callable. This callable is passed the value of the current known version, and is expected to return the subsequent version.

其他类型的值或计数器可用于版本控制。 常用类型包括日期和GUID。 当使用替代类型或计数器方案时，SQLAlchemy使用version\_id\_generator参数为此方案提供了一个钩子，该参数接受版本生成可调用。 这个可调用被传递当前已知版本的值，并且预期返回后续版本。

For example, if we wanted to track the versioning of our User class using a randomly generated GUID, we could do this (note that some backends support a native GUID type, but we illustrate here using a simple string):

例如，如果我们想使用随机生成的GUID跟踪我们的User类的版本，我们可以这样做(请注意，一些后端支持本机GUID类型，但我们在这里使用一个简单的字符串)

**import** **uuid**

**class** **User**(Base):

\_\_tablename\_\_ = 'user'

id = Column(Integer, primary\_key=**True**)

version\_uuid = Column(String(32), nullable=**False**)

name = Column(String(50), nullable=**False**)

\_\_mapper\_args\_\_ = {

'version\_id\_col':version\_uuid,

'version\_id\_generator':**lambda** version: uuid.uuid4().hex

}

The persistence engine will call upon uuid.uuid4() each time a User object is subject to an INSERT or an UPDATE. In this case, our version generation function can disregard the incoming value of version, as the uuid4() function generates identifiers without any prerequisite value. If we were using a sequential versioning scheme such as numeric or a special character system, we could make use of the given version in order to help determine the subsequent value.

每次User对象受到INSERT或UPDATE的影响时，持久化引擎将会调用uuid.uuid4()。 在这种情况下，我们的版本生成函数可以忽略版本的输入值，因为uuid4()函数生成没有任何先决条件值的标识符。 如果我们使用数字或特殊字符系统的顺序版本控制方案，我们可以利用给定的version来帮助确定后续的值。

**See also**

[Backend-agnostic GUID Type](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "custom-guid-type)

Server Side Version Counters

The version\_id\_generator can also be configured to rely upon a value that is generated by the database. In this case, the database would need some means of generating new identifiers when a row is subject to an INSERT as well as with an UPDATE. For the UPDATE case, typically an update trigger is needed, unless the database in question supports some other native version identifier. The PostgreSQL database in particular supports a system column called [xmin](http://www.postgresql.org/docs/9.1/static/ddl-system-columns.html) which provides UPDATE versioning. We can make use of the PostgreSQL xmin column to version our User class as follows:

也可以将version\_id\_generator配置为依赖数据库生成的值。 在这种情况下，数据库将需要一些方法来生成新的标识符，当一行受INSERT以及UPDATE的限制。 对于UPDATE情况，通常需要更新触发器，除非有问题的数据库支持某些其他本机版本标识符。 PostgreSQL数据库特别支持一个名为xmin的系统列，它提供UPDATE版本控制。 我们可以使用PostgreSQL xmin列来对我们的User类进行如下的版本：

**from** **sqlalchemy** **import** FetchedValue

**class** **User**(Base):

\_\_tablename\_\_ = 'user'

id = Column(Integer, primary\_key=**True**)

name = Column(String(50), nullable=**False**)

xmin = Column("xmin", Integer, system=**True**, server\_default=FetchedValue())

\_\_mapper\_args\_\_ = {

'version\_id\_col': xmin,

'version\_id\_generator': **False**

}

With the above mapping, the ORM will rely upon the xmin column for automatically providing the new value of the version id counter.

通过上述映射，ORM将依靠xmin列自动提供版本ID计数器的新值。

**creating tables that refer to system columns**

创建引用系统列的表

In the above scenario, as xmin is a system column provided by PostgreSQL, we use the system=True argument to mark it as a system-provided column, omitted from the CREATETABLE statement.

在上述情况下，由于xmin是由PostgreSQL提供的系统列，因此我们使用system=True参数将其标记为系统提供的列，从CREATETABLE语句中省略。

The ORM typically does not actively fetch the values of database-generated values when it emits an INSERT or UPDATE, instead leaving these columns as "expired" and to be fetched when they are next accessed, unless the eager\_defaults [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper) flag is set. However, when a server side version column is used, the ORM needs to actively fetch the newly generated value. This is so that the version counter is set up *before* any concurrent transaction may update it again. This fetching is also best done simultaneously within the INSERT or UPDATE statement using [RETURNING](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-returning), otherwise if emitting a SELECT statement afterwards, there is still a potential race condition where the version counter may change before it can be fetched.

ORM通常在发出INSERT或UPDATE时不会主动获取数据库生成的值，而是将这些列保留为"已过期"，并在下次访问时被提取，除非设置eager\_defaults [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper)标志。但是，当使用服务器端版本列时，ORM需要主动获取新生成的值。这是为了在任何并发事务可能再次更新之前设置版本计数器。在INSERT或UPDATE语句中也可以使用RETURNING同时获取此提取，否则，如果之后发出SELECT语句，则仍然可能会在版本计数器可能更改之前更改其竞争条件。

When the target database supports RETURNING, an INSERT statement for our User class will look like this:

当目标数据库支持RETURNING时，我们的User类的INSERT语句将如下所示：

INSERT INTO "user" (name) VALUES (%(name)s) RETURNING "user".id, "user".xmin{'name': 'ed'}

Where above, the ORM can acquire any newly generated primary key values along with server-generated version identifiers in one statement. When the backend does not support RETURNING, an additional SELECT must be emitted for ****every**** INSERT and UPDATE, which is much less efficient, and also introduces the possibility of missed version counters:

在上述情况下，ORM可以在一个语句中获取任何新生成的主键值以及服务器生成的版本标识符。 当后端不支持RETURNING时，必须为每个INSERT和UPDATE发出一个额外的SELECT，效率要低得多，并且还引入了错过的版本计数器的可能性：

INSERT INTO "user" (name) VALUES (%(name)s){'name': 'ed'}

SELECT "user".version\_id AS user\_version\_id FROM "user" where"user".id = :param\_1{"param\_1": 1}

It is *strongly recommended* that server side version counters only be used when absolutely necessary and only on backends that support [RETURNING](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-returning), e.g. PostgreSQL, Oracle, SQL Server (though SQL Server has [major caveats](http://blogs.msdn.com/b/sqlprogrammability/archive/2008/07/11/update-with-output-clause-triggers-and-sqlmoreresults.aspx) when triggers are used), Firebird.

强烈建议仅在绝对必要时才使用服务器端版本计数器，并且只能在支持RETURNING的后端使用。 PostgreSQL，Oracle，SQL Server(尽管SQL Server在使用触发器时有重要的警告)，Firebird。

*New in version 0.9.0:*Support for server side version identifier tracking.

Programmatic or Conditional Version Counters

When version\_id\_generator is set to False, we can also programmatically (and conditionally) set the version identifier on our object in the same way we assign any other mapped attribute. Such as if we used our UUID example, but set version\_id\_generator to False, we can set the version identifier at our choosing:

当version\_id\_generator设置为False时，我们还可以以编程方式(有条件地)在我们的对象上设置版本标识符，就像我们分配任何其他映射属性一样。 如果我们使用我们的UUID示例，但是将version\_id\_generator设置为False，我们可以在我们的选择中设置版本标识符：

**import** **uuid**

**class** **User**(Base):

\_\_tablename\_\_ = 'user'

id = Column(Integer, primary\_key=**True**)

version\_uuid = Column(String(32), nullable=**False**)

name = Column(String(50), nullable=**False**)

\_\_mapper\_args\_\_ = {

'version\_id\_col':version\_uuid,

'version\_id\_generator': **False**

}

u1 = User(name='u1', version\_uuid=uuid.uuid4())

session.add(u1)

session.commit()

u1.name = 'u2'

u1.version\_uuid = uuid.uuid4()

session.commit()

We can update our User object without incrementing the version counter as well; the value of the counter will remain unchanged, and the UPDATE statement will still check against the previous value. This may be useful for schemes where only certain classes of UPDATE are sensitive to concurrency issues:

我们可以更新我们的User对象，而不增加版本计数器; 计数器的值将保持不变，UPDATE语句仍将检查以前的值。 这对于只有某些UPDATE类对并发问题敏感的方案可能是有用的：

*# will leave version\_uuid unchanged*

u1.name = 'u3'session.commit()

*New in version 0.9.0:*Support for programmatic and conditional version identifier tracking.

# **2.6 Class Mapping API**

sqlalchemy.orm.**mapper**(*class\_*, *local\_table=None*, *properties=None*, *primary\_key=None*, *non\_primary=False*, *inherits=None*, *inherit\_condition=None*, *inherit\_foreign\_keys=None*, *extension=None*, *order\_by=False*, *always\_refresh=False*, *version\_id\_col=None*, *version\_id\_generator=None*, *polymorphic\_on=None*, *\_polymorphic\_map=None*, *polymorphic\_identity=None*, *concrete=False*, *with\_polymorphic=None*, *allow\_partial\_pks=True*, *batch=True*, *column\_prefix=None*, *include\_properties=None*, *exclude\_properties=None*, *passive\_updates=True*, *passive\_deletes=False*, *confirm\_deleted\_rows=True*, *eager\_defaults=False*, *legacy\_is\_orphan=False*, *\_compiled\_cache\_size=100*)

Return a new [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) object.

This function is typically used behind the scenes via the Declarative extension. When using Declarative, many of the usual [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper) arguments are handled by the Declarative extension itself, including class\_, local\_table, properties, and inherits. Other options are passed to [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper) using the \_\_mapper\_args\_\_ class variable:

**class** **MyClass**(Base):

\_\_tablename\_\_ = 'my\_table'

id = Column(Integer, primary\_key=**True**)

type = Column(String(50))

alt = Column("some\_alt", Integer)

\_\_mapper\_args\_\_ = {

'polymorphic\_on' : type

}

Explicit use of [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper) is often referred to as *classical mapping*. The above declarative example is equivalent in classical form to:

my\_table = Table("my\_table", metadata,

Column('id', Integer, primary\_key=**True**),

Column('type', String(50)),

Column("some\_alt", Integer))

**class** **MyClass**(object):

**pass**

mapper(MyClass, my\_table,

polymorphic\_on=my\_table.c.type,

properties={

'alt':my\_table.c.some\_alt

})

**See also**

[Classical Mappings](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_styles.html" \l "classical-mapping) - discussion of direct usage of [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper)

|  |  |
| --- | --- |
| **Parameters:** | * ****class\_**** – The class to be mapped. When using Declarative, this argument is automatically passed as the declared class itself.要映射的类。 当使用Declarative时，此参数将自动作为声明的类本身传递。 * ****local\_table**** – The [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) or other selectable to which the class is mapped. May be None if this mapper inherits from another mapper using single-table inheritance. When using Declarative, this argument is automatically passed by the extension, based on what is configured via the \_\_table\_\_ argument or via the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) produced as a result of the \_\_tablename\_\_ and [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) arguments present.该类映射的表或其他可选项。 如果此映射器使用单表继承继承自另一个映射器，则可能为“无”。 当使用Declarative时，这个参数是根据通过\_\_table\_\_参数配置的，或者通过\_\_tablename\_\_和Column参数的结果生成的Table自动传递的。 * ****always\_refresh**** – If True, all query operations for this mapped class will overwrite all data within object instances that already exist within the session, erasing any in-memory changes with whatever information was loaded from the database. Usage of this flag is highly discouraged; as an alternative, see the method[Query.populate\_existing()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.populate_existing" \o "sqlalchemy.orm.query.Query.populate_existing).如果为True，则此映射类的所有查询操作将覆盖会话中已存在的对象实例中的所有数据，并使用从数据库加载的任何信息擦除任何内存中更改。 这个国旗的使用是非常不鼓励的; 作为替代方法，请参阅methodQuery.populate\_existing()。 * ****allow\_partial\_pks**** – Defaults to True. Indicates that a composite primary key with some NULL values should be considered as possibly existing within the database. This affects whether a mapper will assign an incoming row to an existing identity, as well as if [Session.merge()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.merge" \o "sqlalchemy.orm.session.Session.merge) will check the database first for a particular primary key value. A "partial primary key" can occur if one has mapped to an OUTER JOIN, for example.默认为True。 表示具有某些NULL值的复合主键应被视为可能存在于数据库中。 这会影响映射器是否会将传入的行分配给现有标识，以及Session.merge()将首先检查数据库的特定主键值。 例如，如果映射到OUTER JOIN，则可能会发生“部分主键”。 * ****batch**** – Defaults to True, indicating that save operations of multiple entities can be batched together for efficiency. Setting to False indicates that an instance will be fully saved before saving the next instance. This is used in the extremely rare case that a [MapperEvents](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents" \o "sqlalchemy.orm.events.MapperEvents) listener requires being called in between individual row persistence operations.默认为True，表示多个实体的保存操作可以一起批处理，以提高效率。 设置为False表示在保存下一个实例之前，实例将被完全保存。 这在非常罕见的情况下被使用，一个MapperEvents监听器需要在单独行持久性操作之间调用。 * ****column\_prefix**** –A string which will be prepended to the mapped attribute name when [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects are automatically assigned as attributes to the mapped class. Does not affect explicitly specified column-based properties.   See the section [Naming All Columns with a Prefix](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_columns.html" \l "column-prefix) for an example.   * ****concrete –****If True, indicates this mapper should use concrete table inheritance with its parent mapper.如果为True，则表示此映射程序应使用其父映射程序的具体表继承。   See the section [Concrete Table Inheritance](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance.html" \l "concrete-inheritance) for an example.   * ****confirm\_deleted\_rows –****defaults to True; when a DELETE occurs of one more rows based on specific primary keys, a warning is emitted when the number of rows matched does not equal the number of rows expected. This parameter may be set to False to handle the case where database ON DELETE CASCADE rules may be deleting some of those rows automatically. The warning may be changed to an exception in a future release.默认为True; 当基于特定主键发生多一行的DELETE时，当匹配的行数不等于预期的行数时，将发出警告。 此参数可能设置为False以处理数据库ON DELETE CASCADE规则可能会自动删除其中某些行的情况。 该警告可能会在以后的版本中更改为例外。   *New in version 0.9.4:*- added [mapper.confirm\_deleted\_rows](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.params.confirm_deleted_rows" \o "sqlalchemy.orm.mapper) as well as conditional matched row checking on delete.   * ****eager\_defaults –****if True, the ORM will immediately fetch the value of server-generated default values after an INSERT or UPDATE, rather than leaving them as expired to be fetched on next access. This can be used for event schemes where the server-generated values are needed immediately before the flush completes. By default, this scheme will emit an individual SELECT statement per row inserted or updated, which note can add significant performance overhead. However, if the target database supports [RETURNING](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-returning), the default values will be returned inline with the INSERT or UPDATE statement, which can greatly enhance performance for an application that needs frequent access to just-generated server defaults.如果为True，则ORM将在INSERT或UPDATE之后立即获取服务器生成的默认值的值，而不是将其保留为在下次访问时提取的值。 这可以用于在刷新完成之前立即需要服务器生成的值的事件方案。 默认情况下，该方案将为每行插入或更新发出一个单独的SELECT语句，该注释可能会增加显着的性能开销。 但是，如果目标数据库支持RETURNING，则默认值将与INSERT或UPDATE语句一起返回，这可以大大提高需要频繁访问刚刚生成的服务器默认值的应用程序的性能。   *Changed in version 0.9.0:*The eager\_defaults option can now make use of [RETURNING](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-returning) for backends which support it.   * ****exclude\_properties –****A list or set of string column names to be excluded from mapping.要从映射中排除的列表或一组字符串列名称。   See [Mapping a Subset of Table Columns](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_columns.html" \l "include-exclude-cols) for an example.   * ****extension**** – A [MapperExtension](http://docs.sqlalchemy.org/en/rel_1_1/orm/deprecated.html" \l "sqlalchemy.orm.interfaces.MapperExtension" \o "sqlalchemy.orm.interfaces.MapperExtension) instance or list of [MapperExtension](http://docs.sqlalchemy.org/en/rel_1_1/orm/deprecated.html" \l "sqlalchemy.orm.interfaces.MapperExtension" \o "sqlalchemy.orm.interfaces.MapperExtension) instances which will be applied to all operations by this [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper). ****Deprecated.****Please see [MapperEvents](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents" \o "sqlalchemy.orm.events.MapperEvents). * ****include\_properties –****   An inclusive list or set of string column names to map.  See [Mapping a Subset of Table Columns](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_columns.html" \l "include-exclude-cols) for an example.   * ****inherits –****A mapped class or the corresponding [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) of one indicating a superclass to which this [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) should *inherit* from. The mapped class here must be a subclass of the other mapper's class. When using Declarative, this argument is passed automatically as a result of the natural class hierarchy of the declared classes.映射类或相应的Mapper，指示此Mapper应该继承的超类。 这里的映射类必须是其他映射器类的子类。 当使用Declarative时，这个参数是被声明的类的自然类层次结构自动传递的。   **See also**  [Mapping Class Inheritance Hierarchies](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance.html)   * ****inherit\_condition**** – For joined table inheritance, a SQL expression which will define how the two tables are joined; defaults to a natural join between the two tables.对于连接表继承，一个SQL表达式将定义两个表的连接方式; 默认为两个表之间的自然连接。 * ****inherit\_foreign\_keys**** – When inherit\_condition is used and the columns present are missing a [ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey) configuration, this parameter can be used to specify which columns are "foreign". In most cases can be left as None.当使用inherit\_condition并且存在的列缺少ForeignKey配置时，该参数可用于指定哪些列为“foreign”。 在大多数情况下可以保持为无。 * ****legacy\_is\_orphan –****Boolean, defaults to False. When True, specifies that "legacy" orphan consideration is to be applied to objects mapped by this mapper, which means that a pending (that is, not persistent) object is auto-expunged from an owning [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) only when it is de-associated from *all* parents that specify a delete-orphan cascade towards this mapper. The new default behavior is that the object is auto-expunged when it is de-associated with *any* of its parents that specify delete-orphan cascade. This behavior is more consistent with that of a persistent object, and allows behavior to be consistent in more scenarios independently of whether or not an orphanable object has been flushed yet or not.布尔值，默认为False。 当为True时，指定“遗留”孤立考虑被应用于由该映射器映射的对象，这意味着只有当从该对象被取消关联时，挂起(即不持久)对象才会从拥有的会话中自动清除 所有父母指定一个删除孤立级联到这个映射器。 新的默认行为是当对象与其任何指定了delete-orphan cascade的父项关联时，该对象被自动清除。 此行为与持久对象的行为更为一致，并且允许行为在更多场景中保持一致，而不管孤立对象是否已被刷新。   See the change note and example at [The consideration of a "pending" object as an "orphan" has been made more aggressive](http://docs.sqlalchemy.org/en/rel_1_1/changelog/migration_08.html" \l "legacy-is-orphan-addition) for more detail on this change.  请参阅更改说明和示例。对于更改此更改的详细信息，将“待定”对象视为“孤儿”进行更为有效的审议。  *New in version 0.8:*- the consideration of a pending object as an "orphan" has been modified to more closely match the behavior as that of persistent objects, which is that the object is expunged from the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) as soon as it is de-associated from any of its orphan-enabled parents. Previously, the pending object would be expunged only if de-associated from all of its orphan-enabled parents. The new flag legacy\_is\_orphan is added to [orm.mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper) which re-establishes the legacy behavior.  版本0.8中的新功能：将待处理对象视为“孤立”的对象已被修改为更加紧密地匹配持久对象的行为，这就是对象在与对象关联后立即被清除 从任何一个孤儿的父母。 以前，待处理的对象将被清除，只有从所有启用了孤儿的父母取消关联。 将新标志legacy\_is\_orphan添加到orm.mapper()中，这将重新建立旧的行为。   * ****non\_primary –****Specify that this [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) is in addition to the "primary" mapper, that is, the one used for persistence. The [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) created here may be used for ad-hoc mapping of the class to an alternate selectable, for loading only.指定此映射器是除“主”映射器之外，也就是用于持久化的映射器。 此处创建的映射器可用于将该类的临时映射替换为可选择的，仅用于加载。   [Mapper.non\_primary](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper.params.non_primary" \o "sqlalchemy.orm.mapper.Mapper) is not an often used option, but is useful in some specific [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) cases.  **See also**  [Relationship to Non Primary Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/join_conditions.html" \l "relationship-non-primary-mapper)   * ****order\_by –****   A single [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) or list of [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects for which selection operations should use as the default ordering for entities. By default mappers have no pre-defined ordering.  *Deprecated since version 1.1:*The [Mapper.order\_by](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper.params.order_by" \o "sqlalchemy.orm.mapper.Mapper) parameter is deprecated. Use [Query.order\_by()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.order_by" \o "sqlalchemy.orm.query.Query.order_by) to determine the ordering of a result set.   * ****passive\_deletes –****   Indicates DELETE behavior of foreign key columns when a joined-table inheritance entity is being deleted. Defaults to False for a base mapper; for an inheriting mapper, defaults to False unless the value is set to True on the superclass mapper.  When True, it is assumed that ON DELETE CASCADE is configured on the foreign key relationships that link this mapper's table to its superclass table, so that when the unit of work attempts to delete the entity, it need only emit a DELETE statement for the superclass table, and not this table.  When False, a DELETE statement is emitted for this mapper's table individually. If the primary key attributes local to this table are unloaded, then a SELECT must be emitted in order to validate these attributes; note that the primary key columns of a joined-table subclass are not part of the "primary key" of the object as a whole.  Note that a value of True is ****always**** forced onto the subclass mappers; that is, it's not possible for a superclass to specify passive\_deletes without this taking effect for all subclass mappers.  *New in version 1.1.*  **See also**  [Using Passive Deletes](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "passive-deletes) - description of similar feature as used with [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)  [mapper.passive\_updates](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.params.passive_updates" \o "sqlalchemy.orm.mapper) - supporting ON UPDATE CASCADE for joined-table inheritance mappers   * ****passive\_updates –****   Indicates UPDATE behavior of foreign key columns when a primary key column changes on a joined-table inheritance mapping. Defaults to True.  When True, it is assumed that ON UPDATE CASCADE is configured on the foreign key in the database, and that the database will handle propagation of an UPDATE from a source column to dependent columns on joined-table rows.  When False, it is assumed that the database does not enforce referential integrity and will not be issuing its own CASCADE operation for an update. The unit of work process will emit an UPDATE statement for the dependent columns during a primary key change.  **See also**  [Mutable Primary Keys / Update Cascades](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_persistence.html" \l "passive-updates) - description of a similar feature as used with [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)  [mapper.passive\_deletes](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.params.passive_deletes" \o "sqlalchemy.orm.mapper) - supporting ON DELETE CASCADE for joined-table inheritance mappers   * ****polymorphic\_on –****   Specifies the column, attribute, or SQL expression used to determine the target class for an incoming row, when inheriting classes are present.  This value is commonly a [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) object that's present in the mapped [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table):  **class** **Employee**(Base):  \_\_tablename\_\_ = 'employee'  id = Column(Integer, primary\_key=**True**)  discriminator = Column(String(50))  \_\_mapper\_args\_\_ = {  "polymorphic\_on":discriminator,  "polymorphic\_identity":"employee"  }  It may also be specified as a SQL expression, as in this example where we use the [case()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.case" \o "sqlalchemy.sql.expression.case) construct to provide a conditional approach:  **class** **Employee**(Base):  \_\_tablename\_\_ = 'employee'  id = Column(Integer, primary\_key=**True**)  discriminator = Column(String(50))  \_\_mapper\_args\_\_ = {  "polymorphic\_on":case([  (discriminator == "EN", "engineer"),  (discriminator == "MA", "manager"),  ], else\_="employee"),  "polymorphic\_identity":"employee"  }  It may also refer to any attribute configured with [column\_property()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_columns.html" \l "sqlalchemy.orm.column_property" \o "sqlalchemy.orm.column_property), or to the string name of one:  **class** **Employee**(Base):  \_\_tablename\_\_ = 'employee'  id = Column(Integer, primary\_key=**True**)  discriminator = Column(String(50))  employee\_type = column\_property(  case([  (discriminator == "EN", "engineer"),  (discriminator == "MA", "manager"),  ], else\_="employee")  )  \_\_mapper\_args\_\_ = {  "polymorphic\_on":employee\_type,  "polymorphic\_identity":"employee"  }  *Changed in version 0.7.4:*polymorphic\_on may be specified as a SQL expression, or refer to any attribute configured with [column\_property()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_columns.html" \l "sqlalchemy.orm.column_property" \o "sqlalchemy.orm.column_property), or to the string name of one.  When setting polymorphic\_on to reference an attribute or expression that's not present in the locally mapped [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table), yet the value of the discriminator should be persisted to the database, the value of the discriminator is not automatically set on new instances; this must be handled by the user, either through manual means or via event listeners. A typical approach to establishing such a listener looks like:  当将polymorphic\_on设置为引用本地映射表中不存在的属性或表达式时，鉴别器的值应该保留在数据库中，鉴别器的值不会在新实例上自动设置; 这必须由用户通过手动方式或通过事件监听器来处理。 建立这样一个监听者的典型方法如下所示：  **from** **sqlalchemy** **import** event  **from** **sqlalchemy.orm** **import** object\_mapper  **@event**.listens\_for(Employee, "init", propagate=**True**)**def** set\_identity(instance, \*arg, \*\*kw):  mapper = object\_mapper(instance)  instance.discriminator = mapper.polymorphic\_identity  Where above, we assign the value of polymorphic\_identity for the mapped class to the discriminator attribute, thus persisting the value to the discriminator column in the database.  在上面的位置，我们将映射类的polymorphic\_identity的值分配给discriminator属性，从而将值保留在数据库中的discriminator列中。  **Warning**  Currently, ****only one discriminator column may be set****, typically on the base-most class in the hierarchy. "Cascading" polymorphic columns are not yet supported.  目前，只能设置一个鉴别符列，通常在层次结构中最基础的类上。 尚未支持“级联”多态列。  **See also**  [Mapping Class Inheritance Hierarchies](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance.html)   * ****polymorphic\_identity**** – Specifies the value which identifies this particular class as returned by the column expression referred to by the polymorphic\_on setting. As rows are received, the value corresponding to the polymorphic\_on column expression is compared to this value, indicating which subclass should be used for the newly reconstructed object.指定由polymorphic\_onsetting引用的列表达式返回的标识此特定类的值。 当接收到行时，将对应于polymorphic\_on列表达式的值与该值进行比较，指示应为新重构的对象使用哪个子类。 * ****properties**** – A dictionary mapping the string names of object attributes to [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty) instances, which define the persistence behavior of that attribute. Note that [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects present in the mapped [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) are automatically placed into ColumnProperty instances upon mapping, unless overridden. When using Declarative, this argument is passed automatically, based on all those [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty) instances declared in the declared class body.将对象属性的字符串名称映射到MapperProperty实例的字典，它定义该属性的持久性行为。 请注意，映射表中存在的列对象在映射时会自动放置到ColumnProperty实例中，除非被覆盖。 当使用Declarative时，此参数将根据声明的类体中声明的所有MapperProperty实例自动传递。 * ****primary\_key**** – A list of [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects which define the primary key to be used against this mapper's selectable unit. This is normally simply the primary key of the local\_table, but can be overridden here.列对象，定义要用于该映射器的可选单元的主键。 这通常只是local\_table的主键，但可以在这里重写。 * ****version\_id\_col –****A [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) that will be used to keep a running version id of rows in the table. This is used to detect concurrent updates or the presence of stale data in a flush. The methodology is to detect if an UPDATE statement does not match the last known version id, a [StaleDataError](http://docs.sqlalchemy.org/en/rel_1_1/orm/exceptions.html" \l "sqlalchemy.orm.exc.StaleDataError" \o "sqlalchemy.orm.exc.StaleDataError) exception is thrown. By default, the column must be of [Integer](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Integer" \o "sqlalchemy.types.Integer) type, unless version\_id\_generator specifies an alternative version generator.将用于保持表中运行的行的版本ID的列。 这用于检测并发更新或刷新中是否存在过时的数据。 该方法是检测UPDATE语句是否与最后一个已知的版本ID不匹配，抛出StaleDataError异常。 默认情况下，列必须为整数类型，除非version\_id\_generator指定替代版本生成器。   **See also**  [Configuring a Version Counter](http://docs.sqlalchemy.org/en/rel_1_1/orm/versioning.html" \l "mapper-version-counter) - discussion of version counting and rationale.   * ****version\_id\_generator –****Define how new version ids should be generated. Defaults to None, which indicates that a simple integer counting scheme be employed. To provide a custom versioning scheme, provide a callable function of the form:定义应该如何生成新版本的ID。 默认为None，表示采用简单的整数计数方案。 提供一个自定义版本控制方案，提供一个可调用的函数形式：   **def** generate\_version(version):  **return** next\_version  Alternatively, server-side versioning functions such as triggers, or programmatic versioning schemes outside of the version id generator may be used, by specifying the value False. Please see [Server Side Version Counters](http://docs.sqlalchemy.org/en/rel_1_1/orm/versioning.html" \l "server-side-version-counter) for a discussion of important points when using this option.或者，可以通过指定值False来使用服务器端版本控制功能，例如版本ID生成器之外的触发器或编程版本控制方案。 请参阅服务器端版本计数器，以便在使用此选项时讨论重点。  *New in version 0.9.0:*version\_id\_generator supports server-side version number generation.  **See also**  [Custom Version Counters / Types](http://docs.sqlalchemy.org/en/rel_1_1/orm/versioning.html" \l "custom-version-counter)  [Server Side Version Counters](http://docs.sqlalchemy.org/en/rel_1_1/orm/versioning.html" \l "server-side-version-counter)   * ****with\_polymorphic –****A tuple in the form (<classes>, <selectable>) indicating the default style of "polymorphic" loading, that is, which tables are queried at once. <classes> is any single or list of mappers and/or classes indicating the inherited classes that should be loaded at once. The special value '\*' may be used to indicate all descending classes should be loaded immediately. The second tuple argument <selectable> indicates a selectable that will be used to query for multiple classes.表单（<classes>，<selectable>）的元组指示“多态”加载的默认样式，即一次查询哪些表。 <classes>是映射器和/或类的任何单个或列表，指示应该一次加载的继承类。 特殊值'\*'可以用于指示所有下降的类应该立即加载。 第二个元组参数<selectable>表示可用于查询多个类的可选项。   **See also**  [Using with\_polymorphic](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "with-polymorphic) - discussion of polymorphic querying techniques.  使用多态查询技术的讨论。 |

sqlalchemy.orm.**object\_mapper**(*instance*)

Given an object, return the primary Mapper associated with the object instance.

Raises [sqlalchemy.orm.exc.UnmappedInstanceError](http://docs.sqlalchemy.org/en/rel_1_1/orm/exceptions.html" \l "sqlalchemy.orm.exc.UnmappedInstanceError" \o "sqlalchemy.orm.exc.UnmappedInstanceError) if no mapping is configured.

This function is available via the inspection system as:

给定一个对象，返回与对象实例关联的主Mapper。

如果未配置映射，则引发[sqlalchemy.orm.exc.UnmappedInstanceError](http://docs.sqlalchemy.org/en/rel_1_1/orm/exceptions.html" \l "sqlalchemy.orm.exc.UnmappedInstanceError" \o "sqlalchemy.orm.exc.UnmappedInstanceError)。

该功能可通过检查系统获得：

inspect(instance).mapper

Using the inspection system will raise [sqlalchemy.exc.NoInspectionAvailable](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.NoInspectionAvailable" \o "sqlalchemy.exc.NoInspectionAvailable) if the instance is not part of a mapping.

如果实例不是映射的一部分，使用检查系统会引发[sqlalchemy.exc.NoInspectionAvailable](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.NoInspectionAvailable" \o "sqlalchemy.exc.NoInspectionAvailable)。

sqlalchemy.orm.**class\_mapper**(*class\_*, *configure=True*)

Given a class, return the primary [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) associated with the key.

给定一个类，返回与该键关联的主Mapper。

Raises [UnmappedClassError](http://docs.sqlalchemy.org/en/rel_1_1/orm/exceptions.html" \l "sqlalchemy.orm.exc.UnmappedClassError" \o "sqlalchemy.orm.exc.UnmappedClassError) if no mapping is configured on the given class, or [ArgumentError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.ArgumentError" \o "sqlalchemy.exc.ArgumentError) if a non-class object is passed.

如果在给定类上未配置映射，则引发UnmappedClassError;如果传递了非类对象，则引发ArgumentError。

Equivalent functionality is available via the [inspect()](http://docs.sqlalchemy.org/en/rel_1_1/core/inspection.html" \l "sqlalchemy.inspection.inspect" \o "sqlalchemy.inspection.inspect) function as:

通过inspect()函数可以获得等效功能：

inspect(some\_mapped\_class)

Using the inspection system will raise [sqlalchemy.exc.NoInspectionAvailable](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.NoInspectionAvailable" \o "sqlalchemy.exc.NoInspectionAvailable) if the class is not mapped.

如果没有映射类，使用检查系统将会提高[sqlalchemy.exc.NoInspectionAvailable](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.NoInspectionAvailable" \o "sqlalchemy.exc.NoInspectionAvailable)。

sqlalchemy.orm.**configure\_mappers**()

Initialize the inter-mapper relationships of all mappers that have been constructed thus far.

初始化到目前为止已经构建的所有映射器之间的映射器之间的关系。

This function can be called any number of times, but in most cases is invoked automatically, the first time mappings are used, as well as whenever mappings are used and additional not-yet-configured mappers have been constructed.

该函数可以被调用任意次数，但是在大多数情况下会自动调用该函数，使用第一次映射，以及每次使用映射和构建其他尚未配置的映射器。

Points at which this occur include when a mapped class is instantiated into an instance, as well as when the [Session.query()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.query" \o "sqlalchemy.orm.session.Session.query) method is used.

发生这种情况的点包括将映射类实例化为实例时，以及何时使用Session.query()方法。

The [configure\_mappers()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.configure_mappers" \o "sqlalchemy.orm.configure_mappers) function provides several event hooks that can be used to augment its functionality. These methods include:

configure\_mappers()函数提供了几个可用于增强其功能的事件钩子。 这些方法包括：

* [MapperEvents.before\_configured()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents.before_configured" \o "sqlalchemy.orm.events.MapperEvents.before_configured) - called once before [configure\_mappers()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.configure_mappers" \o "sqlalchemy.orm.configure_mappers) does any work; this can be used to establish additional options, properties, or related mappings before the operation proceeds.在configure\_mappers()执行任何工作之前调用一次; 这可以用于在操作进行之前建立附加选项，属性或相关映射。
* [MapperEvents.mapper\_configured()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents.mapper_configured" \o "sqlalchemy.orm.events.MapperEvents.mapper_configured) - called as each indivudal [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) is configured within the process; will include all mapper state except for backrefs set up by other mappers that are still to be configured.在进程中配置每个单独的Mapper时调用; 将包括除了仍被配置的其他映射器设置的backref之外的所有映射器状态。
* [MapperEvents.after\_configured()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents.after_configured" \o "sqlalchemy.orm.events.MapperEvents.after_configured) - called once after [configure\_mappers()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.configure_mappers" \o "sqlalchemy.orm.configure_mappers) is complete; at this stage, all [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) objects that are known to SQLAlchemy will be fully configured. Note that the calling application may still have other mappings that haven't been produced yet, such as if they are in modules as yet unimported.configure\_mappers()完成后调用一次; 在此阶段，SQLAlchemy已知的所有Mapper对象都将被完全配置。 请注意，调用应用程序可能仍然具有尚未生成的其他映射，例如，如果它们在尚未未导入的模块中。

sqlalchemy.orm.**clear\_mappers**()

Remove all mappers from all classes.

从所有类中删除所有映射器。

This function removes all instrumentation from classes and disposes of their associated mappers. Once called, the classes are unmapped and can be later re-mapped with new mappers.

此功能将所有仪器从类中删除并处理其关联的映射器。 一旦被调用，这些类就被取消映射，并且可以随后用新的映射器重新映射。

[clear\_mappers()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.clear_mappers" \o "sqlalchemy.orm.clear_mappers) is *not* for normal use, as there is literally no valid usage for it outside of very specific testing scenarios. Normally, mappers are permanent structural components of user-defined classes, and are never discarded independently of their class. If a mapped class itself is garbage collected, its mapper is automatically disposed of as well. As such, [clear\_mappers()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.clear_mappers" \o "sqlalchemy.orm.clear_mappers) is only for usage in test suites that re-use the same classes with different mappings, which is itself an extremely rare use case - the only such use case is in fact SQLAlchemy's own test suite, and possibly the test suites of other ORM extension libraries which intend to test various combinations of mapper construction upon a fixed set of classes.

clear\_mappers()不是正常使用的，因为在非常特定的测试场景之外，它没有任何有效的用法。 通常，映射器是用户定义类的永久结构组件，并且不会独立于其类丢弃。 如果一个映射类本身是垃圾回收的，它的映射器也被自动处理。 因此，clear\_mappers()仅用于在使用不同映射的相同类的测试套件中使用，这本身就是非常罕见的用例 - 实际上SQLAlchemy自己的测试套件只有这样的用例，也可能是测试 其他ORM扩展库的套件，其旨在测试固定的一组类的映射器构造的各种组合。

sqlalchemy.orm.util.**identity\_key**(*\*args*, *\*\*kwargs*)

Generate "identity key" tuples, as are used as keys in the [Session.identity\_map](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.identity_map" \o "sqlalchemy.orm.session.Session.identity_map) dictionary.

生成“identity key”元组，用作Session.identity\_map字典中的键。

This function has several call styles:

此函数有几种调用风格：

identity\_key(class, ident)

This form receives a mapped class and a primary key scalar or tuple as an argument.此表单接收映射类和主键标量或元组作为参数。

E.g.:

**>>>** identity\_key(MyClass, (1, 2))(<class '\_\_main\_\_.MyClass'>, (1, 2))

|  |  |
| --- | --- |
| **param class:** | mapped class (must be a positional argument)映射类（必须是位置参数） |
| **param ident:** | primary key, may be a scalar or tuple argument.主键，可以是标量或元组参数。 |

identity\_key(instance=instance)

This form will produce the identity key for a given instance. The instance need not be persistent, only that its primary key attributes are populated (else the key will contain None for those missing values).此表单将为给定实例生成身份密钥。 该实例不必是持久的，只是它的主键属性被填充（否则键将包含那些缺少的值）。

E.g.:

**>>>** instance = MyClass(1, 2)

**>>>** identity\_key(instance=instance)(<class '\_\_main\_\_.MyClass'>, (1, 2))

In this form, the given instance is ultimately run though [Mapper.identity\_key\_from\_instance()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper.identity_key_from_instance" \o "sqlalchemy.orm.mapper.Mapper.identity_key_from_instance), which will have the effect of performing a database check for the corresponding row if the object is expired.在这种形式中，给定的实例最终通过Mapper.identity\_key\_from\_instance()运行，如果对象已过期，它将对相应行执行数据库检查。

|  |  |
| --- | --- |
| **param instance:** | object instance (must be given as a keyword arg) |

identity\_key(class, row=row)

This form is similar to the class/tuple form, except is passed a database result row as a [RowProxy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.RowProxy" \o "sqlalchemy.engine.RowProxy) object.此表单类似于类/元组形式，除了将数据库结果行作为RowProxy对象传递。

E.g.:

**>>>** row = engine.execute("select \* from table where a=1 and b=2").first()

**>>>** identity\_key(MyClass, row=row)(<class '\_\_main\_\_.MyClass'>, (1, 2))

|  |  |
| --- | --- |
| **param class:** | mapped class (must be a positional argument)映射类（必须是位置参数） |
| **param row:** | [RowProxy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.RowProxy" \o "sqlalchemy.engine.RowProxy) row returned by a [ResultProxy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy" \o "sqlalchemy.engine.ResultProxy) (must be given as a keyword arg)  一个ResultProxy返回的RowProxy行（必须以关键字arg的形式给出） |

sqlalchemy.orm.util.**polymorphic\_union**(*table\_map*, *typecolname*, *aliasname='p\_union'*, *cast\_nulls=True*)

Create a UNION statement used by a polymorphic mapper.

创建多态映射器使用的UNION语句。

See [Concrete Table Inheritance](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance.html" \l "concrete-inheritance) for an example of how this is used.

有关如何使用该示例的示例，请参阅具体表继承。

|  |  |
| --- | --- |
| **Parameters:** | * ****table\_map**** – mapping of polymorphic identities to [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects.将多态标识映射到表对象。 * ****typecolname**** – string name of a "discriminator" column, which will be derived from the query, producing the polymorphic identity for each row. If None, no polymorphic discriminator is generated.一个“鉴别器”列的字符串名称，它将从查询中派生，产生每行的多态标识。 如果None，则不会生成多态鉴别器。 * ****aliasname**** – name of the [alias()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.alias" \o "sqlalchemy.sql.expression.alias) construct generated.生成[alias()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.alias" \o "sqlalchemy.sql.expression.alias)结构的名称。 * ****cast\_nulls**** – if True, non-existent columns, which are represented as labeled NULLs, will be passed into CAST. This is a legacy behavior that is problematic on some backends such as Oracle - in which case it can be set to False.如果为True，表示为标记为NULL的不存在的列将被传递到CAST中。 这是在某些后端（如Oracle）上存在问题的遗留行为 - 在这种情况下，它可以设置为False。 |

*class*sqlalchemy.orm.mapper.**Mapper**(*class\_*, *local\_table=None*, *properties=None*, *primary\_key=None*, *non\_primary=False*, *inherits=None*, *inherit\_condition=None*, *inherit\_foreign\_keys=None*, *extension=None*, *order\_by=False*, *always\_refresh=False*, *version\_id\_col=None*, *version\_id\_generator=None*, *polymorphic\_on=None*, *\_polymorphic\_map=None*, *polymorphic\_identity=None*, *concrete=False*, *with\_polymorphic=None*, *allow\_partial\_pks=True*, *batch=True*, *column\_prefix=None*, *include\_properties=None*, *exclude\_properties=None*, *passive\_updates=True*, *passive\_deletes=False*, *confirm\_deleted\_rows=True*, *eager\_defaults=False*, *legacy\_is\_orphan=False*, *\_compiled\_cache\_size=100*)

Bases: [sqlalchemy.orm.base.InspectionAttr](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.base.InspectionAttr" \o "sqlalchemy.orm.base.InspectionAttr)

Define the correlation of class attributes to database table columns.

定义类属性与数据库表列的相关性。

The [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) object is instantiated using the [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper) function. For information about instantiating new [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) objects, see that function's documentation.

Mapper对象使用mapper()函数实例化。 有关实例化新的Mapper对象的信息，请参阅该函数的文档。

When [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper) is used explicitly to link a user defined class with table metadata, this is referred to as *classical mapping*. Modern SQLAlchemy usage tends to favor the [sqlalchemy.ext.declarative](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "module-sqlalchemy.ext.declarative" \o "sqlalchemy.ext.declarative) extension for class configuration, which makes usage of [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper) behind the scenes.

当显式地使用mapper()将用户定义的类与表元数据链接时，这被称为经典映射。 现代SQLAlchemy使用往往倾向于类配置的sqlalchemy.ext.declarative扩展，这使得在场景中使用mapper()。

Given a particular class known to be mapped by the ORM, the [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) which maintains it can be acquired using the [inspect()](http://docs.sqlalchemy.org/en/rel_1_1/core/inspection.html" \l "sqlalchemy.inspection.inspect" \o "sqlalchemy.inspection.inspect) function:

给定一个已知由ORM映射的特定类，可以使用inspect()函数获取维护它的映射器：

**from** **sqlalchemy** **import** inspect

mapper = inspect(MyClass)

A class which was mapped by the [sqlalchemy.ext.declarative](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "module-sqlalchemy.ext.declarative" \o "sqlalchemy.ext.declarative) extension will also have its mapper available via the \_\_mapper\_\_ attribute.

由[sqlalchemy.ext.declarative](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "module-sqlalchemy.ext.declarative" \o "sqlalchemy.ext.declarative)扩展映射的类也将通过\_\_mapper\_\_属性使其映射器可用。

**\_\_init\_\_**(*class\_*, *local\_table=None*, *properties=None*, *primary\_key=None*, *non\_primary=False*, *inherits=None*, *inherit\_condition=None*, *inherit\_foreign\_keys=None*, *extension=None*, *order\_by=False*, *always\_refresh=False*, *version\_id\_col=None*, *version\_id\_generator=None*, *polymorphic\_on=None*, *\_polymorphic\_map=None*, *polymorphic\_identity=None*, *concrete=False*, *with\_polymorphic=None*, *allow\_partial\_pks=True*, *batch=True*, *column\_prefix=None*, *include\_properties=None*, *exclude\_properties=None*, *passive\_updates=True*, *passive\_deletes=False*, *confirm\_deleted\_rows=True*, *eager\_defaults=False*, *legacy\_is\_orphan=False*, *\_compiled\_cache\_size=100*)

Construct a new [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) object.

构造一个新的Mapper对象。

This constructor is mirrored as a public API function; see [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper) for a full usage and argument description.

该构造函数作为公共API函数进行镜像; 请参阅mapper()以获取完整的用法和参数描述。

**add\_properties**(*dict\_of\_properties*)

Add the given dictionary of properties to this mapper, using add\_property.

使用add\_property将给定的属性字典添加到此映射器。

**add\_property**(*key*, *prop*)

Add an individual MapperProperty to this mapper.

将一个单独的MapperProperty添加到此映射器。

If the mapper has not been configured yet, just adds the property to the initial properties dictionary sent to the constructor. If this Mapper has already been configured, then the given MapperProperty is configured immediately.

如果映射器尚未配置，只需将属性添加到发送到构造函数的初始属性字典。 如果此Mapper已配置，则给定的MapperProperty将立即配置。

**all\_orm\_descriptors**

A namespace of all [InspectionAttr](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.base.InspectionAttr" \o "sqlalchemy.orm.base.InspectionAttr) attributes associated with the mapped class.

与映射类相关联的所有InspectionAttr属性的命名空间。

These attributes are in all cases Python [descriptors](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-descriptors) associated with the mapped class or its superclasses.

这些属性在所有情况下与映射类或其超类相关联的Python描述符。

This namespace includes attributes that are mapped to the class as well as attributes declared by extension modules. It includes any Python descriptor type that inherits from [InspectionAttr](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.base.InspectionAttr" \o "sqlalchemy.orm.base.InspectionAttr). This includes [QueryableAttribute](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.attributes.QueryableAttribute" \o "sqlalchemy.orm.attributes.QueryableAttribute), as well as extension types such as [hybrid\_property](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/hybrid.html" \l "sqlalchemy.ext.hybrid.hybrid_property" \o "sqlalchemy.ext.hybrid.hybrid_property), [hybrid\_method](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/hybrid.html" \l "sqlalchemy.ext.hybrid.hybrid_method" \o "sqlalchemy.ext.hybrid.hybrid_method) and [AssociationProxy](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.AssociationProxy" \o "sqlalchemy.ext.associationproxy.AssociationProxy).

该命名空间包括映射到类的属性以及扩展模块声明的属性。 它包括从InspectionAttr继承的任何Python描述符类型。 这包括QueryableAttribute，以及扩展类型，如hybrid\_property，hybrid\_method和AssociationProxy。

To distinguish between mapped attributes and extension attributes, the attribute [InspectionAttr.extension\_type](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.base.InspectionAttr.extension_type" \o "sqlalchemy.orm.base.InspectionAttr.extension_type) will refer to a constant that distinguishes between different extension types.

为了区分映射属性和扩展属性，InspectionAttr.extension\_type属性将引用一个常量来区分不同的扩展类型。

When dealing with a [QueryableAttribute](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.attributes.QueryableAttribute" \o "sqlalchemy.orm.attributes.QueryableAttribute), the [QueryableAttribute.property](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.attributes.QueryableAttribute.property" \o "sqlalchemy.orm.attributes.QueryableAttribute.property) attribute refers to the [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty) property, which is what you get when referring to the collection of mapped properties via [Mapper.attrs](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper.attrs" \o "sqlalchemy.orm.mapper.Mapper.attrs).

当处理QueryableAttribute时，QueryableAttribute.property属性引用MapperProperty属性，这是通过Mapper.attrs引用映射属性集合时获得的属性。

**Warning**

The [Mapper.all\_orm\_descriptors](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper.all_orm_descriptors" \o "sqlalchemy.orm.mapper.Mapper.all_orm_descriptors) accessor namespace is an instance of OrderedProperties. This is a dictionary-like object which includes a small number of named methods such as OrderedProperties.items() and OrderedProperties.values(). When accessing attributes dynamically, favor using the dict-access scheme, e.g. mapper.all\_orm\_descriptors[somename] over getattr(mapper.all\_orm\_descriptors, somename) to avoid name collisions.

Mapper.all\_orm\_descriptors访问器命名空间是OrderedProperties的一个实例。 这是一个类似字典的对象，其中包含少量命名方法，如OrderedProperties.items()和OrderedProperties.values()。 当动态访问属性时，有利于使用dict-access方案，例如 mapper.all\_orm\_descriptors [somename] over getattr（mapper.all\_orm\_descriptors，somename）以避免名称冲突。

*New in version 0.8.0.*

**See also**

[Mapper.attrs](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper.attrs" \o "sqlalchemy.orm.mapper.Mapper.attrs)

**attrs**

A namespace of all [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty) objects associated this mapper.

与该映射器相关联的所有[MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty)对象的命名空间。

This is an object that provides each property based on its key name. For instance, the mapper for a User class which has User.name attribute would provide mapper.attrs.name, which would be the [ColumnProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.ColumnProperty" \o "sqlalchemy.orm.properties.ColumnProperty) representing the name column. The namespace object can also be iterated, which would yield each [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty).

这是一个基于其名称提供每个属性的对象。 例如，具有User.name属性的User类的映射器将提供mapper.attrs.name，它将是表示名称列的[ColumnProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.ColumnProperty" \o "sqlalchemy.orm.properties.ColumnProperty)。 命名空间对象也可以迭代，这将产生每个[MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty)。

[Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) has several pre-filtered views of this attribute which limit the types of properties returned, inclding [synonyms](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper.synonyms" \o "sqlalchemy.orm.mapper.Mapper.synonyms), [column\_attrs](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper.column_attrs" \o "sqlalchemy.orm.mapper.Mapper.column_attrs),[relationships](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper.relationships" \o "sqlalchemy.orm.mapper.Mapper.relationships), and [composites](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper.composites" \o "sqlalchemy.orm.mapper.Mapper.composites).

Mapper具有此属性的几个预筛选视图，限制返回的属性类型，包含[synonyms](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper.synonyms" \o "sqlalchemy.orm.mapper.Mapper.synonyms)，[column\_attrs](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper.column_attrs" \o "sqlalchemy.orm.mapper.Mapper.column_attrs)，[relationships](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper.relationships" \o "sqlalchemy.orm.mapper.Mapper.relationships)和[composites](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper.composites" \o "sqlalchemy.orm.mapper.Mapper.composites)。

**Warning**

The [Mapper.attrs](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper.attrs" \o "sqlalchemy.orm.mapper.Mapper.attrs) accessor namespace is an instance of OrderedProperties. This is a dictionary-like object which includes a small number of named methods such as OrderedProperties.items() and OrderedProperties.values(). When accessing attributes dynamically, favor using the dict-access scheme, e.g. mapper.attrs[somename] over getattr(mapper.attrs, somename) to avoid name collisions.

Mapper.attrs访问器命名空间是OrderedProperties的一个实例。 这是一个类似字典的对象，其中包含少量命名方法，如OrderedProperties.items()和OrderedProperties.values()。 当动态访问属性时，有利于使用dict-access方案，例如mapper.attrs[somename]over getattr(mapper.attrs, somename)以避免名称冲突。

**See also**

[Mapper.all\_orm\_descriptors](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper.all_orm_descriptors" \o "sqlalchemy.orm.mapper.Mapper.all_orm_descriptors)

**base\_mapper***= None*

The base-most [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) in an inheritance chain.

In a non-inheriting scenario, this attribute will always be this [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper). In an inheritance scenario, it references the [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) which is parent to all other [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) objects in the inheritance chain.

This is a *read only* attribute determined during mapper construction. Behavior is undefined if directly modified.

**c***= None*

A synonym for [columns](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper.columns" \o "sqlalchemy.orm.mapper.Mapper.columns).

**cascade\_iterator**(*type\_*, *state*, *halt\_on=None*)

Iterate each element and its mapper in an object graph, for all relationships that meet the given cascade rule.

|  |  |
| --- | --- |
| **Parameters:** | * ****type\_ –****The name of the cascade rule (i.e. "save-update", "delete", etc.).   **Note**  the "all" cascade is not accepted here. For a generic object traversal function, see [How do I walk all objects that are related to a given object?](http://docs.sqlalchemy.org/en/rel_1_1/faq/sessions.html" \l "faq-walk-objects).  “全部”级联不被接受。 对于通用对象遍历函数，请参阅如何处理与给定对象相关的所有对象？   * ****state –**** The lead InstanceState. child items will be processed per the relationships defined for this object's mapper. |
| **Returns:** | the method yields individual object instances. |

**See also**

[Cascades](http://docs.sqlalchemy.org/en/rel_1_1/orm/cascades.html" \l "unitofwork-cascades)

[How do I walk all objects that are related to a given object?](http://docs.sqlalchemy.org/en/rel_1_1/faq/sessions.html" \l "faq-walk-objects) - illustrates a generic function to traverse all objects without relying on cascades.

**class\_***= None*

The Python class which this [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) maps.

This is a *read only* attribute determined during mapper construction. Behavior is undefined if directly modified.

**class\_manager***= None*

The [ClassManager](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.instrumentation.ClassManager" \o "sqlalchemy.orm.instrumentation.ClassManager) which maintains event listeners and class-bound descriptors for this [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper).

This is a *read only* attribute determined during mapper construction. Behavior is undefined if directly modified.

**column\_attrs**

Return a namespace of all [ColumnProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.ColumnProperty" \o "sqlalchemy.orm.properties.ColumnProperty) properties maintained by this [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper).

**See also**

[Mapper.attrs](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper.attrs" \o "sqlalchemy.orm.mapper.Mapper.attrs) - namespace of all [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty) objects.

**columns***= None*

A collection of [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) or other scalar expression objects maintained by this [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper).

The collection behaves the same as that of the c attribute on any [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object, except that only those columns included in this mapping are present, and are keyed based on the attribute name defined in the mapping, not necessarily the key attribute of the [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) itself. Additionally, scalar expressions mapped by [column\_property()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_columns.html" \l "sqlalchemy.orm.column_property" \o "sqlalchemy.orm.column_property) are also present here.

This is a *read only* attribute determined during mapper construction. Behavior is undefined if directly modified.

**common\_parent**(*other*)

Return true if the given mapper shares a common inherited parent as this mapper.

**composites**

Return a namespace of all [CompositeProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.descriptor_props.CompositeProperty" \o "sqlalchemy.orm.descriptor_props.CompositeProperty) properties maintained by this [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper).

**See also**

[Mapper.attrs](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper.attrs" \o "sqlalchemy.orm.mapper.Mapper.attrs) - namespace of all [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty) objects.

**concrete***= None*

Represent True if this [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) is a concrete inheritance mapper.

This is a *read only* attribute determined during mapper construction. Behavior is undefined if directly modified.

**configured***= None*

Represent True if this [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) has been configured.

This is a *read only* attribute determined during mapper construction. Behavior is undefined if directly modified.

**See also**

[configure\_mappers()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.configure_mappers" \o "sqlalchemy.orm.configure_mappers).

**entity**

Part of the inspection API.

Returns self.class\_.

**get\_property**(*key*, *\_configure\_mappers=True*)

return a MapperProperty associated with the given key.

**get\_property\_by\_column**(*column*)

Given a [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) object, return the [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty) which maps this column.

**identity\_key\_from\_instance**(*instance*)

Return the identity key for the given instance, based on its primary key attributes.

根据其主键属性返回给定实例的标识密钥。

If the instance's state is expired, calling this method will result in a database check to see if the object has been deleted. If the row no longer exists,[ObjectDeletedError](http://docs.sqlalchemy.org/en/rel_1_1/orm/exceptions.html" \l "sqlalchemy.orm.exc.ObjectDeletedError" \o "sqlalchemy.orm.exc.ObjectDeletedError) is raised.

如果实例的状态已过期，则调用此方法将导致数据库检查以查看对象是否已被删除。 如果行不再存在，则引发ObjectDeletedError。

This value is typically also found on the instance state under the attribute name key.

该值通常也在属性名称键下的实例状态下找到。

**identity\_key\_from\_primary\_key**(*primary\_key*)

Return an identity-map key for use in storing/retrieving an item from an identity map.

|  |  |
| --- | --- |
| **Parameters:** | ****primary\_key**** – A list of values indicating the identifier. |

**identity\_key\_from\_row**(*row*, *adapter=None*)

Return an identity-map key for use in storing/retrieving an item from the identity map.

|  |  |
| --- | --- |
| **Parameters:** | ****row**** – A [RowProxy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.RowProxy" \o "sqlalchemy.engine.RowProxy) instance. The columns which are mapped by this [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) should be locatable in the row, preferably via the [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) object directly (as is the case when a [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) construct is executed), or via string names of the form <tablename>\_<colname>. |

**inherits***= None*

References the [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) which this [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) inherits from, if any.

This is a *read only* attribute determined during mapper construction. Behavior is undefined if directly modified.

**is\_mapper***= True*

Part of the inspection API.

**isa**(*other*)

Return True if the this mapper inherits from the given mapper.

**iterate\_properties**

return an iterator of all MapperProperty objects.

**local\_table***= None*

The [Selectable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Selectable" \o "sqlalchemy.sql.expression.Selectable) which this [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) manages.

该[Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper)管理的[Selectable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Selectable" \o "sqlalchemy.sql.expression.Selectable)。

Typically is an instance of [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) or [Alias](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Alias" \o "sqlalchemy.sql.expression.Alias). May also be None.

通常是表或别名的实例。 也可以是无。

The "local" table is the selectable that the [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) is directly responsible for managing from an attribute access and flush perspective. For non-inheriting mappers, the local table is the same as the "mapped" table. For joined-table inheritance mappers, local\_table will be the particular sub-table of the overall "join" which this [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) represents. If this mapper is a single-table inheriting mapper, local\_table will be None.

“本地”表是Mapper直接负责从属性访问和刷新角度进行管理的可选择。 对于非继承映射器，本地表与“映射”表相同。 对于连接表继承映射器，local\_table将是此Mapper表示的整体“join”的特定子表。 如果此映射器是单表继承映射器，则local\_table将为“无”。

**See also**

[mapped\_table](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper.mapped_table" \o "sqlalchemy.orm.mapper.Mapper.mapped_table).

**mapped\_table***= None*

The [Selectable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Selectable" \o "sqlalchemy.sql.expression.Selectable) to which this [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) is mapped.

该[Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper)管理的[Selectable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Selectable" \o "sqlalchemy.sql.expression.Selectable)。

Typically an instance of [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table), [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join), or [Alias](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Alias" \o "sqlalchemy.sql.expression.Alias).

The "mapped" table is the selectable that the mapper selects from during queries. For non-inheriting mappers, the mapped table is the same as the "local" table. For joined-table inheritance mappers, mapped\_table references the full [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) representing full rows for this particular subclass. For single-table inheritance mappers, mapped\_table references the base table.

**See also**

[local\_table](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper.local_table" \o "sqlalchemy.orm.mapper.Mapper.local_table).

**mapper**

Part of the inspection API.

Returns self.

**non\_primary***= None*

Represent True if this [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) is a "non-primary" mapper, e.g. a mapper that is used only to selet rows but not for persistence management.

如果此Mapper是“非主要”映射器，则表示True。 一个仅用于搜索行而不用于持久性管理的映射器。

This is a *read only* attribute determined during mapper construction. Behavior is undefined if directly modified.

这是在映射器构造期间确定的只读属性。 直接修改行为未定义。

**polymorphic\_identity***= None*

Represent an identifier which is matched against the [polymorphic\_on](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper.polymorphic_on" \o "sqlalchemy.orm.mapper.Mapper.polymorphic_on) column during result row loading.

表示在结果行加载期间与polymorphic\_on列匹配的标识符。

Used only with inheritance, this object can be of any type which is comparable to the type of column represented by [polymorphic\_on](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper.polymorphic_on" \o "sqlalchemy.orm.mapper.Mapper.polymorphic_on).

仅用于继承，该对象可以是任何类型，可以类似于由polymorphic\_on表示的列的类型。

This is a *read only* attribute determined during mapper construction. Behavior is undefined if directly modified.

这是在映射器构造期间确定的只读属性。 直接修改行为未定义。

**polymorphic\_iterator**()

Iterate through the collection including this mapper and all descendant mappers.

This includes not just the immediately inheriting mappers but all their inheriting mappers as well.

To iterate through an entire hierarchy, use mapper.base\_mapper.polymorphic\_iterator().

**polymorphic\_map***= None*

A mapping of "polymorphic identity" identifiers mapped to [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) instances, within an inheritance scenario.

The identifiers can be of any type which is comparable to the type of column represented by [polymorphic\_on](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper.polymorphic_on" \o "sqlalchemy.orm.mapper.Mapper.polymorphic_on).

An inheritance chain of mappers will all reference the same polymorphic map object. The object is used to correlate incoming result rows to target mappers.

This is a *read only* attribute determined during mapper construction. Behavior is undefined if directly modified.

**polymorphic\_on***= None*

The [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) or SQL expression specified as the polymorphic\_on argument for this [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper), within an inheritance scenario.

This attribute is normally a [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) instance but may also be an expression, such as one derived from [cast()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.cast" \o "sqlalchemy.sql.expression.cast).

This is a *read only* attribute determined during mapper construction. Behavior is undefined if directly modified.

**primary\_key***= None*

An iterable containing the collection of [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects which comprise the 'primary key' of the mapped table, from the perspective of this [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper).

This list is against the selectable in [mapped\_table](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper.mapped_table" \o "sqlalchemy.orm.mapper.Mapper.mapped_table). In the case of inheriting mappers, some columns may be managed by a superclass mapper. For example, in the case of a [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join), the primary key is determined by all of the primary key columns across all tables referenced by the [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join).

The list is also not necessarily the same as the primary key column collection associated with the underlying tables; the [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) features a primary\_key argument that can override what the [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) considers as primary key columns.

This is a *read only* attribute determined during mapper construction. Behavior is undefined if directly modified.

**primary\_key\_from\_instance**(*instance*)

Return the list of primary key values for the given instance.

If the instance's state is expired, calling this method will result in a database check to see if the object has been deleted. If the row no longer exists,[ObjectDeletedError](http://docs.sqlalchemy.org/en/rel_1_1/orm/exceptions.html" \l "sqlalchemy.orm.exc.ObjectDeletedError" \o "sqlalchemy.orm.exc.ObjectDeletedError) is raised.

**primary\_mapper**()

Return the primary mapper corresponding to this mapper's class key (class).

**relationships**

A namespace of all [RelationshipProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty" \o "sqlalchemy.orm.properties.RelationshipProperty) properties maintained by this [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper).

**Warning**

the [Mapper.relationships](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper.relationships" \o "sqlalchemy.orm.mapper.Mapper.relationships) accessor namespace is an instance of OrderedProperties. This is a dictionary-like object which includes a small number of named methods such as OrderedProperties.items() and OrderedProperties.values(). When accessing attributes dynamically, favor using the dict-access scheme, e.g. mapper.relationships[somename] over getattr(mapper.relationships, somename) to avoid name collisions.

**See also**

[Mapper.attrs](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper.attrs" \o "sqlalchemy.orm.mapper.Mapper.attrs) - namespace of all [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty) objects.

**selectable**

The [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) construct this [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) selects from by default.

Normally, this is equivalent to [mapped\_table](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper.mapped_table" \o "sqlalchemy.orm.mapper.Mapper.mapped_table), unless the with\_polymorphic feature is in use, in which case the full "polymorphic" selectable is returned.

**self\_and\_descendants**

The collection including this mapper and all descendant mappers.

This includes not just the immediately inheriting mappers but all their inheriting mappers as well.

**single***= None*

Represent True if this [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) is a single table inheritance mapper.

[local\_table](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper.local_table" \o "sqlalchemy.orm.mapper.Mapper.local_table) will be None if this flag is set.

This is a *read only* attribute determined during mapper construction. Behavior is undefined if directly modified.

**synonyms**

Return a namespace of all [SynonymProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.descriptor_props.SynonymProperty" \o "sqlalchemy.orm.descriptor_props.SynonymProperty) properties maintained by this [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper).

**See also**

[Mapper.attrs](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper.attrs" \o "sqlalchemy.orm.mapper.Mapper.attrs) - namespace of all [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty) objects.

**tables***= None*

An iterable containing the collection of [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects which this [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) is aware of.

If the mapper is mapped to a [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join), or an [Alias](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Alias" \o "sqlalchemy.sql.expression.Alias) representing a [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select), the individual [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects that comprise the full construct will be represented here.

This is a *read only* attribute determined during mapper construction. Behavior is undefined if directly modified.

**validators***= None*

An immutable dictionary of attributes which have been decorated using the [validates()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapped_attributes.html" \l "sqlalchemy.orm.validates" \o "sqlalchemy.orm.validates) decorator.

The dictionary contains string attribute names as keys mapped to the actual validation method.

**with\_polymorphic\_mappers**

The list of [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) objects included in the default "polymorphic" query.

# 

# Chapter 3 Relationship Configuration

This section describes the [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) function and in depth discussion of its usage. For an introduction to relationships, start with the [Object Relational Tutorial](http://docs.sqlalchemy.org/en/rel_1_1/orm/tutorial.html) and head into [Building a Relationship](http://docs.sqlalchemy.org/en/rel_1_1/orm/tutorial.html" \l "orm-tutorial-relationship).

本节介绍relationship()函数及其使用的深入讨论。 对于关系的介绍，从对象关系教程开始，并开始建立关系。

## 3.1 Basic Relationship Patterns

A quick walkthrough of the basic relational patterns.

The imports used for each of the following sections is as follows:

**from** **sqlalchemy** **import** Table, Column, Integer, ForeignKey

**from** **sqlalchemy.orm** **import** relationship

**from** **sqlalchemy.ext.declarative** **import** declarative\_base

Base = declarative\_base()

### 3.1.1 One To Many

A one to many relationship places a foreign key on the child table referencing the parent. [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) is then specified on the parent, as referencing a collection of items represented by the child:

一对多关系会在引用父项的子表上放置一个外键。 然后在父项上指定relationship()，引用由子代表的项目的集合：

**class** **Parent**(Base):

\_\_tablename\_\_ = 'parent'

id = Column(Integer, primary\_key=**True**)

children = relationship("Child")

**class** **Child**(Base):

\_\_tablename\_\_ = 'child'

id = Column(Integer, primary\_key=**True**)

parent\_id = Column(Integer, ForeignKey('parent.id'))

To establish a bidirectional relationship in one-to-many, where the "reverse" side is a many to one, specify an additional [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) and connect the two using the [relationship.back\_populates](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.back_populates" \o "sqlalchemy.orm.relationship) parameter:

要建立一对多的双向关系，其中"反向"方面是多对一的，请指定一个附加relationship()并使用[relationship.back\_populates](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.back_populates" \o "sqlalchemy.orm.relationship)参数连接两个关系：

**class** **Parent**(Base):

\_\_tablename\_\_ = 'parent'

id = Column(Integer, primary\_key=**True**)

children = relationship("Child", back\_populates="parent")

**class** **Child**(Base):

\_\_tablename\_\_ = 'child'

id = Column(Integer, primary\_key=**True**)

parent\_id = Column(Integer, ForeignKey('parent.id'))

parent = relationship("Parent", back\_populates="children")

Child will get a parent attribute with many-to-one semantics.

Child将获得具有多对一语义的parent属性。

Alternatively, the [backref](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.backref" \o "sqlalchemy.orm.relationship) option may be used on a single [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) instead of using [back\_populates](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.back_populates" \o "sqlalchemy.orm.relationship):

或者，[backref](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.backref" \o "sqlalchemy.orm.relationship)选项可以用于单个[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)而不是使用back\_populates：

**class** **Parent**(Base):

\_\_tablename\_\_ = 'parent'

id = Column(Integer, primary\_key=**True**)

children = relationship("Child", backref="parent")

### 3.1.2 Many To One

Many to one places a foreign key in the parent table referencing the child. [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) is declared on the parent, where a new scalar-holding attribute will be created:

多对一地方在引用子项的父表中放置一个外键。 [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)在父项上声明，其中将创建一个新的标量保持属性：

**class** **Parent**(Base):

\_\_tablename\_\_ = 'parent'

id = Column(Integer, primary\_key=**True**)

child\_id = Column(Integer, ForeignKey('child.id'))

child = relationship("Child")

**class** **Child**(Base):

\_\_tablename\_\_ = 'child'

id = Column(Integer, primary\_key=**True**)

Bidirectional behavior is achieved by adding a second [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) and applying the [relationship.back\_populates](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.back_populates" \o "sqlalchemy.orm.relationship) parameter in both directions:

通过添加第二个relationship()并在两个方向应用[relationship.back\_populates](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.back_populates" \o "sqlalchemy.orm.relationship)参数来实现双向行为：

**class** **Parent**(Base):

\_\_tablename\_\_ = 'parent'

id = Column(Integer, primary\_key=**True**)

child\_id = Column(Integer, ForeignKey('child.id'))

child = relationship("Child", back\_populates="parents")

**class** **Child**(Base):

\_\_tablename\_\_ = 'child'

id = Column(Integer, primary\_key=**True**)

parents = relationship("Parent", back\_populates="child")

Alternatively, the [backref](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.backref" \o "sqlalchemy.orm.relationship) parameter may be applied to a single [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship), such as Parent.child:

**class** **Parent**(Base):

\_\_tablename\_\_ = 'parent'

id = Column(Integer, primary\_key=**True**)

child\_id = Column(Integer, ForeignKey('child.id'))

child = relationship("Child", backref="parents")

### 3.1.3 One To One

One To One is essentially a bidirectional relationship with a scalar attribute on both sides. To achieve this, the [uselist](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.uselist" \o "sqlalchemy.orm.relationship) flag indicates the placement of a scalar attribute instead of a collection on the "many" side of the relationship. To convert one-to-many into one-to-one:

一对一本质上是与两侧的标量属性的双向关系。 为了实现这一点，[uselist](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.uselist" \o "sqlalchemy.orm.relationship)标志指示在关系的"多个"侧上放置一个标量属性而不是一个集合。 将一对多转换为一对一：

**class** **Parent**(Base):

\_\_tablename\_\_ = 'parent'

id = Column(Integer, primary\_key=**True**)

child = relationship("Child", uselist=**False**, back\_populates="parent")

**class** **Child**(Base):

\_\_tablename\_\_ = 'child'

id = Column(Integer, primary\_key=**True**)

parent\_id = Column(Integer, ForeignKey('parent.id'))

parent = relationship("Parent", back\_populates="child")

Or for many-to-one:

**class** **Parent**(Base):

\_\_tablename\_\_ = 'parent'

id = Column(Integer, primary\_key=**True**)

child\_id = Column(Integer, ForeignKey('child.id'))

child = relationship("Child", back\_populates="parent")

**class** **Child**(Base):

\_\_tablename\_\_ = 'child'

id = Column(Integer, primary\_key=**True**)

parent = relationship("Parent", back\_populates="child", uselist=**False**)

As always, the [relationship.backref](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.backref" \o "sqlalchemy.orm.relationship) and [backref()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.backref" \o "sqlalchemy.orm.backref) functions may be used in lieu of the [relationship.back\_populates](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.back_populates" \o "sqlalchemy.orm.relationship) approach; to specify uselist on a backref, use the [backref()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.backref" \o "sqlalchemy.orm.backref) function:

一如以往，relationship.backref和backref()函数可以用来代替relationship.back\_populate方法; 要指定使用backref的用户名，请使用backref()函数：

**from** **sqlalchemy.orm** **import** backref

**class** **Parent**(Base):

\_\_tablename\_\_ = 'parent'

id = Column(Integer, primary\_key=**True**)

child\_id = Column(Integer, ForeignKey('child.id'))

child = relationship("Child", backref=backref("parent", uselist=**False**))

### 3.1.4 Many To Many

Many to Many adds an association table between two classes. The association table is indicated by the [secondary](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.secondary" \o "sqlalchemy.orm.relationship) argument to [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship). Usually, the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) uses the [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) object associated with the declarative base class, so that the [ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey) directives can locate the remote tables with which to link:

多对多添加两个类之间的关联表。 关联表由relationship()的[secondary](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.secondary" \o "sqlalchemy.orm.relationship)参数指示。 通常，表使用与声明基类相关联的MetaData对象，以便ForeignKey指令可以找到要与之链接的远程表：

association\_table = Table('association', Base.metadata,

Column('left\_id', Integer, ForeignKey('left.id')),

Column('right\_id', Integer, ForeignKey('right.id')))

**class** **Parent**(Base):

\_\_tablename\_\_ = 'left'

id = Column(Integer, primary\_key=**True**)

children = relationship("Child",

secondary=association\_table)

**class** **Child**(Base):

\_\_tablename\_\_ = 'right'

id = Column(Integer, primary\_key=**True**)

For a bidirectional relationship, both sides of the relationship contain a collection. Specify using [relationship.back\_populates](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.back_populates" \o "sqlalchemy.orm.relationship), and for each [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) specify the common association table:

对于双向关系，关系的双方都包含一个集合。 指定使用[relationship.back\_populates](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.back_populates" \o "sqlalchemy.orm.relationship)，并为每个relationship()指定公共关联表：

association\_table = Table('association', Base.metadata,

Column('left\_id', Integer, ForeignKey('left.id')),

Column('right\_id', Integer, ForeignKey('right.id')))

**class** **Parent**(Base):

\_\_tablename\_\_ = 'left'

id = Column(Integer, primary\_key=**True**)

children = relationship(

"Child",

secondary=association\_table,

back\_populates="parents")

**class** **Child**(Base):

\_\_tablename\_\_ = 'right'

id = Column(Integer, primary\_key=**True**)

parents = relationship(

"Parent",

secondary=association\_table,

back\_populates="children")

When using the [backref](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.backref" \o "sqlalchemy.orm.relationship) parameter instead of [relationship.back\_populates](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.back_populates" \o "sqlalchemy.orm.relationship), the backref will automatically use the same [secondary](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.secondary" \o "sqlalchemy.orm.relationship) argument for the reverse relationship:

association\_table = Table('association', Base.metadata,

Column('left\_id', Integer, ForeignKey('left.id')),

Column('right\_id', Integer, ForeignKey('right.id')))

**class** **Parent**(Base):

\_\_tablename\_\_ = 'left'

id = Column(Integer, primary\_key=**True**)

children = relationship("Child",

secondary=association\_table,

backref="parents")

**class** **Child**(Base):

\_\_tablename\_\_ = 'right'

id = Column(Integer, primary\_key=**True**)

The [secondary](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.secondary" \o "sqlalchemy.orm.relationship) argument of [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) also accepts a callable that returns the ultimate argument, which is evaluated only when mappers are first used. Using this, we can define the association\_table at a later point, as long as it's available to the callable after all module initialization is complete:

relationship()的[secondary](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.secondary" \o "sqlalchemy.orm.relationship)也接受一个可以返回最终参数的可调用值，该参数只有在首次使用映射器时进行评估。 使用这一点，我们可以在以后定义association\_table，只要所有模块初始化完成后，可调用可用：

**class** **Parent**(Base):

\_\_tablename\_\_ = 'left'

id = Column(Integer, primary\_key=**True**)

children = relationship("Child",

secondary=**lambda**: association\_table,

backref="parents")

With the declarative extension in use, the traditional "string name of the table" is accepted as well, matching the name of the table as stored in Base.metadata.tables:

**class** **Parent**(Base):

\_\_tablename\_\_ = 'left'

id = Column(Integer, primary\_key=**True**)

children = relationship("Child",

secondary="association",

backref="parents")

Deleting Rows from the Many to Many Table

A behavior which is unique to the [secondary](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.secondary" \o "sqlalchemy.orm.relationship) argument to [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) is that the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) which is specified here is automatically subject to INSERT and DELETE statements, as objects are added or removed from the collection. There is ****no need to delete from this table manually****. The act of removing a record from the collection will have the effect of the row being deleted on flush:

对于relationship()的[secondary](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.secondary" \o "sqlalchemy.orm.relationship)参数唯一的行为是在此处指定的表自动受INSERT和DELETE语句的限制，因为对象是从集合中添加或删除的。 无需手动从此表中删除。 从集合中删除记录的行为将会影响正在删除的行的冲洗：

*# row will be deleted from the "secondary" table# automatically*

myparent.children.remove(somechild)

A question which often arises is how the row in the "secondary" table can be deleted when the child object is handed directly to [Session.delete()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.delete" \o "sqlalchemy.orm.session.Session.delete):

经常出现的问题是当子对象直接递交给Session.delete()时，“次”表中的行如何被删除：

session.delete(somechild)

There are several possibilities here:

* If there is a [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) from Parent to Child, but there is ****not**** a reverse-relationship that links a particular Child to each Parent, SQLAlchemy will not have any awareness that when deleting this particular Child object, it needs to maintain the "secondary" table that links it to the Parent. No delete of the "secondary" table will occur.如果从Parent到Child有一个关系()，但是没有将特定的Child链接到每个Parent的反向关系，SQLAlchemy将不会有任何意识，当删除这个特定的Child对象时，它需要维护“secondary “表将其链接到父级。 不会删除“辅助”表。
* If there is a relationship that links a particular Child to each Parent, suppose it's called Child.parents, SQLAlchemy by default will load in the Child.parents collection to locate all Parent objects, and remove each row from the "secondary" table which establishes this link. Note that this relationship does not need to be bidrectional; SQLAlchemy is strictly looking at every [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) associated with the Child object being deleted.如果存在将特定子项链接到每个父项的关系，则假定它称为Child.parents，SQLAlchemy默认将加载到Child.parents集合中以查找所有父对象，并从建立的“secondary”表中删除每一行 这个链接。 请注意，这种关系不一定要加入; SQLAlchemy严格查看与要删除的Child对象关联的每个关系()。
* A higher performing option here is to use ON DELETE CASCADE directives with the foreign keys used by the database. Assuming the database supports this feature, the database itself can be made to automatically delete rows in the "secondary" table as referencing rows in "child" are deleted. SQLAlchemy can be instructed to forego actively loading in the Child.parents collection in this case using the [passive\_deletes](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.passive_deletes" \o "sqlalchemy.orm.relationship) directive on [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship); see [Using Passive Deletes](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "passive-deletes) for more details on this.更高性能的选项是使用ON DELETE CASCADE指令与数据库使用的外键。 假设数据库支持此功能，数据库本身可以自动删除“次”表中的行，因为“子”中的引用被删除。 在这种情况下，可以指示SQLAlchemy放弃在Child.parents集合中主动加载关于relationship()的passive\_deletes指令; 有关详细信息，请参阅使用被动删除。

Note again, these behaviors are *only* relevant to the [secondary](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.secondary" \o "sqlalchemy.orm.relationship) option used with [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship). If dealing with association tables that are mapped explicitly and are *not* present in the [secondary](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.secondary" \o "sqlalchemy.orm.relationship) option of a relevant [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship), cascade rules can be used instead to automatically delete entities in reaction to a related entity being deleted - see [Cascades](http://docs.sqlalchemy.org/en/rel_1_1/orm/cascades.html" \l "unitofwork-cascades) for information on this feature.

再次注意，这些行为仅与与relationship()一起使用的辅助选项相关。 如果处理明确映射并且不存在于相关关系()的辅助选项中的关联表，则可以使用级联规则来自动删除实体以对相关实体进行删除 - 有关此功能的信息，请参阅级联。

### 3.1.5 Association Object

The association object pattern is a variant on many-to-many: it's used when your association table contains additional columns beyond those which are foreign keys to the left and right tables. Instead of using the [secondary](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.secondary" \o "sqlalchemy.orm.relationship) argument, you map a new class directly to the association table. The left side of the relationship references the association object via one-to-many, and the association class references the right side via many-to-one. Below we illustrate an association table mapped to the Association class which includes a column called extra\_data, which is a string value that is stored along with each association between Parent and Child:

关联对象模式是多对多的变体：当关联表包含除了左侧和右侧表的外键之外的其他列时，将使用它。 而不是使用辅助参数，您可以将一个新类直接映射到关联表。 关系的左侧通过一对多引用关联对象，关联类通过多对一引用右侧。 下面我们说明映射到Association类的关联表，它包括一个名为extra\_data的列，它是一个字符串值，与Parent和Child之间的每个关联一起存储：

**class** **Association**(Base):

\_\_tablename\_\_ = 'association'

left\_id = Column(Integer, ForeignKey('left.id'), primary\_key=**True**)

right\_id = Column(Integer, ForeignKey('right.id'), primary\_key=**True**)

extra\_data = Column(String(50))

child = relationship("Child")

**class** **Parent**(Base):

\_\_tablename\_\_ = 'left'

id = Column(Integer, primary\_key=**True**)

children = relationship("Association")

**class** **Child**(Base):

\_\_tablename\_\_ = 'right'

id = Column(Integer, primary\_key=**True**)

As always, the bidirectional version make use of [relationship.back\_populates](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.back_populates" \o "sqlalchemy.orm.relationship) or [relationship.backref](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.backref" \o "sqlalchemy.orm.relationship):

**class** **Association**(Base):

\_\_tablename\_\_ = 'association'

left\_id = Column(Integer, ForeignKey('left.id'), primary\_key=**True**)

right\_id = Column(Integer, ForeignKey('right.id'), primary\_key=**True**)

extra\_data = Column(String(50))

child = relationship("Child", back\_populates="parents")

parent = relationship("Parent", back\_populates="children")

**class** **Parent**(Base):

\_\_tablename\_\_ = 'left'

id = Column(Integer, primary\_key=**True**)

children = relationship("Association", back\_populates="parent")

**class** **Child**(Base):

\_\_tablename\_\_ = 'right'

id = Column(Integer, primary\_key=**True**)

parents = relationship("Association", back\_populates="child")

Working with the association pattern in its direct form requires that child objects are associated with an association instance before being appended to the parent; similarly, access from parent to child goes through the association object:

以直接形式使用关联模式需要将子对象与关联实例关联，然后再附加到父对象; 类似地，从父对子访问通过关联对象：

*# create parent, append a child via association*p = Parent()a = Association(extra\_data="some data")a.child = Child()p.children.append(a)

*# iterate through child objects via association, including association# attributes***for** assoc **in** p.children:

print(assoc.extra\_data)

print(assoc.child)

To enhance the association object pattern such that direct access to the Association object is optional, SQLAlchemy provides the [Association Proxy](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html) extension. This extension allows the configuration of attributes which will access two "hops" with a single access, one "hop" to the associated object, and a second to a target attribute.

为了增强关联对象模式，以便直接访问关联对象是可选的，SQLAlchemy提供了关联代理扩展。 此扩展允许配置属性，这些属性将通过单个访问访问两个"跳"，将一个"跳"连接到关联的对象，另一个到目标属性。

**Warning/**警告

The association object pattern ****does not coordinate changes with a separate relationship that maps the association table as "secondary"****.

关联对象模式不会以将关联表映射为"次要"的单独关系来协调更改。

Below, changes made to Parent.children will not be coordinated with changes made to Parent.child\_associations or Child.parent\_associations in Python; while all of these relationships will continue to function normally by themselves, changes on one will not show up in another until the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) is expired, which normally occurs automatically after [Session.commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.commit" \o "sqlalchemy.orm.session.Session.commit):

以下，对Parent.children所做的更改不会与对Python中的Parent.child\_associations或Child.parent\_associations所做的更改进行协调。 虽然所有这些关系将自动继续正常运行，但是在会话过期之前，一般会发生更改，直到Session.commit()发生：

**class** **Association**(Base):

\_\_tablename\_\_ = 'association'

left\_id = Column(Integer, ForeignKey('left.id'), primary\_key=**True**)

right\_id = Column(Integer, ForeignKey('right.id'), primary\_key=**True**)

extra\_data = Column(String(50))

child = relationship("Child", backref="parent\_associations")

parent = relationship("Parent", backref="child\_associations")

**class** **Parent**(Base):

\_\_tablename\_\_ = 'left'

id = Column(Integer, primary\_key=**True**)

children = relationship("Child", secondary="association")

**class** **Child**(Base):

\_\_tablename\_\_ = 'right'

id = Column(Integer, primary\_key=**True**)

Additionally, just as changes to one relationship aren't reflected in the others automatically, writing the same data to both relationships will cause conflicting INSERT or DELETE statements as well, such as below where we establish the same relationship between a Parent and Child object twice:

另外，正如一个关系的更改不会自动反映在其他关系中的那样，将相同的数据写入两个关系也会导致冲突的INSERT或DELETE语句，如下面我们在父子对象之间建立相同的关系两次：

p1 = Parent()c1 = Child()p1.children.append(c1)

*# redundant, will cause a duplicate INSERT on Association*p1.parent\_associations.append(Association(child=c1))

It's fine to use a mapping like the above if you know what you're doing, though it may be a good idea to apply the viewonly=True parameter to the "secondary" relationship to avoid the issue of redundant changes being logged. However, to get a foolproof pattern that allows a simple two-object Parent->Child relationship while still using the association object pattern, use the association proxy extension as documented at [Association Proxy](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html).

如果您知道自己在做什么，可以使用上述映射，尽管将viewonly = True参数应用于"次要"关系可能是个好主意，以避免记录冗余更改的问题。 但是，为了获得一个简单的双对象Parent-> Child关系，同时仍然使用关联对象模式，可以使用关联代理扩展，如Association Proxy所示。

# **3.2 Adjacency List Relationships**

The ****adjacency list**** pattern is a common relational pattern whereby a table contains a foreign key reference to itself. This is the most common way to represent hierarchical data in flat tables. Other methods include ****nested sets****, sometimes called "modified preorder", as well as ****materialized path****. Despite the appeal that modified preorder has when evaluated for its fluency within SQL queries, the adjacency list model is probably the most appropriate pattern for the large majority of hierarchical storage needs, for reasons of concurrency, reduced complexity, and that modified preorder has little advantage over an application which can fully load subtrees into the application space.

邻接列表模式是公共关系模式，其中表格包含对其自身的外键引用。 这是在平面表中表示层次数据的最常见方式。 其他方法包括嵌套集合，有时称为"修改预订"，以及实现路径。 尽管在SQL查询中评估其流畅程度的情况下，修改后的预订有吸引力，但由于并发性的降低，复杂性降低，相邻列表模型可能是大多数分层存储需求中最合适的模式，而修改后的预订没有任何优势 通过可以将子树完全加载到应用程序空间中的应用程序。

In this example, we'll work with a single mapped class called Node, representing a tree structure:

**class** **Node**(Base):

\_\_tablename\_\_ = 'node'

id = Column(Integer, primary\_key=**True**)

parent\_id = Column(Integer, ForeignKey('node.id'))

data = Column(String(50))

children = relationship("Node")

With this structure, a graph such as the following:

root --+---> child1

+---> child2 --+--> subchild1

| +--> subchild2

+---> child3

Would be represented with data such as:

id parent\_id data--- ------- ----1 NULL root2 1 child13 1 child24 3 subchild15 3 subchild26 1 child3

The [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) configuration here works in the same way as a "normal" one-to-many relationship, with the exception that the "direction", i.e. whether the relationship is one-to-many or many-to-one, is assumed by default to be one-to-many. To establish the relationship as many-to-one, an extra directive is added known as [remote\_side](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.remote_side" \o "sqlalchemy.orm.relationship), which is a [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) or collection of [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects that indicate those which should be considered to be "remote":

这里的relationship()配置与“正常”一对多关系相同，除了“方向”，即关系是一对多还是多对一的关系外 默认假设为一对多。 要建立多对一的关系，一个额外的指令被添加为remote\_side，它是一列Column对象的集合，它们表示应该被认为是“remote”的列：

**class** **Node**(Base):

\_\_tablename\_\_ = 'node'

id = Column(Integer, primary\_key=**True**)

parent\_id = Column(Integer, ForeignKey('node.id'))

data = Column(String(50))

parent = relationship("Node", remote\_side=[id])

Where above, the id column is applied as the [remote\_side](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.remote_side" \o "sqlalchemy.orm.relationship) of the parent [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship), thus establishing parent\_id as the "local" side, and the relationship then behaves as a many-to-one.

As always, both directions can be combined into a bidirectional relationship using the [backref()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.backref" \o "sqlalchemy.orm.backref) function:

在上面的位置，id列被应用为父关系()的remote\_side，因此建立parent\_id作为“局部”一侧，然后该关系表现为多对一。

一如以往，两个方向都可以使用backref()函数组合成双向关系：

**class** **Node**(Base):

\_\_tablename\_\_ = 'node'

id = Column(Integer, primary\_key=**True**)

parent\_id = Column(Integer, ForeignKey('node.id'))

data = Column(String(50))

children = relationship("Node",

backref=backref('parent', remote\_side=[id])

)

There are several examples included with SQLAlchemy illustrating self-referential strategies; these include [Adjacency List](http://docs.sqlalchemy.org/en/rel_1_1/orm/examples.html" \l "examples-adjacencylist) and [XML Persistence](http://docs.sqlalchemy.org/en/rel_1_1/orm/examples.html" \l "examples-xmlpersistence).

SQLAlchemy中包含几个示例，说明自我参照策略; 这些包括邻接列表和XML持久性。

3.2.1 Composite Adjacency Lists

A sub-category of the adjacency list relationship is the rare case where a particular column is present on both the "local" and "remote" side of the join condition. An example is the Folder class below; using a composite primary key, the account\_id column refers to itself, to indicate sub folders which are within the same account as that of the parent; while folder\_id refers to a specific folder within that account:

邻接列表关系的子类别是在连接条件的“本地”和“远程”端都存在特定列的罕见情况。 一个例子是下面的Folder类; 使用复合主键，account\_id列自身指向与父级相同的帐户中的子文件夹; 而folder\_id是指该帐户中的特定文件夹：

**class** **Folder**(Base):

\_\_tablename\_\_ = 'folder'

\_\_table\_args\_\_ = (

ForeignKeyConstraint(

['account\_id', 'parent\_id'],

['folder.account\_id', 'folder.folder\_id']),

)

account\_id = Column(Integer, primary\_key=**True**)

folder\_id = Column(Integer, primary\_key=**True**)

parent\_id = Column(Integer)

name = Column(String)

parent\_folder = relationship("Folder",

backref="child\_folders",

remote\_side=[account\_id, folder\_id]

)

Above, we pass account\_id into the [remote\_side](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.remote_side" \o "sqlalchemy.orm.relationship) list. [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) recognizes that the account\_id column here is on both sides, and aligns the "remote" column along with the folder\_id column, which it recognizes as uniquely present on the "remote" side.

*New in version 0.8:*Support for self-referential composite keys in [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) where a column points to itself.

以上，我们将account\_id传递到remote\_side列表。 relationship()识别出这里的account\_id列在两边，并将“remote”列与folder\_id列对齐，它被识别为“远程”端唯一存在。

版本0.8中的新功能：支持关系()中的自引用复合键，其中列指向自身。

3.2.2 Self-Referential Query Strategies

Querying of self-referential structures works like any other query:

自我参照结构的查询与其他查询一样工作：

*# get all nodes named 'child2'*

session.query(Node).filter(Node.data=='child2')

However extra care is needed when attempting to join along the foreign key from one level of the tree to the next. In SQL, a join from a table to itself requires that at least one side of the expression be "aliased" so that it can be unambiguously referred to.

但是，当尝试沿着外部键从一个级别到另一级连接时，需要特别小心。 在SQL中，从表到本身的连接需要表达式的至少一边“别名”，以便明确地引用它。

Recall from [Using Aliases](http://docs.sqlalchemy.org/en/rel_1_1/orm/tutorial.html" \l "ormtutorial-aliases) in the ORM tutorial that the [orm.aliased()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.aliased" \o "sqlalchemy.orm.aliased) construct is normally used to provide an "alias" of an ORM entity. Joining from Node to itself using this technique looks like:

回想一下在ORM教程中使用别名，orm.aliased()结构通常用于提供ORM实体的“别名”。 使用这种技术从节点到自身的连接如下所示：

**from** **sqlalchemy.orm** **import** aliased

nodealias = aliased(Node)

session.query(Node).filter(Node.data=='subchild1').\

join(nodealias, Node.parent).\

filter(nodealias.data=="child2").\

all()

SELECT node.id AS node\_id,

node.parent\_id AS node\_parent\_id,

node.data AS node\_data

FROM node JOIN node AS node\_1

ON node.parent\_id = node\_1.id

WHERE node.data = ?

AND node\_1.data = ?

['subchild1', 'child2']

[Query.join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join) also includes a feature known as [Query.join.aliased](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join.params.aliased" \o "sqlalchemy.orm.query.Query.join) that can shorten the verbosity self- referential joins, at the expense of query flexibility. This feature performs a similar "aliasing" step to that above, without the need for an explicit entity. Calls to [Query.filter()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.filter" \o "sqlalchemy.orm.query.Query.filter) and similar subsequent to the aliased join will ****adapt**** the Node entity to be that of the alias:

Query.join()还包括一个称为Query.join.aliased的功能，可以缩短冗余自参照连接，牺牲了查询的灵活性。 此功能执行与上述类似的“混叠”步骤，而不需要显式实体。 对别名连接之后的Query.filter()和类似的调用将使Node实体变为别名的实体：

session.query(Node).filter(Node.data=='subchild1').\

join(Node.parent, aliased=True).\

filter(Node.data=='child2').\

all()

SELECT node.id AS node\_id,

node.parent\_id AS node\_parent\_id,

node.data AS node\_data

FROM node

JOIN node AS node\_1 ON node\_1.id = node.parent\_id

WHERE node.data = ? AND node\_1.data = ?

['subchild1', 'child2']

To add criterion to multiple points along a longer join, add [Query.join.from\_joinpoint](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join.params.from_joinpoint" \o "sqlalchemy.orm.query.Query.join) to the additional [join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join) calls:

要将标准添加到更长连接的多个点，请将Query.join.from\_joinpoint添加到附加的join()调用中：

*# get all nodes named 'subchild1' with a# parent named 'child2' and a grandparent 'root'*

session.query(Node).\

filter(Node.data=='subchild1').\

join(Node.parent, aliased=True).\

filter(Node.data=='child2').\

join(Node.parent, aliased=True, from\_joinpoint=True).\

filter(Node.data=='root').\

all()

[Query.reset\_joinpoint()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.reset_joinpoint" \o "sqlalchemy.orm.query.Query.reset_joinpoint) will also remove the "aliasing" from filtering calls:

Query.reset\_joinpoint()还将从过滤调用中删除“别名”：

session.query(Node).\

join(Node.children, aliased=**True**).\

filter(Node.data == 'foo').\

reset\_joinpoint().\

filter(Node.data == 'bar')

For an example of using [Query.join.aliased](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join.params.aliased" \o "sqlalchemy.orm.query.Query.join) to arbitrarily join along a chain of self-referential nodes, see [XML Persistence](http://docs.sqlalchemy.org/en/rel_1_1/orm/examples.html" \l "examples-xmlpersistence).

有关使用Query.join.aliased任意连接自参照节点链接的示例，请参阅XML持久性。

3.2.3 Configuring Self-Referential Eager Loading

Eager loading of relationships occurs using joins or outerjoins from parent to child table during a normal query operation, such that the parent and its immediate child collection or reference can be populated from a single SQL statement, or a second statement for all immediate child collections. SQLAlchemy's joined and subquery eager loading use aliased tables in all cases when joining to related items, so are compatible with self-referential joining. However, to use eager loading with a self-referential relationship, SQLAlchemy needs to be told how many levels deep it should join and/or query; otherwise the eager load will not take place at all. This depth setting is configured via join\_depth:

在正常的查询操作期间，使用从父对象到子表的联接或外联可以加载关系，从而可以从单个SQL语句或所有直接子集合的第二个语句填充父及其直接子集合或引用。 SQLAlchemy的加入和子查询加载使用别名表在所有情况下加入相关项目，所以与自引用加入兼容。 然而，要使用具有自我参照关系的热心加载，SQLAlchemy需要被告知应该加入和/或查询的级别很深; 否则根本不会发生急切的负载。 此深度设置通过join\_depth进行配置：

**class** **Node**(Base):

\_\_tablename\_\_ = 'node'

id = Column(Integer, primary\_key=True)

parent\_id = Column(Integer, ForeignKey('node.id'))

data = Column(String(50))

children = relationship("Node",

lazy="joined",

join\_depth=2)

session.query(Node).all()

SELECT node\_1.id AS node\_1\_id,

node\_1.parent\_id AS node\_1\_parent\_id,

node\_1.data AS node\_1\_data,

node\_2.id AS node\_2\_id,

node\_2.parent\_id AS node\_2\_parent\_id,

node\_2.data AS node\_2\_data,

node.id AS node\_id,

node.parent\_id AS node\_parent\_id,

node.data AS node\_data

FROM node

LEFT OUTER JOIN node AS node\_2

ON node.id = node\_2.parent\_id

LEFT OUTER JOIN node AS node\_1

ON node\_2.id = node\_1.parent\_id

[]

# 3.3 Linking Relationships with Backref

The [backref](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.backref" \o "sqlalchemy.orm.relationship) keyword argument was first introduced in [Object Relational Tutorial](http://docs.sqlalchemy.org/en/rel_1_1/orm/tutorial.html), and has been mentioned throughout many of the examples here. What does it actually do ? Let's start with the canonical User and Address scenario:

backref关键字参数首先在对象关系教程中引入，并且在这里的许多示例中都提到过。 它实际上是做什么的？ 我们从规范的用户和地址情况开始：

**from** **sqlalchemy** **import** Integer, ForeignKey, String, Column

**from** **sqlalchemy.ext.declarative** **import** declarative\_base

**from** **sqlalchemy.orm** **import** relationship

Base = declarative\_base()

**class** **User**(Base):

\_\_tablename\_\_ = 'user'

id = Column(Integer, primary\_key=**True**)

name = Column(String)

addresses = relationship("Address", backref="user")

**class** **Address**(Base):

\_\_tablename\_\_ = 'address'

id = Column(Integer, primary\_key=**True**)

email = Column(String)

user\_id = Column(Integer, ForeignKey('user.id'))

The above configuration establishes a collection of Address objects on User called User.addresses. It also establishes a .user attribute on Address which will refer to the parent User object.

上述配置在用户名为User.addresses的用户上建立了Address对象的集合。 它还在Address上建立一个.user属性，它将引用父User对象。

In fact, the [backref](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.backref" \o "sqlalchemy.orm.relationship) keyword is only a common shortcut for placing a second [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) onto the Address mapping, including the establishment of an event listener on both sides which will mirror attribute operations in both directions. The above configuration is equivalent to:

实际上，backref关键字只是在Address映射中放置第二个关系()的通用快捷方式，包括在两侧建立一个事件侦听器，将镜像两个方向的属性操作。 以上配置相当于：

**from** **sqlalchemy** **import** Integer, ForeignKey, String, Column

**from** **sqlalchemy.ext.declarative** **import** declarative\_base

**from** **sqlalchemy.orm** **import** relationship

Base = declarative\_base()

**class** **User**(Base):

\_\_tablename\_\_ = 'user'

id = Column(Integer, primary\_key=**True**)

name = Column(String)

addresses = relationship("Address", back\_populates="user")

**class** **Address**(Base):

\_\_tablename\_\_ = 'address'

id = Column(Integer, primary\_key=**True**)

email = Column(String)

user\_id = Column(Integer, ForeignKey('user.id'))

user = relationship("User", back\_populates="addresses")

Above, we add a .user relationship to Address explicitly. On both relationships, the [back\_populates](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.back_populates" \o "sqlalchemy.orm.relationship) directive tells each relationship about the other one, indicating that they should establish "bidirectional" behavior between each other. The primary effect of this configuration is that the relationship adds event handlers to both attributes which have the behavior of "when an append or set event occurs here, set ourselves onto the incoming attribute using this particular attribute name". The behavior is illustrated as follows. Start with a User and an Address instance. The .addresses collection is empty, and the .user attribute is None:

上面我们明确地添加一个.user关系到Address。 在这两种关系上，back\_populate指令都会告诉他们关于另外一种关系，指出它们应该在彼此之间建立“双向”行为。 此配置的主要作用是，该关系将事件处理程序添加到这两个属性，这些属性的行为是“当这个附加或设置事件发生在这里，使用这个特定的属性名称设置到传入属性”时。 行为如下所示。 从用户和地址实例开始。 .addresses集合为空，而.user属性为None：

**>>>** u1 = User()

**>>>** a1 = Address()

**>>>** u1.addresses[]

**>>>** print(a1.user)None

However, once the Address is appended to the u1.addresses collection, both the collection and the scalar attribute have been populated:

但是，一旦将地址附加到u1.addresses集合，就会填充集合和标量属性：

**>>>** u1.addresses.append(a1)

**>>>** u1.addresses[<\_\_main\_\_.Address object at 0x12a6ed0>]

**>>>** a1.user<\_\_main\_\_.User object at 0x12a6590>

This behavior of course works in reverse for removal operations as well, as well as for equivalent operations on both sides. Such as when .user is set again to None, the Address object is removed from the reverse collection:

这种行为当然也适用于删除操作以及双方的等效操作。 例如当.user再次设置为None时，Address对象将从反向集合中删除：

**>>>** a1.user = **None**

**>>>** u1.addresses[]

The manipulation of the .addresses collection and the .user attribute occurs entirely in Python without any interaction with the SQL database. Without this behavior, the proper state would be apparent on both sides once the data has been flushed to the database, and later reloaded after a commit or expiration operation occurs. The [backref](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.backref" \o "sqlalchemy.orm.relationship)/[back\_populates](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.back_populates" \o "sqlalchemy.orm.relationship) behavior has the advantage that common bidirectional operations can reflect the correct state without requiring a database round trip.

.addresses集合和.user属性的操作完全在Python中，而不与SQL数据库进行任何交互。 没有这种行为，一旦数据被刷新到数据库，并且稍后在提交或到期操作发生之后被重新加载，则两端的适当状态将是显而易见的。 backref / back\_populate行为的优点是常见的双向操作可以反映正确的状态，而不需要数据库往返。

Remember, when the [backref](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.backref" \o "sqlalchemy.orm.relationship) keyword is used on a single relationship, it's exactly the same as if the above two relationships were created individually using [back\_populates](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.back_populates" \o "sqlalchemy.orm.relationship) on each.

请记住，当在一个关系中使用backref关键字时，与上述两个关系是单独使用back\_populate分别创建的完全相同。

## Backref Arguments

We've established that the [backref](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.backref" \o "sqlalchemy.orm.relationship) keyword is merely a shortcut for building two individual [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) constructs that refer to each other. Part of the behavior of this shortcut is that certain configurational arguments applied to the [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) will also be applied to the other direction - namely those arguments that describe the relationship at a schema level, and are unlikely to be different in the reverse direction. The usual case here is a many-to-many [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) that has a [secondary](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.secondary" \o "sqlalchemy.orm.relationship) argument, or a one-to-many or many-to-one which has a [primaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.primaryjoin" \o "sqlalchemy.orm.relationship) argument (the [primaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.primaryjoin" \o "sqlalchemy.orm.relationship) argument is discussed in [Specifying Alternate Join Conditions](http://docs.sqlalchemy.org/en/rel_1_1/orm/join_conditions.html" \l "relationship-primaryjoin)). Such as if we limited the list of Address objects to those which start with "tony":

我们已经确定，backref关键字只是构建彼此引用的两个单独的关系()结构的捷径。 这个快捷方式的一部分行为是应用于relationship()的某些配置参数也将应用于另一个方向 - 即在模式级别描述关系的那些参数，并且不太可能在相反的方向上有所不同。 这里通常的情况是具有二次参数的多对多关系()，或具有primaryjoin参数的一对多或多对一(在指定备用连接条件中讨论了primaryjoin参数)。 如果我们将Address对象的列表限制为以“tony”开头的列表：

**from** **sqlalchemy** **import** Integer, ForeignKey, String, Column

**from** **sqlalchemy.ext.declarative** **import** declarative\_base

**from** **sqlalchemy.orm** **import** relationship

Base = declarative\_base()

**class** **User**(Base):

\_\_tablename\_\_ = 'user'

id = Column(Integer, primary\_key=**True**)

name = Column(String)

addresses = relationship("Address",

primaryjoin="and\_(User.id==Address.user\_id, "

"Address.email.startswith('tony'))",

backref="user")

**class** **Address**(Base):

\_\_tablename\_\_ = 'address'

id = Column(Integer, primary\_key=**True**)

email = Column(String)

user\_id = Column(Integer, ForeignKey('user.id'))

We can observe, by inspecting the resulting property, that both sides of the relationship have this join condition applied:

我们可以通过检查所产生的属性来观察，双方的关系是否适用了这种加入条件：

**>>>** print(User.addresses.property.primaryjoin)

"user".id = address.user\_id AND address.email LIKE :email\_1 || '%%'>>>

**>>>** print(Address.user.property.primaryjoin)

"user".id = address.user\_id AND address.email LIKE :email\_1 || '%%'>>>

This reuse of arguments should pretty much do the "right thing" - it uses only arguments that are applicable, and in the case of a many-to- many relationship, will reverse the usage of [primaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.primaryjoin" \o "sqlalchemy.orm.relationship) and [secondaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.secondaryjoin" \o "sqlalchemy.orm.relationship) to correspond to the other direction (see the example in [Self-Referential Many-to-Many Relationship](http://docs.sqlalchemy.org/en/rel_1_1/orm/join_conditions.html" \l "self-referential-many-to-many) for this).

这种对参数的重用应该是“正确的事情” - 它只使用适用的参数，而在多对多的关系的情况下，将反向使用[primaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.primaryjoin" \o "sqlalchemy.orm.relationship) 和[secondaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.secondaryjoin" \o "sqlalchemy.orm.relationship) 来对应另一个方向 参见自我参照多对多关系的例子)。

It's very often the case however that we'd like to specify arguments that are specific to just the side where we happened to place the "backref". This includes [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) arguments like [lazy](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.lazy" \o "sqlalchemy.orm.relationship), [remote\_side](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.remote_side" \o "sqlalchemy.orm.relationship), [cascade](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.cascade" \o "sqlalchemy.orm.relationship) and [cascade\_backrefs](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.cascade_backrefs" \o "sqlalchemy.orm.relationship). For this case we use the [backref()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.backref" \o "sqlalchemy.orm.backref) function in place of a string:

然而，通常情况下，我们想指定一些特定于仅仅发生在“backref”位置的参数。 这包括[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)参数，如[lazy](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.lazy" \o "sqlalchemy.orm.relationship)，[remote\_side](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.remote_side" \o "sqlalchemy.orm.relationship)，[cascade](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.cascade" \o "sqlalchemy.orm.relationship)和[cascade\_backrefs](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.cascade_backrefs" \o "sqlalchemy.orm.relationship)。 对于这种情况，我们使用[backref()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.backref" \o "sqlalchemy.orm.backref)函数代替字符串：

*# <other imports>***from** **sqlalchemy.orm** **import** backref

**class** **User**(Base):

\_\_tablename\_\_ = 'user'

id = Column(Integer, primary\_key=**True**)

name = Column(String)

addresses = relationship("Address",

backref=backref("user", lazy="joined"))

Where above, we placed a lazy="joined" directive only on the Address.user side, indicating that when a query against Address is made, a join to the User entity should be made automatically which will populate the .user attribute of each returned Address. The [backref()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.backref" \o "sqlalchemy.orm.backref) function formatted the arguments we gave it into a form that is interpreted by the receiving [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) as additional arguments to be applied to the new relationship it creates.

在上面的地方，我们只在Address.user方面放置了一个lazy="joined"指令，表示当对Address进行查询时，应该自动加入到User 实体，这将填充每个返回的Address的.user属性。 [backref()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.backref" \o "sqlalchemy.orm.backref)函数将我们给它的参数格式化成一个由接收[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)解释的表单作为要应用于它创建的新关系的附加参数。

## One Way Backrefs

An unusual case is that of the "one way backref". This is where the "back-populating" behavior of the backref is only desirable in one direction. An example of this is a collection which contains a filtering [primaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.primaryjoin" \o "sqlalchemy.orm.relationship) condition. We'd like to append items to this collection as needed, and have them populate the "parent" object on the incoming object. However, we'd also like to have items that are not part of the collection, but still have the same "parent" association - these items should never be in the collection.

一个不寻常的情况是“单向backref”。 这就是backref的“反向填充”行为只是在一个方向是可取的。 一个例子是一个包含过滤小连接条件的集合。 我们希望根据需要将项目附加到此集合中，并使其填充传入对象上的“父”对象。 然而，我们也希望有一些不属于集合的项目，但是仍然有相同的“父”关联 - 这些项目不应该在集合中。

Taking our previous example, where we established a [primaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.primaryjoin" \o "sqlalchemy.orm.relationship) that limited the collection only to Address objects whose email address started with the wordtony, the usual backref behavior is that all items populate in both directions. We wouldn't want this behavior for a case like the following:

以我们前面的例子，我们建立了一个仅将收藏仅限于将电子邮件地址以wordtony开头的对象限制在一起的Primaryjoin，通常的backref行为就是所有项目都在两个方向上填充。 我们不希望这种情况如下：

**>>>** u1 = User()

**>>>** a1 = Address(email='mary')

**>>>** a1.user = u1

**>>>** u1.addresses

[<\_\_main\_\_.Address object at 0x1411910>]

Above, the Address object that doesn't match the criterion of "starts with 'tony'" is present in the addresses collection of u1. After these objects are flushed, the transaction committed and their attributes expired for a re-load, the addresses collection will hit the database on next access and no longer have this Addressobject present, due to the filtering condition. But we can do away with this unwanted side of the "backref" behavior on the Python side by using two separate [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) constructs, placing [back\_populates](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.back_populates" \o "sqlalchemy.orm.relationship) only on one side:

以上，与“tony”开头的标准不符的Address对象存在于u1的地址集合中。 在刷新这些对象之后，事务提交及其属性已过期，以重新加载，由于过滤条件，地址集合将在下次访问时遇到数据库，并且不再具有此Address对象。 但是，我们可以通过使用两个单独的关系()构造来消除Python侧面的“backref”行为的不必要的一面，仅在一侧放置back\_populate：

**from** **sqlalchemy** **import** Integer, ForeignKey, String, Column

**from** **sqlalchemy.ext.declarative** **import** declarative\_base

**from** **sqlalchemy.orm** **import** relationship

Base = declarative\_base()

**class** **User**(Base):

\_\_tablename\_\_ = 'user'

id = Column(Integer, primary\_key=**True**)

name = Column(String)

addresses = relationship("Address",

primaryjoin="and\_(User.id==Address.user\_id, "

"Address.email.startswith('tony'))",

back\_populates="user")

**class** **Address**(Base):

\_\_tablename\_\_ = 'address'

id = Column(Integer, primary\_key=**True**)

email = Column(String)

user\_id = Column(Integer, ForeignKey('user.id'))

user = relationship("User")

With the above scenario, appending an Address object to the .addresses collection of a User will always establish the .user attribute on that Address:

使用上述情况，将Address对象附加到用户的.addresses集合将始终在该地址上建立.user属性：

**>>>** u1 = User()

**>>>** a1 = Address(email='tony')

**>>>** u1.addresses.append(a1)

**>>>** a1.user<\_\_main\_\_.User object at 0x1411850>

However, applying a User to the .user attribute of an Address, will not append the Address object to the collection:

但是，将用户应用到Address的.user属性，将不会将Address对象附加到集合中：

**>>>** a2 = Address(email='mary')

**>>>** a2.user = u1

**>>>** a2 **in** u1.addressesFalse

Of course, we've disabled some of the usefulness of [backref](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.backref" \o "sqlalchemy.orm.relationship) here, in that when we do append an Address that corresponds to the criteria ofemail.startswith('tony'), it won't show up in the User.addresses collection until the session is flushed, and the attributes reloaded after a commit or expire operation. While we could consider an attribute event that checks this criterion in Python, this starts to cross the line of duplicating too much SQL behavior in Python. The backref behavior itself is only a slight transgression of this philosophy - SQLAlchemy tries to keep these to a minimum overall.

当然，我们已经禁用了这里的一些backref的用法，因为当我们附加一个与电子邮件标签(“tony”)相对应的Address时，它不会显示在User.addresses集合中，直到 刷新会话，并在提交或过期操作后重新加载属性。 虽然我们可以考虑在Python中检查此标准的属性事件，但是这开始跨越Python中复制太多SQL行为的行。 backref行为本身只是这种哲学的轻微违规 - SQLAlchemy试图将它们保持在最低水平。

# **3.4 Configuring how Relationship Joins**

[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) will normally create a join between two tables by examining the foreign key relationship between the two tables to determine which columns should be compared. There are a variety of situations where this behavior needs to be customized.

relationship()通常会通过检查两个表之间的外键关系来创建两个表之间的连接，以确定应比较哪些列。 存在这种行为需要定制的各种情况。

3.4.1 Handling Multiple Join Paths

One of the most common situations to deal with is when there are more than one foreign key path between two tables.

Consider a Customer class that contains two foreign keys to an Address class:

**from** **sqlalchemy** **import** Integer, ForeignKey, String, Column

**from** **sqlalchemy.ext.declarative** **import** declarative\_base

**from** **sqlalchemy.orm** **import** relationship

Base = declarative\_base()

**class** **Customer**(Base):

\_\_tablename\_\_ = 'customer'

id = Column(Integer, primary\_key=**True**)

name = Column(String)

billing\_address\_id = Column(Integer, ForeignKey("address.id"))

shipping\_address\_id = Column(Integer, ForeignKey("address.id"))

billing\_address = relationship("Address")

shipping\_address = relationship("Address")

**class** **Address**(Base):

\_\_tablename\_\_ = 'address'

id = Column(Integer, primary\_key=**True**)

street = Column(String)

city = Column(String)

state = Column(String)

zip = Column(String)

The above mapping, when we attempt to use it, will produce the error:

sqlalchemy.exc.AmbiguousForeignKeysError: Could **not** determine joincondition between parent/child tables on relationshipCustomer.billing\_address - there are multiple foreign keypaths linking the tables. Specify the 'foreign\_keys' argument,providing a list of those columns which should becounted **as** containing a foreign key reference to the parent table.

The above message is pretty long. There are many potential messages that [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) can return, which have been carefully tailored to detect a variety of common configurational issues; most will suggest the additional configuration that's needed to resolve the ambiguity or other missing information.

上面的消息很长。 [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)可以返回的许多潜在消息，这些消息已经被精心地定制以检测各种常见的配置问题; 大多数将建议解决歧义或其他缺失信息所需的其他配置。

In this case, the message wants us to qualify each [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) by instructing for each one which foreign key column should be considered, and the appropriate form is as follows:

在这种情况下，消息要求我们通过指示每个关系列表中的每个[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)进行限定，并且适当的格式如下：

**class** **Customer**(Base):

\_\_tablename\_\_ = 'customer'

id = Column(Integer, primary\_key=**True**)

name = Column(String)

billing\_address\_id = Column(Integer, ForeignKey("address.id"))

shipping\_address\_id = Column(Integer, ForeignKey("address.id"))

billing\_address = relationship("Address", foreign\_keys=[billing\_address\_id])

shipping\_address = relationship("Address", foreign\_keys=[shipping\_address\_id])

Above, we specify the foreign\_keys argument, which is a [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) or list of [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects which indicate those columns to be considered "foreign", or in other words, the columns that contain a value referring to a parent table. Loading the Customer.billing\_address relationship from a Customer object will use the value present in billing\_address\_id in order to identify the row in Address to be loaded; similarly, shipping\_address\_id is used for the shipping\_address relationship. The linkage of the two columns also plays a role during persistence; the newly generated primary key of a just-inserted Addressobject will be copied into the appropriate foreign key column of an associated Customer object during a flush.

在上面，我们指定了foreign\_keys参数，它是Column对象的列或列表，它们表示这些列被视为“foreign”，也就是说，包含引用父表的值的列。 从Customer对象加载Customer.billing\_address关系将使用billing\_address\_id中存在的值来识别要加载的地址中的行; 类似地，shipping\_address\_id用于shipping\_address关系。 两列的联系在持久性中也起着重要的作用; 刚刚插入的Addressobject的新生成的主键将被刷新到相关联的Customer对象的相应外键列。

When specifying foreign\_keys with Declarative, we can also use string names to specify, however it is important that if using a list, the ****list is part of the string****:

当使用Declarative指定foreign\_key时，我们还可以使用字符串名称来指定，但是，如果使用列表，列表是字符串的一部分是很重要的：

billing\_address = relationship("Address", foreign\_keys="[Customer.billing\_address\_id]")

In this specific example, the list is not necessary in any case as there's only one [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) we need:

在这个具体的例子中，列表在任何情况下都不是必需的，因为我们只需要一[Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column)：

billing\_address = relationship("Address", foreign\_keys="Customer.billing\_address\_id")

*Changed in version 0.8:*[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) can resolve ambiguity between foreign key targets on the basis of the foreign\_keys argument alone; the [primaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.primaryjoin" \o "sqlalchemy.orm.relationship) argument is no longer needed in this situation.

在版本0.8中更改：[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)可以根据foreign\_keys参数单独解决外键目标之间的歧义; 在这种情况下，不再需要小提法。

Specifying Alternate Join Conditions

The default behavior of [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) when constructing a join is that it equates the value of primary key columns on one side to that of foreign-key-referring columns on the other. We can change this criterion to be anything we'd like using the [primaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.primaryjoin" \o "sqlalchemy.orm.relationship) argument, as well as the [secondaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.secondaryjoin" \o "sqlalchemy.orm.relationship) argument in the case when a "secondary" table is used.

在构建连接时，关系()的默认行为是将主键列的值与外键引用列的值相等。 我们可以将此标准更改为使用primaryjoin参数的任何内容，以及在使用“secondary”表时的secondaryjoin参数。

In the example below, using the User class as well as an Address class which stores a street address, we create a relationship boston\_addresses which will only load those Address objects which specify a city of "Boston":

在下面的示例中，使用User类以及存储街道地址的Address类，我们创建一个关系boston\_addresses，它将只加载指定城市“Boston”的Address对象：

**from** **sqlalchemy** **import** Integer, ForeignKey, String, Column

**from** **sqlalchemy.ext.declarative** **import** declarative\_base

**from** **sqlalchemy.orm** **import** relationship

Base = declarative\_base()

**class** **User**(Base):

\_\_tablename\_\_ = 'user'

id = Column(Integer, primary\_key=**True**)

name = Column(String)

boston\_addresses = relationship("Address",

primaryjoin="and\_(User.id==Address.user\_id, "

"Address.city=='Boston')")

**class** **Address**(Base):

\_\_tablename\_\_ = 'address'

id = Column(Integer, primary\_key=**True**)

user\_id = Column(Integer, ForeignKey('user.id'))

street = Column(String)

city = Column(String)

state = Column(String)

zip = Column(String)

Within this string SQL expression, we made use of the [and\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.and_" \o "sqlalchemy.sql.expression.and_) conjunction construct to establish two distinct predicates for the join condition - joining both the User.id and Address.user\_id columns to each other, as well as limiting rows in Address to just city='Boston'. When using Declarative, rudimentary SQL functions like [and\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.and_" \o "sqlalchemy.sql.expression.and_) are automatically available in the evaluated namespace of a string [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) argument.

在这个字符串SQL表达式中，我们使用[and\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.and_" \o "sqlalchemy.sql.expression.and_)连接结构来建立连接条件的两个不同的谓词 - 将User.id和Address.user\_id列相互加入，并将Address中的行限制为只是city='Boston'。当使用Declarative时，基本的SQL函数如[and\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.and_" \o "sqlalchemy.sql.expression.and_)在字符串[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)参数的评估命名空间中可以自动使用。

The custom criteria we use in a [primaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.primaryjoin" \o "sqlalchemy.orm.relationship) is generally only significant when SQLAlchemy is rendering SQL in order to load or represent this relationship. That is, it's used in the SQL statement that's emitted in order to perform a per-attribute lazy load, or when a join is constructed at query time, such as via [Query.join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join), or via the eager "joined" or "subquery" styles of loading. When in-memory objects are being manipulated, we can place any Address object we'd like into the boston\_addresses collection, regardless of what the value of the .city attribute is. The objects will remain present in the collection until the attribute is expired and re-loaded from the database where the criterion is applied. When a flush occurs, the objects inside of boston\_addresses will be flushed unconditionally, assigning value of the primary key user.id column onto the foreign-key-holding address.user\_id column for each row. The city criteria has no effect here, as the flush process only cares about synchronizing primary key values into referencing foreign key values.

我们在[primaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.primaryjoin" \o "sqlalchemy.orm.relationship)中使用的自定义标准通常仅在SQLAlchemy呈现SQL才能加载或表示此关系时才有意义。也就是说，它用于发出的SQL语句，以执行每个属性的懒加载，或者在查询时构建连接，例如通过[Query.join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join)或通过“加入”或“子查询“样式的加载。当内存中的对象被操纵时，我们可以将任何我们想要的Address对象放在boston\_addresses集合中，而不管.city属性的值如何。对象将保留在集合中，直到该属性过期并从应用该标准的数据库重新加载。发生冲突时，boston\_addresses内的对象将被无条件刷新，将主键user.id列的值分配给每行的外键保存地址user.id列。city标准在这里没有任何效果，因为刷新过程只关心将主键值同步到引用外键值。

Creating Custom Foreign Conditions

Another element of the primary join condition is how those columns considered "foreign" are determined. Usually, some subset of [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects will specify [ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey), or otherwise be part of a [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint) that's relevant to the join condition. [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) looks to this foreign key status as it decides how it should load and persist data for this relationship. However, the [primaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.primaryjoin" \o "sqlalchemy.orm.relationship) argument can be used to create a join condition that doesn't involve any "schema" level foreign keys. We can combine [primaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.primaryjoin" \o "sqlalchemy.orm.relationship) along with [foreign\_keys](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.foreign_keys" \o "sqlalchemy.orm.relationship) and [remote\_side](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.remote_side" \o "sqlalchemy.orm.relationship) explicitly in order to establish such a join.

主要连接条件的另一个元素是如何确定这些列被视为“外部”。 通常，[Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column)对象的一些子集将指定[ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey)，否则为与连接条件相关的[ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint)的一部分。 [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)会查看此外键状态，因为它决定如何加载和保留此关系的数据。 但是，[primaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.primaryjoin" \o "sqlalchemy.orm.relationship)参数可用于创建不涉及任何“模式”级外键的连接条件。 我们可以将[primaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.primaryjoin" \o "sqlalchemy.orm.relationship)和[foreign\_keys](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.foreign_keys" \o "sqlalchemy.orm.relationship)和remote\_side明确地结合起来，以建立这样的连接。

Below, a class HostEntry joins to itself, equating the string content column to the ip\_address column, which is a PostgreSQL type called INET. We need to use [cast()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.cast" \o "sqlalchemy.sql.expression.cast) in order to cast one side of the join to the type of the other:

下面，一个HostEntry类连接到本身，将字符串内容列等同于content列，这是一个名为INET的PostgreSQL类型。 我们需要使用[cast()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.cast" \o "sqlalchemy.sql.expression.cast)，以便将连接的一方投射到另一方的类型中：

**from** **sqlalchemy** **import** cast, String, Column, Integer

**from** **sqlalchemy.orm** **import** relationship

**from** **sqlalchemy.dialects.postgresql** **import** INET

**from** **sqlalchemy.ext.declarative** **import** declarative\_base

Base = declarative\_base()

**class** **HostEntry**(Base):

\_\_tablename\_\_ = 'host\_entry'

id = Column(Integer, primary\_key=**True**)

ip\_address = Column(INET)

content = Column(String(50))

*# relationship() using explicit foreign\_keys, remote\_side*

parent\_host = relationship("HostEntry",

primaryjoin=ip\_address == cast(content, INET),

foreign\_keys=content,

remote\_side=ip\_address

)

The above relationship will produce a join like:

SELECT host\_entry.id, host\_entry.ip\_address, host\_entry.content

FROM host\_entry

JOIN host\_entry AS host\_entry\_1

ON host\_entry\_1.ip\_address = CAST(host\_entry.content AS INET)

An alternative syntax to the above is to use the [foreign()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.foreign" \o "sqlalchemy.orm.foreign) and [remote()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.remote" \o "sqlalchemy.orm.remote) [annotations](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-annotations), inline within the [primaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.primaryjoin" \o "sqlalchemy.orm.relationship) expression. This syntax represents the annotations that [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) normally applies by itself to the join condition given the [foreign\_keys](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.foreign_keys" \o "sqlalchemy.orm.relationship) and [remote\_side](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.remote_side" \o "sqlalchemy.orm.relationship) arguments. These functions may be more succinct when an explicit join condition is present, and additionally serve to mark exactly the column that is "foreign" or "remote" independent of whether that column is stated multiple times or within complex SQL expressions:

上述的替代语法是使用[foreign()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.foreign" \o "sqlalchemy.orm.foreign)和[remote()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.remote" \o "sqlalchemy.orm.remote)注释，内联在[primaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.primaryjoin" \o "sqlalchemy.orm.relationship)表达式中。 此语法表示[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)通常适用于给定[foreign\_keys](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.foreign_keys" \o "sqlalchemy.orm.relationship)和[remote\_side](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.remote_side" \o "sqlalchemy.orm.relationship)参数的连接条件的注释。 当显式连接条件存在时，这些功能可能更加简洁，另外还可以完全标记“外部”或“远程”列，而不管该列是多次表达还是在多个SQL表达式中进行说明：

**from** **sqlalchemy.orm** **import** foreign, remote

**class** **HostEntry**(Base):

\_\_tablename\_\_ = 'host\_entry'

id = Column(Integer, primary\_key=**True**)

ip\_address = Column(INET)

content = Column(String(50))

*# relationship() using explicit foreign() and remote() annotations*

*# in lieu of separate arguments*

parent\_host = relationship("HostEntry",

primaryjoin=remote(ip\_address) == \

cast(foreign(content), INET),

)

## Using custom operators in join conditions

Another use case for relationships is the use of custom operators, such as PostgreSQL's "is contained within" << operator when joining with types such as [postgresql.INET](http://docs.sqlalchemy.org/en/rel_1_1/dialects/postgresql.html" \l "sqlalchemy.dialects.postgresql.INET" \o "sqlalchemy.dialects.postgresql.INET) and [postgresql.CIDR](http://docs.sqlalchemy.org/en/rel_1_1/dialects/postgresql.html" \l "sqlalchemy.dialects.postgresql.CIDR" \o "sqlalchemy.dialects.postgresql.CIDR). For custom operators we use the [Operators.op()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.Operators.op" \o "sqlalchemy.sql.operators.Operators.op) function:

关系的另一个用例是使用自定义运算符，例如PostgreSQL的“包含在”<<运算符中时加入类型如postgresql.INET和postgresql.CIDR。 对于自定义运算符，我们使用Operators.op()函数：

inet\_column.op("<<")(cidr\_column)

However, if we construct a [primaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.primaryjoin" \o "sqlalchemy.orm.relationship) using this operator, [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) will still need more information. This is because when it examines our primaryjoin condition, it specifically looks for operators used for ****comparisons****, and this is typically a fixed list containing known comparison operators such as ==, <, etc. So for our custom operator to participate in this system, we need it to register as a comparison operator using the [is\_comparison](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.Operators.op.params.is_comparison" \o "sqlalchemy.sql.operators.Operators.op) parameter:

然而，如果我们使用这个运算符构造一个小连接，则relationship()仍然需要更多的信息。 这是因为当它检查我们的初级条件时，它特别寻找用于比较的运算符，这通常是包含已知的比较运算符(例如==，<等等)的固定列表。所以对于我们的定制运算符来参与这个系统， 我们需要它使用is\_comparison参数注册为比较运算符：

inet\_column.op("<<", is\_comparison=**True**)(cidr\_column)

A complete example:

**class** **IPA**(Base):

\_\_tablename\_\_ = 'ip\_address'

id = Column(Integer, primary\_key=**True**)

v4address = Column(INET)

network = relationship("Network",

primaryjoin="IPA.v4address.op('<<', is\_comparison=True)"

"(foreign(Network.v4representation))",

viewonly=**True**

)**class** **Network**(Base):

\_\_tablename\_\_ = 'network'

id = Column(Integer, primary\_key=**True**)

v4representation = Column(CIDR)

Above, a query such as:

session.query(IPA).join(IPA.network)

Will render as:

SELECT ip\_address.id AS ip\_address\_id, ip\_address.v4address AS ip\_address\_v4addressFROM ip\_address JOIN network ON ip\_address.v4address << network.v4representation

*New in version 0.9.2:*- Added the [Operators.op.is\_comparison](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.Operators.op.params.is_comparison" \o "sqlalchemy.sql.operators.Operators.op) flag to assist in the creation of [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) constructs using custom operators.

Overlapping Foreign Keys

A rare scenario can arise when composite foreign keys are used, such that a single column may be the subject of more than one column referred to via foreign key constraint.

Consider an (admittedly complex) mapping such as the Magazine object, referred to both by the Writer object and the Article object using a composite primary key scheme that includes magazine\_id for both; then to make Article refer to Writer as well, Article.magazine\_id is involved in two separate relationships; Article.magazine and Article.writer:

当使用复合外键时，可能会出现一个罕见的情况，使得单列可能是通过外键约束引用的多个列的主题。

考虑由Writer对象和Article对象使用包含magazine\_id的复合主键方案引用的(对应的复杂)映射(例如Magazine对象)。 那么文章也指作者，Article.magazine\_id涉及两个不同的关系;Article.magazine和Article.writer：

**class** **Magazine**(Base):

\_\_tablename\_\_ = 'magazine'

id = Column(Integer, primary\_key=**True**)

**class** **Article**(Base):

\_\_tablename\_\_ = 'article'

article\_id = Column(Integer)

magazine\_id = Column(ForeignKey('magazine.id'))

writer\_id = Column()

magazine = relationship("Magazine")

writer = relationship("Writer")

\_\_table\_args\_\_ = (

PrimaryKeyConstraint('article\_id', 'magazine\_id'),

ForeignKeyConstraint(

['writer\_id', 'magazine\_id'],

['writer.id', 'writer.magazine\_id']

),

)

**class** **Writer**(Base):

\_\_tablename\_\_ = 'writer'

id = Column(Integer, primary\_key=**True**)

magazine\_id = Column(ForeignKey('magazine.id'), primary\_key=**True**)

magazine = relationship("Magazine")

When the above mapping is configured, we will see this warning emitted:

SAWarning: relationship 'Article.writer' will copy columnwriter.magazine\_id to column article.magazine\_id,which conflicts **with** relationship(s): 'Article.magazine'(copies magazine.id to article.magazine\_id). Consider applyingviewonly=**True** to read-only relationships, **or** provide a primaryjoincondition marking writable columns **with** the foreign() annotation.

What this refers to originates from the fact that Article.magazine\_id is the subject of two different foreign key constraints; it refers to Magazine.id directly as a source column, but also refers to Writer.magazine\_id as a source column in the context of the composite key to Writer. If we associate an Article with a particular Magazine, but then associate the Article with a Writer that's associated with a *different* Magazine, the ORM will overwrite Article.magazine\_id non-deterministically, silently changing which magazine we refer towards; it may also attempt to place NULL into this columnn if we de-associate a Writer from an Article. The warning lets us know this is the case.

这是因为Article.magazine\_id是两个不同的外键约束的主题; 它将Magazine.id直接引用为一个源列，而且还将Writer.magazine\_id作为复合键上下文中的一个源列写入Writer。 如果我们将文章与特定杂志联系起来，然后将文章与与不同杂志相关联的作者相关联，则ORM将非确定性地覆盖Article.magazine\_id，默默地更改我们所指的哪本杂志; 如果我们从文章中删除一个Writer，它也可能尝试将NULL置于此列中。 这个警告让我们知道是这样的。

To solve this, we need to break out the behavior of Article to include all three of the following features:

要解决这个问题，我们需要打破Article的行为，包括以下所有三个特征：

1. Article first and foremost writes to Article.magazine\_id based on data persisted in the Article.magazine relationship only, that is a value copied from Magazine.id.首先，首先根据仅存在于Article.magazine关系中的数据写入Article.magazine\_id，这是从Magazine.id复制的值。
2. Article can write to Article.writer\_id on behalf of data persisted in the Article.writer relationship, but only the Writer.id column; the Writer.magazine\_id column should not be written into Article.magazine\_id as it ultimately is sourced from Magazine.id.文章可以写Article.writer\_id代表持续存在于Article.writer关系中的数据，但只能写入Writer.id列; Writer.magazine\_id列不应该写入Article.magazine\_id，因为它最终来自Magazine.id。
3. Article takes Article.magazine\_id into account when loading Article.writer, even though it *doesn't* write to it on behalf of this relationship.在加载Article.writer时，文章会将Article.magazine\_id考虑在内，即使它不代表这种关系写信给它。

To get just #1 and #2, we could specify only Article.writer\_id as the "foreign keys" for Article.writer:

**class** **Article**(Base):

*# ...*

writer = relationship("Writer", foreign\_keys='Article.writer\_id')

However, this has the effect of Article.writer not taking Article.magazine\_id into account when querying against Writer:

**SELECT** article.article\_id **AS** article\_article\_id,

article.magazine\_id **AS** article\_magazine\_id,

article.writer\_id **AS** article\_writer\_id

**FROM** article

**JOIN** writer **ON** writer.id = article.writer\_id

Therefore, to get at all of #1, #2, and #3, we express the join condition as well as which columns to be written by combining [primaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.primaryjoin" \o "sqlalchemy.orm.relationship) fully, along with either the[foreign\_keys](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.foreign_keys" \o "sqlalchemy.orm.relationship) argument, or more succinctly by annotating with [foreign()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.foreign" \o "sqlalchemy.orm.foreign):

因此，为了得到＃1，＃2和＃3的所有内容，我们通过使用foreign()注释来表达连接条件以及通过将primaryjoin完全组合，或者theforeign\_keys参数或更简洁的结合来写入哪些列，：

**class** **Article**(Base):

*# ...*

writer = relationship(

"Writer",

primaryjoin="and\_(Writer.id == foreign(Article.writer\_id), "

"Writer.magazine\_id == Article.magazine\_id)")

*Changed in version 1.0.0:*the ORM will attempt to warn when a column is used as the synchronization target from more than one relationship simultaneously.

Non-relational Comparisons / Materialized Path

**Warning**

this section details an experimental feature.

Using custom expressions means we can produce unorthodox join conditions that don't obey the usual primary/foreign key model. One such example is the materialized path pattern, where we compare strings for overlapping path tokens in order to produce a tree structure.

使用自定义表达式意味着我们可以产生不符合通常的主/外键模型的非正统连接条件。 一个这样的示例是物化路径模式，其中我们比较重叠路径令牌的字符串，以便生成树结构。

Through careful use of [foreign()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.foreign" \o "sqlalchemy.orm.foreign) and [remote()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.remote" \o "sqlalchemy.orm.remote), we can build a relationship that effectively produces a rudimentary materialized path system. Essentially, when [foreign()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.foreign" \o "sqlalchemy.orm.foreign) and [remote()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.remote" \o "sqlalchemy.orm.remote) are on the *same* side of the comparison expression, the relationship is considered to be "one to many"; when they are on *different* sides, the relationship is considered to be "many to one". For the comparison we'll use here, we'll be dealing with collections so we keep things configured as "one to many":

通过仔细使用foreign()和remote()，我们可以建立一个有效地产生一个基本的物化路径系统的关系。 本质上，当foreign()和remote()位于比较表达式的同一边时，该关系被认为是“一对多”; 当他们在不同的方面，这种关系被认为是“一对一”。 为了比较我们将在这里使用，我们将处理集合，所以我们将配置为“一对多”：

**class** **Element**(Base):

\_\_tablename\_\_ = 'element'

path = Column(String, primary\_key=**True**)

descendants = relationship('Element',

primaryjoin=

remote(foreign(path)).like(

path.concat('/%')),

viewonly=**True**,

order\_by=path)

Above, if given an Element object with a path attribute of "/foo/bar2", we seek for a load of Element.descendants to look like:

SELECT element.path AS element\_path

FROM element

WHERE element.path LIKE ('/foo/bar2' || '/%') ORDER BY element.path

*New in version 0.9.5:*Support has been added to allow a single-column comparison to itself within a primaryjoin condition, as well as for primaryjoin conditions that use [ColumnOperators.like()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.like" \o "sqlalchemy.sql.operators.ColumnOperators.like) as the comparison operator.

版本0.9.5中的新增功能：添加了支持以允许在一个Primary Join条件下对其进行单列比较，以及使用ColumnOperators.like()作为比较运算符的primaryjoin条件。

Self-Referential Many-to-Many Relationship

Many to many relationships can be customized by one or both of [primaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.primaryjoin" \o "sqlalchemy.orm.relationship) and [secondaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.secondaryjoin" \o "sqlalchemy.orm.relationship) - the latter is significant for a relationship that specifies a many-to-many reference using the [secondary](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.secondary" \o "sqlalchemy.orm.relationship) argument. A common situation which involves the usage of [primaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.primaryjoin" \o "sqlalchemy.orm.relationship) and [secondaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.secondaryjoin" \o "sqlalchemy.orm.relationship) is when establishing a many-to-many relationship from a class to itself, as shown below:

许多关系可以由primaryjoin和secondaryjoin中的一个或两个进行自定义 - 后者对于使用secondary参数指定多对多引用的关系很重要。 一个常见的情况，涉及到使用一个小连接和二次连接是建立一个多对多关系从一个类到自己，如下所示：

**from** **sqlalchemy** **import** Integer, ForeignKey, String, Column, Table

**from** **sqlalchemy.ext.declarative** **import** declarative\_base

**from** **sqlalchemy.orm** **import** relationship

Base = declarative\_base()

node\_to\_node = Table("node\_to\_node", Base.metadata,

Column("left\_node\_id", Integer, ForeignKey("node.id"), primary\_key=**True**),

Column("right\_node\_id", Integer, ForeignKey("node.id"), primary\_key=**True**))

**class** **Node**(Base):

\_\_tablename\_\_ = 'node'

id = Column(Integer, primary\_key=**True**)

label = Column(String)

right\_nodes = relationship("Node",

secondary=node\_to\_node,

primaryjoin=id==node\_to\_node.c.left\_node\_id,

secondaryjoin=id==node\_to\_node.c.right\_node\_id,

backref="left\_nodes"

)

Where above, SQLAlchemy can't know automatically which columns should connect to which for the right\_nodes and left\_nodes relationships. The [primaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.primaryjoin" \o "sqlalchemy.orm.relationship) and [secondaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.secondaryjoin" \o "sqlalchemy.orm.relationship) arguments establish how we'd like to join to the association table. In the Declarative form above, as we are declaring these conditions within the Python block that corresponds to the Node class, the id variable is available directly as the [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) object we wish to join with.

在上面的位置，SQLAlchemy无法自动知道哪些列应该连接到right\_nodes和left\_nodes关系。 primaryjoin和secondaryjoin参数建立了我们如何加入关联表。 在上面的声明式中，正如我们在与Node类对应的Python块中声明这些条件一样，id变量直接作为我们希望加入的Column对象可用。

Alternatively, we can define the [primaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.primaryjoin" \o "sqlalchemy.orm.relationship) and [secondaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.secondaryjoin" \o "sqlalchemy.orm.relationship) arguments using strings, which is suitable in the case that our configuration does not have either the Node.id column object available yet or the node\_to\_node table perhaps isn't yet available. When referring to a plain [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object in a declarative string, we use the string name of the table as it is present in the [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData):

或者，我们可以使用字符串定义primaryjoin和secondaryjoin参数，这适用于我们的配置没有可用的Node.id列对象或node\_to\_node表还不可用的情况。 当在声明性字符串中引用一个简单的Table对象时，我们使用MetaData中存在的表的字符串名称：

**class** **Node**(Base):

\_\_tablename\_\_ = 'node'

id = Column(Integer, primary\_key=**True**)

label = Column(String)

right\_nodes = relationship("Node",

secondary="node\_to\_node",

primaryjoin="Node.id==node\_to\_node.c.left\_node\_id",

secondaryjoin="Node.id==node\_to\_node.c.right\_node\_id",

backref="left\_nodes"

)

A classical mapping situation here is similar, where node\_to\_node can be joined to node.c.id:

这里的经典映射情况是类似的，其中node\_to\_node可以加入到node.c.id中：

**from** **sqlalchemy** **import** Integer, ForeignKey, String, Column, Table, MetaData

**from** **sqlalchemy.orm** **import** relationship, mapper

metadata = MetaData()

node\_to\_node = Table("node\_to\_node", metadata,

Column("left\_node\_id", Integer, ForeignKey("node.id"), primary\_key=**True**),

Column("right\_node\_id", Integer, ForeignKey("node.id"), primary\_key=**True**))

node = Table("node", metadata,

Column('id', Integer, primary\_key=**True**),

Column('label', String))**class** **Node**(object):

**pass**

mapper(Node, node, properties={

'right\_nodes':relationship(Node,

secondary=node\_to\_node,

primaryjoin=node.c.id==node\_to\_node.c.left\_node\_id,

secondaryjoin=node.c.id==node\_to\_node.c.right\_node\_id,

backref="left\_nodes"

)})

Note that in both examples, the [backref](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.backref" \o "sqlalchemy.orm.relationship) keyword specifies a left\_nodes backref - when [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) creates the second relationship in the reverse direction, it's smart enough to reverse the [primaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.primaryjoin" \o "sqlalchemy.orm.relationship) and [secondaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.secondaryjoin" \o "sqlalchemy.orm.relationship) arguments.

请注意，在这两个示例中，backref关键字指定一个left\_nodes backref - 当relationship()以相反的方向创建第二个关系时，足够聪明地反转primaryjoin和secondaryjoin参数。

Composite "Secondary" Joins

**Note**注意

This section features some new and experimental features of SQLAlchemy.

本部分介绍了SQLAlchemy的一些新功能和实验功能。

Sometimes, when one seeks to build a [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) between two tables there is a need for more than just two or three tables to be involved in order to join them. This is an area of [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) where one seeks to push the boundaries of what's possible, and often the ultimate solution to many of these exotic use cases needs to be hammered out on the SQLAlchemy mailing list.

有时候，当一个人寻求在两个表之间建立一个关系()时，需要加入不止两个或三个表来加入它们。 这是一个关系领域()，它试图推动可能的界限，并且经常在SQLAlchemy邮件列表中敲击许多这些外来用例的最终解决方案。

In more recent versions of SQLAlchemy, the [secondary](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.secondary" \o "sqlalchemy.orm.relationship) parameter can be used in some of these cases in order to provide a composite target consisting of multiple tables. Below is an example of such a join condition (requires version 0.9.2 at least to function as is):

在最新版本的SQLAlchemy中，可以在这些情况中使用辅助参数，以提供由多个表组成的组合目标。 下面是一个这样的连接条件的例子(至少要0.9.2版本的功能)：

**class** **A**(Base):

\_\_tablename\_\_ = 'a'

id = Column(Integer, primary\_key=**True**)

b\_id = Column(ForeignKey('b.id'))

d = relationship("D",

secondary="join(B, D, B.d\_id == D.id)."

"join(C, C.d\_id == D.id)",

primaryjoin="and\_(A.b\_id == B.id, A.id == C.a\_id)",

secondaryjoin="D.id == B.d\_id",

uselist=**False**

)

**class** **B**(Base):

\_\_tablename\_\_ = 'b'

id = Column(Integer, primary\_key=**True**)

d\_id = Column(ForeignKey('d.id'))

**class** **C**(Base):

\_\_tablename\_\_ = 'c'

id = Column(Integer, primary\_key=**True**)

a\_id = Column(ForeignKey('a.id'))

d\_id = Column(ForeignKey('d.id'))

**class** **D**(Base):

\_\_tablename\_\_ = 'd'

id = Column(Integer, primary\_key=**True**)

In the above example, we provide all three of [secondary](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.secondary" \o "sqlalchemy.orm.relationship), [primaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.primaryjoin" \o "sqlalchemy.orm.relationship), and [secondaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.secondaryjoin" \o "sqlalchemy.orm.relationship), in the declarative style referring to the named tables a, b, c, d directly. A query from A to D looks like:

在上面的例子中，我们提供了所有三个secondary，primaryjoin和secondaryjoin，在声明式中直接引用了命名表a，b，c，d。 从A到D的查询看起来像：

session.query(A).join(A.d).all()

SELECT a.id AS a\_id, a.b\_id AS a\_b\_id

FROM a JOIN (

b AS b\_1 JOIN d AS d\_1 ON b\_1.d\_id = d\_1.id

JOIN c AS c\_1 ON c\_1.d\_id = d\_1.id)

ON a.b\_id = b\_1.id AND a.id = c\_1.a\_id JOIN d ON d.id = b\_1.d\_id

In the above example, we take advantage of being able to stuff multiple tables into a "secondary" container, so that we can join across many tables while still keeping things "simple" for [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship), in that there's just "one" table on both the "left" and the "right" side; the complexity is kept within the middle.

在上面的例子中，我们利用能够将多个表填充到“二次”容器中，以便我们可以跨越多个表来连接，同时保持[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)的“简单”，因为只有“一个”表 在“左”和“右”两侧; 复杂性保持在中间。

*New in version 0.9.2:*

Support is improved for allowing a [join()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.join" \o "sqlalchemy.sql.expression.join) construct to be used directly as the target of the [secondary](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.secondary" \o "sqlalchemy.orm.relationship) argument, including support for joins, eager joins and lazy loading, as well as support within declarative to specify complex conditions such as joins involving class names as targets.

支持被改进，允许将[join()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.join" \o "sqlalchemy.sql.expression.join)结构直接用作[secondary](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.secondary" \o "sqlalchemy.orm.relationship)参数的目标，包括对连接的支持，渴望加入和延迟加载，以及在声明中支持以指定复杂条件，例如涉及类名称的连接目标。

Relationship to Non Primary Mapper

In the previous section, we illustrated a technique where we used [secondary](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.secondary" \o "sqlalchemy.orm.relationship) in order to place additional tables within a join condition. There is one complex join case where even this technique is not sufficient; when we seek to join from A to B, making use of any number of C, D, etc. in between, however there are also join conditions between A and B *directly*. In this case, the join from A to B may be difficult to express with just a complex [primaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.primaryjoin" \o "sqlalchemy.orm.relationship) condition, as the intermediary tables may need special handling, and it is also not expressable with a [secondary](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.secondary" \o "sqlalchemy.orm.relationship) object, since the A->secondary->B pattern does not support any references between A and B directly. When this ****extremely advanced**** case arises, we can resort to creating a second mapping as a target for the relationship. This is where we use [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper) in order to make a mapping to a class that includes all the additional tables we need for this join. In order to produce this mapper as an "alternative" mapping for our class, we use the [non\_primary](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.params.non_primary" \o "sqlalchemy.orm.mapper) flag.

在上一节中，我们说明了一种技术，我们使用二级以在附加条件中放置其他表。有一个复杂的连接情况，即使这种技术还不够;当我们寻求从A到B加入时，利用任何数量的C，D等，但是也直接在A和B之间加入条件。在这种情况下，由于中间表可能需要特殊处理，所以从A到B的连接可能很难表达，因为中间表可能需要特殊处理，并且也不能用一个[secondary](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.secondary" \o "sqlalchemy.orm.relationship)对象来表达，因为A->secondary->B模式不直接支持A和B之间的任何引用。当这种非常先进的情况出现时，我们可以诉诸创建第二个映射作为关系的目标。这是我们使用[mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper)，以便映射到一个类，包括我们需要这个连接的所有其他表。为了产生这个映射器作为我们类的“替代”映射，我们使用[non\_primary](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.params.non_primary" \o "sqlalchemy.orm.mapper)标志。

Below illustrates a [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) with a simple join from A to B, however the primaryjoin condition is augmented with two additional entities C and D, which also must have rows that line up with the rows in both A and B simultaneously:

下面示出了一个与A到B的简单连接的[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)，但是，primaryjoin条件增加了两个附加实体C和D，它们也必须同时具有与A和B中的行对齐的行：

**class** **A**(Base):

\_\_tablename\_\_ = 'a'

id = Column(Integer, primary\_key=**True**)

b\_id = Column(ForeignKey('b.id'))

**class** **B**(Base):

\_\_tablename\_\_ = 'b'

id = Column(Integer, primary\_key=**True**)

**class** **C**(Base):

\_\_tablename\_\_ = 'c'

id = Column(Integer, primary\_key=**True**)

a\_id = Column(ForeignKey('a.id'))

**class** **D**(Base):

\_\_tablename\_\_ = 'd'

id = Column(Integer, primary\_key=**True**)

c\_id = Column(ForeignKey('c.id'))

b\_id = Column(ForeignKey('b.id'))

*# 1. set up the join() as a variable, so we can refer# to it in the mapping multiple times.*

j = join(B, D, D.b\_id == B.id).join(C, C.id == D.c\_id)

*# 2. Create a new mapper() to B, with non\_primary=True.# Columns in the join with the same name must be# disambiguated within the mapping, using named properties.*

B\_viacd = mapper(B, j, non\_primary=**True**, properties={

"b\_id": [j.c.b\_id, j.c.d\_b\_id],

"d\_id": j.c.d\_id

})

A.b = relationship(B\_viacd, primaryjoin=A.b\_id == B\_viacd.c.b\_id)

In the above case, our non-primary mapper for B will emit for additional columns when we query; these can be ignored:

在上述情况下，当我们查询时，我们的非主要映射器B将会发出额外的列; 这些可以忽略：

sess.query(A).join(A.b).all()

SELECT a.id AS a\_id, a.b\_id AS a\_b\_id

FROM a JOIN (b JOIN d ON d.b\_id = b.id JOIN c ON c.id = d.c\_id) ON a.b\_id = b.id

**Building Query-Enabled Properties**

Very ambitious custom join conditions may fail to be directly persistable, and in some cases may not even load correctly. To remove the persistence part of the equation, use the flag [viewonly](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.viewonly" \o "sqlalchemy.orm.relationship) on the [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship), which establishes it as a read-only attribute (data written to the collection will be ignored on flush()). However, in extreme cases, consider using a regular Python property in conjunction with [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) as follows:

非常雄心勃勃的自定义连接条件可能无法直接持久化，并且在某些情况下甚至可能无法正确加载。 要删除方程的持久性部分，请在relationship()上使用该[viewonly](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.viewonly" \o "sqlalchemy.orm.relationship)标志，该关系将它建立为只读属性(写入集合的数据将在flush()上忽略)。 但是，在极端情况下，请考虑将常规Python属性与Query结合使用如下：

**class** **User**(Base):

\_\_tablename\_\_ = 'user'

id = Column(Integer, primary\_key=True)

**def** \_get\_addresses(self):

**return** object\_session(self).query(Address).with\_parent(self).filter(

...

).all()

addresses = property(\_get\_addresses)

# **3.5 Collection Configuration and Techniques**

The [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) function defines a linkage between two classes. When the linkage defines a one-to-many or many-to-many relationship, it's represented as a Python collection when objects are loaded and manipulated. This section presents additional information about collection configuration and techniques.

relationship()函数定义了两个类之间的链接。 当链接定义一对多或多对多关系时，当对象被加载和操作时，它被表示为Python集合。 本节介绍有关集合配置和技术的其他信息。

Working with Large Collections

The default behavior of [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) is to fully load the collection of items in, as according to the loading strategy of the relationship. Additionally, the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)by default only knows how to delete objects which are actually present within the session. When a parent instance is marked for deletion and flushed, the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)loads its full list of child items in so that they may either be deleted as well, or have their foreign key value set to null; this is to avoid constraint violations. For large collections of child items, there are several strategies to bypass full loading of child items both at load time as well as deletion time.

relationship()的默认行为是根据关系的加载策略来完全加载项目的集合。 另外，Sessionby默认只知道如何删除会话中实际存在的对象。 当父实例被标记为删除和刷新时，会话会将其完整的子项目列表加载，以便它们也可能被删除，或者将其外键值设置为null; 这是为了避免约束违规。 对于大量的子项目集合，有几种策略可以在加载时间以及删除时间的情况下绕过子项目的完整加载。

### Dynamic Relationship Loaders

A key feature to enable management of a large collection is the so-called "dynamic" relationship. This is an optional form of [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) which returns a [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query)object in place of a collection when accessed. [filter()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.filter" \o "sqlalchemy.orm.query.Query.filter) criterion may be applied as well as limits and offsets, either explicitly or via array slices:

能够管理大型集合的一个关键特征就是所谓的“动态”关系。 这是一个可选的relationship()形式，它在访问时返回一个Queryobject来代替一个集合。 可以通过显式地或通过数组切片来应用filter()标准以及限制和偏移：

**class** **User**(Base):

\_\_tablename\_\_ = 'user'

posts = relationship(Post, lazy="dynamic")

jack = session.query(User).get(id)

*# filter Jack's blog posts*

posts = jack.posts.filter(Post.headline=='this is a post')

*# apply array slices*

posts = jack.posts[5:20]

The dynamic relationship supports limited write operations, via the append() and remove() methods:

动态关系支持有限的写入操作，通过append()和remove()方法：

oldpost = jack.posts.filter(Post.headline=='old post').one()jack.posts.remove(oldpost)

jack.posts.append(Post('new post'))

Since the read side of the dynamic relationship always queries the database, changes to the underlying collection will not be visible until the data has been flushed. However, as long as "autoflush" is enabled on the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) in use, this will occur automatically each time the collection is about to emit a query.

由于动态关系的读取端始终查询数据库，因此在刷新数据之前，底层集合的更改将不可见。 但是，只要在使用的[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)中启用“autoflush”，每次收集即将发出查询时，都会自动发生。

To place a dynamic relationship on a backref, use the [backref()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.backref" \o "sqlalchemy.orm.backref) function in conjunction with lazy='dynamic':

要在backref上放置动态关系，请将backref()函数与lazy ='dynamic'结合使用：

**class** **Post**(Base):

\_\_table\_\_ = posts\_table

user = relationship(User,

backref=backref('posts', lazy='dynamic')

)

Note that eager/lazy loading options cannot be used in conjunction dynamic relationships at this time.

请注意，此时，热切/延迟加载选项不能与动态关系结合使用。

**Note**

The [dynamic\_loader()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.dynamic_loader" \o "sqlalchemy.orm.dynamic_loader) function is essentially the same as [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) with the lazy='dynamic' argument specified.

[dynamic\_loader()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.dynamic_loader" \o "sqlalchemy.orm.dynamic_loader)函数与与指定的lazy='dynamic'参数的[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)基本相同。

**Warning**

The "dynamic" loader applies to ****collections only****. It is not valid to use "dynamic" loaders with many-to-one, one-to-one, or uselist=False relationships. Newer versions of SQLAlchemy emit warnings or exceptions in these cases.

“动态”加载器仅适用于集合。 使用具有多对一，一对一或uselist = False关系的“动态”装载机是无效的。 在这些情况下，较新版本的SQLAlchemy会发出警告或异常。

### Setting Noload, RaiseLoad

A "noload" relationship never loads from the database, even when accessed. It is configured using lazy='noload':

即使在访问时，“noload”关系也不会从数据库加载。 它使用lazy ='noload'进行配置：

**class** **MyClass**(Base):

\_\_tablename\_\_ = 'some\_table'

children = relationship(MyOtherClass, lazy='noload')

Above, the children collection is fully writeable, and changes to it will be persisted to the database as well as locally available for reading at the time they are added. However when instances of MyClass are freshly loaded from the database, the children collection stays empty. The noload strategy is also available on a query option basis using the [orm.noload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.noload" \o "sqlalchemy.orm.noload) loader option.

以上，孩子集合是完全可写的，并且它的更改将被持久化到数据库，以及当地可用于在添加时进行阅读。 但是，当MyClass的实例从数据库中新加载时，children集合将保持为空。 noload策略也可以使用orm.noload()loader选项基于查询选项。

Alternatively, a "raise"-loaded relationship will raise an [InvalidRequestError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.InvalidRequestError" \o "sqlalchemy.exc.InvalidRequestError) where the attribute would normally emit a lazy load:

或者，“加载”关系将引发一个InvalidRequestError，其中属性通常会发出一个懒加载：

**class** **MyClass**(Base):

\_\_tablename\_\_ = 'some\_table'

children = relationship(MyOtherClass, lazy='raise')

Above, attribute access on the children collection will raise an exception if it was not previously eagerloaded. This includes read access but for collections will also affect write access, as collections can't be mutated without first loading them. The rationale for this is to ensure that an application is not emitting any unexpected lazy loads within a certain context. Rather than having to read through SQL logs to determine that all necessary attributes were eager loaded, the "raise" strategy will cause unloaded attributes to raise immediately if accessed. The raise strategy is also available on a query option basis using the [orm.raiseload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.raiseload" \o "sqlalchemy.orm.raiseload) loader option.

以上，如果以前没有加载，则子集合上的属性访问将引发异常。 这包括读取访问，但是对于集合也会影响写入访问，因为集合在不先进行加载的情况下就不能被突变。 这样做的理由是确保应用程序不会在某个上下文中发出任何意外的惰性负载。 而不必读取SQL日志来确定所有必要的属性都被加载，“加注”策略将导致卸载的属性在访问时立即升高。 升级策略也可以使用orm.raiseload()加载程序选项基于查询选项。

*New in version 1.1:*added the "raise" loader strategy.

1.1版新增：加载“加载”策略。

**See also**

[Preventing unwanted lazy loads using raiseload](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "prevent-lazy-with-raiseload)

### Using Passive Deletes

Use [passive\_deletes](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.passive_deletes" \o "sqlalchemy.orm.relationship) to disable child object loading on a DELETE operation, in conjunction with "ON DELETE (CASCADE|SET NULL)" on your database to automatically cascade deletes to child objects:

使用[passive\_deletes](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.passive_deletes" \o "sqlalchemy.orm.relationship)可以在DELETE操作上禁用子对象加载，并结合数据库中的“ON DELETE(CASCADE | SET NULL)”自动级联删除子对象：

**class** **MyClass**(Base):

\_\_tablename\_\_ = 'mytable'

id = Column(Integer, primary\_key=**True**)

children = relationship("MyOtherClass",

cascade="all, delete-orphan",

passive\_deletes=**True**)

**class** **MyOtherClass**(Base):

\_\_tablename\_\_ = 'myothertable'

id = Column(Integer, primary\_key=**True**)

parent\_id = Column(Integer,

ForeignKey('mytable.id', ondelete='CASCADE')

)

**Note**

To use "ON DELETE CASCADE", the underlying database engine must support foreign keys.

要使用“ON DELETE CASCADE”，底层数据库引擎必须支持外键。

* When using MySQL, an appropriate storage engine must be selected. See [CREATE TABLE arguments including Storage Engines](http://docs.sqlalchemy.org/en/rel_1_1/dialects/mysql.html" \l "mysql-storage-engines) for details.使用MySQL时，必须选择合适的存储引擎。 有关详细信息，请参阅包含存储引擎的CREATE TABLE参数。
* When using SQLite, foreign key support must be enabled explicitly. See [Foreign Key Support](http://docs.sqlalchemy.org/en/rel_1_1/dialects/sqlite.html" \l "sqlite-foreign-keys) for details.使用SQLite时，必须明确启用外键支持。 有关详细信息，请参阅外键支持

When [passive\_deletes](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.passive_deletes" \o "sqlalchemy.orm.relationship) is applied, the children relationship will not be loaded into memory when an instance of MyClass is marked for deletion. The cascade="all, delete-orphan" *will* take effect for instances of MyOtherClass which are currently present in the session; however for instances of MyOtherClass which are not loaded, SQLAlchemy assumes that "ON DELETE CASCADE" rules will ensure that those rows are deleted by the database.

当应用passive\_deletes时，当MyClass的实例被标记为删除时，子关系将不会被加载到内存中。 cascade="all, delete-orphan"将对会话中当前存在的MyOtherClass实例生效; 然而，对于未加载的MyOtherClass的实例，SQLAlchemy假定“ON DELETE CASCADE”规则将确保数据库中的那些行被删除。

**See also**

[orm.mapper.passive\_deletes](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.params.passive_deletes" \o "sqlalchemy.orm.mapper) - similar feature on [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper)

## Customizing Collection Access

Mapping a one-to-many or many-to-many relationship results in a collection of values accessible through an attribute on the parent instance. By default, this collection is a list:

映射一对多或多对多关系导致可通过父实例上的属性访问的值的集合。 默认情况下，此集合是一个列表：

**class** **Parent**(Base):

\_\_tablename\_\_ = 'parent'

parent\_id = Column(Integer, primary\_key=**True**)

children = relationship(Child)

parent = Parent()parent.children.append(Child())print(parent.children[0])

Collections are not limited to lists. Sets, mutable sequences and almost any other Python object that can act as a container can be used in place of the default list, by specifying the [collection\_class](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.collection_class" \o "sqlalchemy.orm.relationship) option on [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship):

集合不限于列表。 通过在[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)上指定[collection\_class](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.collection_class" \o "sqlalchemy.orm.relationship)选项，可以使用集合，可变序列和几乎可以充当容器的任何其他Python对象来代替默认列表：

**class** **Parent**(Base):

\_\_tablename\_\_ = 'parent'

parent\_id = Column(Integer, primary\_key=**True**)

*# use a set*

children = relationship(Child, collection\_class=set)

parent = Parent()child = Child()parent.children.add(child)**assert** child **in** parent.children

### Dictionary Collections

A little extra detail is needed when using a dictionary as a collection. This because objects are always loaded from the database as lists, and a key-generation strategy must be available to populate the dictionary correctly. The [attribute\_mapped\_collection()](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "sqlalchemy.orm.collections.attribute_mapped_collection" \o "sqlalchemy.orm.collections.attribute_mapped_collection) function is by far the most common way to achieve a simple dictionary collection. It produces a dictionary class that will apply a particular attribute of the mapped class as a key. Below we map an Item class containing a dictionary of Note items keyed to the Note.keyword attribute:

使用字典作为集合需要一点额外的细节。 这是因为对象总是作为列表从数据库加载，并且必须可以使用密钥生成策略来正确填充字典。 [attribute\_mapped\_collection()](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "sqlalchemy.orm.collections.attribute_mapped_collection" \o "sqlalchemy.orm.collections.attribute_mapped_collection)函数是实现简单字典集合的最常用方式。 它产生一个字典类，它将映射类的特定属性应用于一个键。 下面我们映射一个Item类，其中包含一个键入到Note.keyword属性的Note项目的字典：

**from** **sqlalchemy** **import** Column, Integer, String, ForeignKey

**from** **sqlalchemy.orm** **import** relationship

**from** **sqlalchemy.orm.collections** **import** attribute\_mapped\_collection

**from** **sqlalchemy.ext.declarative** **import** declarative\_base

Base = declarative\_base()

**class** **Item**(Base):

\_\_tablename\_\_ = 'item'

id = Column(Integer, primary\_key=**True**)

notes = relationship("Note",

collection\_class=attribute\_mapped\_collection('keyword'),

cascade="all, delete-orphan")

**class** **Note**(Base):

\_\_tablename\_\_ = 'note'

id = Column(Integer, primary\_key=**True**)

item\_id = Column(Integer, ForeignKey('item.id'), nullable=**False**)

keyword = Column(String)

text = Column(String)

**def** \_\_init\_\_(self, keyword, text):

self.keyword = keyword

self.text = text

Item.notes is then a dictionary:

**>>>** item = Item()

**>>>** item.notes['a'] = Note('a', 'atext')

**>>>** item.notes.items()

{'a': <\_\_main\_\_.Note object at 0x2eaaf0>}

[attribute\_mapped\_collection()](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "sqlalchemy.orm.collections.attribute_mapped_collection" \o "sqlalchemy.orm.collections.attribute_mapped_collection) will ensure that the .keyword attribute of each Note complies with the key in the dictionary. Such as, when assigning to Item.notes, the dictionary key we supply must match that of the actual Note object:

[attribute\_mapped\_collection()](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "sqlalchemy.orm.collections.attribute_mapped_collection" \o "sqlalchemy.orm.collections.attribute_mapped_collection)将确保每个Note的.keyword属性符合字典中的键。 例如，当分配给Item.notes时，我们提供的字典键必须与实际的Note对象的字典键匹配：

item = Item()item.notes = {

'a': Note('a', 'atext'),

'b': Note('b', 'btext')

}

The attribute which [attribute\_mapped\_collection()](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "sqlalchemy.orm.collections.attribute_mapped_collection" \o "sqlalchemy.orm.collections.attribute_mapped_collection) uses as a key does not need to be mapped at all! Using a regular Python @property allows virtually any detail or combination of details about the object to be used as the key, as below when we establish it as a tuple of Note.keyword and the first ten letters of the Note.text field:

attribute\_mapped\_collection()用作键的属性根本不需要映射！ 使用常规的Python @property几乎可以使用任何关于要用作关键字的对象的细节或细节的组合，如下所示，当我们将其建立为Note.keyword的元组和Note.text字段的前十个字母时：

**class** **Item**(Base):

\_\_tablename\_\_ = 'item'

id = Column(Integer, primary\_key=**True**)

notes = relationship("Note",

collection\_class=attribute\_mapped\_collection('note\_key'),

backref="item",

cascade="all, delete-orphan")

**class** **Note**(Base):

\_\_tablename\_\_ = 'note'

id = Column(Integer, primary\_key=**True**)

item\_id = Column(Integer, ForeignKey('item.id'), nullable=**False**)

keyword = Column(String)

text = Column(String)

**@property**

**def** note\_key(self):

**return** (self.keyword, self.text[0:10])

**def** \_\_init\_\_(self, keyword, text):

self.keyword = keyword

self.text = text

Above we added a Note.item backref. Assigning to this reverse relationship, the Note is added to the Item.notes dictionary and the key is generated for us automatically:

上面我们添加了一个Note.item backref。 分配到这种反向关系，该Note被添加到Item.notes字典中，并且该键自动为我们生成：

**>>>** item = Item()

**>>>** n1 = Note("a", "atext")

**>>>** n1.item = item

**>>>** item.notes

{('a', 'atext'): <\_\_main\_\_.Note object at 0x2eaaf0>}

Other built-in dictionary types include [column\_mapped\_collection()](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "sqlalchemy.orm.collections.column_mapped_collection" \o "sqlalchemy.orm.collections.column_mapped_collection), which is almost like [attribute\_mapped\_collection()](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "sqlalchemy.orm.collections.attribute_mapped_collection" \o "sqlalchemy.orm.collections.attribute_mapped_collection) except given the [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) object directly:

其他内置的字典类型包括[column\_mapped\_collection()](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "sqlalchemy.orm.collections.column_mapped_collection" \o "sqlalchemy.orm.collections.column_mapped_collection)，它几乎像[attribute\_mapped\_collection()](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "sqlalchemy.orm.collections.attribute_mapped_collection" \o "sqlalchemy.orm.collections.attribute_mapped_collection)，除了直接给出[Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column)对象：

**from** **sqlalchemy.orm.collections** **import** column\_mapped\_collection

**class** **Item**(Base):

\_\_tablename\_\_ = 'item'

id = Column(Integer, primary\_key=**True**)

notes = relationship("Note",

collection\_class=column\_mapped\_collection(Note.\_\_table\_\_.c.keyword),

cascade="all, delete-orphan")

as well as [mapped\_collection()](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "sqlalchemy.orm.collections.mapped_collection" \o "sqlalchemy.orm.collections.mapped_collection) which is passed any callable function. Note that it's usually easier to use [attribute\_mapped\_collection()](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "sqlalchemy.orm.collections.attribute_mapped_collection" \o "sqlalchemy.orm.collections.attribute_mapped_collection) along with a @property as mentioned earlier:

以及传递任何可调用函数的[mapped\_collection()](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "sqlalchemy.orm.collections.mapped_collection" \o "sqlalchemy.orm.collections.mapped_collection)。 请注意，如前所述，通常更容易使用[attribute\_mapped\_collection()](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "sqlalchemy.orm.collections.attribute_mapped_collection" \o "sqlalchemy.orm.collections.attribute_mapped_collection)以及@property：

**from** **sqlalchemy.orm.collections** **import** mapped\_collection

**class** **Item**(Base):

\_\_tablename\_\_ = 'item'

id = Column(Integer, primary\_key=**True**)

notes = relationship("Note",

collection\_class=mapped\_collection(**lambda** note: note.text[0:10]),

cascade="all, delete-orphan")

Dictionary mappings are often combined with the "Association Proxy" extension to produce streamlined dictionary views. See [Proxying to Dictionary Based Collections](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "proxying-dictionaries)and [Composite Association Proxies](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "composite-association-proxy) for examples.

字典映射通常与“关联代理”扩展组合，以生成流线型字典视图。 参见代理到基于字典的集合和复合关联代理的例子。

sqlalchemy.orm.collections.**attribute\_mapped\_collection**(*attr\_name*)

A dictionary-based collection type with attribute-based keying.

基于字典的基于属性的键控的集合类型。

Returns a [MappedCollection](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "sqlalchemy.orm.collections.MappedCollection" \o "sqlalchemy.orm.collections.MappedCollection) factory with a keying based on the 'attr\_name' attribute of entities in the collection, where attr\_name is the string name of the attribute.

使用基于集合中实体的“attr\_name”属性的键控返回[MappedCollection](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "sqlalchemy.orm.collections.MappedCollection" \o "sqlalchemy.orm.collections.MappedCollection)工厂，其中attr\_name是属性的字符串名称。

The key value must be immutable for the lifetime of the object. You can not, for example, map on foreign key values if those key values will change during the session, i.e. from None to a database-assigned integer after a session flush.

在对象的生命周期中，关键值必须是不可变的。 例如，如果这些键值在会话中更改，即在会话刷新后，从无到数据库分配的整数，则不能映射外键值。

sqlalchemy.orm.collections.**column\_mapped\_collection**(*mapping\_spec*)

A dictionary-based collection type with column-based keying.

基于字典的基于列的键控的集合类型。

Returns a [MappedCollection](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "sqlalchemy.orm.collections.MappedCollection" \o "sqlalchemy.orm.collections.MappedCollection) factory with a keying function generated from mapping\_spec, which may be a Column or a sequence of Columns.

使用从mapping\_spec生成的键控函数返回一个[MappedCollection](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "sqlalchemy.orm.collections.MappedCollection" \o "sqlalchemy.orm.collections.MappedCollection)工厂，该函数可能是一列或一列列。

The key value must be immutable for the lifetime of the object. You can not, for example, map on foreign key values if those key values will change during the session, i.e. from None to a database-assigned integer after a session flush.

在对象的生命周期中，关键值必须是不可变的。 例如，如果这些键值在会话中更改，即在会话刷新后，从无到数据库分配的整数，则不能映射外键值。

sqlalchemy.orm.collections.**mapped\_collection**(*keyfunc*)

A dictionary-based collection type with arbitrary keying.

具有任意键控的基于字典的收藏类型。

Returns a [MappedCollection](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "sqlalchemy.orm.collections.MappedCollection" \o "sqlalchemy.orm.collections.MappedCollection) factory with a keying function generated from keyfunc, a callable that takes an entity and returns a key value.

使用从keyfunc生成的键控函数返回一个[MappedCollection](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "sqlalchemy.orm.collections.MappedCollection" \o "sqlalchemy.orm.collections.MappedCollection)工厂，该函数需要一个实体并返回一个键值。

The key value must be immutable for the lifetime of the object. You can not, for example, map on foreign key values if those key values will change during the session, i.e. from None to a database-assigned integer after a session flush.

在对象的生命周期中，关键值必须是不可变的。 例如，如果这些键值在会话中更改，即在会话刷新后，从无到数据库分配的整数，则不能映射外键值。

## Custom Collection Implementations

You can use your own types for collections as well. In simple cases, inherting from list or set, adding custom behavior, is all that's needed. In other cases, special decorators are needed to tell SQLAlchemy more detail about how the collection operates.

您也可以使用自己的类型进行收藏。 在简单的情况下，从列表或集合继承，添加自定义行为是所有需要的。 在其他情况下，需要特殊的装饰器来告知SQLAlchemy有关集合如何运作的更多细节。

**Do I need a custom collection implementation?**

**我需要一个自定义集合实现？**

In most cases not at all! The most common use cases for a "custom" collection is one that validates or marshals incoming values into a new form, such as a string that becomes a class instance, or one which goes a step beyond and represents the data internally in some fashion, presenting a "view" of that data on the outside of a different form.

在大多数情况下根本没有！ “自定义”集合的最常见用例是将传入值验证或编组为新表单的字符串，例如变为类实例的字符串，或以某种方式内部表示数据的步骤，在不同形式的外部呈现该数据的“视图”。

For the first use case, the [orm.validates()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapped_attributes.html" \l "sqlalchemy.orm.validates" \o "sqlalchemy.orm.validates) decorator is by far the simplest way to intercept incoming values in all cases for the purposes of validation and simple marshaling. See [Simple Validators](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapped_attributes.html" \l "simple-validators) for an example of this.

对于第一个用例，[orm.validates()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapped_attributes.html" \l "sqlalchemy.orm.validates" \o "sqlalchemy.orm.validates)装饰器是迄今为止，为了进行验证和简单的封送处理，在所有情况下拦截输入值的最简单的方法。请参阅简单验证器，以作为示例。

For the second use case, the [Association Proxy](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html) extension is a well-tested, widely used system that provides a read/write "view" of a collection in terms of some attribute present on the target object. As the target attribute can be a @property that returns virtually anything, a wide array of "alternative" views of a collection can be constructed with just a few functions. This approach leaves the underlying mapped collection unaffected and avoids the need to carefully tailor collection behavior on a method-by-method basis.

对于第二个用例，关联代理扩展是一个经过良好测试的广泛使用的系统，根据目标对象上存在的某些属性提供对集合的读/写“视图”。由于目标属性可以是返回几乎任何东西的@property，所以可以使用一些功能构建集合的“替代”视图。这种方法使底层映射集合不受影响，并避免在逐个方法的基础上仔细地定制集合行为的需要。

Customized collections are useful when the collection needs to have special behaviors upon access or mutation operations that can't otherwise be modeled externally to the collection. They can of course be combined with the above two approaches.

当收集需要在访问或变异操作时具有特殊行为(无法在集合外部进行建模的情况下)时，定制集合很有用。他们当然可以和上述两种方法相结合。

Collections in SQLAlchemy are transparently *instrumented*. Instrumentation means that normal operations on the collection are tracked and result in changes being written to the database at flush time. Additionally, collection operations can fire *events* which indicate some secondary operation must take place. Examples of a secondary operation include saving the child item in the parent's [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) (i.e. the save-update cascade), as well as synchronizing the state of a bi-directional relationship (i.e. a [backref()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.backref" \o "sqlalchemy.orm.backref)).

SQLAlchemy中的集合是透明的。 仪器意味着对集合的正常操作进行跟踪，并导致更改在冲洗时写入数据库。 此外，收集操作可以触发指示必须进行一些二次操作的事件。 次要操作的示例包括将子项目保存在父[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)(即save-update级联)中，以及同步双向关系的状态(即，[backref()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.backref" \o "sqlalchemy.orm.backref))。

The collections package understands the basic interface of lists, sets and dicts and will automatically apply instrumentation to those built-in types and their subclasses. Object-derived types that implement a basic collection interface are detected and instrumented via duck-typing:

集合包了解列表，集合和列表的基本界面，并将自动将工具应用于内置类型及其子类。 通过鸭子类型和检测实现基本收集界面的对象派生类型：

**class** **ListLike**(object):

**def** \_\_init\_\_(self):

self.data = []

**def** append(self, item):

self.data.append(item)

**def** remove(self, item):

self.data.remove(item)

**def** extend(self, items):

self.data.extend(items)

**def** \_\_iter\_\_(self):

**return** iter(self.data)

**def** foo(self):

**return** 'foo'

append, remove, and extend are known list-like methods, and will be instrumented automatically. \_\_iter\_\_ is not a mutator method and won't be instrumented, and foo won't be either.

Append，remove和extend是已知的类似列表的方法，并将自动进行检测。 \_\_iter\_\_不是一个mutator方法，不会被检测，foo也不会。

Duck-typing (i.e. guesswork) isn't rock-solid, of course, so you can be explicit about the interface you are implementing by providing an \_\_emulates\_\_ class attribute:

鸭子类型(即猜测)并不是坚如磐石的，所以你可以通过提供一个\_\_emulate\_\_类属性来明确你正在实现的界面：

**class** **SetLike**(object):

\_\_emulates\_\_ = set

**def** \_\_init\_\_(self):

self.data = set()

**def** append(self, item):

self.data.add(item)

**def** remove(self, item):

self.data.remove(item)

**def** \_\_iter\_\_(self):

**return** iter(self.data)

This class looks list-like because of append, but \_\_emulates\_\_ forces it to set-like. remove is known to be part of the set interface and will be instrumented.

这个类似于列表，因为附加，但\_\_emulate\_\_强制它设置。 已知删除是设置界面的一部分，将被检测。

But this class won't work quite yet: a little glue is needed to adapt it for use by SQLAlchemy. The ORM needs to know which methods to use to append, remove and iterate over members of the collection. When using a type like list or set, the appropriate methods are well-known and used automatically when present. This set-like class does not provide the expected add method, so we must supply an explicit mapping for the ORM via a decorator.

但是这个类别还不行：需要一些胶水来适应SQLAlchemy的使用。 ORM需要知道使用哪些方法来附加，删除和迭代集合的成员。 当使用像列表或集合的类型时，适当的方法是众所周知的，当存在时自动使用。 这个类集合类不提供预期的add方法，所以我们必须通过装饰器为ORM提供一个显式映射。

### Annotating Custom Collections via Decorators

Decorators can be used to tag the individual methods the ORM needs to manage collections. Use them when your class doesn't quite meet the regular interface for its container type, or when you otherwise would like to use a different method to get the job done.

装饰器可用于标记ORM管理集合所需的各种方法。 当您的类不完全满足其容器类型的常规接口时，或者当您以其他方式使用不同的方法来完成工作时，请使用它们。

**from** **sqlalchemy.orm.collections** **import** collection

**class** **SetLike**(object):

\_\_emulates\_\_ = set

**def** \_\_init\_\_(self):

self.data = set()

**@collection.appender**

**def** append(self, item):

self.data.add(item)

**def** remove(self, item):

self.data.remove(item)

**def** \_\_iter\_\_(self):

**return** iter(self.data)

And that's all that's needed to complete the example. SQLAlchemy will add instances via the append method. remove and \_\_iter\_\_ are the default methods for sets and will be used for removing and iteration. Default methods can be changed as well:

这就是完成这个例子所需要的。 SQLAlchemy将通过append方法添加实例。 remove和\_\_iter\_\_是集合的默认方法，将用于删除和迭代。 默认方法也可以改变：

**from** **sqlalchemy.orm.collections** **import** collection

**class** **MyList**(list):

**@collection.remover**

**def** zark(self, item):

*# do something special...*

**@collection.iterator**

**def** hey\_use\_this\_instead\_for\_iteration(self):

*# ...*

There is no requirement to be list-, or set-like at all. Collection classes can be any shape, so long as they have the append, remove and iterate interface marked for SQLAlchemy's use. Append and remove methods will be called with a mapped entity as the single argument, and iterator methods are called with no arguments and must return an iterator.

根本没有要求列表或设置样式。 集合类可以是任何形状，只要它们具有附加，删除和迭代标记为SQLAlchemy使用的接口。 将使用映射实体调用附加和删除方法作为单个参数，并且不使用参数调用迭代器方法，并且必须返回迭代器。

*class*sqlalchemy.orm.collections.**collection**

Decorators for entity collection classes.

实体集合类的装饰器。

The decorators fall into two groups: annotations and interception recipes.

The annotating decorators (appender, remover, iterator, linker, converter, internally\_instrumented) indicate the method's purpose and take no arguments. They are not written with parens:

装饰者分为两组：注释和截取食谱。

注释装饰器(appender，remover，iterator，linker，converter，internal\_instrumented)表示方法的目的，不会引用任何参数。 它们不是用括号写的：

**@collection**.appender**def** append(self, append): ...

The recipe decorators all require parens, even those that take no arguments:

**@collection**.adds('entity')**def** insert(self, position, entity): ...

**@collection**.removes\_return()**def** popitem(self): ...

*static***adds**(*arg*)

Mark the method as adding an entity to the collection.

将方法标记为向集合添加实体。

Adds "add to collection" handling to the method. The decorator argument indicates which method argument holds the SQLAlchemy-relevant value. Arguments can be specified positionally (i.e. integer) or by name:

添加“添加到集合”处理方法。 decorator参数指示哪个方法参数保存SQLAlchemy相关值。 参数可以在位置(即整数)或名称指定：

**@collection**.adds(1)**def** push(self, item): ...

**@collection**.adds('entity')**def** do\_stuff(self, thing, entity=**None**): ...

*static***appender**(*fn*)

Tag the method as the collection appender.

将方法标记为集合追加器。

The appender method is called with one positional argument: the value to append. The method will be automatically decorated with 'adds(1)' if not already decorated:

使用一个位置参数调用appender方法：要追加的值。 如果尚未装饰，该方法将自动用'adds(1)'进行装饰：

**@collection**.appender

**def** add(self, append): ...

*# or, equivalently*

**@collection**.appender

**@collection**.adds(1)

**def** add(self, append): ...

*# for mapping type, an 'append' may kick out a previous value# that occupies that slot. consider d['a'] = 'foo'- any previous# value in d['a'] is discarded.*

**@collection**.appender

**@collection**.replaces(1)

**def** add(self, entity):

key = some\_key\_func(entity)

previous = **None**

**if** key **in** self:

previous = self[key]

self[key] = entity

**return** previous

If the value to append is not allowed in the collection, you may raise an exception. Something to remember is that the appender will be called for each object mapped by a database query. If the database contains rows that violate your collection semantics, you will need to get creative to fix the problem, as access via the collection will not work.

如果收集中不允许附加值，则可能引发异常。 需要记住的是，将为由数据库查询映射的每个对象调用appender。 如果数据库包含违反收集语义的行，则需要获得创造性来解决问题，因为通过集合的访问将无法正常工作。

If the appender method is internally instrumented, you must also receive the keyword argument '\_sa\_initiator' and ensure its promulgation to collection events.

如果appender方法是内部检测的，您还必须收到关键字参数“\_sa\_initiator”，并确保其发布收集事件。

*static***converter**(*fn*)

Tag the method as the collection converter.

将方法标记为集合转换器。

This optional method will be called when a collection is being replaced entirely, as in:

当完全替换集合时，将调用此可选方法，如：

myobj.acollection = [newvalue1, newvalue2]

The converter method will receive the object being assigned and should return an iterable of values suitable for use by the appender method. A converter must not assign values or mutate the collection, its sole job is to adapt the value the user provides into an iterable of values for the ORM's use.

转换器方法将接收被分配的对象，并且应该返回适合于appender方法使用的值的可迭代。 转换器不能分配值或对集合进行变异，其唯一的工作就是使用户提供的值适用于ORM使用的值的迭代。

The default converter implementation will use duck-typing to do the conversion. A dict-like collection will be convert into an iterable of dictionary values, and other types will simply be iterated:

默认的转换器实现将使用鸭型进行转换。 类似dict的集合将被转换成可迭代的字典值，其他类型将被简单地迭代：

**@collection**.converter

**def** convert(self, other): ...

If the duck-typing of the object does not match the type of this collection, a TypeError is raised.

如果对象的鸭子类型与此集合的类型不匹配，则会引发TypeError。

Supply an implementation of this method if you want to expand the range of possible types that can be assigned in bulk or perform validation on the values about to be assigned.

如果要扩展可批量分配的可能类型的范围，或对要分配的值执行验证，请提供此方法的实现。

*static***internally\_instrumented**(*fn*)

Tag the method as instrumented.

将方法标记为仪器化。

This tag will prevent any decoration from being applied to the method. Use this if you are orchestrating your own calls to collection\_adapter() in one of the basic SQLAlchemy interface methods, or to prevent an automatic ABC method decoration from wrapping your implementation:

该标签将防止任何装饰被应用于该方法。 如果您在一个基本的SQLAlchemy接口方法中编制对collection\_adapter()的调用，或者阻止自动ABC方法装饰包装您的实现，请使用此方法：

*# normally an 'extend' method on a list-like class would be# automatically intercepted and re-implemented in terms of# SQLAlchemy events and append(). your implementation will# never be called, unless:*

**@collection**.internally\_instrumented

**def** extend(self, items): ...

*static***iterator**(*fn*)

Tag the method as the collection remover.

将方法标记为收集清除器。

The iterator method is called with no arguments. It is expected to return an iterator over all collection members:

调用iterator方法没有参数。 期望在所有收集成员上返回迭代器：

**@collection**.iterator

**def** \_\_iter\_\_(self): ...

*static***link**(*fn*)

deprecated; synonym for [collection.linker()](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "sqlalchemy.orm.collections.collection.linker" \o "sqlalchemy.orm.collections.collection.linker).

*static***linker**(*fn*)

Tag the method as a "linked to attribute" event handler.

将方法标记为“链接到属性”事件处理程序。

This optional event handler will be called when the collection class is linked to or unlinked from the InstrumentedAttribute. It is invoked immediately after the '\_sa\_adapter' property is set on the instance. A single argument is passed: the collection adapter that has been linked, or None if unlinking.

当将集合类链接到或取消与InstrumentedAttribute链接时，将调用此可选事件处理程序。 在实例上设置“\_sa\_adapter”属性后立即调用它。 传递一个参数：已链接的集合适配器，如果取消链接，则为None。

*Deprecated since version 1.0.0:*- the [collection.linker()](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "sqlalchemy.orm.collections.collection.linker" \o "sqlalchemy.orm.collections.collection.linker) handler is superseded by the [AttributeEvents.init\_collection()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.AttributeEvents.init_collection" \o "sqlalchemy.orm.events.AttributeEvents.init_collection) and [AttributeEvents.dispose\_collection()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.AttributeEvents.dispose_collection" \o "sqlalchemy.orm.events.AttributeEvents.dispose_collection) handlers.

*static***remover**(*fn*)

Tag the method as the collection remover.

将方法标记为收集清除器。

The remover method is called with one positional argument: the value to remove. The method will be automatically decorated with [removes\_return()](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "sqlalchemy.orm.collections.collection.removes_return" \o "sqlalchemy.orm.collections.collection.removes_return)if not already decorated:

使用一个位置参数调用删除方法：要删除的值。 如果尚未装饰，该方法将自动使用removed\_return()进行装饰：

**@collection**.remover

**def** zap(self, entity): ...

*# or, equivalently*

**@collection**.remover**@collection**.removes\_return()

**def** zap(self, ): ...

If the value to remove is not present in the collection, you may raise an exception or return None to ignore the error.

如果要删除的值不存在于集合中，则可能引发异常或返回None以忽略该错误。

If the remove method is internally instrumented, you must also receive the keyword argument '\_sa\_initiator' and ensure its promulgation to collection events.

如果删除方法是内部检测的，您还必须收到关键字参数“\_sa\_initiator”，并确保其发布收集事件。

*static***removes**(*arg*)

Mark the method as removing an entity in the collection.

将方法标记为删除集合中的实体。

Adds "remove from collection" handling to the method. The decorator argument indicates which method argument holds the SQLAlchemy-relevant value to be removed. Arguments can be specified positionally (i.e. integer) or by name:

在方法中添加“从集合中删除”处理。 decorator参数指示哪个方法参数保存要删除的SQLAlchemy相关值。 参数可以在位置(即整数)或名称指定：

**@collection**.removes(1)

**def** zap(self, item): ...

For methods where the value to remove is not known at call-time, use collection.removes\_return.

对于在调用时不知道要删除的值的方法，请使用collection.removes\_return。

*static***removes\_return**()

Mark the method as removing an entity in the collection.

将方法标记为删除集合中的实体。

Adds "remove from collection" handling to the method. The return value of the method, if any, is considered the value to remove. The method arguments are not inspected:

在方法中添加“从集合中删除”处理。 方法的返回值(如果有)被认为是要删除的值。 不检查方法参数：

**@collection**.removes\_return()

**def** pop(self): ...

For methods where the value to remove is known at call-time, use collection.remove.

对于在调用时已知值要删除的方法，请使用collection.remove。

*static***replaces**(*arg*)

Mark the method as replacing an entity in the collection.

将方法标记为替换集合中的实体。

Adds "add to collection" and "remove from collection" handling to the method. The decorator argument indicates which method argument holds the SQLAlchemy-relevant value to be added, and return value, if any will be considered the value to remove.

添加“添加到集合”和“从集合中删除”处理方法。 decorator参数指示哪个方法参数持有要添加的SQLAlchemy相关值，并返回值(如果有的话将被视为要删除的值)。

Arguments can be specified positionally (i.e. integer) or by name:

参数可以在位置(即整数)或名称指定：

**@collection**.replaces(2)

**def** \_\_setitem\_\_(self, index, item): ...

### Custom Dictionary-Based Collections

The [MappedCollection](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "sqlalchemy.orm.collections.MappedCollection" \o "sqlalchemy.orm.collections.MappedCollection) class can be used as a base class for your custom types or as a mix-in to quickly add dict collection support to other classes. It uses a keying function to delegate to \_\_setitem\_\_ and \_\_delitem\_\_:

[MappedCollection](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "sqlalchemy.orm.collections.MappedCollection" \o "sqlalchemy.orm.collections.MappedCollection)类可以用作您的自定义类型的基类，也可以作为混合，以便将dict收藏支持快速添加到其他类。 它使用键控函数来委派\_\_setitem\_\_和\_\_delitem\_\_：

**from** **sqlalchemy.util** **import** OrderedDict

**from** **sqlalchemy.orm.collections** **import** MappedCollection

**class** **NodeMap**(OrderedDict, MappedCollection):

*"""Holds 'Node' objects, keyed by the 'name' attribute with insert order maintained."""*

**def** \_\_init\_\_(self, \*args, \*\*kw):

MappedCollection.\_\_init\_\_(self, keyfunc=**lambda** node: node.name)

OrderedDict.\_\_init\_\_(self, \*args, \*\*kw)

When subclassing [MappedCollection](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "sqlalchemy.orm.collections.MappedCollection" \o "sqlalchemy.orm.collections.MappedCollection), user-defined versions of \_\_setitem\_\_() or \_\_delitem\_\_() should be decorated with [collection.internally\_instrumented()](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "sqlalchemy.orm.collections.collection.internally_instrumented" \o "sqlalchemy.orm.collections.collection.internally_instrumented), ****if**** they call down to those same methods on [MappedCollection](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "sqlalchemy.orm.collections.MappedCollection" \o "sqlalchemy.orm.collections.MappedCollection). This because the methods on [MappedCollection](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "sqlalchemy.orm.collections.MappedCollection" \o "sqlalchemy.orm.collections.MappedCollection) are already instrumented - calling them from within an already instrumented call can cause events to be fired off repeatedly, or inappropriately, leading to internal state corruption in rare cases:

当对[MappedCollection](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "sqlalchemy.orm.collections.MappedCollection" \o "sqlalchemy.orm.collections.MappedCollection)进行子类化时，如果用户定义的\_\_setitem\_\_()或\_\_delitem\_\_()的版本在使用[MappedCollection](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "sqlalchemy.orm.collections.MappedCollection" \o "sqlalchemy.orm.collections.MappedCollection)调用这些方法时，则应使用[collection.internally\_instrumented()](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "sqlalchemy.orm.collections.collection.internally_instrumented" \o "sqlalchemy.orm.collections.collection.internally_instrumented)进行装饰。 这是因为[MappedCollection](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "sqlalchemy.orm.collections.MappedCollection" \o "sqlalchemy.orm.collections.MappedCollection)中的方法已经被调用 - 在已经被调用的调用中调用它们可能会导致事件多次被触发或不适当地导致在极少数情况下导致内部状态损坏：

**from** **sqlalchemy.orm.collections** **import** MappedCollection,\

collection

**class** **MyMappedCollection**(MappedCollection):

*"""Use @internally\_instrumented when your methods call down to already-instrumented methods.*

*"""*

**@collection**.internally\_instrumented

**def** \_\_setitem\_\_(self, key, value, \_sa\_initiator=**None**):

*# do something with key, value*

super(MyMappedCollection, self).\_\_setitem\_\_(key, value, \_sa\_initiator)

**@collection**.internally\_instrumented

**def** \_\_delitem\_\_(self, key, \_sa\_initiator=**None**):

*# do something with key*

super(MyMappedCollection, self).\_\_delitem\_\_(key, \_sa\_initiator)

The ORM understands the dict interface just like lists and sets, and will automatically instrument all dict-like methods if you choose to subclass dict or provide dict-like collection behavior in a duck-typed class. You must decorate appender and remover methods, however- there are no compatible methods in the basic dictionary interface for SQLAlchemy to use by default. Iteration will go through itervalues() unless otherwise decorated.

**Note**

Due to a bug in MappedCollection prior to version 0.7.6, this workaround usually needs to be called before a custom subclass of [MappedCollection](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "sqlalchemy.orm.collections.MappedCollection" \o "sqlalchemy.orm.collections.MappedCollection) which uses [collection.internally\_instrumented()](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "sqlalchemy.orm.collections.collection.internally_instrumented" \o "sqlalchemy.orm.collections.collection.internally_instrumented) can be used:

**from** **sqlalchemy.orm.collections** **import** \_instrument\_class, MappedCollection\_instrument\_class(MappedCollection)

This will ensure that the [MappedCollection](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "sqlalchemy.orm.collections.MappedCollection" \o "sqlalchemy.orm.collections.MappedCollection) has been properly initialized with custom \_\_setitem\_\_() and \_\_delitem\_\_() methods before used in a custom subclass.

*class*sqlalchemy.orm.collections.**MappedCollection**(*keyfunc*)

Bases: \_\_builtin\_\_.dict

A basic dictionary-based collection class.

Extends dict with the minimal bag semantics that collection classes require. set and remove are implemented in terms of a keying function: any callable that takes an object and returns an object for use as a dictionary key.

**\_\_init\_\_**(*keyfunc*)

Create a new collection with keying provided by keyfunc.

keyfunc may be any callable that takes an object and returns an object for use as a dictionary key.

The keyfunc will be called every time the ORM needs to add a member by value-only (such as when loading instances from the database) or remove a member. The usual cautions about dictionary keying apply- keyfunc(object) should return the same output for the life of the collection. Keying based on mutable properties can result in unreachable instances "lost" in the collection.

**clear**() → None. Remove all items from D.

**pop**(*k*[, *d*]) → v, remove specified key and return the corresponding value.

If key is not found, d is returned if given, otherwise KeyError is raised

**popitem**() → (k, v), remove and return some (key, value) pair as a

2-tuple; but raise KeyError if D is empty.

**remove**(*value*, *\_sa\_initiator=None*)

Remove an item by value, consulting the keyfunc for the key.

**set**(*value*, *\_sa\_initiator=None*)

Add an item by value, consulting the keyfunc for the key.

**setdefault**(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

**update**([*E*, ]*\*\*F*) → None. Update D from dict/iterable E and F.

If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

### Instrumentation and Custom Types

Many custom types and existing library classes can be used as a entity collection type as-is without further ado. However, it is important to note that the instrumentation process will modify the type, adding decorators around methods automatically.

The decorations are lightweight and no-op outside of relationships, but they do add unneeded overhead when triggered elsewhere. When using a library class as a collection, it can be good practice to use the "trivial subclass" trick to restrict the decorations to just your usage in relationships. For example:

**class** **MyAwesomeList**(some.great.library.AwesomeList):

**pass**

*# ... relationship(..., collection\_class=MyAwesomeList)*

The ORM uses this approach for built-ins, quietly substituting a trivial subclass when a list, set or dict is used directly.

## Collection Internals

Various internal methods.

sqlalchemy.orm.collections.**bulk\_replace**(*values*, *existing\_adapter*, *new\_adapter*)

Load a new collection, firing events based on prior like membership.

Appends instances in values onto the new\_adapter. Events will be fired for any instance not present in the existing\_adapter. Any instances in existing\_adapter not present in values will have remove events fired upon them.

|  |  |
| --- | --- |
| **Parameters:** | * ****values**** – An iterable of collection member instances * ****existing\_adapter**** – A [CollectionAdapter](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "sqlalchemy.orm.collections.CollectionAdapter" \o "sqlalchemy.orm.collections.CollectionAdapter) of instances to be replaced * ****new\_adapter**** – An empty [CollectionAdapter](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "sqlalchemy.orm.collections.CollectionAdapter" \o "sqlalchemy.orm.collections.CollectionAdapter) to load with values |

*class*sqlalchemy.orm.collections.**collection**

Decorators for entity collection classes.

The decorators fall into two groups: annotations and interception recipes.

The annotating decorators (appender, remover, iterator, linker, converter, internally\_instrumented) indicate the method's purpose and take no arguments. They are not written with parens:

**@collection**.appender**def** append(self, append): ...

The recipe decorators all require parens, even those that take no arguments:

**@collection**.adds('entity')**def** insert(self, position, entity): ...

**@collection**.removes\_return()**def** popitem(self): ...

sqlalchemy.orm.collections.**collection\_adapter***= <operator.attrgetter object>*

Fetch the [CollectionAdapter](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "sqlalchemy.orm.collections.CollectionAdapter" \o "sqlalchemy.orm.collections.CollectionAdapter) for a collection.

*class*sqlalchemy.orm.collections.**CollectionAdapter**(*attr*, *owner\_state*, *data*)

Bridges between the ORM and arbitrary Python collections.

Proxies base-level collection operations (append, remove, iterate) to the underlying Python collection, and emits add/remove events for entities entering or leaving the collection.

The ORM uses [CollectionAdapter](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "sqlalchemy.orm.collections.CollectionAdapter" \o "sqlalchemy.orm.collections.CollectionAdapter) exclusively for interaction with entity collections.

*class*sqlalchemy.orm.collections.**InstrumentedDict**

Bases: \_\_builtin\_\_.dict

An instrumented version of the built-in dict.

*class*sqlalchemy.orm.collections.**InstrumentedList**

Bases: \_\_builtin\_\_.list

An instrumented version of the built-in list.

*class*sqlalchemy.orm.collections.**InstrumentedSet**

Bases: \_\_builtin\_\_.set

An instrumented version of the built-in set.

sqlalchemy.orm.collections.**prepare\_instrumentation**(*factory*)

Prepare a callable for future use as a collection class factory.

Given a collection class factory (either a type or no-arg callable), return another factory that will produce compatible instances when called.

This function is responsible for converting collection\_class=list into the run-time behavior of collection\_class=InstrumentedList.

# **3.6 Special Relationship Persistence Patterns**

# **3.6.1 Rows that point to themselves / Mutually Dependent Rows**

This is a very specific case where relationship() must perform an INSERT and a second UPDATE in order to properly populate a row (and vice versa an UPDATE and DELETE in order to delete without violating foreign key constraints). The two use cases are:

这是一个非常具体的情况，其中relationship()必须执行INSERT和第二个UPDATE，才能正确填充一行(反之亦然UPDATE和DELETE以便删除而不违反外键约束)。 这两个用例是：

* A table contains a foreign key to itself, and a single row will have a foreign key value pointing to its own primary key.
* 表中包含一个自己的外键，单个行将具有指向其主键的外键值。
* Two tables each contain a foreign key referencing the other table, with a row in each table referencing the other.
* 两个表都包含引用另一个表的外键，每个表中的一行引用另一个表。

For example:

User

user\_id name related\_user\_id

1 'ed' 1

Or:

widget entry

------------------------------------------- ---------------------------------

widget\_id name favorite\_entry\_id entry\_id name widget\_id

1 'somewidget' 5 5 'someentry' 1

In the first case, a row points to itself. Technically, a database that uses sequences such as PostgreSQL or Oracle can INSERT the row at once using a previously generated value, but databases which rely upon autoincrement-style primary key identifiers cannot. The [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) always assumes a "parent/child" model of row population during flush, so unless you are populating the primary key/foreign key columns directly, [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) needs to use two statements.

在第一种情况下，一行指向自己。从技术上讲，使用诸如PostgreSQL或Oracle之类的序列的数据库可以使用以前生成的值一次性地插入该行，但依赖于自动增量式主键标识符的数据库不能。 [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)总是在flush期间假定一个"父/子"行的行数，所以除非你直接填充主键/外键列，否则[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)需要使用两个语句。

In the second case, the "widget" row must be inserted before any referring "entry" rows, but then the "favorite\_entry\_id" column of that "widget" row cannot be set until the "entry" rows have been generated. In this case, it's typically impossible to insert the "widget" and "entry" rows using just two INSERT statements; an UPDATE must be performed in order to keep foreign key constraints fulfilled. The exception is if the foreign keys are configured as "deferred until commit" (a feature some databases support) and if the identifiers were populated manually (again essentially bypassing [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)).

在第二种情况下，必须在任何引用"条目"行之前插入"窗口小部件"行，但是直到生成"条目"行才能设置该"窗口小部件"行的"favorite\_entry\_id"列。在这种情况下，通常只能使用两个INSERT语句插入"widget"和"entry"行;必须执行UPDATE才能保持外键约束。例外情况是如果外键配置为"延迟到提交"(某些数据库支持的功能)以及手动填充标识符(再次基本上绕过了relationship())。

To enable the usage of a supplementary UPDATE statement, we use the [post\_update](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.post_update" \o "sqlalchemy.orm.relationship) option of [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship). This specifies that the linkage between the two rows should be created using an UPDATE statement after both rows have been INSERTED; it also causes the rows to be de-associated with each other via UPDATE before a DELETE is emitted. The flag should be placed on just *one* of the relationships, preferably the many-to-one side. Below we illustrate a complete example, including two [ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey) constructs:

为了使用补充UPDATE语句，我们使用了关系()的post\_update选项。这指定两行之间的链接应在两行都被INSERTED之后使用UPDATE语句创建;它也会导致在发出DELETE之前通过UPDATE将行取消关联。国旗应该只是其中一个关系，最好是多对一的。下面我们举一个完整的例子，包括两个ForeignKey结构：

**from** **sqlalchemy** **import** Integer, ForeignKey, Column

**from** **sqlalchemy.ext.declarative** **import** declarative\_base

**from** **sqlalchemy.orm** **import** relationship

Base = declarative\_base()

**class** **Entry**(Base):

\_\_tablename\_\_ = 'entry'

entry\_id = Column(Integer, primary\_key=**True**)

widget\_id = Column(Integer, ForeignKey('widget.widget\_id'))

name = Column(String(50))

**class** **Widget**(Base):

\_\_tablename\_\_ = 'widget'

widget\_id = Column(Integer, primary\_key=**True**)

favorite\_entry\_id = Column(Integer,

ForeignKey('entry.entry\_id',

name="fk\_favorite\_entry"))

name = Column(String(50))

entries = relationship(Entry, primaryjoin=

widget\_id==Entry.widget\_id)

favorite\_entry = relationship(Entry,

primaryjoin=

favorite\_entry\_id==Entry.entry\_id,

post\_update=**True**)

When a structure against the above configuration is flushed, the "widget" row will be INSERTed minus the "favorite\_entry\_id" value, then all the "entry" rows will be INSERTed referencing the parent "widget" row, and then an UPDATE statement will populate the "favorite\_entry\_id" column of the "widget" table (it's one row at a time for the time being):

当针对上述配置的结构被刷新时，"窗口小部件"行将被INSERT减去"favorite\_entry\_id"值，则所有"条目"行将被引用父"widget"行，然后UPDATE语句将填充 "widget"表的"favorite\_entry\_id"列(暂时一次是一行)：

**>>>** w1 = Widget(name='somewidget')

**>>>** e1 = Entry(name='someentry')

**>>>** w1.favorite\_entry = e1

**>>>** w1.entries = [e1]

**>>>** session.add\_all([w1, e1])

**>>>** session.commit()

BEGIN (implicit)

INSERT INTO widget (favorite\_entry\_id, name) VALUES (?, ?)

(None, 'somewidget')

INSERT INTO entry (widget\_id, name) VALUES (?, ?)

(1, 'someentry')

UPDATE widget SET favorite\_entry\_id=? WHERE widget.widget\_id = ?

(1, 1)

COMMIT

An additional configuration we can specify is to supply a more comprehensive foreign key constraint on Widget, such that it's guaranteed that favorite\_entry\_id refers to an Entry that also refers to this Widget. We can use a composite foreign key, as illustrated below:

我们可以指定的另外一个配置是为Widget提供一个更全面的外键约束，这样保证了nice\_entry\_id是指一个也指向这个Widget的Entry。 我们可以使用复合外键，如下图所示：

**from** **sqlalchemy** **import** Integer, ForeignKey, String, \

Column, UniqueConstraint, ForeignKeyConstraint

**from** **sqlalchemy.ext.declarative** **import** declarative\_base

**from** **sqlalchemy.orm** **import** relationship

Base = declarative\_base()

**class** **Entry**(Base):

\_\_tablename\_\_ = 'entry'

entry\_id = Column(Integer, primary\_key=**True**)

widget\_id = Column(Integer, ForeignKey('widget.widget\_id'))

name = Column(String(50))

\_\_table\_args\_\_ = (

UniqueConstraint("entry\_id", "widget\_id"),

)

**class** **Widget**(Base):

\_\_tablename\_\_ = 'widget'

widget\_id = Column(Integer, autoincrement='ignore\_fk', primary\_key=**True**)

favorite\_entry\_id = Column(Integer)

name = Column(String(50))

\_\_table\_args\_\_ = (

ForeignKeyConstraint(

["widget\_id", "favorite\_entry\_id"],

["entry.widget\_id", "entry.entry\_id"],

name="fk\_favorite\_entry"

),

)

entries = relationship(Entry, primaryjoin=

widget\_id==Entry.widget\_id,

foreign\_keys=Entry.widget\_id)

favorite\_entry = relationship(Entry,

primaryjoin=

favorite\_entry\_id==Entry.entry\_id,

foreign\_keys=favorite\_entry\_id,

post\_update=**True**)

The above mapping features a composite [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint) bridging the widget\_id and favorite\_entry\_id columns. To ensure that Widget.widget\_id remains an "autoincrementing" column we specify [autoincrement](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column.params.autoincrement" \o "sqlalchemy.schema.Column) to the value "ignore\_fk" on [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column), and additionally on each[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) we must limit those columns considered as part of the foreign key for the purposes of joining and cross-population.

上述映射的特征是一个复合的[ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint)，用于桥接widget\_id和favorite\_entry\_id列。 为了确保Widget.widget\_id保持为"自动增量"列，我们在[Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column)上指定[autoincrement](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column.params.autoincrement" \o "sqlalchemy.schema.Column)为"ignore\_fk"，另外在每个[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)上，我们必须限制被认为是外键的一部分的列， 人口。

Mutable Primary Keys / Update Cascades

When the primary key of an entity changes, related items which reference the primary key must also be updated as well. For databases which enforce referential integrity, the best strategy is to use the database's ON UPDATE CASCADE functionality in order to propagate primary key changes to referenced foreign keys - the values cannot be out of sync for any moment unless the constraints are marked as "deferrable", that is, not enforced until the transaction completes.

当实体的主键更改时，引用主键的相关项也必须更新。 对于强制引用完整性的数据库，最佳策略是使用数据库的ON UPDATE CASCADE功能，以便将主键更改传播到引用的外键 - 除非将约束标记为"可延迟"，否则值不能同步， ，即在事务完成之前不强制执行。

It is ****highly recommended**** that an application which seeks to employ natural primary keys with mutable values to use the ON UPDATE CASCADE capabilities of the database. An example mapping which illustrates this is:

强烈建议使用可变值使用自然主键的应用程序使用数据库的ON UPDATE CASCADE功能。 说明这一点的示例映射是：

**class** **User**(Base):

\_\_tablename\_\_ = 'user'

\_\_table\_args\_\_ = {'mysql\_engine': 'InnoDB'}

username = Column(String(50), primary\_key=**True**)

fullname = Column(String(100))

addresses = relationship("Address")

**class** **Address**(Base):

\_\_tablename\_\_ = 'address'

\_\_table\_args\_\_ = {'mysql\_engine': 'InnoDB'}

email = Column(String(50), primary\_key=**True**)

username = Column(String(50),

ForeignKey('user.username', onupdate="cascade")

)

Above, we illustrate onupdate="cascade" on the [ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey) object, and we also illustrate the mysql\_engine='InnoDB' setting which, on a MySQL backend, ensures that the InnoDB engine supporting referential integrity is used. When using SQLite, referential integrity should be enabled, using the configuration described at [Foreign Key Support](http://docs.sqlalchemy.org/en/rel_1_1/dialects/sqlite.html" \l "sqlite-foreign-keys).

以上，我们在[ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey)对象上说明了onupdate="cascade"，我们还说明了mysql\_engine='InnoDB'设置，在MySQL后端可以确保使用支持引用完整性的InnoDB引擎。 使用SQLite时，应使用[Foreign Key Support](http://docs.sqlalchemy.org/en/rel_1_1/dialects/sqlite.html" \l "sqlite-foreign-keys)中描述的配置启用引用完整性。

**See also**

[Using Passive Deletes](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "passive-deletes) - supporting ON DELETE CASCADE with relationships

[orm.mapper.passive\_updates](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.params.passive_updates" \o "sqlalchemy.orm.mapper) - similar feature on [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper)

### Simulating limited ON UPDATE CASCADE without foreign key support

In those cases when a database that does not support referential integrity is used, and natural primary keys with mutable values are in play, SQLAlchemy offers a feature in order to allow propagation of primary key values to already-referenced foreign keys to a ****limited**** extent, by emitting an UPDATE statement against foreign key columns that immediately reference a primary key column whose value has changed. The primary platforms without referential integrity features are MySQL when the MyISAM storage engine is used, and SQLite when the PRAGMA foreign\_keys=ON pragma is not used. The Oracle database also has no support for ON UPDATECASCADE, but because it still enforces referential integrity, needs constraints to be marked as deferrable so that SQLAlchemy can emit UPDATE statements.

在使用不支持参照完整性的数据库的情况下，使用具有可变值的自然主键时，SQLAlchemy提供了一个功能，以便允许主键值在有限的程度上传播到已引用的外键，通过向外键列发出UPDATE语句，该列将立即引用其值已更改的主键列。没有引用完整性功能的主要平台是使用MyISAM存储引擎时的MySQL，而不使用PRAGMA foreign\_keys=ONpragma时的SQLite。 Oracle数据库也不支持ON UPDATECASCADE，但是由于它仍然强制引用完整性，所以需要将约束标记为可延迟，以便SQLAlchemy可以发出UPDATE语句。

The feature is enabled by setting the [passive\_updates](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.passive_updates" \o "sqlalchemy.orm.relationship) flag to False, most preferably on a one-to-many or many-to-many [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship). When "updates" are no longer "passive" this indicates that SQLAlchemy will issue UPDATE statements individually for objects referenced in the collection referred to by the parent object with a changing primary key value. This also implies that collections will be fully loaded into memory if not already locally present.

通过将passive\_updates标志设置为False，最优选地在一对多或多对多[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)之间启用该功能。当"更新"不再是"被动"时，这表示SQLAlchemy会针对由父对象引用的集合引用的对象以更改的主键值单独发出UPDATE语句。这也意味着如果不是本地存在的话，这些集合将被完全加载到内存中。

Our previous mapping using passive\_updates=False looks like:

**class** **User**(Base):

\_\_tablename\_\_ = 'user'

username = Column(String(50), primary\_key=**True**)

fullname = Column(String(100))

*# passive\_updates=False \*only\* needed if the database*

*# does not implement ON UPDATE CASCADE*

addresses = relationship("Address", passive\_updates=**False**)

**class** **Address**(Base):

\_\_tablename\_\_ = 'address'

email = Column(String(50), primary\_key=**True**)

username = Column(String(50), ForeignKey('user.username'))

Key limitations of passive\_updates=False include:

passive\_updates = False的主要限制包括：

* it performs much more poorly than direct database ON UPDATE CASCADE, because it needs to fully pre-load affected collections using SELECT and also must emit UPDATE statements against those values, which it will attempt to run in "batches" but still runs on a per-row basis at the DBAPI level.
* 比直接数据库ON UPDATE CASCADE执行的性能要差得多，因为它需要使用SELECT完全预加载受影响的集合，并且还必须针对这些值尝试发出UPDATE语句，这些值将尝试在"批处理"中运行，但仍然运行在 在DBAPI级别的每行基础。
* the feature cannot "cascade" more than one level. That is, if mapping X has a foreign key which refers to the primary key of mapping Y, but then mapping Y's primary key is itself a foreign key to mapping Z, passive\_updates=False cannot cascade a change in primary key value from Z to X.
* 该功能不能"级联"多个级别。 也就是说，如果映射X具有指向映射Y的主键的外键，但是随后映射Y的主键本身是映射Z的外键，passive\_updates=False不能将主键值的更改从Z级联到X。
* Configuring passive\_updates=False only on the many-to-one side of a relationship will not have a full effect, as the unit of work searches only through the current identity map for objects that may be referencing the one with a mutating primary key, not throughout the database.
* 只能在关系的多对一侧配置passive\_updates = False将不会有完整的效果，因为工作单元只能通过当前的身份映射搜索可能引用具有突变主键的对象的对象， 不是整个数据库。

As virtually all databases other than Oracle now support ON UPDATE CASCADE, it is highly recommended that traditional ON UPDATE CASCADE support be used in the case that natural and mutable primary key values are in use.

由于几乎所有Oracle以外的数据库现在都支持ON UPDATE CASCADE，因此强烈建议在使用自然和可变主键值的情况下使用传统的ON UPDATE CASCADE支持。

## 3.7 Relationships API

Functional constructs for ORM configuration.

See the SQLAlchemy object relational tutorial and mapper configuration documentation for an overview of how this module is used.

sqlalchemy.orm.**relationship**(*argument*, *secondary=None*, *primaryjoin=None*, *secondaryjoin=None*, *foreign\_keys=None*, *uselist=None*, *order\_by=False*, *backref=None*, *back\_populates=None*, *post\_update=False*, *cascade=False*, *extension=None*, *viewonly=False*, *lazy=True*, *collection\_class=None*, *passive\_deletes=False*, *passive\_updates=True*, *remote\_side=None*, *enable\_typechecks=True*, *join\_depth=None*, *comparator\_factory=None*, *single\_parent=False*, *innerjoin=False*, *distinct\_target\_key=None*, *doc=None*, *active\_history=False*, *cascade\_backrefs=True*, *load\_on\_pending=False*, *bake\_queries=True*, *\_local\_remote\_pairs=None*, *query\_class=None*, *info=None*)

Provide a relationship between two mapped classes.

This corresponds to a parent-child or associative table relationship. The constructed class is an instance of [RelationshipProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty" \o "sqlalchemy.orm.properties.RelationshipProperty).

A typical [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship), used in a classical mapping:

mapper(Parent, properties={

'children': relationship(Child)})

Some arguments accepted by [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) optionally accept a callable function, which when called produces the desired value. The callable is invoked by the parent [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) at "mapper initialization" time, which happens only when mappers are first used, and is assumed to be after all mappings have been constructed. This can be used to resolve order-of-declaration and other dependency issues, such as if Child is declared below Parent in the same file:

[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)接受的一些参数可以接受一个可调用的函数，当被调用时，它会产生所需的值。 可调用的方法是由父[Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper)在“映射初始化”时调用，只有在首次使用映射时才会发生，并且假定在所有映射都被构造之后。 这可以用于解决声明顺序和其他依赖性问题，例如，如果Child在同一文件中声明为Parent以下：

mapper(Parent, properties={

"children":relationship(**lambda**: Child,

order\_by=**lambda**: Child.id)})

When using the [Declarative](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/index.html) extension, the Declarative initializer allows string arguments to be passed to [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship). These string arguments are converted into callables that evaluate the string as Python code, using the Declarative class-registry as a namespace. This allows the lookup of related classes to be automatic via their string name, and removes the need to import related classes at all into the local module space:

当使用Declarative扩展时，声明式初始化器允许将字符串参数传递给[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)。 这些字符串参数被转换成可以使用Declarative类注册表作为命名空间来将字符串评估为Python代码的可调用函数。 这允许通过其字符串名称自动查找相关类，并且不需要将相关类导入到本地模块空间中：

**from** **sqlalchemy.ext.declarative** **import** declarative\_base

Base = declarative\_base()

**class** **Parent**(Base):

\_\_tablename\_\_ = 'parent'

id = Column(Integer, primary\_key=**True**)

children = relationship("Child", order\_by="Child.id")

**See also**

[Relationship Configuration](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationships.html) - Full introductory and reference documentation for [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship).

[Building a Relationship](http://docs.sqlalchemy.org/en/rel_1_1/orm/tutorial.html" \l "orm-tutorial-relationship) - ORM tutorial introduction.

|  |  |
| --- | --- |
| **Parameters:** | * ****argument –****a mapped class, or actual [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) instance, representing the target of the relationship.映射类或实际的Mapper实例，表示关系的目标。   [argument](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.argument" \o "sqlalchemy.orm.relationship) may also be passed as a callable function which is evaluated at mapper initialization time, and may be passed as a Python-evaluable string when using Declarative.  [argument](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.argument" \o "sqlalchemy.orm.relationship)也可以作为可调用函数传递，该函数在映射器初始化时被评估，并且可以在使用Declarative时作为Python可评估字符串传递。  **See also**  [Configuring Relationships](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/relationships.html" \l "declarative-configuring-relationships) - further detail on relationship configuration when using Declarative.   * ****secondary –****for a many-to-many relationship, specifies the intermediary table, and is typically an instance of [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table). In less common circumstances, the argument may also be specified as an [Alias](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Alias" \o "sqlalchemy.sql.expression.Alias) construct, or even a [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) construct.   [secondary](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.secondary" \o "sqlalchemy.orm.relationship) may also be passed as a callable function which is evaluated at mapper initialization time. When using Declarative, it may also be a string argument noting the name of a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) that is present in the [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) collection associated with the parent-mapped [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table).  The [secondary](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.secondary" \o "sqlalchemy.orm.relationship) keyword argument is typically applied in the case where the intermediary [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) is not otherwise expressed in any direct class mapping. If the "secondary" table is also explicitly mapped elsewhere (e.g. as in [Association Object](http://docs.sqlalchemy.org/en/rel_1_1/orm/basic_relationships.html" \l "association-pattern)), one should consider applying the [viewonly](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.viewonly" \o "sqlalchemy.orm.relationship) flag so that this[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) is not used for persistence operations which may conflict with those of the association object pattern.  **See also**  [Many To Many](http://docs.sqlalchemy.org/en/rel_1_1/orm/basic_relationships.html" \l "relationships-many-to-many) - Reference example of "many to many".  [Building a Many To Many Relationship](http://docs.sqlalchemy.org/en/rel_1_1/orm/tutorial.html" \l "orm-tutorial-many-to-many) - ORM tutorial introduction to many-to-many relationships.  [Self-Referential Many-to-Many Relationship](http://docs.sqlalchemy.org/en/rel_1_1/orm/join_conditions.html" \l "self-referential-many-to-many) - Specifics on using many-to-many in a self-referential case.  [Configuring Many-to-Many Relationships](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/relationships.html" \l "declarative-many-to-many) - Additional options when using Declarative.  [Association Object](http://docs.sqlalchemy.org/en/rel_1_1/orm/basic_relationships.html" \l "association-pattern) - an alternative to [secondary](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.secondary" \o "sqlalchemy.orm.relationship) when composing association table relationships, allowing additional attributes to be specified on the association table.  [Composite "Secondary" Joins](http://docs.sqlalchemy.org/en/rel_1_1/orm/join_conditions.html" \l "composite-secondary-join) - a lesser-used pattern which in some cases can enable complex [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) SQL conditions to be used.  *New in version 0.9.2:*[secondary](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.secondary" \o "sqlalchemy.orm.relationship) works more effectively when referring to a [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) instance.   * ****active\_history=False**** – When True, indicates that the "previous" value for a many-to-one reference should be loaded when replaced, if not already loaded. Normally, history tracking logic for simple many-to-ones only needs to be aware of the "new" value in order to perform a flush. This flag is available for applications that make use of [attributes.get\_history()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.attributes.get_history" \o "sqlalchemy.orm.attributes.get_history) which also need to know the "previous" value of the attribute. * ****backref –****indicates the string name of a property to be placed on the related mapper's class that will handle this relationship in the other direction. The other property will be created automatically when the mappers are configured. Can also be passed as a [backref()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.backref" \o "sqlalchemy.orm.backref) object to control the configuration of the new relationship.表示要放置在相关映射器类上的属性的字符串名称，将在另一个方向处理此关系。 当配置映射器时，将自动创建其他属性。 也可以作为backref()对象传递来控制新关系的配置。   **See also**  [Linking Relationships with Backref](http://docs.sqlalchemy.org/en/rel_1_1/orm/backref.html" \l "relationships-backref) - Introductory documentation and examples.  [back\_populates](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.back_populates" \o "sqlalchemy.orm.relationship) - alternative form of backref specification.  [backref()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.backref" \o "sqlalchemy.orm.backref) - allows control over [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) configuration when using [backref](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.backref" \o "sqlalchemy.orm.relationship).   * ****back\_populates –****Takes a string name and has the same meaning as [backref](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.backref" \o "sqlalchemy.orm.relationship), except the complementing property is ****not**** created automatically, and instead must be configured explicitly on the other mapper. The complementing property should also indicate [back\_populates](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.back_populates" \o "sqlalchemy.orm.relationship) to this relationship to ensure proper functioning.获取一个字符串名称，并且与backref具有相同的含义，但补全属性不会自动创建，而必须在另一个映射器上显式配置。 补充财产也应该表明对这种关系的反应，以确保正常运作。   **See also**  [Linking Relationships with Backref](http://docs.sqlalchemy.org/en/rel_1_1/orm/backref.html" \l "relationships-backref) - Introductory documentation and examples.  [backref](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.backref" \o "sqlalchemy.orm.relationship) - alternative form of backref specification.   * ****bake\_queries=True –****Use the [BakedQuery](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/baked.html" \l "sqlalchemy.ext.baked.BakedQuery" \o "sqlalchemy.ext.baked.BakedQuery) cache to cache the construction of SQL used in lazy loads, when the [bake\_lazy\_loaders()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/baked.html" \l "sqlalchemy.ext.baked.bake_lazy_loaders" \o "sqlalchemy.ext.baked.bake_lazy_loaders) function has first been called. Defaults to True and is intended to provide an "opt out" flag per-relationship when the baked query cache system is in use.当bake\_lazy\_loaders()函数首先被调用时，使用BakedQuery缓存缓存用于延迟加载的SQL的构造。 默认为True，旨在在baked 焙的查询缓存系统正在使用时为每个关系提供“选择退出”标志。   **Warning**  This flag ****only**** has an effect when the application-wide [bake\_lazy\_loaders()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/baked.html" \l "sqlalchemy.ext.baked.bake_lazy_loaders" \o "sqlalchemy.ext.baked.bake_lazy_loaders) function has been called. It defaults to True so is an "opt out" flag.  此标志只有在应用程序范围的bake\_lazy\_loaders()函数被调用时才有效果。 它默认为True，因此是“选择退出”标志。  Setting this flag to False when baked queries are otherwise in use might be to reduce ORM memory use for this [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship), or to work around unresolved stability issues observed within the baked query cache system.  当使用烘焙查询时，将此标志设置为False可能是为了减少此relationship()的ORM内存使用，或者解决在baked 的查询缓存系统中观察到的未解决的稳定性问题。  *New in version 1.0.0.*  **See also**  [Baked Queries](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/baked.html)   * ****cascade –****a comma-separated list of cascade rules which determines how Session operations should be "cascaded" from parent to child. This defaults to False, which means the default cascade should be used - this default cascade is "save-update, merge".一个逗号分隔的级联规则列表，用于确定会话操作如何从父级到子级“级联”。 这默认为False，这意味着应该使用默认级联 - 这个默认级联是"save-update, merge"。   The available cascades are save-update, merge, expunge, delete, delete-orphan, and refresh-expire. An additional option, all indicates shorthand for "save-update, merge, refresh-expire, expunge, delete", and is often used as in "all, delete-orphan" to indicate that related objects should follow along with the parent object in all cases, and be deleted when de-associated.  可用的级联是save-update, merge, expunge, delete, delete-orphan,和refresh-expire。 一个附加选项，都表示“save-update, merge, refresh-expire, expunge, delete”的缩写，通常用在“all, delete-orphan”中，以表示相关对象应该与父对象一起 所有情况，并且在不关联时被删除。  **See also**  [Cascades](http://docs.sqlalchemy.org/en/rel_1_1/orm/cascades.html" \l "unitofwork-cascades) - Full detail on each of the available cascade options.  [Configuring delete/delete-orphan Cascade](http://docs.sqlalchemy.org/en/rel_1_1/orm/tutorial.html" \l "tutorial-delete-cascade) - Tutorial example describing a delete cascade.   * ****cascade\_backrefs=True –****a boolean value indicating if the save-update cascade should operate along an assignment event intercepted by a backref. When set to False, the attribute managed by this relationship will not cascade an incoming transient object into the session of a persistent parent, if the event is received via backref.   **See also**  [Controlling Cascade on Backrefs](http://docs.sqlalchemy.org/en/rel_1_1/orm/cascades.html" \l "backref-cascade) - Full discussion and examples on how the [cascade\_backrefs](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.cascade_backrefs" \o "sqlalchemy.orm.relationship) option is used.   * ****collection\_class –****a class or callable that returns a new list-holding object. will be used in place of a plain list for storing elements.   **See also**  [Customizing Collection Access](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "custom-collections) - Introductory documentation and examples.   * ****comparator\_factory –****a class which extends [RelationshipProperty.Comparator](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator) which provides custom SQL clause generation for comparison operations.   **See also**  [PropComparator](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator" \o "sqlalchemy.orm.interfaces.PropComparator) - some detail on redefining comparators at this level.  [Operator Customization](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapped_attributes.html" \l "custom-comparators) - Brief intro to this feature.   * ****distinct\_target\_key=None –****Indicate if a "subquery" eager load should apply the DISTINCT keyword to the innermost SELECT statement. When left as None, the DISTINCT keyword will be applied in those cases when the target columns do not comprise the full primary key of the target table. When set to True, the DISTINCT keyword is applied to the innermost SELECT unconditionally.   It may be desirable to set this flag to False when the DISTINCT is reducing performance of the innermost subquery beyond that of what duplicate innermost rows may be causing.  *New in version 0.8.3:*- [distinct\_target\_key](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.distinct_target_key" \o "sqlalchemy.orm.relationship) allows the subquery eager loader to apply a DISTINCT modifier to the innermost SELECT.  *Changed in version 0.9.0:*- [distinct\_target\_key](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.distinct_target_key" \o "sqlalchemy.orm.relationship) now defaults to None, so that the feature enables itself automatically for those cases where the innermost query targets a non-unique key.  **See also**  [Relationship Loading Techniques](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html) - includes an introduction to subquery eager loading.   * ****doc**** – docstring which will be applied to the resulting descriptor. * ****extension –****an [AttributeExtension](http://docs.sqlalchemy.org/en/rel_1_1/orm/deprecated.html" \l "sqlalchemy.orm.interfaces.AttributeExtension" \o "sqlalchemy.orm.interfaces.AttributeExtension) instance, or list of extensions, which will be prepended to the list of attribute listeners for the resulting descriptor placed on the class.   *Deprecated since version 0.7:*Please see [AttributeEvents](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.AttributeEvents" \o "sqlalchemy.orm.events.AttributeEvents).   * ****foreign\_keys –**** a list of columns which are to be used as "foreign key" columns, or columns which refer to the value in a remote column, within the context of this [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) object's [primaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.primaryjoin" \o "sqlalchemy.orm.relationship) condition. That is, if the [primaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.primaryjoin" \o "sqlalchemy.orm.relationship) condition of this [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) is a.id == b.a\_id, and the values in b.a\_id are required to be present in a.id, then the "foreign key" column of this [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) is b.a\_id.在该关系()对象的primary join条件的上下文中，要用作“外键”列的列的列表或引用远程列中的值的列。 也就是说，如果这个关系()的primaryjoin条件是a.id == b.a\_id，并且b.a\_id中的值需要存在于a.id中，那么这个关系的“foreign key”列)是b.a\_id。   In normal cases, the [foreign\_keys](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.foreign_keys" \o "sqlalchemy.orm.relationship) parameter is ****not required.**** [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) will automatically determine which columns in the [primaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.primaryjoin" \o "sqlalchemy.orm.relationship)conditition are to be considered "foreign key" columns based on those [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects that specify [ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey), or are otherwise listed as referencing columns in a [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint) construct. [foreign\_keys](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.foreign_keys" \o "sqlalchemy.orm.relationship) is only needed when:  在正常情况下，不需要foreign\_keys参数。 relationship()将根据指定ForeignKey的Column对象自动确定主链接中哪些列被视为“外键”列，否则列为ForeignKeyConstraint构造中的引用列。 仅在以下情况下才需要foreign\_keys：  There is more than one way to construct a join from the local table to the remote table, as there are multiple foreign key references present. Setting foreign\_keys will limit the [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) to consider just those columns specified here as "foreign"  从本地表到远程表构造一个连接有多种方式，因为存在多个外键引用。 设置foreign\_keys将限制relationship()仅将这里指定的列视为“foreign”  *Changed in version 0.8:*A multiple-foreign key join ambiguity can be resolved by setting the [foreign\_keys](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.foreign_keys" \o "sqlalchemy.orm.relationship) parameter alone, without the need to explicitly set [primaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.primaryjoin" \o "sqlalchemy.orm.relationship) as well.  The [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) being mapped does not actually have [ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey) or [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint) constructs present, often because the table was reflected from a database that does not support foreign key reflection (MySQL MyISAM).  正在映射的表实际上不存在ForeignKey或ForeignKeyConstraint结构，通常是因为该表从不支持外键反射(MySQL MyISAM)的数据库中反映出来。  The [primaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.primaryjoin" \o "sqlalchemy.orm.relationship) argument is used to construct a non-standard join condition, which makes use of columns or expressions that do not normally refer to their "parent" column, such as a join condition expressed by a complex comparison using a SQL function.  primaryjoin参数用于构造非标准连接条件，它使用通常不引用其“父”列的列或表达式，例如通过使用SQL函数的复杂比较表示的连接条件。  The [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) construct will raise informative error messages that suggest the use of the [foreign\_keys](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.foreign_keys" \o "sqlalchemy.orm.relationship) parameter when presented with an ambiguous condition. In typical cases, if [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) doesn't raise any exceptions, the [foreign\_keys](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.foreign_keys" \o "sqlalchemy.orm.relationship) parameter is usually not needed.  relationship()构造会引发提示信息性错误消息，该消息在呈现模糊条件时建议使用foreign\_keys参数。 在典型情况下，如果relationship()不引发任何异常，则通常不需要foreign\_keys参数。  [foreign\_keys](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.foreign_keys" \o "sqlalchemy.orm.relationship) may also be passed as a callable function which is evaluated at mapper initialization time, and may be passed as a Python-evaluable string when using Declarative.  foreign\_keys也可以作为可映射函数传递，该函数在映射器初始化时被评估，并且可以在使用Declarative时作为Python可评估字符串传递。  **See also**  [Handling Multiple Join Paths](http://docs.sqlalchemy.org/en/rel_1_1/orm/join_conditions.html" \l "relationship-foreign-keys)  [Creating Custom Foreign Conditions](http://docs.sqlalchemy.org/en/rel_1_1/orm/join_conditions.html" \l "relationship-custom-foreign)  [foreign()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.foreign" \o "sqlalchemy.orm.foreign) - allows direct annotation of the "foreign" columns within a [primaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.primaryjoin" \o "sqlalchemy.orm.relationship) condition.  *New in version 0.8:*The [foreign()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.foreign" \o "sqlalchemy.orm.foreign) annotation can also be applied directly to the [primaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.primaryjoin" \o "sqlalchemy.orm.relationship) expression, which is an alternate, more specific system of describing which columns in a particular [primaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.primaryjoin" \o "sqlalchemy.orm.relationship) should be considered "foreign".  版本0.8中的新功能：[foreign()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.foreign" \o "sqlalchemy.orm.foreign)注释也可以直接应用于[primaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.primaryjoin" \o "sqlalchemy.orm.relationship)表达式，该表达式是一个替代的，更具体的系统，用于描述特定[primaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.primaryjoin" \o "sqlalchemy.orm.relationship)中哪些列应被视为“foreign”。   * ****info –****Optional data dictionary which will be populated into the [MapperProperty.info](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "MapperProperty.info" \o "MapperProperty.info) attribute of this object.可选的数据字典将被填充到该对象的[MapperProperty.info](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "MapperProperty.info" \o "MapperProperty.info)属性中。   *New in version 0.8.*   * ****innerjoin=False –****when True, joined eager loads will use an inner join to join against related tables instead of an outer join. The purpose of this option is generally one of performance, as inner joins generally perform better than outer joins.如果真的，加入的加载将使用内部连接来连接相关表，而不是外连接。 此选项的目的通常是性能之一，因为内部连接通常比外部连接执行得更好。   This flag can be set to True when the relationship references an object via many-to-one using local foreign keys that are not nullable, or when the reference is one-to-one or a collection that is guaranteed to have one or at least one entry.当关系通过多对一使用不可空的本地外键引用对象时，或者当引用是一对一的引用或者一个保证有一个或至少一个的集合时，该标志可以设置为True 一个条目。  The option supports the same "nested" and "unnested" options as that of [joinedload.innerjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.strategy_options.Load.joinedload.params.innerjoin" \o "sqlalchemy.orm.strategy_options.Load.joinedload). See that flag for details on nested / unnested behaviors.该选项支持与joinedload.innerjoin相同的“嵌套”和“不可用”选项。 有关嵌套/不需要的行为的详细信息，请参阅该标志。  **See also**  [joinedload.innerjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.strategy_options.Load.joinedload.params.innerjoin" \o "sqlalchemy.orm.strategy_options.Load.joinedload) - the option as specified by loader option, including detail on nesting behavior.  [What Kind of Loading to Use ?](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "what-kind-of-loading) - Discussion of some details of various loader options.  joinedload.inner join - 由loader选项指定的选项，包括嵌套行为的详细信息。  什么样的加载使用？ - 讨论各种装载机选项的一些细节。   * ****join\_depth –****when non-None, an integer value indicating how many levels deep "eager" loaders should join on a self-referring or cyclical relationship. The number counts how many times the same Mapper shall be present in the loading condition along a particular join branch. When left at its default of None, eager loaders will stop chaining when they encounter a the same target mapper which is already higher up in the chain. This option applies both to joined- and subquery- eager loaders. * 当非“无”时，一个整数值表示深度“渴望”装载机应该参与自我参照或周期性关系的级别。 该数字计算在特定连接分支的加载条件下同一Mapper将存在多少次。 当它的默认值为None时，当它们遇到已经在链中已经较高的目标映射器时，热切的加载器将停止链接。 此选项适用于连接和子查询加载器。   **See also**  [Configuring Self-Referential Eager Loading](http://docs.sqlalchemy.org/en/rel_1_1/orm/self_referential.html" \l "self-referential-eager-loading) - Introductory documentation and examples.   * ****lazy='select' –****specifies how the related items should be loaded. Default value is select. Values include:   select - items should be loaded lazily when the property is first accessed, using a separate SELECT statement, or identity map fetch for simple many-to-one references.选择 - 应首先访问属性时使用单独的SELECT语句或简单的多对一引用的身份映射提取来懒惰地加载项。  immediate - items should be loaded as the parents are loaded, using a separate SELECT statement, or identity map fetch for simple many-to-one references.应当使用单独的SELECT语句或简单的多对一引用的身份映射提取来加载父项的立即项。  joined - items should be loaded "eagerly" in the same query as that of the parent, using a JOIN or LEFT OUTER JOIN. Whether the join is "outer" or not is determined by the [innerjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.innerjoin" \o "sqlalchemy.orm.relationship) parameter.加入的项目应该在与父级的查询相同的查询中使用JOIN或LEFT OUTER JOIN加载。连接是否为“外部”是否由innerjoin参数决定。  subquery - items should be loaded "eagerly" as the parents are loaded, using one additional SQL statement, which issues a JOIN to a subquery of the original statement, for each collection requested.子查询 - 项目应该像父母一样被加载加载，使用一个额外的SQL语句，它为原始语句的子查询发出一个JOIN，用于每个请求的集合。  noload - no loading should occur at any time. This is to support "write-only" attributes, or attributes which are populated in some manner specific to the application.noload - 任何时候都不应该加载。这是为了支持“只写”属性，或以某种特定于应用程序的方式填充的属性。  raise - lazy loading is disallowed; accessing the attribute, if its value were not already loaded via eager loading, will raise an [InvalidRequestError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.InvalidRequestError" \o "sqlalchemy.exc.InvalidRequestError). This strategy can be used when objects are to be detached from their attached [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) after they are loaded.不加懒惰装载;访问该属性(如果其值尚未通过加载加载)将引发InvalidRequestError。当对象在加载后将从其附加的会话中分离出来时，可以使用此策略。  *New in version 1.1.*  1.1版新功能  raise\_on\_sql - lazy loading that emits SQL is disallowed; accessing the attribute, if its value were not already loaded via eager loading, will raise an [InvalidRequestError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.InvalidRequestError" \o "sqlalchemy.exc.InvalidRequestError), ****if the lazy load needs to emit SQL****. If the lazy load can pull the related value from the identity map or determine that it should be None, the value is loaded. This strategy can be used when objects will remain associated with the attached [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session), however additional SELECT statements should be blocked.禁止发出SQL的惰性加载; 访问属性，如果其值尚未通过加载加载加载，则会引发一个InvalidRequestError，如果延迟加载需要发出SQL。 如果懒惰负载可以从身份映射中提取相关值，或确定它应该为无，则该值将被加载。 当对象将保持与附加的会话相关联时，可以使用此策略，但是应该阻止其他SELECT语句。  *New in version 1.1.*  dynamic - the attribute will return a pre-configured  [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) object for all read operations, onto which further filtering operations can be applied before iterating the results. See the section [Dynamic Relationship Loaders](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "dynamic-relationship) for more details.该属性将返回一个预先配置的查询所有读操作的对象，在迭代结果之前可以应用进一步的过滤操作。 有关详细信息，请参阅动态关系装载器一节。  True - a synonym for 'select'  False - a synonym for 'joined'  None - a synonym for 'noload'  **See also**  [Relationship Loading Techniques](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html) - Full documentation on relationship loader configuration.  [Dynamic Relationship Loaders](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "dynamic-relationship) - detail on the dynamic option.  [Setting Noload, RaiseLoad](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "collections-noload-raiseload) - notes on "noload" and "raise"   * ****load\_on\_pending=False –****Indicates loading behavior for transient or pending parent objects.   When set to True, causes the lazy-loader to issue a query for a parent object that is not persistent, meaning it has never been flushed. This may take effect for a pending object when autoflush is disabled, or for a transient object that has been "attached" to a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) but is not part of its pending collection.  当设置为True时，会导致lazy-loader对不持久的父对象发出查询，这意味着它从未被刷新。 当禁用自动冲洗时，这可能会对待处理对象生效，或者对已经“附加”到会话而不是其挂起集合的一部分的瞬态对象可能会生效。  The [load\_on\_pending](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.load_on_pending" \o "sqlalchemy.orm.relationship) flag does not improve behavior when the ORM is used normally - object references should be constructed at the object level, not at the foreign key level, so that they are present in an ordinary way before a flush proceeds. This flag is not not intended for general use.  当正常使用ORM时，load\_on\_pending标志不会改善行为 - 对象引用应在对象级别而不是在外键级别构建，以便在进行刷新之前以普通方式存在对象引用。 该标志不是一般用途。  **See also**  [Session.enable\_relationship\_loading()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.enable_relationship_loading" \o "sqlalchemy.orm.session.Session.enable_relationship_loading) - this method establishes "load on pending" behavior for the whole object, and also allows loading on objects that remain transient or detached.   * ****order\_by –****indicates the ordering that should be applied when loading these items. [order\_by](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.order_by" \o "sqlalchemy.orm.relationship) is expected to refer to one of the [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects to which the target class is mapped, or the attribute itself bound to the target class which refers to the column.   [order\_by](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.order_by" \o "sqlalchemy.orm.relationship) may also be passed as a callable function which is evaluated at mapper initialization time, and may be passed as a Python-evaluable string when using Declarative.   * ****passive\_deletes=False –****Indicates loading behavior during delete operations.   A value of True indicates that unloaded child items should not be loaded during a delete operation on the parent. Normally, when a parent item is deleted, all child items are loaded so that they can either be marked as deleted, or have their foreign key to the parent set to NULL. Marking this flag as True usually implies an ON DELETE <CASCADE|SET NULL> rule is in place which will handle updating/deleting child rows on the database side.  值为True表示在父项的删除操作期间不应加载卸载的子项。 通常，当删除父项时，将加载所有子项，以便将其标记为已删除，或将其外键设置为NULL。 将此标志标记为True通常意味着将执行ON DELETE <CASCADE | SET NULL>规则，该规则将处理数据库端更新/删除子行。  Additionally, setting the flag to the string value 'all' will disable the "nulling out" of the child foreign keys, when the parent object is deleted and there is no delete or delete-orphan cascade enabled. This is typically used when a triggering or error raise scenario is in place on the database side. Note that the foreign key attributes on in-session child objects will not be changed after a flush occurs so this is a very special use-case setting. Additionally, the "nulling out" will still occur if the child object is de-associated with the parent.  此外，将标志设置为字符串值“all”将禁用父对象被删除并且没有启用删除或删除或孤立级联的“外出”的子外键。 通常在数据库端发生触发或错误提升情况时使用。 请注意，冲突发生后，会话中子对象的外键属性将不会更改，因此这是非常特殊的用例设置。 此外，如果子对象与父对象取消关联，则仍将发生“归零”。  **See also**  [Using Passive Deletes](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "passive-deletes) - Introductory documentation and examples.   * ****passive\_updates=True –****Indicates the persistence behavior to take when a referenced primary key value changes in place, indicating that the referencing foreign key columns will also need their value changed.   When True, it is assumed that ON UPDATE CASCADE is configured on the foreign key in the database, and that the database will handle propagation of an UPDATE from a source column to dependent rows. When False, the SQLAlchemy [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) construct will attempt to emit its own UPDATE statements to modify related targets. However note that SQLAlchemy ****cannot**** emit an UPDATE for more than one level of cascade. Also, setting this flag to False is not compatible in the case where the database is in fact enforcing referential integrity, unless those constraints are explicitly "deferred", if the target backend supports it.  当为True时，假定在数据库中的外键上配置了ON UPDATE CASCADE，并且数据库将处理UPDATE从源列到依赖行的传播。 当False时，SQLAlchemy [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)结构将尝试发出自己的UPDATE语句来修改相关的目标。 但是请注意，SQLAlchemy不能为多个级联的级联发出UPDATE。 而且，在数据库实际上执行参照完整性的情况下，将此标志设置为False是不兼容的，除非这些约束被明确地“延迟”，如果目标后端支持它。  It is highly advised that an application which is employing mutable primary keys keeps passive\_updates set to True, and instead uses the referential integrity features of the database itself in order to handle the change efficiently and fully.  强烈建议使用可变主键的应用程序将passive\_updates设置为True，而是使用数据库本身的引用完整性功能，以便有效和全面地处理更改。  **See also**  [Mutable Primary Keys / Update Cascades](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_persistence.html" \l "passive-updates) - Introductory documentation and examples.  [mapper.passive\_updates](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.params.passive_updates" \o "sqlalchemy.orm.mapper) - a similar flag which takes effect for joined-table inheritance mappings.   * ****post\_update –****this indicates that the relationship should be handled by a second UPDATE statement after an INSERT or before a DELETE. Currently, it also will issue an UPDATE after the instance was UPDATEd as well, although this technically should be improved. This flag is used to handle saving bi-directional dependencies between two individual rows (i.e. each row references the other), where it would otherwise be impossible to INSERT or DELETE both rows fully since one row exists before the other. Use this flag when a particular mapping arrangement will incur two rows that are dependent on each other, such as a table that has a one-to-many relationship to a set of child rows, and also has a column that references a single child row within that list (i.e. both tables contain a foreign key to each other). If a flush operation returns an error that a "cyclical dependency" was detected, this is a cue that you might want to use [post\_update](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.post_update" \o "sqlalchemy.orm.relationship) to "break" the cycle. * 这表示该关系应该在INSERT之后或在DELETE之前由第二个UPDATE语句处理。 目前，在实例UPDATEd之后，它也会发布UPDATE，尽管这在技术上应该得到改进。 该标志用于处理两个单独行之间保存双向依赖关系(即每行引用另一行)，否则将不可能完全INSERT或DELETE两行，因为一行存在于另一行之前。 当特定的映射布置将导致彼此依赖的两行(例如与一组子行具有一对多关系的表)，并且还具有引用单个子行的列时，使用此标志 在该列表中(即，两个表彼此之间都包含一个外键)。 如果flush操作返回一个检测到“循环依赖”的错误，这是一个提示，您可能希望使用post\_update来“打破”循环。   **See also**  [Rows that point to themselves / Mutually Dependent Rows](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_persistence.html" \l "post-update) - Introductory documentation and examples.   * ****primaryjoin –****a SQL expression that will be used as the primary join of this child object against the parent object, or in a many-to-many relationship the join of the primary object to the association table. By default, this value is computed based on the foreign key relationships of the parent and child tables (or association table).一个SQL表达式，将被用作该子对象与父对象的主连接，或者以多对多关系将主对象的连接用于关联表。 默认情况下，此值是根据父表和子表(或关联表)的外键关系计算的。   [primaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.primaryjoin" \o "sqlalchemy.orm.relationship) may also be passed as a callable function which is evaluated at mapper initialization time, and may be passed as a Python-evaluable string when using Declarative.  primaryjoin也可以作为可调用函数传递，该函数在映射器初始化时被评估，并且可以在使用Declarative时作为Python可评估字符串传递。  **See also**  [Specifying Alternate Join Conditions](http://docs.sqlalchemy.org/en/rel_1_1/orm/join_conditions.html" \l "relationship-primaryjoin)   * ****remote\_side –****used for self-referential relationships, indicates the column or list of columns that form the "remote side" of the relationship.   [relationship.remote\_side](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.remote_side" \o "sqlalchemy.orm.relationship) may also be passed as a callable function which is evaluated at mapper initialization time, and may be passed as a Python-evaluable string when using Declarative.  *Changed in version 0.8:*The [remote()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.remote" \o "sqlalchemy.orm.remote) annotation can also be applied directly to the primaryjoin expression, which is an alternate, more specific system of describing which columns in a particular primaryjoin should be considered "remote".  **See also**  [Adjacency List Relationships](http://docs.sqlalchemy.org/en/rel_1_1/orm/self_referential.html" \l "self-referential) - in-depth explanation of how [remote\_side](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.remote_side" \o "sqlalchemy.orm.relationship) is used to configure self-referential relationships.  [remote()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.remote" \o "sqlalchemy.orm.remote) - an annotation function that accomplishes the same purpose as [remote\_side](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.remote_side" \o "sqlalchemy.orm.relationship), typically when a custom [primaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.primaryjoin" \o "sqlalchemy.orm.relationship) condition is used.   * ****query\_class –****a [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) subclass that will be used as the base of the "appender query" returned by a "dynamic" relationship, that is, a relationship that specifies lazy="dynamic" or was otherwise constructed using the [orm.dynamic\_loader()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.dynamic_loader" \o "sqlalchemy.orm.dynamic_loader) function.   **See also**  [Dynamic Relationship Loaders](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "dynamic-relationship) - Introduction to "dynamic" relationship loaders.   * ****secondaryjoin –****a SQL expression that will be used as the join of an association table to the child object. By default, this value is computed based on the foreign key relationships of the association and child tables.   [secondaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.secondaryjoin" \o "sqlalchemy.orm.relationship) may also be passed as a callable function which is evaluated at mapper initialization time, and may be passed as a Python-evaluable string when using Declarative.  **See also**  [Specifying Alternate Join Conditions](http://docs.sqlalchemy.org/en/rel_1_1/orm/join_conditions.html" \l "relationship-primaryjoin)   * ****single\_parent –****when True, installs a validator which will prevent objects from being associated with more than one parent at a time. This is used for many-to-one or many-to-many relationships that should be treated either as one-to-one or one-to-many. Its usage is optional, except for [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) constructs which are many-to-one or many-to-many and also specify the delete-orphan cascade option. The [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) construct itself will raise an error instructing when this option is required.   **See also**  [Cascades](http://docs.sqlalchemy.org/en/rel_1_1/orm/cascades.html" \l "unitofwork-cascades) - includes detail on when the [single\_parent](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.single_parent" \o "sqlalchemy.orm.relationship) flag may be appropriate.   * ****uselist –****a boolean that indicates if this property should be loaded as a list or a scalar. In most cases, this value is determined automatically by [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) at mapper configuration time, based on the type and direction of the relationship - one to many forms a list, many to one forms a scalar, many to many is a list. If a scalar is desired where normally a list would be present, such as a bi-directional one-to-one relationship, set [uselist](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.uselist" \o "sqlalchemy.orm.relationship) to False.   The [uselist](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.uselist" \o "sqlalchemy.orm.relationship) flag is also available on an existing [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) construct as a read-only attribute, which can be used to determine if this [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) deals with collections or scalar attributes:  **>>>** User.addresses.property.uselistTrue  **See also**  [One To One](http://docs.sqlalchemy.org/en/rel_1_1/orm/basic_relationships.html" \l "relationships-one-to-one) - Introduction to the "one to one" relationship pattern, which is typically when the [uselist](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.uselist" \o "sqlalchemy.orm.relationship) flag is needed.   * ****viewonly=False**** – when set to True, the relationship is used only for loading objects, and not for any persistence operation. A [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) which specifies [viewonly](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.viewonly" \o "sqlalchemy.orm.relationship) can work with a wider range of SQL operations within the [primaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.primaryjoin" \o "sqlalchemy.orm.relationship) condition, including operations that feature the use of a variety of comparison operators as well as SQL functions such as [cast()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.cast" \o "sqlalchemy.sql.expression.cast). The [viewonly](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.viewonly" \o "sqlalchemy.orm.relationship) flag is also of general use when defining any kind of [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) that doesn't represent the full set of related objects, to prevent modifications of the collection from resulting in persistence operations.当设置为True时，该关系仅用于加载对象，而不是用于任何持久性操作。 指定[viewonly](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.viewonly" \o "sqlalchemy.orm.relationship) 的[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)可以在[primaryjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.primaryjoin" \o "sqlalchemy.orm.relationship)条件下使用更广泛的SQL操作，包括使用各种比较运算符的操作以及诸如[cast()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.cast" \o "sqlalchemy.sql.expression.cast)之类的SQL函数。 定义任何类型的[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)并不代表完整的相关对象集合时，[viewonly](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.viewonly" \o "sqlalchemy.orm.relationship) 标志也是一般用途，以防止对集合的修改导致持久化操作。 |

sqlalchemy.orm.**backref**(*name*, *\*\*kwargs*)

Create a back reference with explicit keyword arguments, which are the same arguments one can send to [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship).

Used with the backref keyword argument to [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) in place of a string argument, e.g.:

'items':relationship(

SomeItem, backref=backref('parent', lazy='subquery'))

**See also**

[Linking Relationships with Backref](http://docs.sqlalchemy.org/en/rel_1_1/orm/backref.html" \l "relationships-backref)

sqlalchemy.orm.**relation**(*\*arg*, *\*\*kw*)

A synonym for [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship).

sqlalchemy.orm.**dynamic\_loader**(*argument*, *\*\*kw*)

Construct a dynamically-loading mapper property.

This is essentially the same as using the lazy='dynamic' argument with [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship):

dynamic\_loader(SomeClass)

*# is the same as*

relationship(SomeClass, lazy="dynamic")

See the section [Dynamic Relationship Loaders](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "dynamic-relationship) for more details on dynamic loading.

sqlalchemy.orm.**foreign**(*expr*)

Annotate a portion of a primaryjoin expression with a 'foreign' annotation.

See the section [Creating Custom Foreign Conditions](http://docs.sqlalchemy.org/en/rel_1_1/orm/join_conditions.html" \l "relationship-custom-foreign) for a description of use.

*New in version 0.8.*

**See also**

[Creating Custom Foreign Conditions](http://docs.sqlalchemy.org/en/rel_1_1/orm/join_conditions.html" \l "relationship-custom-foreign)

[remote()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.remote" \o "sqlalchemy.orm.remote)

sqlalchemy.orm.**remote**(*expr*)

Annotate a portion of a primaryjoin expression with a 'remote' annotation.

See the section [Creating Custom Foreign Conditions](http://docs.sqlalchemy.org/en/rel_1_1/orm/join_conditions.html" \l "relationship-custom-foreign) for a description of use.

*New in version 0.8.*

**See also**

[Creating Custom Foreign Conditions](http://docs.sqlalchemy.org/en/rel_1_1/orm/join_conditions.html" \l "relationship-custom-foreign)

[foreign()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.foreign" \o "sqlalchemy.orm.foreign)

# 

# CHAPTER 4 Loading Objects

## 4.1 Loading Columns

This section presents additional options regarding the loading of columns.

本节介绍有关加载列的其他选项。

### 4.1.1 Deferred Column Loading

This feature allows particular columns of a table be loaded only upon direct access, instead of when the entity is queried using [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query). This feature is useful when one wants to avoid loading a large text or binary field into memory when it's not needed. Individual columns can be lazy loaded by themselves or placed into groups that lazy-load together, using the [orm.deferred()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "sqlalchemy.orm.deferred" \o "sqlalchemy.orm.deferred) function to mark them as "deferred". In the example below, we define a mapping that will load each of .excerpt and .photo in separate, individual-row SELECT statements when each attribute is first referenced on the individual object instance:

此功能允许仅在直接访问时加载表的特定列，而不是使用[Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query)查询实体。 当想要避免在不需要时将大文本或二进制字段加载到内存中时，此功能非常有用。 可以使用orm.deferred()函数将各个列自身进行惰性加载，也可以将其放置在一起，将其标记为"延迟"。 在下面的示例中，我们定义一个映射，每个属性首先在单个对象实例上引用时，会将.excerpt和.photo中的每一个加载到单独的单行SELECT语句中。

**from** **sqlalchemy.orm** **import** deferred

**from** **sqlalchemy** **import** Integer, String, Text, Binary, Column

**class** **Book**(Base):

\_\_tablename\_\_ = 'book'

book\_id = Column(Integer, primary\_key=**True**)

title = Column(String(200), nullable=**False**)

summary = Column(String(2000))

excerpt = deferred(Column(Text))

photo = deferred(Column(Binary))

Classical mappings as always place the usage of [orm.deferred()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "sqlalchemy.orm.deferred" \o "sqlalchemy.orm.deferred) in the properties dictionary against the table-bound [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column):

始终将属性字典中的[orm.deferred()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "sqlalchemy.orm.deferred" \o "sqlalchemy.orm.deferred)的用法放在表格[Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column)中：

mapper(Book, book\_table, properties={

'photo':deferred(book\_table.c.photo)})

Deferred columns can be associated with a "group" name, so that they load together when any of them are first accessed. The example below defines a mapping with a photos deferred group. When one .photo is accessed, all three photos will be loaded in one SELECT statement. The .excerpt will be loaded separately when it is accessed:

延迟列可以与"group"名称相关联，以便在首次访问它们时将它们加载在一起。 下面的示例定义了一个带有照片延迟组的映射。 当访问一个.photo时，所有三张照片将被加载到一个SELECT语句中。 .excerpt将在访问时单独加载：

**class** **Book**(Base):

\_\_tablename\_\_ = 'book'

book\_id = Column(Integer, primary\_key=**True**)

title = Column(String(200), nullable=**False**)

summary = Column(String(2000))

excerpt = deferred(Column(Text))

photo1 = deferred(Column(Binary), group='photos')

photo2 = deferred(Column(Binary), group='photos')

photo3 = deferred(Column(Binary), group='photos')

You can defer or undefer columns at the [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) level using options, including  [orm.defer()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "sqlalchemy.orm.defer" \o "sqlalchemy.orm.defer)  and [orm.undefer()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "sqlalchemy.orm.undefer" \o "sqlalchemy.orm.undefer):

您可以使用选项包括[orm.defer()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "sqlalchemy.orm.defer" \o "sqlalchemy.orm.defer)和[orm.undefer()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "sqlalchemy.orm.undefer" \o "sqlalchemy.orm.undefer)在[Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query)级别延迟或取消冻结列：

**from** **sqlalchemy.orm** **import** defer, undefer

query = session.query(Book)

query = query.options(defer('summary'))

query = query.options(undefer('excerpt'))query.all()

[orm.deferred()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "sqlalchemy.orm.deferred" \o "sqlalchemy.orm.deferred) attributes which are marked with a "group" can be undeferred using  [orm.undefer\_group()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "sqlalchemy.orm.undefer_group" \o "sqlalchemy.orm.undefer_group), sending in the group name:

标有"group"的[orm.deferred()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "sqlalchemy.orm.deferred" \o "sqlalchemy.orm.deferred)属性可以使用[orm.undefer\_group()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "sqlalchemy.orm.undefer_group" \o "sqlalchemy.orm.undefer_group)取消删除，发送组名：

**from** **sqlalchemy.orm** **import** undefer\_group

query = session.query(Book)query.options(undefer\_group('photos')).all()

**Load Only Cols**

An arbitrary set of columns can be selected as "load only" columns, which will be loaded while deferring all other columns on a given entity, using [orm.load\_only()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "sqlalchemy.orm.load_only" \o "sqlalchemy.orm.load_only):

**from** **sqlalchemy.orm** **import** load\_only

session.query(Book).options(load\_only("summary", "excerpt"))

*New in version 0.9.0.*

**Deferred Loading with Multiple Entities**

To specify column deferral options within a [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) that loads multiple types of entity, the [Load](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.Load" \o "sqlalchemy.orm.Load) object can specify which parent entity to start with:

**from** **sqlalchemy.orm** **import** Load

query = session.query(Book, Author).join(Book.author)query = query.options(

Load(Book).load\_only("summary", "excerpt"),

Load(Author).defer("bio")

)

To specify column deferral options along the path of various relationships, the options support chaining, where the loading style of each relationship is specified first, then is chained to the deferral options. Such as, to load Book instances, then joined-eager-load the Author, then apply deferral options to the Author entity:

**from** **sqlalchemy.orm** **import** joinedload

query = session.query(Book)query = query.options(

joinedload(Book.author).load\_only("summary", "excerpt"),

)

In the case where the loading style of parent relationships should be left unchanged, use [orm.defaultload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.defaultload" \o "sqlalchemy.orm.defaultload):

**from** **sqlalchemy.orm** **import** defaultload

query = session.query(Book)query = query.options(

defaultload(Book.author).load\_only("summary", "excerpt"),

)

*New in version 0.9.0:*support for [Load](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.Load" \o "sqlalchemy.orm.Load) and other options which allow for better targeting of deferral options.

**Column Deferral API**

sqlalchemy.orm.**deferred**(*\*columns*, *\*\*kw*)

Indicate a column-based mapped attribute that by default will not load unless accessed.

指示基于列的映射属性，默认情况下不会加载，除非被访问。

|  |  |
| --- | --- |
| **Parameters:** | * ****\*columns**** – columns to be mapped. This is typically a single [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) object, however a collection is supported in order to support multiple columns mapped under the same attribute.要映射的列。 这通常是一个单独的Column对象，但是支持集合以支持在同一属性下映射的多个列。 * ****\*\*kw**** – additional keyword arguments passed to [ColumnProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.ColumnProperty" \o "sqlalchemy.orm.properties.ColumnProperty).传递给ColumnProperty的附加关键字参数。 |

**See also**

[Deferred Column Loading](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "deferred)

sqlalchemy.orm.**defer**(*key*, *\*addl\_attrs*)

Indicate that the given column-oriented attribute should be deferred, e.g. not loaded until accessed.

表明给定的面向列的属性应该被推迟，例如 未加载直到访问。

This function is part of the [Load](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.Load" \o "sqlalchemy.orm.Load) interface and supports both method-chained and standalone operation.

此功能是Load接口的一部分，支持方法链接和独立操作。

e.g.:

**from** **sqlalchemy.orm** **import** defer

session.query(MyClass).options(

defer("attribute\_one"),

defer("attribute\_two"))

session.query(MyClass).options(

defer(MyClass.attribute\_one),

defer(MyClass.attribute\_two))

To specify a deferred load of an attribute on a related class, the path can be specified one token at a time, specifying the loading style for each link along the chain. To leave the loading style for a link unchanged, use [orm.defaultload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.defaultload" \o "sqlalchemy.orm.defaultload):

要指定相关类上的属性的延迟加载，可以一次指定一个令牌，为每个链路指定加载样式。 要使链接的加载样式保持不变，请使用[orm.defaultload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.defaultload" \o "sqlalchemy.orm.defaultload)：

session.query(MyClass).options(defaultload("someattr").defer("some\_column"))

A [Load](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.Load" \o "sqlalchemy.orm.Load) object that is present on a certain path can have [Load.defer()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.strategy_options.Load.defer" \o "sqlalchemy.orm.strategy_options.Load.defer) called multiple times, each will operate on the same parent entity:

存在于某个路径上的[Load](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.Load" \o "sqlalchemy.orm.Load)对象可以多次调用[Load.defer()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.strategy_options.Load.defer" \o "sqlalchemy.orm.strategy_options.Load.defer)，每个对象将在同一个父实体上运行：

session.query(MyClass).options(

defaultload("someattr").

defer("some\_column").

defer("some\_other\_column").

defer("another\_column")

)

|  |  |
| --- | --- |
| **Parameters:** | * ****key**** – Attribute to be deferred.属性被推迟 * ****~~\*addl\_attrs~~****~~– Deprecated; this option supports the old 0.8 style of specifying a path as a series of attributes, which is now superseded by the method-chained style.~~ |

**See also**

[Deferred Column Loading](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "deferred)

[orm.undefer()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "sqlalchemy.orm.undefer" \o "sqlalchemy.orm.undefer)

sqlalchemy.orm.**load\_only**(*\*attrs*)

Indicate that for a particular entity, only the given list of column-based attribute names should be loaded; all others will be deferred.

指示对于特定实体，只应加载给定的基于列的属性名称列表; 所有其他人将被推迟。

This function is part of the [Load](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.Load" \o "sqlalchemy.orm.Load) interface and supports both method-chained and standalone operation.

此功能是Load接口的一部分，支持方法链接和独立操作。

Example - given a class User, load only the name and fullname attributes:

示例 - 给一个类User，只加载名称和全名属性：

session.query(User).options(load\_only("name", "fullname"))

Example - given a relationship User.addresses -> Address, specify subquery loading for the User.addresses collection, but on each Address object load only the email\_address attribute:

示例 - 给定一个关系User.addresses - > Address，为User.addresses集合指定子查询加载，但在每个Address对象上只加载email\_address属性：

session.query(User).options(

subqueryload("addresses").load\_only("email\_address"))

For a [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) that has multiple entities, the lead entity can be specifically referred to using the [Load](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.Load" \o "sqlalchemy.orm.Load) constructor:

对于具有多个实体的查询，可以使用Load构造函数具体引用引导实体：

session.query(User, Address).join(User.addresses).options(

Load(User).load\_only("name", "fullname"),

Load(Address).load\_only("email\_addres")

)

*New in version 0.9.0.*

sqlalchemy.orm.**undefer**(*key*, *\*addl\_attrs*)

Indicate that the given column-oriented attribute should be undeferred, e.g. specified within the SELECT statement of the entity as a whole.

表明给定的面向列的属性应该是不被推迟的，例如 在整个实体的SELECT语句中指定。

The column being undeferred is typically set up on the mapping as a [deferred()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "sqlalchemy.orm.deferred" \o "sqlalchemy.orm.deferred) attribute.

未延迟的列通常作为[deferred()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "sqlalchemy.orm.deferred" \o "sqlalchemy.orm.deferred)属性在映射上进行设置。

This function is part of the [Load](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.Load" \o "sqlalchemy.orm.Load) interface and supports both method-chained and standalone operation.

此功能是Load接口的一部分，支持方法链接和独立操作。

Examples:

*# undefer two columns*session.query(MyClass).options(undefer("col1"), undefer("col2"))

*# undefer all columns specific to a single class using Load + \**session.query(MyClass, MyOtherClass).options(

Load(MyClass).undefer("\*"))

|  |  |
| --- | --- |
| **Parameters:** | * ****key**** – Attribute to be undeferred. * ****~~\*addl\_attrs~~****~~– Deprecated; this option supports the old 0.8 style of specifying a path as a series of attributes, which is now superseded by the method-chained style.~~ |

**See also**

[Deferred Column Loading](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "deferred)

[orm.defer()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "sqlalchemy.orm.defer" \o "sqlalchemy.orm.defer)

[orm.undefer\_group()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "sqlalchemy.orm.undefer_group" \o "sqlalchemy.orm.undefer_group)

sqlalchemy.orm.**undefer\_group**(*name*)

Indicate that columns within the given deferred group name should be undeferred.

The columns being undeferred are set up on the mapping as [deferred()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "sqlalchemy.orm.deferred" \o "sqlalchemy.orm.deferred) attributes and include a "group" name.

指示给定的延迟组名称中的列应该不被延期。

未映射的列在映射上设置为deferred()属性，并包括“组”名称。

E.g:

session.query(MyClass).options(undefer\_group("large\_attrs"))

To undefer a group of attributes on a related entity, the path can be spelled out using relationship loader options, such as [orm.defaultload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.defaultload" \o "sqlalchemy.orm.defaultload):

要取消相关实体上的一组属性，可以使用关系加载器选项（例如orm.defaultload()）来将该路径拼出：

session.query(MyClass).options(

defaultload("someattr").undefer\_group("large\_attrs"))

*Changed in version 0.9.0:*[orm.undefer\_group()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "sqlalchemy.orm.undefer_group" \o "sqlalchemy.orm.undefer_group) is now specific to a particiular entity load path.orm.undefer\_group()现在是特定于分支实体加载路径。

**See also**

[Deferred Column Loading](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "deferred)

[orm.defer()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "sqlalchemy.orm.defer" \o "sqlalchemy.orm.defer)

[orm.undefer()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "sqlalchemy.orm.undefer" \o "sqlalchemy.orm.undefer)

4.1.2 Column Bundles

The [Bundle](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Bundle" \o "sqlalchemy.orm.query.Bundle) may be used to query for groups of columns under one namespace.

Bundle可用于查询一个命名空间下的一组列。

*New in version 0.9.0.*

The bundle allows columns to be grouped together:

该捆绑允许列分组在一起：

**from** **sqlalchemy.orm** **import** Bundle

bn = Bundle('mybundle', MyClass.data1, MyClass.data2)

**for** row **in** session.query(bn).filter(bn.c.data1 == 'd1'):

print(row.mybundle.data1, row.mybundle.data2)

The bundle can be subclassed to provide custom behaviors when results are fetched. The method [Bundle.create\_row\_processor()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Bundle.create_row_processor" \o "sqlalchemy.orm.query.Bundle.create_row_processor) is given the [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) and a set of "row processor" functions at query execution time; these processor functions when given a result row will return the individual attribute value, which can then be adapted into any kind of return data structure. Below illustrates replacing the usual [KeyedTuple](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.util.KeyedTuple" \o "sqlalchemy.util.KeyedTuple) return structure with a straight Python dictionary:

捆绑可以被子类化，以便在获取结果时提供自定义行为。 在查询执行时，[Bundle.create\_row\_processor()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Bundle.create_row_processor" \o "sqlalchemy.orm.query.Bundle.create_row_processor)方法给出了[Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query)和一组“行处理器”函数; 当给定结果行时，这些处理器功能将返回单个属性值，然后可以将其适用于任何种类的返回数据结构。 下面说明用一个简单的Python字典替换通常的[KeyedTuple](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.util.KeyedTuple" \o "sqlalchemy.util.KeyedTuple)返回结构：

**from** **sqlalchemy.orm** **import** Bundle

**class** **DictBundle**(Bundle):

**def** create\_row\_processor(self, query, procs, labels):

*"""Override create\_row\_processor to return values as dictionaries"""*

**def** proc(row):

**return** dict(

zip(labels, (proc(row) **for** proc **in** procs))

)

**return** proc

*Changed in version 1.0:*

The proc() callable passed to the create\_row\_processor() method of custom [Bundle](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Bundle" \o "sqlalchemy.orm.query.Bundle) classes now accepts only a single "row" argument.

A result from the above bundle will return dictionary values:

传递给custom [Bundle](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Bundle" \o "sqlalchemy.orm.query.Bundle)类的create\_row\_processor()方法的proc()可调用现在只接受一个“row”参数。

上述捆绑的结果将返回字典值：

bn = DictBundle('mybundle', MyClass.data1, MyClass.data2)

**for** row **in** session.query(bn).filter(bn.c.data1 == 'd1'):

print(row.mybundle['data1'], row.mybundle['data2'])

The [Bundle](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Bundle" \o "sqlalchemy.orm.query.Bundle) construct is also integrated into the behavior of [composite()](http://docs.sqlalchemy.org/en/rel_1_1/orm/composites.html" \l "sqlalchemy.orm.composite" \o "sqlalchemy.orm.composite), where it is used to return composite attributes as objects when queried as individual attributes.

[Bundle](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Bundle" \o "sqlalchemy.orm.query.Bundle)结构也集成到[composite()](http://docs.sqlalchemy.org/en/rel_1_1/orm/composites.html" \l "sqlalchemy.orm.composite" \o "sqlalchemy.orm.composite)的行为中，在查询时作为单独属性用于将复合属性返回为对象。

## 4.2 Relationship Loading Techniques

A big part of SQLAlchemy is providing a wide range of control over how related objects get loaded when querying. By "related objects" we refer to collections or scalar associations configured on a mapper using [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship). This behavior can be configured at mapper construction time using the [relationship.lazy](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.lazy" \o "sqlalchemy.orm.relationship)parameter to the [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) function, as well as by using options with the [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) object.

SQLAlchemy的很大一部分是对查询中相关对象加载的方式提供了广泛的控制。 通过"相关对象"，我们引用使用relationship()在映射器上配置的集合或标量关联。 可以在mapper构建时使用relationship()函数的relationship.lazyparameter以及使用Query对象的选项来配置此行为。

The loading of relationships falls into three categories; ****lazy**** loading, ****eager**** loading, and ****no**** loading. Lazy loading refers to objects are returned from a query without the related objects loaded at first. When the given collection or reference is first accessed on a particular object, an additional SELECT statement is emitted such that the requested collection is loaded.

关系的加载分为三类： 懒加载，预加载，无加载。 懒加载指的是从查询返回的对象，而相关对象首先被加载。 当给定的集合或引用首先在特定对象上访问时，会发出附加的SELECT语句，以便加载所请求的集合。

Eager loading refers to objects returned from a query with the related collection or scalar reference already loaded up front. The [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) achieves this either by augmenting the SELECT statement it would normally emit with a JOIN to load in related rows simultaneously, or by emitting additional SELECT statements after the primary one to load collections or scalar references at once.

预加载是指从查询返回的对象，相关的集合或标量引用已经被加载到前面。 Query可以通过增加通常使用JOIN发送的SELECT语句来同时加载相关行的SELECT语句，也可以通过在主函数之后发出另外的SELECT语句来一次加载集合或标量引用。

"No" loading refers to the disabling of loading on a given relationship, either that the attribute is empty and is just never loaded, or that it raises an error when it is accessed, in order to guard against unwanted lazy loads.

"否"加载是指在给定关系上禁用加载，无论该属性是空的，并且刚刚被加载，或者在访问时引发错误，以防止不必要的延迟加载。

The primary forms of relationship loading are:

* ****lazy loading**** - available via lazy='select' or the [lazyload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.lazyload" \o "sqlalchemy.orm.lazyload) option, this is the form of loading that emits a SELECT statement at attribute access time to lazily load a related reference on a single object at a time. Lazy loading is detailed at [Lazy Loading](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "lazy-loading).
* 懒加载 - 可通过lazy ='select'或lazyload()选项提供，这是在属性访问时发出SELECT语句的加载形式，以便在单个对象上逐个加载相关引用。 Lazy Loading的详细资料
* ****joined loading**** - available via lazy='joined' or the [joinedload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.joinedload" \o "sqlalchemy.orm.joinedload) option, this form of loading applies a JOIN to the given SELECT statement so that related rows are loaded in the same result set. Joined eager loading is detailed at [Joined Eager Loading](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "joined-eager-loading).
* 联合加载 - 通过lazy ='joined'或joinedload()选项可用，此形式的加载将JOIN应用于给定的SELECT语句，以便将相关行加载到同一结果集中。 加入渴望加载在加入加载中详细。
* ****subquery loading**** - available via lazy='subquery' or the [subqueryload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.subqueryload" \o "sqlalchemy.orm.subqueryload) option, this form of loading emits a second SELECT statement which re-states the original query embedded inside of a subquery, then JOINs that subquery to the related table to be loaded to load all members of related collections / scalar references at once. Subquery eager loading is detailed at [Subquery Eager Loading](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "subquery-eager-loading).
* ****raise loading**** - available via lazy='raise', lazy='raise\_sql', or the [raiseload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.raiseload" \o "sqlalchemy.orm.raiseload) option, this form of loading is triggered at the same time a lazy load would normally occur, except it raises an ORM exception in order to guard against the application making unwanted lazy loads. An introduction to raise loading is at [Preventing unwanted lazy loads using raiseload](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "prevent-lazy-with-raiseload).
* ****no loading**** - available via lazy='noload', or the [noload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.noload" \o "sqlalchemy.orm.noload) option; this loading style turns the attribute into an empty attribute that will never load or have any loading effect. "noload" is a fairly uncommon loader option.

4.2.1 Configuring Loader Strategies at Mapping Time

The loader strategy for a particular relationship can be configured at mapping time to take place in all cases where an object of the mapped type is loaded, in the absense of any query-level options that modify it. This is configured using the [relationship.lazy](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.lazy" \o "sqlalchemy.orm.relationship) parameter to [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship); common values for this parameter include "select", "joined", and "subquery".

For example, to configure a relationship to use joined eager loading when the parent object is queried:

**class** **Parent**(Base):

\_\_tablename\_\_ = 'parent'

id = Column(Integer, primary\_key=**True**)

children = relationship("Child", lazy='joined')

Above, whenever a collection of Parent objects are loaded, each Parent will also have its children collection populated, using rows fetched by adding a JOIN to the query for Parent objects. See [Joined Eager Loading](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "joined-eager-loading) for background on this style of loading.

The default value of the [relationship.lazy](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.lazy" \o "sqlalchemy.orm.relationship) argument is "select", which indicates lazy loading. See [Lazy Loading](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "lazy-loading) for further background.

4.2.2 Controlling Loading via Options

The other, and possibly more common way to configure loading strategies is to set them up on a per-query basis against specific attributes. Very detailed control over relationship loading is available using loader options; the most common are [joinedload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.joinedload" \o "sqlalchemy.orm.joinedload), [subqueryload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.subqueryload" \o "sqlalchemy.orm.subqueryload), and [lazyload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.lazyload" \o "sqlalchemy.orm.lazyload). The option accepts either the string name of an attribute against a parent, or for greater specificity can accommodate a class-bound attribute directly:

*# set children to load lazily*session.query(Parent).options(lazyload('children')).all()

*# same, using class-bound attribute*session.query(Parent).options(lazyload(Parent.children)).all()

*# set children to load eagerly with a join*session.query(Parent).options(joinedload('children')).all()

The loader options can also be "chained" using ****method chaining**** to specify how loading should occur further levels deep:

session.query(Parent).options(

joinedload(Parent.children).

subqueryload(Child.subelements)).all()

Chained loader options can be applied against a "lazy" loaded collection. This means that when a collection or association is lazily loaded upon access, the specified option will then take effect:

session.query(Parent).options(

lazyload(Parent.children).

subqueryload(Child.subelements)).all()

Above, the query will return Parent objects without the children collections loaded. When the children collection on a particular Parent object is first accessed, it will lazy load the related objects, but additionally apply eager loading to the subelements collection on each member of children.

Using method chaining, the loader style of each link in the path is explicitly stated. To navigate along a path without changing the existing loader style of a particular attribute, the [defaultload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.defaultload" \o "sqlalchemy.orm.defaultload) method/function may be used:

session.query(A).options(

defaultload("atob").

joinedload("btoc")).all()

### 4.2.3 Lazy Loading

By default, all inter-object relationships are ****lazy loading****. The scalar or collection attribute associated with a [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) contains a trigger which fires the first time the attribute is accessed. This trigger typically issues a SQL call at the point of access in order to load the related object or objects:

>>> jack.addresses

SELECT

addresses.id AS addresses\_id,

addresses.email\_address AS addresses\_email\_address,

addresses.user\_id AS addresses\_user\_id

FROM addresses

WHERE ? = addresses.user\_id

[5]

[<Address(u'jack@google.com')>, <Address(u'j25@yahoo.com')>]

The one case where SQL is not emitted is for a simple many-to-one relationship, when the related object can be identified by its primary key alone and that object is already present in the current [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session). For this reason, while lazy loading can be expensive for related collections, in the case that one is loading lots of objects with simple many-to-ones against a relatively small set of possible target objects, lazy loading may be able to refer to these objects locally without emitting as many SELECT statements as there are parent objects.

This default behavior of "load upon attribute access" is known as "lazy" or "select" loading - the name "select" because a "SELECT" statement is typically emitted when the attribute is first accessed.

Lazy loading can be enabled for a given attribute that is normally configured in some other way using the [lazyload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.lazyload" \o "sqlalchemy.orm.lazyload) loader option:

**from** **sqlalchemy.orm** **import** lazyload

*# force lazy loading for an attribute that is set to# load some other way normally*session.query(User).options(lazyload(User.addresses))

### Preventing unwanted lazy loads using raiseload

The [lazyload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.lazyload" \o "sqlalchemy.orm.lazyload) strategy produces an effect that is one of the most common issues referred to in object relational mapping; the [N plus one problem](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-n-plus-one-problem), which states that for any N objects loaded, accessing their lazy-loaded attributes means there will be N+1 SELECT statements emitted. In SQLAlchemy, the usual mitigation for the N+1 problem is to make use of its very capable eager load system. However, eager loading requires that the attributes which are to be loaded be specified with the[Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) up front. The problem of code that may access other attributes that were not eagerly loaded, where lazy loading is not desired, may be addressed using the [raiseload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.raiseload" \o "sqlalchemy.orm.raiseload) strategy; this loader strategy replaces the behavior of lazy loading with an informative error being raised:

lazyload()策略产生一个效果，它是对象关系映射中引用的最常见的问题之一; N加一个问题，其中指出，对于任何加载的N对象，访问它们的惰性加载属性意味着将发出N + 1个SELECT语句。 在SQLAlchemy中，N + 1问题的通常的缓解是利用其非常有能力的热切负载系统。 然而，热切的加载要求使用theQuery在前面指定要加载的属性。 可能使用raiseload()策略来解决可能访问其他不被加载的属性的代码问题，其中不需要延迟加载; 这个装载机策略取代了延迟加载的行为，提出了一个内容丰富的错误：

**from** **sqlalchemy.orm** **import** raiseloadsession.query(User).options(raiseload(User.addresses))

Above, a User object loaded from the above query will not have the .addresses collection loaded; if some code later on attempts to access this attribute, an ORM exception is raised.

[raiseload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.raiseload" \o "sqlalchemy.orm.raiseload) may be used with a so-called "wildcard" specifier to indicate that all relationships should use this strategy. For example, to set up only one attribute as eager loading, and all the rest as raise:

session.query(Order).options(

joinedload(Order.items), raiseload('\*'))

The above wildcard will apply to ****all**** relationships not just on Order besides items, but all those on the Item objects as well. To set up [raiseload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.raiseload" \o "sqlalchemy.orm.raiseload) for only the Order objects, specify a full path with [orm.Load](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.Load" \o "sqlalchemy.orm.Load):

**from** **sqlalchemy.orm** **import** Load

session.query(Order).options(

joinedload(Order.items), Load(Order).raiseload('\*'))

Conversely, to set up the raise for just the Item objects:

session.query(Order).options(

joinedload(Order.items).raiseload('\*'))

**See also**

[Wildcard Loading Strategies](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "wildcard-loader-strategies)

4.2.4 Joined Eager Loading

Joined eager loading is the most fundamental style of eager loading in the ORM. It works by connecting a JOIN (by default a LEFT OUTER join) to the SELECT statement emitted by a [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) and populates the target scalar/collection from the same result set as that of the parent.

At the mapping level, this looks like:

**class** **Address**(Base):

*# ...*

user = relationship(User, lazy="joined")

Joined eager loading is usually applied as an option to a query, rather than as a default loading option on the mapping, in particular when used for collections rather than many-to-one-references. This is achieved using the [joinedload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.joinedload" \o "sqlalchemy.orm.joinedload) loader option:

>>> jack = session.query(User).\... options(joinedload(User.addresses)).\... filter\_by(name='jack').all()

SELECT

addresses\_1.id AS addresses\_1\_id,

addresses\_1.email\_address AS addresses\_1\_email\_address,

addresses\_1.user\_id AS addresses\_1\_user\_id,

users.id AS users\_id, users.name AS users\_name,

users.fullname AS users\_fullname,

users.password AS users\_password

FROM users

LEFT OUTER JOIN addresses AS addresses\_1

ON users.id = addresses\_1.user\_id

WHERE users.name = ?

['jack']

The JOIN emitted by default is a LEFT OUTER JOIN, to allow for a lead object that does not refer to a related row. For an attribute that is guaranteed to have an element, such as a many-to-one reference to a related object where the referencing foreign key is NOT NULL, the query can be made more efficient by using an inner join; this is available at the mapping level via the [relationship.innerjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.innerjoin" \o "sqlalchemy.orm.relationship) flag:

**class** **Address**(Base):

*# ...*

user\_id = Column(ForeignKey('users.id'), nullable=**False**)

user = relationship(User, lazy="joined", innerjoin=**True**)

At the query option level, via the [joinedload.innerjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.strategy_options.Load.joinedload.params.innerjoin" \o "sqlalchemy.orm.strategy_options.Load.joinedload) flag:

session.query(Address).options(

joinedload(Address.user, innerjoin=**True**))

The JOIN will right-nest itself when applied in a chain that includes an OUTER JOIN:

>>> session.query(User).options(... joinedload(User.addresses).... joinedload(Address.widgets, innerjoin=True)).all()

SELECT

widgets\_1.id AS widgets\_1\_id,

widgets\_1.name AS widgets\_1\_name,

addresses\_1.id AS addresses\_1\_id,

addresses\_1.email\_address AS addresses\_1\_email\_address,

addresses\_1.user\_id AS addresses\_1\_user\_id,

users.id AS users\_id, users.name AS users\_name,

users.fullname AS users\_fullname,

users.password AS users\_password

FROM users

LEFT OUTER JOIN (

addresses AS addresses\_1 JOIN widgets AS widgets\_1 ON

addresses\_1.widget\_id = widgets\_1.id

) ON users.id = addresses\_1.user\_id

On older versions of SQLite, the above nested right JOIN may be re-rendered as a nested subquery. Older versions of SQLAlchemy would convert right-nested joins into subuqeries in all cases.

### Joined eager loading and result set batching

A central concept of joined eager loading when applied to collections is that the [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) object must de-duplicate rows against the leading entity being queried. Such as above, if the User object we loaded referred to three Address objects, the result of the SQL statement would have had three rows; yet the [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) returns only one User object. As additional rows are received for a User object just loaded in a previous row, the additional columns that refer to new Address objects are directed into additional results within the User.addresses collection of that particular object.

应用于集合时，加入预加载的中心概念是Query对象必须根据被查询的主要实体取消重复行。如上所述，如果我们加载的User对象引用了三个Address对象，则SQL语句的结果将有三行;但是Query只返回一个User对象。由于刚刚加载到上一行的用户对象接收到其他行，所以引用新的Address对象的附加列将被引导到该特定对象的User.addresses集合中的其他结果。

This process is very transparent, however does imply that joined eager loading is incompatible with "batched" query results, provided by the [Query.yield\_per()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.yield_per" \o "sqlalchemy.orm.query.Query.yield_per)method, when used for collection loading. Joined eager loading used for scalar references is however compatible with [Query.yield\_per()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.yield_per" \o "sqlalchemy.orm.query.Query.yield_per). The [Query.yield\_per()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.yield_per" \o "sqlalchemy.orm.query.Query.yield_per) method will result in an exception thrown if a collection based joined eager loader is in play.

这个过程非常透明，但是这意味着加入的加载与[Query.yield\_per()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.yield_per" \o "sqlalchemy.orm.query.Query.yield_per)方法提供的“批量”查询结果不兼容，用于集合加载。然而，加入用于标量引用的加载加载与[Query.yield\_per()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.yield_per" \o "sqlalchemy.orm.query.Query.yield_per)兼容。 [Query.yield\_per()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.yield_per" \o "sqlalchemy.orm.query.Query.yield_per)方法将导致如果基于集合的连接的eager加载程序正在播放时抛出异常。

To "batch" queries with arbitrarily large sets of result data while maintaining compatibility with collection-based joined eager loading, emit multiple SELECT statements, each referring to a subset of rows using the WHERE clause, e.g. windowing.

要使具有任意大量结果数据的“批处理”查询同时保持与基于收集的连接的加载加载的兼容性，则会发出多个SELECT语句，每个SELECT语句都使用WHERE子句引用行的子集，例如。窗口。

### The Zen of Joined Eager Loading

Since joined eager loading seems to have many resemblences to the use of [Query.join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join), it often produces confusion as to when and how it should be used. It is critical to understand the distinction that while [Query.join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join) is used to alter the results of a query, [joinedload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.joinedload" \o "sqlalchemy.orm.joinedload) goes through great lengths to ****not**** alter the results of the query, and instead hide the effects of the rendered join to only allow for related objects to be present.

由于加入的热心加载似乎与Query.join()的使用有很多相似之处，因此通常会产生混淆何时以及如何使用它。了解Query.join()用于更改查询结果的区别至关重要，因此，joinload()非常长时间地不会更改查询的结果，而是将渲染的连接的效果隐藏起来只允许存在相关对象。

The philosophy behind loader strategies is that any set of loading schemes can be applied to a particular query, and *the results don't change* - only the number of SQL statements required to fully load related objects and collections changes. A particular query might start out using all lazy loads. After using it in context, it might be revealed that particular attributes or collections are always accessed, and that it would be more efficient to change the loader strategy for these. The strategy can be changed with no other modifications to the query, the results will remain identical, but fewer SQL statements would be emitted. In theory (and pretty much in practice), nothing you can do to the [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) would make it load a different set of primary or related objects based on a change in loader strategy.

加载器策略背后的理念是，任何一套加载方案都可以应用于特定查询，结果不会改变 - 只有完全加载相关对象和集合才需要的SQL语句数量发生变化。特定的查询可能开始使用所有的惰性负载。在上下文中使用它之后，可能会显示特定的属性或集合总是被访问，更改这些加载器策略会更有效率。该策略可以更改，而不需要对查询进行任何其他修改，结果将保持不变，但会发出较少的SQL语句。在理论上(实际上很多)，对Query可以做的任何事情都将使得根据加载器策略的变化加载一组不同的主要或相关对象。

How [joinedload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.joinedload" \o "sqlalchemy.orm.joinedload) in particular achieves this result of not impacting entity rows returned in any way is that it creates an anonymous alias of the joins it adds to your query, so that they can't be referenced by other parts of the query. For example, the query below uses [joinedload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.joinedload" \o "sqlalchemy.orm.joinedload) to create a LEFT OUTER JOIN from users to addresses, however the ORDER BY added against Address.email\_address is not valid - the Address entity is not named in the query:

对于不影响以任何方式返回的实体行的结果，joinedload()特别实现了这一结果，它创建了一个匿名添加到您的查询中的别名，这样它们不能被查询的其他部分引用。例如，下面的查询使用joinedload()从用户到地址创建一个LEFT OUTER JOIN，但是对于Address.email\_address添加的ORDER BY无效 - 查询中没有命名Address实体：

>>> jack = session.query(User).\

... options(joinedload(User.addresses)).\

... filter(User.name=='jack').\

... order\_by(Address.email\_address).all()

SELECT

addresses\_1.id AS addresses\_1\_id,

addresses\_1.email\_address AS addresses\_1\_email\_address,

addresses\_1.user\_id AS addresses\_1\_user\_id,

users.id AS users\_id,

users.name AS users\_name,

users.fullname AS users\_fullname,

users.password AS users\_password

FROM users

LEFT OUTER JOIN addresses AS addresses\_1

ON users.id = addresses\_1.user\_id

WHERE users.name = ?

ORDER BY addresses.email\_address <-- this part is wrong !

['jack']

Above, ORDER BY addresses.email\_address is not valid since addresses is not in the FROM list. The correct way to load the User records and order by email address is to use [Query.join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join):

以上，ORDER BY addresses.email\_address无效，因为addresses不在FROM列表中。 通过电子邮件地址加载User记录和订单的正确方法是使用[Query.join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join)：

>>> jack = session.query(User).\... join(User.addresses).\... filter(User.name=='jack').\... order\_by(Address.email\_address).all()

SELECT

users.id AS users\_id,

users.name AS users\_name,

users.fullname AS users\_fullname,

users.password AS users\_password

FROM users

JOIN addresses ON users.id = addresses.user\_id

WHERE users.name = ?

ORDER BY addresses.email\_address

['jack']

The statement above is of course not the same as the previous one, in that the columns from addresses are not included in the result at all. We can add [joinedload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.joinedload" \o "sqlalchemy.orm.joinedload) back in, so that there are two joins - one is that which we are ordering on, the other is used anonymously to load the contents of the User.addresses collection:

上面的声明当然不一样，因为addresses中的列根本不包括在结果中。 我们添加[joinedload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.joinedload" \o "sqlalchemy.orm.joinedload)，以便有两个连接 - 一个是我们正在排序的，另一个是匿名使用来加载User.addresses集合的内容：

>>> jack = session.query(User).\

... join(User.addresses).\

... options(joinedload(User.addresses)).\

... filter(User.name=='jack').\

... order\_by(Address.email\_address).all()

SELECT

addresses\_1.id AS addresses\_1\_id,

addresses\_1.email\_address AS addresses\_1\_email\_address,

addresses\_1.user\_id AS addresses\_1\_user\_id,

users.id AS users\_id, users.name AS users\_name,

users.fullname AS users\_fullname,

users.password AS users\_password

FROM users JOIN addresses

ON users.id = addresses.user\_id

LEFT OUTER JOIN addresses AS addresses\_1

ON users.id = addresses\_1.user\_id

WHERE users.name = ?

ORDER BY addresses.email\_address

['jack']

What we see above is that our usage of [Query.join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join) is to supply JOIN clauses we'd like to use in subsequent query criterion, whereas our usage of[joinedload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.joinedload" \o "sqlalchemy.orm.joinedload) only concerns itself with the loading of the User.addresses collection, for each User in the result. In this case, the two joins most probably appear redundant - which they are. If we wanted to use just one JOIN for collection loading as well as ordering, we use the [contains\_eager()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.contains_eager" \o "sqlalchemy.orm.contains_eager) option, described in [Routing Explicit Joins/Statements into Eagerly Loaded Collections](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "contains-eager) below. But to see why [joinedload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.joinedload" \o "sqlalchemy.orm.joinedload) does what it does, consider if we were ****filtering**** on a particular Address:

我们上面看到的是，我们使用Query.join()是提供我们希望在后续查询条件中使用的JOIN子句，而我们使用的joinedload()只关心User.addresses集合的加载， 结果中的每个用户。 在这种情况下，两个连接最有可能是多余的 - 它们是多余的。 如果我们只想使用一个JOIN来进行收集加载和排序，那么我们使用在下面的将Routing Explicit Joins / Statements描述为Eagerly Loaded Collections中的contains\_eager()选项。 但是要看看为什么joinload()会做它，请考虑我们是否在特定的地址进行过滤：

>>> jack = session.query(User).\

... join(User.addresses).\

... options(joinedload(User.addresses)).\

... filter(User.name=='jack').\

... [filter(Address.email\_address=='someaddress@foo.com').\](mailto:filter(Address.email_address=='someaddress@foo.com')./)

... all()

SELECT

addresses\_1.id AS addresses\_1\_id,

addresses\_1.email\_address AS addresses\_1\_email\_address,

addresses\_1.user\_id AS addresses\_1\_user\_id,

users.id AS users\_id, users.name AS users\_name,

users.fullname AS users\_fullname,

users.password AS users\_password

FROM users JOIN addresses

ON users.id = addresses.user\_id

LEFT OUTER JOIN addresses AS addresses\_1

ON users.id = addresses\_1.user\_id

WHERE users.name = ? AND addresses.email\_address = ?

['jack', 'someaddress@foo.com']

Above, we can see that the two JOINs have very different roles. One will match exactly one row, that of the join of User and Address whereAddress.email\_address=='someaddress@foo.com'. The other LEFT OUTER JOIN will match *all* Address rows related to User, and is only used to populate the User.addresses collection, for those User objects that are returned.

以上，我们可以看到两个JOIN有很大的不同。 一个将匹配一行，即User和Address在这Address.email\_address=='someaddress@foo.com'的连接。 另一个LEFT OUTER JOIN将匹配与User相关的所有Address行，仅用于填充User.addresses集合，以返回的User对象。

By changing the usage of [joinedload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.joinedload" \o "sqlalchemy.orm.joinedload) to another style of loading, we can change how the collection is loaded completely independently of SQL used to retrieve the actual User rows we want. Below we change [joinedload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.joinedload" \o "sqlalchemy.orm.joinedload) into [subqueryload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.subqueryload" \o "sqlalchemy.orm.subqueryload):

通过将[joinedload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.joinedload" \o "sqlalchemy.orm.joinedload)的使用更改为另一种加载方式，我们可以更改如何完全加载集合，而不必使用SQL来检索所需的实际User行。 下面我们将[joinedload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.joinedload" \o "sqlalchemy.orm.joinedload)更改为[subqueryload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.subqueryload" \o "sqlalchemy.orm.subqueryload)：

>>> jack = session.query(User).\

... join(User.addresses).\

... options(subqueryload(User.addresses)).\

... filter(User.name=='jack').\

... [filter(Address.email\_address=='someaddress@foo.com').\](mailto:filter(Address.email_address=='someaddress@foo.com')./)

... all()

SELECT

users.id AS users\_id,

users.name AS users\_name,

users.fullname AS users\_fullname,

users.password AS users\_password

FROM users

JOIN addresses ON users.id = addresses.user\_id

WHERE

users.name = ?

AND addresses.email\_address = ?

['jack', 'someaddress@foo.com']

# ... subqueryload() emits a SELECT in order

# to load all address records ...

When using joined eager loading, if the query contains a modifier that impacts the rows returned externally to the joins, such as when using DISTINCT, LIMIT, OFFSET or equivalent, the completed statement is first wrapped inside a subquery, and the joins used specifically for joined eager loading are applied to the subquery. SQLAlchemy's joined eager loading goes the extra mile, and then ten miles further, to absolutely ensure that it does not affect the end result of the query, only the way collections and related objects are loaded, no matter what the format of the query is.

当使用加入的加速加载时，如果查询包含影响连接到外部的行的修饰符，例如在使用DISTINCT，LIMIT，OFFSET或等效文件时，完成的语句首先被包含在子查询中，并且专门用于 加入的加载加载到子查询。 SQLAlchemy加入的渴望加载额外的一英里，然后进一步增加十英里，绝对确保它不影响查询的最终结果，只有加载集合和相关对象的方式，无论查询的格式如何。

**See also**

[Routing Explicit Joins/Statements into Eagerly Loaded Collections](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "contains-eager) - using [contains\_eager()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.contains_eager" \o "sqlalchemy.orm.contains_eager)

4.2.5 Subquery Eager Loading

Subqueryload eager loading is configured in the same manner as that of joined eager loading; for the [relationship.lazy](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.lazy" \o "sqlalchemy.orm.relationship) parameter, we would specify "subquery" rather than "joined", and for the option we use the [subqueryload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.subqueryload" \o "sqlalchemy.orm.subqueryload) option rather than the [joinedload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.joinedload" \o "sqlalchemy.orm.joinedload) option.

子查询加载加载与加入的加载加载相同; 对于[relationship.lazy](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.lazy" \o "sqlalchemy.orm.relationship)参数，我们将指定“subquery”而不是“joined”，对于该选项，我们使用[subqueryload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.subqueryload" \o "sqlalchemy.orm.subqueryload)选项而不是[joinedload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.joinedload" \o "sqlalchemy.orm.joinedload)选项。

The operation of subquery eager loading is to emit a second SELECT statement for each relationship to be loaded, across all result objects at once. This SELECT statement refers to the original SELECT statement, wrapped inside of a subquery, so that we retrieve the same list of primary keys for the primary object being returned, then link that to the sum of all the collection members to load them at once:

子查询加载的操作是针对要加载的每个关系发出一个第二个SELECT语句，同时跨所有结果对象。 这个SELECT语句是指原始的SELECT语句，它包含在一个子查询中，以便我们检索与返回的主对象相同的主键列表，然后链接到所有收集成员的总和，以便一次加载它们：

>>> jack = session.query(User).\

... options(subqueryload(User.addresses)).\

... filter\_by(name='jack').all()

SELECT

users.id AS users\_id,

users.name AS users\_name,

users.fullname AS users\_fullname,

users.password AS users\_password

FROM users

WHERE users.name = ?

('jack',)

SELECT

addresses.id AS addresses\_id,

addresses.email\_address AS addresses\_email\_address,

addresses.user\_id AS addresses\_user\_id,

anon\_1.users\_id AS anon\_1\_users\_id

FROM (

SELECT users.id AS users\_id

FROM users

WHERE users.name = ?) AS anon\_1

JOIN addresses ON anon\_1.users\_id = addresses.user\_id

ORDER BY anon\_1.users\_id, addresses.id

('jack',)

The subqueryload strategy has many advantages over joined eager loading in the area of loading collections. First, it allows the original query to proceed without changing it at all, not introducing in particular a LEFT OUTER JOIN that may make it less efficient. Secondly, it allows for many collections to be eagerly loaded without producing a single query that has many JOINs in it, which can be even less efficient; each relationship is loaded in a fully separate query. Finally, because the additional query only needs to load the collection items and not the lead object, it can use an inner JOIN in all cases for greater query efficiency.

子加载策略与加载集合领域的加载加载有很多优点。首先，它允许原始查询在不改变的情况下继续进行，而不是特别引入可能使其效率较低的LEFT OUTER JOIN。其次，它允许许多集合被加载，而不会产生一个单个查询，其中有很多JOIN，这可能甚至不太有效;每个关系都被加载在完全独立的查询中。最后，因为附加查询只需要加载收集项而不是引导对象，所以在所有情况下都可以使用内部JOIN来提高查询效率。

Disadvantages of subqueryload include that the complexity of the original query is transferred to the relationship queries, which when combined with the use of a subquery, can on some backends in some cases (notably MySQL) produce significantly slow queries. Additionally, the subqueryload strategy can only load the full contents of all collections at once, is therefore incompatible with "batched" loading supplied by [Query.yield\_per()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.yield_per" \o "sqlalchemy.orm.query.Query.yield_per), both for collection and scalar relationships.

子查询负载的缺点包括将原始查询的复杂性转移到关系查询中，当与某些子事务(特别是MySQL)相关联的情况下，在使用子查询的情况下可以产生显着慢的查询。此外，子查询加载策略只能一次加载所有集合的全部内容，因此与[Query.yield\_per()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.yield_per" \o "sqlalchemy.orm.query.Query.yield_per)提供的“批量”加载不兼容，用于集合和标量关系。

### The Importance of Ordering

A query which makes use of [subqueryload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.subqueryload" \o "sqlalchemy.orm.subqueryload) in conjunction with a limiting modifier such as [Query.first()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.first" \o "sqlalchemy.orm.query.Query.first), [Query.limit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.limit" \o "sqlalchemy.orm.query.Query.limit), or [Query.offset()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.offset" \o "sqlalchemy.orm.query.Query.offset) should ****always**** include [Query.order\_by()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.order_by" \o "sqlalchemy.orm.query.Query.order_by) against unique column(s) such as the primary key, so that the additional queries emitted by [subqueryload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.subqueryload" \o "sqlalchemy.orm.subqueryload) include the same ordering as used by the parent query. Without it, there is a chance that the inner query could return the wrong rows:

使用[subqueryload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.subqueryload" \o "sqlalchemy.orm.subqueryload)与限制修饰符(如[Query.first()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.first" \o "sqlalchemy.orm.query.Query.first)，[Query.limit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.limit" \o "sqlalchemy.orm.query.Query.limit)或[Query.offset()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.offset" \o "sqlalchemy.orm.query.Query.offset))一起使用的查询应始终包含[Query.order\_by()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.order_by" \o "sqlalchemy.orm.query.Query.order_by)与唯一列等 作为主键，以便[subqueryload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.subqueryload" \o "sqlalchemy.orm.subqueryload)发出的附加查询包含与父查询使用的相同的排序。 没有它，内部查询有可能返回错误的行：

*# incorrect, no ORDER BY*session.query(User).options(

subqueryload(User.addresses)).first()

*# incorrect if User.name is not unique*session.query(User).options(

subqueryload(User.addresses)).order\_by(User.name).first()

*# correct*session.query(User).options(

subqueryload(User.addresses)).order\_by(User.name, User.id).first()

**See also**

[Why is ORDER BY required with LIMIT (especially with subqueryload())?](http://docs.sqlalchemy.org/en/rel_1_1/faq/ormconfiguration.html" \l "faq-subqueryload-limit-sort) - detailed example

4.2.4 What Kind of Loading to Use ?

Which type of loading to use typically comes down to optimizing the tradeoff between number of SQL executions, complexity of SQL emitted, and amount of data fetched. Lets take two examples, a [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) which references a collection, and a [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) that references a scalar many-to-one reference.

通常使用哪种类型的加载来优化SQL执行数量，发出的SQL的复杂性和获取的数据量之间的权衡。 让我们来看两个例子，一个引用一个collection的[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)和一个引用标量多对一引用的[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)。

* One to Many Collection
* When using the default lazy loading, if you load 100 objects, and then access a collection on each of them, a total of 101 SQL statements will be emitted, although each statement will typically be a simple SELECT without any joins.当使用默认的延迟加载时，如果加载100个对象，然后访问每个对象的集合，则将发出总共101个SQL语句，尽管每个语句通常都是一个简单的SELECT，而不需要任何连接。
* When using joined loading, the load of 100 objects and their collections will emit only one SQL statement. However, the total number of rows fetched will be equal to the sum of the size of all the collections, plus one extra row for each parent object that has an empty collection. Each row will also contain the full set of columns represented by the parents, repeated for each collection item - SQLAlchemy does not re-fetch these columns other than those of the primary key, however most DBAPIs (with some exceptions) will transmit the full data of each parent over the wire to the client connection in any case. Therefore joined eager loading only makes sense when the size of the collections are relatively small. The LEFT OUTER JOIN can also be performance intensive compared to an INNER join.当使用连接加载时，100个对象及其集合的加载将只会发出一个SQL语句。 但是，获取的总行数将等于所有集合的大小的总和，以及每个具有空集合的父对象的另外一行。 每行还将包含父项代表的完整列，为每个集合项重复 - SQLAlchemy不会重新获取除主键以外的列，但是大多数DBAPI(除了一些例外)将传输完整数据 在任何情况下，每个父母通过线路连接到客户端连接。 因此，当集合的大小相对较小时，加入渴望加载才有意义。 与INNER连接相比，LEFT OUTER JOIN也可以是性能密集型。
* When using subquery loading, the load of 100 objects will emit two SQL statements. The second statement will fetch a total number of rows equal to the sum of the size of all collections. An INNER JOIN is used, and a minimum of parent columns are requested, only the primary keys. So a subquery load makes sense when the collections are larger.当使用子查询加载时，100个对象的加载将发出两个SQL语句。 第二个语句将获取总数等于所有集合大小的总和。 使用INNER JOIN，并且请求最少的父列，只有主键。 所以当集合更大时，子查询负载是有意义的。
* When multiple levels of depth are used with joined or subquery loading, loading collections-within- collections will multiply the total number of rows fetched in a cartesian fashion. Both joined and subquery eager loading always join from the original parent class; if loading a collection four levels deep, there will be four JOINs out to the parent.当使用多个级别的深度加入或子查询加载时，加载集合将收集以笛卡尔方式取得的总行数。 加入和子查询加载总是从原来的父类加入; 如果加载一个集合四级深，将有四个JOIN到父级。
* Many to One Reference
* When using the default lazy loading, a load of 100 objects will like in the case of the collection emit as many as 101 SQL statements. However - there is a significant exception to this, in that if the many-to-one reference is a simple foreign key reference to the target's primary key, each reference will be checked first in the current identity map using [Query.get()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.get" \o "sqlalchemy.orm.query.Query.get). So here, if the collection of objects references a relatively small set of target objects, or the full set of possible target objects have already been loaded into the session and are strongly referenced, using the default of lazy='select' is by far the most efficient way to go.当使用默认的惰性加载时，100个对象的加载将在集合的情况下发出多达101个SQL语句。 然而，这是一个很大的例外，因为如果多对一引用是对目标主键的简单外键引用，则将使用[Query.get()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.get" \o "sqlalchemy.orm.query.Query.get)在当前身份映射中首先检查每个引用。 所以在这里，如果对象的集合引用了相对较小的一组目标对象，或者一整套可能的目标对象已被加载到会话中并被强烈引用，则使用默认的lazy ='select'是迄今为止 最有效的方式去。
* When using joined loading, the load of 100 objects will emit only one SQL statement. The join will be a LEFT OUTER JOIN, and the total number of rows will be equal to 100 in all cases. If you know that each parent definitely has a child (i.e. the foreign key reference is NOT NULL), the joined load can be configured with [innerjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.innerjoin" \o "sqlalchemy.orm.relationship) set to True, which is usually specified within the [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship). For a load of objects where there are many possible target references which may have not been loaded already, joined loading with an INNER JOIN is extremely efficient.当使用连接加载时，100个对象的加载将只会发出一个SQL语句。 连接将是一个LEFT OUTER JOIN，在所有情况下，总行数将等于100。 如果你知道每个父对象肯定都有一个小孩(即外键引用不为空)，那么可以通过将[innerjoin](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.innerjoin" \o "sqlalchemy.orm.relationship)设置为True(通常在[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)中指定)来配置连接的负载。 对于可能尚未加载的许多可能目标引用的对象的负载，使用INNER JOIN加载加载非常有效。
* Subquery loading will issue a second load for all the child objects, so for a load of 100 objects there would be two SQL statements emitted. There's probably not much advantage here over joined loading, however, except perhaps that subquery loading can use an INNER JOIN in all cases whereas joined loading requires that the foreign key is NOT NULL.批次加载将为所有子对象发出第二个加载，因此对于100个对象的加载，将发出两个SQL语句。 然而，在加载加载中可能没有太多优点，除了可能在所有情况下，子查询加载可以使用INNER JOIN，而加载加载需要外键为NOT NULL。

4.2.5 Polymorphic Eager Loading

Specification of polymorpic options on a per-eager-load basis is supported. See the section [Eager Loading of Specific or Polymorphic Subtypes](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "eagerloading-polymorphic-subtypes) for examples of the [PropComparator.of\_type()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator.of_type" \o "sqlalchemy.orm.interfaces.PropComparator.of_type) method in conjunction with the [orm.with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic" \o "sqlalchemy.orm.with_polymorphic) function.

支持按照每个热负荷的多义选项的规范。 请参阅与[PropComparator.of\_type()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator.of_type" \o "sqlalchemy.orm.interfaces.PropComparator.of_type)函数一起使用的[orm.with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic" \o "sqlalchemy.orm.with_polymorphic)方法的例子，加载特定或多态子类型。

4.2.6 Wildcard Loading Strategies

Each of [joinedload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.joinedload" \o "sqlalchemy.orm.joinedload), [subqueryload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.subqueryload" \o "sqlalchemy.orm.subqueryload), [lazyload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.lazyload" \o "sqlalchemy.orm.lazyload), [noload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.noload" \o "sqlalchemy.orm.noload), and [raiseload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.raiseload" \o "sqlalchemy.orm.raiseload) can be used to set the default style of [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) loading for a particular query, affecting all [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) -mapped attributes not otherwise specified in the [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query). This feature is available by passing the string '\*' as the argument to any of these options:

[joinedload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.joinedload" \o "sqlalchemy.orm.joinedload)，[subqueryload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.subqueryload" \o "sqlalchemy.orm.subqueryload)，[lazyload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.lazyload" \o "sqlalchemy.orm.lazyload)，[noload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.noload" \o "sqlalchemy.orm.noload)和[raiseload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.raiseload" \o "sqlalchemy.orm.raiseload)可以用于设置特定查询的[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)加载的默认样式，影响所有[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)映射属性，否则 在[Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query)中指定。 通过将字符串'\*'作为参数传递给以下任何选项，可以使用此功能：

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

Above, the lazyload('\*') option will supersede the lazy setting of all [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) constructs in use for that query, except for those which use the 'dynamic' style of loading. If some relationships specify lazy='joined' or lazy='subquery', for example, using lazyload('\*') will unilaterally cause all those relationships to use 'select' loading, e.g. emit a SELECT statement when each attribute is accessed.

以上，lazyload('\*')选项将取代该查询使用的所有[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)结构的lazy设置，但使用“dynamic”加载方式的用户除外。 如果某些关系指定lazy='joined'或lazy='subquery'，例如，使用lazyload('\*')将单方面导致所有这些关系使用“select”加载，例如。 在访问每个属性时发出SELECT语句。

The option does not supersede loader options stated in the query, such as [eagerload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.eagerload" \o "sqlalchemy.orm.eagerload), [subqueryload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.subqueryload" \o "sqlalchemy.orm.subqueryload), etc. The query below will still use joined loading for the widget relationship:

该选项不会取代查询中所述的加载程序选项，例如[eagerload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.eagerload" \o "sqlalchemy.orm.eagerload)，[subqueryload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.subqueryload" \o "sqlalchemy.orm.subqueryload)等)。下面的查询仍将使用加载加载的widget关系：

session.query(MyClass).options(

lazyload('\*'),

joinedload(MyClass.widget))

If multiple '\*' options are passed, the last one overrides those previously passed.

### Per-Entity Wildcard Loading Strategies

A variant of the wildcard loader strategy is the ability to set the strategy on a per-entity basis. For example, if querying for User and Address, we can instruct all relationships on Address only to use lazy loading by first applying the [Load](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.Load" \o "sqlalchemy.orm.Load) object, then specifying the \* as a chained option:

通配符加载器策略的一个变体是能够在每个实体的基础上设置策略。 例如，如果查询User和Address，我们可以通过首先应用[Load](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.Load" \o "sqlalchemy.orm.Load)对象，然后将\*指定为链接选项来指示Address上的所有关系，以便使用延迟加载

session.query(User, Address).options(

Load(Address).lazyload('\*'))

Above, all relationships on Address will be set to a lazy load.

4.2.7 Routing Explicit Joins/Statements into Eagerly Loaded Collections

The behavior of [joinedload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.joinedload" \o "sqlalchemy.orm.joinedload) is such that joins are created automatically, using anonymous aliases as targets, the results of which are routed into collections and scalar references on loaded objects. It is often the case that a query already includes the necessary joins which represent a particular collection or scalar reference, and the joins added by the joinedload feature are redundant - yet you'd still like the collections/references to be populated.

[joinedload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.joinedload" \o "sqlalchemy.orm.joinedload)的行为使得使用匿名别名作为目标自动创建联接，其结果被路由到加载对象的集合和标量引用。 通常情况下，查询已经包含表示特定集合或标量引用的必需连接，并且由joinload功能添加的连接是多余的 - 但您仍然希望填充集合/引用。

For this SQLAlchemy supplies the [contains\_eager()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.contains_eager" \o "sqlalchemy.orm.contains_eager) option. This option is used in the same manner as the [joinedload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.joinedload" \o "sqlalchemy.orm.joinedload) option except it is assumed that the [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) will specify the appropriate joins explicitly. Below, we specify a join between User and Address and additionally establish this as the basis for eager loading of User.addresses:

对于这个SQLAlchemy提供了[contains\_eager()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.contains_eager" \o "sqlalchemy.orm.contains_eager)选项。 此选项的使用方式与[joinedload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.joinedload" \o "sqlalchemy.orm.joinedload)选项相同，只是假定[Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query)将明确指定适当的连接。 下面我们指定一个User和Address之间的联接，并另外建立它作为渴望加载User.addresses的基础：

**class** **User**(Base):

\_\_tablename\_\_ = 'user'

id = Column(Integer, primary\_key=**True**)

addresses = relationship("Address")

**class** **Address**(Base):

\_\_tablename\_\_ = 'address'

*# ...*

q = session.query(User).join(User.addresses).\

options(contains\_eager(User.addresses))

If the "eager" portion of the statement is "aliased", the alias keyword argument to [contains\_eager()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.contains_eager" \o "sqlalchemy.orm.contains_eager) may be used to indicate it. This is sent as a reference to an [aliased()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.aliased" \o "sqlalchemy.orm.aliased) or [Alias](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Alias" \o "sqlalchemy.sql.expression.Alias) construct:

*# use an alias of the Address entity*adalias = aliased(Address)

*# construct a Query object which expects the "addresses" results*query = session.query(User).\

outerjoin(adalias, User.addresses).\

options(contains\_eager(User.addresses, alias=adalias))

*# get results normally*r = query.all()

SELECT

users.user\_id AS users\_user\_id,

users.user\_name AS users\_user\_name,

adalias.address\_id AS adalias\_address\_id,

adalias.user\_id AS adalias\_user\_id,

adalias.email\_address AS adalias\_email\_address,

(...other columns...)

FROM users

LEFT OUTER JOIN email\_addresses AS email\_addresses\_1

ON users.user\_id = email\_addresses\_1.user\_id

The path given as the argument to [contains\_eager()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.contains_eager" \o "sqlalchemy.orm.contains_eager) needs to be a full path from the starting entity. For example if we were loading Users->orders->Order->items->Item, the string version would look like:

作为contains\_eager()的参数给出的路径需要是从起始实体的完整路径。 例如，如果我们正在加载Users-> orders-> Order-> items-> Item，则字符串版本将如下所示：

query(User).options(

contains\_eager('orders').

contains\_eager('items'))

Or using the class-bound descriptor:

query(User).options(

contains\_eager(User.orders).

contains\_eager(Order.items))

### Using contains\_eager() to load a custom-filtered collection result

When we use [contains\_eager()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.contains_eager" \o "sqlalchemy.orm.contains_eager), *we* are constructing ourselves the SQL that will be used to populate collections. From this, it naturally follows that we can opt to ****modify**** what values the collection is intended to store, by writing our SQL to load a subset of elements for collections or scalar attributes.

当我们使用contains\_eager()时，我们正在构建将用于填充集合的SQL。 由此可以看出，我们可以通过编写我们的SQL来加载集合或标量属性的元素的子集，来选择修改集合想要存储的值。

As an example, we can load a User object and eagerly load only particular addresses into its .addresses collection just by filtering:

作为一个例子，我们可以加载一个User对象，并且只需要特定的地址加载到它的.addresses集合中即可。

q = session.query(User).join(User.addresses).\

filter(Address.email.like('*%e*d%')).\

options(contains\_eager(User.addresses))

The above query will load only User objects which contain at least Address object that contains the substring 'ed' in its email field; the User.addressescollection will contain ****only**** these Address entries, and *not* any other Address entries that are in fact associated with the collection.

上述查询将仅加载其电子邮件字段中至少包含“ed”子字段的Address对象的User对象; User.addressescollection将仅包含这些Address条目，而不包含任何实际上与该集合相关联的其他Address条目。

**Warning**

Keep in mind that when we load only a subset of objects into a collection, that collection no longer represents what's actually in the database. If we attempted to add entries to this collection, we might find ourselves conflicting with entries that are already in the database but not locally loaded.

请记住，当我们将一个对象的子集加载到集合中时，该集合不再代表数据库中实际的内容。 如果我们尝试向该集合添加条目，我们可能发现自己与已经在数据库中但不是本地加载的条目冲突。

In addition, the ****collection will fully reload normally**** once the object or attribute is expired. This expiration occurs whenever the [Session.commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.commit" \o "sqlalchemy.orm.session.Session.commit), [Session.rollback()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.rollback" \o "sqlalchemy.orm.session.Session.rollback)methods are used assuming default session settings, or the [Session.expire\_all()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.expire_all" \o "sqlalchemy.orm.session.Session.expire_all) or [Session.expire()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.expire" \o "sqlalchemy.orm.session.Session.expire) methods are used.

此外，一旦对象或属性过期，集合将会正常地重新加载。 只要使用Session.commit()，Session.rollback()方法，假设使用默认会话设置，或者使用Session.expire\_all()或Session.expire()方法，则此过期会发生。

For these reasons, prefer returning separate fields in a tuple rather than artificially altering a collection, when an object plus a custom set of related objects is desired:

由于这些原因，更喜欢在元组中返回单独的字段，而不是人为地更改集合，当需要一个对象加上一组自定义的相关对象时：

q = session.query(User, Address).join(User.addresses).\

filter(Address.email.like('*%e*d%'))

### Advanced Usage with Arbitrary Statements

The alias argument can be more creatively used, in that it can be made to represent any set of arbitrary names to match up into a statement. Below it is linked to a [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) which links a set of column objects to a string SQL statement:

别名alias可以更具创造性地使用，因为它可以用来表示任意一组任意名称来匹配到一个语句中。 下面链接到一个将一组列对象链接到字符串SQL语句的select()：

*# label the columns of the addresses table*

eager\_columns = select([

addresses.c.address\_id.label('a1'),

addresses.c.email\_address.label('a2'),

addresses.c.user\_id.label('a3')])

*# select from a raw SQL statement which uses those label names for the# addresses table. contains\_eager() matches them up.*

query = session.query(User).\

from\_statement("select users.\*, addresses.address\_id as a1, "

"addresses.email\_address as a2, "

"addresses.user\_id as a3 "

"from users left outer join "

"addresses on users.user\_id=addresses.user\_id").\

options(contains\_eager(User.addresses, alias=eager\_columns))

### 4.2.8 Creating Custom Load Rules

**Warning**

This is an advanced technique! Great care and testing should be applied.

The ORM has various edge cases where the value of an attribute is locally available, however the ORM itself doesn't have awareness of this. There are also cases when a user-defined system of loading attributes is desirable. To support the use case of user-defined loading systems, a key function[attributes.set\_committed\_value()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.attributes.set_committed_value" \o "sqlalchemy.orm.attributes.set_committed_value) is provided. This function is basically equivalent to Python's own setattr() function, except that when applied to a target object, SQLAlchemy's "attribute history" system which is used to determine flush-time changes is bypassed; the attribute is assigned in the same way as if the ORM loaded it that way from the database.

ORM具有各种边缘情况，其中属性的值在本地可用，但是ORM本身不具有此意识。还有一种情况是需要用户定义的加载属性系统。为了支持用户定义的加载系统的用例，提供了一个关键的functionattributes.set\_committed\_value()。此功能基本上等同于Python自己的setattr()函数，除了当应用于目标对象时，绕过用于确定刷新时间更改的SQLAlchemy的“属性历史”系统;该属性的分配方式与ORM以数据库方式加载的方式相同。

The use of [attributes.set\_committed\_value()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.attributes.set_committed_value" \o "sqlalchemy.orm.attributes.set_committed_value) can be combined with another key event known as [InstanceEvents.load()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.InstanceEvents.load" \o "sqlalchemy.orm.events.InstanceEvents.load) to produce attribute-population behaviors when an object is loaded. One such example is the bi-directional "one-to-one" case, where loading the "many-to-one" side of a one-to-one should also imply the value of the "one-to-many" side. The SQLAlchemy ORM does not consider backrefs when loading related objects, and it views a "one-to-one" as just another "one-to-many", that just happens to be one row.

使用[attributes.set\_committed\_value()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.attributes.set_committed_value" \o "sqlalchemy.orm.attributes.set_committed_value)可以与另一个称为[InstanceEvents.load()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.InstanceEvents.load" \o "sqlalchemy.orm.events.InstanceEvents.load)的键事件组合，以在加载对象时产生属性总体行为。一个这样的例子是双向“一对一”的情况，一个一对一的“多对一”一边的加载也应该意味着“一对多”一方的价值。 SQLAlchemy ORM在加载相关对象时不会考虑backref，并且将“一对一”视为仅仅是一对一的“一对多”。

Given the following mapping:

给出以下映射：

**from** **sqlalchemy** **import** Integer, ForeignKey, Column

**from** **sqlalchemy.orm** **import** relationship, backref

**from** **sqlalchemy.ext.declarative** **import** declarative\_base

Base = declarative\_base()

**class** **A**(Base):

\_\_tablename\_\_ = 'a'

id = Column(Integer, primary\_key=**True**)

b\_id = Column(ForeignKey('b.id'))

b = relationship(

"B",

backref=backref("a", uselist=**False**),

lazy='joined')

**class** **B**(Base):

\_\_tablename\_\_ = 'b'

id = Column(Integer, primary\_key=**True**)

If we query for an A row, and then ask it for a.b.a, we will get an extra SELECT:

**>>>** a1.b.a

SELECT a.id AS a\_id, a.b\_id AS a\_b\_id

FROM a

WHERE ? = a.b\_id

This SELECT is redundant because b.a is the same value as a1. We can create an on-load rule to populate this for us:

这个SELECT是多余的，因为b.a与a1的值相同。 我们可以为我们创建一个负载规则来填充这个规则：

**from** **sqlalchemy** **import** event

**from** **sqlalchemy.orm** **import** attributes

**@event**.listens\_for(A, "load")

**def** load\_b(target, context):

**if** 'b' **in** target.\_\_dict\_\_:

attributes.set\_committed\_value(target.b, 'a', target)

Now when we query for A, we will get A.b from the joined eager load, and A.b.a from our event:

现在当我们查询A时，我们将从加入的热加载中获取A.b，从我们的事件中得到A.b.a：

a1 = s.query(A).first()

SELECT

a.id AS a\_id,

a.b\_id AS a\_b\_id,

b\_1.id AS b\_1\_id

FROM a

LEFT OUTER JOIN b AS b\_1 ON b\_1.id = a.b\_id

LIMIT ? OFFSET ?

(1, 0)

assert a1.b.a is a1

### 4.2.9 Relationship Loader API

sqlalchemy.orm.**contains\_alias**(*alias*)

Return a MapperOption that will indicate to the [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) that the main table has been aliased.

返回一个MapperOption，它将向Query指示主表已经被别名。

This is a seldom-used option to suit the very rare case that [contains\_eager()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.contains_eager" \o "sqlalchemy.orm.contains_eager) is being used in conjunction with a user-defined SELECT statement that aliases the parent table. E.g.:

这是一个很少使用的选项，以满足非常罕见的情况，其中contains\_eager()与用户定义的SELECT语句一起使用，该SELECT语句使父表别名。 例如。：

*# define an aliased UNION called 'ulist'*

ulist = users.select(users.c.user\_id==7).\

union(users.select(users.c.user\_id>7)).\

alias('ulist')

*# add on an eager load of "addresses"*

statement = ulist.outerjoin(addresses).\

select().apply\_labels()

*# create query, indicating "ulist" will be an# alias for the main table, "addresses"# property should be eager loaded*query = session.query(User).options(

contains\_alias(ulist),

contains\_eager(User.addresses))

*# then get results via the statement*results = query.from\_statement(statement).all()

|  |  |
| --- | --- |
| **Parameters:** | ****alias**** – is the string name of an alias, or a [Alias](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Alias" \o "sqlalchemy.sql.expression.Alias) object representing the alias. |

sqlalchemy.orm.**contains\_eager**(*\*keys*, *\*\*kw*)

Indicate that the given attribute should be eagerly loaded from columns stated manually in the query.

指示给定的属性应该从查询中手动指定的列中加载。

This function is part of the [Load](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.Load" \o "sqlalchemy.orm.Load) interface and supports both method-chained and standalone operation.

此功能是Load接口的一部分，支持方法链接和独立操作。

The option is used in conjunction with an explicit join that loads the desired rows, i.e.:

该选项与加载所需行的显式联接结合使用，即：

sess.query(Order).\

join(Order.user).\

options(contains\_eager(Order.user))

The above query would join from the Order entity to its related User entity, and the returned Order objects would have the Order.user attribute pre-populated.

上述查询将从Order实体加入到其相关的User实体，并且返回的Order对象将预填充Order.user属性。

[contains\_eager()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.contains_eager" \o "sqlalchemy.orm.contains_eager) also accepts an alias argument, which is the string name of an alias, an [alias()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.alias" \o "sqlalchemy.sql.expression.alias) construct, or an [aliased()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.aliased" \o "sqlalchemy.orm.aliased) construct. Use this when the eagerly-loaded rows are to come from an aliased table:

[contains\_eager()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.contains_eager" \o "sqlalchemy.orm.contains_eager)也接受一个别名参数，它是一个别名，一个[alias()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.alias" \o "sqlalchemy.sql.expression.alias)结构或一个[aliased()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.aliased" \o "sqlalchemy.orm.aliased)构造的字符串名称。 当热键加载的行来自别名表时，请使用此选项：

user\_alias = aliased(User)sess.query(Order).\

join((user\_alias, Order.user)).\

options(contains\_eager(Order.user, alias=user\_alias))

**See also**

[Relationship Loading Techniques](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html)

[Routing Explicit Joins/Statements into Eagerly Loaded Collections](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "contains-eager)

sqlalchemy.orm.**defaultload**(*\*keys*)

Indicate an attribute should load using its default loader style.

使用其默认加载器样式指示属性应加载。

This method is used to link to other loader options further into a chain of attributes without altering the loader style of the links along the chain. For example, to set joined eager loading for an element of an element:

该方法用于将其他加载程序选项进一步链接到属性链中，而不会改变链路上的链接的加载程序样式。 例如，为元素的元素设置连接的渴望加载：

session.query(MyClass).options(

defaultload(MyClass.someattribute).

joinedload(MyOtherClass.someotherattribute))

[defaultload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.defaultload" \o "sqlalchemy.orm.defaultload) is also useful for setting column-level options on a related class, namely that of [defer()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "sqlalchemy.orm.defer" \o "sqlalchemy.orm.defer) and [undefer()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "sqlalchemy.orm.undefer" \o "sqlalchemy.orm.undefer):

[defaultload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.defaultload" \o "sqlalchemy.orm.defaultload)也可用于设置相关类的列级选项，即[defer()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "sqlalchemy.orm.defer" \o "sqlalchemy.orm.defer)和[undefer()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "sqlalchemy.orm.undefer" \o "sqlalchemy.orm.undefer)的列级别选项：

session.query(MyClass).options(

defaultload(MyClass.someattribute).

defer("some\_column").

undefer("some\_other\_column"))

**See also**

[Controlling Loading via Options](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "relationship-loader-options)

[Deferred Loading with Multiple Entities](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "deferred-loading-w-multiple)

sqlalchemy.orm.**eagerload**(*\*args*, *\*\*kwargs*)

A synonym for [joinedload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.joinedload" \o "sqlalchemy.orm.joinedload).

joinedload()的同义词。

sqlalchemy.orm.**eagerload\_all**(*\*args*, *\*\*kwargs*)

A synonym for [joinedload\_all()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.joinedload_all" \o "sqlalchemy.orm.joinedload_all)

[joinedload\_all()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.joinedload_all" \o "sqlalchemy.orm.joinedload_all)的同义词。

sqlalchemy.orm.**immediateload**(*\*keys*)

Indicate that the given attribute should be loaded using an immediate load with a per-attribute SELECT statement.

指示给定属性应使用每个属性的SELECT语句立即加载。

This function is part of the [Load](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.Load" \o "sqlalchemy.orm.Load) interface and supports both method-chained and standalone operation.

此功能是Load接口的一部分，支持方法链接和独立操作。

**See also**

[Relationship Loading Techniques](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html)

sqlalchemy.orm.**joinedload**(*\*keys*, *\*\*kw*)

Indicate that the given attribute should be loaded using joined eager loading.

指示给定的属性应该使用加载的加载加载。

This function is part of the [Load](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.Load" \o "sqlalchemy.orm.Load) interface and supports both method-chained and standalone operation.

examples:

此功能是Load接口的一部分，支持方法链接和独立操作。

例子：

*# joined-load the "orders" collection on "User"*query(User).options(joinedload(User.orders))

*# joined-load Order.items and then Item.keywords*query(Order).options(

joinedload(Order.items).joinedload(Item.keywords))

*# lazily load Order.items, but when Items are loaded,# joined-load the keywords collection*query(Order).options(

lazyload(Order.items).joinedload(Item.keywords))

|  |  |
| --- | --- |
| **Parameters:** | ****innerjoin**** –if True, indicates that the joined eager load should use an inner join instead of the default of left outer join:如果为True，表示连接的渴望负载应使用内部连接，而不是左侧外部连接的默认值：  query(Order).options(joinedload(Order.user, innerjoin=**True**))  In order to chain multiple eager joins together where some may be OUTER and others INNER, right-nested joins are used to link them:  为了将多个预加载连接在一起，其中一些可能是OUTER和其他INNER，右嵌套连接用于链接它们：  query(A).options(  joinedload(A.bs, innerjoin=**False**).  joinedload(B.cs, innerjoin=**True**))  The above query, linking A.bs via "outer" join and B.cs via "inner" join would render the joins as "a LEFT OUTER JOIN (b JOIN c)". When using older versions of SQLite (< 3.7.16), this form of JOIN is translated to use full subqueries as this syntax is otherwise not directly supported.  上面的查询，通过“outer”join和B.cs通过“inner”连接来连接A.bs会将连接渲染为“a LEFT OUTER JOIN(b JOIN c)”。 当使用较旧版本的SQLite(<3.7.16)时，此形式的JOIN将被转换为使用完整的子查询，因为否则不直接支持此语法。  The innerjoin flag can also be stated with the term "unnested". This indicates that an INNER JOIN should be used, *unless* the join is linked to a LEFT OUTER JOIN to the left, in which case it will render as LEFT OUTER JOIN. For example, supposing A.bs is an outerjoin:  内联标志也可以用术语“未知的”来表示。 这表示应该使用INNER JOIN，除非连接链接到左侧的LEFT OUTER JOIN，在这种情况下，它将呈现为LEFT OUTER JOIN。 例如，假设A.bs是外联：  query(A).options(  joinedload(A.bs).  joinedload(B.cs, innerjoin="unnested"))  The above join will render as "a LEFT OUTER JOIN b LEFT OUTER JOIN c", rather than as "a LEFT OUTER JOIN (b JOIN c)".  上述连接将呈现为“左侧外部连接”，而不是“左侧外部连接(b加入c)”)。  **Note**  The "unnested" flag does ****not**** affect the JOIN rendered from a many-to-many association table, e.g. a table configured as [relationship.secondary](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.secondary" \o "sqlalchemy.orm.relationship), to the target table; for correctness of results, these joins are always INNER and are therefore right-nested if linked to an OUTER join.  “未知的”标志不会影响从多对多关联表中呈现的JOIN，例如。 一个配置为relation.secondary的表到目标表; 为了结果的正确性，这些连接始终是INNER，因此如果链接到OUTER连接，则它们是正确的。  *Changed in version 1.0.0:*innerjoin=True now implies innerjoin="nested", whereas in 0.9 it implied innerjoin="unnested". In order to achieve the pre-1.0 "unnested" inner join behavior, use the value innerjoin="unnested". See [Right inner join nesting now the default for joinedload with innerjoin=True](http://docs.sqlalchemy.org/en/rel_1_1/changelog/migration_10.html" \l "migration-3008).  更改版本1.0.0：innerjoin = True现在意味着innerjoin =“嵌套”，而在0.9它暗示innerjoin =“未知的”。 为了实现1.0之前的“未知”内连接行为，请使用值innerjoin =“unnested”。 现在看到右内连接嵌套现在为innerjoin = True的joinload的默认值。 |

**Note**

The joins produced by [orm.joinedload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.joinedload" \o "sqlalchemy.orm.joinedload) are ****anonymously aliased****. The criteria by which the join proceeds cannot be modified, nor can the [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) refer to these joins in any way, including ordering. See [The Zen of Joined Eager Loading](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "zen-of-eager-loading) for further detail.

由orm.joinedload()生成的连接是匿名的别名。 连接进行的标准不能修改，也不能以任何方式查询这些连接，包括排序。 参见禅宗加入更多细节。

To produce a specific SQL JOIN which is explicitly available, use [Query.join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join). To combine explicit JOINs with eager loading of collections, use [orm.contains\_eager()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.contains_eager" \o "sqlalchemy.orm.contains_eager); see [Routing Explicit Joins/Statements into Eagerly Loaded Collections](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "contains-eager).

要生成明确可用的特定SQL JOIN，请使用Query.join()。 要结合明确的JOIN加载集合，请使用orm.contains\_eager(); 请参阅路由显式连接/语句到热切加载的集合。

**See also**

[Relationship Loading Techniques](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html)

[Joined Eager Loading](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "joined-eager-loading)

sqlalchemy.orm.**joinedload\_all**(*\*keys*, *\*\*kw*)

Produce a standalone "all" option for [orm.joinedload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.joinedload" \o "sqlalchemy.orm.joinedload).

*Deprecated since version 0.9.0:*The "\_all()" style is replaced by method chaining, e.g.:

session.query(MyClass).options(

joinedload("someattribute").joinedload("anotherattribute"))

sqlalchemy.orm.**lazyload**(*\*keys*)

Indicate that the given attribute should be loaded using "lazy" loading.

指示给定的属性应该使用“懒惰”加载加载。

This function is part of the [Load](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.Load" \o "sqlalchemy.orm.Load) interface and supports both method-chained and standalone operation.

此功能是Load接口的一部分，支持方法链接和独立操作。

**See also**

[Relationship Loading Techniques](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html)

[Lazy Loading](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "lazy-loading)

*class*sqlalchemy.orm.**Load**(*entity*)

Bases: sqlalchemy.sql.expression.Generative, sqlalchemy.orm.interfaces.MapperOption

Represents loader options which modify the state of a [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) in order to affect how various mapped attributes are loaded.

表示修改查询状态的加载程序选项，以影响各种映射属性的加载方式。

The [Load](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.Load" \o "sqlalchemy.orm.Load) object is in most cases used implicitly behind the scenes when one makes use of a query option like [joinedload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.joinedload" \o "sqlalchemy.orm.joinedload), [defer()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "sqlalchemy.orm.defer" \o "sqlalchemy.orm.defer), or similar. However, the [Load](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.Load" \o "sqlalchemy.orm.Load) object can also be used directly, and in some cases can be useful.

大多数情况下，Load对象在使用诸如joinload()，defer()或类似的查询选项时，会隐藏在幕后。 但是，Load对象也可以直接使用，在某些情况下也可以使用。

To use [Load](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.Load" \o "sqlalchemy.orm.Load) directly, instantiate it with the target mapped class as the argument. This style of usage is useful when dealing with a [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) that has multiple entities:

要直接使用Load，请将其与目标映射类作为参数进行实例化。 处理具有多个实体的查询时，这种使用风格很有用：

myopt = Load(MyClass).joinedload("widgets")

The above myopt can now be used with [Query.options()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.options" \o "sqlalchemy.orm.query.Query.options), where it will only take effect for the MyClass entity:

session.query(MyClass, MyOtherClass).options(myopt)

One case where [Load](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.Load" \o "sqlalchemy.orm.Load) is useful as public API is when specifying "wildcard" options that only take effect for a certain class:

session.query(Order).options(Load(Order).lazyload('\*'))

Above, all relationships on Order will be lazy-loaded, but other attributes on those descendant objects will load using their normal loader strategy.

**See also**

[Relationship Loading Techniques](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html)

sqlalchemy.orm.**noload**(*\*keys*)

Indicate that the given relationship attribute should remain unloaded.

This function is part of the [Load](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.Load" \o "sqlalchemy.orm.Load) interface and supports both method-chained and standalone operation.

[orm.noload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.noload" \o "sqlalchemy.orm.noload) applies to [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) attributes; for column-based attributes, see [orm.defer()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "sqlalchemy.orm.defer" \o "sqlalchemy.orm.defer).

**See also**

[Relationship Loading Techniques](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html)

sqlalchemy.orm.**raiseload**(*\*keys*, *\*\*kw*)

Indicate that the given relationship attribute should disallow lazy loads.

表明给定的关系属性应该不允许延迟加载。

A relationship attribute configured with [orm.raiseload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.raiseload" \o "sqlalchemy.orm.raiseload) will raise an [InvalidRequestError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.InvalidRequestError" \o "sqlalchemy.exc.InvalidRequestError) upon access. The typical way this is useful is when an application is attempting to ensure that all relationship attributes that are accessed in a particular context would have been already loaded via eager loading. Instead of having to read through SQL logs to ensure lazy loads aren't occurring, this strategy will cause them to raise immediately.

使用orm.raiseload()配置的关系属性将在访问时引发InvalidRequestError。 典型的方法是当应用程序尝试确保在特定上下文中访问的所有关系属性已经通过热心加载加载时。 而不必阅读SQL日志，以确保不会发生懒惰负载，这种策略将导致它们立即提升。

|  |  |
| --- | --- |
| **Parameters:** | ****sql\_only**** – if True, raise only if the lazy load would emit SQL, but not if it is only checking the identity map, or determining that the related value should just be None due to missing keys. When False, the strategy will raise for all varieties of lazyload.  如果为True，则只有在惰性负载将发出SQL的情况下才会启动，但如果仅检查身份映射，或者由于缺少键确定相关值应为“None”。 当False时，该策略将针对所有的lazyload种类提升。 |

This function is part of the [Load](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.Load" \o "sqlalchemy.orm.Load) interface and supports both method-chained and standalone operation.

[orm.raiseload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.raiseload" \o "sqlalchemy.orm.raiseload) applies to [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) attributes only.

*New in version 1.1.*

**See also**

[Relationship Loading Techniques](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html)

[Preventing unwanted lazy loads using raiseload](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "prevent-lazy-with-raiseload)

sqlalchemy.orm.**subqueryload**(*\*keys*)

Indicate that the given attribute should be loaded using subquery eager loading.

表明给定的属性应该使用子查询加载加载。

This function is part of the [Load](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.Load" \o "sqlalchemy.orm.Load) interface and supports both method-chained and standalone operation.

此功能是Load接口的一部分，支持方法链接和独立操作。

examples:

例子：

*# subquery-load the "orders" collection on "User"*

query(User).options(subqueryload(User.orders))

*# subquery-load Order.items and then Item.keywords*

query(Order).options(subqueryload(Order.items).subqueryload(Item.keywords))

*# lazily load Order.items, but when Items are loaded,# subquery-load the keywords collection*

query(Order).options(lazyload(Order.items).subqueryload(Item.keywords))

**See also**

[Relationship Loading Techniques](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html)

[Subquery Eager Loading](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "subquery-eager-loading)

sqlalchemy.orm.**subqueryload\_all**(*\*keys*)

Produce a standalone "all" option for [orm.subqueryload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.subqueryload" \o "sqlalchemy.orm.subqueryload).

*Deprecated since version 0.9.0:*The "\_all()" style is replaced by method chaining, e.g.:

session.query(MyClass).options(

subqueryload("someattribute").subqueryload("anotherattribute"))

# **4.3 Loading Inheritance Hierarchies**

When classes are mapped in inheritance hierarchies using the "joined", "single", or "concrete" table inheritance styles as described at [Mapping Class Inheritance Hierarchies](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance.html), the usual behavior is that a query for a particular base class will also yield objects corresponding to subclasses as well. When a single query is capable of returning a result with a different class or subclasses per result row, we use the term "polymorphic loading".

当使用映射类继承层次结构中描述的"joined"，"single"或"concrete"表继承样式将类映射到继承层次结构中时，通常的行为是对特定基类的查询也将产生对应于子类也。当单个查询能够使用不同类或每个结果行的子类返回结果时，我们使用术语"多态加载"。

Within the realm of polymorphic loading, specifically with joined and single table inheritance, there is an additional problem of which subclass attributes are to be queried up front, and which are to be loaded later. When an attribute of a particular subclass is queried up front, we can use it in our query as something to filter on, and it also will be loaded when we get our objects back. If it's not queried up front, it gets loaded later when we first need to access it. Basic control of this behavior is provided using the [orm.with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic" \o "sqlalchemy.orm.with_polymorphic) function, as well as two variants, the mapper configuration [mapper.with\_polymorphic](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.params.with_polymorphic" \o "sqlalchemy.orm.mapper) and the [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) -level [Query.with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.with_polymorphic" \o "sqlalchemy.orm.query.Query.with_polymorphic) method. The "with\_polymorphic" family each provide a means of specifying which specific subclasses of a particular base class should be included within a query, which implies what columns and tables will be available in the SELECT.

在多态加载领域，特别是连接和单表继承，还有一个额外的问题，即哪些子类属性要被查询，哪些子类的属性将被稍后加载。当查询特定子类的属性时，我们可以在我们的查询中使用它作为过滤的东西，并且当我们收回我们的对象时也会被加载。如果没有向前查询，当我们第一次需要访问它时，它会被加载。使用[orm.with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic" \o "sqlalchemy.orm.with_polymorphic)函数以及映射器配置[mapper.with\_polymorphic](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.params.with_polymorphic" \o "sqlalchemy.orm.mapper)和[Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) -level [Query.with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.with_polymorphic" \o "sqlalchemy.orm.query.Query.with_polymorphic)方法提供了这种行为的基本控制。 "with\_polymorphic"系列每个都提供一种方法来指定特定基类的哪些特定子类应该包含在查询中，这意味着SELECT中可以使用哪些列和表。

Using with\_polymorphic

For the following sections, assume the Employee / Engineer / Manager examples introduced in [Mapping Class Inheritance Hierarchies](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance.html).

对于以下部分，假设在“映射类继承层次结构”中引入的Employee / Engineer / Manager示例。

Normally, when a [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) specifies the base class of an inheritance hierarchy, only the columns that are local to that base class are queried:

通常，当[Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query)指定继承层次结构的基类时，仅查询该基类本地的列：

session.query(Employee).all()

Above, for both single and joined table inheritance, only the columns local to Employee will be present in the SELECT. We may get back instances of Engineer or Manager, however they will not have the additional attributes loaded until we first access them, at which point a lazy load is emitted.

以上，对于单个和连接表继承，只有Employee本地的列将存在于SELECT中。我们可能会收回Engineer或Manager的实例，但是在我们首次访问它们之前，它们将不会加载额外的属性，此时会发出懒惰的负载。

Similarly, if we wanted to refer to columns mapped to Engineer or Manager in our query that's against Employee, these columns aren't available directly in either the single or joined table inheritance case, since the Employee entity does not refer to these columns (note that for single-table inheritance, this is common if Declarative is used, but not for a classical mapping).

同样，如果我们想在我们的查询中引用与Employee对应映射到Engineer 或Manager的列，那么这些列不能直接在单表继承或连接的表继承的情况下使用，因为Employee实体不引用这些列(注意对于单表继承，如果使用Declarative，而不是经典映射，这是常见的)。

To solve both of these issues, the [orm.with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic" \o "sqlalchemy.orm.with_polymorphic) function provides a special [AliasedClass](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.util.AliasedClass" \o "sqlalchemy.orm.util.AliasedClass) that represents a range of columns across subclasses. This object can be used in a [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) like any other alias. When queried, it represents all the columns present in the classes given:

为了解决这两个问题，[orm.with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic" \o "sqlalchemy.orm.with_polymorphic)函数提供了一个特殊的[AliasedClass](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.util.AliasedClass" \o "sqlalchemy.orm.util.AliasedClass)，它表示跨子类的一系列列。该对象可以像任何其他别名一样在查询中使用。当查询时，它代表给出的类中存在的所有列：

**from** **sqlalchemy.orm** **import** with\_polymorphic

eng\_plus\_manager = with\_polymorphic(Employee, [Engineer, Manager])

query = session.query(eng\_plus\_manager)

If the above mapping were using joined table inheritance, the SELECT statement for the above would be:

如果上述映射使用连接表继承，则上述的SELECT语句将是：

query.all()

SELECT employee.id AS employee\_id,

engineer.id AS engineer\_id,

manager.id AS manager\_id,

employee.name AS employee\_name,

employee.type AS employee\_type,

engineer.engineer\_info AS engineer\_engineer\_info,

manager.manager\_data AS manager\_manager\_data

FROM employee

LEFT OUTER JOIN engineer

ON employee.id = engineer.id

LEFT OUTER JOIN manager

ON employee.id = manager.id

[]

Where above, the additional tables / columns for "engineer" and "manager" are included. Similar behavior occurs in the case of single table inheritance.

在上面，“engineer”和“manager”的附加表/列包括在内。 在单表继承的情况下也会发生类似的行为。

[orm.with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic" \o "sqlalchemy.orm.with_polymorphic) accepts a single class or mapper, a list of classes/mappers, or the string '\*' to indicate all subclasses:

orm.with\_polymorphic()接受单个类或映射器，类/映射器列表或字符串'\*'以指示所有子类：

*# include columns for Engineer*

entity = with\_polymorphic(Employee, Engineer)

*# include columns for Engineer, Manager*

entity = with\_polymorphic(Employee, [Engineer, Manager])

*# include columns for all mapped subclasses*

entity = with\_polymorphic(Employee, '\*')

### Using aliasing with with\_polymorphic

The [orm.with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic" \o "sqlalchemy.orm.with_polymorphic) function also provides "aliasing" of the polymorphic selectable itself, meaning, two different [orm.with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic" \o "sqlalchemy.orm.with_polymorphic)entities, referring to the same class hierarchy, can be used together. This is available using the [orm.with\_polymorphic.aliased](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic.params.aliased" \o "sqlalchemy.orm.with_polymorphic) flag. For a polymorphic selectable that is across multiple tables, the default behavior is to wrap the selectable into a subquery. Below we emit a query that will select for "employee or manager" paired with "employee or engineer" on employees with the same name:

[orm.with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic" \o "sqlalchemy.orm.with_polymorphic)函数还提供了多态可选择本身的“混叠”，意思是引用同一类层次结构的两个不同的[orm.with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic" \o "sqlalchemy.orm.with_polymorphic)实体可以一起使用。 这可以使用[orm.with\_polymorphic.aliased](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic.params.aliased" \o "sqlalchemy.orm.with_polymorphic)标志。 对于跨多个表的多态可选择，默认行为是将可选择包装到子查询中。 下面我们发出一个查询，将选择与雇员或工程师在同一名称的员工配对的“员工或经理”：

engineer\_employee = with\_polymorphic(

Employee, [Engineer], aliased=True)manager\_employee = with\_polymorphic(

Employee, [Manager], aliased=True)

q = s.query(engineer\_employee, manager\_employee).\

join(

manager\_employee,

and\_(

engineer\_employee.id > manager\_employee.id,

engineer\_employee.name == manager\_employee.name

))q.all()

SELECT

anon\_1.employee\_id AS anon\_1\_employee\_id,

anon\_1.employee\_name AS anon\_1\_employee\_name,

anon\_1.employee\_type AS anon\_1\_employee\_type,

anon\_1.engineer\_id AS anon\_1\_engineer\_id,

anon\_1.engineer\_engineer\_name AS anon\_1\_engineer\_engineer\_name,

anon\_2.employee\_id AS anon\_2\_employee\_id,

anon\_2.employee\_name AS anon\_2\_employee\_name,

anon\_2.employee\_type AS anon\_2\_employee\_type,

anon\_2.manager\_id AS anon\_2\_manager\_id,

anon\_2.manager\_manager\_name AS anon\_2\_manager\_manager\_name

FROM (

SELECT

employee.id AS employee\_id,

employee.name AS employee\_name,

employee.type AS employee\_type,

engineer.id AS engineer\_id,

engineer.engineer\_name AS engineer\_engineer\_name

FROM employee

LEFT OUTER JOIN engineer ON employee.id = engineer.id

) AS anon\_1

JOIN (

SELECT

employee.id AS employee\_id,

employee.name AS employee\_name,

employee.type AS employee\_type,

manager.id AS manager\_id,

manager.manager\_name AS manager\_manager\_name

FROM employee

LEFT OUTER JOIN manager ON employee.id = manager.id

) AS anon\_2

ON anon\_1.employee\_id > anon\_2.employee\_id

AND anon\_1.employee\_name = anon\_2.employee\_name

The creation of subqueries above is very verbose. While it creates the best encapsulation of the two distinct queries, it may be inefficient.[orm.with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic" \o "sqlalchemy.orm.with_polymorphic) includes an additional flag to help with this situation, [orm.with\_polymorphic.flat](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic.params.flat" \o "sqlalchemy.orm.with_polymorphic), which will "flatten" the subquery / join combination into straight joins, applying aliasing to the individual tables instead. Setting [orm.with\_polymorphic.flat](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic.params.flat" \o "sqlalchemy.orm.with_polymorphic) implies [orm.with\_polymorphic.aliased](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic.params.aliased" \o "sqlalchemy.orm.with_polymorphic), so only one flag is necessary:

上面的子查询的创建是非常冗长的。 虽然它创建了两个不同查询的最佳封装，但它可能是无效的.[orm.with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic" \o "sqlalchemy.orm.with_polymorphic)包括一个额外的标志来帮助这种情况，[orm.with\_polymorphic.flat](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic.params.flat" \o "sqlalchemy.orm.with_polymorphic)，它会将子查询/连接组合“展平”为直 连接，将别名应用于各个表。 设置[orm.with\_polymorphic.flat](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic.params.flat" \o "sqlalchemy.orm.with_polymorphic)意味着[orm.with\_polymorphic.aliased](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic.params.aliased" \o "sqlalchemy.orm.with_polymorphic)，因此只需要一个标志：

engineer\_employee = with\_polymorphic(

Employee, [Engineer], flat=True)

manager\_employee = with\_polymorphic(

Employee, [Manager], flat=True)

q = s.query(engineer\_employee, manager\_employee).\

join(

manager\_employee,

and\_(

engineer\_employee.id > manager\_employee.id,

engineer\_employee.name == manager\_employee.name

))q.all()

SELECT

employee\_1.id AS employee\_1\_id,

employee\_1.name AS employee\_1\_name,

employee\_1.type AS employee\_1\_type,

engineer\_1.id AS engineer\_1\_id,

engineer\_1.engineer\_name AS engineer\_1\_engineer\_name,

employee\_2.id AS employee\_2\_id,

employee\_2.name AS employee\_2\_name,

employee\_2.type AS employee\_2\_type,

manager\_1.id AS manager\_1\_id,

manager\_1.manager\_name AS manager\_1\_manager\_name

FROM employee AS employee\_1

LEFT OUTER JOIN engineer AS engineer\_1

ON employee\_1.id = engineer\_1.id

JOIN (

employee AS employee\_2

LEFT OUTER JOIN manager AS manager\_1

ON employee\_2.id = manager\_1.id

)

ON employee\_1.id > employee\_2.id

AND employee\_1.name = employee\_2.name

Note above, when using [orm.with\_polymorphic.flat](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic.params.flat" \o "sqlalchemy.orm.with_polymorphic), it is often the case when used in conjunction with joined table inheritance that we get a right-nested JOIN in our statement. Some older databases, in particular older versions of SQLite, may have a problem with this syntax, although virtually all modern database versions now support this syntax.

注意上面，当使用[orm.with\_polymorphic.flat](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic.params.flat" \o "sqlalchemy.orm.with_polymorphic)时，通常在与连接表继承一起使用的情况下，我们在我们的语句中获得了一个右嵌套的JOIN。 一些较旧的数据库，特别是旧版本的SQLite，可能会对此语法有问题，尽管现在几乎所有的现代数据库版本都支持这种语法。

### Referring to Specific Subclass Attributes

The entity returned by [orm.with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic" \o "sqlalchemy.orm.with_polymorphic) is an [AliasedClass](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.util.AliasedClass" \o "sqlalchemy.orm.util.AliasedClass) object, which can be used in a [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) like any other alias, including named attributes for those attributes on the Employee class. In our previous example, eng\_plus\_manager becomes the entity that we use to refer to the three-way outer join above. It also includes namespaces for each class named in the list of classes, so that attributes specific to those subclasses can be called upon as well. The following example illustrates calling upon attributes specific to Engineer as well as Manager in terms of eng\_plus\_manager:

[orm.with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic" \o "sqlalchemy.orm.with_polymorphic)返回的实体是一个[AliasedClass](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.util.AliasedClass" \o "sqlalchemy.orm.util.AliasedClass)对象，可以像[Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query)类似任何其他别名一样使用，包括Employee类上这些属性的命名属性。 在我们前面的例子中，eng\_plus\_manager成为我们用来引用上面三向外连接的实体。 它还包括在类列表中命名的每个类的命名空间，因此也可以调用特定于这些子类的属性。 以下示例说明了在eng\_plus\_manager方面调用特定于Engineer以及Manager的属性：

eng\_plus\_manager = with\_polymorphic(Employee, [Engineer, Manager])query = session.query(eng\_plus\_manager).filter(

or\_(

eng\_plus\_manager.Engineer.engineer\_info=='x',

eng\_plus\_manager.Manager.manager\_data=='y'

)

)

### Setting with\_polymorphic at mapper configuration time

The [orm.with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic" \o "sqlalchemy.orm.with_polymorphic) function serves the purpose of allowing "eager" loading of attributes from subclass tables, as well as the ability to refer to the attributes from subclass tables at query time. Historically, the "eager loading" of columns has been the more important part of the equation. So just as eager loading for relationships can be specified as a configurational option, the [mapper.with\_polymorphic](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.params.with_polymorphic" \o "sqlalchemy.orm.mapper) configuration parameter allows an entity to use a polymorphic load by default. We can add the parameter to our Employee mapping first introduced at [Joined Table Inheritance](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance.html" \l "joined-inheritance):

[orm.with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic" \o "sqlalchemy.orm.with_polymorphic)函数用于允许从子类表“预”加载属性，以及在查询时从子类表引用属性的功能。 历史上，列的“热切加载”一直是方程中更重要的部分。 所以就像为关系加载一样可以被指定为一个配置选项，mapper.with\_polymorphic配置参数允许一个实体默认使用多态加载。 我们可以将参数添加到我们的Employee映射中，首先在“加入表继承”中引入：

**class** **Employee**(Base):

\_\_tablename\_\_ = 'employee'

id = Column(Integer, primary\_key=**True**)

name = Column(String(50))

type = Column(String(50))

\_\_mapper\_args\_\_ = {

'polymorphic\_identity':'employee',

'polymorphic\_on':type,

'with\_polymorphic': '\*'

}

Above is the most common setting for [mapper.with\_polymorphic](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.params.with_polymorphic" \o "sqlalchemy.orm.mapper), which is to indicate an asterisk to load all subclass columns. In the case of joined table inheritance, this option should be used sparingly, as it implies that the mapping will always emit a (often large) series of LEFT OUTER JOIN to many tables, which is not efficient from a SQL perspective. For single table inheritance, specifying the asterisk is often a good idea as the load is still against a single table only, but an additional lazy load of subclass-mapped columns will be prevented.

以上是[mapper.with\_polymorphic](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.params.with_polymorphic" \o "sqlalchemy.orm.mapper)最常见的设置，它指示加载所有子类列的星号。 在连接表继承的情况下，应该谨慎使用此选项，因为它意味着映射将始终向许多表发出一个(通常很大的)LEFT OUTER JOIN系列，这从SQL透视图效率不高。 对于单表继承，指定星号通常是一个好主意，因为加载仍然仅针对单个表，但会阻止子类映射列的附加惰性负载。

Using [orm.with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic" \o "sqlalchemy.orm.with_polymorphic) or [Query.with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.with_polymorphic" \o "sqlalchemy.orm.query.Query.with_polymorphic) will override the mapper-level [mapper.with\_polymorphic](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.params.with_polymorphic" \o "sqlalchemy.orm.mapper) setting.

使用orm.with\_polymorphic()或Query.with\_polymorphic()将覆盖mapper-level mapper.with\_polymorphic设置。

The [mapper.with\_polymorphic](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.params.with_polymorphic" \o "sqlalchemy.orm.mapper) option also accepts a list of classes just like [orm.with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic" \o "sqlalchemy.orm.with_polymorphic) to polymorphically load among a subset of classes, however this API was first designed with classical mapping in mind; when using Declarative, the subclasses aren't available yet. The current workaround is to set the[mapper.with\_polymorphic](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.params.with_polymorphic" \o "sqlalchemy.orm.mapper) setting after all classes have been declared, using the semi-private method mapper.\_set\_with\_polymorphic(). A future release of SQLAlchemy will allow finer control over mapper-level polymorphic loading with declarative, using new options specified on individual subclasses. When using concrete inheritance, special helpers are provided to help with these patterns which are described at [Concrete Polymorphic Loading Configuration](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance.html" \l "concrete-polymorphic).

[mapper.with\_polymorphic](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.params.with_polymorphic" \o "sqlalchemy.orm.mapper)选项还接受类似orm.with\_polymorphic()的类列表，以便在类的一个子集中进行多态加载，但是这个API首先是设计了经典映射; 当使用Declarative时，子类还不可用。 当前的解决方法是在声明所有类之后使用半私有方法mapper.\_set\_with\_polymorphic()来设置themapper.with\_polymorphic设置。 SQLAlchemy的未来版本将允许使用声明式更好地控制映射器级多态加载，并使用在各个子类上指定的新选项。 当使用具体遗传时，提供特殊的帮助者来帮助这些模式，这些模式在Concrete Polymorphic Loading Configuration中有所描述。

### Setting with\_polymorphic against a query

The [orm.with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic" \o "sqlalchemy.orm.with_polymorphic) function evolved from a query-level method [Query.with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.with_polymorphic" \o "sqlalchemy.orm.query.Query.with_polymorphic). This method has the same purpose as [orm.with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic" \o "sqlalchemy.orm.with_polymorphic), except is not as flexible in its usage patterns in that it only applies to the first entity of the [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query). It then takes effect for all occurences of that entity, so that the entity (and its subclasses) can be referred to directly, rather than using an alias object. For simple cases it might be considered to be more succinct:

[orm.with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic" \o "sqlalchemy.orm.with_polymorphic)函数从查询级方法[Query.with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.with_polymorphic" \o "sqlalchemy.orm.query.Query.with_polymorphic)演变而来。 该方法与[orm.with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic" \o "sqlalchemy.orm.with_polymorphic)具有相同的目的，除了它的使用模式不是那么灵活，因为它只适用于Query的第一个实体。 然后它将对该实体的所有出现生效，因此可以直接引用实体(及其子类)，而不是使用别名对象。 对于简单的案例，可能被认为更简洁：

session.query(Employee).\

with\_polymorphic([Engineer, Manager]).\

filter(

or\_(

Engineer.engineer\_info=='w',

Manager.manager\_data=='q'

)

)

The [Query.with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.with_polymorphic" \o "sqlalchemy.orm.query.Query.with_polymorphic) method has a more complicated job than the [orm.with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic" \o "sqlalchemy.orm.with_polymorphic) function, as it needs to correctly transform entities like Engineer and Manager appropriately, but not interfere with other entities. If its flexibility is lacking, switch to using [orm.with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic" \o "sqlalchemy.orm.with_polymorphic).

[Query.with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.with_polymorphic" \o "sqlalchemy.orm.query.Query.with_polymorphic)方法的工作比[orm.with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic" \o "sqlalchemy.orm.with_polymorphic)函数更复杂，因为它需要正确地转换像Engineer和Manager这样的实体，而不会干扰其他实体。 如果缺乏灵活性，切换到使用[orm.with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic" \o "sqlalchemy.orm.with_polymorphic)。

## Referring to specific subtypes on relationships

Mapped attributes which correspond to a [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) are used in querying in order to refer to the linkage between two mappings. Common uses for this are to refer to a [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) in [Query.join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join) as well as in loader options like [joinedload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.joinedload" \o "sqlalchemy.orm.joinedload). When using [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) where the target class is an inheritance hierarchy, the API allows that the join, eager load, or other linkage should target a specific subclass, alias, or [orm.with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic" \o "sqlalchemy.orm.with_polymorphic) alias, of that class hierarchy, rather than the class directly targeted by the [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship).

对应于关系()的映射属性用于查询，以引用两个映射之间的链接。 常用的方法是引用Query.join()中的relationship()以及joinload()这样的加载器选项。 当使用目标类是继承层次结构的relationship()时，API允许连接，热切加载或其他链接应该定位到该类层次结构的特定子类别，别名或orm.with\_polymorphic()别名，而不是 该类直接针对关系()。

The [of\_type()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator.of_type" \o "sqlalchemy.orm.interfaces.PropComparator.of_type) method allows the construction of joins along [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) paths while narrowing the criterion to specific derived aliases or subclasses. Suppose the employees table represents a collection of employees which are associated with a Company object. We'll add a company\_id column to theemployees table and a new table companies:

the\_type()方法允许沿着()路径构建连接，同时将标准缩小到特定的派生别名或子类。 假设employees表表示与公司对象相关联的员工集合。 我们将添加一个company\_id列到employeesees表和一个新的表公司：

**class** **Company**(Base):

\_\_tablename\_\_ = 'company'

id = Column(Integer, primary\_key=True)

name = Column(String(50))

employees = relationship("Employee",

backref='company')

**class** **Employee**(Base):

\_\_tablename\_\_ = 'employee'

id = Column(Integer, primary\_key=True)

type = Column(String(20))

company\_id = Column(Integer, ForeignKey('company.id'))

\_\_mapper\_args\_\_ = {

'polymorphic\_on':type,

'polymorphic\_identity':'employee',

}

**class** **Engineer**(Employee):

\_\_tablename\_\_ = 'engineer'

id = Column(Integer, ForeignKey('employee.id'), primary\_key=True)

engineer\_info = Column(String(50))

\_\_mapper\_args\_\_ = {'polymorphic\_identity':'engineer'}

**class** **Manager**(Employee):

\_\_tablename\_\_ = 'manager'

id = Column(Integer, ForeignKey('employee.id'), primary\_key=True)

manager\_data = Column(String(50))

\_\_mapper\_args\_\_ = {'polymorphic\_identity':'manager'}

When querying from Company onto the Employee relationship, the [Query.join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join) method as well as operators like [PropComparator.any()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator.any" \o "sqlalchemy.orm.interfaces.PropComparator.any) and [PropComparator.has()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator.has" \o "sqlalchemy.orm.interfaces.PropComparator.has) will create a join from company to employee, without including engineer or manager in the mix. If we wish to have criterion which is specifically against the Engineer class, we can tell those methods to join or subquery against the set of columns representing the subclass using the [of\_type()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator.of_type" \o "sqlalchemy.orm.interfaces.PropComparator.of_type)operator:

当从Company查询到Employee关系时，Query.join()方法以及[PropComparator.any()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator.any" \o "sqlalchemy.orm.interfaces.PropComparator.any)和PropComparator.has()之类的运算符将创建一个从公司到员工的连接，而不需要在组合中包括工程师或经理。 如果我们希望具有专门针对Engineer类的标准，那么我们可以使用of\_type()运算符来告诉那些使用代表子类的列加入或子查询的方法：

session.query(Company).\

join(Company.employees.of\_type(Engineer)).\

filter(Engineer.engineer\_info=='someinfo')

Similarly, to join from Company to the polymorphic entity that includes both Engineer and Manager columns:

同样，从Company加入包含Engineer和Manager列的多态实体：

manager\_and\_engineer = with\_polymorphic(

Employee, [Manager, Engineer])

session.query(Company).\

join(Company.employees.of\_type(manager\_and\_engineer)).\

filter(

or\_(

manager\_and\_engineer.Engineer.engineer\_info == 'someinfo',

manager\_and\_engineer.Manager.manager\_data == 'somedata'

)

)

The [PropComparator.any()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator.any" \o "sqlalchemy.orm.interfaces.PropComparator.any) and [PropComparator.has()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator.has" \o "sqlalchemy.orm.interfaces.PropComparator.has) operators also can be used with [of\_type()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator.of_type" \o "sqlalchemy.orm.interfaces.PropComparator.of_type), such as when the embedded criterion is in terms of a subclass:

[PropComparator.any()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator.any" \o "sqlalchemy.orm.interfaces.PropComparator.any)和[PropComparator.has()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator.has" \o "sqlalchemy.orm.interfaces.PropComparator.has)运算符也可以与[of\_type()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator.of_type" \o "sqlalchemy.orm.interfaces.PropComparator.of_type)一起使用，例如当嵌入的标准是一个子类时：

session.query(Company).\

filter(

Company.employees.of\_type(Engineer).

any(Engineer.engineer\_info=='someinfo')

).all()

### Eager Loading of Specific or Polymorphic Subtypes

The [joinedload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.joinedload" \o "sqlalchemy.orm.joinedload), [subqueryload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.subqueryload" \o "sqlalchemy.orm.subqueryload), [contains\_eager()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.contains_eager" \o "sqlalchemy.orm.contains_eager) and other eagerloader options support paths which make use of of\_type(). Below, we load Company rows while eagerly loading related Engineer objects, querying the employee and engineer tables simultaneously:

[joinedload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.joinedload" \o "sqlalchemy.orm.joinedload), [subqueryload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.subqueryload" \o "sqlalchemy.orm.subqueryload), [contains\_eager()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.contains_eager" \o "sqlalchemy.orm.contains_eager)和其他eagerloader选项支持使用of\_type()的路径。 下面，我们加载Company行，同时加载相关的Engineer对象，同时查询employee和engineer表：

session.query(Company).\

options(

subqueryload(Company.employees.of\_type(Engineer)).

subqueryload(Engineer.machines)

)

)

As is the case with [Query.join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join), [of\_type()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator.of_type" \o "sqlalchemy.orm.interfaces.PropComparator.of_type) can be used to combine eager loading and [orm.with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic" \o "sqlalchemy.orm.with_polymorphic), so that all sub-attributes of all referenced subtypes can be loaded:

与[Query.join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join)的情况一样，可以使用[of\_type()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator.of_type" \o "sqlalchemy.orm.interfaces.PropComparator.of_type)来组合加载加载和[orm.with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic" \o "sqlalchemy.orm.with_polymorphic)，以便可以加载所有引用子类型的所有子属性：

manager\_and\_engineer = with\_polymorphic(

Employee, [Manager, Engineer],

flat=**True**)

session.query(Company).\

options(

joinedload(

Company.employees.of\_type(manager\_and\_engineer)

)

)

When using [with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic" \o "sqlalchemy.orm.with_polymorphic) in conjunction with [joinedload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.joinedload" \o "sqlalchemy.orm.joinedload), the [with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic" \o "sqlalchemy.orm.with_polymorphic) object must include the aliased=True or flat=Trueflag, so that the polymorphic selectable is aliased (an informative error message is raised otherwise). "flat" is an alternate form of aliasing that produces fewer subqueries.

当使用with\_polymorphic()与joinedload()一起使用时，with\_polymorphic()对象必须包含aliased = True或flat = Trueflag，因此可选择的多态可以被别名(否则会引发一个有用的错误消息)。 “平面”是产生较少子查询的别名替代形式。

Once [of\_type()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator.of_type" \o "sqlalchemy.orm.interfaces.PropComparator.of_type) is the target of the eager load, that's the entity we would use for subsequent chaining, not the original class or derived class. If we wanted to further eager load a collection on the eager-loaded Engineer class, we access this class from the namespace of the [orm.with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic" \o "sqlalchemy.orm.with_polymorphic) object:

一旦of\_type()是渴望加载的目标，那么我们将用于后续链接的实体，而不是原始类或派生类。 如果我们想要在加载工程师类的进一步加载一个集合，我们从orm.with\_polymorphic()对象的命名空间访问这个类：

session.query(Company).\

options(

joinedload(Company.employees.of\_type(manager\_and\_engineer)).\

subqueryload(manager\_and\_engineer.Engineer.computers)

)

)

## Loading objects with joined table inheritance

When using joined table inheritance, if we query for a specific subclass that represents a JOIN of two tables such as our Engineer example from the inheritance section, the SQL emitted is a join:

当使用连接表继承时，如果我们查询代表两个表的JOIN的特定子类，例如继承部分中的我们的Engineer示例，则发出的SQL是一个连接：

session.query(Engineer).all()

The above query will emit SQL like:

SELECT employee.id AS employee\_id,

employee.name AS employee\_name, employee.type AS employee\_type,

engineer.name AS engineer\_name

FROM employee JOIN engineer

ON employee.id = engineer.id

We will then get a collection of Engineer objects back, which will contain all columns from employee and engineer loaded.

However, when emitting a [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) against a base class, the behavior is to load only from the base table:

然后，我们将收集一系列工程师对象，其中包含加载员工和工程师的所有列。

然而，当对基类发出Query时，行为是仅从基表加载：

session.query(Employee).all()

Above, the default behavior would be to SELECT only from the employee table and not from any "sub" tables (engineer and manager, in our previous examples):

以上，默认行为是仅从employee表中选择SELECT，而不是从任何“子”表(我们以前的示例中的engineer和manager，可看i'7)：

SELECT employee.id AS employee\_id,

employee.name AS employee\_name, employee.type AS employee\_type

FROM employee

[]

After a collection of Employee objects has been returned from the query, and as attributes are requested from those Employee objects which are represented in either the engineer or manager child tables, a second load is issued for the columns in that related row, if the data was not already loaded. So above, after accessing the objects you'd see further SQL issued along the lines of:

从查询返回一个Employee对象的集合后，并且当从engineer或manager子表中表示的那些Employee对象请求属性时，将为相关行中的列发出第二个加载，如果数据 尚未加载 因此，在访问对象之后，您将看到以下行：

SELECT manager.id AS manager\_id,

manager.manager\_data AS manager\_manager\_data

FROM manager

WHERE ? = manager.id

[5]

SELECT engineer.id AS engineer\_id,

engineer.engineer\_info AS engineer\_engineer\_info

FROM engineer

WHERE ? = engineer.id

[2]

The [orm.with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic" \o "sqlalchemy.orm.with_polymorphic) function and related configuration options allow us to instead emit a JOIN up front which will conditionally load against employee, engineer, or manager, very much like joined eager loading works for relationships, removing the necessity for a second per-entity load:

orm.with\_polymorphic()函数和相关配置选项允许我们发出一个JOIN前端，这将有条件地加载到员工，工程师或经理，非常喜欢加入的渴望加载工作的关系，消除了第二个 实体负载：

**from** **sqlalchemy.orm** **import** with\_polymorphic

eng\_plus\_manager = with\_polymorphic(Employee, [Engineer, Manager])

query = session.query(eng\_plus\_manager)

The above produces a query which joins the employee table to both the engineer and manager tables like the following:

以上产生一个查询，将雇员表连接到工程师和管理员表，如下所示：

query.all()

SELECT employee.id AS employee\_id,

engineer.id AS engineer\_id,

manager.id AS manager\_id,

employee.name AS employee\_name,

employee.type AS employee\_type,

engineer.engineer\_info AS engineer\_engineer\_info,

manager.manager\_data AS manager\_manager\_data

FROM employee

LEFT OUTER JOIN engineer

ON employee.id = engineer.id

LEFT OUTER JOIN manager

ON employee.id = manager.id

[]

The section [Using with\_polymorphic](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "with-polymorphic) discusses the [orm.with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic" \o "sqlalchemy.orm.with_polymorphic) function and its configurational variants.

使用with\_polymorphic部分讨论了orm.with\_polymorphic()函数及其配置变体。

**See also**

[Using with\_polymorphic](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "with-polymorphic)

# 4.4 Constructors and Object Initialization

Mapping imposes no restrictions or requirements on the constructor (\_\_init\_\_) method for the class. You are free to require any arguments for the function that you wish, assign attributes to the instance that are unknown to the ORM, and generally do anything else you would normally do when writing a constructor for a Python class.

Mapping对类的构造函数(\_\_init\_\_)方法不施加任何限制或要求。您可以自由地为所需的函数要求任何参数，为ORM未知的实例分配属性，通常在编写Python类的构造函数时通常会执行其他任何操作。

The SQLAlchemy ORM does not call \_\_init\_\_ when recreating objects from database rows. The ORM's process is somewhat akin to the Python standard library's pickle module, invoking the low level \_\_new\_\_ method and then quietly restoring attributes directly on the instance rather than calling \_\_init\_\_.

当从数据库行重新创建对象时，SQLAlchemy ORM不会调用\_\_init\_\_。 ORM的过程有点类似于Python标准库的pickle模块，调用低级\_\_new\_\_方法，然后直接在实例上静态恢复属性，而不是调用\_\_init\_\_。

If you need to do some setup on database-loaded instances before they're ready to use, there is an event hook known as [InstanceEvents.load()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.InstanceEvents.load" \o "sqlalchemy.orm.events.InstanceEvents.load) which can achieve this; it is also available via a class-specific decorator called [orm.reconstructor()](http://docs.sqlalchemy.org/en/rel_1_1/orm/constructors.html" \l "sqlalchemy.orm.reconstructor" \o "sqlalchemy.orm.reconstructor). When using [orm.reconstructor()](http://docs.sqlalchemy.org/en/rel_1_1/orm/constructors.html" \l "sqlalchemy.orm.reconstructor" \o "sqlalchemy.orm.reconstructor), the mapper will invoke the decorated method with no arguments every time it loads or reconstructs an instance of the class. This is useful for recreating transient properties that are normally assigned in \_\_init\_\_:

如果在数据库加载的实例准备使用之前需要进行一些设置，那么就会有一个名为[InstanceEvents.load()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.InstanceEvents.load" \o "sqlalchemy.orm.events.InstanceEvents.load)的事件钩子，可以实现这一点。它也可以通过名为[orm.reconstructor()](http://docs.sqlalchemy.org/en/rel_1_1/orm/constructors.html" \l "sqlalchemy.orm.reconstructor" \o "sqlalchemy.orm.reconstructor)的类特定装饰器来获得。当使用[orm.reconstructor()](http://docs.sqlalchemy.org/en/rel_1_1/orm/constructors.html" \l "sqlalchemy.orm.reconstructor" \o "sqlalchemy.orm.reconstructor)时，mapper将在每次加载或重建类的实例时调用装饰方法，而不使用任何参数。这对于重新创建通常在\_\_init\_\_中分配的瞬态属性很有用：

**from** **sqlalchemy** **import** orm

**class** **MyMappedClass**(object):

**def** \_\_init\_\_(self, data):

self.data = data

*# we need stuff on all instances, but not in the database.*

self.stuff = []

**@orm**.reconstructor

**def** init\_on\_load(self):

self.stuff = []

Above, when obj = MyMappedClass() is executed, the \_\_init\_\_ constructor is invoked normally and the data argument is required. When instances are loaded during a [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) operation as in query(MyMappedClass).one(), init\_on\_load is called.

以上，当obj = MyMappedClass()被执行时，\_\_init\_\_构造函数被正常调用，数据参数是必需的。 在查询操作期间加载实例时，如查询(MyMappedClass).one()中，调用init\_on\_load。

Any method may be tagged as the [orm.reconstructor()](http://docs.sqlalchemy.org/en/rel_1_1/orm/constructors.html" \l "sqlalchemy.orm.reconstructor" \o "sqlalchemy.orm.reconstructor), even the \_\_init\_\_ method itself. It is invoked after all immediate column-level attributes are loaded as well as after eagerly-loaded scalar relationships. Eagerly loaded collections may be only partially populated or not populated at all, depending on the kind of eager loading used.

任何方法都可以被标记为orm.reconstructor()，即使是\_\_init\_\_方法本身也是如此。 在所有直接的列级属性加载之后，以及经过热切的加载的标量关系之后，它将被调用。 根据所使用的渴望加载类型，热门加载的集合可能仅部分填充或不填充。

ORM state changes made to objects at this stage will not be recorded for the next flush operation, so the activity within a reconstructor should be conservative.

在此阶段对对象进行的ORM状态更改将不会被记录下一次刷新操作，因此重建器中的活动应该是保守的。

[orm.reconstructor()](http://docs.sqlalchemy.org/en/rel_1_1/orm/constructors.html" \l "sqlalchemy.orm.reconstructor" \o "sqlalchemy.orm.reconstructor) is a shortcut into a larger system of "instance level" events, which can be subscribed to using the event API - see [InstanceEvents](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.InstanceEvents" \o "sqlalchemy.orm.events.InstanceEvents) for the full API description of these events.

orm.reconstructor()是进入“实例级别”事件的较大系统的快捷方式，可以使用事件API订阅这些事件 - 请参阅InstanceEvents以了解这些事件的完整API描述。

sqlalchemy.orm.**reconstructor**(*fn*)

Decorate a method as the 'reconstructor' hook.

装饰一个方法作为“重建器”钩子。

Designates a method as the "reconstructor", an \_\_init\_\_-like method that will be called by the ORM after the instance has been loaded from the database or otherwise reconstituted.

指定一个方法作为“重建器”，一种\_\_init \_\_类似的方法，在从数据库加载实例或以其他方式重构后，将由ORM调用该方法。

The reconstructor will be invoked with no arguments. Scalar (non-collection) database-mapped attributes of the instance will be available for use within the function. Eagerly-loaded collections are generally not yet available and will usually only contain the first element. ORM state changes made to objects at this stage will not be recorded for the next flush() operation, so the activity within a reconstructor should be conservative.

将无调用重建器。 实例的标量(非收集)数据库映射属性将可用于函数内。 通常热负载集合尚不可用，通常只包含第一个元素。 在此阶段对对象的ORM状态更改将不会被记录下一个flush()操作，因此重建器中的活动应该是保守的。

**See also**

[Constructors and Object Initialization](http://docs.sqlalchemy.org/en/rel_1_1/orm/constructors.html" \l "mapping-constructors)

[InstanceEvents.load()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.InstanceEvents.load" \o "sqlalchemy.orm.events.InstanceEvents.load)

# **4.5 Query API**

# 4.5.1 The Query Object

[Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) is produced in terms of a given [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session), using the [query()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.query" \o "sqlalchemy.orm.session.Session.query) method:

查询根据给定的会话使用query()方法生成：

q = session.query(SomeMappedClass)

Following is the full interface for the [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) object.

以下是Query对象的完整界面。

*class*sqlalchemy.orm.query.**Query**(*entities*, *session=None*)

ORM-level SQL construction object.

ORM级SQL构造对象。

[Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) is the source of all SELECT statements generated by the ORM, both those formulated by end-user query operations as well as by high level internal operations such as related collection loading. It features a generative interface whereby successive calls return a new [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) object, a copy of the former with additional criteria and options associated with it.

查询是由ORM生成的所有SELECT语句的源，由ORM生成的，由最终用户查询操作制定的，以及高级内部操作(如相关的收集加载))。 它具有生成界面，其中连续调用返回一个新的Query对象，前者的副本具有与其相关联的附加条件和选项。

[Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) objects are normally initially generated using the [query()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.query" \o "sqlalchemy.orm.session.Session.query) method of [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session), and in less common cases by instantiating the [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) directly and associating with a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) using the [Query.with\_session()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.with_session" \o "sqlalchemy.orm.query.Query.with_session) method.

For a full walkthrough of [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) usage, see the [Object Relational Tutorial](http://docs.sqlalchemy.org/en/rel_1_1/orm/tutorial.html).

查询对象通常最初使用Session的query()方法生成，而在较不常见的情况下，通过使用Query.with\_session()方法直接实例化查询并与会话相关联。

有关查询用法的完整演练，请参阅对象关系教程。

**\_\_init\_\_**(*entities*, *session=None*)

Construct a [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) directly.

E.g.:

q = Query([User, Address], session=some\_session)

The above is equivalent to:

以上相当于：

q = some\_session.query(User, Address)

|  |  |
| --- | --- |
| **Parameters:** | * ****entities**** – a sequence of entities and/or SQL expressions.一系列实体和/或SQL表达式。 * ****session**** – a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) with which the [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) will be associated. Optional; a [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) can be associated with a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) generatively via the[Query.with\_session()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.with_session" \o "sqlalchemy.orm.query.Query.with_session) method as well.与查询关联的会话。 可选的; 一个查询也可以通过theQuery.with\_session()方法生成一个Session。 |

**See also**

[Session.query()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.query" \o "sqlalchemy.orm.session.Session.query)

[Query.with\_session()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.with_session" \o "sqlalchemy.orm.query.Query.with_session)

**add\_column**(*column*)

Add a column expression to the list of result columns to be returned.

Pending deprecation: [add\_column()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.add_column" \o "sqlalchemy.orm.query.Query.add_column) will be superseded by [add\_columns()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.add_columns" \o "sqlalchemy.orm.query.Query.add_columns).

将列表达式添加到要返回的结果列的列表中。

等待抛弃：add\_column()将被add\_columns()取代。

**add\_columns**(*\*column*)

Add one or more column expressions to the list of result columns to be returned.

将一个或多个列表达式添加到要返回的结果列的列表中。

**add\_entity**(*entity*, *alias=None*)

add a mapped entity to the list of result columns to be returned.

将映射的实体添加到要返回的结果列的列表中。

**all**()

Return the results represented by this Query as a list.

将此查询表示的结果作为列表返回。

This results in an execution of the underlying query.

这将导致基础查询的执行。

**as\_scalar**()

Return the full SELECT statement represented by this [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query), converted to a scalar subquery.

返回由此Query表示的完整SELECT语句，转换为标量子查询。

Analogous to [sqlalchemy.sql.expression.SelectBase.as\_scalar()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.SelectBase.as_scalar" \o "sqlalchemy.sql.expression.SelectBase.as_scalar).

类似于sqlalchemy.sql.expression.SelectBase.as\_scalar()。

*New in version 0.6.5.*

**autoflush**(*setting*)

Return a Query with a specific 'autoflush' setting.

返回具有特定"autoflush"设置的查询。

Note that a Session with autoflush=False will not autoflush, even if this flag is set to True at the Query level. Therefore this flag is usually used only to disable autoflush for a specific Query.

请注意，使用autoflush = False的会话不会自动刷新，即使此标志在查询级别设置为True。 因此，此标志通常仅用于禁用特定查询的自动刷新。

**column\_descriptions**

Return metadata about the columns which would be returned by this [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query).

返回有关此查询返回的列的元数据。

Format is a list of dictionaries:

格式是一个字典列表：

user\_alias = aliased(User, name='user2')

q = sess.query(User, User.id, user\_alias)

*# this expression:*

q.column\_descriptions

*# would return:*

[

{

'name':'User',

'type':User,

'aliased':**False**,

'expr':User,

'entity': User

},

{

'name':'id',

'type':Integer(),

'aliased':**False**,

'expr':User.id,

'entity': User

},

{

'name':'user2',

'type':User,

'aliased':**True**,

'expr':user\_alias,

'entity': user\_alias

}]

**correlate**(*\*args*)

Return a [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) construct which will correlate the given FROM clauses to that of an enclosing [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) or [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select).

返回一个查询结构，它将给定的FROM子句与封闭的Query或select()相关联。

The method here accepts mapped classes, [aliased()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.aliased" \o "sqlalchemy.orm.aliased) constructs, and [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper) constructs as arguments, which are resolved into expression constructs, in addition to appropriate expression constructs.

这里的方法接受映射类，[aliased()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.aliased" \o "sqlalchemy.orm.aliased)结构和[mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper)结构作为参数，除了适当的表达式构造之外，它们被解析为表达式构造。

The correlation arguments are ultimately passed to [Select.correlate()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.correlate" \o "sqlalchemy.sql.expression.Select.correlate) after coercion to expression constructs.

相关参数最终在强制转换为表达式构造后传递给Select.correlate()。

The correlation arguments take effect in such cases as when [Query.from\_self()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.from_self" \o "sqlalchemy.orm.query.Query.from_self) is used, or when a subquery as returned by [Query.subquery()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.subquery" \o "sqlalchemy.orm.query.Query.subquery) is embedded in another [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) construct.

在使用Query.from\_self()或Query.subquery()返回的子查询嵌入到另一个select()结构中时，相关参数会生效。

**count**()

Return a count of rows this Query would return.

返回此Query将返回的行数。

This generates the SQL for this Query as follows:

这将生成此查询的SQL，如下所示：

SELECT count(1) AS count\_1 FROM (

SELECT <rest of query follows...>) AS anon\_1

*Changed in version 0.7:*The above scheme is newly refined as of 0.7b3.

For fine grained control over specific columns to count, to skip the usage of a subquery or otherwise control of the FROM clause, or to use other aggregate functions, use [func](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.func" \o "sqlalchemy.sql.expression.func) expressions in conjunction with [query()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.query" \o "sqlalchemy.orm.session.Session.query), i.e.:

对于对特定列的细粒度控制进行计数，跳过子查询的使用或以其他方式控制FROM子句或使用其他聚合函数，请使用func表达式与query()一起使用，即：

**from** **sqlalchemy** **import** func

*# count User records, without*

*# using a subquery.*

session.query(func.count(User.id))

*# return count of user "id" grouped*

*# by "name"*

session.query(func.count(User.id)).\

group\_by(User.name)

**from** **sqlalchemy** **import** distinct

*# count distinct "name" values*

session.query(func.count(distinct(User.name)))

**cte**(*name=None*, *recursive=False*)

Return the full SELECT statement represented by this [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) represented as a common table expression (CTE).

返回此查询表示的完整SELECT语句，表示为公共表表达式(CTE)。

Parameters and usage are the same as those of the [SelectBase.cte()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.SelectBase.cte" \o "sqlalchemy.sql.expression.SelectBase.cte) method; see that method for further details.

参数和用法与SelectBase.cte()方法相同; 有关详细信息，请参阅该方法。

Here is the [PostgreSQL WITH RECURSIVE example](http://www.postgresql.org/docs/8.4/static/queries-with.html). Note that, in this example, the included\_parts cte and the incl\_alias alias of it are Core selectables, which means the columns are accessed via the .c. attribute. The parts\_alias object is an [orm.aliased()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.aliased" \o "sqlalchemy.orm.aliased) instance of the Partentity, so column-mapped attributes are available directly:

这是PostgreSQL WITH RECURSIVE示例。 请注意，在本示例中，included\_parts cte和incl\_alias的别名是Core可选项，这意味着列可以通过.c访问。 属性。 parts\_alias对象是Partentity的orm.aliased()实例，因此可以直接使用列映射属性：

**from** **sqlalchemy.orm** **import** aliased

**class** **Part**(Base):

\_\_tablename\_\_ = 'part'

part = Column(String, primary\_key=**True**)

sub\_part = Column(String, primary\_key=**True**)

quantity = Column(Integer)

included\_parts = session.query(

Part.sub\_part,

Part.part,

Part.quantity).\

filter(Part.part=="our part").\

cte(name="included\_parts", recursive=**True**)

incl\_alias = aliased(included\_parts, name="pr")parts\_alias = aliased(Part, name="p")included\_parts = included\_parts.union\_all(

session.query(

parts\_alias.sub\_part,

parts\_alias.part,

parts\_alias.quantity).\

filter(parts\_alias.part==incl\_alias.c.sub\_part)

)

q = session.query(

included\_parts.c.sub\_part,

func.sum(included\_parts.c.quantity).

label('total\_quantity')

).\

group\_by(included\_parts.c.sub\_part)

**See also**

[HasCTE.cte()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.HasCTE.cte" \o "sqlalchemy.sql.expression.HasCTE.cte)

**delete**(*synchronize\_session='evaluate'*)

Perform a bulk delete query.

执行批量删除查询。

Deletes rows matched by this query from the database.

从数据库中删除此查询匹配的行。

E.g.:

sess.query(User).filter(User.age == 25).\

delete(synchronize\_session=**False**)

sess.query(User).filter(User.age == 25).\

delete(synchronize\_session='evaluate')

**Warning**

The [Query.delete()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.delete" \o "sqlalchemy.orm.query.Query.delete) method is a "bulk" operation, which bypasses ORM unit-of-work automation in favor of greater performance. ****Please read all caveats and warnings below.****

****Query.delete()方法是一个“批量”操作，它绕过了ORM工作单位自动化，有利于提高性能。 请阅读下面的所有注意事项和警告。****

|  |  |
| --- | --- |
| **Parameters:** | ****synchronize\_session**** –  chooses the strategy for the removal of matched objects from the session. Valid values are:  选择从会话中删除匹配对象的策略。 有效值为：  False - don't synchronize the session. This option is the most efficient and is reliable once the session is expired, which typically occurs after a commit(), or explicitly using expire\_all(). Before the expiration, objects may still remain in the session which were in fact deleted which can lead to confusing results if they are accessed via get() or already loaded collections.  False - 不要同步会话。 该选项是最有效的，并且在会话过期后是可靠的，通常会在commit()或明确使用expire\_all()之后发生。 在到期之前，对象可能仍然保留在实际删除的会话中，如果通过get()或已加载集合访问可能会导致混乱的结果。  'fetch' - performs a select query before the delete to find objects that are matched by the delete query and need to be removed from the session. Matched objects are removed from the session.  'fetch' - 在删除之前执行select查询，以查找与删除查询匹配的对象，并且需要从会话中删除。 匹配的对象将从会话中删除。  'evaluate' - Evaluate the query's criteria in Python straight on the objects in the session. If evaluation of the criteria isn't implemented, an error is raised.  'evaluate' - 在Python中直接对会话中的对象评估查询的标准。 如果没有实施标准的评估，则会出现错误。  The expression evaluator currently doesn't account for differing string collations between the database and Python.  表达式求值器目前并不考虑数据库和Python之间的字符串归类不一致。 |
| **Returns:** | the count of rows matched as returned by the database's "row count" feature. |

**Warning/**警告

****Additional Caveats for bulk query deletes/****批量查询删除的附加警告0.

This method does ****not work for joined inheritance mappings****, since the ****multiple table deletes are not supported by SQL**** as well as that the ****join condition of an inheritance mapper is not automatically rendered****. Care must be taken in any multiple-table delete to first accommodate via some other means how the related table will be deleted, as well as to explicitly include the joining condition between those tables, even in mappings where this is normally automatic. E.g. if a class Engineer subclasses Employee, a DELETE against the Employee table would look like:

此方法对于连接的继承映射不起作用，因为SQL不支持多表删除，并且不会自动呈现继承映射器的连接条件。 在任何多表删除中，必须注意首先通过其他方式容纳相关表将被删除，以及明确地包括这些表之间的连接条件，即使是通常是自动的映射。 例如。 如果一个类工程师子类Employee，则对于Employee表的DELETE将如下所示：

session.query(Engineer).\

filter(Engineer.id == Employee.id).\

filter(Employee.name == 'dilbert').\

delete()

However the above SQL will not delete from the Engineer table, unless an ON DELETE CASCADE rule is established in the database to handle it.

但是，上述SQL不会从Engineer表中删除，除非在数据库中建立ON DELETE CASCADE规则来处理它。

Short story, ****do not use this method for joined inheritance mappings unless you have taken the additional steps to make this feasible****.

简短的故事，不要使用这种方法进行连接继承映射，除非你采取了额外的步骤，使之成为可行的。

The polymorphic identity WHERE criteria is ****not**** included for single- or joined- table updates - this must be added ****manually**** even for single table inheritance.

对于单表或更新表，不包括多态标识WHERE标准，即使对于单表继承也必须手动添加。

The method does ****not**** offer in-Python cascading of relationships - it is assumed that ON DELETE CASCADE/SET NULL/etc. is configured for any foreign key references which require it, otherwise the database may emit an integrity violation if foreign key references are being enforced.

该方法不提供Python中的关联级联 - 假设ON DELETE CASCADE / SET NULL / etc。被配置为需要它的任何外键引用，否则如果正在执行外键引用，数据库可能会发出完整性违例。

After the DELETE, dependent objects in the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) which were impacted by an ON DELETE may not contain the current state, or may have been deleted. This issue is resolved once the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) is expired, which normally occurs upon [Session.commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.commit" \o "sqlalchemy.orm.session.Session.commit) or can be forced by using [Session.expire\_all()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.expire_all" \o "sqlalchemy.orm.session.Session.expire_all). Accessing an expired object whose row has been deleted will invoke a SELECT to locate the row; when the row is not found, an [ObjectDeletedError](http://docs.sqlalchemy.org/en/rel_1_1/orm/exceptions.html" \l "sqlalchemy.orm.exc.ObjectDeletedError" \o "sqlalchemy.orm.exc.ObjectDeletedError) is raised.

在DELETE之后，会话中的依赖对象受到ON DELETE的影响可能不包含当前状态，或者可能已被删除。一旦会话过期，这个问题就会被解决，这通常发生在Session.commit()上，或者可以通过使用Session.expire\_all()强制执行。访问行已删除的过期对象将调用SELECT以找到该行;当行找不到时，会引发一个ObjectDeletedError。

The 'fetch' strategy results in an additional SELECT statement emitted and will significantly reduce performance.

"fetch"策略会导致发出额外的SELECT语句，并将显着降低性能。

The 'evaluate' strategy performs a scan of all matching objects within the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session); if the contents of the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) are expired, such as via a proceeding [Session.commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.commit" \o "sqlalchemy.orm.session.Session.commit) call, ****this will result in SELECT queries emitted for every matching object****.

"评估"策略对会话内的所有匹配对象执行扫描;如果会话的内容已过期，例如通过进行的Session.commit()调用，这将导致为每个匹配对象发出的SELECT查询。

The [MapperEvents.before\_delete()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents.before_delete" \o "sqlalchemy.orm.events.MapperEvents.before_delete) and [MapperEvents.after\_delete()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents.after_delete" \o "sqlalchemy.orm.events.MapperEvents.after_delete) events ****are not invoked**** from this method. Instead, the[SessionEvents.after\_bulk\_delete()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_bulk_delete" \o "sqlalchemy.orm.events.SessionEvents.after_bulk_delete) method is provided to act upon a mass DELETE of entity rows.

MapperEvents.before\_delete()和MapperEvents.after\_delete()事件不会从此方法中调用。相反，提供了SessionEvents.after\_bulk\_delete()方法来对实体行进行大量删除。

**See also**

[Query.update()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.update" \o "sqlalchemy.orm.query.Query.update)

[Inserts, Updates and Deletes](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html" \l "inserts-and-updates) - Core SQL tutorial

**distinct**(*\*criterion*)

Apply a DISTINCT to the query and return the newly resulting Query.

**Note**

The [distinct()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator.distinct" \o "sqlalchemy.orm.interfaces.PropComparator.distinct) call includes logic that will automatically add columns from the ORDER BY of the query to the columns clause of the SELECT statement, to satisfy the common need of the database backend that ORDER BY columns be part of the SELECT list when DISTINCT is used. These columns *are not* added to the list of columns actually fetched by the [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query), however, so would not affect results. The columns are passed through when using the [Query.statement](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.statement" \o "sqlalchemy.orm.query.Query.statement) accessor, however.

distinct()调用包括将自动将查询的ORDER BY中的列添加到SELECT语句的columns子句的逻辑，以满足数据库后端的常见需求，ORDER BY列作为SELECT列表的一部分，当DISTINCT为 用过的。 这些列不会添加到由Query实际获取的列的列表中，因此不会影响结果。 但是，在使用Query.statement访问器时，这些列被传递。

|  |  |
| --- | --- |
| **Parameters:** | ****\*expr**** – optional column expressions. When present, the PostgreSQL dialect will render a DISTINCT ON (<expressions>>) construct. |

**enable\_assertions**(*value*)

Control whether assertions are generated.

控制是否生成断言。

When set to False, the returned Query will not assert its state before certain operations, including that LIMIT/OFFSET has not been applied when filter() is called, no criterion exists when get() is called, and no "from\_statement()" exists when filter()/order\_by()/group\_by() etc. is called. This more permissive mode is used by custom Query subclasses to specify criterion or other modifiers outside of the usual usage patterns.

当设置为False时，返回的查询将不会在某些操作之前断言其状态，包括调用filter()时未应用LIMIT / OFFSET，调用get()时没有标准，并且没有“from\_statement()” 当调用filter()/ order\_by()/ group\_by()等时存在)。 自定义Query子类使用这种更宽容的模式来指定常规使用模式之外的标准或其他修饰符。

Care should be taken to ensure that the usage pattern is even possible. A statement applied by from\_statement() will override any criterion set by filter() or order\_by(), for example.

应注意确保使用模式是可能的。 例如，by\_statement()应用的语句将覆盖由filter()或order\_by()设置的任何标准。

**enable\_eagerloads**(*value*)

Control whether or not eager joins and subqueries are rendered.

控制是否呈现渴望加入和子查询。

When set to False, the returned Query will not render eager joins regardless of [joinedload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.joinedload" \o "sqlalchemy.orm.joinedload), [subqueryload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.subqueryload" \o "sqlalchemy.orm.subqueryload) options or mapper-level lazy='joined'/lazy='subquery' configurations.

当设置为False时，无论joinload()，subqueryload()选项或mapper-level lazy ='joined'/ lazy ='subquery'配置如何，返回的查询将不会呈现eager连接。

This is used primarily when nesting the Query's statement into a subquery or other selectable, or when using [Query.yield\_per()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.yield_per" \o "sqlalchemy.orm.query.Query.yield_per).

这主要用于将Query语句嵌套到子查询或其他可选项中，或在使用Query.yield\_per()时使用。

**except\_**(*\*q*)

Produce an EXCEPT of this Query against one or more queries.

针对一个或多个查询生成此查询的EXCEPT。

Works the same way as [union()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.union" \o "sqlalchemy.orm.query.Query.union). See that method for usage examples.

工作方式与union()相同。 请参阅该方法的使用示例。

**except\_all**(*\*q*)

Produce an EXCEPT ALL of this Query against one or more queries.

根据一个或多个查询生成一个除此查询的所有内容。

Works the same way as [union()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.union" \o "sqlalchemy.orm.query.Query.union). See that method for usage examples.

工作方式与union()相同。 请参阅该方法的使用示例。

**execution\_options**(*\*\*kwargs*)

Set non-SQL options which take effect during execution.

设置在执行期间生效的非SQL选项。

The options are the same as those accepted by [Connection.execution\_options()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.execution_options" \o "sqlalchemy.engine.Connection.execution_options).

选项与Connection.execution\_options()接受的选项相同。

Note that the stream\_results execution option is enabled automatically if the [yield\_per()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.yield_per" \o "sqlalchemy.orm.query.Query.yield_per) method is used.

请注意，如果使用yield\_per()方法，则会自动启用stream\_results执行选项。

**exists**()

A convenience method that turns a query into an EXISTS subquery of the form EXISTS (SELECT 1 FROM … WHERE …).

一种将查询转换为EXISTS(SELECT 1 FROM ... WHERE ...)形式的EXISTS子查询的便捷方法。

e.g.:

q = session.query(User).filter(User.name == 'fred')session.query(q.exists())

Producing SQL similar to:

生成SQL类似于：

SELECT EXISTS (

SELECT 1 FROM users WHERE users.name = :name\_1) AS anon\_1

The EXISTS construct is usually used in the WHERE clause:

EXISTS结构通常用于WHERE子句：

session.query(User.id).filter(q.exists()).scalar()

Note that some databases such as SQL Server don't allow an EXISTS expression to be present in the columns clause of a SELECT. To select a simple boolean value based on the exists as a WHERE, use [literal()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.literal" \o "sqlalchemy.sql.expression.literal):

请注意，某些数据库(如SQL Server)不允许EXISTS表达式存在于SELECT的columns子句中。 要基于存在为WHERE来选择一个简单的布尔值，请使用literal()：

**from** **sqlalchemy** **import** literal

session.query(literal(**True**)).filter(q.exists()).scalar()

*New in version 0.8.1.*

**filter**(*\*criterion*)

apply the given filtering criterion to a copy of this [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query), using SQL expressions.

使用SQL表达式将给定的过滤条件应用于此Query的副本。

e.g.:

例如。：

session.query(MyClass).filter(MyClass.name == 'some name')

Multiple criteria may be specified as comma separated; the effect is that they will be joined together using the [and\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.and_" \o "sqlalchemy.sql.expression.and_) function:

多个标准可以指定为逗号分隔; 效果是它们将使用and\_()函数连接在一起：

session.query(MyClass).\

filter(MyClass.name == 'some name', MyClass.id > 5)

The criterion is any SQL expression object applicable to the WHERE clause of a select. String expressions are coerced into SQL expression constructs via the [text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text) construct.

该标准是适用于select的WHERE子句的任何SQL表达式对象。 字符串表达式通过text()构造强制转换为SQL表达式构造。

**See also**

[Query.filter\_by()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.filter_by" \o "sqlalchemy.orm.query.Query.filter_by) - filter on keyword expressions.

**filter\_by**(*\*\*kwargs*)

apply the given filtering criterion to a copy of this [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query), using keyword expressions.

使用关键词表达式将给定的过滤条件应用于此Query的副本。

e.g.:

session.query(MyClass).filter\_by(name = 'some name')

Multiple criteria may be specified as comma separated; the effect is that they will be joined together using the [and\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.and_" \o "sqlalchemy.sql.expression.and_) function:

多个标准可以指定为逗号分隔; 效果是它们将使用and\_()函数连接在一起：

session.query(MyClass).\

filter\_by(name = 'some name', id = 5)

The keyword expressions are extracted from the primary entity of the query, or the last entity that was the target of a call to [Query.join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join).

关键词表达式是从查询的主实体中提取的，或作为调用Query.join()的目标的最后一个实体。

**See also**

[Query.filter()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.filter" \o "sqlalchemy.orm.query.Query.filter) - filter on SQL expressions.

**first**()

Return the first result of this Query or None if the result doesn't contain any row.

如果结果不包含任何行，则返回此查询的第一个结果或无返回。

first() applies a limit of one within the generated SQL, so that only one primary entity row is generated on the server side (note this may consist of multiple result rows if join-loaded collections are present).

first()在生成的SQL中应用一个限制，因此在服务器端只生成一个主实体行(注意，如果连接加载的集合存在，则可以包含多个结果行)。

Calling [Query.first()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.first" \o "sqlalchemy.orm.query.Query.first) results in an execution of the underlying query.

调用Query.first()导致底层查询的执行。

**See also**

[Query.one()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.one" \o "sqlalchemy.orm.query.Query.one)

[Query.one\_or\_none()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.one_or_none" \o "sqlalchemy.orm.query.Query.one_or_none)

**from\_self**(*\*entities*)

return a Query that selects from this Query's SELECT statement.

返回从该查询的SELECT语句中选择的查询。

[Query.from\_self()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.from_self" \o "sqlalchemy.orm.query.Query.from_self) essentially turns the SELECT statement into a SELECT of itself. Given a query such as:

Query.from\_self()基本上将SELECT语句转换成它自己的SELECT。 给出一个查询，如：

q = session.query(User).filter(User.name.like('e%'))

Given the [Query.from\_self()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.from_self" \o "sqlalchemy.orm.query.Query.from_self) version:

给定Query.from\_self()版本：

q = session.query(User).filter(User.name.like('e%')).from\_self()

This query renders as:

此查询呈现为：

**SELECT** anon\_1.user\_id **AS** anon\_1\_user\_id,

anon\_1.user\_name **AS** anon\_1\_user\_name

**FROM** (**SELECT** "user".id **AS** user\_id, "user".name **AS** user\_name

**FROM** "user"**WHERE** "user".name **LIKE** :name\_1) **AS** anon\_1

There are lots of cases where [Query.from\_self()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.from_self" \o "sqlalchemy.orm.query.Query.from_self) may be useful. A simple one is where above, we may want to apply a row LIMIT to the set of user objects we query against, and then apply additional joins against that row-limited set:

有很多情况下Query.from\_self()可能是有用的。 一个简单的一个是上面的地方，我们可能想要对我们查询的一组用户对象应用一行LIMIT，然后对该行限制集合应用额外的连接：

q = session.query(User).filter(User.name.like('e%')).\

limit(5).from\_self().\

join(User.addresses).filter(Address.email.like('q%'))

The above query joins to the Address entity but only against the first five results of the User query:

上述查询连接到地址实体，但仅针对用户查询的前五个结果：

**SELECT** anon\_1.user\_id **AS** anon\_1\_user\_id,

anon\_1.user\_name **AS** anon\_1\_user\_name

**FROM** (**SELECT** "user".id **AS** user\_id, "user".name **AS** user\_name

**FROM** "user"**WHERE** "user".name **LIKE** :name\_1

**LIMIT** :param\_1) **AS** anon\_1

**JOIN** address **ON** anon\_1.user\_id = address.user\_id

**WHERE** address.email **LIKE** :email\_1

****Automatic Aliasing****

Another key behavior of [Query.from\_self()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.from_self" \o "sqlalchemy.orm.query.Query.from_self) is that it applies ****automatic aliasing**** to the entities inside the subquery, when they are referenced on the outside. Above, if we continue to refer to the User entity without any additional aliasing applied to it, those references wil be in terms of the subquery:

Query.from\_self()的另一个关键行为是它在外部引用时将自动别名应用于子查询中的实体。 以上，如果我们继续引用用户实体，而不附加任何额外的别名，那么这些引用将来自子查询：

q = session.query(User).filter(User.name.like('e%')).\

limit(5).from\_self().\

join(User.addresses).filter(Address.email.like('q%')).\

order\_by(User.name)

The ORDER BY against User.name is aliased to be in terms of the inner subquery:

对于User.name，ORDER BY被别名为内部子查询：

**SELECT** anon\_1.user\_id **AS** anon\_1\_user\_id,

anon\_1.user\_name **AS** anon\_1\_user\_name

**FROM** (**SELECT** "user".id **AS** user\_id, "user".name **AS** user\_name

**FROM** "user"**WHERE** "user".name **LIKE** :name\_1

**LIMIT** :param\_1) **AS** anon\_1**JOIN** address **ON** anon\_1.user\_id = address.user\_id

**WHERE** address.email **LIKE** :email\_1 **ORDER** **BY** anon\_1.user\_name

The automatic aliasing feature only works in a ****limited**** way, for simple filters and orderings. More ambitious constructions such as referring to the entity in joins should prefer to use explicit subquery objects, typically making use of the [Query.subquery()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.subquery" \o "sqlalchemy.orm.query.Query.subquery) method to produce an explicit subquery object. Always test the structure of queries by viewing the SQL to ensure a particular structure does what's expected!

自动混叠功能只能以有限的方式工作，简单的过滤器和排序。 更加雄心勃勃的结构，例如引用连接中的实体，应优先使用显式子查询对象，通常使用Query.subquery()方法来生成一个显式的子查询对象。 始终通过查看SQL来测试查询的结构，以确保特定的结构有所期待！

****Changing the Entities****

[Query.from\_self()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.from_self" \o "sqlalchemy.orm.query.Query.from_self) also includes the ability to modify what columns are being queried. In our example, we want User.id to be queried by the inner query, so that we can join to the Address entity on the outside, but we only wanted the outer query to return the Address.email column:

Query.from\_self()还包括修改正在被查询的列的功能。 在我们的示例中，我们希望内部查询查询User.id，以便我们可以加入到外部的Address实体，但是我们只希望外部查询返回Address.email列：

q = session.query(User).filter(User.name.like('e%')).\

limit(5).from\_self(Address.email).\

join(User.addresses).filter(Address.email.like('q%'))

yielding:

**SELECT** address.email **AS** address\_email

**FROM** (**SELECT** "user".id **AS** user\_id, "user".name **AS** user\_name

**FROM** "user"**WHERE** "user".name **LIKE** :name\_1

**LIMIT** :param\_1) **AS** anon\_1**JOIN** address **ON** anon\_1.user\_id = address.user\_id

**WHERE** address.email **LIKE** :email\_1

****Looking out for Inner / Outer Columns****

寻找内/外栏

Keep in mind that when referring to columns that originate from inside the subquery, we need to ensure they are present in the columns clause of the subquery itself; this is an ordinary aspect of SQL. For example, if we wanted to load from a joined entity inside the subquery using [contains\_eager()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.contains_eager" \o "sqlalchemy.orm.contains_eager), we need to add those columns. Below illustrates a join of Address to User, then a subquery, and then we'd like [contains\_eager()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.contains_eager" \o "sqlalchemy.orm.contains_eager) to access the User columns:

请记住，当引用来自子查询内部的列时，我们需要确保它们存在于子查询本身的columns子句中; 这是SQL的一个普通方面。 例如，如果我们想使用contains\_eager()从子查询中的连接实体加载，则需要添加这些列。 下面说明了一个地址到用户的连接，然后是一个子查询，然后我们想要使用contains\_eager()来访问用户列：

q = session.query(Address).join(Address.user).\

filter(User.name.like('e%'))

q = q.add\_entity(User).from\_self().\

options(contains\_eager(Address.user))

We use [Query.add\_entity()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.add_entity" \o "sqlalchemy.orm.query.Query.add_entity) above ****before**** we call [Query.from\_self()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.from_self" \o "sqlalchemy.orm.query.Query.from_self) so that the User columns are present in the inner subquery, so that they are available to the [contains\_eager()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.contains_eager" \o "sqlalchemy.orm.contains_eager) modifier we are using on the outside, producing:

在调用Query.from\_self()之前，我们使用Query.add\_entity()，以便User列存在于内部子查询中，以便它们可用于外部使用的contains\_eager()修饰符，从而产生：

**SELECT** anon\_1.address\_id **AS** anon\_1\_address\_id,

anon\_1.address\_email **AS** anon\_1\_address\_email,

anon\_1.address\_user\_id **AS** anon\_1\_address\_user\_id,

anon\_1.user\_id **AS** anon\_1\_user\_id,

anon\_1.user\_name **AS** anon\_1\_user\_name

**FROM** (

**SELECT** address.id **AS** address\_id,

address.email **AS** address\_email,

address.user\_id **AS** address\_user\_id,

"user".id **AS** user\_id,

"user".name **AS** user\_name

**FROM** address **JOIN** "user" **ON** "user".id = address.user\_id

**WHERE** "user".name **LIKE** :name\_1) **AS** anon\_1

If we didn't call add\_entity(User), but still asked [contains\_eager()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.contains_eager" \o "sqlalchemy.orm.contains_eager) to load the User entity, it would be forced to add the table on the outside without the correct join criteria - note the anon1, "user" phrase at the end:

如果我们没有调用add\_entity(User)，但是还是要求contains\_eager()来加载User实体，那么在没有正确连接条件的情况下，它将被强制添加到外部表中 - 请注意anon1，“user” 结束：

*-- incorrect query***SELECT** anon\_1.address\_id **AS** anon\_1\_address\_id,

anon\_1.address\_email **AS** anon\_1\_address\_email,

anon\_1.address\_user\_id **AS** anon\_1\_address\_user\_id,

"user".id **AS** user\_id,

"user".name **AS** user\_name

**FROM** (

**SELECT** address.id **AS** address\_id,

address.email **AS** address\_email,

address.user\_id **AS** address\_user\_id

**FROM** address **JOIN** "user" **ON** "user".id = address.user\_id

**WHERE** "user".name **LIKE** :name\_1) **AS** anon\_1, "user"

|  |  |
| --- | --- |
| **Parameters:** | ****\*entities**** – optional list of entities which will replace those being selected.将替代被选择的实体的可选列表。 |

**from\_statement**(*statement*)

Execute the given SELECT statement and return results.

执行给定的SELECT语句并返回结果。

This method bypasses all internal statement compilation, and the statement is executed without modification.

此方法绕过所有内部语句编译，并且该语句在不进行修改的情况下执行。

The statement is typically either a [text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text) or [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) construct, and should return the set of columns appropriate to the entity class represented by this [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query).

该语句通常是text()或select()结构，并且应该返回适合于此查询表示的实体类的列集合。

**See also**

[Using Textual SQL](http://docs.sqlalchemy.org/en/rel_1_1/orm/tutorial.html" \l "orm-tutorial-literal-sql) - usage examples in the ORM tutorial

**get**(*ident*)

Return an instance based on the given primary key identifier, or None if not found.

根据给定的主键标识符返回一个实例，如果没有找到，返回None。

E.g.:

my\_user = session.query(User).get(5)

some\_object = session.query(VersionedFoo).get((5, 10))

[get()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.get" \o "sqlalchemy.orm.query.Query.get) is special in that it provides direct access to the identity map of the owning [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session). If the given primary key identifier is present in the local identity map, the object is returned directly from this collection and no SQL is emitted, unless the object has been marked fully expired. If not present, a SELECT is performed in order to locate the object.

get()是特殊的，它提供对拥有Session的身份映射的直接访问。如果给定的主键标识符存在于本地标识映射中，则该对象将直接从该集合返回，并且不会发出SQL，除非该对象已被标记为已满。如果不存在，则执行SELECT以定位对象。

[get()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.get" \o "sqlalchemy.orm.query.Query.get) also will perform a check if the object is present in the identity map and marked as expired - a SELECT is emitted to refresh the object as well as to ensure that the row is still present. If not, [ObjectDeletedError](http://docs.sqlalchemy.org/en/rel_1_1/orm/exceptions.html" \l "sqlalchemy.orm.exc.ObjectDeletedError" \o "sqlalchemy.orm.exc.ObjectDeletedError) is raised.

如果对象存在于身份映射中并被标记为已过期，则get()也将执行检查 - 发出SELECT以刷新对象以及确保该行仍然存在。如果没有，则引发ObjectDeletedError。

[get()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.get" \o "sqlalchemy.orm.query.Query.get) is only used to return a single mapped instance, not multiple instances or individual column constructs, and strictly on a single primary key value. The originating [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) must be constructed in this way, i.e. against a single mapped entity, with no additional filtering criterion. Loading options via [options()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.options" \o "sqlalchemy.orm.query.Query.options) may be applied however, and will be used if the object is not yet locally present.

get()仅用于返回单个映射实例，而不是多个实例或单个列构造，并严格依赖于单个主键值。必须以这种方式构建始发查询，即针对单个映射的实体，而不需要额外的过滤标准。但是，可以通过options()加载选项，如果对象尚未在本地存在，则可以使用它。

A lazy-loading, many-to-one attribute configured by [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship), using a simple foreign-key-to-primary-key criterion, will also use an operation equivalent to [get()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.get" \o "sqlalchemy.orm.query.Query.get) in order to retrieve the target value from the local identity map before querying the database. See [Relationship Loading Techniques](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html)for further details on relationship loading.

使用简单的外键到主键标准，由relationship()配置的延迟加载，多对一属性也将使用等效于get()的操作，以便从查询数据库之前的本地身份映射。有关关系加载的详细信息，请参阅关系加载技术。

|  |  |
| --- | --- |
| **Parameters:** | ****ident**** – A scalar or tuple value representing the primary key. For a composite primary key, the order of identifiers corresponds in most cases to that of the mapped [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object's primary key columns. For a [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper) that was given the primary key argument during construction, the order of identifiers corresponds to the elements present in this collection.表示主键的标量或元组值。 对于复合主键，标识符的顺序在大多数情况下对应于映射的Table对象的主键列的顺序。 对于在构造期间被赋予主键参数的mapper()，标识符的顺序对应于该集合中存在的元素。 |
| **Returns:** | The object instance, or None. |

**group\_by**(*\*criterion*)

apply one or more GROUP BY criterion to the query and return the newly resulting [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query)

将一个或多个GROUP BY标准应用于查询并返回新生成的Query

All existing GROUP BY settings can be suppressed by passing None - this will suppress any GROUP BY configured on mappers as well.

通过传递None可以抑制所有现有的GROUP BY设置 - 这将抑制在映射器上配置的任何GROUP BY。

*New in version 1.1:*GROUP BY can be cancelled by passing None, in the same way as ORDER BY.

**having**(*criterion*)

apply a HAVING criterion to the query and return the newly resulting [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query).

对查询应用HAVING标准，并返回新生成的Query。

[having()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.having" \o "sqlalchemy.orm.query.Query.having) is used in conjunction with [group\_by()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.group_by" \o "sqlalchemy.orm.query.Query.group_by).

having()与group\_by()一起使用。

HAVING criterion makes it possible to use filters on aggregate functions like COUNT, SUM, AVG, MAX, and MIN, eg.:

HAVING标准使得可以对聚合函数(如COUNT，SUM，AVG，MAX和MIN)使用过滤器，例如：

q = session.query(User.id).\

join(User.addresses).\

group\_by(User.id).\

having(func.count(Address.id) > 2)

**instances**(*cursor*, *\_Query\_\_context=None*)

Given a ResultProxy cursor as returned by connection.execute(), return an ORM result as an iterator.

给定由connection.execute()返回的ResultProxy游标，返回ORM结果作为迭代器。

e.g.:

result = engine.execute("select \* from users")**for** u **in** session.query(User).instances(result):

print u

**intersect**(*\*q*)

Produce an INTERSECT of this Query against one or more queries.

针对一个或多个查询产生此查询的INTERSECT。

Works the same way as [union()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.union" \o "sqlalchemy.orm.query.Query.union). See that method for usage examples.

工作方式与union()相同。 请参阅该方法的使用示例。

**intersect\_all**(*\*q*)

Produce an INTERSECT ALL of this Query against one or more queries.

针对一个或多个查询生成INTERSECT ALL这个查询。

Works the same way as [union()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.union" \o "sqlalchemy.orm.query.Query.union). See that method for usage examples.

工作方式与union()相同。 请参阅该方法的使用示例。

**join**(*\*props*, *\*\*kwargs*)

Create a SQL JOIN against this [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) object's criterion and apply generatively, returning the newly resulting [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query).

根据此Query对象的标准创建SQL JOIN并生成应用程序，返回新生成的Query。

****Simple Relationship Joins****

Consider a mapping between two classes User and Address, with a relationship User.addresses representing a collection of Address objects associated with each User. The most common usage of [join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join) is to create a JOIN along this relationship, using the User.addresses attribute as an indicator for how this should occur:

考虑两个用户和地址之间的映射关系，User.addresses表示与每个用户相关联的Address对象的集合。 join()的最常用的用法是沿着这种关系创建一个JOIN，使用User.addresses属性作为如何发生的一个指标：

q = session.query(User).join(User.addresses)

Where above, the call to [join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join) along User.addresses will result in SQL equivalent to:

在上面的地方，对User.addresses的join()的调用将导致SQL等效于：

SELECT user.\* FROM user JOIN address ON user.id = address.user\_id

In the above example we refer to User.addresses as passed to [join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join) as the *on clause*, that is, it indicates how the "ON" portion of the JOIN should be constructed. For a single-entity query such as the one above (i.e. we start by selecting only from User and nothing else), the relationship can also be specified by its string name:

在上面的例子中，我们将User.addresses作为on子句传递给join()，即表示如何构造JOIN的“ON”部分。 对于单一实体查询(例如上面的单个实体查询)(即，我们从仅从用户而不是别的选择开始)，关系也可以由其字符串名称指定：

q = session.query(User).join("addresses")

[join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join) can also accommodate multiple "on clause" arguments to produce a chain of joins, such as below where a join across four related entities is constructed:

join()也可以容纳多个“on子句”参数，以产生一个连接链，例如下面构建四个相关实体的连接：

q = session.query(User).join("orders", "items", "keywords")

The above would be shorthand for three separate calls to [join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join), each using an explicit attribute to indicate the source entity:

以上是对join()的三个单独调用的简写，每个调用都使用显式属性来指示源实体：

q = session.query(User).\

join(User.orders).\

join(Order.items).\

join(Item.keywords)

****Joins to a Target Entity or Selectable****

A second form of [join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join) allows any mapped entity or core selectable construct as a target. In this usage, [join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join) will attempt to create a JOIN along the natural foreign key relationship between two entities:

[join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join)的第二种形式允许任何映射的实体或核心可选构造作为目标。 在这种用法中，[join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join)将尝试沿着两个实体之间的自然外键关系创建一个JOIN：

q = session.query(User).join(Address)

The above calling form of [join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join) will raise an error if either there are no foreign keys between the two entities, or if there are multiple foreign key linkages between them. In the above calling form, [join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join) is called upon to create the "on clause" automatically for us. The target can be any mapped entity or selectable, such as a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table):

如果两个实体之间没有外键，或者它们之间有多个外键连接，则上述调用形式的join()将引发错误。 在上面的调用形式中，调用join()来为我们自动创建“on子句”。 目标可以是任何映射实体或可选择，如表：

q = session.query(User).join(addresses\_table)

****Joins to a Target with an ON Clause****

The third calling form allows both the target entity as well as the ON clause to be passed explicitly. Suppose for example we wanted to join to Address twice, using an alias the second time. We use [aliased()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.aliased" \o "sqlalchemy.orm.aliased) to create a distinct alias of Address, and join to it using the target, onclause form, so that the alias can be specified explicitly as the target along with the relationship to instruct how the ON clause should proceed:

第三个调用表单允许目标实体以及ON子句被显式传递。 假设我们想加入Address两次，第二次使用别名。 我们使用aliased()来创建一个不同的地址别名，并使用target，onclause表单加入它，以便可以将该别名明确指定为目标以及关系来指示ON子句如何继续：

a\_alias = aliased(Address)

q = session.query(User).\

join(User.addresses).\

join(a\_alias, User.addresses).\

filter(Address.email\_address=='ed@foo.com').\

filter(a\_alias.email\_address=='ed@bar.com')

Where above, the generated SQL would be similar to:

SELECT user.\* FROM user

JOIN address ON user.id = address.user\_id

JOIN address AS address\_1 ON user.id=address\_1.user\_id

WHERE address.email\_address = :email\_address\_1

AND address\_1.email\_address = :email\_address\_2

The two-argument calling form of [join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join) also allows us to construct arbitrary joins with SQL-oriented "on clause" expressions, not relying upon configured relationships at all. Any SQL expression can be passed as the ON clause when using the two-argument form, which should refer to the target entity in some way as well as an applicable source entity:

join()的双参数调用形式还允许我们使用面向SQL的“on子句”表达式构造任意连接，而不是依赖于配置的关系。 当使用双参数形式时，任何SQL表达式都可以作为ON子句传递，该参数应以某种方式引用目标实体以及适用的源实体：

q = session.query(User).join(Address, User.id==Address.user\_id)

*Changed in version 0.7:*In SQLAlchemy 0.6 and earlier, the two argument form of [join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join) requires the usage of a tuple: query(User).join((Address,User.id==Address.user\_id)). This calling form is accepted in 0.7 and further, though is not necessary unless multiple join conditions are passed to a single [join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join) call, which itself is also not generally necessary as it is now equivalent to multiple calls (this wasn't always the case).

在0.7版本中更改：在SQLAlchemy 0.6及更早版本中，join()的两个参数形式需要使用一个元组：query(User).join((Address，User.id == Address.user\_id))。 这种调用形式在0.7中被接受，而且除了多个连接条件被传递给单个join()调用之外，并不是必需的，而且它也不是一般必需的，因为它现在等效于多个调用(这并不总是 案件)。

****Advanced Join Targeting and Adaption****

There is a lot of flexibility in what the "target" can be when using [join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join). As noted previously, it also accepts [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) constructs and other selectables such as [alias()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.alias" \o "sqlalchemy.sql.expression.alias) and [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) constructs, with either the one or two-argument forms:

在使用join()时，“目标”可以有很大的灵活性。 如前所述，它还接受表结构和其他可选参数，如alias()和select()结构，使用一个或两个参数的形式：

addresses\_q = select([Address.user\_id]).\

where(Address.email\_address.endswith("@bar.com")).\

alias()

q = session.query(User).\

join(addresses\_q, addresses\_q.c.user\_id==User.id)

[join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join) also features the ability to *adapt* a [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) -driven ON clause to the target selectable. Below we construct a JOIN from User to a subquery against Address, allowing the relationship denoted by User.addresses to *adapt* itself to the altered target:

join()还具有适应关系()-driven ON子句到目标可选择的能力。 下面我们根据Address构建一个从User到一个子查询的JOIN，允许User.addresses表示的关系适应改变的目标：

address\_subq = session.query(Address).\

filter(Address.email\_address == 'ed@foo.com').\

subquery()

q = session.query(User).join(address\_subq, User.addresses)

Producing SQL similar to:

SELECT user.\* FROM user

JOIN (

SELECT address.id AS id,

address.user\_id AS user\_id,

address.email\_address AS email\_address

FROM address

WHERE address.email\_address = :email\_address\_1

) AS anon\_1 ON user.id = anon\_1.user\_id

The above form allows one to fall back onto an explicit ON clause at any time:

q = session.query(User).\

join(address\_subq, User.id==address\_subq.c.user\_id)

****Controlling what to Join From****

While [join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join) exclusively deals with the "right" side of the JOIN, we can also control the "left" side, in those cases where it's needed, using [select\_from()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.select_from" \o "sqlalchemy.orm.query.Query.select_from). Below we construct a query against Address but can still make usage of User.addresses as our ON clause by instructing the [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) to select first from the User entity:

当join()专门处理JOIN的“正确”方面时，我们还可以使用select\_from()来控制“左”方面，在需要的情况下。 下面我们构造一个针对Address的查询，但仍然可以通过指示Query从User实体中首先选择来将User.addresses用作我们的ON子句：

q = session.query(Address).select\_from(User).\

join(User.addresses).\

filter(User.name == 'ed')

Which will produce SQL similar to:

这将产生类似于SQL的SQL：

SELECT address.\* FROM user

JOIN address ON user.id=address.user\_id

WHERE user.name = :name\_1

****Constructing Aliases Anonymously****

[join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join) can construct anonymous aliases using the aliased=True flag. This feature is useful when a query is being joined algorithmically, such as when querying self-referentially to an arbitrary depth:

join()可以使用aliased = True标志来构造匿名别名。 当查询以算术方式加入时，这一功能非常有用，例如在自行查询到任意深度时：

q = session.query(Node).\

join("children", "children", aliased=**True**)

When aliased=True is used, the actual "alias" construct is not explicitly available. To work with it, methods such as [Query.filter()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.filter" \o "sqlalchemy.orm.query.Query.filter) will adapt the incoming entity to the last join point:

当使用aliased = True时，实际的“别名”结构不可用。 要使用它，诸如Query.filter()之类的方法将使进入的实体适应最后一个连接点：

q = session.query(Node).\

join("children", "children", aliased=**True**).\

filter(Node.name == 'grandchild 1')

When using automatic aliasing, the from\_joinpoint=True argument can allow a multi-node join to be broken into multiple calls to [join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join), so that each path along the way can be further filtered:

当使用自动别名时，from\_joinpoint = True参数可以允许将多节点连接分解为对join()的多个调用，以便可以进一步过滤每个路径：

q = session.query(Node).\

join("children", aliased=**True**).\

filter(Node.name='child 1').\

join("children", aliased=**True**, from\_joinpoint=**True**).\

filter(Node.name == 'grandchild 1')

The filtering aliases above can then be reset back to the original Node entity using [reset\_joinpoint()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.reset_joinpoint" \o "sqlalchemy.orm.query.Query.reset_joinpoint):

然后可以使用reset\_joinpoint()将以上过滤别名重置回原始节点实体：

q = session.query(Node).\

join("children", "children", aliased=**True**).\

filter(Node.name == 'grandchild 1').\

reset\_joinpoint().\

filter(Node.name == 'parent 1)

For an example of aliased=True, see the distribution example [XML Persistence](http://docs.sqlalchemy.org/en/rel_1_1/orm/examples.html" \l "examples-xmlpersistence) which illustrates an XPath-like query system using algorithmic joins.

有关aliased = True的示例，请参阅分布示例XML Persistence，其中说明了使用算法连接的类似XPath的查询系统。

|  |  |
| --- | --- |
| **Parameters:** | * ****\*props**** – A collection of one or more join conditions, each consisting of a relationship-bound attribute or string relationship name representing an "on clause", or a single target entity, or a tuple in the form of (target, onclause). A special two-argument calling form of the form target,onclause is also accepted.一个或多个连接条件的集合，每个连接条件由表示“on子句”的关系绑定属性或字符串关系名称或单个目标实体或以(target，onclause)形式组成的元组组成。 形式目标的特殊双参数调用形式也被接受。 * ****aliased=False**** – If True, indicate that the JOIN target should be anonymously aliased. Subsequent calls to [filter()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.filter" \o "sqlalchemy.orm.query.Query.filter) and similar will adapt the incoming criterion to the target alias, until [reset\_joinpoint()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.reset_joinpoint" \o "sqlalchemy.orm.query.Query.reset_joinpoint) is called.如果为True，则表示JOIN目标应以匿名方式进行别名。 对filter()和类似的后续调用将会调用进入的标准到目标别名，直到调用reset\_joinpoint()为止。 * ****isouter=False –****If True, the join used will be a left outer join, just as if the [Query.outerjoin()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.outerjoin" \o "sqlalchemy.orm.query.Query.outerjoin) method were called. This flag is here to maintain consistency with the same flag as accepted by [FromClause.join()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.join" \o "sqlalchemy.sql.expression.FromClause.join) and other Core constructs.如果为True，则使用的连接将是左外连接，就像调用了Query.outerjoin()方法一样。 这个标志在这里保持与FromClause.join()和其他Core结构所接受的相同标志的一致性。   *New in version 1.0.0.*   * ****full=False –****render FULL OUTER JOIN; implies isouter.渲染完整的FULL OUTER JOIN;意味着isouter。   *New in version 1.1.*   * ****from\_joinpoint=False**** – When using aliased=True, a setting of True here will cause the join to be from the most recent joined target, rather than starting back from the original FROM clauses of the query.当使用aliased = True时，此处设置为True将导致连接来自最近加入的目标，而不是从查询的原始FROM子句开始。 |

**See also**

[Querying with Joins](http://docs.sqlalchemy.org/en/rel_1_1/orm/tutorial.html" \l "ormtutorial-joins) in the ORM tutorial.

[Mapping Class Inheritance Hierarchies](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance.html) for details on how [join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join) is used for inheritance relationships.映射类继承层次结构有关join()如何用于继承关系的详细信息。

[orm.join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.join" \o "sqlalchemy.orm.join) - a standalone ORM-level join function, used internally by [Query.join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join), which in previous SQLAlchemy versions was the primary ORM-level joining interface.一个独立的ORM级连接函数，由Query.join()内部使用，它在以前的SQLAlchemy版本中是主ORM级加入界面。

**label**(*name*)

Return the full SELECT statement represented by this [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query), converted to a scalar subquery with a label of the given name.返回此查询表示的完整SELECT语句，转换为具有给定名称标签的标量子查询。

Analogous to [sqlalchemy.sql.expression.SelectBase.label()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.SelectBase.label" \o "sqlalchemy.sql.expression.SelectBase.label).

*New in version 0.6.5.*

**limit**(*limit*)

Apply a LIMIT to the query and return the newly resulting Query.

对查询应用LIMIT并返回新产生的查询。

**merge\_result**(*iterator*, *load=True*)

Merge a result into this [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) object's Session.

将结果合并到此Query对象的会话中。

Given an iterator returned by a [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) of the same structure as this one, return an identical iterator of results, with all mapped instances merged into the session using [Session.merge()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.merge" \o "sqlalchemy.orm.session.Session.merge). This is an optimized method which will merge all mapped instances, preserving the structure of the result rows and unmapped columns with less method overhead than that of calling [Session.merge()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.merge" \o "sqlalchemy.orm.session.Session.merge) explicitly for each value.

给定由与此类型相同结构的Query返回的迭代器，返回相同的结果迭代器，所有映射实例使用Session.merge()合并到会话中。 这是一种优化的方法，它将合并所有映射实例，以比用于每个值显式调用Session.merge()的方法开销更少的方式保留结果行和未映射列的结构。

The structure of the results is determined based on the column list of this [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) - if these do not correspond, unchecked errors will occur.

结果的结构是根据此查询的列表列表确定的 - 如果这些列表不对应，则会出现未经检查的错误。

The 'load' argument is the same as that of [Session.merge()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.merge" \o "sqlalchemy.orm.session.Session.merge).

'load'参数与Session.merge()的参数相同。

For an example of how [merge\_result()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.merge_result" \o "sqlalchemy.orm.query.Query.merge_result) is used, see the source code for the example [Dogpile Caching](http://docs.sqlalchemy.org/en/rel_1_1/orm/examples.html" \l "examples-caching), where [merge\_result()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.merge_result" \o "sqlalchemy.orm.query.Query.merge_result) is used to efficiently restore state from a cache back into a target [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session).

有关如何使用merge\_result()的示例，请参阅Dogpile Caching示例的源代码，其中merge\_result()用于将状态从缓存有效地恢复到目标会话。

**offset**(*offset*)

Apply an OFFSET to the query and return the newly resulting Query.

**one**()

Return exactly one result or raise an exception.

只返回一个结果或引发异常。

Raises sqlalchemy.orm.exc.NoResultFound if the query selects no rows. Raises sqlalchemy.orm.exc.MultipleResultsFound if multiple object identities are returned, or if multiple rows are returned for a query that returns only scalar values as opposed to full identity-mapped entities.

如果查询不选择任何行，则引发sqlalchemy.orm.exc.NoResultFound。 如果返回多个对象标识，或者返回仅返回标量值而不是全标识映射实体的查询的多行，则引发sqlalchemy.orm.exc.MultipleResultsFound。

Calling [one()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/baked.html" \l "sqlalchemy.ext.baked.Result.one" \o "sqlalchemy.ext.baked.Result.one) results in an execution of the underlying query.

调用one()将导致底层查询的执行。

**See also**

[Query.first()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.first" \o "sqlalchemy.orm.query.Query.first)

[Query.one\_or\_none()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.one_or_none" \o "sqlalchemy.orm.query.Query.one_or_none)

**one\_or\_none**()

Return at most one result or raise an exception.

最多返回一个结果或引发异常。

Returns None if the query selects no rows. Raises sqlalchemy.orm.exc.MultipleResultsFound if multiple object identities are returned, or if multiple rows are returned for a query that returns only scalar values as opposed to full identity-mapped entities.

如果查询不选择行，则返回None。 如果返回多个对象标识，或者返回仅返回标量值而不是全标识映射实体的查询的多行，则引发sqlalchemy.orm.exc.MultipleResultsFound。

Calling [Query.one\_or\_none()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.one_or_none" \o "sqlalchemy.orm.query.Query.one_or_none) results in an execution of the underlying query.

调用Query.one\_or\_none()导致基础查询的执行。

*New in version 1.0.9:*Added [Query.one\_or\_none()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.one_or_none" \o "sqlalchemy.orm.query.Query.one_or_none)

**See also**

[Query.first()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.first" \o "sqlalchemy.orm.query.Query.first)

[Query.one()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.one" \o "sqlalchemy.orm.query.Query.one)

**options**(*\*args*)

Return a new Query object, applying the given list of mapper options.

返回一个新的Query对象，应用给定的mapper选项列表。

Most supplied options regard changing how column- and relationship-mapped attributes are loaded. See the sections [Deferred Column Loading](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "deferred) and [Relationship Loading Techniques](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html) for reference documentation.

大多数提供的选项都意味着改变列和关系映射属性的加载方式。 有关参考文档，请参阅“延迟列加载和关系加载技术”部分。

**order\_by**(*\*criterion*)

apply one or more ORDER BY criterion to the query and return the newly resulting Query

对查询应用一个或多个ORDER BY标准，并返回新产生的Query

All existing ORDER BY settings can be suppressed by passing None - this will suppress any ORDER BY configured on mappers as well.

通过传递None可以抑制所有现有的ORDER BY设置 - 这将抑制在映射器上配置的任何ORDER BY。

Alternatively, passing False will reset ORDER BY and additionally re-allow default mapper.order\_by to take place. Note mapper.order\_by is deprecated.

或者，传递False将重置ORDER BY，并另外重新允许默认的mapper.order\_by发生。 注意mapper.order\_by已被弃用。

**outerjoin**(*\*props*, *\*\*kwargs*)

Create a left outer join against this Query object's criterion and apply generatively, returning the newly resulting Query.

根据此Query对象的标准创建一个左外连接并生成应用，返回新生成的Query。

Usage is the same as the join() method.

用法与join()方法相同。

**params**(*\*args*, *\*\*kwargs*)

add values for bind parameters which may have been specified in filter().

parameters may be specified using \*\*kwargs, or optionally a single dictionary as the first positional argument. The reason for both is that \*\*kwargs is convenient, however some parameter dictionaries contain unicode keys in which case \*\*kwargs cannot be used.

添加可能已在filter()中指定的绑定参数的值。

可以使用\*\* kwargs或可选地单个字典作为第一位置参数来指定参数。 两者的原因是\*\* kwargs很方便，但是一些参数字典包含unicode键，在这种情况下\*\* kwargs不能使用。

**populate\_existing**()

Return a [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) that will expire and refresh all instances as they are loaded, or reused from the current [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session).

返回一个查询，它会在所有实例加载完毕并刷新所有实例，或从当前会话重用。

[populate\_existing()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.populate_existing" \o "sqlalchemy.orm.query.Query.populate_existing) does not improve behavior when the ORM is used normally - the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) object's usual behavior of maintaining a transaction and expiring all attributes after rollback or commit handles object state automatically. This method is not intended for general use.

当正常使用ORM时，populate\_existing()不会改善行为 - 在对象回滚或提交之后，Session对象通常维护一个事务并且终止所有属性的行为会自动处理对象状态。 此方法不适用于一般用途。

**prefix\_with**(*\*prefixes*)

Apply the prefixes to the query and return the newly resulting Query.

将前缀应用于查询并返回新生成的查询。

|  |  |
| --- | --- |
| **Parameters:** | ****\*prefixes**** – optional prefixes, typically strings, not using any commas. In particular is useful for MySQL keywords.可选前缀，通常是字符串，不使用任何逗号。 特别对MySQL关键字很有用。 |

e.g.:

query = sess.query(User.name).\

prefix\_with('HIGH\_PRIORITY').\

prefix\_with('SQL\_SMALL\_RESULT', 'ALL')

Would render:

SELECT HIGH\_PRIORITY SQL\_SMALL\_RESULT ALL users.name AS users\_nameFROM users

*New in version 0.7.7.*

**See also**

[HasPrefixes.prefix\_with()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.HasPrefixes.prefix_with" \o "sqlalchemy.sql.expression.HasPrefixes.prefix_with)

**reset\_joinpoint**()

Return a new [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query), where the "join point" has been reset back to the base FROM entities of the query.

返回一个新的查询，其中“连接点”已被重置回查询的基本FROM实体。

This method is usually used in conjunction with the aliased=True feature of the [join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join) method. See the example in [join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join) for how this is used.

该方法通常与join()方法的aliased = True特征配合使用。 请参阅join()中的示例，了解如何使用。

**scalar**()

Return the first element of the first result or None if no rows present. If multiple rows are returned, raises MultipleResultsFound.

**>>>** session.query(Item).scalar()

<Item>

**>>>** session.query(Item.id).scalar()

1

**>>>** session.query(Item.id).filter(Item.id < 0).scalar()

None

**>>>** session.query(Item.id, Item.name).scalar()1

**>>>** session.query(func.count(Parent.id)).scalar()

20

This results in an execution of the underlying query.

**select\_entity\_from**(*from\_obj*)

Set the FROM clause of this [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) to a core selectable, applying it as a replacement FROM clause for corresponding mapped entities.

The [Query.select\_entity\_from()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.select_entity_from" \o "sqlalchemy.orm.query.Query.select_entity_from) method supplies an alternative approach to the use case of applying an [aliased()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.aliased" \o "sqlalchemy.orm.aliased) construct explicitly throughout a query. Instead of referring to the [aliased()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.aliased" \o "sqlalchemy.orm.aliased) construct explicitly, [Query.select\_entity\_from()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.select_entity_from" \o "sqlalchemy.orm.query.Query.select_entity_from) automatically *adapts* all occurences of the entity to the target selectable.

Given a case for [aliased()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.aliased" \o "sqlalchemy.orm.aliased) such as selecting User objects from a SELECT statement:

select\_stmt = select([User]).where(User.id == 7)user\_alias = aliased(User, select\_stmt)

q = session.query(user\_alias).\

filter(user\_alias.name == 'ed')

Above, we apply the user\_alias object explicitly throughout the query. When it's not feasible for user\_alias to be referenced explicitly in many places, [Query.select\_entity\_from()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.select_entity_from" \o "sqlalchemy.orm.query.Query.select_entity_from) may be used at the start of the query to adapt the existing User entity:

q = session.query(User).\

select\_entity\_from(select\_stmt).\

filter(User.name == 'ed')

Above, the generated SQL will show that the User entity is adapted to our statement, even in the case of the WHERE clause:

**SELECT** anon\_1.id **AS** anon\_1\_id, anon\_1.name **AS** anon\_1\_name**FROM** (**SELECT** "user".id **AS** id, "user".name **AS** name**FROM** "user"**WHERE** "user".id = :id\_1) **AS** anon\_1**WHERE** anon\_1.name = :name\_1

The [Query.select\_entity\_from()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.select_entity_from" \o "sqlalchemy.orm.query.Query.select_entity_from) method is similar to the [Query.select\_from()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.select_from" \o "sqlalchemy.orm.query.Query.select_from) method, in that it sets the FROM clause of the query. The difference is that it additionally applies adaptation to the other parts of the query that refer to the primary entity. If above we had used [Query.select\_from()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.select_from" \o "sqlalchemy.orm.query.Query.select_from) instead, the SQL generated would have been:

*-- uses plain select\_from(), not select\_entity\_from()***SELECT** "user".id **AS** user\_id, "user".name **AS** user\_name**FROM** "user", (**SELECT** "user".id **AS** id, "user".name **AS** name**FROM** "user"**WHERE** "user".id = :id\_1) **AS** anon\_1**WHERE** "user".name = :name\_1

To supply textual SQL to the [Query.select\_entity\_from()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.select_entity_from" \o "sqlalchemy.orm.query.Query.select_entity_from) method, we can make use of the [text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text) construct. However, the [text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text) construct needs to be aligned with the columns of our entity, which is achieved by making use of the [TextClause.columns()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.TextClause.columns" \o "sqlalchemy.sql.expression.TextClause.columns) method:

text\_stmt = text("select id, name from user").columns(

User.id, User.name)q = session.query(User).select\_entity\_from(text\_stmt)

[Query.select\_entity\_from()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.select_entity_from" \o "sqlalchemy.orm.query.Query.select_entity_from) itself accepts an [aliased()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.aliased" \o "sqlalchemy.orm.aliased) object, so that the special options of [aliased()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.aliased" \o "sqlalchemy.orm.aliased) such as[aliased.adapt\_on\_names](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.aliased.params.adapt_on_names" \o "sqlalchemy.orm.aliased) may be used within the scope of the [Query.select\_entity\_from()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.select_entity_from" \o "sqlalchemy.orm.query.Query.select_entity_from) method's adaptation services. Suppose a view user\_view also returns rows from user. If we reflect this view into a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table), this view has no relationship to the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) to which we are mapped, however we can use name matching to select from it:

user\_view = Table('user\_view', metadata,

autoload\_with=engine)user\_view\_alias = aliased(

User, user\_view, adapt\_on\_names=**True**)q = session.query(User).\

select\_entity\_from(user\_view\_alias).\

order\_by(User.name)

*Changed in version 1.1.7:*The [Query.select\_entity\_from()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.select_entity_from" \o "sqlalchemy.orm.query.Query.select_entity_from) method now accepts an [aliased()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.aliased" \o "sqlalchemy.orm.aliased) object as an alternative to a [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) object.

|  |  |
| --- | --- |
| **Parameters:** | ****from\_obj**** – a [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) object that will replace the FROM clause of this [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query). It also may be an instance of [aliased()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.aliased" \o "sqlalchemy.orm.aliased). |

**See also**

[Query.select\_from()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.select_from" \o "sqlalchemy.orm.query.Query.select_from)

**select\_from**(*\*from\_obj*)

Set the FROM clause of this [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) explicitly.

[Query.select\_from()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.select_from" \o "sqlalchemy.orm.query.Query.select_from) is often used in conjunction with [Query.join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join) in order to control which entity is selected from on the "left" side of the join.

The entity or selectable object here effectively replaces the "left edge" of any calls to [join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join), when no joinpoint is otherwise established - usually, the default "join point" is the leftmost entity in the [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) object's list of entities to be selected.

A typical example:

q = session.query(Address).select\_from(User).\

join(User.addresses).\

filter(User.name == 'ed')

Which produces SQL equivalent to:

SELECT address.\* FROM userJOIN address ON user.id=address.user\_idWHERE user.name = :name\_1

|  |  |
| --- | --- |
| **Parameters:** | ****\*from\_obj**** – collection of one or more entities to apply to the FROM clause. Entities can be mapped classes, [AliasedClass](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.util.AliasedClass" \o "sqlalchemy.orm.util.AliasedClass) objects, [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) objects as well as core [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) elements like subqueries. |

*Changed in version 0.9:*This method no longer applies the given FROM object to be the selectable from which matching entities select from; the [select\_entity\_from()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.select_entity_from" \o "sqlalchemy.orm.query.Query.select_entity_from) method now accomplishes this. See that method for a description of this behavior.

**See also**

[join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join)

[Query.select\_entity\_from()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.select_entity_from" \o "sqlalchemy.orm.query.Query.select_entity_from)

**selectable**

Return the [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select) object emitted by this [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query).

Used for [inspect()](http://docs.sqlalchemy.org/en/rel_1_1/core/inspection.html" \l "sqlalchemy.inspection.inspect" \o "sqlalchemy.inspection.inspect) compatibility, this is equivalent to:

query.enable\_eagerloads(**False**).with\_labels().statement

**slice**(*start*, *stop*)

Computes the "slice" of the [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) represented by the given indices and returns the resulting [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query).

The start and stop indices behave like the argument to Python's built-in range() function. This method provides an alternative to using LIMIT/OFFSETto get a slice of the query.

For example,

session.query(User).order\_by(User.id).slice(1, 3)

renders as

**SELECT** users.id **AS** users\_id,

users.name **AS** users\_name**FROM** users **ORDER** **BY** users.id**LIMIT** ? **OFFSET** ?(2, 1)

**See also**

[Query.limit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.limit" \o "sqlalchemy.orm.query.Query.limit)

[Query.offset()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.offset" \o "sqlalchemy.orm.query.Query.offset)

**statement**

The full SELECT statement represented by this Query.

The statement by default will not have disambiguating labels applied to the construct unless with\_labels(True) is called first.

**subquery**(*name=None*, *with\_labels=False*, *reduce\_columns=False*)

return the full SELECT statement represented by this [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query), embedded within an [Alias](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Alias" \o "sqlalchemy.sql.expression.Alias).

Eager JOIN generation within the query is disabled.

|  |  |
| --- | --- |
| **Parameters:** | * ****name**** – string name to be assigned as the alias; this is passed through to [FromClause.alias()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.alias" \o "sqlalchemy.sql.expression.FromClause.alias). If None, a name will be deterministically generated at compile time. * ****with\_labels**** – if True, [with\_labels()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.with_labels" \o "sqlalchemy.orm.query.Query.with_labels) will be called on the [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) first to apply table-qualified labels to all columns. * ****reduce\_columns –****if True, [Select.reduce\_columns()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.reduce_columns" \o "sqlalchemy.sql.expression.Select.reduce_columns) will be called on the resulting [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) construct, to remove same-named columns where one also refers to the other via foreign key or WHERE clause equivalence.   *Changed in version 0.8:*the with\_labels and reduce\_columns keyword arguments were added. |

**suffix\_with**(*\*suffixes*)

Apply the suffix to the query and return the newly resulting Query.

|  |  |
| --- | --- |
| **Parameters:** | ****\*suffixes**** – optional suffixes, typically strings, not using any commas. |

*New in version 1.0.0.*

**See also**

[Query.prefix\_with()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.prefix_with" \o "sqlalchemy.orm.query.Query.prefix_with)

[HasSuffixes.suffix\_with()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.HasSuffixes.suffix_with" \o "sqlalchemy.sql.expression.HasSuffixes.suffix_with)

**union**(*\*q*)

Produce a UNION of this Query against one or more queries.

e.g.:

q1 = sess.query(SomeClass).filter(SomeClass.foo=='bar')

q2 = sess.query(SomeClass).filter(SomeClass.bar=='foo')

q3 = q1.union(q2)

The method accepts multiple Query objects so as to control the level of nesting. A series of union() calls such as:

x.union(y).union(z).all()

will nest on each union(), and produces:

SELECT \* FROM (SELECT \* FROM (SELECT \* FROM X UNION

SELECT \* FROM y) UNION SELECT \* FROM Z)

Whereas:

x.union(y, z).all()

produces:

SELECT \* FROM (SELECT \* FROM X UNION SELECT \* FROM y UNION

SELECT \* FROM Z)

Note that many database backends do not allow ORDER BY to be rendered on a query called within UNION, EXCEPT, etc. To disable all ORDER BY clauses including those configured on mappers, issue query.order\_by(None) - the resulting [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) object will not render ORDER BY within its SELECT statement.

**union\_all**(*\*q*)

Produce a UNION ALL of this Query against one or more queries.

Works the same way as [union()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.union" \o "sqlalchemy.orm.query.Query.union). See that method for usage examples.

**update**(*values*, *synchronize\_session='evaluate'*, *update\_args=None*)

Perform a bulk update query.

Updates rows matched by this query in the database.

E.g.:

sess.query(User).filter(User.age == 25).\

update({User.age: User.age - 10}, synchronize\_session=**False**)

sess.query(User).filter(User.age == 25).\

update({"age": User.age - 10}, synchronize\_session='evaluate')

**Warning**

The [Query.update()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.update" \o "sqlalchemy.orm.query.Query.update) method is a "bulk" operation, which bypasses ORM unit-of-work automation in favor of greater performance. ****Please read all caveats and warnings below.****

|  |  |
| --- | --- |
| **Parameters:** | * ****values**** –   a dictionary with attributes names, or alternatively mapped attributes or SQL expressions, as keys, and literal values or sql expressions as values. If [parameter-ordered mode](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html" \l "updates-order-parameters) is desired, the values can be passed as a list of 2-tuples; this requires that the [preserve\_parameter\_order](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.update.params.preserve_parameter_order" \o "sqlalchemy.sql.expression.update) flag is passed to the [Query.update.update\_args](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.update.params.update_args" \o "sqlalchemy.orm.query.Query.update) dictionary as well.  *Changed in version 1.0.0:*- string names in the values dictionary are now resolved against the mapped entity; previously, these strings were passed as literal column names with no mapper-level translation.   * ****synchronize\_session**** –   chooses the strategy to update the attributes on objects in the session. Valid values are:  False - don't synchronize the session. This option is the most efficient and is reliable once the session is expired, which typically occurs after a commit(), or explicitly using expire\_all(). Before the expiration, updated objects may still remain in the session with stale values on their attributes, which can lead to confusing results.  'fetch' - performs a select query before the update to find objects that are matched by the update query. The updated attributes are expired on matched objects.  'evaluate' - Evaluate the Query's criteria in Python straight on the objects in the session. If evaluation of the criteria isn't implemented, an exception is raised.  The expression evaluator currently doesn't account for differing string collations between the database and Python.   * ****update\_args**** –   Optional dictionary, if present will be passed to the underlying [update()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.update" \o "sqlalchemy.sql.expression.update) construct as the \*\*kw for the object. May be used to pass dialect-specific arguments such as mysql\_limit, as well as other special arguments such as [preserve\_parameter\_order](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.update.params.preserve_parameter_order" \o "sqlalchemy.sql.expression.update).  *New in version 1.0.0.* |
| **Returns:** | the count of rows matched as returned by the database's "row count" feature. |

**Warning**

****Additional Caveats for bulk query updates****

The method does ****not**** offer in-Python cascading of relationships - it is assumed that ON UPDATE CASCADE is configured for any foreign key references which require it, otherwise the database may emit an integrity violation if foreign key references are being enforced.

该方法不提供Python中的关联级联 - 假定ON UPDATE CASCADE被配置为需要它的任何外键引用，否则如果正在执行外键引用，数据库可能会发出完整性违例。

After the UPDATE, dependent objects in the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) which were impacted by an ON UPDATE CASCADE may not contain the current state; this issue is resolved once the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) is expired, which normally occurs upon [Session.commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.commit" \o "sqlalchemy.orm.session.Session.commit) or can be forced by using [Session.expire\_all()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.expire_all" \o "sqlalchemy.orm.session.Session.expire_all).

在UPDATE之后，由ON UPDATE CASCADE影响的会话中的依赖对象可能不包含当前状态;一旦会话过期，这个问题就会被解决，这通常发生在Session.commit()上，或者可以通过使用Session.expire\_all()强制执行。

The 'fetch' strategy results in an additional SELECT statement emitted and will significantly reduce performance.

“fetch”策略会导致发出额外的SELECT语句，并将显着降低性能。

The 'evaluate' strategy performs a scan of all matching objects within the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session); if the contents of the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) are expired, such as via a proceeding [Session.commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.commit" \o "sqlalchemy.orm.session.Session.commit) call, ****this will result in SELECT queries emitted for every matching object****.

“评估”策略对会话内的所有匹配对象执行扫描;如果会话的内容过期，例如通过进行的Session.commit()调用，这将导致为每个匹配对象发出的SELECT查询。

The method supports multiple table updates, as detailed in [Multiple Table Updates](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html" \l "multi-table-updates), and this behavior does extend to support updates of joined-inheritance and other multiple table mappings. However, the ****join condition of an inheritance mapper is not automatically rendered****. Care must be taken in any multiple-table update to explicitly include the joining condition between those tables, even in mappings where this is normally automatic. E.g. if a class Engineer subclasses Employee, an UPDATE of the Engineer local table using criteria against the Employee local table might look like:

该方法支持多表更新，如“多表更新”中所详述的，并且此行为扩展为支持加入继承和其他多表映射的更新。但是，继承映射器的连接条件不会自动呈现。在任何多表更新中必须注意明确地包括这些表之间的连接条件，即使是通常是自动的映射。例如。如果一个类工程师子类Employee，那么使用雇员本地表的标准的工程师本地表的UPDATE可能如下所示：

session.query(Engineer).\

filter(Engineer.id == Employee.id).\

filter(Employee.name == 'dilbert').\

update({"engineer\_type": "programmer"})

The polymorphic identity WHERE criteria is ****not**** included for single- or joined- table updates - this must be added ****manually****, even for single table inheritance.

对于单表或更新表，不包括多态标识WHERE标准 - 即使对于单表继承，也必须手动添加。

The [MapperEvents.before\_update()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents.before_update" \o "sqlalchemy.orm.events.MapperEvents.before_update) and [MapperEvents.after\_update()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents.after_update" \o "sqlalchemy.orm.events.MapperEvents.after_update) events ****are not invoked from this method****. Instead, the[SessionEvents.after\_bulk\_update()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_bulk_update" \o "sqlalchemy.orm.events.SessionEvents.after_bulk_update) method is provided to act upon a mass UPDATE of entity rows.

MapperEvents.before\_update()和MapperEvents.after\_update()事件不会从此方法调用。 相反，提供了SessionEvents.after\_bulk\_update()方法来对实体行的大规模UPDATE进行操作。

**See also**

[Query.delete()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.delete" \o "sqlalchemy.orm.query.Query.delete)

[Inserts, Updates and Deletes](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html" \l "inserts-and-updates) - Core SQL tutorial

**value**(*column*)

Return a scalar result corresponding to the given column expression.

返回与给定列表达式相对应的标量结果。

**values**(*\*columns*)

Return an iterator yielding result tuples corresponding to the given list of columns

返回一个迭代器，产生与给定列列相对应的结果元组

**whereclause**

A readonly attribute which returns the current WHERE criterion for this Query.

一个只读属性，返回此查询的当前WHERE标准。

This returned value is a SQL expression construct, or None if no criterion has been established.

这个返回的值是一个SQL表达式构造，如果没有建立标准，则返回None。

**with\_entities**(*\*entities*)

Return a new [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) replacing the SELECT list with the given entities.

e.g.:

*# Users, filtered on some arbitrary criterion# and then ordered by related email address*

q = session.query(User). join(User.address). filter(User.name.like('*%e*d%')). order\_by(Address.email)

*# given \*only\* User.id==5, Address.email, and 'q', what# would the \*next\* User in the result be ?*

subq = q.with\_entities(Address.email). order\_by(**None**). filter(User.id==5). subquery()

q = q.join((subq, subq.c.email < Address.email)). limit(1)

*New in version 0.6.5.*

**with\_for\_update**(*read=False*, *nowait=False*, *of=None*, *skip\_locked=False*, *key\_share=False*)

return a new [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) with the specified options for the FOR UPDATE clause.

使用FOR UPDATE子句的指定选项返回一个新的Query。

The behavior of this method is identical to that of SelectBase.with\_for\_update(). When called with no arguments, the resulting SELECTstatement will have a FOR UPDATE clause appended. When additional arguments are specified, backend-specific options such as FOR UPDATENOWAIT or LOCK IN SHARE MODE can take effect.

此方法的行为与SelectBase.with\_for\_update()的行为相同。 当没有参数调用时，生成的SELECTstatement将附加一个FOR UPDATE子句。 当指定其他参数时，后端特定的选项(如FOR UPDATENOWAIT或LOCK IN SHARE MODE)可以生效。

E.g.:

q = sess.query(User).with\_for\_update(nowait=**True**, of=User)

The above query on a PostgreSQL backend will render like:

对PostgreSQL后端的上述查询将呈现如下：

SELECT users.id AS users\_id

FROM users FOR UPDATE OF users NOWAIT

*New in version 0.9.0:*[Query.with\_for\_update()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.with_for_update" \o "sqlalchemy.orm.query.Query.with_for_update) supersedes the [Query.with\_lockmode()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.with_lockmode" \o "sqlalchemy.orm.query.Query.with_lockmode) method.

**See also**

[GenerativeSelect.with\_for\_update()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect.with_for_update" \o "sqlalchemy.sql.expression.GenerativeSelect.with_for_update) - Core level method with full argument and behavioral description.

**with\_hint**(*selectable*, *text*, *dialect\_name='\*'*)

Add an indexing or other executional context hint for the given entity or selectable to this [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query).

为给定实体添加索引或其他执行上下文提示或可选择此查询。

Functionality is passed straight through to [with\_hint()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.with_hint" \o "sqlalchemy.sql.expression.Select.with_hint), with the addition that selectable can be a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table), [Alias](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Alias" \o "sqlalchemy.sql.expression.Alias), or ORM entity / mapped class /etc.

功能直接传递给with\_hint()，另外可选择的可以是Table，Alias或ORM实体/映射类/ etc。

**See also**

[Query.with\_statement\_hint()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.with_statement_hint" \o "sqlalchemy.orm.query.Query.with_statement_hint)

**with\_labels**()

Apply column labels to the return value of Query.statement.

将列标签应用于Query.statement的返回值。

Indicates that this Query's statement accessor should return a SELECT statement that applies labels to all columns in the form <tablename>\_<columnname>; this is commonly used to disambiguate columns from multiple tables which have the same name.

表示此Query的语句访问器应返回一个SELECT语句，该语句将标签应用于<tablename> \_ <columnname>的所有列; 这通常用于消除具有相同名称的多个表中的列的歧义。

When the Query actually issues SQL to load rows, it always uses column labeling.

当Query实际上发出SQL来加载行时，它总是使用列标签。

**Note**

The [Query.with\_labels()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.with_labels" \o "sqlalchemy.orm.query.Query.with_labels) method *only* applies the output of [Query.statement](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.statement" \o "sqlalchemy.orm.query.Query.statement), and *not* to any of the result-row invoking systems of [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) itself, e.g.[Query.first()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.first" \o "sqlalchemy.orm.query.Query.first), [Query.all()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.all" \o "sqlalchemy.orm.query.Query.all), etc. To execute a query using [Query.with\_labels()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.with_labels" \o "sqlalchemy.orm.query.Query.with_labels), invoke the [Query.statement](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.statement" \o "sqlalchemy.orm.query.Query.statement) using [Session.execute()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.execute" \o "sqlalchemy.orm.session.Session.execute):

Query.with\_labels()方法仅应用Query.statement的输出，而不适用于Query本身的任何结果行调用系统，例如Query.all()，Query.all()等)。执行查询 使用Query.with\_labels()，使用Session.execute()调用Query.statement：

result = session.execute(query.with\_labels().statement)

**~~with\_lockmode~~**~~(~~*~~mode~~*~~)~~

~~Return a new [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) object with the specified "locking mode", which essentially refers to the FOR UPDATE clause.~~

*~~Deprecated since version 0.9.0:~~*~~superseded by [Query.with\_for\_update()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.with_for_update" \o "sqlalchemy.orm.query.Query.with_for_update).~~

|  |  |
| --- | --- |
| **~~Parameters:~~** | ****~~mode~~****~~–a string representing the desired locking mode. Valid values are:~~   * ~~None - translates to no lockmode~~ * ~~'update' - translates to FOR UPDATE (standard SQL, supported by most dialects)~~ * ~~'update\_nowait' - translates to FOR UPDATE NOWAIT (supported by Oracle, PostgreSQL 8.1 upwards)~~ * ~~'read' - translates to LOCK IN SHARE MODE (for MySQL), and FOR SHARE (for PostgreSQL)~~ |

**See also**

[Query.with\_for\_update()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.with_for_update" \o "sqlalchemy.orm.query.Query.with_for_update) - improved API for specifying the FOR UPDATE clause.

**with\_parent**(*instance*, *property=None*)

Add filtering criterion that relates the given instance to a child object or collection, using its attribute state as well as an established [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)configuration.

添加将给定实例与子对象或集合相关联的过滤条件，使用其属性状态以及已建立的relationship()配置。

The method uses the [with\_parent()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.with_parent" \o "sqlalchemy.orm.with_parent) function to generate the clause, the result of which is passed to [Query.filter()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.filter" \o "sqlalchemy.orm.query.Query.filter).

该方法使用with\_parent()函数生成子句，其结果传递给Query.filter()。

Parameters are the same as [with\_parent()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.with_parent" \o "sqlalchemy.orm.with_parent), with the exception that the given property can be None, in which case a search is performed against this [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) object's target mapper.

参数与with\_parent()相同，但给定属性可以为None，在这种情况下，将针对此Query对象的目标映射器执行搜索。

**with\_polymorphic**(*cls\_or\_mappers*, *selectable=None*, *polymorphic\_on=None*)

Load columns for inheriting classes.

加载用于继承类的列。

[Query.with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.with_polymorphic" \o "sqlalchemy.orm.query.Query.with_polymorphic) applies transformations to the "main" mapped class represented by this [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query). The "main" mapped class here means the [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) object's first argument is a full class, i.e. session.query(SomeClass). These transformations allow additional tables to be present in the FROM clause so that columns for a joined-inheritance subclass are available in the query, both for the purposes of load-time efficiency as well as the ability to use these columns at query time.

Query.with\_polymorphic()将转换应用于此查询表示的“主”映射类。 这里的“main”映射类表示Query对象的第一个参数是一个完整的类，即session.query(SomeClass)。 这些转换允许在FROM子句中存在其他表，以便在查询中可以使用连接继承子类的列，既用于加载时效率，也可以在查询时使用这些列。

See the documentation section [Using with\_polymorphic](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "with-polymorphic) for details on how this method is used.

有关如何使用此方法的详细信息，请参阅文档部分使用with\_polymorphic。

*Changed in version 0.8:*A new and more flexible function [orm.with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic" \o "sqlalchemy.orm.with_polymorphic) supersedes [Query.with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.with_polymorphic" \o "sqlalchemy.orm.query.Query.with_polymorphic), as it can apply the equivalent functionality to any set of columns or classes in the [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query), not just the "zero mapper". See that function for a description of arguments.

在版本0.8中更改：一个新的和更灵活的函数orm.with\_polymorphic()取代了Query.with\_polymorphic()，因为它可以将等效的功能应用于Query中的任何列或类的集合，而不仅仅是“零映射器”。 参见该函数的参数说明。

**with\_session**(*session*)

Return a [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) that will use the given [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session).

返回将使用给定会话的查询。

While the [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) object is normally instantiated using the [Session.query()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.query" \o "sqlalchemy.orm.session.Session.query) method, it is legal to build the [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) directly without necessarily using a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session). Such a [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) object, or any [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) already associated with a different [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session), can produce a new [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) object associated with a target session using this method:

虽然Query对象通常使用Session.query()方法实例化，但直接构建Query并不一定要使用Session是合法的。 这样一个Query对象或任何已经与不同Session相关联的Query可以使用此方法生成与目标会话关联的新Query对象：

**from** **sqlalchemy.orm** **import** Query

query = Query([MyClass]).filter(MyClass.id == 5)

result = query.with\_session(my\_session).one()

**with\_statement\_hint**(*text*, *dialect\_name='\*'*)

add a statement hint to this [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select).

This method is similar to [Select.with\_hint()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.with_hint" \o "sqlalchemy.sql.expression.Select.with_hint) except that it does not require an individual table, and instead applies to the statement as a whole.

This feature calls down into [Select.with\_statement\_hint()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.with_statement_hint" \o "sqlalchemy.sql.expression.Select.with_statement_hint).

*New in version 1.0.0.*

**See also**

[Query.with\_hint()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.with_hint" \o "sqlalchemy.orm.query.Query.with_hint)

**with\_transformation**(*fn*)

Return a new [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) object transformed by the given function.

E.g.:

**def** filter\_something(criterion):

**def** transform(q):

**return** q.filter(criterion)

**return** transform

q = q.with\_transformation(filter\_something(x==5))

This allows ad-hoc recipes to be created for [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) objects. See the example at [Building Transformers](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/hybrid.html" \l "hybrid-transformers).

*New in version 0.7.4.*

**yield\_per**(*count*)

Yield only count rows at a time.

The purpose of this method is when fetching very large result sets (> 10K rows), to batch results in sub-collections and yield them out partially, so that the Python interpreter doesn't need to declare very large areas of memory which is both time consuming and leads to excessive memory use. The performance from fetching hundreds of thousands of rows can often double when a suitable yield-per setting (e.g. approximately 1000) is used, even with DBAPIs that buffer rows (which are most).

The [Query.yield\_per()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.yield_per" \o "sqlalchemy.orm.query.Query.yield_per) method ****is not compatible with most eager loading schemes, including subqueryload and joinedload with collections****. For this reason, it may be helpful to disable eager loads, either unconditionally with [Query.enable\_eagerloads()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.enable_eagerloads" \o "sqlalchemy.orm.query.Query.enable_eagerloads):

q = sess.query(Object).yield\_per(100).enable\_eagerloads(**False**)

Or more selectively using [lazyload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.lazyload" \o "sqlalchemy.orm.lazyload); such as with an asterisk to specify the default loader scheme:

q = sess.query(Object).yield\_per(100).\

options(lazyload('\*'), joinedload(Object.some\_related))

**Warning**

Use this method with caution; if the same instance is present in more than one batch of rows, end-user changes to attributes will be overwritten.

In particular, it's usually impossible to use this setting with eagerly loaded collections (i.e. any lazy='joined' or 'subquery') since those collections will be cleared for a new load when encountered in a subsequent result batch. In the case of 'subquery' loading, the full result for all rows is fetched which generally defeats the purpose of[yield\_per()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.yield_per" \o "sqlalchemy.orm.query.Query.yield_per).

Also note that while [yield\_per()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.yield_per" \o "sqlalchemy.orm.query.Query.yield_per) will set the stream\_results execution option to True, currently this is only understood by [psycopg2](http://docs.sqlalchemy.org/en/rel_1_1/dialects/postgresql.html" \l "module-sqlalchemy.dialects.postgresql.psycopg2" \o "sqlalchemy.dialects.postgresql.psycopg2), [mysqldb](http://docs.sqlalchemy.org/en/rel_1_1/dialects/mysql.html" \l "module-sqlalchemy.dialects.mysql.mysqldb" \o "sqlalchemy.dialects.mysql.mysqldb) and [pymysql](http://docs.sqlalchemy.org/en/rel_1_1/dialects/mysql.html" \l "module-sqlalchemy.dialects.mysql.pymysql" \o "sqlalchemy.dialects.mysql.pymysql)dialects which will stream results using server side cursors instead of pre-buffer all rows for this query. Other DBAPIs ****pre-buffer all rows**** before making them available. The memory use of raw database rows is much less than that of an ORM-mapped object, but should still be taken into consideration when benchmarking.

**See also**

[Query.enable\_eagerloads()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.enable_eagerloads" \o "sqlalchemy.orm.query.Query.enable_eagerloads)

4.5.2 ORM-Specific Query Constructs

sqlalchemy.orm.**aliased**(*element*, *alias=None*, *name=None*, *flat=False*, *adapt\_on\_names=False*)

Produce an alias of the given element, usually an [AliasedClass](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.util.AliasedClass" \o "sqlalchemy.orm.util.AliasedClass) instance.

E.g.:

my\_alias = aliased(MyClass)

session.query(MyClass, my\_alias).filter(MyClass.id > my\_alias.id)

The [aliased()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.aliased" \o "sqlalchemy.orm.aliased) function is used to create an ad-hoc mapping of a mapped class to a new selectable. By default, a selectable is generated from the normally mapped selectable (typically a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)) using the [FromClause.alias()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.alias" \o "sqlalchemy.sql.expression.FromClause.alias) method. However, [aliased()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.aliased" \o "sqlalchemy.orm.aliased) can also be used to link the class to a new [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) statement. Also, the [with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic" \o "sqlalchemy.orm.with_polymorphic) function is a variant of [aliased()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.aliased" \o "sqlalchemy.orm.aliased) that is intended to specify a so-called "polymorphic selectable", that corresponds to the union of several joined-inheritance subclasses at once.

For convenience, the [aliased()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.aliased" \o "sqlalchemy.orm.aliased) function also accepts plain [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) constructs, such as a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) or [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) construct. In those cases, the [FromClause.alias()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.alias" \o "sqlalchemy.sql.expression.FromClause.alias) method is called on the object and the new [Alias](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Alias" \o "sqlalchemy.sql.expression.Alias) object returned. The returned [Alias](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Alias" \o "sqlalchemy.sql.expression.Alias) is not ORM-mapped in this case.

|  |  |
| --- | --- |
| **Parameters:** | * ****element**** – element to be aliased. Is normally a mapped class, but for convenience can also be a [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) element. * ****alias**** – Optional selectable unit to map the element to. This should normally be a [Alias](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Alias" \o "sqlalchemy.sql.expression.Alias) object corresponding to the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) to which the class is mapped, or to a [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) construct that is compatible with the mapping. By default, a simple anonymous alias of the mapped table is generated. * ****name**** – optional string name to use for the alias, if not specified by the alias parameter. The name, among other things, forms the attribute name that will be accessible via tuples returned by a [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) object. * ****flat –****   Boolean, will be passed through to the [FromClause.alias()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.alias" \o "sqlalchemy.sql.expression.FromClause.alias) call so that aliases of [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) objects don't include an enclosing SELECT. This can lead to more efficient queries in many circumstances. A JOIN against a nested JOIN will be rewritten as a JOIN against an aliased SELECT subquery on backends that don't support this syntax.  *New in version 0.9.0.*  **See also**  [Join.alias()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join.alias" \o "sqlalchemy.sql.expression.Join.alias)   * ****adapt\_on\_names –****   if True, more liberal "matching" will be used when mapping the mapped columns of the ORM entity to those of the given selectable - a name-based match will be performed if the given selectable doesn't otherwise have a column that corresponds to one on the entity. The use case for this is when associating an entity with some derived selectable such as one that uses aggregate functions:  **class** **UnitPrice**(Base):  \_\_tablename\_\_ = 'unit\_price'  ...  unit\_id = Column(Integer)  price = Column(Numeric)  aggregated\_unit\_price = Session.query(  func.sum(UnitPrice.price).label('price')  ).group\_by(UnitPrice.unit\_id).subquery()  aggregated\_unit\_price = aliased(UnitPrice,  alias=aggregated\_unit\_price, adapt\_on\_names=**True**)  Above, functions on aggregated\_unit\_price which refer to .price will return the func.sum(UnitPrice.price).label('price') column, as it is matched on the name "price". Ordinarily, the "price" function wouldn't have any "column correspondence" to the actual UnitPrice.price column as it is not a proxy of the original.  *New in version 0.7.3.* |

*class*sqlalchemy.orm.util.**AliasedClass**(*cls*, *alias=None*, *name=None*, *flat=False*, *adapt\_on\_names=False*, *with\_polymorphic\_mappers=()*, *with\_polymorphic\_discriminator=None*, *base\_alias=None*, *use\_mapper\_path=False*)

Represents an "aliased" form of a mapped class for usage with Query.

The ORM equivalent of a [sqlalchemy.sql.expression.alias()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.alias" \o "sqlalchemy.sql.expression.alias) construct, this object mimics the mapped class using a \_\_getattr\_\_ scheme and maintains a reference to a real [Alias](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Alias" \o "sqlalchemy.sql.expression.Alias) object.

Usage is via the [orm.aliased()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.aliased" \o "sqlalchemy.orm.aliased) function, or alternatively via the [orm.with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic" \o "sqlalchemy.orm.with_polymorphic) function.

Usage example:

*# find all pairs of users with the same name*

user\_alias = aliased(User)session.query(User, user\_alias).\

join((user\_alias, User.id > user\_alias.id)).\

filter(User.name==user\_alias.name)

The resulting object is an instance of [AliasedClass](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.util.AliasedClass" \o "sqlalchemy.orm.util.AliasedClass). This object implements an attribute scheme which produces the same attribute and method interface as the original mapped class, allowing [AliasedClass](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.util.AliasedClass" \o "sqlalchemy.orm.util.AliasedClass) to be compatible with any attribute technique which works on the original class, including hybrid attributes (see [Hybrid Attributes](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/hybrid.html)).

The [AliasedClass](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.util.AliasedClass" \o "sqlalchemy.orm.util.AliasedClass) can be inspected for its underlying [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper), aliased selectable, and other information using [inspect()](http://docs.sqlalchemy.org/en/rel_1_1/core/inspection.html" \l "sqlalchemy.inspection.inspect" \o "sqlalchemy.inspection.inspect):

**from** **sqlalchemy** **import** inspect

my\_alias = aliased(MyClass)

insp = inspect(my\_alias)

The resulting inspection object is an instance of [AliasedInsp](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.util.AliasedInsp" \o "sqlalchemy.orm.util.AliasedInsp).

See [aliased()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.aliased" \o "sqlalchemy.orm.aliased) and [with\_polymorphic()](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance_loading.html" \l "sqlalchemy.orm.with_polymorphic" \o "sqlalchemy.orm.with_polymorphic) for construction argument descriptions.

*class*sqlalchemy.orm.util.**AliasedInsp**(*entity*, *mapper*, *selectable*, *name*, *with\_polymorphic\_mappers*, *polymorphic\_on*, *\_base\_alias*, *\_use\_mapper\_path*, *adapt\_on\_names*)

Bases: [sqlalchemy.orm.base.InspectionAttr](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.base.InspectionAttr" \o "sqlalchemy.orm.base.InspectionAttr)

Provide an inspection interface for an [AliasedClass](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.util.AliasedClass" \o "sqlalchemy.orm.util.AliasedClass) object.

The [AliasedInsp](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.util.AliasedInsp" \o "sqlalchemy.orm.util.AliasedInsp) object is returned given an [AliasedClass](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.util.AliasedClass" \o "sqlalchemy.orm.util.AliasedClass) using the [inspect()](http://docs.sqlalchemy.org/en/rel_1_1/core/inspection.html" \l "sqlalchemy.inspection.inspect" \o "sqlalchemy.inspection.inspect) function:

**from** **sqlalchemy** **import** inspect

**from** **sqlalchemy.orm** **import** aliased

my\_alias = aliased(MyMappedClass)

insp = inspect(my\_alias)

Attributes on [AliasedInsp](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.util.AliasedInsp" \o "sqlalchemy.orm.util.AliasedInsp) include:

* entity - the [AliasedClass](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.util.AliasedClass" \o "sqlalchemy.orm.util.AliasedClass) represented.
* mapper - the [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) mapping the underlying class.
* selectable - the [Alias](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Alias" \o "sqlalchemy.sql.expression.Alias) construct which ultimately represents an aliased [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) or [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select) construct.
* name - the name of the alias. Also is used as the attribute name when returned in a result tuple from [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query).
* with\_polymorphic\_mappers - collection of [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) objects indicating all those mappers expressed in the select construct for the [AliasedClass](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.util.AliasedClass" \o "sqlalchemy.orm.util.AliasedClass).
* polymorphic\_on - an alternate column or SQL expression which will be used as the "discriminator" for a polymorphic load.

**See also**

[Runtime Inspection API](http://docs.sqlalchemy.org/en/rel_1_1/core/inspection.html)

*class*sqlalchemy.orm.query.**Bundle**(*name*, *\*exprs*, *\*\*kw*)

Bases: [sqlalchemy.orm.base.InspectionAttr](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.base.InspectionAttr" \o "sqlalchemy.orm.base.InspectionAttr)

A grouping of SQL expressions that are returned by a [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) under one namespace.

The [Bundle](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Bundle" \o "sqlalchemy.orm.query.Bundle) essentially allows nesting of the tuple-based results returned by a column-oriented [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) object. It also is extensible via simple subclassing, where the primary capability to override is that of how the set of expressions should be returned, allowing post-processing as well as custom return types, without involving ORM identity-mapped classes.

*New in version 0.9.0.*

**See also**

[Column Bundles](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "bundles)

**\_\_init\_\_**(*name*, *\*exprs*, *\*\*kw*)

Construct a new [Bundle](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Bundle" \o "sqlalchemy.orm.query.Bundle).

e.g.:

bn = Bundle("mybundle", MyClass.x, MyClass.y)

**for** row **in** session.query(bn).filter(

bn.c.x == 5).filter(bn.c.y == 4):

print(row.mybundle.x, row.mybundle.y)

|  |  |
| --- | --- |
| **Parameters:** | * ****name**** – name of the bundle. * ****\*exprs**** – columns or SQL expressions comprising the bundle. * ****single\_entity=False**** – if True, rows for this [Bundle](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Bundle" \o "sqlalchemy.orm.query.Bundle) can be returned as a "single entity" outside of any enclosing tuple in the same manner as a mapped entity. |

**c***= None*

An alias for [Bundle.columns](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Bundle.columns" \o "sqlalchemy.orm.query.Bundle.columns).

**columns***= None*

A namespace of SQL expressions referred to by this [Bundle](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Bundle" \o "sqlalchemy.orm.query.Bundle).

e.g.:

bn = Bundle("mybundle", MyClass.x, MyClass.y)

q = sess.query(bn).filter(bn.c.x == 5)

Nesting of bundles is also supported:

b1 = Bundle("b1",

Bundle('b2', MyClass.a, MyClass.b),

Bundle('b3', MyClass.x, MyClass.y)

)

q = sess.query(b1).filter(

b1.c.b2.c.a == 5).filter(b1.c.b3.c.y == 9)

**See also**

[Bundle.c](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Bundle.c" \o "sqlalchemy.orm.query.Bundle.c)

**create\_row\_processor**(*query*, *procs*, *labels*)

Produce the "row processing" function for this [Bundle](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Bundle" \o "sqlalchemy.orm.query.Bundle).

May be overridden by subclasses.

**See also**

[Column Bundles](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "bundles) - includes an example of subclassing.

**label**(*name*)

Provide a copy of this [Bundle](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Bundle" \o "sqlalchemy.orm.query.Bundle) passing a new label.

**single\_entity***= False*

If True, queries for a single Bundle will be returned as a single entity, rather than an element within a keyed tuple.

*class*sqlalchemy.util.**KeyedTuple**

Bases: sqlalchemy.util.\_collections.AbstractKeyedTuple

tuple subclass that adds labeled names.

E.g.:

**>>>** k = KeyedTuple([1, 2, 3], labels=["one", "two", "three"])

**>>>** k.one1

**>>>** k.two2

Result rows returned by [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) that contain multiple ORM entities and/or column expressions make use of this class to return rows.

The [KeyedTuple](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.util.KeyedTuple" \o "sqlalchemy.util.KeyedTuple) exhibits similar behavior to the collections.namedtuple() construct provided in the Python standard library, however is architected very differently. Unlike collections.namedtuple(), [KeyedTuple](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.util.KeyedTuple" \o "sqlalchemy.util.KeyedTuple) is does not rely on creation of custom subtypes in order to represent a new series of keys, instead each [KeyedTuple](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.util.KeyedTuple" \o "sqlalchemy.util.KeyedTuple) instance receives its list of keys in place. The subtype approach of collections.namedtuple() introduces significant complexity and performance overhead, which is not necessary for the [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) object's use case.

*Changed in version 0.8:*Compatibility methods with collections.namedtuple() have been added including [KeyedTuple.\_fields](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.util.KeyedTuple._fields" \o "sqlalchemy.util.KeyedTuple._fields) and [KeyedTuple.\_asdict()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.util.KeyedTuple._asdict" \o "sqlalchemy.util.KeyedTuple._asdict).

**See also**

[Querying](http://docs.sqlalchemy.org/en/rel_1_1/orm/tutorial.html" \l "ormtutorial-querying)

**\_asdict**()

Return the contents of this [KeyedTuple](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.util.KeyedTuple" \o "sqlalchemy.util.KeyedTuple) as a dictionary.

This method provides compatibility with collections.namedtuple(), with the exception that the dictionary returned is ****not**** ordered.

*New in version 0.8.*

**\_fields**

Return a tuple of string key names for this [KeyedTuple](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.util.KeyedTuple" \o "sqlalchemy.util.KeyedTuple).

This method provides compatibility with collections.namedtuple().

*New in version 0.8.*

**See also**

[KeyedTuple.keys()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.util.KeyedTuple.keys" \o "sqlalchemy.util.KeyedTuple.keys)

**keys**()

*inherited from the* keys() *method of* AbstractKeyedTuple

Return a list of string key names for this [KeyedTuple](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.util.KeyedTuple" \o "sqlalchemy.util.KeyedTuple).

**See also**

[KeyedTuple.\_fields](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.util.KeyedTuple._fields" \o "sqlalchemy.util.KeyedTuple._fields)

*class*sqlalchemy.orm.strategy\_options.**Load**(*entity*)

Bases: sqlalchemy.sql.expression.Generative, sqlalchemy.orm.interfaces.MapperOption

Represents loader options which modify the state of a [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) in order to affect how various mapped attributes are loaded.

The [Load](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.Load" \o "sqlalchemy.orm.Load) object is in most cases used implicitly behind the scenes when one makes use of a query option like [joinedload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.joinedload" \o "sqlalchemy.orm.joinedload), [defer()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "sqlalchemy.orm.defer" \o "sqlalchemy.orm.defer), or similar. However, the [Load](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.Load" \o "sqlalchemy.orm.Load) object can also be used directly, and in some cases can be useful.

To use [Load](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.Load" \o "sqlalchemy.orm.Load) directly, instantiate it with the target mapped class as the argument. This style of usage is useful when dealing with a [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) that has multiple entities:

myopt = Load(MyClass).joinedload("widgets")

The above myopt can now be used with [Query.options()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.options" \o "sqlalchemy.orm.query.Query.options), where it will only take effect for the MyClass entity:

session.query(MyClass, MyOtherClass).options(myopt)

One case where [Load](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.Load" \o "sqlalchemy.orm.Load) is useful as public API is when specifying "wildcard" options that only take effect for a certain class:

session.query(Order).options(Load(Order).lazyload('\*'))

Above, all relationships on Order will be lazy-loaded, but other attributes on those descendant objects will load using their normal loader strategy.

**See also**

[Relationship Loading Techniques](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html)

**baked\_lazyload**(*loadopt*, *attr*)

Produce a new [Load](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.Load" \o "sqlalchemy.orm.Load) object with the orm.baked\_lazyload() option applied.

See orm.baked\_lazyload() for usage examples.

**contains\_eager**(*loadopt*, *attr*, *alias=None*)

Produce a new [Load](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.Load" \o "sqlalchemy.orm.Load) object with the [orm.contains\_eager()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.contains_eager" \o "sqlalchemy.orm.contains_eager) option applied.

See [orm.contains\_eager()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.contains_eager" \o "sqlalchemy.orm.contains_eager) for usage examples.

**defaultload**(*loadopt*, *attr*)

Produce a new [Load](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.Load" \o "sqlalchemy.orm.Load) object with the [orm.defaultload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.defaultload" \o "sqlalchemy.orm.defaultload) option applied.

See [orm.defaultload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.defaultload" \o "sqlalchemy.orm.defaultload) for usage examples.

**defer**(*loadopt*, *key*)

Produce a new [Load](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.Load" \o "sqlalchemy.orm.Load) object with the [orm.defer()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "sqlalchemy.orm.defer" \o "sqlalchemy.orm.defer) option applied.

See [orm.defer()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "sqlalchemy.orm.defer" \o "sqlalchemy.orm.defer) for usage examples.

**immediateload**(*loadopt*, *attr*)

Produce a new [Load](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.Load" \o "sqlalchemy.orm.Load) object with the [orm.immediateload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.immediateload" \o "sqlalchemy.orm.immediateload) option applied.

See [orm.immediateload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.immediateload" \o "sqlalchemy.orm.immediateload) for usage examples.

**joinedload**(*loadopt*, *attr*, *innerjoin=None*)

Produce a new [Load](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.Load" \o "sqlalchemy.orm.Load) object with the [orm.joinedload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.joinedload" \o "sqlalchemy.orm.joinedload) option applied.

See [orm.joinedload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.joinedload" \o "sqlalchemy.orm.joinedload) for usage examples.

**lazyload**(*loadopt*, *attr*)

Produce a new [Load](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.Load" \o "sqlalchemy.orm.Load) object with the [orm.lazyload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.lazyload" \o "sqlalchemy.orm.lazyload) option applied.

See [orm.lazyload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.lazyload" \o "sqlalchemy.orm.lazyload) for usage examples.

**load\_only**(*loadopt*, *\*attrs*)

Produce a new [Load](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.Load" \o "sqlalchemy.orm.Load) object with the [orm.load\_only()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "sqlalchemy.orm.load_only" \o "sqlalchemy.orm.load_only) option applied.

See [orm.load\_only()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "sqlalchemy.orm.load_only" \o "sqlalchemy.orm.load_only) for usage examples.

**noload**(*loadopt*, *attr*)

Produce a new [Load](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.Load" \o "sqlalchemy.orm.Load) object with the [orm.noload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.noload" \o "sqlalchemy.orm.noload) option applied.

See [orm.noload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.noload" \o "sqlalchemy.orm.noload) for usage examples.

**raiseload**(*loadopt*, *attr*, *sql\_only=False*)

Produce a new [Load](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.Load" \o "sqlalchemy.orm.Load) object with the [orm.raiseload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.raiseload" \o "sqlalchemy.orm.raiseload) option applied.

See [orm.raiseload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.raiseload" \o "sqlalchemy.orm.raiseload) for usage examples.

**subqueryload**(*loadopt*, *attr*)

Produce a new [Load](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.Load" \o "sqlalchemy.orm.Load) object with the [orm.subqueryload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.subqueryload" \o "sqlalchemy.orm.subqueryload) option applied.

See [orm.subqueryload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.subqueryload" \o "sqlalchemy.orm.subqueryload) for usage examples.

**undefer**(*loadopt*, *key*)

Produce a new [Load](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.Load" \o "sqlalchemy.orm.Load) object with the [orm.undefer()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "sqlalchemy.orm.undefer" \o "sqlalchemy.orm.undefer) option applied.

See [orm.undefer()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "sqlalchemy.orm.undefer" \o "sqlalchemy.orm.undefer) for usage examples.

**undefer\_group**(*loadopt*, *name*)

Produce a new [Load](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.Load" \o "sqlalchemy.orm.Load) object with the [orm.undefer\_group()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "sqlalchemy.orm.undefer_group" \o "sqlalchemy.orm.undefer_group) option applied.

See [orm.undefer\_group()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "sqlalchemy.orm.undefer_group" \o "sqlalchemy.orm.undefer_group) for usage examples.

sqlalchemy.orm.**join**(*left*, *right*, *onclause=None*, *isouter=False*, *full=False*, *join\_to\_left=None*)

Produce an inner join between left and right clauses.

[orm.join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.join" \o "sqlalchemy.orm.join) is an extension to the core join interface provided by [sql.expression.join()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.join" \o "sqlalchemy.sql.expression.join), where the left and right selectables may be not only core selectable objects such as [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table), but also mapped classes or [AliasedClass](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.util.AliasedClass" \o "sqlalchemy.orm.util.AliasedClass) instances. The "on" clause can be a SQL expression, or an attribute or string name referencing a configured [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship).

[orm.join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.join" \o "sqlalchemy.orm.join) is not commonly needed in modern usage, as its functionality is encapsulated within that of the [Query.join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join) method, which features a significant amount of automation beyond [orm.join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.join" \o "sqlalchemy.orm.join) by itself. Explicit usage of [orm.join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.join" \o "sqlalchemy.orm.join) with [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) involves usage of the [Query.select\_from()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.select_from" \o "sqlalchemy.orm.query.Query.select_from)method, as in:

**from** **sqlalchemy.orm** **import** join

session.query(User).\

select\_from(join(User, Address, User.addresses)).\

filter(Address.email\_address=='foo@bar.com')

In modern SQLAlchemy the above join can be written more succinctly as:

session.query(User).\

join(User.addresses).\

filter(Address.email\_address=='foo@bar.com')

See [Query.join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join) for information on modern usage of ORM level joins.

*Changed in version 0.8.1:*- the join\_to\_left parameter is no longer used, and is deprecated.

sqlalchemy.orm.**outerjoin**(*left*, *right*, *onclause=None*, *full=False*, *join\_to\_left=None*)

Produce a left outer join between left and right clauses.

This is the "outer join" version of the [orm.join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.join" \o "sqlalchemy.orm.join) function, featuring the same behavior except that an OUTER JOIN is generated. See that function's documentation for other usage details.

sqlalchemy.orm.**with\_parent**(*instance*, *prop*)

Create filtering criterion that relates this query's primary entity to the given related instance, using established [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) configuration.

创建使用建立的[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)配置将此查询的主实体与给定的相关实例相关联的过滤条件。

The SQL rendered is the same as that rendered when a lazy loader would fire off from the given parent on that attribute, meaning that the appropriate state is taken from the parent object in Python without the need to render joins to the parent table in the rendered statement.

渲染的SQL与渲染的SQL语句相同，这意味着适当的状态是从Python中的父对象获取的，而不需要将渲染连接到呈现的父表中。 声明。

*Changed in version 0.6.4:*This method accepts parent instances in all persistence states, including transient, persistent, and detached. Only the requisite primary key/foreign key attributes need to be populated. Previous versions didn't work with transient instances.

|  |  |
| --- | --- |
| **Parameters:** | * ****instance**** – An instance which has some [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship). * ****property****[¶](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.with_parent.params.property) – String property name, or class-bound attribute, which indicates what relationship from the instance should be used to reconcile the parent/child relationship. |

# Chapter 5 Using the Session

The [orm.mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper) function and [declarative](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "module-sqlalchemy.ext.declarative" \o "sqlalchemy.ext.declarative) extensions are the primary configurational interface for the ORM. Once mappings are configured, the primary usage interface for persistence operations is the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session).

## 5.1 Session Basics

5.1.1 What does the Session do ?

In the most general sense, the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) establishes all conversations with the database and represents a "holding zone" for all the objects which you've loaded or associated with it during its lifespan. It provides the entrypoint to acquire a [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) object, which sends queries to the database using the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) object's current database connection, populating result rows into objects that are then stored in the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session), inside a structure called the [Identity Map](http://martinfowler.com/eaaCatalog/identityMap.html) - a data structure that maintains unique copies of each object, where "unique" means "only one object with a particular primary key".

在最普遍的意义上，[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)建立与数据库的所有会话，并在其生命周期内为所有加载或与之关联的对象表示"保持区域"。它提供了获取[Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query)对象的入口点，该对象使用Session对象的当前数据库连接向数据库发送查询，将结果行填充到然后存储在[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)中的对象中，称为Identity Map - 维护的数据结构每个对象的唯一副本，其中"唯一"是指"只有一个具有特定主键的对象"。

The [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) begins in an essentially stateless form. Once queries are issued or other objects are persisted with it, it requests a connection resource from an [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine)that is associated either with the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) itself or with the mapped [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects being operated upon. This connection represents an ongoing transaction, which remains in effect until the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) is instructed to commit or roll back its pending state.

[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)以基本无状态的形式开始。一旦发出查询或其他对象被持久化，它将请求来自一个[Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine)其连接资源与会话本身或被操作的映射[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)对象相关联。此连接表示正在进行的事务，该事务在指示[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)被提交或回滚到其挂起状态之前仍然有效。

All changes to objects maintained by a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) are tracked - before the database is queried again or before the current transaction is committed, it ****flushes**** all pending changes to the database. This is known as the [Unit of Work](http://martinfowler.com/eaaCatalog/unitOfWork.html) pattern.

跟踪由[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)维护的对象的所有更改 - 在重新查询数据库或提交当前事务之前，将刷新所有未完成的更改到数据库。这被称为工作单位模式。

When using a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session), it's important to note that the objects which are associated with it are ****proxy objects**** to the transaction being held by the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) - there are a variety of events that will cause objects to re-access the database in order to keep synchronized. It is possible to "detach" objects from a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session), and to continue using them, though this practice has its caveats. It's intended that usually, you'd re-associate detached objects with another [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) when you want to work with them again, so that they can resume their normal task of representing database state.

当使用[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)时，重要的是要注意与它相关联的对象是由[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)持有的事务的代理对象 - 存在将导致对象重新访问数据库以保持同步的各种事件。可以从[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)中"分离"对象，并继续使用它们，尽管这种做法有其注意事项。通常，当您想要再次使用它们时，通常会将分离的对象重新关联到另一个会话，以便他们可以恢复表示数据库状态的正常任务。

5.1.2 Getting a Session

[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) is a regular Python class which can be directly instantiated. However, to standardize how sessions are configured and acquired, the [sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker) class is normally used to create a top level [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) configuration which can then be used throughout an application without the need to repeat the configurational arguments.

Session是一个常规的Python类，可以直接实例化。 然而，为了标准化会话的配置和获取方式，[sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker)类通常用于创建一个顶级会话配置，然后可以在整个应用程序中使用，而无需重复配置参数。

The usage of [sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker) is illustrated below:

sessionmaker的用法如下所示：

**from** **sqlalchemy** **import** create\_engine

**from** **sqlalchemy.orm** **import** sessionmaker

*# an Engine, which the Session will use for connection# resources*

some\_engine = create\_engine('postgresql://scott:tiger@localhost/')

*# create a configured "Session" class*

Session = sessionmaker(bind=some\_engine)

*# create a Session*

session = Session()

*# work with sess*

myobject = MyObject('foo', 'bar')

session.add(myobject)

session.commit()

Above, the [sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker) call creates a factory for us, which we assign to the name Session. This factory, when called, will create a new [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) object using the configurational arguments we've given the factory. In this case, as is typical, we've configured the factory to specify a particular [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) for connection resources.

以上，[sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker)调用为我们创建一个工厂，我们将其命名为Session。这个工厂在被调用时，将使用我们给出工厂的配置参数创建一个新的Session对象。在这种情况下，通常情况下，我们已将工厂配置为指定特定的连接资源引擎。

A typical setup will associate the [sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker) with an [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine), so that each [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) generated will use this [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) to acquire connection resources. This association can be set up as in the example above, using the bind argument.

典型的设置将会将会话制作者与引擎相关联，以便生成的每个会话将使用此引擎来获取连接资源。可以使用bind参数在上面的示例中设置此关联。

When you write your application, place the [sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker) factory at the global level. This factory can then be used by the rest of the application as the source of new [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) instances, keeping the configuration for how [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) objects are constructed in one place.

编写应用程序时，请将会话制作工厂置于全局级别。然后，该工厂可以被其他应用程序用作新的Session实例的源，保持Session对象在一个地方如何构造的配置。

The [sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker) factory can also be used in conjunction with other helpers, which are passed a user-defined [sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker) that is then maintained by the helper. Some of these helpers are discussed in the section [When do I construct a Session, when do I commit it, and when do I close it?](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_basics.html" \l "session-faq-whentocreate).

[sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker)工厂也可以与其他助手一起使用，其他帮助者通过用户定义的[sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker)，然后由助手维护。这些帮助器中的一些在本节中讨论了什么时候构建一个Session，什么时候提交它，什么时候关闭它？

### Adding Additional Configuration to an Existing sessionmaker()

A common scenario is where the [sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker) is invoked at module import time, however the generation of one or more [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) instances to be associated with the [sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker) has not yet proceeded. For this use case, the [sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker) construct offers the [sessionmaker.configure()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker.configure" \o "sqlalchemy.orm.session.sessionmaker.configure) method, which will place additional configuration directives into an existing [sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker) that will take place when the construct is invoked:

常见的情况是会话制作者在模块导入时调用的地方，但是与会话制作者关联的一个或多个Engine实例的生成尚未进行。 对于这种用例，sessionmaker构造提供了sessionmaker.configure()方法，该方法会将额外的配置指令放置在调用该构造时将会发生的现有会话制造者中：

**from** **sqlalchemy.orm** **import** sessionmaker

**from** **sqlalchemy** **import** create\_engine

*# configure Session class with desired options*

Session = sessionmaker()

*# later, we create the engine*

engine = create\_engine('postgresql://...')

*# associate it with our custom Session class*

Session.configure(bind=engine)

*# work with the session*

session = Session()

### Creating Ad-Hoc Session Objects with Alternate Arguments

For the use case where an application needs to create a new [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) with special arguments that deviate from what is normally used throughout the application, such as a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) that binds to an alternate source of connectivity, or a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) that should have other arguments such as expire\_on\_commit established differently from what most of the application wants, specific arguments can be passed to the [sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker) factory's [sessionmaker.\_\_call\_\_()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker.__call__" \o "sqlalchemy.orm.session.sessionmaker.__call__) method. These arguments will override whatever configurations have already been placed, such as below, where a new [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) is constructed against a specific [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection):

对于应用程序需要创建一个具有与整个应用程序通常使用的特殊参数（例如绑定到备用连接源的会话）的特殊参数或应具有其他参数的会话（例如， expire\_on\_commit与大多数应用程序所要求的不同，具体的参数可以传递给sessionmaker工厂的sessionmaker .\_\_调用\_\_()方法。 这些参数将覆盖已经放置的任何配置，例如下面，针对特定连接构建新的会话：

*# at the module level, the global sessionmaker,# bound to a specific Engine*

Session = sessionmaker(bind=engine)

*# later, some unit of code wants to create a# Session that is bound to a specific Connection*

conn = engine.connect()session = Session(bind=conn)

The typical rationale for the association of a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) with a specific [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) is that of a test fixture that maintains an external transaction - see [Joining a Session into an External Transaction (such as for test suites)](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_transaction.html" \l "session-external-transaction) for an example of this.

会话与特定连接关联的典型理由是维护外部事务的测试夹具 - 参见将会话连接到外部事务（例如测试套件）中，作为示例。

5.1.3 Session Frequently Asked Questions

By this point, many users already have questions about sessions. This section presents a mini-FAQ (note that we have also a [real FAQ](http://docs.sqlalchemy.org/en/rel_1_1/faq/index.html)) of the most basic issues one is presented with when using a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session).

到目前为止，很多用户已经对会话有疑问了。 本节介绍使用会话时提供的最基本的问题的小型常见问题（请注意，我们也有一个真实的FAQ）。

### When do I make a [sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker)?

Just one time, somewhere in your application's global scope. It should be looked upon as part of your application's configuration. If your application has three .py files in a package, you could, for example, place the [sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker) line in your \_\_init\_\_.py file; from that point on your other modules say "from mypackage import Session". That way, everyone else just uses [Session()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session), and the configuration of that session is controlled by that central point.

只是一次，在你的应用程序的全球范围的某个地方。 应该将其视为应用程序配置的一部分。 如果您的应用程序在程序包中有三个.py文件，则可以将会话制作行放在\_\_init\_\_.py文件中; 从其他模块的那一点说“从mypackage导入会话”。 这样，所有其他人只使用Session()，并且该会话的配置由该中心点控制。

If your application starts up, does imports, but does not know what database it's going to be connecting to, you can bind the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) at the "class" level to the engine later on, using [sessionmaker.configure()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker.configure" \o "sqlalchemy.orm.session.sessionmaker.configure).

如果您的应用程序启动，导入，但不知道将连接到哪个数据库，则可以使用sessionmaker.configure()将会话在“类”级别绑定到引擎。

In the examples in this section, we will frequently show the [sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker) being created right above the line where we actually invoke [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session). But that's just for example's sake! In reality, the [sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker) would be somewhere at the module level. The calls to instantiate [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) would then be placed at the point in the application where database conversations begin.

在本节的例子中，我们将经常显示正在创建正好在实际调用Session的行上面的会话生成器。 但这只是举个例子！ 实际上，会议制作者将处于模块级的某处。 然后，将实例化Session的调用放在数据库对话开始的应用程序中。

### When do I construct a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session), when do I commit it, and when do I close it?

**tl;dr;**

1. As a general rule, keep the lifecycle of the session ****separate and external**** from functions and objects that access and/or manipulate database data. This will greatly help with achieving a predictable and consistent transactional scope.作为一般规则，将会话的生命周期与访问和/或操作数据库数据的函数和对象保持一致。 这将大大有助于实现可预测和一致的交易范围。
2. Make sure you have a clear notion of where transactions begin and end, and keep transactions ****short****, meaning, they end at the series of a sequence of operations, instead of being held open indefinitely.2.确保你有一个明确的交易开始和结束的概念，并保持交易的短暂，意味着它们结束于一系列的操作，而不是无限期地被打开。

A [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) is typically constructed at the beginning of a logical operation where database access is potentially anticipated.

会话通常在逻辑操作开始时构建，其中可能预期数据库访问。

The [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session), whenever it is used to talk to the database, begins a database transaction as soon as it starts communicating. Assuming the autocommit flag is left at its recommended default of False, this transaction remains in progress until the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) is rolled back, committed, or closed.

[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)，无论何时用于与数据库通信，一开始通信就开始数据库事务。 假设autocommit 标志设为默认值False，则该事务将保持进行中，直到[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) 被回滚，提交或关闭为止。

The [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) will begin a new transaction if it is used again, subsequent to the previous transaction ending; from this it follows that the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) is capable of having a lifespan across many transactions, though only one at a time. We refer to these two concepts as ****transaction scope**** and ****session scope****.

如果再次使用[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) ，则会话开始一个新的事务，紧接着之前结束的事务; 因此，[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)拥有的寿命能够跨越多个事务，尽管每次只有一个。 我们将这两个概念称为事务范围和会话范围。

The implication here is that the SQLAlchemy ORM is encouraging the developer to establish these two scopes in their application, including not only when the scopes begin and end, but also the expanse of those scopes, for example should a single [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) instance be local to the execution flow within a function or method, should it be a global object used by the entire application, or somewhere in between these two.

这里的含义是，SQLAlchemy ORM鼓励开发人员在其应用程序中建立这两个范围，包括不仅在范围开始和结束时，而且还扩展了这些范围，例如，如果一个Session实例本地到 一个函数或方法中的执行流程，如果它是整个应用程序使用的全局对象，或者这两者之间的某处。

The burden placed on the developer to determine this scope is one area where the SQLAlchemy ORM necessarily has a strong opinion about how the database should be used. The [unit of work](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-unit-of-work) pattern is specifically one of accumulating changes over time and flushing them periodically, keeping in-memory state in sync with what's known to be present in a local transaction. This pattern is only effective when meaningful transaction scopes are in place.

开发人员确定此范围的负担是SQLAlchemy ORM必须对如何使用数据库有强烈意见的一个领域。 工作模式的单位特别是随时间累积变化并周期性地刷新工作模式，保持内存状态与本地事务中已知存在的状态同步。 此模式仅在有意义的事务范围到位时有效。

It's usually not very hard to determine the best points at which to begin and end the scope of a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session), though the wide variety of application architectures possible can introduce challenging situations.

确定开始和结束会话范围的最佳点通常不是很困难，尽管各种各样的应用程序架构可能会引入具有挑战性的情况。

A common choice is to tear down the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) at the same time the transaction ends, meaning the transaction and session scopes are the same. This is a great choice to start out with as it removes the need to consider session scope as separate from transaction scope.

常见的选择是在事务结束的同时删除会话，这意味着事务和会话作用域是相同的。 这是一个很好的选择，因为它消除了需要将会话范围与事务范围分开。

While there's no one-size-fits-all recommendation for how transaction scope should be determined, there are common patterns. Especially if one is writing a web application, the choice is pretty much established.

虽然对于如何确定交易范围没有一个适合所有的建议，但是有一些常见的模式。 特别是如果一个人正在编写一个Web应用程序，那么这个选择已经很成熟了

A web application is the easiest case because such an application is already constructed around a single, consistent scope - this is the ****request****, which represents an incoming request from a browser, the processing of that request to formulate a response, and finally the delivery of that response back to the client. Integrating web applications with the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) is then the straightforward task of linking the scope of the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) to that of the request. The [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) can be established as the request begins, or using a lazy initialization pattern which establishes one as soon as it is needed. The request then proceeds, with some system in place where application logic can access the current [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) in a manner associated with how the actual request object is accessed. As the request ends, the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) is torn down as well, usually through the usage of event hooks provided by the web framework. The transaction used by the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) may also be committed at this point, or alternatively the application may opt for an explicit commit pattern, only committing for those requests where one is warranted, but still always tearing down the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) unconditionally at the end.

Web应用程序是最简单的情况，因为这样的应用程序已经围绕一个单一的，一致的范围构建 - 即****request****，它表示来自浏览器的传入请求，处理该请求以制定响应，以及最终交付该响应回到客户端。将Web应用程序与会话集成是将[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) 的范围与请求的范围进行链接的直接任务。可以根据请求开始建立[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) ，也可以使用延迟初始化模式，在需要时立即建立该模式。然后，该请求继续进行，其中一些系统就位，其中应用程序逻辑可以以与实际请求对象的访问相关联的方式访问当前[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) 。当请求结束时，通常会通过使用Web框架提供的事件钩子来将会话拆除。会话使用的交易也可能在此时被提交，或者应用程序可能会选择一个明确的提交模式，只允许那些需要保证的请求，但是仍然总是无条件的在最后分解会话。

Some web frameworks include infrastructure to assist in the task of aligning the lifespan of a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) with that of a web request. This includes products such as [Flask-SQLAlchemy](http://packages.python.org/Flask-SQLAlchemy/), for usage in conjunction with the Flask web framework, and [Zope-SQLAlchemy](http://pypi.python.org/pypi/zope.sqlalchemy), typically used with the Pyramid framework. SQLAlchemy recommends that these products be used as available.

一些网络框架包括基础设施，以帮助将会话的使用寿命与Web请求的寿命保持一致。这包括与Flask-SQLAlchemy一起使用的产品，与Flask Web框架一起使用，以及通常与Pyramid框架一起使用的Zope-SQLAlchemy。 SQLAlchemy建议将这些产品可用的来使用。

In those situations where the integration libraries are not provided or are insufficient, SQLAlchemy includes its own "helper" class known as [scoped\_session](http://docs.sqlalchemy.org/en/rel_1_1/orm/contextual.html" \l "sqlalchemy.orm.scoping.scoped_session" \o "sqlalchemy.orm.scoping.scoped_session). A tutorial on the usage of this object is at [Contextual/Thread-local Sessions](http://docs.sqlalchemy.org/en/rel_1_1/orm/contextual.html" \l "unitofwork-contextual). It provides both a quick way to associate a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) with the current thread, as well as patterns to associate [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) objects with other kinds of scopes.

在没有提供或不足够的集成库的情况下，SQLAlchemy包含自己的“助手”类，称为scoped\_session。关于此对象的使用的教程是在[Contextual/Thread-local Sessions](http://docs.sqlalchemy.org/en/rel_1_1/orm/contextual.html" \l "unitofwork-contextual)。它提供了将会话与当前线程关联的快速方式，以及将会话对象与其他种类的作用域相关联的模式。

As mentioned before, for non-web applications there is no one clear pattern, as applications themselves don't have just one pattern of architecture. The best strategy is to attempt to demarcate "operations", points at which a particular thread begins to perform a series of operations for some period of time, which can be committed at the end. Some examples:

如前所述，对于非Web应用程序，没有一个明确的模式，因为应用程序本身不具有一种架构模式。最好的策略是尝试划分“操作”，特定线程在某段时间内开始执行一系列操作的点，最后可以进行一些操作。一些例子：

* A background daemon which spawns off child forks would want to create a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) local to each child process, work with that [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) through the life of the "job" that the fork is handling, then tear it down when the job is completed.产生子分支的背景守护进程将希望为每个子进程创建一个本地Session，在该分支处理的“作业”的整个生命周期内与该Session一起工作，然后在完成作业时将其删除。
* For a command-line script, the application would create a single, global [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) that is established when the program begins to do its work, and commits it right as the program is completing its task.
* For a GUI interface-driven application, the scope of the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) may best be within the scope of a user-generated event, such as a button push. Or, the scope may correspond to explicit user interaction, such as the user "opening" a series of records, then "saving" them.

As a general rule, the application should manage the lifecycle of the session *externally* to functions that deal with specific data. This is a fundamental separation of concerns which keeps data-specific operations agnostic of the context in which they access and manipulate that data.

作为一般规则，应用程序应将外部会话的生命周期管理到处理特定数据的功能。 这是将数据特定操作与其访问和操纵数据的上下文无关的基本分离。

E.g. ****don't do this****:

*### this is the \*\*wrong way to do it\*\* ###*

**class** **ThingOne**(object):

**def** go(self):

session = Session()

**try**:

session.query(FooBar).update({"x": 5})

session.commit()

**except**:

session.rollback()

**raise**

**class** **ThingTwo**(object):

**def** go(self):

session = Session()

**try**:

session.query(Widget).update({"q": 18})

session.commit()

**except**:

session.rollback()

**raise**

**def** run\_my\_program():

ThingOne().go()

ThingTwo().go()

Keep the lifecycle of the session (and usually the transaction) ****separate and external****:

*### this is a \*\*better\*\* (but not the only) way to do it ###*

**class** **ThingOne**(object):

**def** go(self, session):

session.query(FooBar).update({"x": 5})

**class** **ThingTwo**(object):

**def** go(self, session):

session.query(Widget).update({"q": 18})

**def** run\_my\_program():

session = Session()

**try**:

ThingOne().go(session)

ThingTwo().go(session)

session.commit()

**except**:

session.rollback()

**raise**

**finally**:

session.close()

The advanced developer will try to keep the details of session, transaction and exception management as far as possible from the details of the program doing its work. For example, we can further separate concerns using a [context manager](http://docs.python.org/3/library/contextlib.html" \l "contextlib.contextmanager):

高级开发人员将尽量保持会话，事务和异常管理的细节，尽可能避免程序工作的细节。 例如，我们可以使用上下文管理器进一步分离问题：

*### another way (but again \*not the only way\*) to do it ###*

**from** **contextlib** **import** contextmanager

**@contextmanager**

**def** session\_scope():

*"""Provide a transactional scope around a series of operations."""*

session = Session()

**try**:

**yield** session

session.commit()

**except**:

session.rollback()

**raise**

**finally**:

session.close()

**def** run\_my\_program():

**with** session\_scope() **as** session:

ThingOne().go(session)

ThingTwo().go(session)

### Is the Session a cache?

Yeee…no. It's somewhat used as a cache, in that it implements the [identity map](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-identity-map) pattern, and stores objects keyed to their primary key. However, it doesn't do any kind of query caching. This means, if you say session.query(Foo).filter\_by(name='bar'), even if Foo(name='bar') is right there, in the identity map, the session has no idea about that. It has to issue SQL to the database, get the rows back, and then when it sees the primary key in the row, *then* it can look in the local identity map and see that the object is already there. It's only when you say query.get({some primary key}) that the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) doesn't have to issue a query.

Yeee ...没有。它有点用作缓存，因为它实现了标识映射模式，并存储了键入其主键的对象。但是，它不会执行任何类型的查询缓存。这意味着，如果你说session.query（Foo）.filter\_by（name ='bar'），即使Foo（name ='bar'）在那里，在身份地图中，会话也不知道这个。它必须向数据库发出SQL，取回行，然后当它看到行中的主键时，它可以在本地标识映射中查看该对象已经存在。只有当你说query.get（{某个主键}）时，Session才不必发出查询。

Additionally, the Session stores object instances using a weak reference by default. This also defeats the purpose of using the Session as a cache.

此外，Session会默认使用弱引用来存储对象实例。这也违背了使用Session作为缓存的目的。

The [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) is not designed to be a global object from which everyone consults as a "registry" of objects. That's more the job of a ****second level cache****. SQLAlchemy provides a pattern for implementing second level caching using [dogpile.cache](https://dogpilecache.readthedocs.io/), via the [Dogpile Caching](http://docs.sqlalchemy.org/en/rel_1_1/orm/examples.html" \l "examples-caching) example.

[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)的目的不是要成为一个全局的对象，每个人都可以作为对象的“注册表”进行咨询。这更多的是二级缓存的工作。 SQLAlchemy提供了使用dogpile.cache实现二级缓存的模式，通过Dogpile Caching示例。

### How can I get the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) for a certain object?

Use the [object\_session()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.object_session" \o "sqlalchemy.orm.session.Session.object_session) classmethod available on [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session):

使用Session上提供的object\_session()类方法：

session = Session.object\_session(someobject)

The newer [Runtime Inspection API](http://docs.sqlalchemy.org/en/rel_1_1/core/inspection.html) system can also be used:

较新的运行时检测API系统也可以使用：

**from** **sqlalchemy** **import** inspect

session = inspect(someobject).session

### Is the session thread-safe?会话是线程安全的吗?

The [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) is very much intended to be used in a ****non-concurrent**** fashion, which usually means in only one thread at a time.

会话非常有意用于非并发的方式，这通常意味着一次只有一个线程。

The [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) should be used in such a way that one instance exists for a single series of operations within a single transaction. One expedient way to get this effect is by associating a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) with the current thread (see [Contextual/Thread-local Sessions](http://docs.sqlalchemy.org/en/rel_1_1/orm/contextual.html" \l "unitofwork-contextual) for background). Another is to use a pattern where the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) is passed between functions and is otherwise not shared with other threads.

应该以这样一种方式使用会话，即在单个事务中存在单个一系列操作的一个实例。 获得这种效果的一种便捷方式是将会话与当前线程关联（请参阅上下文/线程本地会话作为背景）。 另一个是使用一个模式，其中Session是在函数之间传递的，否则不与其他线程共享。

The bigger point is that you should not *want* to use the session with multiple concurrent threads. That would be like having everyone at a restaurant all eat from the same plate. The session is a local "workspace" that you use for a specific set of tasks; you don't want to, or need to, share that session with other threads who are doing some other task.

更重要的一点是，你不应该使用多个并发线程的会话。 这就好像让一家餐厅的每个人都从同一个盘子里吃东西一样。 会话是一个本地的“工作空间”，您用于一组特定的任务; 您不希望或需要与其他正在执行其他任务的线程共享该会话。

Making sure the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) is only used in a single concurrent thread at a time is called a "share nothing" approach to concurrency. But actually, not sharing the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) implies a more significant pattern; it means not just the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) object itself, but also ****all objects that are associated with that Session****, must be kept within the scope of a single concurrent thread. The set of mapped objects associated with a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) are essentially proxies for data within database rows accessed over a database connection, and so just like the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) itself, the whole set of objects is really just a large-scale proxy for a database connection (or connections). Ultimately, it's mostly the DBAPI connection itself that we're keeping away from concurrent access; but since the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) and all the objects associated with it are all proxies for that DBAPI connection, the entire graph is essentially not safe for concurrent access.

确保会话只在一个单一的并发线程中一次使用称为“无共享”的方法并发。但实际上，不分享会话意味着更重要的模式;它不仅意味着Session对象本身，而且与该Session关联的所有对象都必须保持在单个并发线程的范围内。与Session关联的一组映射对象本质上是数据库连接中访问的数据库行中的数据的代理，所以就像Session本身一样，整个对象实际上只是一个数据库连接的大规模代理（或连接）。最终，它主要是我们远离并发访问的DBAPI连接本身;但是由于会话及与其关联的所有对象都是该DBAPI连接的代理，因此整个图对于并发访问本质上是不安全的。

If there are in fact multiple threads participating in the same task, then you may consider sharing the session and its objects between those threads; however, in this extremely unusual scenario the application would need to ensure that a proper locking scheme is implemented so that there isn't *concurrent* access to the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) or its state. A more common approach to this situation is to maintain a single [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) per concurrent thread, but to instead *copy* objects from one [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) to another, often using the [Session.merge()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.merge" \o "sqlalchemy.orm.session.Session.merge) method to copy the state of an object into a new object local to a different [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session).

如果实际上有多个线程参与同一个任务，则可以考虑在这些线程之间共享会话及其对象;然而，在这种非常不寻常的情况下，应用程序将需要确保实施适当的锁定方案，以便没有并发访问会话或其状态。对于这种情况，更常见的方法是每个并发线程维护一个会话，而是将对象从一个会话复制到另一个会话，通常使用Session.merge（）方法将对象的状态复制到本地的新对象一个不同的会议。

5.1.4 Basics of Using a Session

The most basic [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) use patterns are presented here.

这里介绍最基本的会话使用模式。

### Querying

The [query()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.query" \o "sqlalchemy.orm.session.Session.query) function takes one or more *entities* and returns a new [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) object which will issue mapper queries within the context of this Session. An entity is defined as a mapped class, a [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) object, an orm-enabled *descriptor*, or an AliasedClass object:

[query()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.query" \o "sqlalchemy.orm.session.Session.query)函数接受一个或多个实体，并返回一个新的[Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query)对象，该对象将在本Session的上下文中发出映射器查询。 一个实体被定义为一个映射类，一个[Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper)对象，一个启用orm-enabled的描述符或一个AliasedClass对象：

*# query from a class*

session.query(User).filter\_by(name='ed').all()

*# query with multiple classes, returns tuples*

session.query(User, Address).join('addresses').filter\_by(name='ed').all()

*# query using orm-enabled descriptors*

session.query(User.name, User.fullname).all()

*# query from a mapper*

user\_mapper = class\_mapper(User)session.query(user\_mapper)

When [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) returns results, each object instantiated is stored within the identity map. When a row matches an object which is already present, the same object is returned. In the latter case, whether or not the row is populated onto an existing object depends upon whether the attributes of the instance have been *expired* or not. A default-configured [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) automatically expires all instances along transaction boundaries, so that with a normally isolated transaction, there shouldn't be any issue of instances representing data which is stale with regards to the current transaction.

当查询返回结果时，实例化的每个对象都存储在标识映射中。 当一行匹配已经存在的对象时，返回相同的对象。 在后一种情况下，行是否填充到现有对象取决于实例的属性是否已过期。 默认配置的Session会自动使事务边界上的所有实例过期，所以对于通常是孤立的事务，不应该存在表示与当前事务相关的数据的实例的问题。

The [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) object is introduced in great detail in [Object Relational Tutorial](http://docs.sqlalchemy.org/en/rel_1_1/orm/tutorial.html), and further documented in query\_api\_toplevel.

[Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query)对象在“对象关系教程”中有详细介绍，并在query\_api\_toplevel中进一步介绍。

### Adding New or Existing Items

[add()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.add" \o "sqlalchemy.orm.session.Session.add) is used to place instances in the session. For *transient* (i.e. brand new) instances, this will have the effect of an INSERT taking place for those instances upon the next flush. For instances which are *persistent* (i.e. were loaded by this session), they are already present and do not need to be added. Instances which are *detached* (i.e. have been removed from a session) may be re-associated with a session using this method:

[add()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.add" \o "sqlalchemy.orm.session.Session.add)用于在会话中放置实例。 对于瞬态(即全新)实例，这将对下一次刷新时对这些实例发生INSERT的影响。 对于持久性(即由此会话加载)的实例，它们已经存在并且不需要添加。 分离的实例(即已从会话中删除)可以使用该方法与会话重新关联：

user1 = User(name='user1')

user2 = User(name='user2')

session.add(user1)

session.add(user2)

session.commit() *# write changes to the database*

To add a list of items to the session at once, use [add\_all()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.add_all" \o "sqlalchemy.orm.session.Session.add_all):

要一次向会话添加项目列表，请使用add\_all()：

session.add\_all([item1, item2, item3])

The [add()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.add" \o "sqlalchemy.orm.session.Session.add) operation ****cascades**** along the save-update cascade. For more details see the section [Cascades](http://docs.sqlalchemy.org/en/rel_1_1/orm/cascades.html" \l "unitofwork-cascades).

[add()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.add" \o "sqlalchemy.orm.session.Session.add)操作级联，其沿着save-update级联的。 有关更多详细信息，请参阅Cascades部分。

### Deleting

The [delete()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.delete" \o "sqlalchemy.orm.session.Session.delete) method places an instance into the Session's list of objects to be marked as deleted:

[delete()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.delete" \o "sqlalchemy.orm.session.Session.delete)方法将一个实例放入要被标记为已删除的Session对象列表中：

*# mark two objects to be deleted*

session.delete(obj1)session.delete(obj2)

*# commit (or flush)*

session.commit()

#### Deleting from Collections

A common confusion that arises regarding [delete()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.delete" \o "sqlalchemy.orm.session.Session.delete) is when objects which are members of a collection are being deleted. While the collection member is marked for deletion from the database, this does not impact the collection itself in memory until the collection is expired. Below, we illustrate that even after an Address object is marked for deletion, it's still present in the collection associated with the parent User, even after a flush:

对于[delete()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.delete" \o "sqlalchemy.orm.session.Session.delete)产生的常见混淆是当正在删除作为集合成员的对象时。 当收集成员被标记为从数据库中删除时，这不会在收集过期之前影响内存中的集合本身。 下面我们说明即使在将Address对象标记为删除之后，它仍然存在于与父User关联的集合中，即使是在刷新之后：

**>>>** address = user.addresses[1]

**>>>** session.delete(address)

**>>>** session.flush()

**>>>** address **in** user.addresses

True

When the above session is committed, all attributes are expired. The next access of user.addresses will re-load the collection, revealing the desired state:

当提交上述会话时，所有属性都已过期。 user.addresses的下一次访问将重新加载集合，显示所需的状态：

**>>>** session.commit()

**>>>** address **in** user.addresses

False

The usual practice of deleting items within collections is to forego the usage of [delete()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.delete" \o "sqlalchemy.orm.session.Session.delete) directly, and instead use cascade behavior to automatically invoke the deletion as a result of removing the object from the parent collection. The delete-orphan cascade accomplishes this, as illustrated in the example below:

在集合中删除项目的通常做法是直接放弃[delete()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.delete" \o "sqlalchemy.orm.session.Session.delete)的使用，而是使用级联行为自动调用删除作为从父集合中删除对象的结果。 delete-orphan级联完成此操作，如下面的示例所示：

mapper(User, users\_table, properties={

'addresses':relationship(Address, cascade="all, delete, delete-orphan")})

**del** user.addresses[1]

session.flush()

Where above, upon removing the Address object from the User.addresses collection, the delete-orphan cascade has the effect of marking the Address object for deletion in the same way as passing it to [delete()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.delete" \o "sqlalchemy.orm.session.Session.delete).

在上面的地方，在从User.addresses集合中删除Address对象时，删除孤立级联具有将Address对象标记为删除的效果，就像将其传递给delete()一样。

See also [Cascades](http://docs.sqlalchemy.org/en/rel_1_1/orm/cascades.html" \l "unitofwork-cascades) for detail on cascades.

有关级联的详细信息，请参见级联。

#### Deleting based on Filter Criterion基于过滤标准的删除

The caveat with Session.delete() is that you need to have an object handy already in order to delete. The Query includes a [delete()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.delete" \o "sqlalchemy.orm.query.Query.delete) method which deletes based on filtering criteria:

与Session.delete()的警告是，您需要一个方便的对象才能删除。 查询包括基于过滤条件删除的delete()方法：

session.query(User).filter(User.id==7).delete()

The Query.delete() method includes functionality to "expire" objects already in the session which match the criteria. However it does have some caveats, including that "delete" and "delete-orphan" cascades won't be fully expressed for collections which are already loaded. See the API docs for [delete()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.delete" \o "sqlalchemy.orm.query.Query.delete) for more details.

Query.delete()方法包括“过期”会话中已经符合条件的对象的功能。 然而，它确实有一些注意事项，包括“已删除”和“删除孤立”级联不会完全表达为已经加载的集合。 有关详细信息，请参阅API文档delete()。

### Flushing

When the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) is used with its default configuration, the flush step is nearly always done transparently. Specifically, the flush occurs before any individual [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query)is issued, as well as within the [commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.commit" \o "sqlalchemy.orm.session.Session.commit) call before the transaction is committed. It also occurs before a SAVEPOINT is issued when [begin\_nested()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.begin_nested" \o "sqlalchemy.orm.session.Session.begin_nested) is used.

当Session使用默认配置时，flush操作几乎总是透明地完成的。 具体来说，刷新发生在任何单独的Queryis发出之前，以及在事务提交之前的commit()调用中。 在使用begin\_nested()时也发生SAVEPOINT之前。

Regardless of the autoflush setting, a flush can always be forced by issuing [flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.flush" \o "sqlalchemy.orm.session.Session.flush):

无论自动冲洗设置如何，总是可以通过发出flush()来强制刷新：

session.flush()

The "flush-on-Query" aspect of the behavior can be disabled by constructing [sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker) with the flag autoflush=False:

可以通过使用flag autoflush = False构造sessionmaker来禁用该行为的“flush-on-Query”方面：

Session = sessionmaker(autoflush=**False**)

Additionally, autoflush can be temporarily disabled by setting the autoflush flag at any time:

此外，可以随时设置自动冲洗标志来暂时禁用自动冲洗：

mysession = Session()

mysession.autoflush = **False**

Some autoflush-disable recipes are available at [DisableAutoFlush](http://www.sqlalchemy.org/trac/wiki/UsageRecipes/DisableAutoflush).

DisableAutoFlush提供了一些自动禁止配方。

The flush process *always* occurs within a transaction, even if the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) has been configured with autocommit=True, a setting that disables the session's persistent transactional state. If no transaction is present, [flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.flush" \o "sqlalchemy.orm.session.Session.flush) creates its own transaction and commits it. Any failures during flush will always result in a rollback of whatever transaction is present. If the Session is not in autocommit=True mode, an explicit call to [rollback()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.rollback" \o "sqlalchemy.orm.session.Session.rollback) is required after a flush fails, even though the underlying transaction will have been rolled back already - this is so that the overall nesting pattern of so-called "subtransactions" is consistently maintained.

刷新过程始终发生在事务中，即使会话已配置为autocommit = True，该设置禁用会话的持续事务状态。 如果没有事务存在，flush()将创建自己的事务并提交它。 刷新过程中的任何故障将总是导致任何事务的回滚。 如果会话不在autocommit = True模式，即使底层事务已经被回滚，即使刷新失败，还需要显式调用rollback()，这就是所谓的“ 子事务“一贯保持。

### Committing提交

[commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.commit" \o "sqlalchemy.orm.session.Session.commit) is used to commit the current transaction. It always issues [flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.flush" \o "sqlalchemy.orm.session.Session.flush) beforehand to flush any remaining state to the database; this is independent of the "autoflush" setting. If no transaction is present, it raises an error. Note that the default behavior of the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) is that a "transaction" is always present; this behavior can be disabled by setting autocommit=True. In autocommit mode, a transaction can be initiated by calling the [begin()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.begin" \o "sqlalchemy.orm.session.Session.begin) method.

[commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.commit" \o "sqlalchemy.orm.session.Session.commit)用于提交当前事务。 它总是发生flush()事先刷新任何剩余的状态到数据库; 这是独立于“autoflush”设置。 如果没有事务存在，则会引发错误。 请注意，Session的默认行为是“事务”始终存在; 可以通过设置autocommit = True来禁用此行为。 在自动提交模式下，可以通过调用begin()方法来启动事务。

**Note**

The term "transaction" here refers to a transactional construct within the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) itself which may be maintaining zero or more actual database (DBAPI) transactions. An individual DBAPI connection begins participation in the "transaction" as it is first used to execute a SQL statement, then remains present until the session-level "transaction" is completed. See [Managing Transactions](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_transaction.html" \l "unitofwork-transaction) for further detail.

术语“事务”这里是指会话本身内的一个事务性结构，它可以保持零个或更多的实际数据库(DBAPI)事务。 单个DBAPI连接开始参与“事务”，因为它首先用于执行SQL语句，然后保持存在，直到会话级“事务”完成。 有关详细信息，请参阅管理事务

Another behavior of [commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.commit" \o "sqlalchemy.orm.session.Session.commit) is that by default it expires the state of all instances present after the commit is complete. This is so that when the instances are next accessed, either through attribute access or by them being present in a [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) result set, they receive the most recent state. To disable this behavior, configure[sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker) with expire\_on\_commit=False.

commit()的另一个行为是，默认情况下，它会在提交完成后过期所有存在的实例的状态。 这样当下一次访问实例时，通过属性访问或者通过它们存在于查询结果集中，它们将接收最近的状态。 要禁用此行为，请使用expire\_on\_commit = False配置创建者。

Normally, instances loaded into the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) are never changed by subsequent queries; the assumption is that the current transaction is isolated so the state most recently loaded is correct as long as the transaction continues. Setting autocommit=True works against this model to some degree since the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) behaves in exactly the same way with regard to attribute state, except no transaction is present.

通常，加载到会话中的实例从不会被后续查询所改变; 假设当前事务是隔离的，所以只要事务继续，最近加载的状态是正确的。 设置autocommit = True在某种程度上可以对该模型起作用，因为除了不存在事务之外，会话的行为与属性状态完全相同。

### Rolling Back

[rollback()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.rollback" \o "sqlalchemy.orm.session.Session.rollback) rolls back the current transaction. With a default configured session, the post-rollback state of the session is as follows:

rollback()回滚当前事务。 使用默认配置的会话，会话的回滚后状态如下所示：

* All transactions are rolled back and all connections returned to the connection pool, unless the Session was bound directly to a Connection, in which case the connection is still maintained (but still rolled back).所有事务都将回滚，所有连接都返回到连接池，除非会话直接绑定到连接，在这种情况下，连接仍然维护(但仍然回滚)。
* Objects which were initially in the *pending* state when they were added to the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) within the lifespan of the transaction are expunged, corresponding to their INSERT statement being rolled back. The state of their attributes remains unchanged.在事务生命周期内添加到Session中的初始处于待处理状态的对象将被清除，对应于它们正在回滚的INSERT语句。 他们的属性状态保持不变。
* Objects which were marked as *deleted* within the lifespan of the transaction are promoted back to the *persistent* state, corresponding to their DELETE statement being rolled back. Note that if those objects were first *pending* within the transaction, that operation takes precedence instead.
* 在事务生命周期中被标记为已删除的对象将被提升回持久状态，对应于它们的DELETE语句正在回滚。 请注意，如果这些对象在事务中首次挂起，则该操作将优先。
* All objects not expunged are fully expired.未被清除的所有对象都已完全过期。

With that state understood, the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) may safely continue usage after a rollback occurs.

有了这个状态，会话可以在回滚发生后安全地继续使用。

When a [flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.flush" \o "sqlalchemy.orm.session.Session.flush) fails, typically for reasons like primary key, foreign key, or "not nullable" constraint violations, a [rollback()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.rollback" \o "sqlalchemy.orm.session.Session.rollback) is issued automatically (it's currently not possible for a flush to continue after a partial failure). However, the flush process always uses its own transactional demarcator called a *subtransaction*, which is described more fully in the docstrings for [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session). What it means here is that even though the database transaction has been rolled back, the end user must still issue [rollback()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.rollback" \o "sqlalchemy.orm.session.Session.rollback) to fully reset the state of the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session).

当[flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.flush" \o "sqlalchemy.orm.session.Session.flush)失败时，通常由于主键，外键或“不可为空”约束违反等原因，会自动发出[rollback()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.rollback" \o "sqlalchemy.orm.session.Session.rollback)（部分失败后，当前不可能继续刷新）。 但是，flush过程总是使用自己的称为子事务的事务分界符，在[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)的文档字符串中将对此进行更全面的描述。 这意味着即使数据库事务已经回滚，最终用户仍然必须发出[rollback()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.rollback" \o "sqlalchemy.orm.session.Session.rollback)来完全重置[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)的状态。

### Closing

The [close()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.close" \o "sqlalchemy.orm.session.Session.close) method issues a [expunge\_all()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.expunge_all" \o "sqlalchemy.orm.session.Session.expunge_all), and [releases](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-releases) any transactional/connection resources. When connections are returned to the connection pool, transactional state is rolled back as well.

close()方法发出一个expunge\_all()，并释放任何事务/连接资源。 当连接返回到连接池时，事务状态也被回滚。

## 5.2 State Management

# 5.2.1 Quickie Intro to Object States

It's helpful to know the states which an instance can have within a session:

知道实例在会话中可以拥有的状态是有帮助的：

****Transient**** - an instance that's not in a session, and is not saved to the database; i.e. it has no database identity. The only relationship such an object has to the ORM is that its class has a mapper() associated with it.

瞬态 - 不在会话中的实例，不保存到数据库中; 即它没有数据库标识。 这样一个对象对ORM的唯一关系就是它的类有一个与它相关联的mapper()。

****Pending**** - when you [add()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.add" \o "sqlalchemy.orm.session.Session.add) a transient instance, it becomes pending. It still wasn't actually flushed to the database yet, but it will be when the next flush occurs.

待处理 - 当您添加()一个瞬态实例时，它将变为挂起。 它仍然没有被实际刷新到数据库，但是当下一次冲突发生时。

****Persistent**** - An instance which is present in the session and has a record in the database. You get persistent instances by either flushing so that the pending instances become persistent, or by querying the database for existing instances (or moving persistent instances from other sessions into your local session).

持久性 - 会话中存在并在数据库中具有记录的实例。 您可以通过刷新来获得持久性实例，以便挂起的实例变为持久性，或者通过查询数据库来查找现有实例(或将持久性实例从其他会话移动到本地会话中)。

****Deleted**** - An instance which has been deleted within a flush, but the transaction has not yet completed. Objects in this state are essentially in the opposite of "pending" state; when the session's transaction is committed, the object will move to the detached state. Alternatively, when the session's transaction is rolled back, a deleted object moves *back* to the persistent state.

已删除 - 已在同级别中删除的实例，但该事务尚未完成。 这种状态下的对象基本上与“待决”状态相反; 当会话的事务被提交时，对象将移动到分离状态。 或者，当会话的事务被回滚时，被删除的对象移回到持续状态。

*Changed in version 1.1:*The 'deleted' state is a newly added session object state distinct from the 'persistent' state.

在版本1.1中更改：“已删除”状态是与“持久”状态不同的新添加的会话对象状态。

****Detached**** - an instance which corresponds, or previously corresponded, to a record in the database, but is not currently in any session. The detached object will contain a database identity marker, however because it is not associated with a session, it is unknown whether or not this database identity actually exists in a target database. Detached objects are safe to use normally, except that they have no ability to load unloaded attributes or attributes that were previously marked as "expired".

分离 - 与数据库中的记录对应或之前对应的实例，但当前不在任何会话中。 分离的对象将包含一个数据库标识标识符，但是由于它不与一个会话关联，所以不知道这个数据库标识是否真的存在于目标数据库中。 分离的对象可以正常使用，除了它们无法加载先前标记为“已过期”的卸载属性或属性。

For a deeper dive into all possible state transitions, see the section [Object Lifecycle Events](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_events.html" \l "session-lifecycle-events) which describes each transition as well as how to programmatically track each one.

要深入了解所有可能的状态转换，请参阅描述每个转换的对象生命周期事件以及如何以编程方式跟踪每个转换。

### Getting the Current State of an Object

The actual state of any mapped object can be viewed at any time using the [inspect()](http://docs.sqlalchemy.org/en/rel_1_1/core/inspection.html" \l "sqlalchemy.inspection.inspect" \o "sqlalchemy.inspection.inspect) system:

任何映射对象的实际状态可以随时使用inspect()系统查看：

**>>> from** **sqlalchemy** **import** inspect

**>>>** insp = inspect(my\_object)

**>>>** insp.persistent

True

**See also**

[InstanceState.transient](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState.transient" \o "sqlalchemy.orm.state.InstanceState.transient)

[InstanceState.pending](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState.pending" \o "sqlalchemy.orm.state.InstanceState.pending)

[InstanceState.persistent](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState.persistent" \o "sqlalchemy.orm.state.InstanceState.persistent)

[InstanceState.deleted](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState.deleted" \o "sqlalchemy.orm.state.InstanceState.deleted)

[InstanceState.detached](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState.detached" \o "sqlalchemy.orm.state.InstanceState.detached)

5.2.2 Session Attributes

The [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) itself acts somewhat like a set-like collection. All items present may be accessed using the iterator interface:

会话本身的行为有点像一个集合集合。 存在的所有项目可以使用迭代器接口访问：

**for** obj **in** session:

print(obj)

And presence may be tested for using regular "contains" semantics:

并且可以使用常规的“包含”语义来测试存在：

**if** obj **in** session:

print("Object is present")

The session is also keeping track of all newly created (i.e. pending) objects, all objects which have had changes since they were last loaded or saved (i.e. "dirty"), and everything that's been marked as deleted:

会话还跟踪所有新创建（即挂起）的对象，所有对象自上次加载或保存以来都有更改（即“脏”），以及所有被标记为已删除的对象：

*# pending objects recently added to the Session*

session.new

*# persistent objects which currently have changes detected# (this collection is now created on the fly each time the property is called)*

session.dirty

*# persistent objects that have been marked as deleted via session.delete(obj)*

session.deleted

*# dictionary of all persistent objects, keyed on their# identity key*

session.identity\_map

(Documentation: [Session.new](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.new" \o "sqlalchemy.orm.session.Session.new), [Session.dirty](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.dirty" \o "sqlalchemy.orm.session.Session.dirty), [Session.deleted](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.deleted" \o "sqlalchemy.orm.session.Session.deleted), [Session.identity\_map](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.identity_map" \o "sqlalchemy.orm.session.Session.identity_map)).

5.2.3 Session Referencing Behavior

Objects within the session are *weakly referenced*. This means that when they are dereferenced in the outside application, they fall out of scope from within the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) as well and are subject to garbage collection by the Python interpreter. The exceptions to this include objects which are pending, objects which are marked as deleted, or persistent objects which have pending changes on them. After a full flush, these collections are all empty, and all objects are again weakly referenced.

会话中的对象被弱引用。这意味着当它们在外部应用程序中被取消引用时，它们也从Session内部脱离了范围，并且被Python解释器收集垃圾。其例外包括待处理的对象，被标记为已删除的对象，或持久对象在其上具有未决更改。完全刷新后，这些集合都是空的，所有对象都被弱引用。

To cause objects in the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) to remain strongly referenced, usually a simple approach is all that's needed. Examples of externally managed strong-referencing behavior include loading objects into a local dictionary keyed to their primary key, or into lists or sets for the span of time that they need to remain referenced. These collections can be associated with a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session), if desired, by placing them into the [Session.info](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.info" \o "sqlalchemy.orm.session.Session.info) dictionary.

为了使会话中的对象保持强烈引用，通常需要一个简单的方法。外部管理的强引用行为的示例包括将对象加载到键入其主键的本地字典中，或将其添加到需要保持引用的时间跨度的列表中。如果需要，这些集合可以与会话关联，将它们放在Session.info字典中。

An event based approach is also feasible. A simple recipe that provides "strong referencing" behavior for all objects as they remain within the [persistent](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-persistent) state is as follows:

基于事件的方法也是可行的。所有对象在持续状态下提供“强引用”行为的简单配方如下：

**from** **sqlalchemy** **import** event

**def** strong\_reference\_session(session):

**@event**.listens\_for(session, "pending\_to\_persistent")

**@event**.listens\_for(session, "deleted\_to\_persistent")

**@event**.listens\_for(session, "detached\_to\_persistent")

**@event**.listens\_for(session, "loaded\_as\_persistent")

**def** strong\_ref\_object(sess, instance):

**if** 'refs' **not** **in** sess.info:

sess.info['refs'] = refs = set()

**else**:

refs = sess.info['refs']

refs.add(instance)

**@event**.listens\_for(session, "persistent\_to\_detached")

**@event**.listens\_for(session, "persistent\_to\_deleted")

**@event**.listens\_for(session, "persistent\_to\_transient")

**def** deref\_object(sess, instance):

sess.info['refs'].discard(instance)

Above, we intercept the [SessionEvents.pending\_to\_persistent()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.pending_to_persistent" \o "sqlalchemy.orm.events.SessionEvents.pending_to_persistent), [SessionEvents.detached\_to\_persistent()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.detached_to_persistent" \o "sqlalchemy.orm.events.SessionEvents.detached_to_persistent),[SessionEvents.deleted\_to\_persistent()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.deleted_to_persistent" \o "sqlalchemy.orm.events.SessionEvents.deleted_to_persistent) and [SessionEvents.loaded\_as\_persistent()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.loaded_as_persistent" \o "sqlalchemy.orm.events.SessionEvents.loaded_as_persistent) event hooks in order to intercept objects as they enter the [persistent](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-persistent) transition, and the [SessionEvents.persistent\_to\_detached()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.persistent_to_detached" \o "sqlalchemy.orm.events.SessionEvents.persistent_to_detached) and [SessionEvents.persistent\_to\_deleted()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.persistent_to_deleted" \o "sqlalchemy.orm.events.SessionEvents.persistent_to_deleted) hooks to intercept objects as they leave the persistent state.

以上，我们拦截SessionEvents.pending\_to\_persistent()，SessionEvents.detached\_to\_persistent()，SessionEvents.deleted\_to\_persistent()和SessionEvents.loaded\_as\_persistent()事件钩子，以便在进入永久转换时截取对象，并且SessionEvents.persistent\_to\_detached()和 SessionEvents.persistent\_to\_deleted()挂钩拦截对象，因为它们离开持久状态。

The above function may be called for any [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) in order to provide strong-referencing behavior on a per-[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) basis:

可以为任何Session调用上述函数，以便在每个Session的基础上提供强引用行为：

**from** **sqlalchemy.orm** **import** Session

my\_session = Session()

strong\_reference\_session(my\_session)

It may also be called for any [sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker):

也可以要求任何[sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker)：

**from** **sqlalchemy.orm** **import** sessionmaker

maker = sessionmaker()

strong\_reference\_session(maker)

### 5.2.4 Merging

### [merge()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.merge" \o "sqlalchemy.orm.session.Session.merge) transfers state from an outside object into a new or already existing instance within a session. It also reconciles the incoming data against the state of the database, producing a history stream which will be applied towards the next flush, or alternatively can be made to produce a simple "transfer" of state without producing change history or accessing the database. Usage is as follows:

merge()将状态从外部对象转移到会话中的新的或已存在的实例。 它还将输入的数据与数据库的状态进行对照，产生将被应用到下一个刷新的历史流，或者可以使其产生简单的“传送”状态而不产生改变历史或访问数据库。 用法如下：

merged\_object = session.merge(existing\_object)

When given an instance, it follows these steps:

当给出一个实例时，它遵循以下步骤：

It examines the primary key of the instance. If it's present, it attempts to locate that instance in the local identity map. If the load=True flag is left at its default, it also checks the database for this primary key if not located locally.

它检查实例的主键。如果它存在，它会尝试在该本地身份映射中找到该实例。如果load = True标志处于默认状态，它还会检查数据库是否存在此主键，如果不在本地。

If the given instance has no primary key, or if no instance can be found with the primary key given, a new instance is created.

如果给定的实例没有主键，或者如果没有给出主键可以找到实例，则会创建一个新的实例。

The state of the given instance is then copied onto the located/newly created instance. For attributes which are present on the source instance, the value is transferred to the target instance. For mapped attributes which aren't present on the source, the attribute is expired on the target instance, discarding its existing value.

然后将给定实例的状态复制到定位/新创建的实例上。对于源实例上存在的属性，该值将传输到目标实例。对于源中不存在的映射属性，属性在目标实例上过期，丢弃其现有值。

If the load=True flag is left at its default, this copy process emits events and will load the target object's unloaded collections for each attribute present on the source object, so that the incoming state can be reconciled against what's present in the database. If load is passed as False, the incoming data is "stamped" directly without producing any history.

如果load = True标志处于默认状态，则此复制过程会发出事件，并将为源对象上存在的每个属性加载目标对象的未加载集合，以便可以将传入状态与数据库中存在的状态进行对帐。如果加载被传递为False，则传入的数据将直接“加盖”而不产生任何历史记录。

The operation is cascaded to related objects and collections, as indicated by the merge cascade (see [Cascades](http://docs.sqlalchemy.org/en/rel_1_1/orm/cascades.html" \l "unitofwork-cascades)).

该操作级联到相关对象和集合，如并置级联所示（请参阅级联）。

The new instance is returned.

新实例被返回。

With [merge()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.merge" \o "sqlalchemy.orm.session.Session.merge), the given "source" instance is not modified nor is it associated with the target [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session), and remains available to be merged with any number of other [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) objects. [merge()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.merge" \o "sqlalchemy.orm.session.Session.merge) is useful for taking the state of any kind of object structure without regard for its origins or current session associations and copying its state into a new session. Here's some examples:

使用merge()，给定的“源”实例不会被修改，也不与目标会话相关联，并且仍然可以与任何其他Session对象合并。 merge()对于采取任何类型的对象结构的状态是有用的，而不考虑其起源或当前会话关联，并将其状态复制到新会话中。 这里有一些例子：

An application which reads an object structure from a file and wishes to save it to the database might parse the file, build up the structure, and then use[merge()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.merge" \o "sqlalchemy.orm.session.Session.merge) to save it to the database, ensuring that the data within the file is used to formulate the primary key of each element of the structure. Later, when the file has changed, the same process can be re-run, producing a slightly different object structure, which can then be merged in again, and the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) will automatically update the database to reflect those changes, loading each object from the database by primary key and then updating its state with the new state given.

从文件读取对象结构并希望将其保存到数据库的应用程序可能会解析文件，构建结构，然后使用usemerge()将其保存到数据库，确保文件中的数据被用于 制定结构的每个元素的主键。 后来，当文件发生变化时，可以重新运行相同的进程，产生稍微不同的对象结构，然后可以再次合并，会话将自动更新数据库以反映这些更改，从 数据库通过主键，然后用给定的新状态更新其状态。

An application is storing objects in an in-memory cache, shared by many [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) objects simultaneously. [merge()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.merge" \o "sqlalchemy.orm.session.Session.merge) is used each time an object is retrieved from the cache to create a local copy of it in each [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) which requests it. The cached object remains detached; only its state is moved into copies of itself that are local to individual [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) objects.

应用程序将对象存储在内存中的高速缓存中，同时由许多Session对象共享。 每次从缓存中检索到一个对象时，会使用merge()来在每个请求它的Session中创建一个本地副本。 缓存的对象保持分离; 只有它的状态被移动到单个Session对象本地的本身的副本。

In the caching use case, it's common to use the load=False flag to remove the overhead of reconciling the object's state with the database. There's also a "bulk" version of [merge()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.merge" \o "sqlalchemy.orm.session.Session.merge) called [merge\_result()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.merge_result" \o "sqlalchemy.orm.query.Query.merge_result) that was designed to work with cache-extended [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) objects - see the section [Dogpile Caching](http://docs.sqlalchemy.org/en/rel_1_1/orm/examples.html" \l "examples-caching).

在缓存用例中，通常使用load = False标志来消除将对象的状态与数据库协调的开销。 还有一个名为merge\_result()的“批量”版本被设计为使用缓存扩展的Query对象 - 参见Dogpile Caching部分。

An application wants to transfer the state of a series of objects into a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) maintained by a worker thread or other concurrent system. [merge()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.merge" \o "sqlalchemy.orm.session.Session.merge) makes a copy of each object to be placed into this new [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session). At the end of the operation, the parent thread/process maintains the objects it started with, and the thread/worker can proceed with local copies of those objects.

应用程序希望将一系列对象的状态转移到由工作线程或其他并发系统维护的会话中。 merge()使每个对象的副本被放置到这个新的Session中。 在操作结束时，父线程/进程维护它启动的对象，线程/工作人员可以继续执行这些对象的本地副本。

In the "transfer between threads/processes" use case, the application may want to use the load=False flag as well to avoid overhead and redundant SQL queries as the data is transferred.

在“线程/进程之间的传输”用例中，应用程序可能希望使用load = False标志，以避免数据传输时的开销和冗余SQL查询。

### Merge Tips

[merge()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.merge" \o "sqlalchemy.orm.session.Session.merge) is an extremely useful method for many purposes. However, it deals with the intricate border between objects that are transient/detached and those that are persistent, as well as the automated transference of state. The wide variety of scenarios that can present themselves here often require a more careful approach to the state of objects. Common problems with merge usually involve some unexpected state regarding the object being passed to [merge()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.merge" \o "sqlalchemy.orm.session.Session.merge).

对于许多目的，merge()是非常有用的方法。 然而，它涉及瞬态/分离的对象与持久的对象之间的错综复杂的边界，以及状态的自动转移。 可以呈现在这里的各种场景通常需要对对象状态的更仔细的处理。 合并的常见问题通常涉及到传递给merge()的对象的一些意外状态。

Lets use the canonical example of the User and Address objects:

让我们使用用户和地址对象的规范示例：

**class** **User**(Base):

\_\_tablename\_\_ = 'user'

id = Column(Integer, primary\_key=**True**)

name = Column(String(50), nullable=**False**)

addresses = relationship("Address", backref="user")

**class** **Address**(Base):

\_\_tablename\_\_ = 'address'

id = Column(Integer, primary\_key=**True**)

email\_address = Column(String(50), nullable=**False**)

user\_id = Column(Integer, ForeignKey('user.id'), nullable=**False**)

Assume a User object with one Address, already persistent:

假设一个User对象具有一个Address，已经持久：

**>>>** u1 = User(name='ed', [addresses=[Address(email\_address='ed@ed.com')])](mailto:addresses=[Address(email_address='ed@ed.com')]))

**>>>** session.add(u1)

**>>>** session.commit()

We now create a1, an object outside the session, which we'd like to merge on top of the existing Address:

我们现在创建一个会话之外的一个对象，我们希望在现有的地址之上合并：

**>>>** existing\_a1 = u1.addresses[0]

**>>>** a1 = Address(id=existing\_a1.id)

A surprise would occur if we said this:

如果我们这样说：

**>>>** a1.user = u1

**>>>** a1 = session.merge(a1)

**>>>** session.commit()

sqlalchemy.orm.exc.FlushError: New instance <Address at 0x1298f50>with identity key (<class '\_\_main\_\_.Address'>, (1,)) conflicts withpersistent instance <Address at 0x12a25d0>

Why is that ? We weren't careful with our cascades. The assignment of a1.user to a persistent object cascaded to the backref of User.addresses and made our a1 object pending, as though we had added it. Now we have *two* Address objects in the session:

这是为什么 ？ 我们并不小心我们的瀑布。 将a1.user分配给一个持久化对象，级联到User.addresses的backref，并将我们的a1对象挂起，就像我们已经添加一样。 现在我们在会话中有两个Address对象：

**>>>** a1 = Address()

**>>>** a1.user = u1

**>>>** a1 **in** sessionTrue

**>>>** existing\_a1 **in** sessionTrue

**>>>** a1 **is** existing\_a1

False

Above, our a1 is already pending in the session. The subsequent [merge()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.merge" \o "sqlalchemy.orm.session.Session.merge) operation essentially does nothing. Cascade can be configured via the [cascade](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.cascade" \o "sqlalchemy.orm.relationship) option on [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship), although in this case it would mean removing the save-update cascade from the User.addresses relationship - and usually, that behavior is extremely convenient. The solution here would usually be to not assign a1.user to an object already persistent in the target session.

以上，我们的a1在会议中已经有待处理。 后续的merge()操作基本上什么都不做。 可以通过关联()的级联选项来配置级联，尽管在这种情况下，这意味着从User.addresses关系中删除保存更新级联 - 通常，该行为非常方便。 这里的解决方法通常是不将a1.user分配给目标会话中已经持久化的对象。

The cascade\_backrefs=False option of [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) will also prevent the Address from being added to the session via the a1.user = u1assignment.

relationship()的cascade\_backrefs = False选项也将阻止通过a1.user = u1assignment将地址添加到会话中。

Further detail on cascade operation is at [Cascades](http://docs.sqlalchemy.org/en/rel_1_1/orm/cascades.html" \l "unitofwork-cascades).

级联操作的更多细节是Cascades。

Another example of unexpected state:

意外状态的另一个例子：

**>>>** a1 = Address(id=existing\_a1.id, user\_id=u1.id)

**>>> assert** a1.user **is** **None**

**>>> True**

**>>>** a1 = session.merge(a1)

**>>>** session.commit()

sqlalchemy.exc.IntegrityError:

(IntegrityError) address.user\_idmay not be NULL

Here, we accessed a1.user, which returned its default value of None, which as a result of this access, has been placed in the \_\_dict\_\_ of our object a1. Normally, this operation creates no change event, so the user\_id attribute takes precedence during a flush. But when we merge the Address object into the session, the operation is equivalent to:

在这里，我们访问了a1.user，它返回的默认值为None，作为此访问的结果已被放置在对象a1的\_\_dict\_\_中。 通常，此操作不会创建任何更改事件，因此在刷新过程中，user\_id属性优先。 但是当我们将Address对象合并到会话中时，操作等效于：

**>>>** existing\_a1.id = existing\_a1.id

**>>>** existing\_a1.user\_id = u1.id

**>>>** existing\_a1.user = **None**

Where above, both user\_id and user are assigned to, and change events are emitted for both. The user association takes precedence, and None is applied to user\_id, causing a failure.

在上面的位置，user\_id和user都被分配给，并且为两者都发出改变事件。 用户关联优先，“None”应用于user\_id，导致失败。

Most [merge()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.merge" \o "sqlalchemy.orm.session.Session.merge) issues can be examined by first checking - is the object prematurely in the session ?

大多数merge()问题可以通过首先检查来检查 - 会话中提前的对象？

>>> a1 = Address(id=existing\_a1, user\_id=user.id)

>>> **assert** a1 **not** **in** session

>>> a1 = session.merge(a1)

Or is there state on the object that we don't want ? Examining \_\_dict\_\_ is a quick way to check:

还是在对象上有我们不想要的状态？ 检查\_\_dict\_\_是一种快速检查方法：

**>>>** a1 = Address(id=existing\_a1, user\_id=user.id)

**>>>** a1.user

**>>>** a1.\_\_dict\_\_

{'\_sa\_instance\_state': <sqlalchemy.orm.state.InstanceState object at 0x1298d10>, 'user\_id': 1, 'id': 1, 'user': None}

**>>>** *# we don't want user=None merged, remove it*

**>>> del** a1.user

**>>>** a1 = session.merge(a1)

**>>>** *# success*

**>>>** session.commit()

### 5.2.5 Expunging

Expunge removes an object from the Session, sending persistent instances to the detached state, and pending instances to the transient state:

Expunge从会话中删除一个对象，将持久性实例发送到分离状态，并将待处理实例发送到暂态状态：

session.expunge(obj1)

To remove all items, call [expunge\_all()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.expunge_all" \o "sqlalchemy.orm.session.Session.expunge_all) (this method was formerly known as clear()).

要删除所有项目，请调用expunge\_all()（此方法以前称为clear()）。

5.2.6 Refreshing / Expiring

[Expiring](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-expiring) means that the database-persisted data held inside a series of object attributes is erased, in such a way that when those attributes are next accessed, a SQL query is emitted which will refresh that data from the database.

到期意味着在一系列对象属性中保存的数据库持久化数据将被清除，以使得当下一次访问这些属性时，会发出一个SQL查询，该查询将从数据库刷新该数据。

When we talk about expiration of data we are usually talking about an object that is in the [persistent](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-persistent) state. For example, if we load an object as follows:

当我们谈论数据的到期时，我们通常在谈论处于持久状态的对象。 例如，如果我们加载一个对象，如下所示：

user = session.query(User).filter\_by(name='user1').first()

The above User object is persistent, and has a series of attributes present; if we were to look inside its \_\_dict\_\_, we'd see that state loaded:

上述用户对象是持久性的，具有一系列属性; 如果我们要查看它的\_\_dict\_\_，我们会看到状态加载：

**>>>** user.\_\_dict\_\_{ 'id': 1, 'name': u'user1', '\_sa\_instance\_state': <...>,}

where id and name refer to those columns in the database. \_sa\_instance\_state is a non-database-persisted value used by SQLAlchemy internally (it refers to the [InstanceState](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState" \o "sqlalchemy.orm.state.InstanceState) for the instance. While not directly relevant to this section, if we want to get at it, we should use the [inspect()](http://docs.sqlalchemy.org/en/rel_1_1/core/inspection.html" \l "sqlalchemy.inspection.inspect" \o "sqlalchemy.inspection.inspect) function to access it).

其中id和name指的是数据库中的那些列。 \_sa\_instance\_state是SQLAlchemy内部使用的非数据库持久值（它指的是实例的InstanceState），虽然与本节不直接相关，但如果我们想要得到它，我们应该使用inspect()函数来访问它）。

At this point, the state in our User object matches that of the loaded database row. But upon expiring the object using a method such as [Session.expire()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.expire" \o "sqlalchemy.orm.session.Session.expire), we see that the state is removed:

此时，User对象中的状态与加载的数据库行的状态相匹配。 但是在使用诸如Session.expire()之类的方法到期时，我们看到状态被删除：

**>>>** session.expire(user)

**>>>** user.\_\_dict\_\_{'\_sa\_instance\_state': <...>}

We see that while the internal "state" still hangs around, the values which correspond to the id and name columns are gone. If we were to access one of these columns and are watching SQL, we'd see this:

我们看到，虽然内部的“状态”仍然挂起，但对应于id和name列的值已经消失了。 如果我们要访问这些列之一并正在观察SQL，我们会看到：

>>> **print**(user.name)

user1

SELECT user.id AS user\_id, user.name AS user\_name

FROM user

WHERE user.id = ?

(1,)

Above, upon accessing the expired attribute user.name, the ORM initiated a [lazy load](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-lazy-load) to retrieve the most recent state from the database, by emitting a SELECT for the user row to which this user refers. Afterwards, the \_\_dict\_\_ is again populated:

以上，在访问过期的属性user.name时，ORM通过为用户引用的用户行发出一个SELECT，发起了一个延迟加载来从数据库中检索最近的状态。 之后，\_\_dict\_\_再次填充：

**>>>** user.\_\_dict\_\_

{ 'id': 1, 'name': u'user1', '\_sa\_instance\_state': <...>,}

**Note**

While we are peeking inside of \_\_dict\_\_ in order to see a bit of what SQLAlchemy does with object attributes, we ****should not modify**** the contents of \_\_dict\_\_ directly, at least as far as those attributes which the SQLAlchemy ORM is maintaining (other attributes outside of SQLA's realm are fine). This is because SQLAlchemy uses [descriptors](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-descriptors) in order to track the changes we make to an object, and when we modify \_\_dict\_\_ directly, the ORM won't be able to track that we changed something.

虽然我们正在窥视\_\_dict\_\_内部以查看SQLAlchemy对象属性的一些内容，但我们不应该直接修改\_\_dict\_\_的内容，至少与SQLAlchemy ORM所维护的属性一致（SQLA的其他属性 领域都很好） 这是因为SQLAlchemy使用描述符来跟踪对对象所做的更改，当我们直接修改\_\_dict\_\_时，ORM将无法跟踪我们改变的内容。

Another key behavior of both [expire()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.expire" \o "sqlalchemy.orm.session.Session.expire) and [refresh()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.refresh" \o "sqlalchemy.orm.session.Session.refresh) is that all un-flushed changes on an object are discarded. That is, if we were to modify an attribute on our User:

expire()和refresh()的另一个关键行为是丢弃对象上的所有未刷新的更改。 也就是说，如果我们修改我们的用户的属性：

**>>>** user.name = 'user2'

but then we call [expire()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.expire" \o "sqlalchemy.orm.session.Session.expire) without first calling [flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.flush" \o "sqlalchemy.orm.session.Session.flush), our pending value of 'user2' is discarded:

但是我们在没有调用flush()的情况下调用expire()，我们的待处理值“user2”被丢弃：

**>>>** session.expire(user)

**>>>** user.name'user1'

The [expire()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.expire" \o "sqlalchemy.orm.session.Session.expire) method can be used to mark as "expired" all ORM-mapped attributes for an instance:

expire()方法可以用于将一个实例的所有ORM映射属性标记为“已过期”：

*# expire all ORM-mapped attributes on obj1*

session.expire(obj1)

it can also be passed a list of string attribute names, referring to specific attributes to be marked as expired:

它也可以传递字符串属性名称列表，引用要标记为已过期的特定属性：

*# expire only attributes obj1.attr1, obj1.attr2*

session.expire(obj1, ['attr1', 'attr2'])

The [refresh()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.refresh" \o "sqlalchemy.orm.session.Session.refresh) method has a similar interface, but instead of expiring, it emits an immediate SELECT for the object's row immediately:

refresh()方法有一个类似的接口，但是不是过期，它立即为对象的行发出立即SELECT：

*# reload all attributes on obj1*

session.refresh(obj1)

[refresh()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.refresh" \o "sqlalchemy.orm.session.Session.refresh) also accepts a list of string attribute names, but unlike [expire()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.expire" \o "sqlalchemy.orm.session.Session.expire), expects at least one name to be that of a column-mapped attribute:

refresh()也接受字符串属性名称列表，但与expire()不同，期望至少有一个名称是列映射属性的名称：

*# reload obj1.attr1, obj1.attr2*

session.refresh(obj1, ['attr1', 'attr2'])

The [Session.expire\_all()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.expire_all" \o "sqlalchemy.orm.session.Session.expire_all) method allows us to essentially call [Session.expire()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.expire" \o "sqlalchemy.orm.session.Session.expire) on all objects contained within the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) at once:

Session.expire\_all()方法使我们能够在Session中包含的所有对象基本上调用Session.expire()：

session.expire\_all()

### What Actually Loads

The SELECT statement that's emitted when an object marked with [expire()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.expire" \o "sqlalchemy.orm.session.Session.expire) or loaded with [refresh()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.refresh" \o "sqlalchemy.orm.session.Session.refresh) varies based on several factors, including:

标记为expire()或加载了refresh()的对象在基于以下几个因素的情况下发生的SELECT语句，包括：

* The load of expired attributes is triggered from ****column-mapped attributes only****. While any kind of attribute can be marked as expired, including a[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) - mapped attribute, accessing an expired [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) attribute will emit a load only for that attribute, using standard relationship-oriented lazy loading. Column-oriented attributes, even if expired, will not load as part of this operation, and instead will load when any column-oriented attribute is accessed.过期属性的加载仅从列映射属性触发。 虽然任何类型的属性都可以被标记为已过期，包括arelationship() - 映射属性，访问过期的关系()属性将仅为该属性发出一个加载，使用标准的面向对象的延迟加载。 即使过期的列导向属性也不会作为此操作的一部分加载，而是在访问任何面向列的属性时加载。
* [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)- mapped attributes will not load in response to expired column-based attributes being accessed.relationship() - 映射属性将不会加载以响应正在访问的基于列的属性。
* Regarding relationships, [refresh()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.refresh" \o "sqlalchemy.orm.session.Session.refresh) is more restrictive than [expire()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.expire" \o "sqlalchemy.orm.session.Session.expire) with regards to attributes that aren't column-mapped. Calling [refresh()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.refresh" \o "sqlalchemy.orm.session.Session.refresh) and passing a list of names that only includes relationship-mapped attributes will actually raise an error. In any case, non-eager-loading [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) attributes will not be included in any refresh operation.关于关系，与没有列映射的属性相比，refresh()比expire()更受限制。 调用refresh()并传递只包含关系映射属性的名称列表实际上会引发错误。 在任何情况下，非加载关系()属性将不包括在任何刷新操作中。
* [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) attributes configured as "eager loading" via the [lazy](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.lazy" \o "sqlalchemy.orm.relationship) parameter will load in the case of [refresh()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.refresh" \o "sqlalchemy.orm.session.Session.refresh), if either no attribute names are specified, or if their names are inclued in the list of attributes to be refreshed.如果没有指定属性名称，或者如果其名称包含在要刷新的属性列表中，则通过lazy参数配置为“eager loading”的relationship()属性将在refresh()的情况下加载。
* Attributes that are configured as [deferred()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "sqlalchemy.orm.deferred" \o "sqlalchemy.orm.deferred) will not normally load, during either the expired-attribute load or during a refresh. An unloaded attribute that's [deferred()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "sqlalchemy.orm.deferred" \o "sqlalchemy.orm.deferred) instead loads on its own when directly accessed, or if part of a "group" of deferred attributes where an unloaded attribute in that group is accessed.在过期属性加载期间或刷新期间，配置为deferred()的属性通常不会加载。 deferred()的卸载属性可以直接访问时自行加载，也可以访问该组中未加载属性的延迟属性的“组”的一部分。
* For expired attributes that are loaded on access, a joined-inheritance table mapping will emit a SELECT that typically only includes those tables for which unloaded attributes are present. The action here is sophisticated enough to load only the parent or child table, for example, if the subset of columns that were originally expired encompass only one or the other of those tables.对于在访问上加载的过期属性，连接继承表映射将发出一个SELECT，通常只包括存在卸载属性的那些表。 这里的操作非常复杂，只能加载父表或子表，例如，如果最初过期的列的子集只包含这些表中的一个或另一个。
* When [refresh()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.refresh" \o "sqlalchemy.orm.session.Session.refresh) is used on a joined-inheritance table mapping, the SELECT emitted will resemble that of when [Session.query()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.query" \o "sqlalchemy.orm.session.Session.query) is used on the target object's class. This is typically all those tables that are set up as part of the mapping.当在连接继承表映射上使用refresh()时，发出的SELECT将类似于在目标对象的类上使用Session.query()时。 这通常是作为映射的一部分设置的所有这些表。

### When to Expire or Refresh

The [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) uses the expiration feature automatically whenever the transaction referred to by the session ends. Meaning, whenever [Session.commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.commit" \o "sqlalchemy.orm.session.Session.commit) or [Session.rollback()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.rollback" \o "sqlalchemy.orm.session.Session.rollback) is called, all objects within the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) are expired, using a feature equivalent to that of the [Session.expire\_all()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.expire_all" \o "sqlalchemy.orm.session.Session.expire_all) method. The rationale is that the end of a transaction is a demarcating point at which there is no more context available in order to know what the current state of the database is, as any number of other transactions may be affecting it. Only when a new transaction starts can we again have access to the current state of the database, at which point any number of changes may have occurred.会话结束时，Session会自动使用到期功能。 意思是，每当调用Session.commit()或Session.rollback()时，会话中的所有对象都将过期，使用与Session.expire\_all()方法相同的功能。 理由是交易的结束是一个划分点，在这个分界点上没有更多的上下文可用，以便知道数据库的当前状态是什么，因为任何数量的其他交易都可能会影响它。 只有当新的事务启动时，我们可以再次访问数据库的当前状态，此时可能会发生任何数量的更改。

**Transaction Isolation**

Of course, most databases are capable of handling multiple transactions at once, even involving the same rows of data. When a relational database handles multiple transactions involving the same tables or rows, this is when the [isolation](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-isolation) aspect of the database comes into play. The isolation behavior of different databases varies considerably and even on a single database can be configured to behave in different ways (via the so-called isolation level setting). In that sense, the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) can't fully predict when the same SELECT statement, emitted a second time, will definitely return the data we already have, or will return new data. So as a best guess, it assumes that within the scope of a transaction, unless it is known that a SQL expression has been emitted to modify a particular row, there's no need to refresh a row unless explicitly told to do so.

当然，大多数数据库能够一次性处理多个事务，甚至涉及相同的数据行。 当关系数据库处理涉及相同表或行的多个事务时，这是数据库的隔离方面发挥作用。 不同数据库的隔离行为有很大差异，即使在单个数据库中也可以配置为以不同的方式运行(通过所谓的隔离级别设置)。 在这个意义上说，Session不能完全预测什么时候同一个SELECT语句，第二次发出，一定会返回我们已经拥有的数据，否则会返回新的数据。 所以作为一个最好的猜测，它假设在一个事务的范围内，除非知道已经发出了一个SQL表达式来修改一个特定的行，否则不需要刷新一行，除非有明确的说法。

The [Session.expire()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.expire" \o "sqlalchemy.orm.session.Session.expire) and [Session.refresh()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.refresh" \o "sqlalchemy.orm.session.Session.refresh) methods are used in those cases when one wants to force an object to re-load its data from the database, in those cases when it is known that the current state of data is possibly stale. Reasons for this might include:

在这些情况下，使用Session.expire()和Session.refresh()方法，当需要强制对象从数据库重新加载数据时，在这种情况下，当知道数据的当前状态可能 陈旧。 其原因可能包括：

* some SQL has been emitted within the transaction outside of the scope of the ORM's object handling, such as if a [Table.update()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.update" \o "sqlalchemy.schema.Table.update) construct were emitted using the [Session.execute()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.execute" \o "sqlalchemy.orm.session.Session.execute) method;一些SQL已经在ORM的对象处理范围之外的事务中被发出，例如使用Session.execute()方法发出一个Table.update()结构;
* if the application is attempting to acquire data that is known to have been modified in a concurrent transaction, and it is also known that the isolation rules in effect allow this data to be visible.如果应用程序尝试获取已知在并发事务中已被修改的数据，并且还知道隔离规则实际上允许此数据可见。

The second bullet has the important caveat that "it is also known that the isolation rules in effect allow this data to be visible." This means that it cannot be assumed that an UPDATE that happened on another database connection will yet be visible here locally; in many cases, it will not. This is why if one wishes to use [expire()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.expire" \o "sqlalchemy.orm.session.Session.expire) or [refresh()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.refresh" \o "sqlalchemy.orm.session.Session.refresh) in order to view data between ongoing transactions, an understanding of the isolation behavior in effect is essential.

第二个项目符号有一个重要的注意事项，“也已知隔离规则实际上允许这些数据可见。” 这意味着不能假定在另一个数据库连接上发生的UPDATE在本地将不可见; 在许多情况下，它不会。 这就是为什么如果希望使用expire()或refresh()来查看正在进行的事务之间的数据，那么对实际的隔离行为的理解至关重要。

**See also**

[Session.expire()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.expire" \o "sqlalchemy.orm.session.Session.expire)

[Session.expire\_all()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.expire_all" \o "sqlalchemy.orm.session.Session.expire_all)

[Session.refresh()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.refresh" \o "sqlalchemy.orm.session.Session.refresh)

[isolation](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-isolation) - glossary explanation of isolation which includes links to Wikipedia.

隔离 - 词汇解释隔离，其中包括维基百科的链接。

[The SQLAlchemy Session In-Depth](http://techspot.zzzeek.org/2012/11/14/pycon-canada-the-sqlalchemy-session-in-depth/) - a video + slides with an in-depth discussion of the object lifecycle including the role of data expiration.

SQLAlchemy Session In-Depth - 视频+幻灯片，深入讨论对象生命周期，包括数据到期的角色。

## 5.3 Cascades

Mappers support the concept of configurable cascade behavior on [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) constructs. This refers to how operations performed on a "parent" object relative to a particular [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) should be propagated to items referred to by that relationship (e.g. "child" objects), and is affected by the [relationship.cascade](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.cascade" \o "sqlalchemy.orm.relationship)option.

Mappers支持关于()结构的可配置级联行为的概念。 这是指对相对于特定会话的“父”对象执行的操作应如何传播到由该关系引用的项目(例如“子”对象)中，并受到关系对象的影响。

The default behavior of cascade is limited to cascades of the so-called [save-update](http://docs.sqlalchemy.org/en/rel_1_1/orm/cascades.html" \l "cascade-save-update) and [merge](http://docs.sqlalchemy.org/en/rel_1_1/orm/cascades.html" \l "cascade-merge) settings. The typical "alternative" setting for cascade is to add the [delete](http://docs.sqlalchemy.org/en/rel_1_1/orm/cascades.html" \l "cascade-delete)and [delete-orphan](http://docs.sqlalchemy.org/en/rel_1_1/orm/cascades.html" \l "cascade-delete-orphan) options; these settings are appropriate for related objects which only exist as long as they are attached to their parent, and are otherwise deleted.

Cascade behavior is configured using the [cascade](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.cascade" \o "sqlalchemy.orm.relationship) option on [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship):

级联的默认行为仅限于所谓的保存更新和合并设置的级联。 级联的典型“替代”设置是添加deleteand delete-orphan选项; 这些设置适用于只存在于相关对象的相关对象，只要它们附加到其父级，并以其他方式删除。

使用关系()上的级联选项配置级联行为：

**class** **Order**(Base):

\_\_tablename\_\_ = 'order'

items = relationship("Item", cascade="all, delete-orphan")

customer = relationship("User", cascade="save-update")

To set cascades on a backref, the same flag can be used with the [backref()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.backref" \o "sqlalchemy.orm.backref) function, which ultimately feeds its arguments back into [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship):

要在backref上设置级联，相同的标志可以与backref()函数一起使用，后者最终将其参数反馈给relationship()：

**class** **Item**(Base):

\_\_tablename\_\_ = 'item'

order = relationship("Order",

backref=backref("items", cascade="all, delete-orphan")

)

**The Origins of Cascade/**级联的起源

SQLAlchemy's notion of cascading behavior on relationships, as well as the options to configure them, are primarily derived from the similar feature in the Hibernate ORM; Hibernate refers to "cascade" in a few places such as in [Example: Parent/Child](https://docs.jboss.org/hibernate/orm/3.3/reference/en-US/html/example-parentchild.html). If cascades are confusing, we'll refer to their conclusion, stating "The sections we have just covered can be a bit confusing. However, in practice, it all works out nicely."

SQLAlchemy对关系上的级联行为以及配置它们的选项的概念主要源自Hibernate ORM中的类似功能; Hibernate在几个地方引用“cascade”，例如：Parent / Child。 如果级联混乱，我们会参考他们的结论，指出“我们刚才所介绍的部分可能会有点混乱，但实际上，这一切都很好。”

The default value of [cascade](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.cascade" \o "sqlalchemy.orm.relationship) is save-update, merge. The typical alternative setting for this parameter is either all or more commonly all, delete-orphan. The all symbol is a synonym for save-update, merge, refresh-expire, expunge, delete, and using it in conjunction with delete-orphan indicates that the child object should follow along with its parent in all cases, and be deleted once it is no longer associated with that parent.

[cascade](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.cascade" \o "sqlalchemy.orm.relationship)的默认值为save-update， merge。 此参数的典型替代设置是all或更多通常是all，delete-orphan。 全部符号是save-update， merge， refresh-expire，expunge，delete以及与delete-orphan一起使用的同义词，表示子对象在所有情况下都应与其父项一起遵循，并且一旦被删除 不再与该父母关联。

The list of available values which can be specified for the [cascade](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.cascade" \o "sqlalchemy.orm.relationship) parameter are described in the following subsections.

可以为级联参数指定的可用值的列表在以下小节中进行了说明。

5.3.1 save-update

save-update cascade indicates that when an object is placed into a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) via [Session.add()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.add" \o "sqlalchemy.orm.session.Session.add), all the objects associated with it via this [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)should also be added to that same [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session). Suppose we have an object user1 with two related objects address1, address2:

save-update cascade指示当通过Session.add()将对象放入Session时，通过该关系()关联的所有对象也应该添加到同一个Session中。 假设我们有一个对象user1，其中有两个相关对象address1，address2：

**>>>** user1 = User()

**>>>** address1, address2 = Address(), Address()

**>>>** user1.addresses = [address1, address2]

If we add user1 to a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session), it will also add address1, address2 implicitly:

如果我们将一个user1添加到一个Session中，那么它也会隐含地添加address1，address2：

**>>>** sess = Session()

**>>>** sess.add(user1)

**>>>** address1 **in** sessTrue

save-update cascade also affects attribute operations for objects that are already present in a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session). If we add a third object, address3 to the user1.addresses collection, it becomes part of the state of that [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session):

保存更新级联还会影响会话中已存在的对象的属性操作。 如果我们向user1.addresses集合添加了第三个对象address3，那么它将成为该Session的状态的一部分：

**>>>** address3 = Address()

**>>>** user1.append(address3)

**>>>** address3 **in** sess

**>>> True**

save-update has the possibly surprising behavior which is that persistent objects which were *removed* from a collection or in some cases a scalar attribute may also be pulled into the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) of a parent object; this is so that the flush process may handle that related object appropriately. This case can usually only arise if an object is removed from one [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) and added to another:

保存更新具有可能令人惊讶的行为，即从集合中删除的持久对象或在某些情况下，标量属性也可以被拉入父对象的会话; 这是为了使冲洗过程可以适当地处理该相关对象。 这种情况通常只有在从一个会话中删除对象并添加到另一个会话时才会出现：

**>>>** user1 = sess1.query(User).filter\_by(id=1).first()

**>>>** address1 = user1.addresses[0]

**>>>** sess1.close() *# user1, address1 no longer associated with sess1*

**>>>** user1.addresses.remove(address1) *# address1 no longer associated with user1*

**>>>** sess2 = Session()

**>>>** sess2.add(user1) *# ... but it still gets added to the new session,*

**>>>** address1 **in** sess2 *# because it's still "pending" for flush*

True

The save-update cascade is on by default, and is typically taken for granted; it simplifies code by allowing a single call to [Session.add()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.add" \o "sqlalchemy.orm.session.Session.add) to register an entire structure of objects within that [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) at once. While it can be disabled, there is usually not a need to do so.

保存更新级联默认情况下启用，通常被认为是理所当然的; 它通过允许单次调用Session.add()来一次注册该会话中的对象的整个结构来简化代码。 虽然它可以被禁用，但通常不需要这样做。

One case where save-update cascade does sometimes get in the way is in that it takes place in both directions for bi-directional relationships, e.g. backrefs, meaning that the association of a child object with a particular parent can have the effect of the parent object being implicitly associated with that child object's [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session); this pattern, as well as how to modify its behavior using the [cascade\_backrefs](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.cascade_backrefs" \o "sqlalchemy.orm.relationship) flag, is discussed in the section [Controlling Cascade on Backrefs](http://docs.sqlalchemy.org/en/rel_1_1/orm/cascades.html" \l "backref-cascade).

保存更新级联有时会妨碍的一种情况是它在两个方向上发生双向关系，例如。 backrefs，意味着子对象与特定父对象的关联可以使父对象与该子对象的Session隐含关联的效果; 此模式以及如何使用cascade\_backrefs标志修改其行为，将在“控制Backref中的级联”一节中讨论。

5.3.2 delete

The delete cascade indicates that when a "parent" object is marked for deletion, its related "child" objects should also be marked for deletion. If for example we we have a relationship User.addresses with delete cascade configured:

删除级联表示当“父”对象被标记为删除时，其相关的“子”对象也应标记为删除。 如果我们有一个关系User.addresses与删除级联配置：

**class** **User**(Base):

*# ...*

addresses = relationship("Address", cascade="save-update, merge, delete")

If using the above mapping, we have a User object and two related Address objects:

如果使用上述映射，我们有一个User对象和两个相关的Address对象：

**>>>** user1 = sess.query(User).filter\_by(id=1).first()

**>>>** address1, address2 = user1.addresses

If we mark user1 for deletion, after the flush operation proceeds, address1 and address2 will also be deleted:

如果我们将user1标记为删除，则在刷新操作继续之后，address1和address2也将被删除：

>>> sess.delete(user1)

>>> sess.commit()

DELETE FROM address WHERE address.id = ?

((1,), (2,))

DELETE FROM user WHERE user.id = ?

(1,)

COMMIT

Alternatively, if our User.addresses relationship does *not* have delete cascade, SQLAlchemy's default behavior is to instead de-associate address1 and address2 from user1 by setting their foreign key reference to NULL. Using a mapping as follows:

或者，如果我们的User.addresses关系没有删除级联，那么SQLAlchemy的默认行为是通过将其外键引用设置为NULL而将与user1的address1和address2关联。 使用映射如下：

**class** **User**(Base):

*# ...*

addresses = relationship("Address")

Upon deletion of a parent User object, the rows in address are not deleted, but are instead de-associated:

删除父用户对象后，地址中的行不会被删除，而是删除关联：

>>> sess.delete(user1)

>>> sess.commit()

UPDATE address SET user\_id=? WHERE address.id = ?

(None, 1)

UPDATE address SET user\_id=? WHERE address.id = ?

(None, 2)

DELETE FROM user WHERE user.id = ?

(1,)

COMMIT

delete cascade is more often than not used in conjunction with [delete-orphan](http://docs.sqlalchemy.org/en/rel_1_1/orm/cascades.html" \l "cascade-delete-orphan) cascade, which will emit a DELETE for the related row if the "child" object is deassociated from the parent. The combination of delete and delete-orphan cascade covers both situations where SQLAlchemy has to decide between setting a foreign key column to NULL versus deleting the row entirely.

删除级联通常不会与删除孤立级联结合使用，如果“子”对象与父级脱离关联，那么它将为相关行发出DELETE。 删除和删除孤立级联的组合包括SQLAlchemy必须在将外键列设置为NULL而完全删除行之间决定的情况。

**ORM-level "delete" cascade vs. FOREIGN KEY level "ON DELETE" cascade**

The behavior of SQLAlchemy's "delete" cascade has a lot of overlap with the ON DELETE CASCADE feature of a database foreign key, as well as with that of the ON DELETE SETNULL foreign key setting when "delete" cascade is not specified. Database level "ON DELETE" cascades are specific to the "FOREIGN KEY" construct of the relational database; SQLAlchemy allows configuration of these schema-level constructs at the [DDL](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-ddl) level using options on [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint) which are described at [ON UPDATE and ON DELETE](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "on-update-on-delete).

SQLAlchemy的“删除”级联的行为与数据库外键的ON DELETE CASCADE功能以及未指定“删除”级联时的ON DELETE SETNULL外键设置的行为有很大的重叠。 数据库级别“ON DELETE”级联特定于关系数据库的“FOREIGN KEY”结构; SQLAlchemy允许使用在ON UPDATE和ON DELETE中描述的ForeignKeyConstraint上的选项在DDL级别配置这些模式级结构。

It is important to note the differences between the ORM and the relational database's notion of "cascade" as well as how they integrate:

重要的是要注意ORM和关系数据库的“级联”概念之间的差异以及它们如何整合：

A database level ON DELETE cascade is configured effectively on the ****many-to-one**** side of the relationship; that is, we configure it relative to the FOREIGN KEY constraint that is the "many" side of a relationship. At the ORM level, ****this direction is reversed****. SQLAlchemy handles the deletion of "child" objects relative to a "parent" from the "parent" side, which means that delete and delete-orphan cascade are configured on the ****one-to-many**** side.

数据库级别ON DELETE级联在关系的多对一方面有效配置; 也就是说，我们配置它相对于FOREIGN KEY约束是一个关系的“很多”一边。 在ORM级别，这个方向是相反的。 SQLAlchemy从“父”方面处理相对于“父”的“子”对象的删除，这意味着在一对多方面配置了删除和删除孤立级联。

Database level foreign keys with no ON DELETE setting are often used to ****prevent**** a parent row from being removed, as it would necessarily leave an unhandled related row present. If this behavior is desired in a one-to-many relationship, SQLAlchemy's default behavior of setting a foreign key to NULL can be caught in one of two ways:

数据库级别外键通常用于禁止删除父行，因为它必然会留下未处理的相关行。 如果在一对多关系中需要此行为，SQLAlchemy将外键设置为NULL的默认行为可以通过以下两种方式之一捕获：

* + The easiest and most common is just to set the foreign-key-holding column to NOT NULL at the database schema level. An attempt by SQLAlchemy to set the column to NULL will fail with a simple NOT NULL constraint exception.最简单和最常见的仅仅是在数据库架构级别将外键保持列设置为NOT NULL。 SQLAlchemy尝试将列设置为NULL将失败，并带有简单的NOT NULL约束异常。
  + The other, more special case way is to set the [passive\_deletes](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.passive_deletes" \o "sqlalchemy.orm.relationship) flag to the string "all". This has the effect of entirely disabling SQLAlchemy's behavior of setting the foreign key column to NULL, and a DELETE will be emitted for the parent row without any affect on the child row, even if the child row is present in memory. This may be desirable in the case when database-level foreign key triggers, either special ON DELETE settings or otherwise, need to be activated in all cases when a parent row is deleted.另一种更特殊的情况是将passive\_deletes标志设置为字符串“all”。 这具有完全禁用SQLAlchemy将外键列设置为NULL的行为的效果，并且即使子行存在于内存中，对父行也不会对子行发生任何影响，将删除DELETE。 在删除父行时，在所有情况下需要激活数据库级外键触发(特殊ON DELETE设置或其他方式)的情况下，这可能是理想的。

Database level ON DELETE cascade is ****vastly more efficient**** than that of SQLAlchemy. The database can chain a series of cascade operations across many relationships at once; e.g. if row A is deleted, all the related rows in table B can be deleted, and all the C rows related to each of those B rows, and on and on, all within the scope of a single DELETE statement. SQLAlchemy on the other hand, in order to support the cascading delete operation fully, has to individually load each related collection in order to target all rows that then may have further related collections. That is, SQLAlchemy isn't sophisticated enough to emit a DELETE for all those related rows at once within this context.

数据库级别ON DELETE级联比SQLAlchemy更有效率。该数据库可以一次跨多个关系链接一系列级联操作;例如如果行A被删除，则表B中的所有相关行可以被删除，并且与每个B行相关的所有C行以及on和on都在单个DELETE语句的范围内。 SQLAlchemy另一方面，为了充分支持级联删除操作，必须单独加载每个相关的集合，以便定位所有可能具有进一步相关集合的行。也就是说，SQLAlchemy不够复杂，无法在此上下文中同时针对所有这些相关行发出DELETE。

SQLAlchemy doesn't ****need**** to be this sophisticated, as we instead provide smooth integration with the database's own ON DELETE functionality, by using the [passive\_deletes](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.passive_deletes" \o "sqlalchemy.orm.relationship) option in conjunction with properly configured foreign key constraints. Under this behavior, SQLAlchemy only emits DELETE for those rows that are already locally present in the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session); for any collections that are unloaded, it leaves them to the database to handle, rather than emitting a SELECT for them. The section [Using Passive Deletes](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "passive-deletes) provides an example of this use.

SQLAlchemy不需要这么复杂，因为我们通过与被正确配置的外键约束结合使用[passive\_deletes](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.passive_deletes" \o "sqlalchemy.orm.relationship)选项来提供与数据库自己的ON DELETE功能的平滑集成。在这种情况下，SQLAlchemy只会为[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)中本地存在的那些行发出DELETE;对于卸载的任何集合，它们将它们留给数据库来处理，而不是为它们发出SELECT。 “使用被动删除”部分提供了此用法的示例。

While database-level ON DELETE functionality works only on the "many" side of a relationship, SQLAlchemy's "delete" cascade has ****limited**** ability to operate in the *reverse*direction as well, meaning it can be configured on the "many" side to delete an object on the "one" side when the reference on the "many" side is deleted. However this can easily result in constraint violations if there are other objects referring to this "one" side from the "many", so it typically is only useful when a relationship is in fact a "one to one". The [single\_parent](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.single_parent" \o "sqlalchemy.orm.relationship) flag should be used to establish an in-Python assertion for this case.

虽然数据库级ON DELETE功能仅在关系的“许多”方面起作用，但SQLAlchemy的“删除”级联也具有限制在逆向操作的能力，这意味着可以在“许多”一边配置删除对象在“多”一侧的引用被删除时，在“一”一侧。然而，如果有其他对象从“许多”引用这个“一个”方面，这很容易导致约束违规，所以通常只有当一个关系实际上是“一对一”时才有用。应该使用[single\_parent](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.single_parent" \o "sqlalchemy.orm.relationship)标志在这种情况下建立一个Python的断言。

When using a [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) that also includes a many-to-many table using the [secondary](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.secondary" \o "sqlalchemy.orm.relationship) option, SQLAlchemy's delete cascade handles the rows in this many-to-many table automatically. Just like, as described in [Deleting Rows from the Many to Many Table](http://docs.sqlalchemy.org/en/rel_1_1/orm/basic_relationships.html" \l "relationships-many-to-many-deletion), the addition or removal of an object from a many-to-many collection results in the INSERT or DELETE of a row in the many-to-many table, the delete cascade, when activated as the result of a parent object delete operation, will DELETE not just the row in the "child" table but also in the many-to-many table.

当使用还包括使用[secondary](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.secondary" \o "sqlalchemy.orm.relationship)选项的多对多表的[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)时，SQLAlchemy的删除级联将自动处理这个多对多表中的行。 就像“从多个到多个表删除行”中所述，从多对多集合中添加或删除对象导致多对多表中的行的INSERT或DELETE，delete级联当作为父对象删除操作的结果而被激活时，将不仅将删除“子”表中的行，而且还将在多对多表中删除。

5.3.3 delete-orphan

delete-orphan cascade adds behavior to the delete cascade, such that a child object will be marked for deletion when it is de-associated from the parent, not just when the parent is marked for deletion. This is a common feature when dealing with a related object that is "owned" by its parent, with a NOT NULL foreign key, so that removal of the item from the parent collection results in its deletion.

delete-orphan级联会将delete级联的行为添加到删除级别中，从而当父对象被删除时，子对象将被从父项取消关联时被标记为删除。 当处理由其父级“拥有”的相关对象，使用NOT NULL外键时，这是一个常见的功能，因此从父集合中删除该项导致其删除。

delete-orphan cascade implies that each child object can only have one parent at a time, so is configured in the vast majority of cases on a one-to-many relationship. Setting it on a many-to-one or many-to-many relationship is more awkward; for this use case, SQLAlchemy requires that the [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) be configured with the [single\_parent](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.single_parent" \o "sqlalchemy.orm.relationship) argument, establishes Python-side validation that ensures the object is associated with only one parent at a time.

删除孤立级联意味着每个子对象一次只能有一个父项，所以在绝大多数情况下，一对多关系配置。 以多对一或多对多的关系来设置它更为尴尬; 对于这种用例，SQLAlchemy要求将relationship()配置为single\_parent参数，建立Python侧验证，确保对象一次只能与一个父对象相关联。

5.3.4 merge

merge cascade indicates that the [Session.merge()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.merge" \o "sqlalchemy.orm.session.Session.merge) operation should be propagated from a parent that's the subject of the [Session.merge()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.merge" \o "sqlalchemy.orm.session.Session.merge) call down to referred objects. This cascade is also on by default.

merge cascade表示Session.merge()操作应该从作为Session.merge()调用的对象的父级传播给引用对象。 这个级联默认情况下也是这样。

5.3.5 refresh-expire

refresh-expire is an uncommon option, indicating that the [Session.expire()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.expire" \o "sqlalchemy.orm.session.Session.expire) operation should be propagated from a parent down to referred objects. When using [Session.refresh()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.refresh" \o "sqlalchemy.orm.session.Session.refresh), the referred objects are expired only, but not actually refreshed.

refresh-expire是一个不常见的选项，指示Session.expire()操作应该从父级传播到引用对象。 当使用Session.refresh()时，引用的对象只能过期，但实际上没有刷新。

5.3.6 expunge

expunge cascade indicates that when the parent object is removed from the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) using [Session.expunge()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.expunge" \o "sqlalchemy.orm.session.Session.expunge), the operation should be propagated down to referred objects.

expunge级联表示当使用Session.expunge()从会话中删除父对象时，该操作应该传播到引用对象。

5.3.7 Controlling Cascade on Backrefs

The [save-update](http://docs.sqlalchemy.org/en/rel_1_1/orm/cascades.html" \l "cascade-save-update) cascade by default takes place on attribute change events emitted from backrefs. This is probably a confusing statement more easily described through demonstration; it means that, given a mapping such as this:

默认情况下，保存更新级联发生在从backrefs发出的属性更改事件上。 这可能是通过示范更容易描述的令人困惑的声明; 这意味着，给定一个这样的映射：

mapper(Order, order\_table, properties={

'items' : relationship(Item, backref='order')})

If an Order is already in the session, and is assigned to the order attribute of an Item, the backref appends the Item to the items collection of that Order, resulting in the save-update cascade taking place:

如果一个订单已经在会话中，并被分配给一个项目的订单属性，则backref将该Item追加到该订单的items集合，导致save-update级联发生：

**>>>** o1 = Order()

**>>>** session.add(o1)

**>>>** o1 **in** sessionTrue

**>>>** i1 = Item()

**>>>** i1.order = o1

**>>>** i1 **in** o1.items

True

**>>>** i1 **in** session

True

This behavior can be disabled using the [cascade\_backrefs](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.cascade_backrefs" \o "sqlalchemy.orm.relationship) flag:

mapper(Order, order\_table, properties={

'items' : relationship(Item, backref='order',

cascade\_backrefs=**False**)})

So above, the assignment of i1.order = o1 will append i1 to the items collection of o1, but will not add i1 to the session. You can, of course, [add()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.add" \o "sqlalchemy.orm.session.Session.add) i1 to the session at a later point. This option may be helpful for situations where an object needs to be kept out of a session until it's construction is completed, but still needs to be given associations to objects which are already persistent in the target session.

所以上面这样，i1.order = o1的赋值会将i1追加到o1的项目集合中，但不会将i1添加到会话中。 当然，您可以在稍后的一段时间里将add()添加到会话中。 此选项可能有助于在需要将对象保留在会话之外，直到其构建完成为止，但仍需要给予与目标会话中已持久存在的对象相关联的情况。

# **5.4 Transactions and Connection Management**

# **5.4.1 Managing Transactions**

A newly constructed [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) may be said to be in the "begin" state. In this state, the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) has not established any connection or transactional state with any of the [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) objects that may be associated with it.

新建的会话可能被称为处于“开始”状态。在这种状态下，会话没有与可能与之相关联的任何Engine对象建立任何连接或事务状态。

The [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) then receives requests to operate upon a database connection. Typically, this means it is called upon to execute SQL statements using a particular[Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine), which may be via [Session.query()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.query" \o "sqlalchemy.orm.session.Session.query), [Session.execute()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.execute" \o "sqlalchemy.orm.session.Session.execute), or within a flush operation of pending data, which occurs when such state exists and [Session.commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.commit" \o "sqlalchemy.orm.session.Session.commit) or [Session.flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.flush" \o "sqlalchemy.orm.session.Session.flush) is called.

然后，会话接收到对数据库连接进行操作的请求。通常，这意味着它被要求使用特定引擎执行SQL语句，特定引擎可以通过Session.query()，Session.execute()或在等待数据的刷新操作中发生，当这种状态存在时，会发生该事件。 commit()或Session.flush()被调用。

As these requests are received, each new [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) encountered is associated with an ongoing transactional state maintained by the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session). When the first [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) is operated upon, the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) can be said to have left the "begin" state and entered "transactional" state. For each [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) encountered, a [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) is associated with it, which is acquired via the [Engine.contextual\_connect()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.contextual_connect" \o "sqlalchemy.engine.Engine.contextual_connect) method. If a [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) was directly associated with the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) (see [Joining a Session into an External Transaction (such as for test suites)](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_transaction.html" \l "session-external-transaction) for an example of this), it is added to the transactional state directly.

当接收到这些请求时，遇到的每个新引擎与由Session维护的正在进行的事务状态相关联。当第一台引擎运行时，会话可以说已经离开“开始”状态并进入“事务”状态。对于遇到的每个引擎，连接与它相关联，它通过Engine.contextual\_connect()方法获取。如果连接与会话直接关联(请参阅将会话连接到外部事务(例如测试套件)作为示例)，它将直接添加到事务状态。

For each [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection), the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) also maintains a [Transaction](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction" \o "sqlalchemy.engine.Transaction) object, which is acquired by calling [Connection.begin()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.begin" \o "sqlalchemy.engine.Connection.begin) on each [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection), or if the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) object has been established using the flag twophase=True, a [TwoPhaseTransaction](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.TwoPhaseTransaction" \o "sqlalchemy.engine.TwoPhaseTransaction) object acquired via [Connection.begin\_twophase()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.begin_twophase" \o "sqlalchemy.engine.Connection.begin_twophase). These transactions are all committed or rolled back corresponding to the invocation of the [Session.commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.commit" \o "sqlalchemy.orm.session.Session.commit) and [Session.rollback()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.rollback" \o "sqlalchemy.orm.session.Session.rollback) methods. A commit operation will also call the [TwoPhaseTransaction.prepare()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.TwoPhaseTransaction.prepare" \o "sqlalchemy.engine.TwoPhaseTransaction.prepare) method on all transactions if applicable.

对于每个[Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection)，会话还维护一个[Transaction](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction" \o "sqlalchemy.engine.Transaction)对象，它通过在每个Connection上调用[Connection.begin()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.begin" \o "sqlalchemy.engine.Connection.begin)获取，或者如果使用标志twophase=True建立了Session对象，则通过Connection.begin\_twophase获取的TwoPhaseTransaction对象)。这些事务都是根据Session.commit()和Session.rollback()方法的调用而提交或回滚的。如果适用，提交操作也将调用所有事务的TwoPhaseTransaction.prepare()方法。

When the transactional state is completed after a rollback or commit, the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) [releases](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-releases) all [Transaction](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction" \o "sqlalchemy.engine.Transaction) and [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) resources, and goes back to the "begin" state, which will again invoke new [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) and [Transaction](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction" \o "sqlalchemy.engine.Transaction) objects as new requests to emit SQL statements are received.

当事务状态在回滚或提交后完成时，会话释放所有事务和连接资源，并返回到“开始”状态，这将再次调用新的连接和事务对象，因为接收到新的发出SQL语句的请求。

The example below illustrates this lifecycle:

下面的例子说明了这个生命周期：

engine = create\_engine("...")

Session = sessionmaker(bind=engine)

*# new session. no connections are in use.*

session = Session()**try**:

*# first query. a Connection is acquired*

*# from the Engine, and a Transaction*

*# started.*

item1 = session.query(Item).get(1)

*# second query. the same Connection/Transaction*

*# are used.*

item2 = session.query(Item).get(2)

*# pending changes are created.*

item1.foo = 'bar'

item2.bar = 'foo'

*# commit. The pending changes above*

*# are flushed via flush(), the Transaction*

*# is committed, the Connection object closed*

*# and discarded, the underlying DBAPI connection*

*# returned to the connection pool.*

session.commit()

**except**:

*# on rollback, the same closure of state*

*# as that of commit proceeds.*

session.rollback()

**raise**

**finally**:

*# close the Session. This will expunge any remaining*

*# objects as well as reset any existing SessionTransaction*

*# state. Neither of these steps are usually essential.*

*# However, if the commit() or rollback() itself experienced*

*# an unanticipated internal failure (such as due to a mis-behaved*

*# user-defined event handler), .close() will ensure that*

*# invalid state is removed.*

session.close()

### Using SAVEPOINT

SAVEPOINT transactions, if supported by the underlying engine, may be delineated using the [begin\_nested()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.begin_nested" \o "sqlalchemy.orm.session.Session.begin_nested) method:

如果底层引擎支持SAVEPOINT事务，可以使用begin\_nested()方法进行描述：

Session = sessionmaker()

session = Session()

session.add(u1)

session.add(u2)

session.begin\_nested() *# establish a savepoint*

session.add(u3)

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

session.commit() *# commits u1 and u2*

[begin\_nested()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.begin_nested" \o "sqlalchemy.orm.session.Session.begin_nested) may be called any number of times, which will issue a new SAVEPOINT with a unique identifier for each call. For each [begin\_nested()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.begin_nested" \o "sqlalchemy.orm.session.Session.begin_nested) call, a corresponding [rollback()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.rollback" \o "sqlalchemy.orm.session.Session.rollback) or [commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.commit" \o "sqlalchemy.orm.session.Session.commit) must be issued. (But note that if the return value is used as a context manager, i.e. in a with-statement, then this rollback/commit is issued by the context manager upon exiting the context, and so should not be added explicitly.)

begin\_nested()可以被调用任意次数，这将为每个调用发出一个新的SAVEPOINT，并具有一个唯一的标识符。对于每个begin\_nested()调用，必须发出相应的rollback()或commit()。 (但是请注意，如果返回值用作上下文管理器，即在with-statement中，则在退出上下文时由上下文管理器发出该回滚/提交，因此不应显式添加。)

When [begin\_nested()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.begin_nested" \o "sqlalchemy.orm.session.Session.begin_nested) is called, a [flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.flush" \o "sqlalchemy.orm.session.Session.flush) is unconditionally issued (regardless of the autoflush setting). This is so that when a [rollback()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.rollback" \o "sqlalchemy.orm.session.Session.rollback) occurs, the full state of the session is expired, thus causing all subsequent attribute/instance access to reference the full state of the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) right before [begin\_nested()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.begin_nested" \o "sqlalchemy.orm.session.Session.begin_nested) was called.

当调用begin\_nested()时，无条件地发出flush()(无论自动冲洗设置如何)。这样当回滚()发生时，会话的完整状态已过期，从而导致所有后续的属性/实例访问在begin\_nested()被调用之前引用会话的完整状态。

[begin\_nested()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.begin_nested" \o "sqlalchemy.orm.session.Session.begin_nested), in the same manner as the less often used [begin()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.begin" \o "sqlalchemy.orm.session.Session.begin) method, returns a [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction) object which works as a context manager. It can be succinctly used around individual record inserts in order to catch things like unique constraint exceptions:

begin\_nested()以与较少使用的begin()方法相同的方式返回一个作为上下文管理器的SessionTransaction对象。它可以简洁地用于个别记录插入，以捕获像唯一约束异常的东西：

**for** record **in** records:

**try**:

**with** session.begin\_nested():

session.merge(record)

**except**:

print("Skipped record *%s*" % record)session.commit()

### Autocommit Mode

The example of [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) transaction lifecycle illustrated at the start of [Managing Transactions](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_transaction.html" \l "unitofwork-transaction) applies to a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) configured in the default mode of autocommit=False. Constructing a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) with autocommit=True produces a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) placed into "autocommit" mode, where each SQL statement invoked by a [Session.query()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.query" \o "sqlalchemy.orm.session.Session.query) or [Session.execute()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.execute" \o "sqlalchemy.orm.session.Session.execute) occurs using a new connection from the connection pool, discarding it after results have been iterated. The [Session.flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.flush" \o "sqlalchemy.orm.session.Session.flush) operation still occurs within the scope of a single transaction, though this transaction is closed out after the [Session.flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.flush" \o "sqlalchemy.orm.session.Session.flush) operation completes.

管理事务开始时说明的会话事务生命周期的示例适用于以默认模式autocommit = False配置的会话。 使用autocommit = True构造会话将产生一个放置在“自动提交”模式中的会话，其中由Session.query()或Session.execute()调用的每个SQL语句都使用连接池中的新连接进行，结果具有 被迭代。 Session.flush()操作仍然发生在单个事务的范围内，尽管在Session.flush()操作完成后此事务被关闭。

**Warning**

"autocommit" mode should ****not be considered for general use****. If used, it should always be combined with the usage of [Session.begin()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.begin" \o "sqlalchemy.orm.session.Session.begin) and [Session.commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.commit" \o "sqlalchemy.orm.session.Session.commit), to ensure a transaction demarcation.

“自动提交”模式不应被视为一般用途。如果使用，它应该总是与Session.begin()和Session.commit()的使用相结合，以确保事务划分。

Executing queries outside of a demarcated transaction is a legacy mode of usage, and can in some cases lead to concurrent connection checkouts.

在划分的事务之外执行查询是传统的使用模式，并且在某些情况下可能导致并发连接检出。

In the absence of a demarcated transaction, the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) cannot make appropriate decisions as to when autoflush should occur nor when auto-expiration should occur, so these features should be disabled with autoflush=False, expire\_on\_commit=False.

在没有划分的事务的情况下，会话无法做出适当的决定，关于自动冲洗应该发生什么时候，也不应该发生自动过期，因此应该使用autoflush = False，expire\_on\_commit = False禁用这些功能。

Modern usage of "autocommit" is for framework integrations that need to control specifically when the "begin" state occurs. A session which is configured withautocommit=True may be placed into the "begin" state using the [Session.begin()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.begin" \o "sqlalchemy.orm.session.Session.begin) method. After the cycle completes upon [Session.commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.commit" \o "sqlalchemy.orm.session.Session.commit) or [Session.rollback()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.rollback" \o "sqlalchemy.orm.session.Session.rollback), connection and transaction resources are [released](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-released) and the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) goes back into "autocommit" mode, until [Session.begin()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.begin" \o "sqlalchemy.orm.session.Session.begin) is called again:

“自动提交”的现代用法是用于在“开始”状态发生时需要特别控制的框架集成。可以使用Session.begin()方法将配置为withautocommit = True的会话置于“开始”状态。在Session.commit()或Session.rollback()上的循环完成后，连接和事务资源被释放，会话返回到“自动提交”模式，直到再次调用Session.begin()：

Session = sessionmaker(bind=engine, autocommit=**True**)

session = Session()

session.begin()

**try**:

item1 = session.query(Item).get(1)

item2 = session.query(Item).get(2)

item1.foo = 'bar'

item2.bar = 'foo'

session.commit()

**except**:

session.rollback()

**raise**

The [Session.begin()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.begin" \o "sqlalchemy.orm.session.Session.begin) method also returns a transactional token which is compatible with the Python 2.6 with statement:

Session.begin()方法还返回一个与Python 2.6语句兼容的事务性令牌：

Session = sessionmaker(bind=engine, autocommit=**True**)

session = Session()

**with** session.begin():

item1 = session.query(Item).get(1)

item2 = session.query(Item).get(2)

item1.foo = 'bar'

item2.bar = 'foo'

#### Using Subtransactions with Autocommit

A subtransaction indicates usage of the [Session.begin()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.begin" \o "sqlalchemy.orm.session.Session.begin) method in conjunction with the subtransactions=True flag. This produces a non-transactional, delimiting construct that allows nesting of calls to [begin()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.begin" \o "sqlalchemy.orm.session.Session.begin) and [commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.commit" \o "sqlalchemy.orm.session.Session.commit). Its purpose is to allow the construction of code that can function within a transaction both independently of any external code that starts a transaction, as well as within a block that has already demarcated a transaction.

子事务表示使用Session.begin()方法结合subtransactions = True标志。 这产生一个非事务性的分隔结构，允许将调用嵌套到begin()和commit()。 其目的是允许构建可以在事务中运行的代码，独立于启动事务的任何外部代码，以及已经划定事务的块中。

subtransactions=True is generally only useful in conjunction with autocommit, and is equivalent to the pattern described at [Nesting of Transaction Blocks](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "connections-nested-transactions), where any number of functions can call [Connection.begin()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.begin" \o "sqlalchemy.engine.Connection.begin) and [Transaction.commit()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction.commit" \o "sqlalchemy.engine.Transaction.commit) as though they are the initiator of the transaction, but in fact may be participating in an already ongoing transaction:

subtransactions = True通常仅适用于自动提交，并且等效于嵌套事务块中描述的模式，其中任何数量的函数可以调用Connection.begin()和Transaction.commit()，就好像它们是 交易，但实际上可能正在参与已经持续的交易：

*# method\_a starts a transaction and calls method\_b*

**def** method\_a(session):

session.begin(subtransactions=**True**)

**try**:

method\_b(session)

session.commit() *# transaction is committed here*

**except**:

session.rollback() *# rolls back the transaction*

**raise**

*# method\_b also starts a transaction, but when# called from method\_a participates in the ongoing# transaction.*

**def** method\_b(session):

session.begin(subtransactions=**True**)

**try**:

session.add(SomeObject('bat', 'lala'))

session.commit() *# transaction is not committed yet*

**except**:

session.rollback() *# rolls back the transaction, in this case*

*# the one that was initiated in method\_a().*

**raise**

*# create a Session and call method\_a*

session = Session(autocommit=**True**)

method\_a(session)

session.close()

Subtransactions are used by the [Session.flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.flush" \o "sqlalchemy.orm.session.Session.flush) process to ensure that the flush operation takes place within a transaction, regardless of autocommit. When autocommit is disabled, it is still useful in that it forces the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) into a "pending rollback" state, as a failed flush cannot be resumed in mid-operation, where the end user still maintains the "scope" of the transaction overall.

Session.flush()进程使用子事务来确保在事务中进行刷新操作，而不管自动提交。 当自动提交被禁用时，它仍然有用，因为它强制会话进入“待执行回滚”状态，因为在中期操作中无法恢复失败的刷新，最终用户仍然维护整个事务的“范围”。

### Enabling Two-Phase Commit

For backends which support two-phase operaration (currently MySQL and PostgreSQL), the session can be instructed to use two-phase commit semantics. This will coordinate the committing of transactions across databases so that the transaction is either committed or rolled back in all databases. You can also [prepare()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.prepare" \o "sqlalchemy.orm.session.Session.prepare) the session for interacting with transactions not managed by SQLAlchemy. To use two phase transactions set the flag twophase=True on the session:

engine1 = create\_engine('postgresql://db1')

engine2 = create\_engine('postgresql://db2')

Session = sessionmaker(twophase=**True**)

*# bind User operations to engine 1, Account operations to engine 2*

Session.configure(binds={User:engine1, Account:engine2})

session = Session()

*# .... work with accounts and users*

*# commit. session will issue a flush to all DBs, and a prepare step to all DBs,# before committing both transactions*

session.commit()

### Setting Transaction Isolation Levels

[Isolation](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-isolation) refers to the behavior of the transaction at the database level in relation to other transactions occurring concurrently. There are four well-known modes of isolation, and typically the Python DBAPI allows these to be set on a per-connection basis, either through explicit APIs or via database-specific calls.

SQLAlchemy's dialects support settable isolation modes on a per-[Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) or per-[Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) basis, using flags at both the [create\_engine()](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "sqlalchemy.create_engine" \o "sqlalchemy.create_engine) level as well as at the [Connection.execution\_options()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.execution_options" \o "sqlalchemy.engine.Connection.execution_options) level.

When using the ORM [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session), it acts as a *facade* for engines and connections, but does not expose transaction isolation directly. So in order to affect transaction isolation level, we need to act upon the [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) or [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) as appropriate.

**See also**

[create\_engine.isolation\_level](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "sqlalchemy.create_engine.params.isolation_level" \o "sqlalchemy.create_engine)

[SQLite Transaction Isolation](http://docs.sqlalchemy.org/en/rel_1_1/dialects/sqlite.html" \l "sqlite-isolation-level)

[PostgreSQL Isolation Level](http://docs.sqlalchemy.org/en/rel_1_1/dialects/postgresql.html" \l "postgresql-isolation-level)

[MySQL Isolation Level](http://docs.sqlalchemy.org/en/rel_1_1/dialects/mysql.html" \l "mysql-isolation-level)

#### Setting Isolation Engine-Wide

To set up a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) or [sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker) with a specific isolation level globally, use the [create\_engine.isolation\_level](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "sqlalchemy.create_engine.params.isolation_level" \o "sqlalchemy.create_engine) parameter:

要设置具有全局特定隔离级别的会话或会话生成器，请使用create\_engine.isolation\_level参数：

**from** **sqlalchemy** **import** create\_engine

**from** **sqlalchemy.orm** **import** sessionmaker

eng = create\_engine(

"postgresql://scott:tiger@localhost/test",

isolation\_level='REPEATABLE\_READ')

maker = sessionmaker(bind=eng)

session = maker()

#### Setting Isolation for Individual Sessions

When we make a new [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session), either using the constructor directly or when we call upon the callable produced by a [sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker), we can pass the bindargument directly, overriding the pre-existing bind. We can combine this with the [Engine.execution\_options()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.execution_options" \o "sqlalchemy.engine.Engine.execution_options) method in order to produce a copy of the original [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) that will add this option:

当我们创建一个新的[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)时，直接使用构造函数，或者当我们调用[sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker)生成的callable时，我们可以直接传递bind参数，覆盖预先存在的bind。 我们可以将其与Engine.execution\_options()方法相结合，以生成将添加此选项的原始引擎的副本：

session = maker(

bind=engine.execution\_options(isolation\_level='SERIALIZABLE'))

For the case where the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) or [sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker) is configured with multiple "binds", we can either re-specify the binds argument fully, or if we want to only replace specific binds, we can use the [Session.bind\_mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.bind_mapper" \o "sqlalchemy.orm.session.Session.bind_mapper) or [Session.bind\_table()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.bind_table" \o "sqlalchemy.orm.session.Session.bind_table) methods:

对于Session或SessionMaker配置了多个“binding”的情况，我们可以重新指定binding参数，或者我们只想替换特定的绑定，我们可以使用Session.bind\_mapper()或Session.bind\_table () 方法：

session = maker()

session.bind\_mapper(

User, user\_engine.execution\_options(isolation\_level='SERIALIZABLE'))

We can also use the individual transaction method that follows.

#### Setting Isolation for Individual Transactions

A key caveat regarding isolation level is that the setting cannot be safely modified on a [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) where a transaction has already started. Databases cannot change the isolation level of a transaction in progress, and some DBAPIs and SQLAlchemy dialects have inconsistent behaviors in this area. Some may implicitly emit a ROLLBACK and some may implicitly emit a COMMIT, others may ignore the setting until the next transaction. Therefore SQLAlchemy emits a warning if this option is set when a transaction is already in play. The [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) object does not provide for us a [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) for use in a transaction where the transaction is not already begun. So here, we need to pass execution options to the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) at the start of a transaction by passing [Session.connection.execution\_options](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.connection.params.execution_options" \o "sqlalchemy.orm.session.Session.connection)provided by the [Session.connection()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.connection" \o "sqlalchemy.orm.session.Session.connection) method:

关于隔离级别的关键警告是无法在事务已启动的[Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) 上安全地修改该设置。 数据库无法更改正在进行的事务的隔离级别，并且某些DBAPI和SQLAlchemy方言在此区域中具有不一致的行为。 有些可能会隐式地发出一个ROLLBACK，有些可能会隐式地发出一个COMMIT，其他的可能忽略该设置，直到下一个事务。 因此，SQLAlchemy会在事务已经播放时设置此选项时发出警告。 Session对象不为我们提供一个在事务尚未开始的事务中使用的[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) 。 所以在这里，我们需要通过传递Session.connection()方法提供的Session.connection.execution\_options在事务开始时将执行选项传递给Session：

**from** **sqlalchemy.orm** **import** Session

sess = Session(bind=engine)

sess.connection(execution\_options={'isolation\_level': 'SERIALIZABLE'})

*# work with session*

*# commit transaction. the connection is released# and reverted to its previous isolation level.*

sess.commit()

Above, we first produce a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) using either the constructor or a [sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker). Then we explicitly set up the start of a transaction by calling upon [Session.connection()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.connection" \o "sqlalchemy.orm.session.Session.connection), which provides for execution options that will be passed to the connection before the transaction is begun. If we are working with a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) that has multiple binds or some other custom scheme for [Session.get\_bind()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.get_bind" \o "sqlalchemy.orm.session.Session.get_bind), we can pass additional arguments to [Session.connection()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.connection" \o "sqlalchemy.orm.session.Session.connection) in order to affect how the bind is procured:

以上，我们首先使用构造函数或sessionmaker生成一个Session。 然后，我们通过调用Session.connection()来明确设置事务的开始，该事件提供了在事务开始之前传递给连接的执行选项。 如果我们正在使用具有多个绑定或Session.get\_bind()的其他自定义方案的会话，那么我们可以将其他参数传递给Session.connection()，以便影响绑定的采用方式：

sess = my\_sesssionmaker()

*# set up a transaction for the bind associated with# the User mapper*

sess.connection(

mapper=User,

execution\_options={'isolation\_level': 'SERIALIZABLE'})

*# work with session*

*# commit transaction. the connection is released# and reverted to its previous isolation level.*

sess.commit()

The [Session.connection.execution\_options](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.connection.params.execution_options" \o "sqlalchemy.orm.session.Session.connection) argument is only accepted on the ****first**** call to [Session.connection()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.connection" \o "sqlalchemy.orm.session.Session.connection) for a particular bind within a transaction. If a transaction is already begun on the target connection, a warning is emitted:

Session.connection.execution\_options参数仅在事务中对于特定绑定的第一次调用Session.connection()时才被接受。 如果目标连接上的事务已经开始，则会发出警告：

**>>>** session = Session(eng)

**>>>** session.execute("select 1")

<sqlalchemy.engine.result.ResultProxy object at 0x1017a6c50>

**>>>** session.connection(execution\_options={'isolation\_level': 'SERIALIZABLE'})

sqlalchemy/orm/session.py:310: SAWarning: Connection is already establishedfor the given bind; execution\_options ignored

*New in version 0.9.9:*Added the [Session.connection.execution\_options](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.connection.params.execution_options" \o "sqlalchemy.orm.session.Session.connection) parameter to [Session.connection()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.connection" \o "sqlalchemy.orm.session.Session.connection).

### Tracking Transaction State with Events

See the section [Transaction Events](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_events.html" \l "session-transaction-events) for an overview of the available event hooks for session transaction state changes.

有关会话事务状态更改的可用事件钩子的概述，请参阅事务事件部分。

5.4.2 Joining a Session into an External Transaction (such as for test suites)

If a [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) is being used which is already in a transactional state (i.e. has a [Transaction](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction" \o "sqlalchemy.engine.Transaction) established), a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) can be made to participate within that transaction by just binding the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) to that [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection). The usual rationale for this is a test suite that allows ORM code to work freely with a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session), including the ability to call [Session.commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.commit" \o "sqlalchemy.orm.session.Session.commit), where afterwards the entire database interaction is rolled back:

如果正在使用已经处于事务状态的连接(即建立了事务)，则可以通过将会话绑定到该连接来使会话参与该事务。 通常的理由是允许ORM代码与会话自由工作的测试套件，包括调用Session.commit()的功能，此后，整个数据库交互将回滚：

**from** **sqlalchemy.orm** **import** sessionmaker

**from** **sqlalchemy** **import** create\_engine

**from** **unittest** **import** TestCase

*# global application scope. create Session class, engine*

Session = sessionmaker()

engine = create\_engine('postgresql://...')

**class** **SomeTest**(TestCase):

**def** setUp(self):

*# connect to the database*

self.connection = engine.connect()

*# begin a non-ORM transaction*

self.trans = self.connection.begin()

*# bind an individual Session to the connection*

self.session = Session(bind=self.connection)

**def** test\_something(self):

*# use the session in tests.*

self.session.add(Foo())

self.session.commit()

**def** tearDown(self):

self.session.close()

*# rollback - everything that happened with the*

*# Session above (including calls to commit())*

*# is rolled back.*

self.trans.rollback()

*# return connection to the Engine*

self.connection.close()

Above, we issue [Session.commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.commit" \o "sqlalchemy.orm.session.Session.commit) as well as [Transaction.rollback()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction.rollback" \o "sqlalchemy.engine.Transaction.rollback). This is an example of where we take advantage of the [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) object's ability to maintain *subtransactions*, or nested begin/commit-or-rollback pairs where only the outermost begin/commit pair actually commits the transaction, or if the outermost block rolls back, everything is rolled back.

以上，我们发行Session.commit()以及Transaction.rollback()。 这是一个例子，我们利用Connection对象维护子事务的能力，或嵌套的begin / commit-or-rollback对，其中只有最外层的begin/commit对实际提交事务，或者如果最外层的块回滚， 所有都回滚了

**Supporting Tests with Rollbacks**

The above recipe works well for any kind of database enabled test, except for a test that needs to actually invoke [Session.rollback()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.rollback" \o "sqlalchemy.orm.session.Session.rollback) within the scope of the test itself. The above recipe can be expanded, such that the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) always runs all operations within the scope of a SAVEPOINT, which is established at the start of each transaction, so that tests can also rollback the "transaction" as well while still remaining in the scope of a larger "transaction" that's never committed, using two extra events:

上述方法适用于任何类型的数据库启用测试，除了需要在测试范围内实际调用Session.rollback()的测试外。 上述配方可以扩展，以便会话始终运行在每个事务开始时建立的SAVEPOINT范围内的所有操作，以便测试也可以回滚“事务”，同时仍保留在范围内 一个更大的“事务”，从未承诺，使用两个额外的事件：

**from** **sqlalchemy** **import** event

**class** **SomeTest**(TestCase):

**def** setUp(self):

*# connect to the database*

self.connection = engine.connect()

*# begin a non-ORM transaction*

self.trans = connection.begin()

*# bind an individual Session to the connection*

self.session = Session(bind=self.connection)

*# start the session in a SAVEPOINT...*

self.session.begin\_nested()

*# then each time that SAVEPOINT ends, reopen it*

**@event**.listens\_for(self.session, "after\_transaction\_end")

**def** restart\_savepoint(session, transaction):

**if** transaction.nested **and** **not** transaction.\_parent.nested:

*# ensure that state is expired the way*

*# session.commit() at the top level normally does*

*# (optional step)*

session.expire\_all()

session.begin\_nested()

*# ... the tearDown() method stays the same*

# **5.5 Additional Persistence Techniques**

# **5.5.1 Embedding SQL Insert/Update Expressions into a Flush**

This feature allows the value of a database column to be set to a SQL expression instead of a literal value. It's especially useful for atomic updates, calling stored procedures, etc. All you do is assign an expression to an attribute:

此功能允许将数据库列的值设置为SQL表达式而不是字面值。 它对于原子更新，调用存储过程等特别有用。您所做的就是将一个表达式赋给一个属性：

**class** **SomeClass**(object):

**Pass**

mapper(SomeClass, some\_table)

someobject = session.query(SomeClass).get(5)

*# set 'value' attribute to a SQL expression adding one*

someobject.value = some\_table.c.value + 1

*# issues "UPDATE some\_table SET value=value+1"*

session.commit()

This technique works both for INSERT and UPDATE statements. After the flush/commit operation, the value attribute on someobject above is expired, so that when next accessed the newly generated value will be loaded from the database.

此技术适用于INSERT和UPDATE语句。 在刷新/提交操作之后，上面某个对象上的value属性已过期，以便下次访问时，新生成的值将从数据库加载。

5.5.2 Using SQL Expressions with Sessions

SQL expressions and strings can be executed via the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) within its transactional context. This is most easily accomplished using the [execute()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.execute" \o "sqlalchemy.orm.session.Session.execute) method, which returns a [ResultProxy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy" \o "sqlalchemy.engine.ResultProxy) in the same manner as an [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) or [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection):

SQL表达式和字符串可以通过事务上下文中的Session执行。 这是使用execute()方法最容易实现的，它以与Engine或Connection相同的方式返回ResultProxy：

Session = sessionmaker(bind=engine)

session = Session()

*# execute a string statement*

result = session.execute("select \* from table where id=:id", {'id':7})

*# execute a SQL expression construct*

result = session.execute(select([mytable]).where(mytable.c.id==7))

The current [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) held by the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) is accessible using the [connection()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.connection" \o "sqlalchemy.orm.session.Session.connection) method:

Session持有的当前连接可以使用connection()方法访问：

connection = session.connection()

The examples above deal with a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) that's bound to a single [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) or [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection). To execute statements using a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) which is bound either to multiple engines, or none at all (i.e. relies upon bound metadata), both [execute()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.execute" \o "sqlalchemy.orm.session.Session.execute) and [connection()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.connection" \o "sqlalchemy.orm.session.Session.connection) accept a mapper keyword argument, which is passed a mapped class or [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) instance, which is used to locate the proper context for the desired engine:

上面的示例处理绑定到单个引擎或连接的会话。 要使用绑定到多个引擎的Session执行语句，或者根本没有绑定元数据(即依赖于绑定的元数据)，execute()和connection()都接受映射类或Mapper实例的mapper关键字参数， 用于定位所需引擎的正确上下文：

Session = sessionmaker()session = Session()

*# need to specify mapper or class when executing*

result = session.execute("select \* from table where id=:id", {'id':7}, mapper=MyMappedClass)

result = session.execute(select([mytable], mytable.c.id==7), mapper=MyMappedClass)

connection = session.connection(MyMappedClass)

### 5.5.3 Forcing NULL on a column with a default

The ORM considers any attribute that was never set on an object as a "default" case; the attribute will be omitted from the INSERT statement:

ORM将从未在对象上设置的任何属性视为“默认”情况; 该属性将从INSERT语句中省略：

**class** **MyObject**(Base):

\_\_tablename\_\_ = 'my\_table'

id = Column(Integer, primary\_key=**True**)

data = Column(String(50), nullable=**True**)

obj = MyObject(id=1)

session.add(obj)

session.commit() *# INSERT with the 'data' column omitted; the database*

*# itself will persist this as the NULL value*

Omitting a column from the INSERT means that the column will have the NULL value set, *unless* the column has a default set up, in which case the default value will be persisted. This holds true both from a pure SQL perspective with server-side defaults, as well as the behavior of SQLAlchemy's insert behavior with both client-side and server-side defaults:

从INSERT中省略列表示列将具有NULL值集，除非该列具有默认设置，在这种情况下，默认值将被保留。 这同样适用于纯SQL视图与服务器端默认值，以及SQLAlchemy的插入行为与客户端和服务器端默认值的行为：

**class** **MyObject**(Base):

\_\_tablename\_\_ = 'my\_table'

id = Column(Integer, primary\_key=**True**)

data = Column(String(50), nullable=**True**, server\_default="default")

obj = MyObject(id=1)

session.add(obj)

session.commit() *# INSERT with the 'data' column omitted; the database*

*# itself will persist this as the value 'default'*

However, in the ORM, even if one assigns the Python value None explicitly to the object, this is treated the ****same**** as though the value were never assigned:

但是，在ORM中，即使将Python值无明确地分配给对象，这一点与该值永远不分配一样：

**class** **MyObject**(Base):

\_\_tablename\_\_ = 'my\_table'

id = Column(Integer, primary\_key=**True**)

data = Column(String(50), nullable=**True**, server\_default="default")

obj = MyObject(id=1, data=**None**)

session.add(obj)

session.commit() *# INSERT with the 'data' column explicitly set to None;*

*# the ORM still omits it from the statement and the*

*# database will still persist this as the value 'default'*

The above operation will persist into the data column the server default value of "default" and not SQL NULL, even though None was passed; this is a long-standing behavior of the ORM that many applications hold as an assumption.

上述操作将持续到数据列服务器默认值“default”而不是SQL NULL，即使没有通过; 这是许多应用程序作为假设的ORM的长期行为。

So what if we want to actually put NULL into this column, even though the column has a default value? There are two approaches. One is that on a per-instance level, we assign the attribute using the [null](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.null" \o "sqlalchemy.sql.expression.null) SQL construct:

那么如果我们想把NULL放在这个列中呢，即使这个列有一个默认值呢？ 有两种方法。 一个是在每个实例级别上，我们使用null SQL构造来分配属性：

**from** **sqlalchemy** **import** null

obj = MyObject(id=1, data=null())

session.add(obj)

session.commit() *# INSERT with the 'data' column explicitly set as null();*

*# the ORM uses this directly, bypassing all client-*

*# and server-side defaults, and the database will*

*# persist this as the NULL value*

The [null](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.null" \o "sqlalchemy.sql.expression.null) SQL construct always translates into the SQL NULL value being directly present in the target INSERT statement.

空SQL构造总是转换为目标INSERT语句中直接存在的SQL NULL值。

If we'd like to be able to use the Python value None and have this also be persisted as NULL despite the presence of column defaults, we can configure this for the ORM using a Core-level modifier [TypeEngine.evaluates\_none()](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.evaluates_none" \o "sqlalchemy.types.TypeEngine.evaluates_none), which indicates a type where the ORM should treat the value None the same as any other value and pass it through, rather than omitting it as a "missing" value:

如果我们希望能够使用Python值None，并且尽管存在列默认值也可以将其持久化为NULL，我们可以使用核心级别的修饰符TypeEngine.evaluates\_none()将其配置为ORM，这表示 一种类型，其中ORM应该将值与任何其他值相同，并将其传递，而不是将其忽略为“缺少”值：

**class** **MyObject**(Base):

\_\_tablename\_\_ = 'my\_table'

id = Column(Integer, primary\_key=**True**)

data = Column(

String(50).evaluates\_none(), *# indicate that None should always be passed*

nullable=**True**, server\_default="default")

obj = MyObject(id=1, data=**None**)

session.add(obj)

session.commit() *# INSERT with the 'data' column explicitly set to None;*

*# the ORM uses this directly, bypassing all client-*

*# and server-side defaults, and the database will*

*# persist this as the NULL value*

**Evaluating None**

The [TypeEngine.evaluates\_none()](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.evaluates_none" \o "sqlalchemy.types.TypeEngine.evaluates_none) modifier is primarily intended to signal a type where the Python value "None" is significant, the primary example being a JSON type which may want to persist the JSON null value rather than SQL NULL. We are slightly repurposing it here in order to signal to the ORM that we'd like None to be passed into the type whenever present, even though no special type-level behaviors are assigned to it.

TypeEngine.evaluates\_none()修饰符主要用于表示Python值“None”的重要类型，主要示例是可能要保留JSON空值而不是SQL NULL的JSON类型。 我们在这里稍微重新使用它，以便向ORM发出信号，即使没有特定的类型级别的行为被分配给我们，我们也希望None被传递到类型。

*New in version 1.1:*added the [TypeEngine.evaluates\_none()](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.evaluates_none" \o "sqlalchemy.types.TypeEngine.evaluates_none) method in order to indicate that a "None" value should be treated as significant.

版本1.1中的新功能：添加了TypeEngine.evaluates\_none()方法，以指示“无”值应被视为重要。

5.5.4 Partitioning Strategies

### Simple Vertical Partitioning

Vertical partitioning places different kinds of objects, or different tables, across multiple databases:

垂直分区可以跨多个数据库放置不同类型的对象或不同的表：

engine1 = create\_engine('postgresql://db1')

engine2 = create\_engine('postgresql://db2')

Session = sessionmaker(twophase=**True**)

*# bind User operations to engine 1, Account operations to engine 2*

Session.configure(binds={User:engine1, Account:engine2})

session = Session()

Above, operations against either class will make usage of the [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) linked to that class. Upon a flush operation, similar rules take place to ensure each class is written to the right database.

以上，针对任一类的操作将使用与该类相关联的引擎。 在刷新操作时，会发生类似的规则，以确保每个类都写入正确的数据库。

The transactions among the multiple databases can optionally be coordinated via two phase commit, if the underlying backend supports it. See [Enabling Two-Phase Commit](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_transaction.html" \l "session-twophase) for an example.

如果底层后端支持多个数据库之间的事务可以通过两阶段提交进行协调。 有关示例，请参阅启用两阶段提交。

### Custom Vertical Partitioning

More comprehensive rule-based class-level partitioning can be built by overriding the [Session.get\_bind()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.get_bind" \o "sqlalchemy.orm.session.Session.get_bind) method. Below we illustrate a custom [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) which delivers the following rules:

可以通过覆盖Session.get\_bind()方法来构建更全面的基于规则的类级别分区。 下面我们说明一个自定义会话，它提供以下规则：

1. Flush operations are delivered to the engine named master.
2. Operations on objects that subclass MyOtherClass all occur on the other engine.
3. Read operations for all other classes occur on a random choice of the slave1 or slave2 database.

engines = {

'master':create\_engine("sqlite:///master.db"),

'other':create\_engine("sqlite:///other.db"),

'slave1':create\_engine("sqlite:///slave1.db"),

'slave2':create\_engine("sqlite:///slave2.db"),}

**from** **sqlalchemy.orm** **import** Session, sessionmaker

**import** **random**

**class** **RoutingSession**(Session):

**def** get\_bind(self, mapper=**None**, clause=**None**):

**if** mapper **and** issubclass(mapper.class\_, MyOtherClass):

**return** engines['other']

**elif** self.\_flushing:

**return** engines['master']

**else**:

**return** engines[

random.choice(['slave1','slave2'])

]

The above [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) class is plugged in using the class\_ argument to [sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker):

Session = sessionmaker(class\_=RoutingSession)

This approach can be combined with multiple [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) objects, using an approach such as that of using the declarative \_\_abstract\_\_ keyword, described at [\_\_abstract\_\_](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "declarative-abstract).

这种方法可以使用诸如使用\_\_abstract\_\_中描述的声明性\_\_abstract\_\_关键字的方法与多个MetaData对象相结合。

### Horizontal Partitioning

Horizontal partitioning partitions the rows of a single table (or a set of tables) across multiple databases.

水平分区跨多个数据库划分单个表(或一组表)的行。

See the "sharding" example: [Horizontal Sharding](http://docs.sqlalchemy.org/en/rel_1_1/orm/examples.html" \l "examples-sharding).

请参见“分片”示例：“水平分片”。

5.5.5 Bulk Operations

**Note**

Bulk Operations mode is a new series of operations made available on the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) object for the purpose of invoking INSERT and UPDATE statements with greatly reduced Python overhead, at the expense of much less functionality, automation, and error checking. As of SQLAlchemy 1.0, these features should be considered as "beta", and additionally are intended for advanced users.

批量操作模式是在Session对象上提供的一系列新操作，用于以大大减少Python开销的方式调用INSERT和UPDATE语句，牺牲了更少的功能，自动化和错误检查。 从SQLAlchemy 1.0开始，这些功能应被视为“beta”，另外还适用于高级用户。

*New in version 1.0.0.*

Bulk operations on the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) include [Session.bulk\_save\_objects()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.bulk_save_objects" \o "sqlalchemy.orm.session.Session.bulk_save_objects), [Session.bulk\_insert\_mappings()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.bulk_insert_mappings" \o "sqlalchemy.orm.session.Session.bulk_insert_mappings), and [Session.bulk\_update\_mappings()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.bulk_update_mappings" \o "sqlalchemy.orm.session.Session.bulk_update_mappings). The purpose of these methods is to directly expose internal elements of the unit of work system, such that facilities for emitting INSERT and UPDATE statements given dictionaries or object states can be utilized alone, bypassing the normal unit of work mechanics of state, relationship and attribute management. The advantages to this approach is strictly one of reduced Python overhead:

[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)上的批量操作包括[Session.bulk\_save\_objects()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.bulk_save_objects" \o "sqlalchemy.orm.session.Session.bulk_save_objects), [Session.bulk\_insert\_mappings()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.bulk_insert_mappings" \o "sqlalchemy.orm.session.Session.bulk_insert_mappings), 和[Session.bulk\_update\_mappings()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.bulk_update_mappings" \o "sqlalchemy.orm.session.Session.bulk_update_mappings)。 这些方法的目的是直接暴露工作系统单元的内部元素，以便单独使用给出字典或对象状态的INSERT和UPDATE语句的设施，绕过状态，关系和属性的正常工作机制管理。 这种方法的优点是减少Python开销之一：

* The flush() process, including the survey of all objects, their state, their cascade status, the status of all objects associated with them via [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship), and the topological sort of all operations to be performed is completely bypassed. This reduces a great amount of Python overhead.

flush()进程，包括对所有对象的调查，状态，级联状态，通过[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)与它们相关联的所有对象的状态以及要执行的所有操作的拓扑排序完全绕过。这减少了大量的Python开销。

* The objects as given have no defined relationship to the target [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session), even when the operation is complete, meaning there's no overhead in attaching them or managing their state in terms of the identity map or session.给定的对象与目标会话没有定义的关系，即使操作完成，这意味着在身份映射或会话方面附加他们或管理他们的状态没有开销。
* The [Session.bulk\_insert\_mappings()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.bulk_insert_mappings" \o "sqlalchemy.orm.session.Session.bulk_insert_mappings) and [Session.bulk\_update\_mappings()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.bulk_update_mappings" \o "sqlalchemy.orm.session.Session.bulk_update_mappings) methods accept lists of plain Python dictionaries, not objects; this further reduces a large amount of overhead associated with instantiating mapped objects and assigning state to them, which normally is also subject to expensive tracking of history on a per-attribute basis.Session.bulk\_insert\_mappings()和Session.bulk\_update\_mappings()方法接受纯Python字典列表，而不是对象;这进一步减少了与实例化映射对象相关联的大量开销，并向其分配状态，这通常也在每个属性的基础上经历对历史的昂贵跟踪。
* The set of objects passed to all bulk methods are processed in the order they are received. In the case of [Session.bulk\_save\_objects()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.bulk_save_objects" \o "sqlalchemy.orm.session.Session.bulk_save_objects), when objects of different types are passed, the INSERT and UPDATE statements are necessarily broken up into per-type groups. In order to reduce the number of batch INSERT or UPDATE statements passed to the DBAPI, ensure that the incoming list of objects are grouped by type.传递给所有批量方法的对象集合按照接收的顺序进行处理。在Session.bulk\_save\_objects()的情况下，当传递不同类型的对象时，INSERT和UPDATE语句必须分解成每个类型的组。为了减少传递给DBAPI的批处理INSERT或UPDATE语句的数量，请确保对象的传入列表按类型进行分组。
* The process of fetching primary keys after an INSERT also is disabled by default. When performed correctly, INSERT statements can now more readily be batched by the unit of work process into executemany() blocks, which perform vastly better than individual statement invocations.默认情况下禁用INSERT之后获取主键的过程。当正确执行时，INSERT语句现在可以更容易地由工作流程单元分批到executemany()块中，执行比单独的语句调用更好。
* UPDATE statements can similarly be tailored such that all attributes are subject to the SET clase unconditionally, again making it much more likely that executemany() blocks can be used.UPDATE语句可以类似地进行定制，使得所有属性都无条件地受到SET契约的约束，从而更有可能使用executemany()块。

The performance behavior of the bulk routines should be studied using the [Performance](http://docs.sqlalchemy.org/en/rel_1_1/orm/examples.html" \l "examples-performance) example suite. This is a series of example scripts which illustrate Python call-counts across a variety of scenarios, including bulk insert and update scenarios.

应该使用Performance示例套件来研究批量例程的性能。 这是一系列示例脚本，用于说明各种场景中的Python调用计数，包括批量插入和更新方案。

**See also**

[Performance](http://docs.sqlalchemy.org/en/rel_1_1/orm/examples.html" \l "examples-performance) - includes detailed examples of bulk operations contrasted against traditional Core and ORM methods, including performance metrics.

性能 - 包括大量操作的详细示例，与传统的Core和ORM方法(包括性能指标)形成对照。

### Usage

The methods each work in the context of the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) object's transaction, like any other:

这些方法都可以在Session对象的事务的上下文中工作，就像其他任何一样：

s = Session()objects = [

User(name="u1"),

User(name="u2"),

User(name="u3")]s.bulk\_save\_objects(objects)

For [Session.bulk\_insert\_mappings()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.bulk_insert_mappings" \o "sqlalchemy.orm.session.Session.bulk_insert_mappings), and [Session.bulk\_update\_mappings()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.bulk_update_mappings" \o "sqlalchemy.orm.session.Session.bulk_update_mappings), dictionaries are passed:

对于Session.bulk\_insert\_mappings()和Session.bulk\_update\_mappings()，会传递字典：

s.bulk\_insert\_mappings(User,

[dict(name="u1"), dict(name="u2"), dict(name="u3")])

**See also**

[Session.bulk\_save\_objects()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.bulk_save_objects" \o "sqlalchemy.orm.session.Session.bulk_save_objects)

[Session.bulk\_insert\_mappings()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.bulk_insert_mappings" \o "sqlalchemy.orm.session.Session.bulk_insert_mappings)

[Session.bulk\_update\_mappings()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.bulk_update_mappings" \o "sqlalchemy.orm.session.Session.bulk_update_mappings)

### Comparison to Core Insert / Update Constructs

The bulk methods offer performance that under particular circumstances can be close to that of using the core [Insert](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert" \o "sqlalchemy.sql.expression.Insert) and [Update](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Update" \o "sqlalchemy.sql.expression.Update) constructs in an "executemany" context (for a description of "executemany", see [Executing Multiple Statements](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html" \l "execute-multiple) in the Core tutorial). In order to achieve this, the [Session.bulk\_insert\_mappings.return\_defaults](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.bulk_insert_mappings.params.return_defaults" \o "sqlalchemy.orm.session.Session.bulk_insert_mappings) flag should be disabled so that rows can be batched together. The example suite in [Performance](http://docs.sqlalchemy.org/en/rel_1_1/orm/examples.html" \l "examples-performance) should be carefully studied in order to gain familiarity with how fast bulk performance can be achieved.

大量方法提供了在特定情况下可以接近在“执行者”上下文中使用核心插入和更新构造的性能(有关“执行”的描述，请参阅在Core教程中执行多个语句)。 为了实现这一点，应该禁用Session.bulk\_insert\_mappings.return\_defaults标志，以便将行分批在一起。 应该仔细研究Performance中的示例套件，以便了解如何实现快速批量性能。

### ORM Compatibility

The bulk insert / update methods lose a significant amount of functionality versus traditional ORM use. The following is a listing of features that are ****not available**** when using these methods:

与传统ORM使用相比，批量插入/更新方法失去了大量的功能。 以下是使用以下方法时不可用的功能列表：

* persistence along [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) linkages沿着关系()链接的持久性
* sorting of rows within order of dependency; rows are inserted or updated directly in the order in which they are passed to the methods依赖顺序排列行; 行按照它们传递给方法的顺序直接插入或更新
* Session-management on the given objects, including attachment to the session, identity map management.对给定对象的会话管理，包括对会话的附加，身份映射管理。
* Functionality related to primary key mutation, ON UPDATE cascade与主键突变相关的功能，ON UPDATE级联
* SQL expression inserts / updates (e.g. [Embedding SQL Insert/Update Expressions into a Flush](http://docs.sqlalchemy.org/en/rel_1_1/orm/persistence_techniques.html" \l "flush-embedded-sql-expressions))SQL表达式插入/更新(例如将SQL插入/更新表达式嵌入到冲洗中)
* ORM events such as [MapperEvents.before\_insert()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents.before_insert" \o "sqlalchemy.orm.events.MapperEvents.before_insert), etc. The bulk session methods have no event support.MOM事件，如MapperEvents.before\_insert()等等。大容量会话方法没有事件支持。

Features that ****are available**** include:

* INSERTs and UPDATEs of mapped objects
* Version identifier support
* Multi-table mappings, such as joined-inheritance - however, an object to be inserted across multiple tables either needs to have primary key identifiers fully populated ahead of time, else the [Session.bulk\_save\_objects.return\_defaults](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.bulk_save_objects.params.return_defaults" \o "sqlalchemy.orm.session.Session.bulk_save_objects) flag must be used, which will greatly reduce the performance benefits

# 5.6 Contextual/Thread-local Sessions

Recall from the section [When do I construct a Session, when do I commit it, and when do I close it?](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_basics.html" \l "session-faq-whentocreate), the concept of "session scopes" was introduced, with an emphasis on web applications and the practice of linking the scope of a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) with that of a web request. Most modern web frameworks include integration tools so that the scope of the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) can be managed automatically, and these tools should be used as they are available.

回想一下，在什么时候构建一个Session，什么时候我提交它，什么时候关闭它，引入了“会话范围”的概念，重点是Web应用程序和链接的范围 与Web请求的会话。 大多数现代Web框架包括集成工具，以便可以自动管理会话的范围，并且应该使用这些工具。

SQLAlchemy includes its own helper object, which helps with the establishment of user-defined [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) scopes. It is also used by third-party integration systems to help construct their integration schemes.

SQLAlchemy包括其自己的帮助对象，这有助于建立用户定义的会话范围。 它也被第三方集成系统用于帮助构建其集成方案。

The object is the [scoped\_session](http://docs.sqlalchemy.org/en/rel_1_1/orm/contextual.html" \l "sqlalchemy.orm.scoping.scoped_session" \o "sqlalchemy.orm.scoping.scoped_session) object, and it represents a ****registry**** of [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) objects. If you're not familiar with the registry pattern, a good introduction can be found in [Patterns of Enterprise Architecture](http://martinfowler.com/eaaCatalog/registry.html).

该对象是scoped\_session对象，它代表一个Session对象的注册表。 如果您不熟悉注册表模式，可以在企业架构模式中找到一个很好的介绍。

**Note**

The [scoped\_session](http://docs.sqlalchemy.org/en/rel_1_1/orm/contextual.html" \l "sqlalchemy.orm.scoping.scoped_session" \o "sqlalchemy.orm.scoping.scoped_session) object is a very popular and useful object used by many SQLAlchemy applications. However, it is important to note that it presents ****only one approach**** to the issue of [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) management. If you're new to SQLAlchemy, and especially if the term "thread-local variable" seems strange to you, we recommend that if possible you familiarize first with an off-the-shelf integration system such as [Flask-SQLAlchemy](http://packages.python.org/Flask-SQLAlchemy/) or [zope.sqlalchemy](http://pypi.python.org/pypi/zope.sqlalchemy).

scoped\_session对象是许多SQLAlchemy应用程序使用的非常受欢迎和有用的对象。 但是，重要的是要注意，会议管理问题只提出一种方法。 如果您刚接触SQLAlchemy，并且特别是如果术语“线程局部变量”对您来说似乎很奇怪，我们建议您尽可能先熟悉现有的集成系统，例如Flask-SQLAlchemy或zope.sqlalchemy。

A [scoped\_session](http://docs.sqlalchemy.org/en/rel_1_1/orm/contextual.html" \l "sqlalchemy.orm.scoping.scoped_session" \o "sqlalchemy.orm.scoping.scoped_session) is constructed by calling it, passing it a ****factory**** which can create new [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) objects. A factory is just something that produces a new object when called, and in the case of [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session), the most common factory is the [sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker), introduced earlier in this section. Below we illustrate this usage:

scoped\_session是通过调用它来构造的，它传递一个可以创建新的Session对象的工厂。 一个工厂只是在调用时产生一个新的对象，而在Session的情况下，最常见的工厂就是本节前面介绍的sessionmaker。 下面我们来说明这个用法：

**>>> from** **sqlalchemy.orm** **import** scoped\_session

**>>> from** **sqlalchemy.orm** **import** sessionmaker

**>>>** session\_factory = sessionmaker(bind=some\_engine)

**>>>** Session = scoped\_session(session\_factory)

The [scoped\_session](http://docs.sqlalchemy.org/en/rel_1_1/orm/contextual.html" \l "sqlalchemy.orm.scoping.scoped_session" \o "sqlalchemy.orm.scoping.scoped_session) object we've created will now call upon the [sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker) when we "call" the registry:

当我们“调用”注册表时，我们创建的[scoped\_session](http://docs.sqlalchemy.org/en/rel_1_1/orm/contextual.html" \l "sqlalchemy.orm.scoping.scoped_session" \o "sqlalchemy.orm.scoping.scoped_session)对象现在将调用[sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker)：

**>>>** some\_session = Session()

Above, some\_session is an instance of [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session), which we can now use to talk to the database. This same [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) is also present within the [scoped\_session](http://docs.sqlalchemy.org/en/rel_1_1/orm/contextual.html" \l "sqlalchemy.orm.scoping.scoped_session" \o "sqlalchemy.orm.scoping.scoped_session)registry we've created. If we call upon the registry a second time, we get back the ****same**** [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session):

以上，some\_session是Session的一个实例，我们现在可以使用它来与数据库通信。 同样的Session也存在于我们创建的scoped\_sessionregistry中。 如果我们再次呼吁注册机构，我们会回到同一个会话：

**>>>** some\_other\_session = Session()

**>>>** some\_session **is** some\_other\_session

True

This pattern allows disparate sections of the application to call upon a global [scoped\_session](http://docs.sqlalchemy.org/en/rel_1_1/orm/contextual.html" \l "sqlalchemy.orm.scoping.scoped_session" \o "sqlalchemy.orm.scoping.scoped_session), so that all those areas may share the same session without the need to pass it explicitly. The [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) we've established in our registry will remain, until we explicitly tell our registry to dispose of it, by calling [scoped\_session.remove()](http://docs.sqlalchemy.org/en/rel_1_1/orm/contextual.html" \l "sqlalchemy.orm.scoping.scoped_session.remove" \o "sqlalchemy.orm.scoping.scoped_session.remove):

此模式允许应用程序的不同部分调用全局scoped\_session，以便所有这些区域可能共享相同的会话而不需要明确地传递。 我们在注册表中建立的[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)将保留，直到我们明确地告诉我们的注册表来处理它，通过调用scoped\_session.remove()

**>>>** Session.remove()

The [scoped\_session.remove()](http://docs.sqlalchemy.org/en/rel_1_1/orm/contextual.html" \l "sqlalchemy.orm.scoping.scoped_session.remove" \o "sqlalchemy.orm.scoping.scoped_session.remove) method first calls [Session.close()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.close" \o "sqlalchemy.orm.session.Session.close) on the current [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session), which has the effect of releasing any connection/transactional resources owned by the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) first, then discarding the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) itself. "Releasing" here means that connections are returned to their connection pool and any transactional state is rolled back, ultimately using the rollback() method of the underlying DBAPI connection.

scoped\_session.remove()方法首先在当前会话上调用Session.close()，该方法首先释放由Session拥有的任何连接/事务资源，然后丢弃会话本身。 这里的“释放”表示连接将返回到其连接池，并且任何事务状态都将回滚，最终使用底层DBAPI连接的rollback()方法。

At this point, the [scoped\_session](http://docs.sqlalchemy.org/en/rel_1_1/orm/contextual.html" \l "sqlalchemy.orm.scoping.scoped_session" \o "sqlalchemy.orm.scoping.scoped_session) object is "empty", and will create a ****new**** [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) when called again. As illustrated below, this is not the same [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) we had before:

此时，scoped\_session对象为“空”，并在再次调用时创建一个新的Session。 如下图所示，这与我们以前的会话不一样：

**>>>** new\_session = Session()

**>>>** new\_session **is** some\_session

False

The above series of steps illustrates the idea of the "registry" pattern in a nutshell. With that basic idea in hand, we can discuss some of the details of how this pattern proceeds.

上述一系列步骤简要说明了“注册表”模式的想法。 有了这个基本的想法，我们可以讨论一下这个模式如何进行的一些细节。

5.6.1 Implicit Method Access

The job of the [scoped\_session](http://docs.sqlalchemy.org/en/rel_1_1/orm/contextual.html" \l "sqlalchemy.orm.scoping.scoped_session" \o "sqlalchemy.orm.scoping.scoped_session) is simple; hold onto a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) for all who ask for it. As a means of producing more transparent access to this [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session), the [scoped\_session](http://docs.sqlalchemy.org/en/rel_1_1/orm/contextual.html" \l "sqlalchemy.orm.scoping.scoped_session" \o "sqlalchemy.orm.scoping.scoped_session) also includes ****proxy behavior****, meaning that the registry itself can be treated just like a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) directly; when methods are called on this object, they are ****proxied**** to the underlying [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) being maintained by the registry:

scoped\_session的工作很简单; 为所有要求的人举行会议。 作为生成更透明访问此会话的一种手段，scoped\_session还包括代理行为，这意味着注册表本身可以直接像Session一样对待; 当对这个对象调用方法时，它们被代理到由注册表维护的底层Session：

Session = scoped\_session(some\_factory)

*# equivalent to:## session = Session()# print(session.query(MyClass).all())#*

print(Session.query(MyClass).all())

The above code accomplishes the same task as that of acquiring the current [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) by calling upon the registry, then using that [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session).

上述代码通过调用注册表，然后使用该Session完成与获取当前会话相同的任务。

5.6.2 Thread-Local Scope

Users who are familiar with multithreaded programming will note that representing anything as a global variable is usually a bad idea, as it implies that the global object will be accessed by many threads concurrently. The [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) object is entirely designed to be used in a ****non-concurrent**** fashion, which in terms of multithreading means "only in one thread at a time". So our above example of [scoped\_session](http://docs.sqlalchemy.org/en/rel_1_1/orm/contextual.html" \l "sqlalchemy.orm.scoping.scoped_session" \o "sqlalchemy.orm.scoping.scoped_session) usage, where the same [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) object is maintained across multiple calls, suggests that some process needs to be in place such that multiple calls across many threads don't actually get a handle to the same session. We call this notion ****thread local storage****, which means, a special object is used that will maintain a distinct object per each application thread. Python provides this via the [threading.local()](http://docs.python.org/library/threading.html" \l "threading.local)construct. The [scoped\_session](http://docs.sqlalchemy.org/en/rel_1_1/orm/contextual.html" \l "sqlalchemy.orm.scoping.scoped_session" \o "sqlalchemy.orm.scoping.scoped_session) object by default uses this object as storage, so that a single [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) is maintained for all who call upon the [scoped\_session](http://docs.sqlalchemy.org/en/rel_1_1/orm/contextual.html" \l "sqlalchemy.orm.scoping.scoped_session" \o "sqlalchemy.orm.scoping.scoped_session)registry, but only within the scope of a single thread. Callers who call upon the registry in a different thread get a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) instance that is local to that other thread.

熟悉多线程编程的用户将会注意到，将任何东西代表全局变量通常是一个坏主意，因为它意味着全局对象将被许多线程并发访问。 Session对象完全被设计为以非并发方式使用，这在多线程方面意味着“一次只在一个线程中”。所以我们上面的scoped\_session用法的例子，在同一个Session对象在多个调用之间进行维护，这表明一些进程需要到位，这样多个线程的多个调用实际上并没有获得同一个会话的句柄。我们称这个概念线程本地存储，这意味着使用一个特殊对象，每个应用程序线程将维护一个不同的对象。 Python通过threading.local()构造提供了这一点。默认情况下，scoped\_session对象使用此对象作为存储，以便为调用scoped\_sessionregistry的所有人保留一个会话，但仅在单个线程的范围内。在不同线程中调用注册表的调用者获取该另一个线程本地的Session实例。

Using this technique, the [scoped\_session](http://docs.sqlalchemy.org/en/rel_1_1/orm/contextual.html" \l "sqlalchemy.orm.scoping.scoped_session" \o "sqlalchemy.orm.scoping.scoped_session) provides a quick and relatively simple (if one is familiar with thread-local storage) way of providing a single, global object in an application that is safe to be called upon from multiple threads.

使用这种技术，scoped\_session提供了一种快速而相对简单(如果您熟悉线程本地存储)方式，可以在多个线程安全地调用的应用程序中提供单个全局对象。

The [scoped\_session.remove()](http://docs.sqlalchemy.org/en/rel_1_1/orm/contextual.html" \l "sqlalchemy.orm.scoping.scoped_session.remove" \o "sqlalchemy.orm.scoping.scoped_session.remove) method, as always, removes the current [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) associated with the thread, if any. However, one advantage of thethreading.local() object is that if the application thread itself ends, the "storage" for that thread is also garbage collected. So it is in fact "safe" to use thread local scope with an application that spawns and tears down threads, without the need to call [scoped\_session.remove()](http://docs.sqlalchemy.org/en/rel_1_1/orm/contextual.html" \l "sqlalchemy.orm.scoping.scoped_session.remove" \o "sqlalchemy.orm.scoping.scoped_session.remove). However, the scope of transactions themselves, i.e. ending them via [Session.commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.commit" \o "sqlalchemy.orm.session.Session.commit) or [Session.rollback()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.rollback" \o "sqlalchemy.orm.session.Session.rollback), will usually still be something that must be explicitly arranged for at the appropriate time, unless the application actually ties the lifespan of a thread to the lifespan of a transaction.

与往常一样，scoped\_session.remove()方法会删除与线程关联的当前会话(如果有的话)。 但是，threading.local()对象的一个优点是，如果应用程序线程本身结束，该线程的“存储”也将被垃圾回收。 所以实际上是“安全的”使用线程本地范围与产生和撕下线程的应用程序，而不需要调用scoped\_session.remove()。 然而，交易本身的范围，即通过Session.commit()或Session.rollback()来结束它们，通常仍然是在适当的时候必须明确排列的东西，除非应用程序实际上关联线程的使用寿命 到一个交易的寿命。

5.6.3 Using Thread-Local Scope with Web Applications

As discussed in the section [When do I construct a Session, when do I commit it, and when do I close it?](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_basics.html" \l "session-faq-whentocreate), a web application is architected around the concept of a ****web request****, and integrating such an application with the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) usually implies that the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) will be associated with that request. As it turns out, most Python web frameworks, with notable exceptions such as the asynchronous frameworks Twisted and Tornado, use threads in a simple way, such that a particular web request is received, processed, and completed within the scope of a single *worker thread*. When the request ends, the worker thread is released to a pool of workers where it is available to handle another request.

在本节中讨论的部分我何时构建会话，何时提交会话，何时关闭它？Web应用程序围绕Web请求的概念进行架构，并将此类应用程序与会话集成通常意味着会话将与该请求相关联。事实证明，大多数Python Web框架(包括异步框架Twisted和Tornado)都有明显的异常，以简单的方式使用线程，以便在单个工作线程的范围内接收，处理和完成特定的Web请求。当请求结束时，工作线程被释放到可用于处理其他请求的工作池中。

This simple correspondence of web request and thread means that to associate a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) with a thread implies it is also associated with the web request running within that thread, and vice versa, provided that the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) is created only after the web request begins and torn down just before the web request ends. So it is a common practice to use [scoped\_session](http://docs.sqlalchemy.org/en/rel_1_1/orm/contextual.html" \l "sqlalchemy.orm.scoping.scoped_session" \o "sqlalchemy.orm.scoping.scoped_session) as a quick way to integrate the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) with a web application. The sequence diagram below illustrates this flow:

Web请求和线程的这种简单对应意味着将会话与线程相关联意味着它也与在该线程内运行的Web请求相关联，反之亦然，只要在Web请求开始和拆除之后创建会话就在Web请求结束之前。所以通常的做法是使用scoped\_session作为将会话与Web应用程序进行集成的快速方法。下面的顺序图说明了这个流程：

Web Server Web Framework SQLAlchemy ORM Code-------------- -------------- ------------------------------startup -> Web framework *# Session registry is established*

initializes Session = scoped\_session(sessionmaker())

incomingweb request -> web request -> *# The registry is \*optionally\**

starts *# called upon explicitly to create*

*# a Session local to the thread and/or request*

Session()

*# the Session registry can otherwise*

*# be used at any time, creating the*

*# request-local Session() if not present,*

*# or returning the existing one*

Session.query(MyClass) *# ...*

Session.add(some\_object) *# ...*

*# if data was modified, commit the*

*# transaction*

Session.commit()

web request ends -> *# the registry is instructed to*

*# remove the Session*

Session.remove()

sends output <-outgoing web <-response

Using the above flow, the process of integrating the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) with the web application has exactly two requirements:

使用上述流程，将会话与Web应用程序集成的过程有两个要求：

1. Create a single [scoped\_session](http://docs.sqlalchemy.org/en/rel_1_1/orm/contextual.html" \l "sqlalchemy.orm.scoping.scoped_session" \o "sqlalchemy.orm.scoping.scoped_session) registry when the web application first starts, ensuring that this object is accessible by the rest of the application.在Web应用程序首次启动时创建单个Scoped\_session注册表，确保该对象可以由应用程序的其余部分访问。
2. Ensure that [scoped\_session.remove()](http://docs.sqlalchemy.org/en/rel_1_1/orm/contextual.html" \l "sqlalchemy.orm.scoping.scoped_session.remove" \o "sqlalchemy.orm.scoping.scoped_session.remove) is called when the web request ends, usually by integrating with the web framework's event system to establish an "on request end" event..确保在Web请求结束时调用scoped\_session.remove()，通常通过与Web框架的事件系统集成来建立“请求结束”事件。

As noted earlier, the above pattern is ****just one potential way**** to integrate a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) with a web framework, one which in particular makes the significant assumption that the ****web framework associates web requests with application threads****. It is however ****strongly recommended that the integration tools provided with the web framework itself be used, if available****, instead of [scoped\_session](http://docs.sqlalchemy.org/en/rel_1_1/orm/contextual.html" \l "sqlalchemy.orm.scoping.scoped_session" \o "sqlalchemy.orm.scoping.scoped_session).

如前所述，上述模式只是将会话与Web框架集成在一起的一个潜在方法，这特别使得Web框架将Web请求与应用程序线程相关联的重要假设。 但是强烈建议使用Web框架本身提供的集成工具(如果可用)而不是scoped\_session。

In particular, while using a thread local can be convenient, it is preferable that the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) be associated ****directly with the request****, rather than with the current thread. The next section on custom scopes details a more advanced configuration which can combine the usage of [scoped\_session](http://docs.sqlalchemy.org/en/rel_1_1/orm/contextual.html" \l "sqlalchemy.orm.scoping.scoped_session" \o "sqlalchemy.orm.scoping.scoped_session) with direct request based scope, or any kind of scope.

特别地，在使用线程本地时可以方便，优选的是，Session直接与请求相关联，而不是与当前线程相关联。 关于自定义作用域的下一节详细介绍了一个更高级的配置，它可以将scoped\_session的用法与直接请求的范围或任何种类的范围相结合。

5.6.4 Using Custom Created Scopes

The [scoped\_session](http://docs.sqlalchemy.org/en/rel_1_1/orm/contextual.html" \l "sqlalchemy.orm.scoping.scoped_session" \o "sqlalchemy.orm.scoping.scoped_session) object's default behavior of "thread local" scope is only one of many options on how to "scope" a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session). A custom scope can be defined based on any existing system of getting at "the current thing we are working with".

scoped\_session对象的“线程本地”范围的默认行为只是“范围”会话的许多选项之一。 可以根据任何现有的“我们正在使用的事情”的系统来定义自定义范围。

Suppose a web framework defines a library function get\_current\_request(). An application built using this framework can call this function at any time, and the result will be some kind of Request object that represents the current request being processed. If the Request object is hashable, then this function can be easily integrated with [scoped\_session](http://docs.sqlalchemy.org/en/rel_1_1/orm/contextual.html" \l "sqlalchemy.orm.scoping.scoped_session" \o "sqlalchemy.orm.scoping.scoped_session) to associate the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) with the request. Below we illustrate this in conjunction with a hypothetical event marker provided by the web framework on\_request\_end, which allows code to be invoked whenever a request ends:

假设一个web框架定义了一个库函数get\_current\_request()。 使用此框架构建的应用程序可以随时调用此函数，结果将是表示正在处理的当前请求的某种Request对象。 如果Request对象是可哈希的，那么这个函数可以很容易地与scoped\_session集成，以将Session与请求相关联。 下面我们结合由web框架on\_request\_end提供的假设事件标记来说明这一点，这允许每当请求结束时调用代码：

**from** **my\_web\_framework** **import** get\_current\_request, on\_request\_end

**from** **sqlalchemy.orm** **import** scoped\_session, sessionmaker

Session = scoped\_session(sessionmaker(bind=some\_engine), scopefunc=get\_current\_request)

**@on\_request\_end**

**def** remove\_session(req):

Session.remove()

Above, we instantiate [scoped\_session](http://docs.sqlalchemy.org/en/rel_1_1/orm/contextual.html" \l "sqlalchemy.orm.scoping.scoped_session" \o "sqlalchemy.orm.scoping.scoped_session) in the usual way, except that we pass our request-returning function as the "scopefunc". This instructs [scoped\_session](http://docs.sqlalchemy.org/en/rel_1_1/orm/contextual.html" \l "sqlalchemy.orm.scoping.scoped_session" \o "sqlalchemy.orm.scoping.scoped_session)to use this function to generate a dictionary key whenever the registry is called upon to return the current [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session). In this case it is particularly important that we ensure a reliable "remove" system is implemented, as this dictionary is not otherwise self-managed.

以上，我们以通常的方式实例化scoped\_session，除了我们将请求返回函数作为“scopefunc”传递。 这将指示scoped\_session在调用注册表返回当前会话时使用此函数生成字典键。 在这种情况下，特别重要的是我们确保实现可靠的“删除”系统，因为这个字典不是其他的自我管理的。

5.6.4 Contextual Session API

*class*sqlalchemy.orm.scoping.**scoped\_session**(*session\_factory*, *scopefunc=None*)

Provides scoped management of [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) objects.

提供会话对象的范围管理。

See [Contextual/Thread-local Sessions](http://docs.sqlalchemy.org/en/rel_1_1/orm/contextual.html" \l "unitofwork-contextual) for a tutorial.

有关教程，请参阅上下文/线程本地会话。

**\_\_call\_\_**(*\*\*kw*)

Return the current [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session), creating it using the [scoped\_session.session\_factory](http://docs.sqlalchemy.org/en/rel_1_1/orm/contextual.html" \l "sqlalchemy.orm.scoping.scoped_session.session_factory" \o "sqlalchemy.orm.scoping.scoped_session.session_factory) if not present.

|  |  |
| --- | --- |
| **Parameters:** | ****\*\*kw**** – Keyword arguments will be passed to the [scoped\_session.session\_factory](http://docs.sqlalchemy.org/en/rel_1_1/orm/contextual.html" \l "sqlalchemy.orm.scoping.scoped_session.session_factory" \o "sqlalchemy.orm.scoping.scoped_session.session_factory) callable, if an existing [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) is not present. If the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) is present and keyword arguments have been passed, [InvalidRequestError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.InvalidRequestError" \o "sqlalchemy.exc.InvalidRequestError) is raised  如果现有会话不存在，关键字参数将被传递给scoped\_session.session\_factory可调用。 如果Session存在并且传递了关键字参数，则引发InvalidRequestError。. |

**\_\_init\_\_**(*session\_factory*, *scopefunc=None*)

Construct a new [scoped\_session](http://docs.sqlalchemy.org/en/rel_1_1/orm/contextual.html" \l "sqlalchemy.orm.scoping.scoped_session" \o "sqlalchemy.orm.scoping.scoped_session).

|  |  |
| --- | --- |
| **Parameters:** | * ****session\_factory**** – a factory to create new [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) instances. This is usually, but not necessarily, an instance of [sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker). * session\_factory - 创建新的Session实例的工厂。 这通常是但不一定是会话制造者的一个实例。 * ****scopefunc**** – optional function which defines the current scope. If not passed, the [scoped\_session](http://docs.sqlalchemy.org/en/rel_1_1/orm/contextual.html" \l "sqlalchemy.orm.scoping.scoped_session" \o "sqlalchemy.orm.scoping.scoped_session) object assumes "thread-local" scope, and will use a Python threading.local() in order to maintain the current [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session). If passed, the function should return a hashable token; this token will be used as the key in a dictionary in order to store and retrieve the current [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session). * scopefunc - 定义当前范围的可选功能。 如果没有传递，则scoped\_session对象假定为“线程本地”范围，并且将使用Python threading.local()来维护当前的会话。 如果通过，该函数应该返回一个可哈希的令牌; 该标记将用作字典中的键，以便存储和检索当前会话。 |

**configure**(*\*\*kwargs*)

reconfigure the [sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker) used by this [scoped\_session](http://docs.sqlalchemy.org/en/rel_1_1/orm/contextual.html" \l "sqlalchemy.orm.scoping.scoped_session" \o "sqlalchemy.orm.scoping.scoped_session).

See [sessionmaker.configure()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker.configure" \o "sqlalchemy.orm.session.sessionmaker.configure).

**query\_property**(*query\_cls=None*)

return a class property which produces a [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) object against the class and the current [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) when called.

e.g.:

Session = scoped\_session(sessionmaker())

**class** **MyClass**(object):

query = Session.query\_property()

*# after mappers are defined*result = MyClass.query.filter(MyClass.name=='foo').all()

Produces instances of the session's configured query class by default. To override and use a custom implementation, provide a query\_cls callable. The callable will be invoked with the class's mapper as a positional argument and a session keyword argument.

默认情况下生成会话配置的查询类的实例。 要覆盖并使用自定义实现，请提供一个query\_cls可调用。 callable将被调用与类的映射器作为位置参数和会话关键字参数。

There is no limit to the number of query properties placed on a class.

对类放置的查询属性的数量没有限制。

**remove**()

Dispose of the current [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session), if present.

处理当前会话(如果存在)。

This will first call [Session.close()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.close" \o "sqlalchemy.orm.session.Session.close) method on the current [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session), which releases any existing transactional/connection resources still being held; transactions specifically are rolled back. The [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) is then discarded. Upon next usage within the same scope, the [scoped\_session](http://docs.sqlalchemy.org/en/rel_1_1/orm/contextual.html" \l "sqlalchemy.orm.scoping.scoped_session" \o "sqlalchemy.orm.scoping.scoped_session) will produce a new [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) object.

这将首先在当前Session上调用Session.close()方法，它释放任何现有的事务/连接资源， 交易专门回滚。 会话被丢弃。 在相同范围内的下一次使用时，scoped\_session将生成一个新的Session对象。

**session\_factory***= None*

The session\_factory provided to \_\_init\_\_ is stored in this attribute and may be accessed at a later time. This can be useful when a new non-scoped [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) or [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) to the database is needed.

*class*sqlalchemy.util.**ScopedRegistry**(*createfunc*, *scopefunc*)

A Registry that can store one or multiple instances of a single class on the basis of a "scope" function.

The object implements \_\_call\_\_ as the "getter", so by calling myregistry() the contained object is returned for the current scope.

|  |  |
| --- | --- |
| **Parameters:** | * ****createfunc**** – a callable that returns a new object to be placed in the registry * ****scopefunc**** – a callable that will return a key to store/retrieve an object. |

**\_\_init\_\_**(*createfunc*, *scopefunc*)

Construct a new [ScopedRegistry](http://docs.sqlalchemy.org/en/rel_1_1/orm/contextual.html" \l "sqlalchemy.util.ScopedRegistry" \o "sqlalchemy.util.ScopedRegistry).

|  |  |
| --- | --- |
| **Parameters:** | * ****createfunc**** – A creation function that will generate a new value for the current scope, if none is present. * ****scopefunc**** – A function that returns a hashable token representing the current scope (such as, current thread identifier). |

**clear**()

Clear the current scope, if any.

**has**()

Return True if an object is present in the current scope.

**set**(*obj*)

Set the value for the current scope.

*class*sqlalchemy.util.**ThreadLocalRegistry**(*createfunc*)

Bases: sqlalchemy.util.\_collections.ScopedRegistry

A [ScopedRegistry](http://docs.sqlalchemy.org/en/rel_1_1/orm/contextual.html" \l "sqlalchemy.util.ScopedRegistry" \o "sqlalchemy.util.ScopedRegistry) that uses a threading.local() variable for storage.

# **5.7 Tracking Object and Session Changes with Events**

SQLAlchemy features an extensive [Event Listening](http://docs.sqlalchemy.org/en/rel_1_1/core/event.html) system used throughout the Core and ORM. Within the ORM, there are a wide variety of event listener hooks, which are documented at an API level at [ORM Events](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html). This collection of events has grown over the years to include lots of very useful new events as well as some older events that aren't as relevant as they once were. This section will attempt to introduce the major event hooks and when they might be used.

SQLAlchemy具有在整个Core和ORM中使用的广泛的事件侦听系统。 在ORM中，有各种各样的事件侦听器钩子，它们在ORM事件的API级别被记录。 这些事件多年来一直在增长，包括许多非常有用的新事件以及一些与之前没有关系的较旧的事件。 本节将尝试介绍主要事件钩子以及何时使用它们。

5.7.1 Persistence Events

Probably the most widely used series of events are the "persistence" events, which correspond to the [flush process](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_basics.html" \l "session-flushing). The flush is where all the decisions are made about pending changes to objects and are then emitted out to the database in the form of INSERT, UPDATE, and DELETE staetments.

可能最广泛使用的一系列事件是对应于flush过程的“持久性”事件。 刷新是在对对象的未决更改做出所有决定的基础上，然后以INSERT，UPDATE和DELETE statements的形式发送到数据库。

### before\_flush()

The [SessionEvents.before\_flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.before_flush" \o "sqlalchemy.orm.events.SessionEvents.before_flush) hook is by far the most generally useful event to use when an application wants to ensure that additional persistence changes to the database are made when a flush proceeds. Use [SessionEvents.before\_flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.before_flush" \o "sqlalchemy.orm.events.SessionEvents.before_flush) in order to operate upon objects to validate their state as well as to compose additional objects and references before they are persisted. Within this event, it is ****safe to manipulate the Session's state****, that is, new objects can be attached to it, objects can be deleted, and indivual attributes on objects can be changed freely, and these changes will be pulled into the flush process when the event hook completes.

[SessionEvents.before\_flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.before_flush" \o "sqlalchemy.orm.events.SessionEvents.before_flush)挂钩是当应用程序想要确保在进行刷新时进行数据库的额外持久性更改时最常用的事件。 使用[SessionEvents.before\_flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.before_flush" \o "sqlalchemy.orm.events.SessionEvents.before_flush)来操作对象以验证其状态，以及在持久化之前组合其他对象和引用。 在这个事件中，可以安全地操纵Session的状态，也就是说可以将新对象附加到对象上，可以删除对象，并且可以自由地改变对象上的各种属性，这些变化将被拉入到刷新流程中 事件钩子完成。

The typical [SessionEvents.before\_flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.before_flush" \o "sqlalchemy.orm.events.SessionEvents.before_flush) hook will be tasked with scanning the collections [Session.new](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.new" \o "sqlalchemy.orm.session.Session.new), [Session.dirty](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.dirty" \o "sqlalchemy.orm.session.Session.dirty) and [Session.deleted](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.deleted" \o "sqlalchemy.orm.session.Session.deleted) in order to look for objects where something will be happening.

典型的SessionEvents.before\_flush()钩子将负责扫描集合Session.new，Session.dirty和Session.deleted，以便查找会发生什么的对象。

For illustrations of [SessionEvents.before\_flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.before_flush" \o "sqlalchemy.orm.events.SessionEvents.before_flush), see examples such as [Versioning with a History Table](http://docs.sqlalchemy.org/en/rel_1_1/orm/examples.html" \l "examples-versioned-history) and [Versioning using Temporal Rows](http://docs.sqlalchemy.org/en/rel_1_1/orm/examples.html" \l "examples-versioned-rows).

有关SessionEvents.before\_flush()的插图，请参阅例如使用历史表的版本控制和使用时间行的版本控制。

### after\_flush()

The [SessionEvents.after\_flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_flush" \o "sqlalchemy.orm.events.SessionEvents.after_flush) hook is called after the SQL has been emitted for a flush process, but ****before**** the state of the objects that were flushed has been altered. That is, you can still inspect the [Session.new](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.new" \o "sqlalchemy.orm.session.Session.new), [Session.dirty](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.dirty" \o "sqlalchemy.orm.session.Session.dirty) and [Session.deleted](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.deleted" \o "sqlalchemy.orm.session.Session.deleted) collections to see what was just flushed, and you can also use history tracking features like the ones provided by [AttributeState](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.AttributeState" \o "sqlalchemy.orm.state.AttributeState) to see what changes were just persisted. In the [SessionEvents.after\_flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_flush" \o "sqlalchemy.orm.events.SessionEvents.after_flush)event, additional SQL can be emitted to the database based on what's observed to have changed.

在已经为刷新过程发出SQL之后，但在刷新的对象的状态已更改之前，会调用[SessionEvents.after\_flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_flush" \o "sqlalchemy.orm.events.SessionEvents.after_flush)钩子。 也就是说，您仍然可以检查[Session.new](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.new" \o "sqlalchemy.orm.session.Session.new)，[Session.dirty](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.dirty" \o "sqlalchemy.orm.session.Session.dirty)和[Session.deleted](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.deleted" \o "sqlalchemy.orm.session.Session.deleted)集合以查看刚被刷新的内容，还可以使用[AttributeState](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.AttributeState" \o "sqlalchemy.orm.state.AttributeState)提供的历史记录跟踪功能来查看刚刚保留的更改。 在[SessionEvents.after\_flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_flush" \o "sqlalchemy.orm.events.SessionEvents.after_flush)事件中，可以根据观察到的更改发送附加的SQL到数据库。

### after\_flush\_postexec()

[SessionEvents.after\_flush\_postexec()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_flush_postexec" \o "sqlalchemy.orm.events.SessionEvents.after_flush_postexec) is called soon after [SessionEvents.after\_flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_flush" \o "sqlalchemy.orm.events.SessionEvents.after_flush), but is invoked ****after**** the state of the objects has been modified to account for the flush that just took place. The [Session.new](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.new" \o "sqlalchemy.orm.session.Session.new), [Session.dirty](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.dirty" \o "sqlalchemy.orm.session.Session.dirty) and [Session.deleted](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.deleted" \o "sqlalchemy.orm.session.Session.deleted) collections are normally completely empty here. Use [SessionEvents.after\_flush\_postexec()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_flush_postexec" \o "sqlalchemy.orm.events.SessionEvents.after_flush_postexec) to inspect the identity map for finalized objects and possibly emit additional SQL. In this hook, there is the ability to make new changes on objects, which means the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) will again go into a "dirty" state; the mechanics of the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) here will cause it to flush ****again**** if new changes are detected in this hook if the flush were invoked in the context of [Session.commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.commit" \o "sqlalchemy.orm.session.Session.commit); otherwise, the pending changes will be bundled as part of the next normal flush. When the hook detects new changes within a [Session.commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.commit" \o "sqlalchemy.orm.session.Session.commit), a counter ensures that an endless loop in this regard is stopped after 100 iterations, in the case that an [SessionEvents.after\_flush\_postexec()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_flush_postexec" \o "sqlalchemy.orm.events.SessionEvents.after_flush_postexec) hook continually adds new state to be flushed each time it is called.

SessionEvents.after\_flush\_postexec()在SessionEvents.after\_flush()之后很快被调用，但在对象的状态被修改以解决恰好发生的flush之后被调用。 Session.new，Session.dirty和Session.deleted集合通常在这里完全为空。使用SessionEvents.after\_flush\_postexec()来检查定位对象的身份映射，并可能发出额外的SQL。在这个钩子中，有能力对对象进行新的更改，这意味着会话将再次进入“脏”状态;如果在Session.commit()的上下文中调用了flush，则此会话的机制将导致它再次刷新，如果在此钩子中检测到新的更改;否则，挂起的更改将作为下一个正常刷新的一部分进行捆绑。当钩子检测到一个Session.commit()中的新变化时，一个计数器确保在100次迭代之后停止循环，在一次SessionEvents.after\_flush\_postexec()挂起的情况下，每次都会持续添加要刷新的新状态它被称为。

### Mapper-level Events

In addition to the flush-level hooks, there is also a suite of hooks that are more fine-grained, in that they are called on a per-object basis and are broken out based on INSERT, UPDATE or DELETE. These are the mapper persistence hooks, and they too are very popular, however these events need to be approached more cautiously, as they proceed within the context of the flush process that is already ongoing; many operations are not safe to proceed here.

除了flush级别钩子之外，还有一组钩子更细致，因为它们是以每个对象为基础调用的，并且基于INSERT，UPDATE或DELETE进行分解。 这些是映射器持久性钩子，它们也非常受欢迎，但是这些事件需要更仔细地接近，因为它们已经在已经进行的刷新过程的上下文中进行; 许多操作是不安全的。

The events are:

* [MapperEvents.before\_insert()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents.before_insert" \o "sqlalchemy.orm.events.MapperEvents.before_insert)
* [MapperEvents.after\_insert()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents.after_insert" \o "sqlalchemy.orm.events.MapperEvents.after_insert)
* [MapperEvents.before\_update()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents.before_update" \o "sqlalchemy.orm.events.MapperEvents.before_update)
* [MapperEvents.after\_update()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents.after_update" \o "sqlalchemy.orm.events.MapperEvents.after_update)
* [MapperEvents.before\_delete()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents.before_delete" \o "sqlalchemy.orm.events.MapperEvents.before_delete)
* [MapperEvents.after\_delete()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents.after_delete" \o "sqlalchemy.orm.events.MapperEvents.after_delete)

Each event is passed the [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper), the mapped object itself, and the [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) which is being used to emit an INSERT, UPDATE or DELETE statement. The appeal of these events is clear, in that if an application wants to tie some activity to when a specific type of object is persisted with an INSERT, the hook is very specific; unlike the [SessionEvents.before\_flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.before_flush" \o "sqlalchemy.orm.events.SessionEvents.before_flush) event, there's no need to search through collections like [Session.new](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.new" \o "sqlalchemy.orm.session.Session.new) in order to find targets. However, the flush plan which represents the full list of every single INSERT, UPDATE, DELETE statement to be emitted has *already been decided* when these events are called, and no changes may be made at this stage. Therefore the only changes that are even possible to the given objects are upon attributes ****local**** to the object's row. Any other change to the object or other objects will impact the state of the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session), which will fail to function properly.

每个事件都被传递给Mapper，映射对象本身以及用于发出INSERT，UPDATE或DELETE语句的Connection。 这些事件的吸引力是清楚的，因为如果一个应用程序想要将一些活动绑定到一个特定类型的对象被INSERT持久化的时候，钩子是非常具体的; 与SessionEvents.before\_flush()事件不同，不需要搜索Session.new等集合来查找目标。 然而，表示要发出的每个单个INSERT，UPDATE，DELETE语句的完整列表的flush计划已经在调用这些事件时已经决定，并且在此阶段不可能进行更改。 因此，对给定对象甚至可能的唯一更改是对象行的本地属性。 对象或其他对象的任何其他更改将影响会话的状态，这将无法正常运行。

Operations that are not supported within these mapper-level persistence events include:

* [Session.add()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.add" \o "sqlalchemy.orm.session.Session.add)
* [Session.delete()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.delete" \o "sqlalchemy.orm.session.Session.delete)
* Mapped collection append, add, remove, delete, discard, etc.
* Mapped relationship attribute set/del events, i.e. someobject.related = someotherobject

The reason the [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) is passed is that it is encouraged that ****simple SQL operations take place here****, directly on the [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection), such as incrementing counters or inserting extra rows within log tables. When dealing with the [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection), it is expected that Core-level SQL operations will be used; e.g. those described in [SQL Expression Language Tutorial](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html).

连接传递的原因是鼓励简单的SQL操作在这里直接在Connection上进行，例如增加计数器或在日志表中插入额外的行。 在处理Connection时，期望使用Core-level SQL操作; 例如 在SQL表达语言教程中描述的那些。

There are also many per-object operations that don't need to be handled within a flush event at all. The most common alternative is to simply establish additional state along with an object inside its \_\_init\_\_() method, such as creating additional objects that are to be associated with the new object. Using validators as described in [Simple Validators](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapped_attributes.html" \l "simple-validators) is another approach; these functions can intercept changes to attributes and establish additional state changes on the target object in response to the attribute change. With both of these approaches, the object is in the correct state before it ever gets to the flush step.

还有许多每个对象的操作根本不需要在flush事件中处理。 最常见的替代方法是简单地在\_\_init \_\_()方法内建立一个对象，例如创建与新对象关联的其他对象。 使用简单验证器中所述的验证器是另一种方法; 这些功能可以拦截对属性的更改，并根据属性更改在目标对象上建立附加状态更改。 使用这两种方法，对象在进入冲洗步骤之前处于正确的状态。

5.7.2 Object Lifecycle Events

Another use case for events is to track the lifecycle of objects. This refers to the states first introduced at [Quickie Intro to Object States](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_state_management.html" \l "session-object-states).

事件的另一个用例是跟踪对象的生命周期。 这是指首先在[Quickie Intro to Object States](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_state_management.html" \l "session-object-states)介绍的状态。

*New in version 1.1:*added a system of events that intercept all possible state transitions of an object within the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session).添加了一个事件系统，该事件拦截会话内的对象的所有可能状态转换。

All the states above can be tracked fully with events. Each event represents a distinct state transition, meaning, the starting state and the destination state are both part of what are tracked. With the exception of the initial transient event, all the events are in terms of the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) object or class, meaning they can be associated either with a specific [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) object:

上述所有状态都可以充分跟踪事件。 每个事件代表不同的状态转换，意思是起始状态和目标状态都是跟踪的一部分。 除了初始瞬态事件之外，所有事件都是以Session对象或类来表示的，这意味着它们可以与特定的Session对象相关联：

**from** **sqlalchemy** **import** event

**from** **sqlalchemy.orm** **import** Session

session = Session()

**@event**.listens\_for(session, 'transient\_to\_pending')

**def** object\_is\_pending(session, obj):

print("new pending: *%s*" % obj)

Or with the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) class itself, as well as with a specific [sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker), which is likely the most useful form:

或者使用[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)类本身，以及一个特定的[sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker)，这可能是最有用的形式：

**from** **sqlalchemy** **import** event

**from** **sqlalchemy.orm** **import** sessionmaker

maker = sessionmaker()

**@event**.listens\_for(maker, 'transient\_to\_pending')

**def** object\_is\_pending(session, obj):

print("new pending: *%s*" % obj)

The listeners can of course be stacked on top of one function, as is likely to be common. For example, to track all objects that are entering the persistent state:

听众当然可以堆叠在一个功能的顶部，这很可能是常见的。 例如，要跟踪进入持久状态的所有对象：

**@event**.listens\_for(maker, "pending\_to\_persistent")

**@event**.listens\_for(maker, "deleted\_to\_persistent")

**@event**.listens\_for(maker, "detached\_to\_persistent")

**@event**.listens\_for(maker, "loaded\_as\_persistent")

**def** detect\_all\_persistent(session, instance):

print("object is now persistent: *%s*" % instance)

### Transient暂态

All mapped objects when first constructed start out as [transient](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-transient). In this state, the object exists alone and doesn't have an association with any [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session). For this initial state, there's no specific "transition" event since there is no [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session), however if one wanted to intercept when any transient object is created, the[InstanceEvents.init()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.InstanceEvents.init" \o "sqlalchemy.orm.events.InstanceEvents.init) method is probably the best event. This event is applied to a specific class or superclass. For example, to intercept all new objects for a particular declarative base:

第一次构造时的所有映射对象开始为瞬态。 在这种状态下，对象独立存在，并且与任何[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)都没有关联。 对于这个初始状态，没有特定的“转换”事件，因为没有[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)，但是如果想在任何瞬态对象被创建时拦截，那么这个[InstanceEvents.init()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.InstanceEvents.init" \o "sqlalchemy.orm.events.InstanceEvents.init)方法可能是最好的事件。 此事件应用于特定的类或超类。 例如，为了拦截特定声明性基础的所有新对象：

**from** **sqlalchemy.ext.declarative** **import** declarative\_base

**from** **sqlalchemy** **import** event

Base = declarative\_base()

**@event**.listens\_for(Base, "init", propagate=**True**)

**def** intercept\_init(instance, args, kwargs):

print("new transient: *%s*" % instance)

### Transient to Pending暂态到挂起

The transient object becomes [pending](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-pending) when it is first associated with a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) via the [Session.add()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.add" \o "sqlalchemy.orm.session.Session.add) or [Session.add\_all()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.add_all" \o "sqlalchemy.orm.session.Session.add_all) method. An object may also become part of a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) as a result of a ["cascade"](http://docs.sqlalchemy.org/en/rel_1_1/orm/cascades.html" \l "unitofwork-cascades) from a referencing object that was explicitly added. The transient to pending transition is detectable using the [SessionEvents.transient\_to\_pending()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.transient_to_pending" \o "sqlalchemy.orm.events.SessionEvents.transient_to_pending) event:

当它首先通过Session.add()或Session.add\_all()方法与会话相关联时，临时对象将变为挂起状态。 由于明确添加的引用对象的“级联”，对象也可能成为会话的一部分。 使用SessionEvents.transient\_to\_pending()事件可以检测到暂挂到待处理的转换：

**@event**.listens\_for(sessionmaker, "transient\_to\_pending")

**def** intercept\_transient\_to\_pending(session, object\_):

print("transient to pending: *%s*" % object\_)

### Pending to Persistent

The [pending](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-pending) object becomes [persistent](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-persistent) when a flush proceeds and an INSERT statement takes place for the instance. The object now has an identity key. Track pending to persistent with the [SessionEvents.pending\_to\_persistent()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.pending_to_persistent" \o "sqlalchemy.orm.events.SessionEvents.pending_to_persistent) event:

当进行刷新并对该实例执行INSERT语句时，待处理对象将变为持久状态。 对象现在有一个身份密钥。 跟踪待执行SessionEvents.pending\_to\_persistent()事件：

**@event**.listens\_for(sessionmaker, "pending\_to\_persistent")

**def** intercept\_pending\_to\_persistent(session, object\_):

print("pending to persistent: *%s*" % object\_)

### Pending to Transient

The [pending](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-pending) object can revert back to [transient](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-transient) if the [Session.rollback()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.rollback" \o "sqlalchemy.orm.session.Session.rollback) method is called before the pending object has been flushed, or if the [Session.expunge()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.expunge" \o "sqlalchemy.orm.session.Session.expunge) method is called for the object before it is flushed. Track pending to transient with the [SessionEvents.pending\_to\_transient()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.pending_to_transient" \o "sqlalchemy.orm.events.SessionEvents.pending_to_transient)event:

如果在清除挂起对象之前调用了Session.rollback()方法，或者在刷新对象之前调用了Session.expunge()方法，则挂起对象可以恢复为transient。 使用SessionEvents.pending\_to\_transient()事件跟踪等待瞬态：

**@event**.listens\_for(sessionmaker, "pending\_to\_transient")

**def** intercept\_pending\_to\_transient(session, object\_):

print("transient to pending: *%s*" % object\_)

### Loaded as Persistent

Objects can appear in the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) directly in the [persistent](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-persistent) state when they are loaded from the database. Tracking this state transition is synonymous with tracking objects as they are loaded, and is synonymous with using the [InstanceEvents.load()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.InstanceEvents.load" \o "sqlalchemy.orm.events.InstanceEvents.load) instance-level event. However, the[SessionEvents.loaded\_as\_persistent()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.loaded_as_persistent" \o "sqlalchemy.orm.events.SessionEvents.loaded_as_persistent) event is provided as a session-centric hook for intercepting objects as they enter the persistent state via this particular avenue:

当数据库加载时，对象可以直接出现在持久状态中。 跟踪此状态转换是跟踪对象加载时的同义词，与使用[InstanceEvents.load()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.InstanceEvents.load" \o "sqlalchemy.orm.events.InstanceEvents.load)实例级事件同义。 但是，[SessionEvents.loaded\_as\_persistent()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.loaded_as_persistent" \o "sqlalchemy.orm.events.SessionEvents.loaded_as_persistent)事件是作为一个以会话为中心的挂钩提供的，用于在通过该特定通道进入持久状态时拦截对象：

**@event**.listens\_for(sessionmaker, "loaded\_as\_persistent")

**def** intercept\_loaded\_as\_persistent(session, object\_):

print("object loaded into persistent state: *%s*" % object\_)

### Persistent to Transient

The persistent object can revert to the transient state if the [Session.rollback()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.rollback" \o "sqlalchemy.orm.session.Session.rollback) method is called for a transaction where the object was first added as pending. In the case of the ROLLBACK, the INSERT statement that made this object persistent is rolled back, and the object is evicted from the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) to again become transient. Track objects that were reverted to transient from persistent using the [SessionEvents.persistent\_to\_transient()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.persistent_to_transient" \o "sqlalchemy.orm.events.SessionEvents.persistent_to_transient) event hook:

如果对象首先被添加为挂起的事务调用了Session.rollback()方法，持久化对象可以恢复为暂态。 在ROLLBACK的情况下，导致此对象持久的INSERT语句被回滚，并且对象从会话中逐出被再次变为暂时性。 使用SessionEvents.persistent\_to\_transient()事件钩子跟踪从持久性恢复为瞬态的对象：

**@event**.listens\_for(sessionmaker, "persistent\_to\_transient")

**def** intercept\_persistent\_to\_transient(session, object\_):

print("persistent to transient: *%s*" % object\_)

### Persistent to Deleted

The persistent object enters the [deleted](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-deleted) state when an object marked for deletion is deleted from the database within the flush process. Note that this is ****not the same****as when the [Session.delete()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.delete" \o "sqlalchemy.orm.session.Session.delete) method is called for a target object. The [Session.delete()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.delete" \o "sqlalchemy.orm.session.Session.delete) method only ****marks**** the object for deletion; the actual DELETE statement is not emitted until the flush proceeds. It is subsequent to the flush that the "deleted" state is present for the target object.

当冲洗过程中从数据库中删除标记为删除的对象时，持久对象进入已删除状态。 请注意，当为目标对象调用[Session.delete()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.delete" \o "sqlalchemy.orm.session.Session.delete)方法时，这不一样。 [Session.delete()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.delete" \o "sqlalchemy.orm.session.Session.delete)方法只标记要删除的对象; 在刷新进行之前，不会发出实际的DELETE语句。 在刷新之后，目标对象存在“已删除”状态。

Within the "deleted" state, the object is only marginally associated with the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session). It is not present in the identity map nor is it present in the [Session.deleted](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.deleted" \o "sqlalchemy.orm.session.Session.deleted) collection that refers to when it was pending for deletion.

在“已删除”状态下，该对象仅与会话相关联。 它不存在于身份映射中，也不存在于Session.deleted集合中，当它被等待删除时。

From the "deleted" state, the object can go either to the detached state when the transaction is committed, or back to the persistent state if the transaction is instead rolled back.

从“已删除”状态，当事务提交时，对象可以转到分离状态，如果事务被回滚则返回到持久状态。

Track the persistent to deleted transition with [SessionEvents.persistent\_to\_deleted()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.persistent_to_deleted" \o "sqlalchemy.orm.events.SessionEvents.persistent_to_deleted):

跟踪SessionEvents.persistent\_to\_deleted()的持久删除转换：

**@event**.listens\_for(sessionmaker, "persistent\_to\_deleted")

**def** intercept\_persistent\_to\_deleted(session, object\_):

print("object was DELETEd, is now in deleted state: *%s*" % object\_)

### Deleted to Detached

The deleted object becomes [detached](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-detached) when the session's transaction is committed. After the [Session.commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.commit" \o "sqlalchemy.orm.session.Session.commit) method is called, the database transaction is final and the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) now fully discards the deleted object and removes all associations to it. Track the deleted to detached transition using [SessionEvents.deleted\_to\_detached()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.deleted_to_detached" \o "sqlalchemy.orm.events.SessionEvents.deleted_to_detached):

当会话的事务提交时，被删除的对象变得分离。 在调用Session.commit()方法之后，数据库事务是final，Session现在完全丢弃被删除的对象，并删除所有关联。 使用SessionEvents.deleted\_to\_detached()跟踪已删除到分离的转换：

**@event**.listens\_for(sessionmaker, "deleted\_to\_detached")

**def** intercept\_deleted\_to\_detached(session, object\_):

print("deleted to detached: *%s*" % object\_)

**Note**

While the object is in the deleted state, the [InstanceState.deleted](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState.deleted" \o "sqlalchemy.orm.state.InstanceState.deleted) attribute, accessible using inspect(object).deleted, returns True. However when the object is detached, [InstanceState.deleted](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState.deleted" \o "sqlalchemy.orm.state.InstanceState.deleted) will again return False. To detect that an object was deleted, regardless of whether or not it is detached, use the [InstanceState.was\_deleted](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState.was_deleted" \o "sqlalchemy.orm.state.InstanceState.was_deleted) accessor.

### Persistent to Detached

The persistent object becomes [detached](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-detached) when the object is de-associated with the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session), via the [Session.expunge()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.expunge" \o "sqlalchemy.orm.session.Session.expunge), [Session.expunge\_all()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.expunge_all" \o "sqlalchemy.orm.session.Session.expunge_all), or [Session.close()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.close" \o "sqlalchemy.orm.session.Session.close) methods.

**Note**

An object may also become ****implicitly detached**** if its owning [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) is dereferenced by the application and discarded due to garbage collection. In this case, ****no event is emitted****.

Track objects as they move from persistent to detached using the [SessionEvents.persistent\_to\_detached()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.persistent_to_detached" \o "sqlalchemy.orm.events.SessionEvents.persistent_to_detached) event:

**@event**.listens\_for(sessionmaker, "persistent\_to\_detached")

**def** intecept\_persistent\_to\_detached(session, object\_):

print("object became detached: *%s*" % object\_)

### Detached to Persistent

The detached object becomes persistent when it is re-associated with a session using the [Session.add()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.add" \o "sqlalchemy.orm.session.Session.add) or equivalent method. Track objects moving back to persistent from detached using the [SessionEvents.detached\_to\_persistent()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.detached_to_persistent" \o "sqlalchemy.orm.events.SessionEvents.detached_to_persistent) event:

当使用[Session.add()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.add" \o "sqlalchemy.orm.session.Session.add)或等效方法与会话重新关联时，分离的对象将变得持久。 使用[SessionEvents.detached\_to\_persistent()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.detached_to_persistent" \o "sqlalchemy.orm.events.SessionEvents.detached_to_persistent)事件跟踪从分离中移回到持久性的对象：

**@event**.listens\_for(sessionmaker, "detached\_to\_persistent")

**def** intecept\_detached\_to\_persistent(session, object\_):

print("object became persistent again: *%s*" % object\_)

### Deleted to Persistent

The [deleted](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-deleted) object can be reverted to the [persistent](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-persistent) state when the transaction in which it was DELETEd was rolled back using the [Session.rollback()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.rollback" \o "sqlalchemy.orm.session.Session.rollback) method. Track deleted objects moving back to the persistent state using the [SessionEvents.deleted\_to\_persistent()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.deleted_to_persistent" \o "sqlalchemy.orm.events.SessionEvents.deleted_to_persistent) event:

当使用Session.rollback()方法回滚DELETED的事务时，删除的对象可以恢复为持久状态。 使用SessionEvents.deleted\_to\_persistent()事件跟踪删除的对象回到持久状态：

**@event**.listens\_for(sessionmaker, "transient\_to\_pending")

**def** intercept\_transient\_to\_pending(session, object\_):

print("transient to pending: *%s*" % object\_)

### 5.7.3 Transaction Events

Transaction events allow an application to be notifed when transaction boundaries occur at the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) level as well as when the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) changes the transactional state on [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) objects.

* [SessionEvents.after\_transaction\_create()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_transaction_create" \o "sqlalchemy.orm.events.SessionEvents.after_transaction_create), [SessionEvents.after\_transaction\_end()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_transaction_end" \o "sqlalchemy.orm.events.SessionEvents.after_transaction_end) - these events track the logical transaction scopes of the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) in a way that is not specific to individual database connections. These events are intended to help with integration of transaction-tracking systems such as zope.sqlalchemy. Use these events when the application needs to align some external scope with the transactional scope of the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session). These hooks mirror the "nested" transactional behavior of the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session), in that they track logical "subtransactions" as well as "nested" (e.g. SAVEPOINT) transactions.
* [SessionEvents.before\_commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.before_commit" \o "sqlalchemy.orm.events.SessionEvents.before_commit), [SessionEvents.after\_commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_commit" \o "sqlalchemy.orm.events.SessionEvents.after_commit), [SessionEvents.after\_begin()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_begin" \o "sqlalchemy.orm.events.SessionEvents.after_begin),[SessionEvents.after\_rollback()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_rollback" \o "sqlalchemy.orm.events.SessionEvents.after_rollback), [SessionEvents.after\_soft\_rollback()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_soft_rollback" \o "sqlalchemy.orm.events.SessionEvents.after_soft_rollback) - These events allow tracking of transaction events from the perspective of database connections. [SessionEvents.after\_begin()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_begin" \o "sqlalchemy.orm.events.SessionEvents.after_begin) in particular is a per-connection event; a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) that maintains more than one connection will emit this event for each connection individually as those connections become used within the current transaction. The rollback and commit events then refer to when the DBAPI connections themselves have received rollback or commit instructions directly.

5.7.4 Attribute Change Events

The attribute change events allow interception of when specific attributes on an object are modified. These events include [AttributeEvents.set()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.AttributeEvents.set" \o "sqlalchemy.orm.events.AttributeEvents.set),[AttributeEvents.append()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.AttributeEvents.append" \o "sqlalchemy.orm.events.AttributeEvents.append), and [AttributeEvents.remove()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.AttributeEvents.remove" \o "sqlalchemy.orm.events.AttributeEvents.remove). These events are extremely useful, particularly for per-object validation operations; however, it is often much more convenient to use a "validator" hook, which uses these hooks behind the scenes; see [Simple Validators](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapped_attributes.html" \l "simple-validators) for background on this. The attribute events are also behind the mechanics of back references. An example illustrating use of attribute events is in [Attribute Instrumentation](http://docs.sqlalchemy.org/en/rel_1_1/orm/examples.html" \l "examples-instrumentation).

属性更改事件允许拦截对象上的特定属性进行修改。 这些事件包括[AttributeEvents.set()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.AttributeEvents.set" \o "sqlalchemy.orm.events.AttributeEvents.set)，[AttributeEvents.append()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.AttributeEvents.append" \o "sqlalchemy.orm.events.AttributeEvents.append)和[AttributeEvents.remove()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.AttributeEvents.remove" \o "sqlalchemy.orm.events.AttributeEvents.remove)。 这些事件非常有用，特别是对于每个对象的验证操作; 然而，使用“验证器”钩子通常更方便，后者在后台使用这些钩子; 请参阅[Simple Validators](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapped_attributes.html" \l "simple-validators)的背景。 属性事件也是反向引用的机制。 说明属性事件的使用示例在Attribute Instrumentation中。

# **5.8 Session API**

# 5.8.1 Session and sessionmaker()

*class*sqlalchemy.orm.session.**sessionmaker**(*bind=None*, *class\_=<class 'sqlalchemy.orm.session.Session'>*, *autoflush=True*, *autocommit=False*, *expire\_on\_commit=True*, *info=None*, *\*\*kw*)

Bases: sqlalchemy.orm.session.\_SessionClassMethods

A configurable [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) factory.

可配置的会话工厂。

The [sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker) factory generates new [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) objects when called, creating them given the configurational arguments established here.

会话制作工厂在调用时生成新的Session对象，给出了这里建立的配置参数。

e.g.:

*# global scope*

Session = sessionmaker(autoflush=**False**)

*# later, in a local scope, create and use a session:*

sess = Session()

Any keyword arguments sent to the constructor itself will override the "configured" keywords:

发送到构造函数本身的任何关键字参数将覆盖"配置"关键字：

Session = sessionmaker()

*# bind an individual session to a connection*

sess = Session(bind=connection)

The class also includes a method [configure()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker.configure" \o "sqlalchemy.orm.session.sessionmaker.configure), which can be used to specify additional keyword arguments to the factory, which will take effect for subsequent [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) objects generated. This is usually used to associate one or more [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) objects with an existing [sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker) factory before it is first used:

该类还包括一个方法configure()，可以用于指定工厂的其他关键字参数，这将对生成的后续会话对象生效。 这通常用于将一个或多个引擎对象与现有的会话制造商工厂首次关联之前关联：

*# application starts*

Session = sessionmaker()

*# ... later*

engine = create\_engine('sqlite:///foo.db')

Session.configure(bind=engine)

sess = Session()

**\_\_call\_\_**(*\*\*local\_kw*)

Produce a new [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) object using the configuration established in this [sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker).

In Python, the \_\_call\_\_ method is invoked on an object when it is "called" in the same way as a function:

Session = sessionmaker()

session = Session() *# invokes sessionmaker.\_\_call\_\_()*

**\_\_init\_\_**(*bind=None*, *class\_=<class 'sqlalchemy.orm.session.Session'>*, *autoflush=True*, *autocommit=False*, *expire\_on\_commit=True*, *info=None*, *\*\*kw*)

Construct a new [sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker).

All arguments here except for class\_ correspond to arguments accepted by [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) directly. See the [Session.\_\_init\_\_()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.__init__" \o "sqlalchemy.orm.session.Session.__init__) docstring for more details on parameters.

这里除了class\_之外的所有参数都对应于Session直接接受的参数。 有关参数的更多详细信息，请参阅Session.\_\_ init \_\_()docstring。

|  |  |
| --- | --- |
| **Parameters:** | * ****bind**** – a [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) or other [Connectable](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connectable" \o "sqlalchemy.engine.Connectable) with which newly created [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) objects will be associated. * ****class\_**** – class to use in order to create new [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) objects. Defaults to [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session). * ****autoflush**** – The autoflush setting to use with newly created [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) objects. * ****autocommit**** – The autocommit setting to use with newly created [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) objects. * ****expire\_on\_commit=True**** – the expire\_on\_commit setting to use with newly created [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) objects. * ****info –****optional dictionary of information that will be available via [Session.info](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.info" \o "sqlalchemy.orm.session.Session.info). Note this dictionary is *updated*, not replaced, when the info parameter is specified to the specific [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) construction operation.   *New in version 0.9.0.*   * ****\*\*kw**** – all other keyword arguments are passed to the constructor of newly created [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) objects. |

**close\_all**()

*inherited from the* close\_all() *method of* \_SessionClassMethods

Close *all* sessions in memory.

**configure**(*\*\*new\_kw*)

(Re)configure the arguments for this sessionmaker.

e.g.:

Session = sessionmaker()

Session.configure(bind=create\_engine('sqlite://'))

**identity\_key**(*\*args*, *\*\*kwargs*)

*inherited from the* identity\_key() *method of* \_SessionClassMethods

Return an identity key.

This is an alias of [util.identity\_key()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.util.identity_key" \o "sqlalchemy.orm.util.identity_key).

**object\_session**(*instance*)

*inherited from the* object\_session() *method of* \_SessionClassMethods

Return the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) to which an object belongs.

This is an alias of [object\_session()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.object_session" \o "sqlalchemy.orm.session.object_session).

*class*sqlalchemy.orm.session.**Session**(*bind=None*, *autoflush=True*, *expire\_on\_commit=True*, *\_enable\_transaction\_accounting=True*, *autocommit=False*, *twophase=False*, *weak\_identity\_map=True*, *binds=None*, *extension=None*, *info=None*, *query\_cls=<class 'sqlalchemy.orm.query.Query'>*)

Bases: sqlalchemy.orm.session.\_SessionClassMethods

Manages persistence operations for ORM-mapped objects.

The Session's usage paradigm is described at [Using the Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session.html).

**\_\_init\_\_**(*bind=None*, *autoflush=True*, *expire\_on\_commit=True*, *\_enable\_transaction\_accounting=True*, *autocommit=False*, *twophase=False*, *weak\_identity\_map=True*, *binds=None*, *extension=None*, *info=None*, *query\_cls=<class 'sqlalchemy.orm.query.Query'>*)

Construct a new Session.

See also the [sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker) function which is used to generate a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)-producing callable with a given set of arguments.

|  |  |
| --- | --- |
| **Parameters:** | * ****autocommit –****   **Warning**  The autocommit flag is ****not for general use****, and if it is used, queries should only be invoked within the span of a [Session.begin()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.begin" \o "sqlalchemy.orm.session.Session.begin) / [Session.commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.commit" \o "sqlalchemy.orm.session.Session.commit) pair. Executing queries outside of a demarcated transaction is a legacy mode of usage, and can in some cases lead to concurrent connection checkouts.  自动提交标志不是一般用途，如果使用，则只能在Session.begin()/ Session.commit()对的范围内调用查询。 在划分的事务之外执行查询是传统的使用模式，并且在某些情况下可能导致并发连接检出。  Defaults to False. When True, the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) does not keep a persistent transaction running, and will acquire connections from the engine on an as-needed basis, returning them immediately after their use. Flushes will begin and commit (or possibly rollback) their own transaction if no transaction is present. When using this mode, the [Session.begin()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.begin" \o "sqlalchemy.orm.session.Session.begin) method is used to explicitly start transactions.  默认为False。 当为True时，会话不会持续执行事务，并且将根据需要从引擎获取连接，并在使用后立即返回。 如果没有事务存在，刷新将开始并提交(或可能回滚)自己的事务。 当使用此模式时，会使用Session.begin()方法来显式启动事务。  **See also**  [Autocommit Mode](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_transaction.html" \l "session-autocommit)   * ****autoflush**** – When True, all query operations will issue a [flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.flush" \o "sqlalchemy.orm.session.Session.flush) call to this Session before proceeding. This is a convenience feature so that [flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.flush" \o "sqlalchemy.orm.session.Session.flush) need not be called repeatedly in order for database queries to retrieve results. It's typical that autoflush is used in conjunction with autocommit=False. In this scenario, explicit calls to [flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.flush" \o "sqlalchemy.orm.session.Session.flush) are rarely needed; you usually only need to call [commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.commit" \o "sqlalchemy.orm.session.Session.commit) (which flushes) to finalize changes.当为True时，所有查询操作将在继续操作之前发出对此Session的flush()调用。 这是一个方便的功能，因此不需要重复调用flush()，以便数据库查询检索结果。 通常，autoflush与autocommit = False结合使用。 在这种情况下，很少需要对flush()的显式调用; 你通常只需要调用commit()(它刷新)来完成更改 * ****bind**** – An optional [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) or [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) to which this Session should be bound. When specified, all SQL operations performed by this session will execute via this connectable.该会话应绑定到的可选引擎或连接。 指定时，此会话执行的所有SQL操作将通过此可连接执行。 * ****binds –****An optional dictionary which contains more granular"bind" information than the bind parameter provides. This dictionary can map individual :class`.Table` instances as well as [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) instances to individual [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) or [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) objects. Operations which proceed relative to a particular [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) will consult this dictionary for the direct [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) instance as well as the mapper's mapped\_table attribute in order to locate a connectable to use. The full resolution is described in the[Session.get\_bind()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.get_bind" \o "sqlalchemy.orm.session.Session.get_bind). Usage looks like:包含比bind参数提供的更细粒度的“bind”信息的可选字典。 这个字典可以将单独的：class`.Table`实例以及Mapper实例映射到单独的Engine或Connection对象。 相对于特定映射器进行的操作将参考该字典的直接Mapper实例以及映射器的mapped\_table属性，以便找到可连接使用。 完整的分辨率在TheSession.get\_bind()中描述。 用法看起来像：   Session = sessionmaker(binds={  SomeMappedClass: create\_engine('postgresql://engine1'),  somemapper: create\_engine('postgresql://engine2'),  some\_table: create\_engine('postgresql://engine3'),  })  Also see the [Session.bind\_mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.bind_mapper" \o "sqlalchemy.orm.session.Session.bind_mapper) and [Session.bind\_table()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.bind_table" \o "sqlalchemy.orm.session.Session.bind_table) methods.   * ****class\_**** – Specify an alternate class other than sqlalchemy.orm.session.Session which should be used by the returned class. This is the only argument that is local to the [sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker) function, and is not sent directly to the constructor for Session. * ****\_enable\_transaction\_accounting**** – Defaults to True. A legacy-only flag which when False disables *all* 0.5-style object accounting on transaction boundaries, including auto-expiry of instances on rollback and commit, maintenance of the "new" and "deleted" lists upon rollback, and autoflush of pending changes upon [begin()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.begin" \o "sqlalchemy.orm.session.Session.begin), all of which are interdependent. * ****expire\_on\_commit**** – Defaults to True. When True, all instances will be fully expired after each [commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.commit" \o "sqlalchemy.orm.session.Session.commit), so that all attribute/object access subsequent to a completed transaction will load from the most recent database state. * ****extension**** – An optional [SessionExtension](http://docs.sqlalchemy.org/en/rel_1_1/orm/deprecated.html" \l "sqlalchemy.orm.interfaces.SessionExtension" \o "sqlalchemy.orm.interfaces.SessionExtension) instance, or a list of such instances, which will receive pre- and post- commit and flush events, as well as a post-rollback event. ****Deprecated.**** Please see [SessionEvents](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents" \o "sqlalchemy.orm.events.SessionEvents). * ****info –****optional dictionary of arbitrary data to be associated with this [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session). Is available via the [Session.info](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.info" \o "sqlalchemy.orm.session.Session.info) attribute. Note the dictionary is copied at construction time so that modifications to the per- [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) dictionary will be local to that [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session).   *New in version 0.9.0.*   * ****query\_cls**** – Class which should be used to create new Query objects, as returned by the [query()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.query" \o "sqlalchemy.orm.session.Session.query) method. Defaults to [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query). * ****twophase**** – When True, all transactions will be started as a "two phase" transaction, i.e. using the "two phase" semantics of the database in use along with an XID. During a [commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.commit" \o "sqlalchemy.orm.session.Session.commit), after [flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.flush" \o "sqlalchemy.orm.session.Session.flush) has been issued for all attached databases, the [prepare()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.TwoPhaseTransaction.prepare" \o "sqlalchemy.engine.TwoPhaseTransaction.prepare) method on each database's [TwoPhaseTransaction](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.TwoPhaseTransaction" \o "sqlalchemy.engine.TwoPhaseTransaction) will be called. This allows each database to roll back the entire transaction, before each transaction is committed. * ****weak\_identity\_map**** – Defaults to True - when set to False, objects placed in the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) will be strongly referenced until explicitly removed or the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) is closed. ****Deprecated**** - The strong reference identity map is legacy. See the recipe at [Session Referencing Behavior](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_state_management.html" \l "session-referencing-behavior) for an event-based approach to maintaining strong identity references. |

**add**(*instance*, *\_warn=True*)

Place an object in the Session.

Its state will be persisted to the database on the next flush operation.

Repeated calls to add() will be ignored. The opposite of add() is expunge().

**add\_all**(*instances*)

Add the given collection of instances to this Session.

**begin**(*subtransactions=False*, *nested=False*)

Begin a transaction on this [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session).

The [Session.begin()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.begin" \o "sqlalchemy.orm.session.Session.begin) method is only meaningful if this session is in ****autocommit mode**** prior to it being called; see [Autocommit Mode](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_transaction.html" \l "session-autocommit) for background on this setting.

The method will raise an error if this [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) is already inside of a transaction, unless [Session.begin.subtransactions](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.begin.params.subtransactions" \o "sqlalchemy.orm.session.Session.begin) or[Session.begin.nested](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.begin.params.nested" \o "sqlalchemy.orm.session.Session.begin) are specified.

|  |  |
| --- | --- |
| **Parameters:** | * ****subtransactions**** – if True, indicates that this [begin()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.begin" \o "sqlalchemy.orm.session.Session.begin) can create a subtransaction if a transaction is already in progress. For documentation on subtransactions, please see [Using Subtransactions with Autocommit](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_transaction.html" \l "session-subtransactions). * ****nested**** – if True, begins a SAVEPOINT transaction and is equivalent to calling [begin\_nested()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.begin_nested" \o "sqlalchemy.orm.session.Session.begin_nested). For documentation on SAVEPOINT transactions, please see [Using SAVEPOINT](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_transaction.html" \l "session-begin-nested). |
| **Returns:** | the [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction) object. Note that [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction) acts as a Python context manager, allowing [Session.begin()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.begin" \o "sqlalchemy.orm.session.Session.begin) to be used in a "with" block. See [Autocommit Mode](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_transaction.html" \l "session-autocommit) for an example. |

**See also**

[Autocommit Mode](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_transaction.html" \l "session-autocommit)

[Session.begin\_nested()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.begin_nested" \o "sqlalchemy.orm.session.Session.begin_nested)

**begin\_nested**()

Begin a "nested" transaction on this Session, e.g. SAVEPOINT.

The target database(s) and associated drivers must support SQL SAVEPOINT for this method to function correctly.

For documentation on SAVEPOINT transactions, please see [Using SAVEPOINT](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_transaction.html" \l "session-begin-nested).

在此会话中开始“嵌套”事务，例如SAVEPOINT。

目标数据库和关联的驱动程序必须支持SQL SAVEPOINT才能使此方法正常运行。

有关SAVEPOINT交易的文档，请参阅[Using SAVEPOINT](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_transaction.html" \l "session-begin-nested)。

|  |  |
| --- | --- |
| **Returns:** | the [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction) object. Note that [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction) acts as a context manager, allowing [Session.begin\_nested()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.begin_nested" \o "sqlalchemy.orm.session.Session.begin_nested) to be used in a "with" block. See [Using SAVEPOINT](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_transaction.html" \l "session-begin-nested) for a usage example. |

**See also**

[Using SAVEPOINT](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_transaction.html" \l "session-begin-nested)

[Serializable isolation / Savepoints / Transactional DDL](http://docs.sqlalchemy.org/en/rel_1_1/dialects/sqlite.html" \l "pysqlite-serializable) - special workarounds required with the SQLite driver in order for SAVEPOINT to work correctly.

**bind\_mapper**(*mapper*, *bind*)

Associate a [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) with a "bind", e.g. a [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) or [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection).

The given mapper is added to a lookup used by the [Session.get\_bind()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.get_bind" \o "sqlalchemy.orm.session.Session.get_bind) method.

**bind\_table**(*table*, *bind*)

Associate a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) with a "bind", e.g. a [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) or [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection).

The given mapper is added to a lookup used by the [Session.get\_bind()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.get_bind" \o "sqlalchemy.orm.session.Session.get_bind) method.

**bulk\_insert\_mappings**(*mapper*, *mappings*, *return\_defaults=False*, *render\_nulls=False*)

Perform a bulk insert of the given list of mapping dictionaries.

执行给定的映射字典列表的批量插入。

The bulk insert feature allows plain Python dictionaries to be used as the source of simple INSERT operations which can be more easily grouped together into higher performing "execute many" operations. Using dictionaries, there is no "history" or session state management features in use, reducing latency when inserting large numbers of simple rows.

批量插入功能允许使用简单的Python字典作为简单INSERT操作的源，可以更容易地将其组合到更高性能的"执行多个"操作中。 使用字典，没有使用"历史"或会话状态管理功能，减少插入大量简单行时的延迟。

The values within the dictionaries as given are typically passed without modification into Core Insert() constructs, after organizing the values within them across the tables to which the given mapper is mapped.

在给定映射器映射的表中组织其中的值之后，给定的字典中的值通常会在Core Insert()构造中无修改地传递。

*New in version 1.0.0.*

**Warning**

The bulk insert feature allows for a lower-latency INSERT of rows at the expense of most other unit-of-work features. Features such as object management, relationship handling, and SQL clause support are ****silently omitted**** in favor of raw INSERT of records.

****批量插入功能允许以较低延迟的行INSERT为代价牺牲大多数其他工作单元功能。 静态地省略了对象管理，关系处理和SQL子句支持等功能，支持原始的INSERT记录。****

****Please read the list of caveats at**** [Bulk Operations](http://docs.sqlalchemy.org/en/rel_1_1/orm/persistence_techniques.html" \l "bulk-operations) ****before using this method, and fully test and confirm the functionality of all code developed using these systems.****

****在使用此方法之前，请阅读批量操作中的注意事项列表，并全面测试并确认使用这些系统开发的所有代码的功能。****

|  |  |
| --- | --- |
| **Parameters:** | * ****mapper**** – a mapped class, or the actual [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) object, representing the single kind of object represented within the mapping list. * ****mappings**** – a list of dictionaries, each one containing the state of the mapped row to be inserted, in terms of the attribute names on the mapped class. If the mapping refers to multiple tables, such as a joined-inheritance mapping, each dictionary must contain all keys to be populated into all tables. * ****return\_defaults**** – when True, rows that are missing values which generate defaults, namely integer primary key defaults and sequences, will be inserted ****one at a time****, so that the primary key value is available. In particular this will allow joined-inheritance and other multi-table mappings to insert correctly without the need to provide primary key values ahead of time; however, [Session.bulk\_insert\_mappings.return\_defaults](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.bulk_insert_mappings.params.return_defaults" \o "sqlalchemy.orm.session.Session.bulk_insert_mappings)****greatly reduces the performance gains**** of the method overall. If the rows to be inserted only refer to a single table, then there is no reason this flag should be set as the returned default information is not used. * ****render\_nulls –****When True, a value of None will result in a NULL value being included in the INSERT statement, rather than the column being omitted from the INSERT. This allows all the rows being INSERTed to have the identical set of columns which allows the full set of rows to be batched to the DBAPI. Normally, each column-set that contains a different combination of NULL values than the previous row must omit a different series of columns from the rendered INSERT statement, which means it must be emitted as a separate statement. By passing this flag, the full set of rows are guaranteed to be batchable into one batch; the cost however is that server-side defaults which are invoked by an omitted column will be skipped, so care must be taken to ensure that these are not necessary.   当为True时，值为None将导致INSERT语句中包含NULL值，而不是从INSERT中省略列。 这允许INSERTed的所有行具有相同的列集合，允许将完整的行集合批量到DBAPI。 通常，包含与上一行的NULL值不同的组合的每个列集必须从渲染的INSERT语句中省略不同的列，这意味着它必须作为单独的语句发出。 通过传递该标志，保证一整行的批处理完整的一行; 然而，成本是忽略由省略的列调用的服务器端默认值，因此必须注意确保这些不是必需的。  **Warning**  When this flag is set, ****server side default SQL values will not be invoked**** for those columns that are inserted as NULL; the NULL value will be sent explicitly. Care must be taken to ensure that no server-side default functions need to be invoked for the operation as a whole.  设置此标志时，不会为插入为NULL的列调用服务器端的默认SQL值; NULL值将被明确发送。 必须注意确保不需要为整个操作调用服务器端默认功能。  *New in version 1.1.* |

**See also**

[Bulk Operations](http://docs.sqlalchemy.org/en/rel_1_1/orm/persistence_techniques.html" \l "bulk-operations)

[Session.bulk\_save\_objects()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.bulk_save_objects" \o "sqlalchemy.orm.session.Session.bulk_save_objects)

[Session.bulk\_update\_mappings()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.bulk_update_mappings" \o "sqlalchemy.orm.session.Session.bulk_update_mappings)

**bulk\_save\_objects**(*objects*, *return\_defaults=False*, *update\_changed\_only=True*)

Perform a bulk save of the given list of objects.

执行给定的对象列表的批量保存。

The bulk save feature allows mapped objects to be used as the source of simple INSERT and UPDATE operations which can be more easily grouped together into higher performing "executemany" operations; the extraction of data from the objects is also performed using a lower-latency process that ignores whether or not attributes have actually been modified in the case of UPDATEs, and also ignores SQL expressions.

批量保存功能允许将映射对象用作简单的INSERT和UPDATE操作的源，这些操作可以更容易地分组到更高性能的"执行"操作中; 也可以使用较低延迟的进程来执行数据从对象的提取，忽略在UPDATE的情况下属性是否已被实际修改，并且也忽略SQL表达式。

The objects as given are not added to the session and no additional state is established on them, unless the return\_defaults flag is also set, in which case primary key attributes and server-side default values will be populated.

给定的对象不会添加到会话中，并且不会在其上建立附加状态，除非还设置了return\_defaults标志，在这种情况下将填充主键属性和服务器端默认值。

*New in version 1.0.0.*

**Warning**

The bulk save feature allows for a lower-latency INSERT/UPDATE of rows at the expense of most other unit-of-work features. Features such as object management, relationship handling, and SQL clause support are ****silently omitted**** in favor of raw INSERT/UPDATES of records.

****批量保存功能允许以牺牲大多数其他工作单元功能为代价的较低延迟的INSERT / UPDATE行。 静态地省略了对象管理，关系处理和SQL子句支持等功能，有利于原始的INSERT / UPDATES记录。****

****Please read the list of caveats at**** [Bulk Operations](http://docs.sqlalchemy.org/en/rel_1_1/orm/persistence_techniques.html" \l "bulk-operations) ****before using this method, and fully test and confirm the functionality of all code developed using these systems.****

****在使用此方法之前，请阅读批量操作中的注意事项列表，并全面测试并确认使用这些系统开发的所有代码的功能。****

|  |  |
| --- | --- |
| **Parameters:** | * ****objects –****a list of mapped object instances. The mapped objects are persisted as is, and are ****not**** associated with the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) afterwards.   For each object, whether the object is sent as an INSERT or an UPDATE is dependent on the same rules used by the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) in traditional operation; if the object has the InstanceState.key attribute set, then the object is assumed to be "detached" and will result in an UPDATE. Otherwise, an INSERT is used.  In the case of an UPDATE, statements are grouped based on which attributes have changed, and are thus to be the subject of each SET clause. If update\_changed\_only is False, then all attributes present within each object are applied to the UPDATE statement, which may help in allowing the statements to be grouped together into a larger executemany(), and will also reduce the overhead of checking history on attributes.   * ****return\_defaults**** – when True, rows that are missing values which generate defaults, namely integer primary key defaults and sequences, will be inserted ****one at a time****, so that the primary key value is available. In particular this will allow joined-inheritance and other multi-table mappings to insert correctly without the need to provide primary key values ahead of time; however, [Session.bulk\_save\_objects.return\_defaults](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.bulk_save_objects.params.return_defaults" \o "sqlalchemy.orm.session.Session.bulk_save_objects) ****greatly reduces the performance gains**** of the method overall. * ****update\_changed\_only**** – when True, UPDATE statements are rendered based on those attributes in each state that have logged changes. When False, all attributes present are rendered into the SET clause with the exception of primary key attributes. |

**See also**

[Bulk Operations](http://docs.sqlalchemy.org/en/rel_1_1/orm/persistence_techniques.html" \l "bulk-operations)

[Session.bulk\_insert\_mappings()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.bulk_insert_mappings" \o "sqlalchemy.orm.session.Session.bulk_insert_mappings)

[Session.bulk\_update\_mappings()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.bulk_update_mappings" \o "sqlalchemy.orm.session.Session.bulk_update_mappings)

**bulk\_update\_mappings**(*mapper*, *mappings*)

Perform a bulk update of the given list of mapping dictionaries.

对给定的映射字典列表执行批量更新。

The bulk update feature allows plain Python dictionaries to be used as the source of simple UPDATE operations which can be more easily grouped together into higher performing "executemany" operations. Using dictionaries, there is no "history" or session state management features in use, reducing latency when updating large numbers of simple rows.

批量更新功能允许使用简单的Python字典作为简单UPDATE操作的源，它可以更容易地分组到更高性能的"执行"操作中。 使用字典，没有使用"历史"或会话状态管理功能，从而在更新大量简单行时减少延迟。

*New in version 1.0.0.*

**Warning**

The bulk update feature allows for a lower-latency UPDATE of rows at the expense of most other unit-of-work features. Features such as object management, relationship handling, and SQL clause support are ****silently omitted**** in favor of raw UPDATES of records.

****批量更新功能允许以更多的其他工作单元功能为代价来降低行延迟UPDATE。 静态地省略了对象管理，关系处理和SQL子句支持等特性，有利于记录的原始UPDATES。****

****Please read the list of caveats at**** [Bulk Operations](http://docs.sqlalchemy.org/en/rel_1_1/orm/persistence_techniques.html" \l "bulk-operations) ****before using this method, and fully test and confirm the functionality of all code developed using these systems.****

****在使用此方法之前，请阅读批量操作中的注意事项列表，并全面测试并确认使用这些系统开发的所有代码的功能。****

|  |  |
| --- | --- |
| **Parameters:** | * ****mapper**** – a mapped class, or the actual [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) object, representing the single kind of object represented within the mapping list.映射类或实际的Mapper对象，表示映射列表中表示的单一对象类型。 * ****mappings**** – a list of dictionaries, each one containing the state of the mapped row to be updated, in terms of the attribute names on the mapped class. If the mapping refers to multiple tables, such as a joined-inheritance mapping, each dictionary may contain keys corresponding to all tables. All those keys which are present and are not part of the primary key are applied to the SET clause of the UPDATE statement; the primary key values, which are required, are applied to the WHERE clause.根据映射类的属性名称，列出每个字典，每个字典包含要更新的映射行的状态。 如果映射引用多个表，例如连接继承映射，则每个字典都可能包含与所有表相对应的键。 所有存在且不属于主键的键都将应用于UPDATE语句的SET子句; 所需的主键值将应用于WHERE子句。 |

**See also**

[Bulk Operations](http://docs.sqlalchemy.org/en/rel_1_1/orm/persistence_techniques.html" \l "bulk-operations)

[Session.bulk\_insert\_mappings()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.bulk_insert_mappings" \o "sqlalchemy.orm.session.Session.bulk_insert_mappings)

[Session.bulk\_save\_objects()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.bulk_save_objects" \o "sqlalchemy.orm.session.Session.bulk_save_objects)

**close**()

Close this Session.

关闭此会话。

This clears all items and ends any transaction in progress.

这将清除所有项目并结束正在进行的任何事务。

If this session were created with autocommit=False, a new transaction is immediately begun. Note that this new transaction does not use any connection resources until they are first needed.

如果此会话使用autocommit = False创建，则立即开始新的事务。 请注意，此新事务在首次需要之前不使用任何连接资源。

**close\_all**()

*inherited from the* close\_all() *method of* \_SessionClassMethods

Close *all* sessions in memory.

**commit**()

Flush pending changes and commit the current transaction.

刷新挂起的更改并提交当前事务。

If no transaction is in progress, this method raises an [InvalidRequestError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.InvalidRequestError" \o "sqlalchemy.exc.InvalidRequestError).

如果没有事务正在进行中，此方法会引发一个InvalidRequestError。

By default, the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) also expires all database loaded state on all ORM-managed attributes after transaction commit. This so that subsequent operations load the most recent data from the database. This behavior can be disabled using the expire\_on\_commit=False option to [sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker) or the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) constructor.

默认情况下，会话还将在事务提交之后的所有ORM管理的属性上过期所有数据库加载状态。 这使得后续操作从数据库加载最新的数据。 可以使用expire\_on\_commit = False选项对sessionmaker或Session构造函数禁用此行为。

If a subtransaction is in effect (which occurs when begin() is called multiple times), the subtransaction will be closed, and the next call to commit() will operate on the enclosing transaction.

如果子事务处理有效(当多次调用begin()时)，则子事务将被关闭，下一次对commit()的调用将对封闭事务进行操作。

When using the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) in its default mode of autocommit=False, a new transaction will be begun immediately after the commit, but note that the newly begun transaction does *not* use any connection resources until the first SQL is actually emitted.

当以默认模式autocommit = False使用Session时，提交后将立即开始一个新事务，但请注意，在第一个SQL实际发出之前，新开始的事务不会使用任何连接资源。

**See also**

[Committing](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_basics.html" \l "session-committing)

**connection**(*mapper=None*, *clause=None*, *bind=None*, *close\_with\_result=False*, *execution\_options=None*, *\*\*kw*)

Return a [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) object corresponding to this [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) object's transactional state.

返回与此Session对象的事务状态相对应的Connection对象。

If this [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) is configured with autocommit=False, either the [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) corresponding to the current transaction is returned, or if no transaction is in progress, a new one is begun and the [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) returned (note that no transactional state is established with the DBAPI until the first SQL statement is emitted).

如果此会话配置为autocommit = False，则返回与当前事务相对应的Connection，或者如果没有事务正在进行，则会启动一个新事务，并返回Connection(请注意，不会与DBAPI建立事务状态，直到 第一个SQL语句被发出)。

Alternatively, if this [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) is configured with autocommit=True, an ad-hoc [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) is returned using [Engine.contextual\_connect()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.contextual_connect" \o "sqlalchemy.engine.Engine.contextual_connect)on the underlying [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine).

或者，如果此会话配置为autocommit = True，则使用底层引擎上的Engine.contextual\_connect()返回一个ad-hoc连接。

Ambiguity in multi-bind or unbound [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) objects can be resolved through any of the optional keyword arguments. This ultimately makes usage of the [get\_bind()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.get_bind" \o "sqlalchemy.orm.session.Session.get_bind) method for resolution.

多绑定或未绑定会话对象中的歧义可以通过任何可选关键字参数来解决。 这最终使得get\_bind()方法用于解析。

|  |  |
| --- | --- |
| **Parameters:** | * ****bind**** – Optional [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) to be used as the bind. If this engine is already involved in an ongoing transaction, that connection will be used. This argument takes precedence over mapper, clause. * ****mapper**** – Optional [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper) mapped class, used to identify the appropriate bind. This argument takes precedence over clause. * ****clause**** – A [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement) (i.e. [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select), [text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text), etc.) which will be used to locate a bind, if a bind cannot otherwise be identified. * ****close\_with\_result**** – Passed to [Engine.connect()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.connect" \o "sqlalchemy.engine.Engine.connect), indicating the [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) should be considered "single use", automatically closing when the first result set is closed. This flag only has an effect if this [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) is configured with autocommit=True and does not already have a transaction in progress. * ****execution\_options –****a dictionary of execution options that will be passed to [Connection.execution\_options()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.execution_options" \o "sqlalchemy.engine.Connection.execution_options), ****when the connection is first procured only****. If the connection is already present within the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session), a warning is emitted and the arguments are ignored.   *New in version 0.9.9.*  **See also**  [Setting Transaction Isolation Levels](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_transaction.html" \l "session-transaction-isolation)   * ****\*\*kw**** – Additional keyword arguments are sent to [get\_bind()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.get_bind" \o "sqlalchemy.orm.session.Session.get_bind), allowing additional arguments to be passed to custom implementations of [get\_bind()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.get_bind" \o "sqlalchemy.orm.session.Session.get_bind). |

**delete**(*instance*)

Mark an instance as deleted.

The database delete operation occurs upon flush().

**deleted**

The set of all instances marked as 'deleted' within this Session

**dirty**

The set of all persistent instances considered dirty.

E.g.:

some\_mapped\_object **in** session.dirty

Instances are considered dirty when they were modified but not deleted.

Note that this 'dirty' calculation is 'optimistic'; most attribute-setting or collection modification operations will mark an instance as 'dirty' and place it in this set, even if there is no net change to the attribute's value. At flush time, the value of each attribute is compared to its previously saved value, and if there's no net change, no SQL operation will occur (this is a more expensive operation so it's only done at flush time).

To check if an instance has actionable net changes to its attributes, use the [Session.is\_modified()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.is_modified" \o "sqlalchemy.orm.session.Session.is_modified) method.

**enable\_relationship\_loading**(*obj*)

Associate an object with this [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) for related object loading.

**Warning**

[enable\_relationship\_loading()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.enable_relationship_loading" \o "sqlalchemy.orm.session.Session.enable_relationship_loading) exists to serve special use cases and is not recommended for general use.

Accesses of attributes mapped with [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) will attempt to load a value from the database using this [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) as the source of connectivity. The values will be loaded based on foreign key values present on this object - it follows that this functionality generally only works for many-to-one-relationships.

The object will be attached to this session, but will ****not**** participate in any persistence operations; its state for almost all purposes will remain either "transient" or "detached", except for the case of relationship loading.

Also note that backrefs will often not work as expected. Altering a relationship-bound attribute on the target object may not fire off a backref event, if the effective value is what was already loaded from a foreign-key-holding value.

The [Session.enable\_relationship\_loading()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.enable_relationship_loading" \o "sqlalchemy.orm.session.Session.enable_relationship_loading) method is similar to the load\_on\_pending flag on [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship). Unlike that flag, [Session.enable\_relationship\_loading()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.enable_relationship_loading" \o "sqlalchemy.orm.session.Session.enable_relationship_loading) allows an object to remain transient while still being able to load related items.

To make a transient object associated with a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) via [Session.enable\_relationship\_loading()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.enable_relationship_loading" \o "sqlalchemy.orm.session.Session.enable_relationship_loading) pending, add it to the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) using [Session.add()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.add" \o "sqlalchemy.orm.session.Session.add) normally.

[Session.enable\_relationship\_loading()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.enable_relationship_loading" \o "sqlalchemy.orm.session.Session.enable_relationship_loading) does not improve behavior when the ORM is used normally - object references should be constructed at the object level, not at the foreign key level, so that they are present in an ordinary way before flush() proceeds. This method is not intended for general use.

*New in version 0.8.*

**See also**

load\_on\_pending at [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) - this flag allows per-relationship loading of many-to-ones on items that are pending.

**execute**(*clause*, *params=None*, *mapper=None*, *bind=None*, *\*\*kw*)

Execute a SQL expression construct or string statement within the current transaction.

Returns a [ResultProxy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy" \o "sqlalchemy.engine.ResultProxy) representing results of the statement execution, in the same manner as that of an [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) or [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection).

E.g.:

result = session.execute(

user\_table.select().where(user\_table.c.id == 5)

)

[execute()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.execute" \o "sqlalchemy.orm.session.Session.execute) accepts any executable clause construct, such as [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select), [insert()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.insert" \o "sqlalchemy.sql.expression.insert), [update()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.update" \o "sqlalchemy.sql.expression.update), [delete()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.delete" \o "sqlalchemy.sql.expression.delete), and [text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text). Plain SQL strings can be passed as well, which in the case of [Session.execute()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.execute" \o "sqlalchemy.orm.session.Session.execute) only will be interpreted the same as if it were passed via a [text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text) construct. That is, the following usage:

result = session.execute(

"SELECT \* FROM user WHERE id=:param",

{"param":5}

)

is equivalent to:

**from** **sqlalchemy** **import** textresult = session.execute(

text("SELECT \* FROM user WHERE id=:param"),

{"param":5}

)

The second positional argument to [Session.execute()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.execute" \o "sqlalchemy.orm.session.Session.execute) is an optional parameter set. Similar to that of [Connection.execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.execute" \o "sqlalchemy.engine.Connection.execute), whether this is passed as a single dictionary, or a list of dictionaries, determines whether the DBAPI cursor's execute() or executemany() is used to execute the statement. An INSERT construct may be invoked for a single row:

result = session.execute(

users.insert(), {"id": 7, "name": "somename"})

or for multiple rows:

result = session.execute(users.insert(), [

{"id": 7, "name": "somename7"},

{"id": 8, "name": "somename8"},

{"id": 9, "name": "somename9"}

])

The statement is executed within the current transactional context of this [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session). The [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) which is used to execute the statement can also be acquired directly by calling the [Session.connection()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.connection" \o "sqlalchemy.orm.session.Session.connection) method. Both methods use a rule-based resolution scheme in order to determine the[Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection), which in the average case is derived directly from the "bind" of the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) itself, and in other cases can be based on the [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper) and [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects passed to the method; see the documentation for [Session.get\_bind()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.get_bind" \o "sqlalchemy.orm.session.Session.get_bind) for a full description of this scheme.

The [Session.execute()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.execute" \o "sqlalchemy.orm.session.Session.execute) method does *not* invoke autoflush.

The [ResultProxy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy" \o "sqlalchemy.engine.ResultProxy) returned by the [Session.execute()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.execute" \o "sqlalchemy.orm.session.Session.execute) method is returned with the "close\_with\_result" flag set to true; the significance of this flag is that if this [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) is autocommitting and does not have a transaction-dedicated [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) available, a temporary [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) is established for the statement execution, which is closed (meaning, returned to the connection pool) when the [ResultProxy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy" \o "sqlalchemy.engine.ResultProxy) has consumed all available data. This applies *only* when the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) is configured with autocommit=True and no transaction has been started.

|  |  |
| --- | --- |
| **Parameters:** | * ****clause**** – An executable statement (i.e. an [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable) expression such as [expression.select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select)) or string SQL statement to be executed. * ****params**** – Optional dictionary, or list of dictionaries, containing bound parameter values. If a single dictionary, single-row execution occurs; if a list of dictionaries, an "executemany" will be invoked. The keys in each dictionary must correspond to parameter names present in the statement. * ****mapper**** – Optional [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper) or mapped class, used to identify the appropriate bind. This argument takes precedence over clause when locating a bind. See [Session.get\_bind()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.get_bind" \o "sqlalchemy.orm.session.Session.get_bind) for more details. * ****bind**** – Optional [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) to be used as the bind. If this engine is already involved in an ongoing transaction, that connection will be used. This argument takes precedence over mapper and clause when locating a bind. * ****\*\*kw**** – Additional keyword arguments are sent to [Session.get\_bind()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.get_bind" \o "sqlalchemy.orm.session.Session.get_bind) to allow extensibility of "bind" schemes. |

**See also**

[SQL Expression Language Tutorial](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html) - Tutorial on using Core SQL constructs.

[Working with Engines and Connections](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html) - Further information on direct statement execution.

[Connection.execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.execute" \o "sqlalchemy.engine.Connection.execute) - core level statement execution method, which is [Session.execute()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.execute" \o "sqlalchemy.orm.session.Session.execute) ultimately uses in order to execute the statement.

**expire**(*instance*, *attribute\_names=None*)

Expire the attributes on an instance.

使实例上的属性过期。

Marks the attributes of an instance as out of date. When an expired attribute is next accessed, a query will be issued to the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) object's current transactional context in order to load all expired attributes for the given instance. Note that a highly isolated transaction will return the same values as were previously read in that same transaction, regardless of changes in database state outside of that transaction.

将实例的属性标记为过期。 当下一次访问到期属性时，将向Session对象的当前事务上下文发出一个查询，以加载给定实例的所有过期属性。 请注意，高度隔离的事务将返回与先前在同一事务中读取的值相同的值，而不管该事务之外的数据库状态发生更改。

To expire all objects in the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) simultaneously, use [Session.expire\_all()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.expire_all" \o "sqlalchemy.orm.session.Session.expire_all).

要让话中的所有对象同时过期会，请使用Session.expire\_all()。

The [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) object's default behavior is to expire all state whenever the [Session.rollback()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.rollback" \o "sqlalchemy.orm.session.Session.rollback) or [Session.commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.commit" \o "sqlalchemy.orm.session.Session.commit) methods are called, so that new state can be loaded for the new transaction. For this reason, calling [Session.expire()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.expire" \o "sqlalchemy.orm.session.Session.expire) only makes sense for the specific case that a non-ORM SQL statement was emitted in the current transaction.

Session对象的默认行为是在Session.rollback()或Session.commit()方法被调用时到期，以便可以为新事务加载新的状态。 因此，调用Session.expire()仅适用于在当前事务中发出非ORM SQL语句的具体情况。

|  |  |
| --- | --- |
| **Parameters:** | * ****instance**** – The instance to be refreshed. * ****attribute\_names**** – optional list of string attribute names indicating a subset of attributes to be expired. |

**See also**

[Refreshing / Expiring](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_state_management.html" \l "session-expire) - introductory material

[Session.expire()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.expire" \o "sqlalchemy.orm.session.Session.expire)

[Session.refresh()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.refresh" \o "sqlalchemy.orm.session.Session.refresh)

**expire\_all**()

Expires all persistent instances within this Session.

在此会话中过期所有持久化实例。

When any attributes on a persistent instance is next accessed, a query will be issued using the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) object's current transactional context in order to load all expired attributes for the given instance. Note that a highly isolated transaction will return the same values as were previously read in that same transaction, regardless of changes in database state outside of that transaction.

当下一次访问持久性实例上的任何属性时，将使用[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)对象的当前事务上下文发出查询，以加载给定实例的所有过期属性。 请注意，高度隔离的事务将返回与先前在同一事务中读取的值相同的值，而不管该事务之外的数据库状态发生更改。

To expire individual objects and individual attributes on those objects, use [Session.expire()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.expire" \o "sqlalchemy.orm.session.Session.expire).

要让这些对象上的单个对象和单个属性过期，请使用[Session.expire()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.expire" \o "sqlalchemy.orm.session.Session.expire)。

The [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) object's default behavior is to expire all state whenever the [Session.rollback()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.rollback" \o "sqlalchemy.orm.session.Session.rollback) or [Session.commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.commit" \o "sqlalchemy.orm.session.Session.commit) methods are called, so that new state can be loaded for the new transaction. For this reason, calling [Session.expire\_all()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.expire_all" \o "sqlalchemy.orm.session.Session.expire_all) should not be needed when autocommit is False, assuming the transaction is isolated.

[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)对象的默认行为是在[Session.rollback()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.rollback" \o "sqlalchemy.orm.session.Session.rollback)或[Session.commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.commit" \o "sqlalchemy.orm.session.Session.commit)方法被调用时到期，以便可以为新事务加载新的状态。 因此，假设事务是隔离的，当autocommit为False时，不需要调用[Session.expire\_all()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.expire_all" \o "sqlalchemy.orm.session.Session.expire_all)。

**See also**

[Refreshing / Expiring](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_state_management.html" \l "session-expire) - introductory material

[Session.expire()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.expire" \o "sqlalchemy.orm.session.Session.expire)

[Session.refresh()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.refresh" \o "sqlalchemy.orm.session.Session.refresh)

**expunge**(*instance*)

Remove the instance from this Session.

This will free all internal references to the instance. Cascading will be applied according to the *expunge* cascade rule.

**expunge\_all**()

Remove all object instances from this Session.

This is equivalent to calling expunge(obj) on all objects in this Session.

**flush**(*objects=None*)

Flush all the object changes to the database.

将所有对象更改刷新到数据库。

Writes out all pending object creations, deletions and modifications to the database as INSERTs, DELETEs, UPDATEs, etc. Operations are automatically ordered by the Session's unit of work dependency solver.

将所有待处理的对象创建，删除和对数据库的修改写入INSERT，DELETE，UPDATE等。操作由会话的工作单位依赖性求解程序自动排序。

Database operations will be issued in the current transactional context and do not affect the state of the transaction, unless an error occurs, in which case the entire transaction is rolled back. You may flush() as often as you like within a transaction to move changes from Python to the database's transaction buffer.

数据库操作将在当前事务上下文中发布，并且不影响事务的状态，除非发生错误，在这种情况下整个事务将回滚。 您可以在事务中随意flush() ，以将更改从Python转移到数据库的事务缓冲区。

For autocommit Sessions with no active manual transaction, flush() will create a transaction on the fly that surrounds the entire set of operations into the flush.

对于没有活动的手动事务的autocommit会话，flush()将在运行时创建一个围绕整个操作集合进行刷新的事务。

|  |  |
| --- | --- |
| **Parameters:** | ****objects**** –Optional; restricts the flush operation to operate only on elements that are in the given collection.  This feature is for an extremely narrow set of use cases where particular objects may need to be operated upon before the full flush() occurs. It is not intended for general use. |

**get\_bind**(*mapper=None*, *clause=None*)

Return a "bind" to which this [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) is bound.

The "bind" is usually an instance of [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine), except in the case where the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) has been explicitly bound directly to a [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection).

For a multiply-bound or unbound [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session), the mapper or clause arguments are used to determine the appropriate bind to return.

Note that the "mapper" argument is usually present when [Session.get\_bind()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.get_bind" \o "sqlalchemy.orm.session.Session.get_bind) is called via an ORM operation such as a [Session.query()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.query" \o "sqlalchemy.orm.session.Session.query), each individual INSERT/UPDATE/DELETE operation within a [Session.flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.flush" \o "sqlalchemy.orm.session.Session.flush), call, etc.

The order of resolution is:

1. if mapper given and session.binds is present, locate a bind based on mapper.
2. if clause given and session.binds is present, locate a bind based on [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects found in the given clause present in session.binds.
3. if session.bind is present, return that.
4. if clause given, attempt to return a bind linked to the [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) ultimately associated with the clause.
5. if mapper given, attempt to return a bind linked to the [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) ultimately associated with the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) or other selectable to which the mapper is mapped.
6. No bind can be found, [UnboundExecutionError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.UnboundExecutionError" \o "sqlalchemy.exc.UnboundExecutionError) is raised.

|  |  |
| --- | --- |
| **Parameters:** | * ****mapper**** – Optional [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper) mapped class or instance of [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper). The bind can be derived from a [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) first by consulting the "binds" map associated with this [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session), and secondly by consulting the [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) associated with the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) to which the [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) is mapped for a bind. * ****clause**** – A [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement) (i.e. [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select), [text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text), etc.). If the mapper argument is not present or could not produce a bind, the given expression construct will be searched for a bound element, typically a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) associated with bound [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData). |

**identity\_key**(*\*args*, *\*\*kwargs*)

*inherited from the* identity\_key() *method of* \_SessionClassMethods

Return an identity key.

This is an alias of [util.identity\_key()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.util.identity_key" \o "sqlalchemy.orm.util.identity_key).

**identity\_map***= None*

A mapping of object identities to objects themselves.

Iterating through Session.identity\_map.values() provides access to the full set of persistent objects (i.e., those that have row identity) currently in the session.

**See also**

[identity\_key()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.util.identity_key" \o "sqlalchemy.orm.util.identity_key) - helper function to produce the keys used in this dictionary.

**info**

A user-modifiable dictionary.

The initial value of this dictionary can be populated using the info argument to the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) constructor or [sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker) constructor or factory methods. The dictionary here is always local to this [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) and can be modified independently of all other [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) objects.

*New in version 0.9.0.*

**invalidate**()

Close this Session, using connection invalidation.

This is a variant of [Session.close()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.close" \o "sqlalchemy.orm.session.Session.close) that will additionally ensure that the [Connection.invalidate()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.invalidate" \o "sqlalchemy.engine.Connection.invalidate) method will be called on all [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection)objects. This can be called when the database is known to be in a state where the connections are no longer safe to be used.

E.g.:

**try**:

sess = Session()

sess.add(User())

sess.commit()**except** gevent.Timeout:

sess.invalidate()

**raiseexcept**:

sess.rollback()

**raise**

This clears all items and ends any transaction in progress.

If this session were created with autocommit=False, a new transaction is immediately begun. Note that this new transaction does not use any connection resources until they are first needed.

*New in version 0.9.9.*

**is\_active**

True if this [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) is in "transaction mode" and is not in "partial rollback" state.

如果此会话处于“事务模式”并且不处于“部分回滚”状态，则为真。

The [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) in its default mode of autocommit=False is essentially always in "transaction mode", in that a [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction) is associated with it as soon as it is instantiated. This [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction) is immediately replaced with a new one as soon as it is ended, due to a rollback, commit, or close operation.

autocommit= False的默认模式下的会话基本上始终处于“事务模式”，因为SessionTransaction在实例化时立即与之关联。由于回滚，提交或关闭操作，此SessionTransaction将立即被一个新的替换。

"Transaction mode" does *not* indicate whether or not actual database connection resources are in use; the [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction) object coordinates among zero or more actual database transactions, and starts out with none, accumulating individual DBAPI connections as different data sources are used within its scope. The best way to track when a particular [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) has actually begun to use DBAPI resources is to implement a listener using the [SessionEvents.after\_begin()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_begin" \o "sqlalchemy.orm.events.SessionEvents.after_begin) method, which will deliver both the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) as well as the target [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) to a user-defined event listener.

“事务模式”不表示实际的数据库连接资源是否正在使用中; SessionTransaction对象在零个或多个实际数据库事务之间进行协调，并以none开头，在其范围内使用不同的数据源时累积单个DBAPI连接。跟踪特定会话实际开始使用DBAPI资源的最佳方式是使用SessionEvents.after\_begin()方法来实现一个监听器，该方法将会将Session和目标Connection都传递给用户定义的事件侦听器。

The "partial rollback" state refers to when an "inner" transaction, typically used during a flush, encounters an error and emits a rollback of the DBAPI connection. At this point, the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) is in "partial rollback" and awaits for the user to call [Session.rollback()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.rollback" \o "sqlalchemy.orm.session.Session.rollback), in order to close out the transaction stack. It is in this "partial rollback" period that the [is\_active](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.is_active" \o "sqlalchemy.orm.session.Session.is_active) flag returns False. After the call to [Session.rollback()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.rollback" \o "sqlalchemy.orm.session.Session.rollback), the [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction)is replaced with a new one and [is\_active](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.is_active" \o "sqlalchemy.orm.session.Session.is_active) returns True again.

“部分回滚”状态是指通常在刷新期间使用的“内部”事务遇到错误并发出DBAPI连接的回滚。此时，会话处于“部分回滚”状态，等待用户调用Session.rollback()，以关闭事务堆栈。在这个“部分回滚”期间，is\_active标志返回False。在调用Session.rollback()之后，SessionTransactionis被替换为一个新的，is\_active再次返回True。

When a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) is used in autocommit=True mode, the [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction) is only instantiated within the scope of a flush call, or when [Session.begin()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.begin" \o "sqlalchemy.orm.session.Session.begin) is called. So [is\_active](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.is_active" \o "sqlalchemy.orm.session.Session.is_active) will always be False outside of a flush or [Session.begin()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.begin" \o "sqlalchemy.orm.session.Session.begin) block in this mode, and will be True within the [Session.begin()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.begin" \o "sqlalchemy.orm.session.Session.begin) block as long as it doesn't enter "partial rollback" state.

当在autocommit=True模式下使用[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)时，[SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction)只在调用调用范围内实例化，或者当调用Session.begin()时，所以在这种模式下，is\_active将会是一个flush或者Session.begin() 块之外的False，只要没有进入“partial rollback”状态，Session.begin()块就是True。

From all the above, it follows that the only purpose to this flag is for application frameworks that wish to detect is a "rollback" is necessary within a generic error handling routine, for [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) objects that would otherwise be in "partial rollback" mode. In a typical integration case, this is also not necessary as it is standard practice to emit [Session.rollback()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.rollback" \o "sqlalchemy.orm.session.Session.rollback) unconditionally within the outermost exception catch.

从上述可以看出，这个标志的唯一目的是对于希望检测的应用程序框架，在通用错误处理例程中，对于否则将处于“部分回滚”模式的Session对象来说，需要“回滚”。在典型的集成案例中，这也不是必需的，因为在最外层的异常捕获中无条件地发出Session.rollback()是标准做法。

To track the transactional state of a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) fully, use event listeners, primarily the [SessionEvents.after\_begin()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_begin" \o "sqlalchemy.orm.events.SessionEvents.after_begin),[SessionEvents.after\_commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_commit" \o "sqlalchemy.orm.events.SessionEvents.after_commit), [SessionEvents.after\_rollback()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_rollback" \o "sqlalchemy.orm.events.SessionEvents.after_rollback) and related events.

要完全跟踪会话的事务状态，请使用事件侦听器，主要是SessionEvents.after\_begin()，SessionEvents.after\_commit()，SessionEvents.after\_rollback()和相关事件。

**is\_modified**(*instance*, *include\_collections=True*, *passive=True*)

Return True if the given instance has locally modified attributes.

如果给定实例具有本地修改的属性，则返回True。

This method retrieves the history for each instrumented attribute on the instance and performs a comparison of the current value to its previously committed value, if any.

此方法将检索实例上每个检测属性的历史记录，并将当前值与之前提交的值（如果有）进行比较。

It is in effect a more expensive and accurate version of checking for the given instance in the [Session.dirty](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.dirty" \o "sqlalchemy.orm.session.Session.dirty) collection; a full test for each attribute's net "dirty" status is performed.

实际上，这是Session.dirty集合中给定实例的更昂贵和准确的检查版本; 对每个属性的“脏”状态进行全面测试。

E.g.:

**return** session.is\_modified(someobject)

*Changed in version 0.8:*When using SQLAlchemy 0.7 and earlier, the passive flag should ****always**** be explicitly set to True, else SQL loads/autoflushes may proceed which can affect the modified state itself: session.is\_modified(someobject, passive=True). In 0.8 and above, the behavior is corrected and this flag is ignored.

在版本0.8中更改：使用SQLAlchemy 0.7及更早版本时，被动标志应始终显式设置为True，否则SQL加载/自动刷新可能会进行，这可能会影响修改的状态本身：session.is\_modified（someobject，passive = True）。 在0.8及以上时，行为被纠正，并且该标志被忽略。

A few caveats to this method apply:

这个方法的一些注意事项适用于：

Instances present in the [Session.dirty](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.dirty" \o "sqlalchemy.orm.session.Session.dirty) collection may report False when tested with this method. This is because the object may have received change events via attribute mutation, thus placing it in [Session.dirty](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.dirty" \o "sqlalchemy.orm.session.Session.dirty), but ultimately the state is the same as that loaded from the database, resulting in no net change here.

[Session.dirty](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.dirty" \o "sqlalchemy.orm.session.Session.dirty)集合中的实例可以使用此方法进行测试时可能会报告False。 这是因为对象可能通过属性突变接收到更改事件，因此将其放置在[Session.dirty](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.dirty" \o "sqlalchemy.orm.session.Session.dirty)中，但最终状态与从数据库加载的状态相同，从而在此处无任何更改。

Scalar attributes may not have recorded the previously set value when a new value was applied, if the attribute was not loaded, or was expired, at the time the new value was received - in these cases, the attribute is assumed to have a change, even if there is ultimately no net change against its database value. SQLAlchemy in most cases does not need the "old" value when a set event occurs, so it skips the expense of a SQL call if the old value isn't present, based on the assumption that an UPDATE of the scalar value is usually needed, and in those few cases where it isn't, is less expensive on average than issuing a defensive SELECT.

当应用新值时，标量属性可能未记录先前设置的值，如果属性未加载或已到期，则在接收到新值时 - 在这些情况下，属性被假定为有变化， 即使最终没有对其数据库值的净变化。 在大多数情况下，SQLAlchemy在set事件发生时不需要“旧”值，因此如果旧值不存在，则跳过SQL调用的费用，这基于通常需要标量值的UPDATE ，而在少数情况下，与发出防御性SELECT相比，平均价格便宜。

The "old" value is fetched unconditionally upon set only if the attribute container has the active\_history flag set to True. This flag is set typically for primary key attributes and scalar object references that are not a simple many-to-one. To set this flag for any arbitrary mapped column, use the active\_history argument with [column\_property()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_columns.html" \l "sqlalchemy.orm.column_property" \o "sqlalchemy.orm.column_property).

只有当属性容器的active\_history标志设置为True时，才会无条件地取回“旧”值。 此标志通常用于主键属性和标量对象引用，这些引用不是一个简单的多对一。 要为任意任意映射列设置此标志，请对[column\_property()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_columns.html" \l "sqlalchemy.orm.column_property" \o "sqlalchemy.orm.column_property)使用active\_history参数。

|  |  |
| --- | --- |
| **Parameters:** | * ****instance**** – mapped instance to be tested for pending changes.映射的实例将被测试以进行未决更改。 * ****include\_collections**** – Indicates if multivalued collections should be included in the operation. Setting this to False is a way to detect only local-column based properties (i.e. scalar columns or many-to-one foreign keys) that would result in an UPDATE for this instance upon flush.指示是否应在操作中包含多值集合。 将其设置为False是一种仅检测基于本地列的属性（即标量列或多对一外键）的方法，这些属性在刷新时将导致此实例的更新。 * ****passive –****   *Changed in version 0.8:*Ignored for backwards compatibility. When using SQLAlchemy 0.7 and earlier, this flag should always be set to True.  在版本0.8中更改：忽略向后兼容性。 当使用SQLAlchemy 0.7和更早的版本时，这个标志应该总是被设置为True。 |

**merge**(*instance*, *load=True*)

Copy the state of a given instance into a corresponding instance within this [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session).

将给定实例的状态复制到此[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)内的相应实例中。

[Session.merge()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.merge" \o "sqlalchemy.orm.session.Session.merge) examines the primary key attributes of the source instance, and attempts to reconcile it with an instance of the same primary key in the session. If not found locally, it attempts to load the object from the database based on primary key, and if none can be located, creates a new instance. The state of each attribute on the source instance is then copied to the target instance. The resulting target instance is then returned by the method; the original source instance is left unmodified, and un-associated with the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) if not already.

[Session.merge()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.merge" \o "sqlalchemy.orm.session.Session.merge)检查源实例的主键属性，并尝试将其与会话中相同主键的实例进行协调。 如果在本地找不到，则会尝试根据主键从数据库加载对象，如果没有可以找到，则创建一个新实例。 然后将源实例上的每个属性的状态复制到目标实例。 然后通过该方法返回生成的目标实例; 原始源实例未修改，如果尚未与会话关联。

This operation cascades to associated instances if the association is mapped with cascade="merge".

如果关联映射为级联=“合并”，则此操作级联到关联的实例。

See [Merging](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_state_management.html" \l "unitofwork-merging) for a detailed discussion of merging.

有关合并的详细讨论，请参阅合并。

*Changed in version 1.1:*- [Session.merge()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.merge" \o "sqlalchemy.orm.session.Session.merge) will now reconcile pending objects with overlapping primary keys in the same way as persistent. See [Session.merge resolves pending conflicts the same as persistent](http://docs.sqlalchemy.org/en/rel_1_1/changelog/migration_11.html" \l "change-3601) for discussion.

版本1.1中更改： -[Session.merge()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.merge" \o "sqlalchemy.orm.session.Session.merge)现在将以与persistent相同的方式将挂起的对象与重叠的主键对齐。 请参阅Session.merge解决等待冲突与持续讨论相同。

|  |  |
| --- | --- |
| **Parameters:** | * ****instance**** – Instance to be merged.实例被合并。 * ****load –****Boolean, when False, [merge()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.merge" \o "sqlalchemy.orm.session.Session.merge) switches into a "high performance" mode which causes it to forego emitting history events as well as all database access. This flag is used for cases such as transferring graphs of objects into a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) from a second level cache, or to transfer just-loaded objects into the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) owned by a worker thread or process without re-querying the database.布尔值，当False时，merge()切换到“高性能”模式，导致它放弃发布历史事件以及所有数据库访问。该标志用于将对象的图形从第二级缓存传输到会话中，或将刚加载对象传输到由工作线程或进程拥有的会话中，而无需重新查询数据库的情况。   The load=False use case adds the caveat that the given object has to be in a "clean" state, that is, has no pending changes to be flushed - even if the incoming object is detached from any [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session). This is so that when the merge operation populates local attributes and cascades to related objects and collections, the values can be "stamped" onto the target object as is, without generating any history or attribute events, and without the need to reconcile the incoming data with any existing related objects or collections that might not be loaded. The resulting objects from load=False are always produced as "clean", so it is only appropriate that the given objects should be "clean" as well, else this suggests a mis-use of the method.  load=False用例添加了给定对象必须处于“干净”状态的警告，也就是说，即使传入的对象与任何会话分离，也不会有待刷新的待处理更改。这样，当合并操作填充本地属性并级联到相关对象和集合时，这些值可以按原样“标记”到目标对象上，而不生成任何历史记录或属性事件，并且不需要调整传入数据与任何可能无法加载的相关对象或集合。来自load=False的结果对象总是生成为“clean”，所以给定对象也应该是“clean”，否则这意味着错误地使用该方法。 |

**new**

The set of all instances marked as 'new' within this Session.

本次会议中标记为“新”的所有实例的集合。

**no\_autoflush**

Return a context manager that disables autoflush.

返回禁用自动刷新的上下文管理器。

e.g.:

**with** session.no\_autoflush:

some\_object = SomeClass()

session.add(some\_object)

*# won't autoflush*

some\_object.related\_thing = session.query(SomeRelated).first()

Operations that proceed within the with: block will not be subject to flushes occurring upon query access. This is useful when initializing a series of objects which involve existing database queries, where the uncompleted object should not yet be flushed.

在with内进行的操作：block中进行的操作在查询访问时不会发生刷新。 当初始化一系列涉及现有数据库查询的对象(其中尚未完成未完成的对象)时，这很有用。

*New in version 0.7.6.*

**object\_session**(*instance*)

*inherited from the* object\_session() *method of* \_SessionClassMethods

Return the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) to which an object belongs.

返回对象所属的会话。

This is an alias of [object\_session()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.object_session" \o "sqlalchemy.orm.session.object_session).

这是[object\_session()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.object_session" \o "sqlalchemy.orm.session.object_session)的别名。

**prepare**()

Prepare the current transaction in progress for two phase commit.

准备正在进行的两阶段提交的当前事务。

If no transaction is in progress, this method raises an [InvalidRequestError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.InvalidRequestError" \o "sqlalchemy.exc.InvalidRequestError).

如果没有事务正在进行中，此方法会引发一个[InvalidRequestError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.InvalidRequestError" \o "sqlalchemy.exc.InvalidRequestError)。

Only root transactions of two phase sessions can be prepared. If the current transaction is not such, an [InvalidRequestError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.InvalidRequestError" \o "sqlalchemy.exc.InvalidRequestError) is raised.

只能编写两阶段会话的根交易。 如果当前事务不是这样，则会引发[InvalidRequestError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.InvalidRequestError" \o "sqlalchemy.exc.InvalidRequestError)。

**~~prune~~**~~()~~

~~Remove unreferenced instances cached in the identity map.~~

~~删除身份映射中缓存的未引用的实例。~~

*~~Deprecated since version 0.7:~~*~~The non-weak-referencing identity map feature is no longer needed.~~

~~自0.7版以来不推荐使用：不再需要非弱引用身份映射功能。~~

~~Note that this method is only meaningful if "weak\_identity\_map" is set to False. The default weak identity map is self-pruning.~~

~~请注意，如果“weak\_identity\_map”设置为False，则此方法仅有意义。 默认的弱身份图是自我修剪的。~~

~~Removes any object in this Session's identity map that is not referenced in user code, modified, new or scheduled for deletion. Returns the number of objects pruned.~~

~~删除此会话的身份映射中未在用户代码中引用的任何对象，修改，新建或排除删除。 返回修剪的对象数。~~

**query**(*\*entities*, *\*\*kwargs*)

Return a new [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) object corresponding to this [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session).

返回与此Session相对应的新[Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query)对象。

**refresh**(*instance*, *attribute\_names=None*, *lockmode=None*)

Expire and refresh the attributes on the given instance.

过期并刷新给定实例上的属性。

A query will be issued to the database and all attributes will be refreshed with their current database value.

将向数据库发出查询，所有属性将使用其当前数据库值进行刷新。

Lazy-loaded relational attributes will remain lazily loaded, so that the instance-wide refresh operation will be followed immediately by the lazy load of that attribute.

延迟加载的关系属性将保持缓慢加载，以便实时范围的刷新操作将立即被该属性的延迟加载所遵循。

Eagerly-loaded relational attributes will eagerly load within the single refresh operation.

热心加载的关系属性将在单次刷新操作中加载。

Note that a highly isolated transaction will return the same values as were previously read in that same transaction, regardless of changes in database state outside of that transaction - usage of [refresh()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.refresh" \o "sqlalchemy.orm.session.Session.refresh) usually only makes sense if non-ORM SQL statement were emitted in the ongoing transaction, or if autocommit mode is turned on.

请注意，高度隔离的事务将返回与先前在同一事务中读取的值相同的值，而不管该事务之外的数据库状态如何变化 - refresh()的使用通常只有在非ORM SQL语句在 正在进行的交易，或者如果自动提交模式打开。

|  |  |
| --- | --- |
| **Parameters:** | * ****attribute\_names**** – optional. An iterable collection of string attribute names indicating a subset of attributes to be refreshed. * ****lockmode**** – Passed to the [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) as used by [with\_lockmode()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.with_lockmode" \o "sqlalchemy.orm.query.Query.with_lockmode). |

**See also**

[Refreshing / Expiring](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_state_management.html" \l "session-expire) - introductory material

[Session.expire()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.expire" \o "sqlalchemy.orm.session.Session.expire)

[Session.expire\_all()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.expire_all" \o "sqlalchemy.orm.session.Session.expire_all)

**rollback**()

Rollback the current transaction in progress.

回滚正在进行的当前事务。

If no transaction is in progress, this method is a pass-through.

如果没有交易正在进行中，则此方法是传递。

This method rolls back the current transaction or nested transaction regardless of subtransactions being in effect. All subtransactions up to the first real transaction are closed. Subtransactions occur when [begin()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.begin" \o "sqlalchemy.orm.session.Session.begin) is called multiple times.

此方法回滚当前事务或嵌套事务，而不管子事务是否生效。 到第一个真实事务的所有子事务都是关闭的。 当[begin()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.begin" \o "sqlalchemy.orm.session.Session.begin)被多次调用时，会发生子事务。

**See also**

[Rolling Back](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_basics.html" \l "session-rollback)

**scalar**(*clause*, *params=None*, *mapper=None*, *bind=None*, *\*\*kw*)

Like [execute()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.execute" \o "sqlalchemy.orm.session.Session.execute) but return a scalar result.

像execute()一样，但返回一个标量结果。

**transaction***= None*

The current active or inactive [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction).

当前活动或不活动的[SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction)。

*class*sqlalchemy.orm.session.**SessionTransaction**(*session*, *parent=None*, *nested=False*)

A [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)-level transaction.

会话级别的事务。

[SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction) is a mostly behind-the-scenes object not normally referenced directly by application code. It coordinates among multiple [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) objects, maintaining a database transaction for each one individually, committing or rolling them back all at once. It also provides optional two-phase commit behavior which can augment this coordination operation.

[SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction)是通常不被应用程序代码直接引用的大多数幕后对象。 它在多个Connection对象之间进行协调，为每个Connection对象维护一个数据库事务，一次提交或将它们全部回滚。 它还提供可选的两阶段提交行为，可以增加此协调操作。

The [Session.transaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.transaction" \o "sqlalchemy.orm.session.Session.transaction) attribute of [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) refers to the current [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction) object in use, if any. The [SessionTransaction.parent](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction.parent" \o "sqlalchemy.orm.session.SessionTransaction.parent) attribute refers to the parent [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction) in the stack of [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction) objects. If this attribute is None, then this is the top of the stack. If non-None, then this [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction) refers either to a so-called "subtransaction" or a "nested" transaction. A "subtransaction" is a scoping concept that demarcates an inner portion of the outermost "real" transaction. A nested transaction, which is indicated when the [SessionTransaction.nested](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction.nested" \o "sqlalchemy.orm.session.SessionTransaction.nested) attribute is also True, indicates that this [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction) corresponds to a SAVEPOINT.

[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)的[Session.transaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.transaction" \o "sqlalchemy.orm.session.Session.transaction)属性是指正在使用的当前[SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction)对象(如果有的话)。 [SessionTransaction.parent](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction.parent" \o "sqlalchemy.orm.session.SessionTransaction.parent)属性引用SessionTransaction对象堆栈中的父SessionTransaction。 如果此属性为None，那么这是堆栈的顶部。 如果非“无”，则此SessionTransaction指的是所谓的“子事务”或“嵌套”事务。 “子事务”是一个范围界定概念，用于划分最外层的“真实”事务的内部部分。 当SessionTransaction.nested属性也为True时指示的嵌套事务表示此SessionTransaction对应于SAVEPOINT。

****Life Cycle****

A [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction) is associated with a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) in its default mode of autocommit=False immediately, associated with no database connections. As the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) is called upon to emit SQL on behalf of various [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) or [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) objects, a corresponding [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) and associated[Transaction](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction" \o "sqlalchemy.engine.Transaction) is added to a collection within the [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction) object, becoming one of the connection/transaction pairs maintained by the[SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction). The start of a [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction) can be tracked using the [SessionEvents.after\_transaction\_create()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_transaction_create" \o "sqlalchemy.orm.events.SessionEvents.after_transaction_create) event.

SessionTransaction与其默认模式autocommit = False立即关联，与没有数据库连接相关联。 随着会话被要求代表各种Engine或Connection对象发出SQL，将对应的Connection和associatedTransaction添加到SessionTransaction对象中的集合中，成为由SessionTransaction维护的连接/事务对之一。 SessionTransaction的开始可以使用SessionEvents.after\_transaction\_create()事件来跟踪。

The lifespan of the [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction) ends when the [Session.commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.commit" \o "sqlalchemy.orm.session.Session.commit), [Session.rollback()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.rollback" \o "sqlalchemy.orm.session.Session.rollback) or [Session.close()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.close" \o "sqlalchemy.orm.session.Session.close) methods are called. At this point, the [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction) removes its association with its parent [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session). A [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) that is in autocommit=False mode will create a new [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction) to replace it immediately, whereas a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) that's in autocommit=True mode will remain without a [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction)until the [Session.begin()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.begin" \o "sqlalchemy.orm.session.Session.begin) method is called. The end of a [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction) can be tracked using the[SessionEvents.after\_transaction\_end()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_transaction_end" \o "sqlalchemy.orm.events.SessionEvents.after_transaction_end) event.

当Session.commit()，Session.rollback()或Session.close()方法被调用时，SessionTransaction的寿命结束。 此时，SessionTransaction将删除与其父Session的关联。 处于autocommit = False模式的会话将创建一个新的SessionTransaction来立即替换它，而在autocommit = True模式下的Session将保留Session.begin()方法被调用的SessionTransactionteril。 SessionTransaction的结束可以使用SessionEvents.after\_transaction\_end()事件进行跟踪。

****Nesting and Subtransactions****

Another detail of [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction) behavior is that it is capable of "nesting". This means that the [Session.begin()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.begin" \o "sqlalchemy.orm.session.Session.begin) method can be called while an existing [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction) is already present, producing a new [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction) that temporarily replaces the parent [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction). When a [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction) is produced as nested, it assigns itself to the [Session.transaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.transaction" \o "sqlalchemy.orm.session.Session.transaction) attribute, and it additionally will assign the previous [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction) to its Session.parent attribute. The behavior is effectively a stack, where [Session.transaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.transaction" \o "sqlalchemy.orm.session.Session.transaction) refers to the current head of the stack, and the [SessionTransaction.parent](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction.parent" \o "sqlalchemy.orm.session.SessionTransaction.parent) attribute allows traversal up the stack until [SessionTransaction.parent](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction.parent" \o "sqlalchemy.orm.session.SessionTransaction.parent) is None, indicating the top of the stack.

SessionTransaction行为的另一个细节是它能够“嵌套”。这意味着可以在现有的SessionTransaction已经存在的情况下调用Session.begin()方法，从而产生临时替换父SessionTransaction的新SessionTransaction。当SessionTransaction生成为嵌套时，它将自己分配给Session.transaction属性，并且还会将之前的SessionTransaction分配给其Session.parent属性。该行为实际上是一个堆栈，其中Session.transaction引用堆栈的当前头，SessionTransaction.parent属性允许遍历堆栈，直到SessionTransaction.parent为None，表示堆栈的顶部。

When the scope of [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction) is ended via [Session.commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.commit" \o "sqlalchemy.orm.session.Session.commit) or [Session.rollback()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.rollback" \o "sqlalchemy.orm.session.Session.rollback), it restores its parent [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction)back onto the [Session.transaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.transaction" \o "sqlalchemy.orm.session.Session.transaction) attribute.

当SessionTransaction的范围通过Session.commit()或Session.rollback()结束时，它将其父SessionTransactionback恢复到Session.transaction属性。

The purpose of this stack is to allow nesting of [Session.rollback()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.rollback" \o "sqlalchemy.orm.session.Session.rollback) or [Session.commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.commit" \o "sqlalchemy.orm.session.Session.commit) calls in context with various flavors of [Session.begin()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.begin" \o "sqlalchemy.orm.session.Session.begin). This nesting behavior applies to when [Session.begin\_nested()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.begin_nested" \o "sqlalchemy.orm.session.Session.begin_nested) is used to emit a SAVEPOINT transaction, and is also used to produce a so-called "subtransaction" which allows a block of code to use a begin/rollback/commit sequence regardless of whether or not its enclosing code block has begun a transaction. The [flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.flush" \o "sqlalchemy.orm.session.Session.flush) method, whether called explicitly or via autoflush, is the primary consumer of the "subtransaction" feature, in that it wishes to guarantee that it works within in a transaction block regardless of whether or not the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) is in transactional mode when the method is called.

此堆栈的目的是允许使用各种各样的[Session.commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.commit" \o "sqlalchemy.orm.session.Session.commit)方式在上下文中嵌套Session.rollback()或Session.commit()调用。这种嵌套行为适用于当Session.begin\_nested()用于发出SAVEPOINT事务时，也用于产生所谓的“子事务”，它允许代码块使用开始/回滚/提交序列，无论是否或者不是其封闭的代码块已经开始了一个事务。无论是显式地还是通过自动冲洗，flush()方法都是“子事务”功能的主要用户，因为它希望保证它在事务块中工作，而不管Session是否处于事务模式该方法被调用。

Note that the flush process that occurs within the "autoflush" feature as well as when the [Session.flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.flush" \o "sqlalchemy.orm.session.Session.flush) method is used ****always**** creates a [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction) object. This object is normally a subtransaction, unless the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) is in autocommit mode and no transaction exists at all, in which case it's the outermost transaction. Any event-handling logic or other inspection logic needs to take into account whether a [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction) is the outermost transaction, a subtransaction, or a "nested" / SAVEPOINT transaction.

请注意，在“自动冲洗”功能中以及使用Session.flush()方法时发生的刷新过程总是创建一个SessionTransaction对象。该对象通常是子事务处理，除非会话处于自动提交模式，并且完全没有事务存在，在这种情况下，它是最外层的事务。任何事件处理逻辑或其他检查逻辑都需要考虑SessionTransaction是最外层事务，子事务处理还是“嵌套”/ SAVEPOINT事务。

[Session.rollback()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.rollback" \o "sqlalchemy.orm.session.Session.rollback)

[Session.commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.commit" \o "sqlalchemy.orm.session.Session.commit)

[Session.begin()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.begin" \o "sqlalchemy.orm.session.Session.begin)

[Session.begin\_nested()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.begin_nested" \o "sqlalchemy.orm.session.Session.begin_nested)

[Session.is\_active](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.is_active" \o "sqlalchemy.orm.session.Session.is_active)

[SessionEvents.after\_transaction\_create()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_transaction_create" \o "sqlalchemy.orm.events.SessionEvents.after_transaction_create)

[SessionEvents.after\_transaction\_end()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_transaction_end" \o "sqlalchemy.orm.events.SessionEvents.after_transaction_end)

[SessionEvents.after\_commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_commit" \o "sqlalchemy.orm.events.SessionEvents.after_commit)

[SessionEvents.after\_rollback()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_rollback" \o "sqlalchemy.orm.events.SessionEvents.after_rollback)

[SessionEvents.after\_soft\_rollback()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_soft_rollback" \o "sqlalchemy.orm.events.SessionEvents.after_soft_rollback)

**nested***= False*

Indicates if this is a nested, or SAVEPOINT, transaction.

When [SessionTransaction.nested](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction.nested" \o "sqlalchemy.orm.session.SessionTransaction.nested) is True, it is expected that [SessionTransaction.parent](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction.parent" \o "sqlalchemy.orm.session.SessionTransaction.parent) will be True as well.

**parent**

The parent [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction) of this [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction).

If this attribute is None, indicates this [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction) is at the top of the stack, and corresponds to a real "COMMIT"/"ROLLBACK" block. If non-None, then this is either a "subtransaction" or a "nested" / SAVEPOINT transaction. If the [SessionTransaction.nested](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction.nested" \o "sqlalchemy.orm.session.SessionTransaction.nested) attribute is True, then this is a SAVEPOINT, and if False, indicates this a subtransaction.

*New in version 1.0.16:*- use .\_parent for previous versions

5.8.3 Session Utilities

sqlalchemy.orm.session.**make\_transient**(*instance*)

Alter the state of the given instance so that it is [transient](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-transient).

**Note**

[make\_transient()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.make_transient" \o "sqlalchemy.orm.session.make_transient) is a special-case function for advanced use cases only.

The given mapped instance is assumed to be in the [persistent](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-persistent) or [detached](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-detached) state. The function will remove its association with any [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) as well as its [InstanceState.identity](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState.identity" \o "sqlalchemy.orm.state.InstanceState.identity). The effect is that the object will behave as though it were newly constructed, except retaining any attribute / collection values that were loaded at the time of the call. The [InstanceState.deleted](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState.deleted" \o "sqlalchemy.orm.state.InstanceState.deleted) flag is also reset if this object had been deleted as a result of using[Session.delete()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.delete" \o "sqlalchemy.orm.session.Session.delete).

**Warning**

[make\_transient()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.make_transient" \o "sqlalchemy.orm.session.make_transient) does ****not**** "unexpire" or otherwise eagerly load ORM-mapped attributes that are not currently loaded at the time the function is called. This includes attributes which:

* were expired via [Session.expire()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.expire" \o "sqlalchemy.orm.session.Session.expire)
* were expired as the natural effect of committing a session transaction, e.g. [Session.commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.commit" \o "sqlalchemy.orm.session.Session.commit)
* are normally [lazy loaded](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-lazy-loaded) but are not currently loaded
* are "deferred" via [Deferred Column Loading](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "deferred) and are not yet loaded
* were not present in the query which loaded this object, such as that which is common in joined table inheritance and other scenarios.

After [make\_transient()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.make_transient" \o "sqlalchemy.orm.session.make_transient) is called, unloaded attributes such as those above will normally resolve to the value None when accessed, or an empty collection for a collection-oriented attribute. As the object is transient and un-associated with any database identity, it will no longer retrieve these values.

**See also**

[make\_transient\_to\_detached()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.make_transient_to_detached" \o "sqlalchemy.orm.session.make_transient_to_detached)

sqlalchemy.orm.session.**make\_transient\_to\_detached**(*instance*)

Make the given transient instance [detached](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-detached).

**Note**

[make\_transient\_to\_detached()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.make_transient_to_detached" \o "sqlalchemy.orm.session.make_transient_to_detached) is a special-case function for advanced use cases only.

All attribute history on the given instance will be reset as though the instance were freshly loaded from a query. Missing attributes will be marked as expired. The primary key attributes of the object, which are required, will be made into the "key" of the instance.

The object can then be added to a session, or merged possibly with the load=False flag, at which point it will look as if it were loaded that way, without emitting SQL.

This is a special use case function that differs from a normal call to [Session.merge()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.merge" \o "sqlalchemy.orm.session.Session.merge) in that a given persistent state can be manufactured without any SQL calls.

*New in version 0.9.5.*

**See also**

[make\_transient()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.make_transient" \o "sqlalchemy.orm.session.make_transient)

sqlalchemy.orm.session.**object\_session**(*instance*)

Return the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) to which the given instance belongs.

This is essentially the same as the [InstanceState.session](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState.session" \o "sqlalchemy.orm.state.InstanceState.session) accessor. See that attribute for details.

sqlalchemy.orm.util.**was\_deleted**(*object*)

Return True if the given object was deleted within a session flush.

This is regardless of whether or not the object is persistent or detached.

*New in version 0.8.0.*

**See also**

[InstanceState.was\_deleted](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState.was_deleted" \o "sqlalchemy.orm.state.InstanceState.was_deleted)

5.8.3 Attribute and State Management Utilities

These functions are provided by the SQLAlchemy attribute instrumentation API to provide a detailed interface for dealing with instances, attribute values, and history. Some of them are useful when constructing event listener functions, such as those described in [ORM Events](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html).

sqlalchemy.orm.util.**object\_state**(*instance*)

Given an object, return the [InstanceState](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState" \o "sqlalchemy.orm.state.InstanceState) associated with the object.

Raises [sqlalchemy.orm.exc.UnmappedInstanceError](http://docs.sqlalchemy.org/en/rel_1_1/orm/exceptions.html" \l "sqlalchemy.orm.exc.UnmappedInstanceError" \o "sqlalchemy.orm.exc.UnmappedInstanceError) if no mapping is configured.

Equivalent functionality is available via the [inspect()](http://docs.sqlalchemy.org/en/rel_1_1/core/inspection.html" \l "sqlalchemy.inspection.inspect" \o "sqlalchemy.inspection.inspect) function as:

inspect(instance)

Using the inspection system will raise [sqlalchemy.exc.NoInspectionAvailable](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.NoInspectionAvailable" \o "sqlalchemy.exc.NoInspectionAvailable) if the instance is not part of a mapping.

sqlalchemy.orm.attributes.**del\_attribute**(*instance*, *key*)

Delete the value of an attribute, firing history events.

This function may be used regardless of instrumentation applied directly to the class, i.e. no descriptors are required. Custom attribute management schemes will need to make usage of this method to establish attribute state as understood by SQLAlchemy.

sqlalchemy.orm.attributes.**get\_attribute**(*instance*, *key*)

Get the value of an attribute, firing any callables required.

This function may be used regardless of instrumentation applied directly to the class, i.e. no descriptors are required. Custom attribute management schemes will need to make usage of this method to make usage of attribute state as understood by SQLAlchemy.

sqlalchemy.orm.attributes.**get\_history**(*obj*, *key*, *passive=symbol('PASSIVE\_OFF')*)

Return a [History](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.attributes.History" \o "sqlalchemy.orm.attributes.History) record for the given object and attribute key.

|  |  |
| --- | --- |
| **Parameters:** | * ****obj**** – an object whose class is instrumented by the attributes package. * ****key**** – string attribute name. * ****passive**** – indicates loading behavior for the attribute if the value is not already present. This is a bitflag attribute, which defaults to the symbolPASSIVE\_OFF indicating all necessary SQL should be emitted. |

sqlalchemy.orm.attributes.**init\_collection**(*obj*, *key*)

Initialize a collection attribute and return the collection adapter.

This function is used to provide direct access to collection internals for a previously unloaded attribute. e.g.:

collection\_adapter = init\_collection(someobject, 'elements')**for** elem **in** values:

collection\_adapter.append\_without\_event(elem)

For an easier way to do the above, see [set\_committed\_value()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.attributes.set_committed_value" \o "sqlalchemy.orm.attributes.set_committed_value).

obj is an instrumented object instance. An InstanceState is accepted directly for backwards compatibility but this usage is deprecated.

sqlalchemy.orm.attributes.**flag\_modified**(*instance*, *key*)

Mark an attribute on an instance as 'modified'.

This sets the 'modified' flag on the instance and establishes an unconditional change event for the given attribute.

sqlalchemy.orm.attributes.**instance\_state**()

Return the [InstanceState](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState" \o "sqlalchemy.orm.state.InstanceState) for a given mapped object.

This function is the internal version of [object\_state()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.util.object_state" \o "sqlalchemy.orm.util.object_state). The [object\_state()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.util.object_state" \o "sqlalchemy.orm.util.object_state) and/or the [inspect()](http://docs.sqlalchemy.org/en/rel_1_1/core/inspection.html" \l "sqlalchemy.inspection.inspect" \o "sqlalchemy.inspection.inspect) function is preferred here as they each emit an informative exception if the given object is not mapped.

sqlalchemy.orm.instrumentation.**is\_instrumented**(*instance*, *key*)

Return True if the given attribute on the given instance is instrumented by the attributes package.

This function may be used regardless of instrumentation applied directly to the class, i.e. no descriptors are required.

sqlalchemy.orm.attributes.**set\_attribute**(*instance*, *key*, *value*)

Set the value of an attribute, firing history events.

This function may be used regardless of instrumentation applied directly to the class, i.e. no descriptors are required. Custom attribute management schemes will need to make usage of this method to establish attribute state as understood by SQLAlchemy.

sqlalchemy.orm.attributes.**set\_committed\_value**(*instance*, *key*, *value*)

Set the value of an attribute with no history events.

Cancels any previous history present. The value should be a scalar value for scalar-holding attributes, or an iterable for any collection-holding attribute.

This is the same underlying method used when a lazy loader fires off and loads additional data from the database. In particular, this method can be used by application code which has loaded additional attributes or collections through separate queries, which can then be attached to an instance as though it were part of its original loaded state.

*class*sqlalchemy.orm.attributes.**History**

Bases: [sqlalchemy.orm.attributes.History](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.attributes.History" \o "sqlalchemy.orm.attributes.History)

A 3-tuple of added, unchanged and deleted values, representing the changes which have occurred on an instrumented attribute.

The easiest way to get a [History](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.attributes.History" \o "sqlalchemy.orm.attributes.History) object for a particular attribute on an object is to use the [inspect()](http://docs.sqlalchemy.org/en/rel_1_1/core/inspection.html" \l "sqlalchemy.inspection.inspect" \o "sqlalchemy.inspection.inspect) function:

**from** **sqlalchemy** **import** inspect

hist = inspect(myobject).attrs.myattribute.history

Each tuple member is an iterable sequence:

* added - the collection of items added to the attribute (the first tuple element).
* unchanged - the collection of items that have not changed on the attribute (the second tuple element).
* deleted - the collection of items that have been removed from the attribute (the third tuple element).

**empty**()

Return True if this [History](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.attributes.History" \o "sqlalchemy.orm.attributes.History) has no changes and no existing, unchanged state.

**has\_changes**()

Return True if this [History](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.attributes.History" \o "sqlalchemy.orm.attributes.History) has changes.

**non\_added**()

Return a collection of unchanged + deleted.

**non\_deleted**()

Return a collection of added + unchanged.

**sum**()

Return a collection of added + unchanged + deleted.

# Chapter 6 Events and Internals

## 6.1 ORM Events

The ORM includes a wide variety of hooks available for subscription.

ORM包括可用于订阅的各种钩子。

For an introduction to the most commonly used ORM events, see the section [Tracking Object and Session Changes with Events](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_events.html). The event system in general is discussed at [Events](http://docs.sqlalchemy.org/en/rel_1_1/core/event.html). Non-ORM events such as those regarding connections and low-level statement execution are described in [Core Events](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html).

有关最常用ORM事件的介绍，请参阅跟踪对象和会话更改事件一节。 事件系统一般在事件中进行讨论。 核心事件中描述了非ORM事件，如关于连接和低级语句执行的事件。

### 6.1.1 Attribute Events

*class*sqlalchemy.orm.events.**AttributeEvents**

Bases: [sqlalchemy.event.base.Events](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.event.base.Events" \o "sqlalchemy.event.base.Events)

Define events for object attributes.

These are typically defined on the class-bound descriptor for the target class.

定义对象属性的事件。

这些通常在目标类的类绑定描述符上定义。

e.g.:

**from** **sqlalchemy** **import** event

**def** my\_append\_listener(target, value, initiator):

print "received append event for target: *%s*" % target

event.listen(MyClass.collection, 'append', my\_append\_listener)

Listeners have the option to return a possibly modified version of the value, when the retval=True flag is passed to [listen()](http://docs.sqlalchemy.org/en/rel_1_1/core/event.html" \l "sqlalchemy.event.listen" \o "sqlalchemy.event.listen):

当retval = True标志传递给listen()时，侦听器可以选择返回值的可能修改版本：

**def** validate\_phone(target, value, oldvalue, initiator):

"Strip non-numeric characters from a phone number"

**return** re.sub(r'\D', '', value)

*# setup listener on UserContact.phone attribute, instructing# it to use the return value*

listen(UserContact.phone, 'set', validate\_phone, retval=**True**)

A validation function like the above can also raise an exception such as ValueError to halt the operation.

类似上述的验证函数也可能会引发诸如ValueError之类的异常以停止该操作。

Several modifiers are available to the [listen()](http://docs.sqlalchemy.org/en/rel_1_1/core/event.html" \l "sqlalchemy.event.listen" \o "sqlalchemy.event.listen) function.

listen()函数有几个修饰符可用。

|  |  |
| --- | --- |
| **Parameters:** | * ****active\_history=False**** – When True, indicates that the "set" event would like to receive the "old" value being replaced unconditionally, even if this requires firing off database loads. Note that active\_history can also be set directly via [column\_property()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_columns.html" \l "sqlalchemy.orm.column_property" \o "sqlalchemy.orm.column_property) and [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship).当为True时，表示“set”事件想要无条件地接收“旧”值，即使这需要触发数据库加载。 请注意，active\_history也可以通过column\_property()和relationship()直接设置。 * ****propagate=False**** – When True, the listener function will be established not just for the class attribute given, but for attributes of the same name on all current subclasses of that class, as well as all future subclasses of that class, using an additional listener that listens for instrumentation events.- 当为True时，将不仅仅为给定的类属性创建监听器函数，而且将为该类的所有当前子类以及该类的所有未来子类使用相同名称的属性，使用另一个侦听器 instrumentation 事件。 * ****raw=False**** – When True, the "target" argument to the event will be the [InstanceState](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState" \o "sqlalchemy.orm.state.InstanceState) management object, rather than the mapped instance itself.当为True时，事件的“target”参数将是InstanceState管理对象，而不是映射的实例本身。 * ****retval=False**** – when True, the user-defined event listening must return the "value" argument from the function. This gives the listening function the opportunity to change the value that is ultimately used for a "set" or "append" event.当为True时，用户定义的事件侦听必须从函数返回“value”参数。 这使聆听功能有机会更改最终用于“设置”或“附加”事件的值。 |

**append**(*target*, *value*, *initiator*)

Receive a collection append event.

接收集合添加事件。

Example argument forms:

示例参数形式：

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeClass.some\_attribute, 'append')

**def** receive\_append(target, value, initiator):

"listen for the 'append' event"

*# ... (event handling logic) ...*

|  |  |
| --- | --- |
| **Parameters:** | * ****target**** – the object instance receiving the event. If the listener is registered with raw=True, this will be the [InstanceState](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState" \o "sqlalchemy.orm.state.InstanceState) object.对象实例接收事件。 如果监听器用raw = True注册，则这将是InstanceState对象。 * ****value**** – the value being appended. If this listener is registered with retval=True, the listener function must return this value, or a new value which replaces it.添加的值。 如果此监听器注册了retval = True，则侦听器函数必须返回此值或替换该值的新值。 * ****initiator –****An instance of [attributes.Event](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.attributes.Event" \o "sqlalchemy.orm.attributes.Event) representing the initiation of the event. May be modified from its original value by backref handlers in order to control chained event propagation.属性的一个实例。表示事件启动的事件。 可以通过backref处理程序从其原始值修改以控制链接事件传播。   *Changed in version 0.9.0:*the initiator argument is now passed as a [attributes.Event](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.attributes.Event" \o "sqlalchemy.orm.attributes.Event) object, and may be modified by backref handlers within a chain of backref-linked events.  在版本0.9.0中更改：initiator参数现在作为attributes.Event对象传递，并且可以由反向链接事件链中的backref处理程序修改。 |
| **Returns:** | if the event was registered with retval=True, the given value, or a new effective value, should be returned.  如果事件已注册为retval = True，则应返回给定值或新的有效值。 |

**dispose\_collection**(*target*, *collection*, *collection\_adpater*)

Receive a 'collection dispose' event.

收到“收集处理”事件。

Example argument forms:

示例参数形式：

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeClass.some\_attribute, 'dispose\_collection')

**def** receive\_dispose\_collection(target, collection, collection\_adpater):

"listen for the 'dispose\_collection' event"

*# ... (event handling logic) ...*

This event is triggered for a collection-based attribute when a collection is replaced, that is:

替换集合时，为基于集合的属性触发此事件，即：

u1.addresses.append(a1)

u1.addresses = [a2, a3] *# <- old collection is disposed*

The mechanics of the event will typically include that the given collection is empty, even if it stored objects while being replaced.

事件的机制通常包括给定的集合是空的，即使它被替换时存储对象。

*New in version 1.0.0:*the [AttributeEvents.init\_collection()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.AttributeEvents.init_collection" \o "sqlalchemy.orm.events.AttributeEvents.init_collection) and [AttributeEvents.dispose\_collection()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.AttributeEvents.dispose_collection" \o "sqlalchemy.orm.events.AttributeEvents.dispose_collection) events supersede the collection.linker hook.

**init\_collection**(*target*, *collection*, *collection\_adapter*)

Receive a 'collection init' event.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeClass.some\_attribute, 'init\_collection')

**def** receive\_init\_collection(target, collection, collection\_adapter):

"listen for the 'init\_collection' event"

*# ... (event handling logic) ...*

This event is triggered for a collection-based attribute, when the initial "empty collection" is first generated for a blank attribute, as well as for when the collection is replaced with a new one, such as via a set event.

当首先为空白属性生成初始“空集合”时，以及何时将集合替换为新集合，例如通过集合事件，此事件将触发基于集合的属性。

E.g., given that User.addresses is a relationship-based collection, the event is triggered here:

例如，由于User.addresses是一个基于关系的集合，所以在这里触发事件：

u1 = User()

u1.addresses.append(a1) *# <- new collection*

and also during replace operations:

u1.addresses = [a2, a3] *# <- new collection*

|  |  |
| --- | --- |
| **Parameters:** | * ****target**** – the object instance receiving the event. If the listener is registered with raw=True, this will be the [InstanceState](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState" \o "sqlalchemy.orm.state.InstanceState) object. * ****collection**** – the new collection. This will always be generated from what was specified as [RelationshipProperty.collection\_class](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.params.collection_class" \o "sqlalchemy.orm.properties.RelationshipProperty), and will always be empty. * ****collection\_adpater**** – the [CollectionAdapter](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "sqlalchemy.orm.collections.CollectionAdapter" \o "sqlalchemy.orm.collections.CollectionAdapter) that will mediate internal access to the collection. |

*New in version 1.0.0:*the [AttributeEvents.init\_collection()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.AttributeEvents.init_collection" \o "sqlalchemy.orm.events.AttributeEvents.init_collection) and [AttributeEvents.dispose\_collection()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.AttributeEvents.dispose_collection" \o "sqlalchemy.orm.events.AttributeEvents.dispose_collection) events supersede the collection.linker hook.

**init\_scalar**(*target*, *value*, *dict\_*)

Receive a scalar "init" event.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeClass.some\_attribute, 'init\_scalar')

**def** receive\_init\_scalar(target, value, dict\_):

"listen for the 'init\_scalar' event"

*# ... (event handling logic) ...*

This event is invoked when an uninitialized, unpersisted scalar attribute is accessed. A value of None is typically returned in this case; no changes are made to the object's state.

当访问未初始化，未持久化的标量属性时，将调用此事件。 在这种情况下通常返回值None 没有对对象的状态进行更改。

The event handler can alter this behavior in two ways. One is that a value other than None may be returned. The other is that the value may be established as part of the object's state, which will also have the effect that it is persisted.

事件处理程序可以通过两种方式改变此行为。 一个是可以返回除None以外的值。 另一个是该值可以被建立为对象状态的一部分，这也将具有持久化的作用。

Typical use is to establish a specific default value of an attribute upon access:

典型用途是在访问时建立属性的特定默认值：

SOME\_CONSTANT = 3.1415926

**@event**.listens\_for(

MyClass.some\_attribute, "init\_scalar",

retval=**True**, propagate=**True**)

**def** \_init\_some\_attribute(target, dict\_, value):

dict\_['some\_attribute'] = SOME\_CONSTANT

**return** SOME\_CONSTANT

Above, we initialize the attribute MyClass.some\_attribute to the value of SOME\_CONSTANT. The above code includes the following features:

* By setting the value SOME\_CONSTANT in the given dict\_, we indicate that the value is to be persisted to the database. ****The given value is only persisted to the database if we explicitly associate it with the object****. The dict\_ given is the \_\_dict\_\_ element of the mapped object, assuming the default attribute instrumentation system is in place.通过在给定的dict\_中设置值SOME\_CONSTANT，我们指出该值将被持久化到数据库。 如果我们明确地将其与对象关联，那么给定的值只会持久保存到数据库中。 dict\_ given是映射对象的\_\_dict\_\_元素，假设默认属性检测系统已到位。
* By establishing the retval=True flag, the value we return from the function will be returned by the attribute getter. Without this flag, the event is assumed to be a passive observer and the return value of our function is ignored.通过建立retval = True标志，我们从函数返回的值将由属性getter返回。 没有这个标志，事件被假定为一个被动观察者，我们的函数的返回值被忽略。
* The propagate=True flag is significant if the mapped class includes inheriting subclasses, which would also make use of this event listener. Without this flag, an inheriting subclass will not use our event handler.如果映射的类包含继承的子类，那么propagate = True标志是重要的，这也将利用该事件侦听器。 没有这个标志，继承的子类不会使用我们的事件处理程序。

When we establish the value in the given dictionary, the value will be used in the INSERT statement established by the unit of work. Normally, the default returned value of None is not established as part of the object, to avoid the issue of mutations occurring to the object in response to a normally passive "get" operation, and also sidesteps the issue of whether or not the [AttributeEvents.set()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.AttributeEvents.set" \o "sqlalchemy.orm.events.AttributeEvents.set) event should be awkwardly fired off during an attribute access operation. This does not impact the INSERT operation since the None value matches the value of NULL that goes into the database in any case; note that None is skipped during the INSERT to ensure that column and SQL-level default functions can fire off.

当我们在给定的字典中建立值时，该值将用于由工作单元建立的INSERT语句。通常，默认返回值None不作为对象的一部分建立，以避免响应于常规被动“get”操作对象发生突变的问题，并且还回避AttributeEvents的问题。在属性访问操作期间，set()事件应该被尴尬地关闭。这不会影响INSERT操作，因为在任何情况下，None值与进入数据库的NULL值匹配;请注意，INSERT期间无跳过，以确保列和SQL级别的默认功能可以触发。

The attribute set event [AttributeEvents.set()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.AttributeEvents.set" \o "sqlalchemy.orm.events.AttributeEvents.set) as well as the related validation feature provided by [orm.validates](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapped_attributes.html" \l "sqlalchemy.orm.validates" \o "sqlalchemy.orm.validates) is ****not**** invoked when we apply our value to the given dict\_. To have these events to invoke in response to our newly generated value, apply the value to the given object as a normal attribute set operation:

当我们将值应用于给定的dict\_时，不会调用属性集事件AttributeEvents.set()以及由orm.validates提供的相关验证特征。要使这些事件能够响应我们新生成的值进行调用，请将该值作为普通属性集操作应用于给定对象：

SOME\_CONSTANT = 3.1415926

**@event**.listens\_for(

MyClass.some\_attribute, "init\_scalar",

retval=**True**, propagate=**True**)

**def** \_init\_some\_attribute(target, dict\_, value):

*# will also fire off attribute set events*

target.some\_attribute = SOME\_CONSTANT

**return** SOME\_CONSTANT

When multiple listeners are set up, the generation of the value is "chained" from one listener to the next by passing the value returned by the previous listener that specifies retval=True as the value argument of the next listener.

当设置多个监听器时，通过传递指定retval = True的上一个监听器返回的值作为下一个监听器的值参数，将该值的生成从一个监听器“链接”到另一个监听器。

The [AttributeEvents.init\_scalar()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.AttributeEvents.init_scalar" \o "sqlalchemy.orm.events.AttributeEvents.init_scalar) event may be used to extract values from the default values and/or callables established on mapped [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects. See the "active column defaults" example in [Attribute Instrumentation](http://docs.sqlalchemy.org/en/rel_1_1/orm/examples.html" \l "examples-instrumentation) for an example of

this.

[AttributeEvents.init\_scalar()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.AttributeEvents.init_scalar" \o "sqlalchemy.orm.events.AttributeEvents.init_scalar)事件可用于从映射的[Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column)对象上建立的默认值和/或可调用中提取值。 有关示例，请参见属性仪器中的“活动列默认值”示例。

*New in version 1.1.*

|  |  |
| --- | --- |
| **Parameters:** | * ****target**** – the object instance receiving the event. If the listener is registered with raw=True, this will be the [InstanceState](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState" \o "sqlalchemy.orm.state.InstanceState) object. * ****value**** – the value that is to be returned before this event listener were invoked. This value begins as the value None, however will be the return value of the previous event handler function if multiple listeners are present. * ****dict\_**** – the attribute dictionary of this mapped object. This is normally the \_\_dict\_\_ of the object, but in all cases represents the destination that the attribute system uses to get at the actual value of this attribute. Placing the value in this dictionary has the effect that the value will be used in the INSERT statement generated by the unit of work. |

**See also**

[Attribute Instrumentation](http://docs.sqlalchemy.org/en/rel_1_1/orm/examples.html" \l "examples-instrumentation) - see the active\_column\_defaults.py example.

**remove**(*target*, *value*, *initiator*)

Receive a collection remove event.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeClass.some\_attribute, 'remove')

**def** receive\_remove(target, value, initiator):

"listen for the 'remove' event"

*# ... (event handling logic) ...*

|  |  |
| --- | --- |
| **Parameters:** | * ****target**** – the object instance receiving the event. If the listener is registered with raw=True, this will be the [InstanceState](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState" \o "sqlalchemy.orm.state.InstanceState) object. * ****value**** – the value being removed. * ****initiator –****An instance of [attributes.Event](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.attributes.Event" \o "sqlalchemy.orm.attributes.Event) representing the initiation of the event. May be modified from its original value by backref handlers in order to control chained event propagation.   *Changed in version 0.9.0:*the initiator argument is now passed as a [attributes.Event](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.attributes.Event" \o "sqlalchemy.orm.attributes.Event) object, and may be modified by backref handlers within a chain of backref-linked events. |
| **Returns:** | No return value is defined for this event. |

**set**(*target*, *value*, *oldvalue*, *initiator*)

Receive a scalar set event.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeClass.some\_attribute, 'set')

**def** receive\_set(target, value, oldvalue, initiator):

"listen for the 'set' event"

*# ... (event handling logic) ...*

*# named argument style (new in 0.9)*

**@event**.listens\_for(SomeClass.some\_attribute, 'set', named=**True**)

**def** receive\_set(\*\*kw):

"listen for the 'set' event"

target = kw['target']

value = kw['value']

*# ... (event handling logic) ...*

|  |  |
| --- | --- |
| **Parameters:** | * ****target**** – the object instance receiving the event. If the listener is registered with raw=True, this will be the [InstanceState](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState" \o "sqlalchemy.orm.state.InstanceState) object. * ****value**** – the value being set. If this listener is registered with retval=True, the listener function must return this value, or a new value which replaces it. * ****oldvalue**** – the previous value being replaced. This may also be the symbol NEVER\_SET or NO\_VALUE. If the listener is registered with active\_history=True, the previous value of the attribute will be loaded from the database if the existing value is currently unloaded or expired. * ****initiator –****An instance of [attributes.Event](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.attributes.Event" \o "sqlalchemy.orm.attributes.Event) representing the initiation of the event. May be modified from its original value by backref handlers in order to control chained event propagation.   *Changed in version 0.9.0:*the initiator argument is now passed as a [attributes.Event](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.attributes.Event" \o "sqlalchemy.orm.attributes.Event) object, and may be modified by backref handlers within a chain of backref-linked events. |
| **Returns:** | if the event was registered with retval=True, the given value, or a new effective value, should be returned. |

### 6.1.2 Mapper Events

*class*sqlalchemy.orm.events.**MapperEvents**

Bases: [sqlalchemy.event.base.Events](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.event.base.Events" \o "sqlalchemy.event.base.Events)

Define events specific to mappings.

e.g.:

**from** **sqlalchemy** **import** event

**def** my\_before\_insert\_listener(mapper, connection, target):

*# execute a stored procedure upon INSERT,*

*# apply the value to the row to be inserted*

target.calculated\_value = connection.scalar(

"select my\_special\_function(*%d*)"

% target.special\_number)

*# associate the listener function with SomeClass,# to execute during the "before\_insert" hook*

event.listen(

SomeClass, 'before\_insert', my\_before\_insert\_listener)

Available targets include:

* mapped classes
* unmapped superclasses of mapped or to-be-mapped classes (using the propagate=True flag)
* [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) objects
* the [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) class itself and the [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper) function indicate listening for all mappers.

*Changed in version 0.8.0:*mapper events can be associated with unmapped superclasses of mapped classes.

Mapper events provide hooks into critical sections of the mapper, including those related to object instrumentation, object loading, and object persistence. In particular, the persistence methods [before\_insert()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents.before_insert" \o "sqlalchemy.orm.events.MapperEvents.before_insert), and [before\_update()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents.before_update" \o "sqlalchemy.orm.events.MapperEvents.before_update) are popular places to augment the state being persisted - however, these methods operate with several significant restrictions. The user is encouraged to evaluate the [SessionEvents.before\_flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.before_flush" \o "sqlalchemy.orm.events.SessionEvents.before_flush) and [SessionEvents.after\_flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_flush" \o "sqlalchemy.orm.events.SessionEvents.after_flush) methods as more flexible and user-friendly hooks in which to apply additional database state during a flush.

Mapper事件提供了映射器的关键部分的钩子，包括与对象检测，对象加载和对象持久性相关的部分。 特别地，持久化方法[before\_insert()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents.before_insert" \o "sqlalchemy.orm.events.MapperEvents.before_insert)和[before\_update()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents.before_update" \o "sqlalchemy.orm.events.MapperEvents.before_update)是广泛使用的方法来扩展持久化的状态 - 然而，这些方法操作有几个重要的限制。 鼓励用户评估[SessionEvents.before\_flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.before_flush" \o "sqlalchemy.orm.events.SessionEvents.before_flush)和[SessionEvents.after\_flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_flush" \o "sqlalchemy.orm.events.SessionEvents.after_flush)方法作为更灵活和用户友好的钩子，以便在刷新期间应用其他数据库状态。

When using [MapperEvents](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents" \o "sqlalchemy.orm.events.MapperEvents), several modifiers are available to the [event.listen()](http://docs.sqlalchemy.org/en/rel_1_1/core/event.html" \l "sqlalchemy.event.listen" \o "sqlalchemy.event.listen) function.

当使用[MapperEvents](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents" \o "sqlalchemy.orm.events.MapperEvents)时，几个修饰符可用于[event.listen()](http://docs.sqlalchemy.org/en/rel_1_1/core/event.html" \l "sqlalchemy.event.listen" \o "sqlalchemy.event.listen)函数。

|  |  |
| --- | --- |
| **Parameters:** | * ****propagate=False**** – When True, the event listener should be applied to all inheriting mappers and/or the mappers of inheriting classes, as well as any mapper which is the target of this listener. * ****raw=False**** – When True, the "target" argument passed to applicable event listener functions will be the instance's [InstanceState](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState" \o "sqlalchemy.orm.state.InstanceState) management object, rather than the mapped instance itself. * ****retval=False –****when True, the user-defined event function must have a return value, the purpose of which is either to control subsequent event propagation, or to otherwise alter the operation in progress by the mapper. Possible return values are:   + sqlalchemy.orm.interfaces.EXT\_CONTINUE - continue event processing normally.   + sqlalchemy.orm.interfaces.EXT\_STOP - cancel all subsequent event handlers in the chain.   + other values - the return value specified by specific listeners. |

**after\_configured**()

Called after a series of mappers have been configured.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeClass, 'after\_configured')

**def** receive\_after\_configured():

"listen for the 'after\_configured' event"

*# ... (event handling logic) ...*

The [MapperEvents.after\_configured()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents.after_configured" \o "sqlalchemy.orm.events.MapperEvents.after_configured) event is invoked each time the [orm.configure\_mappers()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.configure_mappers" \o "sqlalchemy.orm.configure_mappers) function is invoked, after the function has completed its work. [orm.configure\_mappers()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.configure_mappers" \o "sqlalchemy.orm.configure_mappers) is typically invoked automatically as mappings are first used, as well as each time new mappers have been made available and new mapper use is detected.

每次调用orm.configure\_mappers()函数时，在函数完成其工作后，将调用MapperEvents.after\_configured()事件。 通常会首先使用orm.configure\_mappers()作为映射自动调用，并且每次新的映射器都可用并且检测到新的映射器使用时。

Contrast this event to the [MapperEvents.mapper\_configured()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents.mapper_configured" \o "sqlalchemy.orm.events.MapperEvents.mapper_configured) event, which is called on a per-mapper basis while the configuration operation proceeds; unlike that event, when this event is invoked, all cross-configurations (e.g. backrefs) will also have been made available for any mappers that were pending. Also contrast to [MapperEvents.before\_configured()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents.before_configured" \o "sqlalchemy.orm.events.MapperEvents.before_configured), which is invoked before the series of mappers has been configured.

将此事件与MapperEvents.mapper\_configured()事件进行对比，MapperEvents.mapper\_configured()事件在配置操作进行时以每个映射器为基础调用; 与该事件不同的是，当调用此事件时，所有交叉配置(例如backref)也将被用于任何待处理的映射器。 也与MapperEvents.before\_configured()进行对比，MapperEvents.before\_configured()在配置了一系列映射器之前被调用。

This event can ****only**** be applied to the [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) class or [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper) function, and not to individual mappings or mapped classes. It is only invoked for all mappings as a whole:

此事件只能应用于Mapper类或mapper()函数，而不能应用于单个映射或映射类。 它只对整个映射进行调用：

**from** **sqlalchemy.orm** **import** mapper

**@event**.listens\_for(mapper, "after\_configured")

**def** go():

*# ...*

Theoretically this event is called once per application, but is actually called any time new mappers have been affected by a [orm.configure\_mappers()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.configure_mappers" \o "sqlalchemy.orm.configure_mappers) call. If new mappings are constructed after existing ones have already been used, this event will likely be called again. To ensure that a particular event is only called once and no further, the once=True argument (new in 0.9.4) can be applied:

理论上，这个事件每个应用程序被调用一次，但是实际上在新的映射器受到orm.configure\_mappers()调用影响的时候实际调用。 如果在已经使用现有映射之后构建新的映射，则可能会再次调用此事件。 为了确保特定的事件只被调用一次，没有进一步的，可以应用once = True参数(0.9.4中的新值)：

**from** **sqlalchemy.orm** **import** mapper

**@event**.listens\_for(mapper, "after\_configured", once=**True**)

**def** go():

*# ...*

**See also**

[MapperEvents.mapper\_configured()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents.mapper_configured" \o "sqlalchemy.orm.events.MapperEvents.mapper_configured)

[MapperEvents.before\_configured()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents.before_configured" \o "sqlalchemy.orm.events.MapperEvents.before_configured)

**after\_delete**(*mapper*, *connection*, *target*)

Receive an object instance after a DELETE statement has been emitted corresponding to that instance.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeClass, 'after\_delete')

**def** receive\_after\_delete(mapper, connection, target):

"listen for the 'after\_delete' event"

*# ... (event handling logic) ...*

This event is used to emit additional SQL statements on the given connection as well as to perform application specific bookkeeping related to a deletion event.

The event is often called for a batch of objects of the same class after their DELETE statements have been emitted at once in a previous step.

**Warning**

Mapper-level flush events only allow ****very limited operations****, on attributes local to the row being operated upon only, as well as allowing any SQL to be emitted on the given [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection). ****Please read fully**** the notes at [Mapper-level Events](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_events.html" \l "session-persistence-mapper) for guidelines on using these methods; generally, the [SessionEvents.before\_flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.before_flush" \o "sqlalchemy.orm.events.SessionEvents.before_flush)method should be preferred for general on-flush changes.

|  |  |
| --- | --- |
| **Parameters:** | * ****mapper**** – the [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) which is the target of this event. * ****connection**** – the [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) being used to emit DELETE statements for this instance. This provides a handle into the current transaction on the target database specific to this instance. * ****target**** – the mapped instance being deleted. If the event is configured with raw=True, this will instead be the [InstanceState](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState" \o "sqlalchemy.orm.state.InstanceState) state-management object associated with the instance. |
| **Returns:** | No return value is supported by this event. |

**See also**

[Persistence Events](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_events.html" \l "session-persistence-events)

**after\_insert**(*mapper*, *connection*, *target*)

Receive an object instance after an INSERT statement is emitted corresponding to that instance.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeClass, 'after\_insert')

**def** receive\_after\_insert(mapper, connection, target):

"listen for the 'after\_insert' event"

*# ... (event handling logic) ...*

This event is used to modify in-Python-only state on the instance after an INSERT occurs, as well as to emit additional SQL statements on the given connection.

此事件用于在发生INSERT之后修改实例上仅适用于Python的状态，以及在给定连接上发出其他SQL语句。

The event is often called for a batch of objects of the same class after their INSERT statements have been emitted at once in a previous step. In the extremely rare case that this is not desirable, the [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper) can be configured with batch=False, which will cause batches of instances to be broken up into individual (and more poorly performing) event->persist->event steps.

在之前的步骤中它们的INSERT语句被一次发出之后，通常会调用该批对象的同一类对象。 在非常罕见的情况下，这是不希望的，mapper()可以配置为batch = False，这将导致批次的实例被分解为单个(以及更差的性能)事件 - > persist->事件步骤。

**Warning**

Mapper-level flush events only allow ****very limited operations****, on attributes local to the row being operated upon only, as well as allowing any SQL to be emitted on the given [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection). ****Please read fully**** the notes at [Mapper-level Events](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_events.html" \l "session-persistence-mapper) for guidelines on using these methods; generally, the [SessionEvents.before\_flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.before_flush" \o "sqlalchemy.orm.events.SessionEvents.before_flush)method should be preferred for general on-flush changes.

映射器级别刷新事件仅允许非常有限的操作，仅对行进行操作的本地属性以及允许在给定连接上发出任何SQL。 请参阅Mapper级别活动中的注意事项，了解有关使用这些方法的指导; 一般来说，SessionEvents.before\_flush()方法应该是一般的刷新更改的首选项。

|  |  |
| --- | --- |
| **Parameters:** | * ****mapper**** – the [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) which is the target of this event.该映射器是此事件的目标。 * ****connection**** – the [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) being used to emit INSERT statements for this instance. This provides a handle into the current transaction on the target database specific to this instance.Connection用于为此实例发出INSERT语句。 这为特定于此实例的目标数据库上的当前事务提供了一个句柄。 * ****target**** – the mapped instance being persisted. If the event is configured with raw=True, this will instead be the [InstanceState](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState" \o "sqlalchemy.orm.state.InstanceState) state-management object associated with the instance.映射实例被持久化。 如果事件配置为raw = True，则将与实例相关联的InstanceState状态管理对象。 |
| **Returns:** | No return value is supported by this event. |

**See also**

[Persistence Events](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_events.html" \l "session-persistence-events)

**after\_update**(*mapper*, *connection*, *target*)

Receive an object instance after an UPDATE statement is emitted corresponding to that instance.

在与该实例相对应的UPDATE语句发出后接收对象实例。

Example argument forms:

示例参数形式：

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeClass, 'after\_update')

**def** receive\_after\_update(mapper, connection, target):

"listen for the 'after\_update' event"

*# ... (event handling logic) ...*

This event is used to modify in-Python-only state on the instance after an UPDATE occurs, as well as to emit additional SQL statements on the given connection.

此事件用于在UPDATE发生后在实例上修改仅适用于Python的状态，以及在给定连接上发出附加的SQL语句。

This method is called for all instances that are marked as "dirty", *even those which have no net changes to their column-based attributes*, and for which no UPDATE statement has proceeded. An object is marked as dirty when any of its column-based attributes have a "set attribute" operation called or when any of its collections are modified. If, at update time, no column-based attributes have any net changes, no UPDATE statement will be issued. This means that an instance being sent to [after\_update()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents.after_update" \o "sqlalchemy.orm.events.MapperEvents.after_update) is *not* a guarantee that an UPDATE statement has been issued.

所有被标记为“脏”的实例被调用，即使那些没有对其基于列的属性进行任何净更改，也没有UPDATE语句已经进行的那些。当任何一个基于列的属性具有调用的“set attribute”操作或者当其任何集合被修改时，对象被标记为脏。如果在更新时，没有基于列的属性有任何净更改，则不会发出UPDATE语句。这意味着发送到after\_update()的实例不是保证UPDATE语句已经被发出的。

To detect if the column-based attributes on the object have net changes, and therefore resulted in an UPDATE statement, use object\_session(instance).is\_modified(instance, include\_collections=False).

要检测对象中基于列的属性是否具有净更改，因此导致UPDATE语句，请使用object\_session(instance).is\_modified(instance, include\_collections=False)。

The event is often called for a batch of objects of the same class after their UPDATE statements have been emitted at once in a previous step. In the extremely rare case that this is not desirable, the [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper) can be configured with batch=False, which will cause batches of instances to be broken up into individual (and more poorly performing) event->persist->event steps.

在之前的步骤中它们的UPDATE语句被一次发出之后，这个事件通常被称为同一类的一批对象。在非常罕见的情况下，这是不希望的，mapper()可以配置为batch = False，这将导致批次的实例被分解为单个(以及更差的性能)事件 - > persist->事件步骤。

**Warning**

Mapper-level flush events only allow ****very limited operations****, on attributes local to the row being operated upon only, as well as allowing any SQL to be emitted on the given [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection). ****Please read fully**** the notes at [Mapper-level Events](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_events.html" \l "session-persistence-mapper) for guidelines on using these methods; generally, the [SessionEvents.before\_flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.before_flush" \o "sqlalchemy.orm.events.SessionEvents.before_flush)method should be preferred for general on-flush changes.

映射器级别刷新事件仅允许非常有限的操作，仅对行的本地属性进行操作，以及允许在给定连接上发出任何SQL。 请参阅Mapper级别活动中的注意事项，了解有关使用这些方法的指导; 一般来说，SessionEvents.before\_flush()方法应该是一般的刷新更改的首选项。

|  |  |
| --- | --- |
| **Parameters:** | * ****mapper**** – the [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) which is the target of this event. * ****connection**** – the [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) being used to emit UPDATE statements for this instance. This provides a handle into the current transaction on the target database specific to this instance. * ****target**** – the mapped instance being persisted. If the event is configured with raw=True, this will instead be the [InstanceState](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState" \o "sqlalchemy.orm.state.InstanceState) state-management object associated with the instance. |
| **Returns:** | No return value is supported by this event. |

**See also**

[Persistence Events](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_events.html" \l "session-persistence-events)

**before\_configured**()

Called before a series of mappers have been configured.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeClass, 'before\_configured')

**def** receive\_before\_configured():

"listen for the 'before\_configured' event"

*# ... (event handling logic) ...*

The [MapperEvents.before\_configured()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents.before_configured" \o "sqlalchemy.orm.events.MapperEvents.before_configured) event is invoked each time the [orm.configure\_mappers()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.configure_mappers" \o "sqlalchemy.orm.configure_mappers) function is invoked, before the function has done any of its work. [orm.configure\_mappers()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.configure_mappers" \o "sqlalchemy.orm.configure_mappers) is typically invoked automatically as mappings are first used, as well as each time new mappers have been made available and new mapper use is detected.

This event can ****only**** be applied to the [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) class or [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper) function, and not to individual mappings or mapped classes. It is only invoked for all mappings as a whole:

**from** **sqlalchemy.orm** **import** mapper

**@event**.listens\_for(mapper, "before\_configured")

**def** go():

*# ...*

Contrast this event to [MapperEvents.after\_configured()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents.after_configured" \o "sqlalchemy.orm.events.MapperEvents.after_configured), which is invoked after the series of mappers has been configured, as well as [MapperEvents.mapper\_configured()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents.mapper_configured" \o "sqlalchemy.orm.events.MapperEvents.mapper_configured), which is invoked on a per-mapper basis as each one is configured to the extent possible.

Theoretically this event is called once per application, but is actually called any time new mappers are to be affected by a [orm.configure\_mappers()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.configure_mappers" \o "sqlalchemy.orm.configure_mappers)call. If new mappings are constructed after existing ones have already been used, this event will likely be called again. To ensure that a particular event is only called once and no further, the once=True argument (new in 0.9.4) can be applied:

**from** **sqlalchemy.orm** **import** mapper

**@event**.listens\_for(mapper, "before\_configured", once=**True**)

**def** go():

*# ...*

*New in version 0.9.3.*

**See also**

[MapperEvents.mapper\_configured()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents.mapper_configured" \o "sqlalchemy.orm.events.MapperEvents.mapper_configured)

[MapperEvents.after\_configured()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents.after_configured" \o "sqlalchemy.orm.events.MapperEvents.after_configured)

**before\_delete**(*mapper*, *connection*, *target*)

Receive an object instance before a DELETE statement is emitted corresponding to that instance.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeClass, 'before\_delete')

**def** receive\_before\_delete(mapper, connection, target):

"listen for the 'before\_delete' event"

*# ... (event handling logic) ...*

This event is used to emit additional SQL statements on the given connection as well as to perform application specific bookkeeping related to a deletion event.

The event is often called for a batch of objects of the same class before their DELETE statements are emitted at once in a later step.

**Warning**

Mapper-level flush events only allow ****very limited operations****, on attributes local to the row being operated upon only, as well as allowing any SQL to be emitted on the given [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection). ****Please read fully**** the notes at [Mapper-level Events](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_events.html" \l "session-persistence-mapper) for guidelines on using these methods; generally, the [SessionEvents.before\_flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.before_flush" \o "sqlalchemy.orm.events.SessionEvents.before_flush)method should be preferred for general on-flush changes.

|  |  |
| --- | --- |
| **Parameters:** | * ****mapper**** – the [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) which is the target of this event. * ****connection**** – the [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) being used to emit DELETE statements for this instance. This provides a handle into the current transaction on the target database specific to this instance. * ****target**** – the mapped instance being deleted. If the event is configured with raw=True, this will instead be the [InstanceState](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState" \o "sqlalchemy.orm.state.InstanceState) state-management object associated with the instance. |
| **Returns:** | No return value is supported by this event. |

**See also**

[Persistence Events](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_events.html" \l "session-persistence-events)

**before\_insert**(*mapper*, *connection*, *target*)

Receive an object instance before an INSERT statement is emitted corresponding to that instance.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeClass, 'before\_insert')

**def** receive\_before\_insert(mapper, connection, target):

"listen for the 'before\_insert' event"

*# ... (event handling logic) ...*

This event is used to modify local, non-object related attributes on the instance before an INSERT occurs, as well as to emit additional SQL statements on the given connection.

The event is often called for a batch of objects of the same class before their INSERT statements are emitted at once in a later step. In the extremely rare case that this is not desirable, the [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper) can be configured with batch=False, which will cause batches of instances to be broken up into individual (and more poorly performing) event->persist->event steps.

**Warning**

Mapper-level flush events only allow ****very limited operations****, on attributes local to the row being operated upon only, as well as allowing any SQL to be emitted on the given [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection). ****Please read fully**** the notes at [Mapper-level Events](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_events.html" \l "session-persistence-mapper) for guidelines on using these methods; generally, the [SessionEvents.before\_flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.before_flush" \o "sqlalchemy.orm.events.SessionEvents.before_flush)method should be preferred for general on-flush changes.

|  |  |
| --- | --- |
| **Parameters:** | * ****mapper**** – the [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) which is the target of this event. * ****connection**** – the [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) being used to emit INSERT statements for this instance. This provides a handle into the current transaction on the target database specific to this instance. * ****target**** – the mapped instance being persisted. If the event is configured with raw=True, this will instead be the [InstanceState](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState" \o "sqlalchemy.orm.state.InstanceState) state-management object associated with the instance. |
| **Returns:** | No return value is supported by this event. |

**See also**

[Persistence Events](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_events.html" \l "session-persistence-events)

**before\_update**(*mapper*, *connection*, *target*)

Receive an object instance before an UPDATE statement is emitted corresponding to that instance.

在对应于该实例的UPDATE语句发出之前接收对象实例。

Example argument forms:

示例参数形式：

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeClass, 'before\_update')

**def** receive\_before\_update(mapper, connection, target):

"listen for the 'before\_update' event"

*# ... (event handling logic) ...*

This event is used to modify local, non-object related attributes on the instance before an UPDATE occurs, as well as to emit additional SQL statements on the given connection.

此事件用于在UPDATE发生之前修改实例上的本地非对象相关属性，以及在给定连接上发出其他SQL语句。

This method is called for all instances that are marked as "dirty", *even those which have no net changes to their column-based attributes*. An object is marked as dirty when any of its column-based attributes have a "set attribute" operation called or when any of its collections are modified. If, at update time, no column-based attributes have any net changes, no UPDATE statement will be issued. This means that an instance being sent to [before\_update()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents.before_update" \o "sqlalchemy.orm.events.MapperEvents.before_update) is *not* a guarantee that an UPDATE statement will be issued, although you can affect the outcome here by modifying attributes so that a net change in value does exist.

对所有被标记为“脏”的实例，即使那些对其基于列的属性没有任何净更改的实例，也会调用此方法。当任何一个基于列的属性具有调用的“set attribute”操作或者当其任何集合被修改时，对象被标记为脏。如果在更新时，没有基于列的属性有任何净更改，则不会发出UPDATE语句。这意味着发送到before\_update()的实例不是保证UPDATE语句将被发出的，尽管您可以通过修改属性来影响此结果，从而确实存在值的净变化。

To detect if the column-based attributes on the object have net changes, and will therefore generate an UPDATE statement, use object\_session(instance).is\_modified(instance, include\_collections=False).

要检测对象上的基于列的属性是否具有净更改，并且将生成一个UPDATE语句，请使用object\_session(instance).is\_modified(instance, include\_collections=False)。

The event is often called for a batch of objects of the same class before their UPDATE statements are emitted at once in a later step. In the extremely rare case that this is not desirable, the [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper) can be configured with batch=False, which will cause batches of instances to be broken up into individual (and more poorly performing) event->persist->event steps.

在之后的步骤中，它们的UPDATE语句一次发出之前，通常会调用该批同一类的对象。 在非常罕见的情况下，这是不希望的，mapper()可以配置为batch = False，这将导致批次的实例被分解为单个(以及更差的性能)事件 - > persist->事件步骤。

**Warning**

Mapper-level flush events only allow ****very limited operations****, on attributes local to the row being operated upon only, as well as allowing any SQL to be emitted on the given [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection). ****Please read fully**** the notes at [Mapper-level Events](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_events.html" \l "session-persistence-mapper) for guidelines on using these methods; generally, the [SessionEvents.before\_flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.before_flush" \o "sqlalchemy.orm.events.SessionEvents.before_flush)method should be preferred for general on-flush changes.

映射器级别刷新事件仅允许非常有限的操作，仅对行的本地属性进行操作，以及允许在给定[Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection)上发出任何SQL。 请参阅Mapper级别活动中的注意事项，了解有关使用这些方法的指导; 一般来说，[SessionEvents.before\_flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.before_flush" \o "sqlalchemy.orm.events.SessionEvents.before_flush)方法应该是一般的刷新更改的首选项。

|  |  |
| --- | --- |
| **Parameters:** | * ****mapper**** – the [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) which is the target of this event. * ****connection**** – the [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) being used to emit UPDATE statements for this instance. This provides a handle into the current transaction on the target database specific to this instance. * ****target**** – the mapped instance being persisted. If the event is configured with raw=True, this will instead be the [InstanceState](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState" \o "sqlalchemy.orm.state.InstanceState) state-management object associated with the instance. |
| **Returns:** | No return value is supported by this event. |

**See also**

[Persistence Events](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_events.html" \l "session-persistence-events)

**instrument\_class**(*mapper*, *class\_*)

Receive a class when the mapper is first constructed, before instrumentation is applied to the mapped class.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeClass, 'instrument\_class')

**def** receive\_instrument\_class(mapper, class\_):

"listen for the 'instrument\_class' event"

*# ... (event handling logic) ...*

This event is the earliest phase of mapper construction. Most attributes of the mapper are not yet initialized.

This listener can either be applied to the [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) class overall, or to any un-mapped class which serves as a base for classes that will be mapped (using the propagate=True flag):

Base = declarative\_base()

**@event**.listens\_for(Base, "instrument\_class", propagate=**True**)

**def** on\_new\_class(mapper, cls\_):

" ... "

|  |  |
| --- | --- |
| **Parameters:** | * ****mapper**** – the [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) which is the target of this event. * ****class\_**** – the mapped class. |

**mapper\_configured**(*mapper*, *class\_*)

Called when a specific mapper has completed its own configuration within the scope of the [configure\_mappers()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.configure_mappers" \o "sqlalchemy.orm.configure_mappers) call.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeClass, 'mapper\_configured')

**def** receive\_mapper\_configured(mapper, class\_):

"listen for the 'mapper\_configured' event"

*# ... (event handling logic) ...*

The [MapperEvents.mapper\_configured()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents.mapper_configured" \o "sqlalchemy.orm.events.MapperEvents.mapper_configured) event is invoked for each mapper that is encountered when the [orm.configure\_mappers()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.configure_mappers" \o "sqlalchemy.orm.configure_mappers)function proceeds through the current list of not-yet-configured mappers. [orm.configure\_mappers()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.configure_mappers" \o "sqlalchemy.orm.configure_mappers) is typically invoked automatically as mappings are first used, as well as each time new mappers have been made available and new mapper use is detected.

当orm.configure\_mappers()函数通过尚未配置的映射器的当前列表时遇到的每个映射器调用MapperEvents.mapper\_configured()事件。通常会首先使用orm.configure\_mappers()作为映射自动调用，并且每次新的映射器都可用并且检测到新的映射器使用时。

When the event is called, the mapper should be in its final state, but ****not including backrefs**** that may be invoked from other mappers; they might still be pending within the configuration operation. Bidirectional relationships that are instead configured via the [orm.relationship.back\_populates](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.back_populates" \o "sqlalchemy.orm.relationship) argument *will* be fully available, since this style of relationship does not rely upon other possibly-not-configured mappers to know that they exist.

当事件被调用时，映射器应该处于其最终状态，但不包括可以从其他映射器调用的backref;它们在配置操作中可能仍然处于待处理状态。通过[orm.relationship.back\_populates](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.back_populates" \o "sqlalchemy.orm.relationship)参数配置的双向关系将完全可用，因为这种关系风格不依赖于其他可能未配置的映射器来知道它们存在。

For an event that is guaranteed to have ****all**** mappers ready to go including backrefs that are defined only on other mappings, use the [MapperEvents.after\_configured()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents.after_configured" \o "sqlalchemy.orm.events.MapperEvents.after_configured) event; this event invokes only after all known mappings have been fully configured.

对于保证所有映射器准备就绪的事件，包括仅在其他映射上定义的backref，请使用MapperEvents.after\_configured()事件;只有在所有已知映射完全配置之后，才会调用此事件。

The [MapperEvents.mapper\_configured()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents.mapper_configured" \o "sqlalchemy.orm.events.MapperEvents.mapper_configured) event, unlike [MapperEvents.before\_configured()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents.before_configured" \o "sqlalchemy.orm.events.MapperEvents.before_configured) or[MapperEvents.after\_configured()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents.after_configured" \o "sqlalchemy.orm.events.MapperEvents.after_configured), is called for each mapper/class individually, and the mapper is passed to the event itself. It also is called exactly once for a particular mapper. The event is therefore useful for configurational steps that benefit from being invoked just once on a specific mapper basis, which don't require that "backref" configurations are necessarily ready yet.

MapperEvents.mapper\_configured()事件与MapperEvents.before\_configured()或MapperEvents.after\_configured()不同，每个映射器/类分别被调用，映射器被传递给事件本身。它也被称为一个特定的映射器一次。因此，该事件对于在特定映射器基础上仅被调用一次的配置步骤是有用的，这不需要“backref”配置一定就绪。

|  |  |
| --- | --- |
| **Parameters:** | * ****mapper**** – the [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) which is the target of this event. * ****class\_**** – the mapped class. |

**See also**

[MapperEvents.before\_configured()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents.before_configured" \o "sqlalchemy.orm.events.MapperEvents.before_configured)

[MapperEvents.after\_configured()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents.after_configured" \o "sqlalchemy.orm.events.MapperEvents.after_configured)

6.1.3 Instance Events

*class*sqlalchemy.orm.events.**InstanceEvents**

Bases: [sqlalchemy.event.base.Events](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.event.base.Events" \o "sqlalchemy.event.base.Events)

Define events specific to object lifecycle.

e.g.:

**from** **sqlalchemy** **import** event

**def** my\_load\_listener(target, context):

print("on load!")

event.listen(SomeClass, 'load', my\_load\_listener)

Available targets include:

* mapped classes
* unmapped superclasses of mapped or to-be-mapped classes (using the propagate=True flag)
* [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) objects
* the [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) class itself and the [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper) function indicate listening for all mappers.

*Changed in version 0.8.0:*instance events can be associated with unmapped superclasses of mapped classes.

Instance events are closely related to mapper events, but are more specific to the instance and its instrumentation, rather than its system of persistence.

When using [InstanceEvents](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.InstanceEvents" \o "sqlalchemy.orm.events.InstanceEvents), several modifiers are available to the [event.listen()](http://docs.sqlalchemy.org/en/rel_1_1/core/event.html" \l "sqlalchemy.event.listen" \o "sqlalchemy.event.listen) function.

|  |  |
| --- | --- |
| **Parameters:** | * ****propagate=False**** – When True, the event listener should be applied to all inheriting classes as well as the class which is the target of this listener. * ****raw=False**** – When True, the "target" argument passed to applicable event listener functions will be the instance's [InstanceState](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState" \o "sqlalchemy.orm.state.InstanceState) management object, rather than the mapped instance itself. |

**expire**(*target*, *attrs*)

Receive an object instance after its attributes or some subset have been expired.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeClass, 'expire')

**def** receive\_expire(target, attrs):

"listen for the 'expire' event"

*# ... (event handling logic) ...*

'keys' is a list of attribute names. If None, the entire state was expired.

|  |  |
| --- | --- |
| **Parameters:** | * ****target**** – the mapped instance. If the event is configured with raw=True, this will instead be the [InstanceState](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState" \o "sqlalchemy.orm.state.InstanceState) state-management object associated with the instance. * ****attrs**** – sequence of attribute names which were expired, or None if all attributes were expired. |

**first\_init**(*manager*, *cls*)

Called when the first instance of a particular mapping is called.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeClass, 'first\_init')

**def** receive\_first\_init(manager, cls):

"listen for the 'first\_init' event"

*# ... (event handling logic) ...*

This event is called when the \_\_init\_\_ method of a class is called the first time for that particular class. The event invokes before \_\_init\_\_ actually proceeds as well as before the [InstanceEvents.init()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.InstanceEvents.init" \o "sqlalchemy.orm.events.InstanceEvents.init) event is invoked.

**init**(*target*, *args*, *kwargs*)

Receive an instance when its constructor is called.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeClass, 'init')

**def** receive\_init(target, args, kwargs):

"listen for the 'init' event"

*# ... (event handling logic) ...*

This method is only called during a userland construction of an object, in conjunction with the object's constructor, e.g. its \_\_init\_\_ method. It is not called when an object is loaded from the database; see the [InstanceEvents.load()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.InstanceEvents.load" \o "sqlalchemy.orm.events.InstanceEvents.load) event in order to intercept a database load.

The event is called before the actual \_\_init\_\_ constructor of the object is called. The kwargs dictionary may be modified in-place in order to affect what is passed to \_\_init\_\_.

|  |  |
| --- | --- |
| **Parameters:** | * ****target**** – the mapped instance. If the event is configured with raw=True, this will instead be the [InstanceState](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState" \o "sqlalchemy.orm.state.InstanceState) state-management object associated with the instance. * ****args**** – positional arguments passed to the \_\_init\_\_ method. This is passed as a tuple and is currently immutable. * ****kwargs**** – keyword arguments passed to the \_\_init\_\_ method. This structure *can* be altered in place. |

**See also**

[InstanceEvents.init\_failure()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.InstanceEvents.init_failure" \o "sqlalchemy.orm.events.InstanceEvents.init_failure)

[InstanceEvents.load()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.InstanceEvents.load" \o "sqlalchemy.orm.events.InstanceEvents.load)

**init\_failure**(*target*, *args*, *kwargs*)

Receive an instance when its constructor has been called, and raised an exception.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeClass, 'init\_failure')

**def** receive\_init\_failure(target, args, kwargs):

"listen for the 'init\_failure' event"

*# ... (event handling logic) ...*

This method is only called during a userland construction of an object, in conjunction with the object's constructor, e.g. its \_\_init\_\_ method. It is not called when an object is loaded from the database.

该方法仅在对象的用户空间构造期间，与对象的构造函数(例如，它的\_\_init\_\_方法。 当从数据库加载对象时，不会调用它。

The event is invoked after an exception raised by the \_\_init\_\_ method is caught. After the event is invoked, the original exception is re-raised outwards, so that the construction of the object still raises an exception. The actual exception and stack trace raised should be present in sys.exc\_info().

在\_\_init\_\_方法引发的异常被捕获后调用事件。 在事件被调用之后，原始异常被向外重新提升，使对象的构造仍然引起异常。 引发的实际异常和堆栈跟踪应该存在于sys.exc\_info()中。

|  |  |
| --- | --- |
| **Parameters:** | * ****target**** – the mapped instance. If the event is configured with raw=True, this will instead be the [InstanceState](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState" \o "sqlalchemy.orm.state.InstanceState) state-management object associated with the instance. * ****args**** – positional arguments that were passed to the \_\_init\_\_ method. * ****kwargs**** – keyword arguments that were passed to the \_\_init\_\_ method. |

**See also**

[InstanceEvents.init()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.InstanceEvents.init" \o "sqlalchemy.orm.events.InstanceEvents.init)

[InstanceEvents.load()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.InstanceEvents.load" \o "sqlalchemy.orm.events.InstanceEvents.load)

**load**(*target*, *context*)

Receive an object instance after it has been created via \_\_new\_\_, and after initial attribute population has occurred.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeClass, 'load')

**def** receive\_load(target, context):

"listen for the 'load' event"

*# ... (event handling logic) ...*

This typically occurs when the instance is created based on incoming result rows, and is only called once for that instance's lifetime.

这通常在基于传入结果行创建实例时发生，并且仅在该实例的生命周期中调用一次。

Note that during a result-row load, this method is called upon the first row received for this instance. Note that some attributes and collections may or may not be loaded or even initialized, depending on what's present in the result rows.

请注意，在结果行加载期间，此方法在为此实例接收到的第一行时调用。 请注意，根据结果行中存在的内容，某些属性和集合可能被加载也可能不会被加载甚至被初始化。

The [InstanceEvents.load()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.InstanceEvents.load" \o "sqlalchemy.orm.events.InstanceEvents.load) event is also available in a class-method decorator format called [orm.reconstructor()](http://docs.sqlalchemy.org/en/rel_1_1/orm/constructors.html" \l "sqlalchemy.orm.reconstructor" \o "sqlalchemy.orm.reconstructor).

InstanceEvents.load()事件也可以使用名为orm.reconstructor()的类方法装饰器格式。

|  |  |
| --- | --- |
| **Parameters:** | * ****target**** – the mapped instance. If the event is configured with raw=True, this will instead be the [InstanceState](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState" \o "sqlalchemy.orm.state.InstanceState) state-management object associated with the instance.映射的实例。 如果事件配置为raw = True，则将与实例相关联的InstanceState状态管理对象。 * ****context**** – the [QueryContext](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.query.QueryContext" \o "sqlalchemy.orm.query.QueryContext) corresponding to the current [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) in progress. This argument may be None if the load does not correspond to a [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query), such as during [Session.merge()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.merge" \o "sqlalchemy.orm.session.Session.merge).对应于当前Query的QueryContext正在进行中。 如果负载与Query不对应，则该参数可能为None，例如Session.merge()期间。 |

**See also**

[InstanceEvents.init()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.InstanceEvents.init" \o "sqlalchemy.orm.events.InstanceEvents.init)

[InstanceEvents.refresh()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.InstanceEvents.refresh" \o "sqlalchemy.orm.events.InstanceEvents.refresh)

[SessionEvents.loaded\_as\_persistent()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.loaded_as_persistent" \o "sqlalchemy.orm.events.SessionEvents.loaded_as_persistent)

[Constructors and Object Initialization](http://docs.sqlalchemy.org/en/rel_1_1/orm/constructors.html" \l "mapping-constructors)

**pickle**(*target*, *state\_dict*)

Receive an object instance when its associated state is being pickled.

当其关联状态被**pickle**时接收对象实例。

Example argument forms:

示例参数形式：

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeClass, 'pickle')

**def** receive\_pickle(target, state\_dict):

"listen for the 'pickle' event"

*# ... (event handling logic) ...*

|  |  |
| --- | --- |
| **Parameters:** | * ****target**** – the mapped instance. If the event is configured with raw=True, this will instead be the [InstanceState](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState" \o "sqlalchemy.orm.state.InstanceState) state-management object associated with the instance.映射的实例。 如果事件配置为raw = True，则将与实例相关联的InstanceState状态管理对象。 * ****state\_dict**** – the dictionary returned by InstanceState.\_\_getstate\_\_, containing the state to be pickled.由InstanceState返回的字典.\_\_ getstate\_\_，包含要腌制的状态。 |

**refresh**(*target*, *context*, *attrs*)

Receive an object instance after one or more attributes have been refreshed from a query.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeClass, 'refresh')

**def** receive\_refresh(target, context, attrs):

"listen for the 'refresh' event"

*# ... (event handling logic) ...*

Contrast this to the [InstanceEvents.load()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.InstanceEvents.load" \o "sqlalchemy.orm.events.InstanceEvents.load) method, which is invoked when the object is first loaded from a query.

|  |  |
| --- | --- |
| **Parameters:** | * ****target**** – the mapped instance. If the event is configured with raw=True, this will instead be the [InstanceState](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState" \o "sqlalchemy.orm.state.InstanceState) state-management object associated with the instance. * ****context**** – the [QueryContext](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.query.QueryContext" \o "sqlalchemy.orm.query.QueryContext) corresponding to the current [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) in progress. * ****attrs**** – sequence of attribute names which were populated, or None if all column-mapped, non-deferred attributes were populated. |

**See also**

[InstanceEvents.load()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.InstanceEvents.load" \o "sqlalchemy.orm.events.InstanceEvents.load)

**refresh\_flush**(*target*, *flush\_context*, *attrs*)

Receive an object instance after one or more attributes have been refreshed within the persistence of the object.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeClass, 'refresh\_flush')

**def** receive\_refresh\_flush(target, flush\_context, attrs):

"listen for the 'refresh\_flush' event"

*# ... (event handling logic) ...*

This event is the same as [InstanceEvents.refresh()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.InstanceEvents.refresh" \o "sqlalchemy.orm.events.InstanceEvents.refresh) except it is invoked within the unit of work flush process, and the values here typically come from the process of handling an INSERT or UPDATE, such as via the RETURNING clause or from Python-side default values.

此事件与InstanceEvents.refresh()相同，除了它在工作清理过程的单位内调用，这里的值通常来自处理INSERT或UPDATE的过程，例如通过RETURNING子句或从Python侧 默认值。

*New in version 1.0.5.*

|  |  |
| --- | --- |
| **Parameters:** | * ****target**** – the mapped instance. If the event is configured with raw=True, this will instead be the [InstanceState](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState" \o "sqlalchemy.orm.state.InstanceState) state-management object associated with the instance. * ****flush\_context**** – Internal [UOWTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.session.UOWTransaction" \o "sqlalchemy.orm.session.UOWTransaction) object which handles the details of the flush. * ****attrs**** – sequence of attribute names which were populated. |

**unpickle**(*target*, *state\_dict*)

Receive an object instance after its associated state has been unpickled.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeClass, 'unpickle')

**def** receive\_unpickle(target, state\_dict):

"listen for the 'unpickle' event"

*# ... (event handling logic) ...*

|  |  |
| --- | --- |
| **Parameters:** | * ****target**** – the mapped instance. If the event is configured with raw=True, this will instead be the [InstanceState](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState" \o "sqlalchemy.orm.state.InstanceState) state-management object associated with the instance.映射的实例。 如果事件配置为raw = True，则将与实例相关联的InstanceState状态管理对象。 * ****state\_dict**** – the dictionary sent to InstanceState.\_\_setstate\_\_, containing the state dictionary which was pickled.该字典发送到InstanceState .\_\_ setstate\_\_，包含被腌制的状态字典。 |

### 6.1.4 Session Events

*class*sqlalchemy.orm.events.**SessionEvents**

Bases: [sqlalchemy.event.base.Events](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.event.base.Events" \o "sqlalchemy.event.base.Events)

Define events specific to [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) lifecycle.

e.g.:

**from** **sqlalchemy** **import** event

**from** **sqlalchemy.orm** **import** sessionmaker

**def** my\_before\_commit(session):

print("before commit!")

Session = sessionmaker()

event.listen(Session, "before\_commit", my\_before\_commit)

The [listen()](http://docs.sqlalchemy.org/en/rel_1_1/core/event.html" \l "sqlalchemy.event.listen" \o "sqlalchemy.event.listen) function will accept [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) objects as well as the return result of [sessionmaker()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker) and [scoped\_session()](http://docs.sqlalchemy.org/en/rel_1_1/orm/contextual.html" \l "sqlalchemy.orm.scoping.scoped_session" \o "sqlalchemy.orm.scoping.scoped_session).

listen()函数将接受Session对象以及sessionmaker()和scoped\_session()的返回结果。

Additionally, it accepts the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) class which will apply listeners to all [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) instances globally.

此外，它接受将Class List应用于所有Session实例的Session类。

**after\_attach**(*session*, *instance*)

Execute after an instance is attached to a session.

在实例附加到会话之后执行。

Example argument forms:

示例参数形式：

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeSessionOrFactory, 'after\_attach')

**def** receive\_after\_attach(session, instance):

"listen for the 'after\_attach' event"

*# ... (event handling logic) ...*

This is called after an add, delete or merge.

这是在添加，删除或合并后调用的。

**Note**

As of 0.8, this event fires off *after* the item has been fully associated with the session, which is different than previous releases. For event handlers that require the object not yet be part of session state (such as handlers which may autoflush while the target object is not yet complete) consider the new [before\_attach()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.before_attach" \o "sqlalchemy.orm.events.SessionEvents.before_attach) event.

从0.8开始，该事件在与会话完全关联之后触发，这与以前的版本不同。 对于需要对象尚未成为会话状态的事件的处理程序(例如在目标对象尚未完成时可能自动刷新的处理程序)考虑新的before\_attach()事件。

**See also**

[before\_attach()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.before_attach" \o "sqlalchemy.orm.events.SessionEvents.before_attach)

[Object Lifecycle Events](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_events.html" \l "session-lifecycle-events)

**after\_begin**(*session*, *transaction*, *connection*)

Execute after a transaction is begun on a connection

在连接上开始事务后执行

Example argument forms:

示例参数形式：

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeSessionOrFactory, 'after\_begin')

**def** receive\_after\_begin(session, transaction, connection):

"listen for the 'after\_begin' event"

*# ... (event handling logic) ...*

|  |  |
| --- | --- |
| **Parameters:** | * ****session**** – The target [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session). * ****transaction**** – The [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction). * ****connection**** – The [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) object which will be used for SQL statements. |

**See also**

[before\_commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.before_commit" \o "sqlalchemy.orm.events.SessionEvents.before_commit)

[after\_commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_commit" \o "sqlalchemy.orm.events.SessionEvents.after_commit)

[after\_transaction\_create()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_transaction_create" \o "sqlalchemy.orm.events.SessionEvents.after_transaction_create)

[after\_transaction\_end()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_transaction_end" \o "sqlalchemy.orm.events.SessionEvents.after_transaction_end)

**after\_bulk\_delete**(*delete\_context*)

Execute after a bulk delete operation to the session.

在大量删除操作后执行会话。

Example argument forms:

示例参数形式：

**from** **sqlalchemy** **import** event

*# standard decorator style (arguments as of 0.9)*

**@event**.listens\_for(SomeSessionOrFactory, 'after\_bulk\_delete')

**def** receive\_after\_bulk\_delete(delete\_context):

"listen for the 'after\_bulk\_delete' event"

*# ... (event handling logic) ...*

*# legacy calling style (pre-0.9)*

**@event**.listens\_for(SomeSessionOrFactory, 'after\_bulk\_delete')

**def** receive\_after\_bulk\_delete(session, query, query\_context, result):

"listen for the 'after\_bulk\_delete' event"

*# ... (event handling logic) ...*

*Changed in version 0.9:*The after\_bulk\_delete event now accepts the arguments delete\_context. Listener functions which accept the previous argument signature(s) listed above will be automatically adapted to the new signature.

0.9版本更改：after\_bulk\_delete事件现在接受参数delete\_context。 接受以上列出的参数签名的侦听器功能将自动适应新的签名。

This is called as a result of the [Query.delete()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.delete" \o "sqlalchemy.orm.query.Query.delete) method.

作为Query.delete()方法的结果调用。

|  |  |
| --- | --- |
| **Parameters:** | ****delete\_context**** –a "delete context" object which contains details about the update, including these attributes:一个“删除上下文”对象，其中包含有关更新的详细信息，包括以下属性：   * session - the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) involved * query -the [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) object that this update operation was called upon. * context The [QueryContext](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.query.QueryContext" \o "sqlalchemy.orm.query.QueryContext) object, corresponding to the invocation of an ORM query. * result the [ResultProxy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy" \o "sqlalchemy.engine.ResultProxy) returned as a result of the bulk DELETE operation. |

**after\_bulk\_update**(*update\_context*)

Execute after a bulk update operation to the session.

在会话的批量更新操作后执行。

Example argument forms:

示例参数形式：

**from** **sqlalchemy** **import** event

*# standard decorator style (arguments as of 0.9)*

**@event**.listens\_for(SomeSessionOrFactory, 'after\_bulk\_update')

**def** receive\_after\_bulk\_update(update\_context):

"listen for the 'after\_bulk\_update' event"

*# ... (event handling logic) ...*

*# legacy calling style (pre-0.9)*

**@event**.listens\_for(SomeSessionOrFactory, 'after\_bulk\_update')

**def** receive\_after\_bulk\_update(session, query, query\_context, result):

"listen for the 'after\_bulk\_update' event"

*# ... (event handling logic) ...*

*Changed in version 0.9:*The after\_bulk\_update event now accepts the arguments update\_context. Listener functions which accept the previous argument signature(s) listed above will be automatically adapted to the new signature.

在版本0.9中更改：after\_bulk\_update事件现在接受参数update\_context。 接受以上列出的参数签名的侦听器功能将自动适应新的签名。

This is called as a result of the [Query.update()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.update" \o "sqlalchemy.orm.query.Query.update) method.

这是作为Query.update()方法的结果调用的。

|  |  |
| --- | --- |
| **Parameters:** | ****update\_context**** –  an "update context" object which contains details about the update, including these attributes:  一个“更新上下文”对象，其中包含有关更新的详细信息，包括以下属性：   * session - the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) involved * query -the [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) object that this update operation was called upon. * context The [QueryContext](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.query.QueryContext" \o "sqlalchemy.orm.query.QueryContext) object, corresponding to the invocation of an ORM query. * result the [ResultProxy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy" \o "sqlalchemy.engine.ResultProxy) returned as a result of the bulk UPDATE operation. |

**after\_commit**(*session*)

Execute after a commit has occurred.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeSessionOrFactory, 'after\_commit')

**def** receive\_after\_commit(session):

"listen for the 'after\_commit' event"

*# ... (event handling logic) ...*

**Note**

The [after\_commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_commit" \o "sqlalchemy.orm.events.SessionEvents.after_commit) hook is *not* per-flush, that is, the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) can emit SQL to the database many times within the scope of a transaction. For interception of these events, use the [before\_flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.before_flush" \o "sqlalchemy.orm.events.SessionEvents.before_flush), [after\_flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_flush" \o "sqlalchemy.orm.events.SessionEvents.after_flush), or [after\_flush\_postexec()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_flush_postexec" \o "sqlalchemy.orm.events.SessionEvents.after_flush_postexec) events.

**Note**

The [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) is not in an active transaction when the [after\_commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_commit" \o "sqlalchemy.orm.events.SessionEvents.after_commit) event is invoked, and therefore can not emit SQL. To emit SQL corresponding to every transaction, use the [before\_commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.before_commit" \o "sqlalchemy.orm.events.SessionEvents.before_commit) event.

|  |  |
| --- | --- |
| **Parameters:** | ****session**** – The target [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session). |

**See also**

[before\_commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.before_commit" \o "sqlalchemy.orm.events.SessionEvents.before_commit)

[after\_begin()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_begin" \o "sqlalchemy.orm.events.SessionEvents.after_begin)

[after\_transaction\_create()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_transaction_create" \o "sqlalchemy.orm.events.SessionEvents.after_transaction_create)

[after\_transaction\_end()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_transaction_end" \o "sqlalchemy.orm.events.SessionEvents.after_transaction_end)

**after\_flush**(*session*, *flush\_context*)

Execute after flush has completed, but before commit has been called.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeSessionOrFactory, 'after\_flush')

**def** receive\_after\_flush(session, flush\_context):

"listen for the 'after\_flush' event"

*# ... (event handling logic) ...*

Note that the session's state is still in pre-flush, i.e. 'new', 'dirty', and 'deleted' lists still show pre-flush state as well as the history settings on instance attributes.

|  |  |
| --- | --- |
| **Parameters:** | * ****session**** – The target [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session). * ****flush\_context**** – Internal [UOWTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.session.UOWTransaction" \o "sqlalchemy.orm.session.UOWTransaction) object which handles the details of the flush. |

**See also**

[before\_flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.before_flush" \o "sqlalchemy.orm.events.SessionEvents.before_flush)

[after\_flush\_postexec()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_flush_postexec" \o "sqlalchemy.orm.events.SessionEvents.after_flush_postexec)

[Persistence Events](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_events.html" \l "session-persistence-events)

**after\_flush\_postexec**(*session*, *flush\_context*)

Execute after flush has completed, and after the post-exec state occurs.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeSessionOrFactory, 'after\_flush\_postexec')

**def** receive\_after\_flush\_postexec(session, flush\_context):

"listen for the 'after\_flush\_postexec' event"

*# ... (event handling logic) ...*

This will be when the 'new', 'dirty', and 'deleted' lists are in their final state. An actual commit() may or may not have occurred, depending on whether or not the flush started its own transaction or participated in a larger transaction.

|  |  |
| --- | --- |
| **Parameters:** | * ****session**** – The target [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session). * ****flush\_context**** – Internal [UOWTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.session.UOWTransaction" \o "sqlalchemy.orm.session.UOWTransaction) object which handles the details of the flush. |

**See also**

[before\_flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.before_flush" \o "sqlalchemy.orm.events.SessionEvents.before_flush)

[after\_flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_flush" \o "sqlalchemy.orm.events.SessionEvents.after_flush)

[Persistence Events](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_events.html" \l "session-persistence-events)

**after\_rollback**(*session*)

Execute after a real DBAPI rollback has occurred.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeSessionOrFactory, 'after\_rollback')

**def** receive\_after\_rollback(session):

"listen for the 'after\_rollback' event"

*# ... (event handling logic) ...*

Note that this event only fires when the *actual* rollback against the database occurs - it does *not* fire each time the [Session.rollback()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.rollback" \o "sqlalchemy.orm.session.Session.rollback) method is called, if the underlying DBAPI transaction has already been rolled back. In many cases, the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) will not be in an "active" state during this event, as the current transaction is not valid. To acquire a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) which is active after the outermost rollback has proceeded, use the [SessionEvents.after\_soft\_rollback()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_soft_rollback" \o "sqlalchemy.orm.events.SessionEvents.after_soft_rollback) event, checking the [Session.is\_active](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.is_active" \o "sqlalchemy.orm.session.Session.is_active) flag.

|  |  |
| --- | --- |
| **Parameters:** | ****session**** – The target [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session). |

**after\_soft\_rollback**(*session*, *previous\_transaction*)

Execute after any rollback has occurred, including "soft" rollbacks that don't actually emit at the DBAPI level.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeSessionOrFactory, 'after\_soft\_rollback')

**def** receive\_after\_soft\_rollback(session, previous\_transaction):

"listen for the 'after\_soft\_rollback' event"

*# ... (event handling logic) ...*

This corresponds to both nested and outer rollbacks, i.e. the innermost rollback that calls the DBAPI's rollback() method, as well as the enclosing rollback calls that only pop themselves from the transaction stack.

The given [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) can be used to invoke SQL and [Session.query()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.query" \o "sqlalchemy.orm.session.Session.query) operations after an outermost rollback by first checking the [Session.is\_active](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.is_active" \o "sqlalchemy.orm.session.Session.is_active) flag:

**@event**.listens\_for(Session, "after\_soft\_rollback")

**def** do\_something(session, previous\_transaction):

**if** session.is\_active:

session.execute("select \* from some\_table")

|  |  |
| --- | --- |
| **Parameters:** | * ****session**** – The target [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session). * ****previous\_transaction**** – The [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction) transactional marker object which was just closed. The current [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction) for the given [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) is available via the [Session.transaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.transaction" \o "sqlalchemy.orm.session.Session.transaction) attribute. |

*New in version 0.7.3.*

**after\_transaction\_create**(*session*, *transaction*)

Execute when a new [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction) is created.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeSessionOrFactory, 'after\_transaction\_create')

**def** receive\_after\_transaction\_create(session, transaction):

"listen for the 'after\_transaction\_create' event"

*# ... (event handling logic) ...*

This event differs from [after\_begin()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_begin" \o "sqlalchemy.orm.events.SessionEvents.after_begin) in that it occurs for each [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction) overall, as opposed to when transactions are begun on individual database connections. It is also invoked for nested transactions and subtransactions, and is always matched by a corresponding[after\_transaction\_end()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_transaction_end" \o "sqlalchemy.orm.events.SessionEvents.after_transaction_end) event (assuming normal operation of the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)).

|  |  |
| --- | --- |
| **Parameters:** | * ****session**** – the target [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session). * ****transaction –****the target [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction).   To detect if this is the outermost [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction), as opposed to a "subtransaction" or a SAVEPOINT, test that the [SessionTransaction.parent](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction.parent" \o "sqlalchemy.orm.session.SessionTransaction.parent) attribute is None:  **@event**.listens\_for(session, "after\_transaction\_create")  **def** after\_transaction\_create(session, transaction):  **if** transaction.parent **is** **None**:  *# work with top-level transaction*  To detect if the [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction) is a SAVEPOINT, use the [SessionTransaction.nested](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction.nested" \o "sqlalchemy.orm.session.SessionTransaction.nested) attribute:  **@event**.listens\_for(session, "after\_transaction\_create")  **def** after\_transaction\_create(session, transaction):  **if** transaction.nested:  *# work with SAVEPOINT transaction* |

**See also**

[SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction)

[after\_transaction\_end()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_transaction_end" \o "sqlalchemy.orm.events.SessionEvents.after_transaction_end)

**after\_transaction\_end**(*session*, *transaction*)

Execute when the span of a [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction) ends.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeSessionOrFactory, 'after\_transaction\_end')

**def** receive\_after\_transaction\_end(session, transaction):

"listen for the 'after\_transaction\_end' event"

*# ... (event handling logic) ...*

This event differs from [after\_commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_commit" \o "sqlalchemy.orm.events.SessionEvents.after_commit) in that it corresponds to all [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction) objects in use, including those for nested transactions and subtransactions, and is always matched by a corresponding [after\_transaction\_create()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_transaction_create" \o "sqlalchemy.orm.events.SessionEvents.after_transaction_create) event.

此事件与[after\_commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_commit" \o "sqlalchemy.orm.events.SessionEvents.after_commit)不同之处在于它对应于正在使用的所有[SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction)对象，包括用于嵌套事务和子事务的SessionTransaction对象，并且始终与相应的[after\_transaction\_create()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_transaction_create" \o "sqlalchemy.orm.events.SessionEvents.after_transaction_create)事件匹配。

|  |  |
| --- | --- |
| **Parameters:** | * ****session**** – the target [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session). * ****transaction –****   the target [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction).  To detect if this is the outermost [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction), as opposed to a "subtransaction" or a SAVEPOINT, test that the [SessionTransaction.parent](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction.parent" \o "sqlalchemy.orm.session.SessionTransaction.parent) attribute is None:  **@event**.listens\_for(session, "after\_transaction\_create")  **def** after\_transaction\_end(session, transaction):  **if** transaction.parent **is** **None**:  *# work with top-level transaction*  To detect if the [SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction) is a SAVEPOINT, use the [SessionTransaction.nested](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction.nested" \o "sqlalchemy.orm.session.SessionTransaction.nested) attribute:  **@event**.listens\_for(session, "after\_transaction\_create")  **def** after\_transaction\_end(session, transaction):  **if** transaction.nested:  *# work with SAVEPOINT transaction* |

**See also**

[SessionTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.SessionTransaction" \o "sqlalchemy.orm.session.SessionTransaction)

[after\_transaction\_create()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_transaction_create" \o "sqlalchemy.orm.events.SessionEvents.after_transaction_create)

**before\_attach**(*session*, *instance*)

Execute before an instance is attached to a session.

在实例附加到会话之前执行。

Example argument forms:

示例参数形式：

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeSessionOrFactory, 'before\_attach')

**def** receive\_before\_attach(session, instance):

"listen for the 'before\_attach' event"

*# ... (event handling logic) ...*

This is called before an add, delete or merge causes the object to be part of the session.

这是在添加，删除或合并之前调用的，使对象成为会话的一部分。

*New in version 0.8.:*Note that [after\_attach()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_attach" \o "sqlalchemy.orm.events.SessionEvents.after_attach) now fires off after the item is part of the session. [before\_attach()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.before_attach" \o "sqlalchemy.orm.events.SessionEvents.before_attach) is provided for those cases where the item should not yet be part of the session state.

新版本0.8：请注意，[after\_attach()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_attach" \o "sqlalchemy.orm.events.SessionEvents.after_attach)现在在该项目是会话的一部分后触发。 对于项目不应该是会话状态的一部分的情况，提供了[before\_attach()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.before_attach" \o "sqlalchemy.orm.events.SessionEvents.before_attach)。

**See also**

[after\_attach()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_attach" \o "sqlalchemy.orm.events.SessionEvents.after_attach)

[Object Lifecycle Events](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_events.html" \l "session-lifecycle-events)

**before\_commit**(*session*)

Execute before commit is called.

调用commit之前执行。

Example argument forms:

示例参数形式：

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeSessionOrFactory, 'before\_commit')

**def** receive\_before\_commit(session):

"listen for the 'before\_commit' event"

*# ... (event handling logic) ...*

**Note**

The [before\_commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.before_commit" \o "sqlalchemy.orm.events.SessionEvents.before_commit) hook is *not* per-flush, that is, the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) can emit SQL to the database many times within the scope of a transaction. For interception of these events, use the [before\_flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.before_flush" \o "sqlalchemy.orm.events.SessionEvents.before_flush), [after\_flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_flush" \o "sqlalchemy.orm.events.SessionEvents.after_flush), or [after\_flush\_postexec()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_flush_postexec" \o "sqlalchemy.orm.events.SessionEvents.after_flush_postexec) events.

[before\_commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.before_commit" \o "sqlalchemy.orm.events.SessionEvents.before_commit)钩子不是每次刷新，也就是说，[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)可以在事务范围内多次向数据库发出SQL数据。 要拦截这些事件，请使用[before\_flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.before_flush" \o "sqlalchemy.orm.events.SessionEvents.before_flush)，[after\_flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_flush" \o "sqlalchemy.orm.events.SessionEvents.after_flush)或[after\_flush\_postexec()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_flush_postexec" \o "sqlalchemy.orm.events.SessionEvents.after_flush_postexec)事件。

|  |  |
| --- | --- |
| **Parameters:** | ****session**** – The target [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session). |

**See also**

[after\_commit()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_commit" \o "sqlalchemy.orm.events.SessionEvents.after_commit)

[after\_begin()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_begin" \o "sqlalchemy.orm.events.SessionEvents.after_begin)

[after\_transaction\_create()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_transaction_create" \o "sqlalchemy.orm.events.SessionEvents.after_transaction_create)

[after\_transaction\_end()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_transaction_end" \o "sqlalchemy.orm.events.SessionEvents.after_transaction_end)

**before\_flush**(*session*, *flush\_context*, *instances*)

Execute before flush process has started.

在flush进程开始之前执行。

Example argument forms:

示例参数形式：

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeSessionOrFactory, 'before\_flush')

**def** receive\_before\_flush(session, flush\_context, instances):

"listen for the 'before\_flush' event"

*# ... (event handling logic) ...*

|  |  |
| --- | --- |
| **Parameters:** | * ****session**** – The target [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session). * ****flush\_context**** – Internal [UOWTransaction](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.session.UOWTransaction" \o "sqlalchemy.orm.session.UOWTransaction) object which handles the details of the flush. * ****instances**** – Usually None, this is the collection of objects which can be passed to the [Session.flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.flush" \o "sqlalchemy.orm.session.Session.flush) method (note this usage is deprecated). |

**See also**

[after\_flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_flush" \o "sqlalchemy.orm.events.SessionEvents.after_flush)

[after\_flush\_postexec()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_flush_postexec" \o "sqlalchemy.orm.events.SessionEvents.after_flush_postexec)

[Persistence Events](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_events.html" \l "session-persistence-events)

**deleted\_to\_detached**(*session*, *instance*)

Intercept the "deleted to detached" transition for a specific object.

拦截特定对象的“已删除分离”转换。

Example argument forms:

示例参数形式：

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeSessionOrFactory, 'deleted\_to\_detached')

**def** receive\_deleted\_to\_detached(session, instance):

"listen for the 'deleted\_to\_detached' event"

*# ... (event handling logic) ...*

This event is invoked when a deleted object is evicted from the session. The typical case when this occurs is when the transaction for a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) in which the object was deleted is committed; the object moves from the deleted state to the detached state.

当删除的对象从会话中逐出时，将调用此事件。 发生这种情况的典型情况是提交删除对象的会话的事务; 对象从删除状态移动到分离状态。

It is also invoked for objects that were deleted in a flush when the [Session.expunge\_all()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.expunge_all" \o "sqlalchemy.orm.session.Session.expunge_all) or [Session.close()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.close" \o "sqlalchemy.orm.session.Session.close) events are called, as well as if the object is individually expunged from its deleted state via [Session.expunge()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.expunge" \o "sqlalchemy.orm.session.Session.expunge).

当调用[Session.expunge\_all()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.expunge_all" \o "sqlalchemy.orm.session.Session.expunge_all)或[Session.close()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.close" \o "sqlalchemy.orm.session.Session.close)事件时，以及如果对象通过[Session.expunge()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.expunge" \o "sqlalchemy.orm.session.Session.expunge)被单独地从其删除的状态清除，也会调用在flush中删除的对象。

*New in version 1.1.*

**See also**

[Object Lifecycle Events](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_events.html" \l "session-lifecycle-events)

**deleted\_to\_persistent**(*session*, *instance*)

Intercept the "deleted to persistent" transition for a specific object.

拦截特定对象的“已删除到持久”转换。

Example argument forms:

示例参数形式：

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeSessionOrFactory, 'deleted\_to\_persistent')

**def** receive\_deleted\_to\_persistent(session, instance):

"listen for the 'deleted\_to\_persistent' event"

*# ... (event handling logic) ...*

This transition occurs only when an object that's been deleted successfully in a flush is restored due to a call to [Session.rollback()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.rollback" \o "sqlalchemy.orm.session.Session.rollback). The event is not called under any other circumstances.

只有当由于调用[Session.rollback()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.rollback" \o "sqlalchemy.orm.session.Session.rollback)而恢复在flush中成功删除的对象时，才会恢复此转换。 在任何其他情况下都不会调用该事件。

*New in version 1.1.*

**See also**

[Object Lifecycle Events](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_events.html" \l "session-lifecycle-events)

**detached\_to\_persistent**(*session*, *instance*)

Intercept the "detached to persistent" transition for a specific object.

拦截特定对象的“分离到持久”转换。

Example argument forms:

示例参数形式：

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeSessionOrFactory, 'detached\_to\_persistent')

**def** receive\_detached\_to\_persistent(session, instance):

"listen for the 'detached\_to\_persistent' event"

*# ... (event handling logic) ...*

This event is a specialization of the [SessionEvents.after\_attach()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_attach" \o "sqlalchemy.orm.events.SessionEvents.after_attach) event which is only invoked for this specific transition. It is invoked typically during the [Session.add()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.add" \o "sqlalchemy.orm.session.Session.add) call, as well as during the [Session.delete()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.delete" \o "sqlalchemy.orm.session.Session.delete) call if the object was not previously associated with the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) (note that an object marked as "deleted" remains in the "persistent" state until the flush proceeds).

此事件是[SessionEvents.after\_attach()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_attach" \o "sqlalchemy.orm.events.SessionEvents.after_attach)事件的专门化，它仅在此特定转换时被调用。 通常在[Session.add()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.add" \o "sqlalchemy.orm.session.Session.add)调用期间以及[Session.delete()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.delete" \o "sqlalchemy.orm.session.Session.delete)调用期间调用该对象以前没有与[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)关联(请注意，标记为“已删除”的对象仍保留在“persistent” 状态，直到冲洗进行)。

**Note**

If the object becomes persistent as part of a call to [Session.delete()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.delete" \o "sqlalchemy.orm.session.Session.delete), the object is ****not**** yet marked as deleted when this event is called. To detect deleted objects, check the deleted flag sent to the [SessionEvents.persistent\_to\_detached()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.persistent_to_detached" \o "sqlalchemy.orm.events.SessionEvents.persistent_to_detached) to event after the flush proceeds, or check the [Session.deleted](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.deleted" \o "sqlalchemy.orm.session.Session.deleted) collection within the [SessionEvents.before\_flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.before_flush" \o "sqlalchemy.orm.events.SessionEvents.before_flush) event if deleted objects need to be intercepted before the flush.

如果该对象作为对[Session.delete()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.delete" \o "sqlalchemy.orm.session.Session.delete)的调用的一部分而持久存在，则调用此事件时，该对象尚未被标记为已删除。 要检测已删除的对象，请检查发送到[SessionEvents.persistent\_to\_detached()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.persistent_to_detached" \o "sqlalchemy.orm.events.SessionEvents.persistent_to_detached)的已删除标志，以便在刷新进行之后发生事件，或者检查SessionEvents.before\_flush()事件中的[Session.deleted](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.deleted" \o "sqlalchemy.orm.session.Session.deleted)集合，如果删除的对象需要在flush之前被拦截。

|  |  |
| --- | --- |
| **Parameters:** | * ****session**** – target [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) * ****instance**** – the ORM-mapped instance being operated upon. |

*New in version 1.1.*

**See also**

[Object Lifecycle Events](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_events.html" \l "session-lifecycle-events)

**loaded\_as\_persistent**(*session*, *instance*)

Intercept the "loaded as persistent" transition for a specific object.

拦截特定对象的“加载为持久”转换。

Example argument forms:

示例参数形式：

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeSessionOrFactory, 'loaded\_as\_persistent')

**def** receive\_loaded\_as\_persistent(session, instance):

"listen for the 'loaded\_as\_persistent' event"

*# ... (event handling logic) ...*

This event is invoked within the ORM loading process, and is invoked very similarly to the [InstanceEvents.load()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.InstanceEvents.load" \o "sqlalchemy.orm.events.InstanceEvents.load) event. However, the event here is linkable to a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) class or instance, rather than to a mapper or class hierarchy, and integrates with the other session lifecycle events smoothly. The object is guaranteed to be present in the session's identity map when this event is called.

此事件在ORM加载过程中被调用，并且与[InstanceEvents.load()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.InstanceEvents.load" \o "sqlalchemy.orm.events.InstanceEvents.load)事件非常相似。 但是，这里的事件可以链接到[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)类或实例，而不是映射器或类层次结构，并且可以顺利地与其他会话生命周期事件集成。 当调用此事件时，对象将保证存在于会话的身份映射中。

|  |  |
| --- | --- |
| **Parameters:** | * ****session**** – target [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) * ****instance**** – the ORM-mapped instance being operated upon. |

*New in version 1.1.*

**See also**

[Object Lifecycle Events](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_events.html" \l "session-lifecycle-events)

**pending\_to\_persistent**(*session*, *instance*)

Intercept the "pending to persistent"" transition for a specific object.

拦截特定对象的“等待持久”转换。

Example argument forms:

示例参数形式：

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeSessionOrFactory, 'pending\_to\_persistent')

**def** receive\_pending\_to\_persistent(session, instance):

"listen for the 'pending\_to\_persistent' event"

*# ... (event handling logic) ...*

This event is invoked within the flush process, and is similar to scanning the [Session.new](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.new" \o "sqlalchemy.orm.session.Session.new) collection within the [SessionEvents.after\_flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_flush" \o "sqlalchemy.orm.events.SessionEvents.after_flush)event. However, in this case the object has already been moved to the persistent state when the event is called.

此事件在flush过程中被调用，类似于在[SessionEvents.after\_flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_flush" \o "sqlalchemy.orm.events.SessionEvents.after_flush)事件中扫描[Session.new](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.new" \o "sqlalchemy.orm.session.Session.new)集合。 但是，在这种情况下，当调用事件时，对象已经被移动到持久状态。

|  |  |
| --- | --- |
| **Parameters:** | * ****session**** – target [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) * ****instance**** – the ORM-mapped instance being operated upon. |

*New in version 1.1.*

**See also**

[Object Lifecycle Events](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_events.html" \l "session-lifecycle-events)

**pending\_to\_transient**(*session*, *instance*)

Intercept the "pending to transient" transition for a specific object.

拦截特定对象的“等待瞬态”转换。

Example argument forms:

示例参数形式：

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeSessionOrFactory, 'pending\_to\_transient')

**def** receive\_pending\_to\_transient(session, instance):

"listen for the 'pending\_to\_transient' event"

*# ... (event handling logic) ...*

This less common transition occurs when an pending object that has not been flushed is evicted from the session; this can occur when the [Session.rollback()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.rollback" \o "sqlalchemy.orm.session.Session.rollback) method rolls back the transaction, or when the [Session.expunge()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.expunge" \o "sqlalchemy.orm.session.Session.expunge) method is used.

当没有刷新的待处理对象从会话中被逐出时，发生这种不太常见的转换; 当Session.rollback()方法回滚事务时，或者当使用Session.expunge()方法时，可能会发生这种情况。

|  |  |
| --- | --- |
| **Parameters:** | * ****session**** – target [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) * ****instance**** – the ORM-mapped instance being operated upon. |

*New in version 1.1.*

**See also**

[Object Lifecycle Events](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_events.html" \l "session-lifecycle-events)

**persistent\_to\_deleted**(*session*, *instance*)

Intercept the "persistent to deleted" transition for a specific object.

拦截特定对象的“持久删除”转换。

Example argument forms:

示例参数形式：

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeSessionOrFactory, 'persistent\_to\_deleted')

**def** receive\_persistent\_to\_deleted(session, instance):

"listen for the 'persistent\_to\_deleted' event"

*# ... (event handling logic) ...*

This event is invoked when a persistent object's identity is deleted from the database within a flush, however the object still remains associated with the [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) until the transaction completes.

当持久对象的身份从刷新中的数据库中删除时，会调用此事件，但是对象仍然保持与会话关联，直到事务完成。

If the transaction is rolled back, the object moves again to the persistent state, and the [SessionEvents.deleted\_to\_persistent()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.deleted_to_persistent" \o "sqlalchemy.orm.events.SessionEvents.deleted_to_persistent) event is called. If the transaction is committed, the object becomes detached, which will emit the [SessionEvents.deleted\_to\_detached()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.deleted_to_detached" \o "sqlalchemy.orm.events.SessionEvents.deleted_to_detached) event.

如果事务被回滚，对象将再次移动到持久状态，并调用SessionEvents.deleted\_to\_persistent()事件。 如果事务被提交，对象变得分离，这将发出SessionEvents.deleted\_to\_detached()事件。

Note that while the [Session.delete()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.delete" \o "sqlalchemy.orm.session.Session.delete) method is the primary public interface to mark an object as deleted, many objects get deleted due to cascade rules, which are not always determined until flush time. Therefore, there's no way to catch every object that will be deleted until the flush has proceeded. the [SessionEvents.persistent\_to\_deleted()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.persistent_to_deleted" \o "sqlalchemy.orm.events.SessionEvents.persistent_to_deleted) event is therefore invoked at the end of a flush.

请注意，虽然Session.delete()方法是将对象标记为已删除的主要公共接口，但是由于级联规则，许多对象都被删除，这些规则在刷新时间之前并不总是被确定。 因此，没有办法捕捉将要删除的所有对象，直到flush进行。 因此在刷新结束时调用SessionEvents.persistent\_to\_deleted()事件。

*New in version 1.1.*

**See also**

[Object Lifecycle Events](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_events.html" \l "session-lifecycle-events)

**persistent\_to\_detached**(*session*, *instance*)

Intercept the "persistent to detached" transition for a specific object.

拦截特定对象的“持久到分离”转换。

Example argument forms:

示例参数形式：

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeSessionOrFactory, 'persistent\_to\_detached')

**def** receive\_persistent\_to\_detached(session, instance):

"listen for the 'persistent\_to\_detached' event"

*# ... (event handling logic) ...*

This event is invoked when a persistent object is evicted from the session. There are many conditions that cause this to happen, including:

* using a method such as [Session.expunge()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.expunge" \o "sqlalchemy.orm.session.Session.expunge) or [Session.close()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.close" \o "sqlalchemy.orm.session.Session.close)
* Calling the [Session.rollback()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.rollback" \o "sqlalchemy.orm.session.Session.rollback) method, when the object was part of an INSERT statement for that session's transaction

|  |  |
| --- | --- |
| **Parameters:** | * ****session**** – target [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) * ****instance**** – the ORM-mapped instance being operated upon. * ****deleted**** – boolean. If True, indicates this object moved to the detached state because it was marked as deleted and flushed. |

*New in version 1.1.*

**See also**

[Object Lifecycle Events](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_events.html" \l "session-lifecycle-events)

**persistent\_to\_transient**(*session*, *instance*)

Intercept the "persistent to transient" transition for a specific object.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style***@event**.listens\_for(SomeSessionOrFactory, 'persistent\_to\_transient')**def** receive\_persistent\_to\_transient(session, instance):

"listen for the 'persistent\_to\_transient' event"

*# ... (event handling logic) ...*

This less common transition occurs when an pending object that has has been flushed is evicted from the session; this can occur when the [Session.rollback()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.rollback" \o "sqlalchemy.orm.session.Session.rollback) method rolls back the transaction.

|  |  |
| --- | --- |
| **Parameters:** | * ****session**** – target [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) * ****instance**** – the ORM-mapped instance being operated upon. |

*New in version 1.1.*

**See also**

[Object Lifecycle Events](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_events.html" \l "session-lifecycle-events)

**transient\_to\_pending**(*session*, *instance*)

Intercept the "transient to pending" transition for a specific object.

截取特定对象的“暂态到挂起”转换。

Example argument forms:

示例参数表单：

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeSessionOrFactory, 'transient\_to\_pending')

**def** receive\_transient\_to\_pending(session, instance):

"listen for the 'transient\_to\_pending' event"

*# ... (event handling logic) ...*

This event is a specialization of the [SessionEvents.after\_attach()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents.after_attach" \o "sqlalchemy.orm.events.SessionEvents.after_attach) event which is only invoked for this specific transition. It is invoked typically during the [Session.add()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.add" \o "sqlalchemy.orm.session.Session.add) call.

|  |  |
| --- | --- |
| **Parameters:** | * ****session**** – target [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) * ****instance**** – the ORM-mapped instance being operated upon. |

*New in version 1.1.*

**See also**

[Object Lifecycle Events](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_events.html" \l "session-lifecycle-events)

### 6.1.5 Query Events

*class*sqlalchemy.orm.events.**QueryEvents**

Bases: [sqlalchemy.event.base.Events](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.event.base.Events" \o "sqlalchemy.event.base.Events)

Represent events within the construction of a [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) object.

在Query对象的构造中表示事件。

The events here are intended to be used with an as-yet-unreleased inspection system for [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query). Some very basic operations are possible now, however the inspection system is intended to allow complex query manipulations to be automated.

此处的事件旨在用于查询的尚未发布的检查系统。 现在可以进行一些非常基本的操作，但是检查系统的目的是使复杂的查询操作自动化。

*New in version 1.0.0.*

**before\_compile**(*query*)

Receive the [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) object before it is composed into a core [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select) object.

在将其组成核心Select对象之前接收Query对象。

Example argument forms:

示例参数表单：

**from** **sqlalchemy** **import** event

*# standard decorator style***@event**.listens\_for(SomeQuery, 'before\_compile')**def** receive\_before\_compile(query):

"listen for the 'before\_compile' event"

*# ... (event handling logic) ...*

This event is intended to allow changes to the query given:

**@event**.listens\_for(Query, "before\_compile", retval=**True**)**def** no\_deleted(query):

**for** desc **in** query.column\_descriptions:

**if** desc['type'] **is** User:

entity = desc['entity']

query = query.filter(entity.deleted == **False**)

**return** query

The event should normally be listened with the retval=True parameter set, so that the modified query may be returned.

### 6.1.6 Instrumentation Events

Defines SQLAlchemy's system of class instrumentation.

This module is usually not directly visible to user applications, but defines a large part of the ORM's interactivity.

instrumentation.py deals with registration of end-user classes for state tracking. It interacts closely with state.py and attributes.py which establish per-instance and per-class-attribute instrumentation, respectively.

The class instrumentation system can be customized on a per-class or global basis using the [sqlalchemy.ext.instrumentation](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/instrumentation.html" \l "module-sqlalchemy.ext.instrumentation" \o "sqlalchemy.ext.instrumentation) module, which provides the means to build and specify alternate instrumentation forms.

*class*sqlalchemy.orm.events.**InstrumentationEvents**

Bases: [sqlalchemy.event.base.Events](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.event.base.Events" \o "sqlalchemy.event.base.Events)

Events related to class instrumentation events.

The listeners here support being established against any new style class, that is any object that is a subclass of 'type'. Events will then be fired off for events against that class. If the "propagate=True" flag is passed to event.listen(), the event will fire off for subclasses of that class as well.

The Python type builtin is also accepted as a target, which when used has the effect of events being emitted for all classes.

Note the "propagate" flag here is defaulted to True, unlike the other class level events where it defaults to False. This means that new subclasses will also be the subject of these events, when a listener is established on a superclass.

*Changed in version 0.8:*- events here will emit based on comparing the incoming class to the type of class passed to [event.listen()](http://docs.sqlalchemy.org/en/rel_1_1/core/event.html" \l "sqlalchemy.event.listen" \o "sqlalchemy.event.listen). Previously, the event would fire for any class unconditionally regardless of what class was sent for listening, despite documentation which stated the contrary.

**attribute\_instrument**(*cls*, *key*, *inst*)

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style***@event**.listens\_for(SomeBaseClass, 'attribute\_instrument')**def** receive\_attribute\_instrument(cls, key, inst):

"listen for the 'attribute\_instrument' event"

*# ... (event handling logic) ...*

Called when an attribute is instrumented.

**class\_instrument**(*cls*)

Called after the given class is instrumented.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style***@event**.listens\_for(SomeBaseClass, 'class\_instrument')**def** receive\_class\_instrument(cls):

"listen for the 'class\_instrument' event"

*# ... (event handling logic) ...*

To get at the [ClassManager](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.instrumentation.ClassManager" \o "sqlalchemy.orm.instrumentation.ClassManager), use manager\_of\_class().

**class\_uninstrument**(*cls*)

Called before the given class is uninstrumented.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style***@event**.listens\_for(SomeBaseClass, 'class\_uninstrument')**def** receive\_class\_uninstrument(cls):

"listen for the 'class\_uninstrument' event"

*# ... (event handling logic) ...*

To get at the [ClassManager](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.instrumentation.ClassManager" \o "sqlalchemy.orm.instrumentation.ClassManager), use manager\_of\_class().

## 6.2 ORM Internals

Key ORM constructs, not otherwise covered in other sections, are listed here.

*class*sqlalchemy.orm.state.**AttributeState**(*state*, *key*)

Provide an inspection interface corresponding to a particular attribute on a particular mapped object.

The [AttributeState](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.AttributeState" \o "sqlalchemy.orm.state.AttributeState) object is accessed via the [InstanceState.attrs](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState.attrs" \o "sqlalchemy.orm.state.InstanceState.attrs) collection of a particular [InstanceState](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState" \o "sqlalchemy.orm.state.InstanceState):

**from** **sqlalchemy** **import** inspect

insp = inspect(some\_mapped\_object)attr\_state = insp.attrs.some\_attribute

**history**

Return the current pre-flush change history for this attribute, via the [History](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.attributes.History" \o "sqlalchemy.orm.attributes.History) interface.

This method will ****not**** emit loader callables if the value of the attribute is unloaded.

**See also**

[AttributeState.load\_history()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.AttributeState.load_history" \o "sqlalchemy.orm.state.AttributeState.load_history) - retrieve history using loader callables if the value is not locally present.

[attributes.get\_history()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.attributes.get_history" \o "sqlalchemy.orm.attributes.get_history) - underlying function

**load\_history**()

Return the current pre-flush change history for this attribute, via the [History](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.attributes.History" \o "sqlalchemy.orm.attributes.History) interface.

This method ****will**** emit loader callables if the value of the attribute is unloaded.

**See also**

[AttributeState.history](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.AttributeState.history" \o "sqlalchemy.orm.state.AttributeState.history)

[attributes.get\_history()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.attributes.get_history" \o "sqlalchemy.orm.attributes.get_history) - underlying function

*New in version 0.9.0.*

**loaded\_value**

The current value of this attribute as loaded from the database.

If the value has not been loaded, or is otherwise not present in the object's dictionary, returns NO\_VALUE.

**value**

Return the value of this attribute.

This operation is equivalent to accessing the object's attribute directly or via getattr(), and will fire off any pending loader callables if needed.

*class*sqlalchemy.orm.util.**CascadeOptions**

Bases: \_\_builtin\_\_.frozenset

Keeps track of the options sent to relationship().cascade

*class*sqlalchemy.orm.instrumentation.**ClassManager**(*class\_*)

Bases: \_\_builtin\_\_.dict

tracks state information at the class level.

**\_\_le\_\_**

*inherited from the* \_\_le\_\_ *attribute of* dict

x.\_\_le\_\_(y) <==> x<=y

**\_\_lt\_\_**

*inherited from the* \_\_lt\_\_ *attribute of* dict

x.\_\_lt\_\_(y) <==> x<y

**\_\_ne\_\_**

*inherited from the* \_\_ne\_\_ *attribute of* dict

x.\_\_ne\_\_(y) <==> x!=y

**clear**() → None. Remove all items from D.

*inherited from the* clear() *method of* dict

**copy**() → a shallow copy of D

*inherited from the* copy() *method of* dict

**dispose**()

Dissasociate this manager from its class.

**fromkeys**(*S*[, *v*]) → New dict with keys from S and values equal to v.

*inherited from the* fromkeys() *method of* dict

v defaults to None.

**get**(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

*inherited from the* get() *method of* dict

**has\_key**(*k*) → True if D has a key k, else False

*inherited from the* has\_key() *method of* dict

**has\_parent**(*state*, *key*, *optimistic=False*)

TODO

**items**() → list of D's (key, value) pairs, as 2-tuples

*inherited from the* items() *method of* dict

**iteritems**() → an iterator over the (key, value) items of D

*inherited from the* iteritems() *method of* dict

**iterkeys**() → an iterator over the keys of D

*inherited from the* iterkeys() *method of* dict

**itervalues**() → an iterator over the values of D

*inherited from the* itervalues() *method of* dict

**keys**() → list of D's keys

*inherited from the* keys() *method of* dict

**manage**()

Mark this instance as the manager for its class.

**original\_init**

x.\_\_init\_\_(…) initializes x; see help(type(x)) for signature

**pop**(*k*[, *d*]) → v, remove specified key and return the corresponding value.

*inherited from the* pop() *method of* dict

If key is not found, d is returned if given, otherwise KeyError is raised

**popitem**() → (k, v), remove and return some (key, value) pair as a

*inherited from the* popitem() *method of* dict

2-tuple; but raise KeyError if D is empty.

**setdefault**(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

*inherited from the* setdefault() *method of* dict

**state\_getter**()

Return a (instance) -> InstanceState callable.

"state getter" callables should raise either KeyError or AttributeError if no InstanceState could be found for the instance.

**unregister**()

remove all instrumentation established by this ClassManager.

**update**([*E*, ]*\*\*F*) → None. Update D from dict/iterable E and F.

*inherited from the* update() *method of* dict

If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

**values**() → list of D's values

*inherited from the* values() *method of* dict

**viewitems**() → a set-like object providing a view on D's items

*inherited from the* viewitems() *method of* dict

**viewkeys**() → a set-like object providing a view on D's keys

*inherited from the* viewkeys() *method of* dict

**viewvalues**() → an object providing a view on D's values

*inherited from the* viewvalues() *method of* dict

*class*sqlalchemy.orm.properties.**ColumnProperty**(*\*columns*, *\*\*kwargs*)

Bases: sqlalchemy.orm.interfaces.StrategizedProperty

Describes an object attribute that corresponds to a table column.

Public constructor is the [orm.column\_property()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_columns.html" \l "sqlalchemy.orm.column_property" \o "sqlalchemy.orm.column_property) function.

*class***Comparator**(*prop*, *parentmapper*, *adapt\_to\_entity=None*)

Bases: sqlalchemy.util.langhelpers.MemoizedSlots, [sqlalchemy.orm.interfaces.PropComparator](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator" \o "sqlalchemy.orm.interfaces.PropComparator)

Produce boolean, comparison, and other operators for [ColumnProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.ColumnProperty" \o "sqlalchemy.orm.properties.ColumnProperty) attributes.

See the documentation for [PropComparator](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator" \o "sqlalchemy.orm.interfaces.PropComparator) for a brief overview.

See also:

[PropComparator](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator" \o "sqlalchemy.orm.interfaces.PropComparator)

[ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

[Redefining and Creating New Operators](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "types-operators)

[TypeEngine.comparator\_factory](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.comparator_factory" \o "sqlalchemy.types.TypeEngine.comparator_factory)

**\_\_eq\_\_**(*other*)

*inherited from the* [\_\_eq\_\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.__eq__" \o "sqlalchemy.sql.operators.ColumnOperators.__eq__) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the == operator.

In a column context, produces the clause a = b. If the target is None, produces a IS NULL.

**\_\_le\_\_**(*other*)

*inherited from the* [\_\_le\_\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.__le__" \o "sqlalchemy.sql.operators.ColumnOperators.__le__) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the <= operator.

In a column context, produces the clause a <= b.

**\_\_lt\_\_**(*other*)

*inherited from the* [\_\_lt\_\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.__lt__" \o "sqlalchemy.sql.operators.ColumnOperators.__lt__) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the < operator.

In a column context, produces the clause a < b.

**\_\_ne\_\_**(*other*)

*inherited from the* [\_\_ne\_\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.__ne__" \o "sqlalchemy.sql.operators.ColumnOperators.__ne__) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the != operator.

In a column context, produces the clause a != b. If the target is None, produces a IS NOT NULL.

**adapt\_to\_entity**(*adapt\_to\_entity*)

*inherited from the* [adapt\_to\_entity()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator.adapt_to_entity" \o "sqlalchemy.orm.interfaces.PropComparator.adapt_to_entity) *method of* [PropComparator](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator" \o "sqlalchemy.orm.interfaces.PropComparator)

Return a copy of this PropComparator which will use the given [AliasedInsp](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.util.AliasedInsp" \o "sqlalchemy.orm.util.AliasedInsp) to produce corresponding expressions.

**adapter**

*inherited from the* [adapter](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator.adapter" \o "sqlalchemy.orm.interfaces.PropComparator.adapter) *attribute of* [PropComparator](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator" \o "sqlalchemy.orm.interfaces.PropComparator)

Produce a callable that adapts column expressions to suit an aliased version of this comparator.

**all\_**()

*inherited from the* [all\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.all_" \o "sqlalchemy.sql.operators.ColumnOperators.all_) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Produce a [all\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.all_" \o "sqlalchemy.sql.expression.all_) clause against the parent object.

*New in version 1.1.*

**any**(*criterion=None*, *\*\*kwargs*)

*inherited from the* [any()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator.any" \o "sqlalchemy.orm.interfaces.PropComparator.any) *method of* [PropComparator](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator" \o "sqlalchemy.orm.interfaces.PropComparator)

Return true if this collection contains any member that meets the given criterion.

The usual implementation of any() is [RelationshipProperty.Comparator.any()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator.any" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator.any).

|  |  |
| --- | --- |
| **Parameters:** | * ****criterion**** – an optional ClauseElement formulated against the member class' table or attributes. * ****\*\*kwargs**** – key/value pairs corresponding to member class attribute names which will be compared via equality to the corresponding values. |

**any\_**()

*inherited from the* [any\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.any_" \o "sqlalchemy.sql.operators.ColumnOperators.any_) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Produce a [any\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.any_" \o "sqlalchemy.sql.expression.any_) clause against the parent object.

*New in version 1.1.*

**asc**()

*inherited from the* [asc()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.asc" \o "sqlalchemy.sql.operators.ColumnOperators.asc) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Produce a [asc()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.asc" \o "sqlalchemy.sql.expression.asc) clause against the parent object.

**between**(*cleft*, *cright*, *symmetric=False*)

*inherited from the* [between()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.between" \o "sqlalchemy.sql.operators.ColumnOperators.between) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Produce a [between()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.between" \o "sqlalchemy.sql.expression.between) clause against the parent object, given the lower and upper range.

**collate**(*collation*)

*inherited from the* [collate()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.collate" \o "sqlalchemy.sql.operators.ColumnOperators.collate) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Produce a [collate()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.collate" \o "sqlalchemy.sql.expression.collate) clause against the parent object, given the collation string.

**concat**(*other*)

*inherited from the* [concat()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.concat" \o "sqlalchemy.sql.operators.ColumnOperators.concat) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the 'concat' operator.

In a column context, produces the clause a || b, or uses the concat() operator on MySQL.

**contains**(*other*, *\*\*kwargs*)

*inherited from the* [contains()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.contains" \o "sqlalchemy.sql.operators.ColumnOperators.contains) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the 'contains' operator.

In a column context, produces the clause LIKE '%<other>%'

**desc**()

*inherited from the* [desc()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.desc" \o "sqlalchemy.sql.operators.ColumnOperators.desc) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Produce a [desc()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.desc" \o "sqlalchemy.sql.expression.desc) clause against the parent object.

**distinct**()

*inherited from the* [distinct()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.distinct" \o "sqlalchemy.sql.operators.ColumnOperators.distinct) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Produce a [distinct()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.distinct" \o "sqlalchemy.sql.expression.distinct) clause against the parent object.

**endswith**(*other*, *\*\*kwargs*)

*inherited from the* [endswith()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.endswith" \o "sqlalchemy.sql.operators.ColumnOperators.endswith) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the 'endswith' operator.

In a column context, produces the clause LIKE '%<other>'

**has**(*criterion=None*, *\*\*kwargs*)

*inherited from the* [has()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator.has" \o "sqlalchemy.orm.interfaces.PropComparator.has) *method of* [PropComparator](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator" \o "sqlalchemy.orm.interfaces.PropComparator)

Return true if this element references a member which meets the given criterion.

The usual implementation of has() is [RelationshipProperty.Comparator.has()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator.has" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator.has).

|  |  |
| --- | --- |
| **Parameters:** | * ****criterion**** – an optional ClauseElement formulated against the member class' table or attributes. * ****\*\*kwargs**** – key/value pairs corresponding to member class attribute names which will be compared via equality to the corresponding values. |

**ilike**(*other*, *escape=None*)

*inherited from the* [ilike()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.ilike" \o "sqlalchemy.sql.operators.ColumnOperators.ilike) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the ilike operator, e.g. case insensitive LIKE.

实现ilike操作符，例如 不区分大小写。

In a column context, produces an expression either of the form:

在列上下文中，生成一个表达式：

lower(a) LIKE lower(other)

Or on backends that support the ILIKE operator:

a ILIKE other

E.g.:

stmt = select([sometable]).\

where(sometable.c.column.ilike("*%f*oobar%"))

|  |  |
| --- | --- |
| **Parameters:** | * ****other**** – expression to be compared * ****escape –****optional escape character, renders the ESCAPE keyword, e.g.:   somecolumn.ilike("foo/%bar", escape="/") |

**See also**

[ColumnOperators.like()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.like" \o "sqlalchemy.sql.operators.ColumnOperators.like)

**in\_**(*other*)

*inherited from the* [in\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.in_" \o "sqlalchemy.sql.operators.ColumnOperators.in_) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the in operator.

实现in运算符。

In a column context, produces the clause a IN other. "other" may be a tuple/list of column expressions, or a [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) construct.

在列上下文中生成子句a IN其他。 “其他”可能是列表达式或select()构造的元组/列表。

**is\_**(*other*)

*inherited from the* [is\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.is_" \o "sqlalchemy.sql.operators.ColumnOperators.is_) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the IS operator.

实现IS运算符。

Normally, IS is generated automatically when comparing to a value of None, which resolves to NULL. However, explicit usage of IS may be desirable if comparing to boolean values on certain platforms.

通常情况下，与无效的值进行比较时会自动生成IS，这个值解析为NULL。 然而，如果与某些平台上的布尔值进行比较，则可能需要明确使用IS。

*New in version 0.7.9.*

**See also**

[ColumnOperators.isnot()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.isnot" \o "sqlalchemy.sql.operators.ColumnOperators.isnot)

**is\_distinct\_from**(*other*)

*inherited from the* [is\_distinct\_from()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.is_distinct_from" \o "sqlalchemy.sql.operators.ColumnOperators.is_distinct_from) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the IS DISTINCT FROM operator.

Renders "a IS DISTINCT FROM b" on most platforms; on some such as SQLite may render "a IS NOT b".

*New in version 1.1.*

**isnot**(*other*)

*inherited from the* [isnot()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.isnot" \o "sqlalchemy.sql.operators.ColumnOperators.isnot) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the IS NOT operator.

实现IS NOT运算符。

Normally, IS NOT is generated automatically when comparing to a value of None, which resolves to NULL. However, explicit usage of IS NOT may be desirable if comparing to boolean values on certain platforms.

通常情况下，与None值进行比较时会自动生成IS NOT，该值解析为NULL。 但是，如果与某些平台上的布尔值进行比较，那么IS NOT的显式使用可能是可取的。

*New in version 0.7.9.*

**See also**

[ColumnOperators.is\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.is_" \o "sqlalchemy.sql.operators.ColumnOperators.is_)

**isnot\_distinct\_from**(*other*)

*inherited from the* [isnot\_distinct\_from()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.isnot_distinct_from" \o "sqlalchemy.sql.operators.ColumnOperators.isnot_distinct_from) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the IS NOT DISTINCT FROM operator.

Renders "a IS NOT DISTINCT FROM b" on most platforms; on some such as SQLite may render "a IS b".

在大多数平台上渲染“a IS NOT DISTINCT FROM b”; 在一些如SQLite可能会渲染“a IS b”。

*New in version 1.1.*

**like**(*other*, *escape=None*)

*inherited from the* [like()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.like" \o "sqlalchemy.sql.operators.ColumnOperators.like) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the like operator.

In a column context, produces the expression:

a LIKE other

E.g.:

stmt = select([sometable]).\

where(sometable.c.column.like("*%f*oobar%"))

|  |  |
| --- | --- |
| **Parameters:** | * ****other**** – expression to be compared * ****escape –****optional escape character, renders the ESCAPE keyword, e.g.:   somecolumn.like("foo/%bar", escape="/") |

**See also**

[ColumnOperators.ilike()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.ilike" \o "sqlalchemy.sql.operators.ColumnOperators.ilike)

**match**(*other*, *\*\*kwargs*)

*inherited from the* [match()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.match" \o "sqlalchemy.sql.operators.ColumnOperators.match) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implements a database-specific 'match' operator.

[match()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.match" \o "sqlalchemy.sql.operators.ColumnOperators.match) attempts to resolve to a MATCH-like function or operator provided by the backend. Examples include:

* PostgreSQL - renders x @@ to\_tsquery(y)
* MySQL - renders MATCH (x) AGAINST (y IN BOOLEAN MODE)
* Oracle - renders CONTAINS(x, y)
* other backends may provide special implementations.
* Backends without any special implementation will emit the operator as "MATCH". This is compatible with SQlite, for example.

**notilike**(*other*, *escape=None*)

*inherited from the* [notilike()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.notilike" \o "sqlalchemy.sql.operators.ColumnOperators.notilike) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

implement the NOT ILIKE operator.

This is equivalent to using negation with [ColumnOperators.ilike()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.ilike" \o "sqlalchemy.sql.operators.ColumnOperators.ilike), i.e. ~x.ilike(y).

*New in version 0.8.*

**See also**

[ColumnOperators.ilike()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.ilike" \o "sqlalchemy.sql.operators.ColumnOperators.ilike)

**notin\_**(*other*)

*inherited from the* [notin\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.notin_" \o "sqlalchemy.sql.operators.ColumnOperators.notin_) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

implement the NOT IN operator.

This is equivalent to using negation with [ColumnOperators.in\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.in_" \o "sqlalchemy.sql.operators.ColumnOperators.in_), i.e. ~x.in\_(y).

*New in version 0.8.*

**See also**

[ColumnOperators.in\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.in_" \o "sqlalchemy.sql.operators.ColumnOperators.in_)

**notlike**(*other*, *escape=None*)

*inherited from the* [notlike()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.notlike" \o "sqlalchemy.sql.operators.ColumnOperators.notlike) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

implement the NOT LIKE operator.

This is equivalent to using negation with [ColumnOperators.like()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.like" \o "sqlalchemy.sql.operators.ColumnOperators.like), i.e. ~x.like(y).

*New in version 0.8.*

**See also**

[ColumnOperators.like()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.like" \o "sqlalchemy.sql.operators.ColumnOperators.like)

**nullsfirst**()

*inherited from the* [nullsfirst()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.nullsfirst" \o "sqlalchemy.sql.operators.ColumnOperators.nullsfirst) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Produce a [nullsfirst()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.nullsfirst" \o "sqlalchemy.sql.expression.nullsfirst) clause against the parent object.

**nullslast**()

*inherited from the* [nullslast()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.nullslast" \o "sqlalchemy.sql.operators.ColumnOperators.nullslast) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Produce a [nullslast()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.nullslast" \o "sqlalchemy.sql.expression.nullslast) clause against the parent object.

**of\_type**(*class\_*)

*inherited from the* [of\_type()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator.of_type" \o "sqlalchemy.orm.interfaces.PropComparator.of_type) *method of* [PropComparator](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator" \o "sqlalchemy.orm.interfaces.PropComparator)

Redefine this object in terms of a polymorphic subclass.

Returns a new PropComparator from which further criterion can be evaluated.

e.g.:

query.join(Company.employees.of\_type(Engineer)).\

filter(Engineer.name=='foo')

|  |  |
| --- | --- |
| **Parameters:** | ****class\_**** – a class or mapper indicating that criterion will be against this specific subclass. |

**op**(*opstring*, *precedence=0*, *is\_comparison=False*)

*inherited from the* [op()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.Operators.op" \o "sqlalchemy.sql.operators.Operators.op) *method of* [Operators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.Operators" \o "sqlalchemy.sql.operators.Operators)

produce a generic operator function.

e.g.:

somecolumn.op("\*")(5)

produces:

somecolumn \* 5

This function can also be used to make bitwise operators explicit. For example:

somecolumn.op('&')(0xff)

is a bitwise AND of the value in somecolumn.

|  |  |
| --- | --- |
| **Parameters:** | * ****operator**** – a string which will be output as the infix operator between this element and the expression passed to the generated function. * ****precedence –****precedence to apply to the operator, when parenthesizing expressions. A lower number will cause the expression to be parenthesized when applied against another operator with higher precedence. The default value of 0 is lower than all operators except for the comma (,) and AS operators. A value of 100 will be higher or equal to all operators, and -100 will be lower than or equal to all operators.   *New in version 0.8:*- added the 'precedence' argument.   * ****is\_comparison –****if True, the operator will be considered as a "comparison" operator, that is which evaluates to a boolean true/false value, like ==, >, etc. This flag should be set so that ORM relationships can establish that the operator is a comparison operator when used in a custom join condition.   *New in version 0.9.2:*- added the [Operators.op.is\_comparison](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.Operators.op.params.is_comparison" \o "sqlalchemy.sql.operators.Operators.op) flag. |

**See also**

[Redefining and Creating New Operators](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "types-operators)

[Using custom operators in join conditions](http://docs.sqlalchemy.org/en/rel_1_1/orm/join_conditions.html" \l "relationship-custom-operator)

**startswith**(*other*, *\*\*kwargs*)

*inherited from the* [startswith()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.startswith" \o "sqlalchemy.sql.operators.ColumnOperators.startswith) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the startwith operator.

In a column context, produces the clause LIKE '<other>%'

**\_\_init\_\_**(*\*columns*, *\*\*kwargs*)

Construct a new [ColumnProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.ColumnProperty" \o "sqlalchemy.orm.properties.ColumnProperty) object.

This constructor is mirrored as a public API function; see [column\_property()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_columns.html" \l "sqlalchemy.orm.column_property" \o "sqlalchemy.orm.column_property) for a full usage and argument description.

**cascade\_iterator**(*type\_*, *state*, *visited\_instances=None*, *halt\_on=None*)

*inherited from the* [cascade\_iterator()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty.cascade_iterator" \o "sqlalchemy.orm.interfaces.MapperProperty.cascade_iterator) *method of* [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty)

Iterate through instances related to the given instance for a particular 'cascade', starting with this MapperProperty.

Return an iterator3-tuples (instance, mapper, state).

Note that the 'cascade' collection on this MapperProperty is checked first for the given type before cascade\_iterator is called.

This method typically only applies to RelationshipProperty.

**class\_attribute**

*inherited from the* [class\_attribute](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty.class_attribute" \o "sqlalchemy.orm.interfaces.MapperProperty.class_attribute) *attribute of* [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty)

Return the class-bound descriptor corresponding to this [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty).

This is basically a getattr() call:

**return** getattr(self.parent.class\_, self.key)

I.e. if this [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty) were named addresses, and the class to which it is mapped is User, this sequence is possible:

**>>> from** **sqlalchemy** **import** inspect

**>>>** mapper = inspect(User)

**>>>** addresses\_property = mapper.attrs.addresses

**>>>** addresses\_property.class\_attribute **is** User.addressesTrue

**>>>** User.addresses.property **is** addresses\_propertyTrue

**expression**

Return the primary column or expression for this ColumnProperty.

**extension\_type***= symbol('NOT\_EXTENSION')*

**init**()

*inherited from the* [init()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty.init" \o "sqlalchemy.orm.interfaces.MapperProperty.init) *method of* [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty)

Called after all mappers are created to assemble relationships between mappers and perform other post-mapper-creation initialization steps.

**set\_parent**(*parent*, *init*)

*inherited from the* [set\_parent()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty.set_parent" \o "sqlalchemy.orm.interfaces.MapperProperty.set_parent) *method of* [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty)

Set the parent mapper that references this MapperProperty.

This method is overridden by some subclasses to perform extra setup when the mapper is first known.

*class*sqlalchemy.orm.properties.**ComparableProperty**(*comparator\_factory*, *descriptor=None*, *doc=None*, *info=None*)

Bases: sqlalchemy.orm.descriptor\_props.DescriptorProperty

Instruments a Python property for use in query expressions.

**\_\_init\_\_**(*comparator\_factory*, *descriptor=None*, *doc=None*, *info=None*)

Construct a new [ComparableProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.ComparableProperty" \o "sqlalchemy.orm.properties.ComparableProperty) object.

This constructor is mirrored as a public API function; see comparable\_property() for a full usage and argument description.

*class*sqlalchemy.orm.descriptor\_props.**CompositeProperty**(*class\_*, *\*attrs*, *\*\*kwargs*)

Bases: sqlalchemy.orm.descriptor\_props.DescriptorProperty

Defines a "composite" mapped attribute, representing a collection of columns as one attribute.

[CompositeProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.descriptor_props.CompositeProperty" \o "sqlalchemy.orm.descriptor_props.CompositeProperty) is constructed using the [composite()](http://docs.sqlalchemy.org/en/rel_1_1/orm/composites.html" \l "sqlalchemy.orm.composite" \o "sqlalchemy.orm.composite) function.

**See also**

[Composite Column Types](http://docs.sqlalchemy.org/en/rel_1_1/orm/composites.html" \l "mapper-composite)

*class***Comparator**(*prop*, *parentmapper*, *adapt\_to\_entity=None*)

Bases: [sqlalchemy.orm.interfaces.PropComparator](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator" \o "sqlalchemy.orm.interfaces.PropComparator)

Produce boolean, comparison, and other operators for [CompositeProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.descriptor_props.CompositeProperty" \o "sqlalchemy.orm.descriptor_props.CompositeProperty) attributes.

See the example in [Redefining Comparison Operations for Composites](http://docs.sqlalchemy.org/en/rel_1_1/orm/composites.html" \l "composite-operations) for an overview of usage , as well as the documentation for [PropComparator](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator" \o "sqlalchemy.orm.interfaces.PropComparator).

See also:

[PropComparator](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator" \o "sqlalchemy.orm.interfaces.PropComparator)

[ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

[Redefining and Creating New Operators](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "types-operators)

[TypeEngine.comparator\_factory](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.comparator_factory" \o "sqlalchemy.types.TypeEngine.comparator_factory)

**\_\_init\_\_**(*class\_*, *\*attrs*, *\*\*kwargs*)

Construct a new [CompositeProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.descriptor_props.CompositeProperty" \o "sqlalchemy.orm.descriptor_props.CompositeProperty) object.

This constructor is mirrored as a public API function; see [composite()](http://docs.sqlalchemy.org/en/rel_1_1/orm/composites.html" \l "sqlalchemy.orm.composite" \o "sqlalchemy.orm.composite) for a full usage and argument description.

**do\_init**()

Initialization which occurs after the [CompositeProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.descriptor_props.CompositeProperty" \o "sqlalchemy.orm.descriptor_props.CompositeProperty) has been associated with its parent mapper.

**get\_history**(*state*, *dict\_*, *passive=symbol('PASSIVE\_OFF')*)

Provided for userland code that uses attributes.get\_history().

*class*sqlalchemy.orm.attributes.**Event**(*attribute\_impl*, *op*)

A token propagated throughout the course of a chain of attribute events.

Serves as an indicator of the source of the event and also provides a means of controlling propagation across a chain of attribute operations.

The [Event](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.attributes.Event" \o "sqlalchemy.orm.attributes.Event) object is sent as the initiator argument when dealing with the [AttributeEvents.append()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.AttributeEvents.append" \o "sqlalchemy.orm.events.AttributeEvents.append), [AttributeEvents.set()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.AttributeEvents.set" \o "sqlalchemy.orm.events.AttributeEvents.set), and [AttributeEvents.remove()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.AttributeEvents.remove" \o "sqlalchemy.orm.events.AttributeEvents.remove) events.

The [Event](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.attributes.Event" \o "sqlalchemy.orm.attributes.Event) object is currently interpreted by the backref event handlers, and is used to control the propagation of operations across two mutually-dependent attributes.

*New in version 0.9.0.*

|  |  |
| --- | --- |
| **Variables:** | * **[impl](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Interval.impl" \o "sqlalchemy.types.Interval.impl)** – The AttributeImpl which is the current event initiator. * **[op](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.attributes.QueryableAttribute.op" \o "sqlalchemy.orm.attributes.QueryableAttribute.op)** – The symbol OP\_APPEND, OP\_REMOVE or OP\_REPLACE, indicating the source operation. |

*class*sqlalchemy.orm.identity.**IdentityMap**

**check\_modified**()

return True if any InstanceStates present have been marked as 'modified'.

*class*sqlalchemy.orm.base.**InspectionAttr**

A base class applied to all ORM objects that can be returned by the [inspect()](http://docs.sqlalchemy.org/en/rel_1_1/core/inspection.html" \l "sqlalchemy.inspection.inspect" \o "sqlalchemy.inspection.inspect) function.

The attributes defined here allow the usage of simple boolean checks to test basic facts about the object returned.

While the boolean checks here are basically the same as using the Python isinstance() function, the flags here can be used without the need to import all of these classes, and also such that the SQLAlchemy class system can change while leaving the flags here intact for forwards-compatibility.

**extension\_type***= symbol('NOT\_EXTENSION')*

The extension type, if any. Defaults to [interfaces.NOT\_EXTENSION](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.NOT_EXTENSION" \o "sqlalchemy.orm.interfaces.NOT_EXTENSION)

*New in version 0.8.0.*

**See also**

[HYBRID\_METHOD](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/hybrid.html" \l "sqlalchemy.ext.hybrid.HYBRID_METHOD" \o "sqlalchemy.ext.hybrid.HYBRID_METHOD)

[HYBRID\_PROPERTY](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/hybrid.html" \l "sqlalchemy.ext.hybrid.HYBRID_PROPERTY" \o "sqlalchemy.ext.hybrid.HYBRID_PROPERTY)

[ASSOCIATION\_PROXY](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.ASSOCIATION_PROXY" \o "sqlalchemy.ext.associationproxy.ASSOCIATION_PROXY)

**is\_aliased\_class***= False*

True if this object is an instance of [AliasedClass](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.util.AliasedClass" \o "sqlalchemy.orm.util.AliasedClass).

**is\_attribute***= False*

True if this object is a Python [descriptor](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-descriptor).

如果此对象是Python描述符，则为true。

This can refer to one of many types. Usually a [QueryableAttribute](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.attributes.QueryableAttribute" \o "sqlalchemy.orm.attributes.QueryableAttribute) which handles attributes events on behalf of a [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty). But can also be an extension type such as [AssociationProxy](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.AssociationProxy" \o "sqlalchemy.ext.associationproxy.AssociationProxy) or [hybrid\_property](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/hybrid.html" \l "sqlalchemy.ext.hybrid.hybrid_property" \o "sqlalchemy.ext.hybrid.hybrid_property). The [InspectionAttr.extension\_type](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.base.InspectionAttr.extension_type" \o "sqlalchemy.orm.base.InspectionAttr.extension_type) will refer to a constant identifying the specific subtype.

这可以指许多类型之一。 通常是一个[QueryableAttribute](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.attributes.QueryableAttribute" \o "sqlalchemy.orm.attributes.QueryableAttribute)，它代表[MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty)处理属性事件。 但也可以是一个扩展类型，如[AssociationProxy](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.AssociationProxy" \o "sqlalchemy.ext.associationproxy.AssociationProxy)或[hybrid\_property](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/hybrid.html" \l "sqlalchemy.ext.hybrid.hybrid_property" \o "sqlalchemy.ext.hybrid.hybrid_property)。 [InspectionAttr.extension\_type](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.base.InspectionAttr.extension_type" \o "sqlalchemy.orm.base.InspectionAttr.extension_type)将引用一个标识特定子类型的常量。

**See also**

[Mapper.all\_orm\_descriptors](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper.all_orm_descriptors" \o "sqlalchemy.orm.mapper.Mapper.all_orm_descriptors)

**is\_clause\_element***= False*

True if this object is an instance of [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement).

**is\_instance***= False*

True if this object is an instance of [InstanceState](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState" \o "sqlalchemy.orm.state.InstanceState).

**is\_mapper***= False*

True if this object is an instance of [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper).

**is\_property***= False*

True if this object is an instance of [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty).

**is\_selectable***= False*

Return True if this object is an instance of [Selectable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Selectable" \o "sqlalchemy.sql.expression.Selectable).

*class*sqlalchemy.orm.base.**InspectionAttrInfo**

Bases: [sqlalchemy.orm.base.InspectionAttr](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.base.InspectionAttr" \o "sqlalchemy.orm.base.InspectionAttr)

Adds the .info attribute to [InspectionAttr](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.base.InspectionAttr" \o "sqlalchemy.orm.base.InspectionAttr).

The rationale for [InspectionAttr](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.base.InspectionAttr" \o "sqlalchemy.orm.base.InspectionAttr) vs. [InspectionAttrInfo](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.base.InspectionAttrInfo" \o "sqlalchemy.orm.base.InspectionAttrInfo) is that the former is compatible as a mixin for classes that specify \_\_slots\_\_; this is essentially an implementation artifact.

**info**

Info dictionary associated with the object, allowing user-defined data to be associated with this [InspectionAttr](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.base.InspectionAttr" \o "sqlalchemy.orm.base.InspectionAttr).

The dictionary is generated when first accessed. Alternatively, it can be specified as a constructor argument to the [column\_property()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_columns.html" \l "sqlalchemy.orm.column_property" \o "sqlalchemy.orm.column_property), [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship), or [composite()](http://docs.sqlalchemy.org/en/rel_1_1/orm/composites.html" \l "sqlalchemy.orm.composite" \o "sqlalchemy.orm.composite) functions.

*New in version 0.8:*Added support for .info to all [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty) subclasses.

*Changed in version 1.0.0:*[MapperProperty.info](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "MapperProperty.info" \o "MapperProperty.info) is also available on extension types via the [InspectionAttrInfo.info](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.base.InspectionAttrInfo.info" \o "sqlalchemy.orm.base.InspectionAttrInfo.info) attribute, so that it can apply to a wider variety of ORM and extension constructs.

**See also**

[QueryableAttribute.info](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.attributes.QueryableAttribute.info" \o "sqlalchemy.orm.attributes.QueryableAttribute.info)

[SchemaItem.info](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem.info" \o "sqlalchemy.schema.SchemaItem.info)

*class*sqlalchemy.orm.state.**InstanceState**(*obj*, *manager*)

Bases: [sqlalchemy.orm.base.InspectionAttr](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.base.InspectionAttr" \o "sqlalchemy.orm.base.InspectionAttr)

tracks state information at the instance level.

跟踪实例级的状态信息。

The [InstanceState](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState" \o "sqlalchemy.orm.state.InstanceState) is a key object used by the SQLAlchemy ORM in order to track the state of an object; it is created the moment an object is instantiated, typically as a result of [instrumentation](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-instrumentation) which SQLAlchemy applies to the \_\_init\_\_() method of the class.

InstanceState是SQLAlchemy ORM用于跟踪对象状态的关键对象; 它是在对象被实例化的时刻创建的，通常是由SQLAlchemy应用于类的\_\_init \_\_()方法的[instrumentation](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-instrumentation)的结果。

[InstanceState](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState" \o "sqlalchemy.orm.state.InstanceState) is also a semi-public object, available for runtime inspection as to the state of a mapped instance, including information such as its current status within a particular [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) and details about data on individual attributes. The public API in order to acquire a [InstanceState](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState" \o "sqlalchemy.orm.state.InstanceState) object is to use the [inspect()](http://docs.sqlalchemy.org/en/rel_1_1/core/inspection.html" \l "sqlalchemy.inspection.inspect" \o "sqlalchemy.inspection.inspect) system:

InstanceState也是半公开对象，可用于对映射实例的状态进行运行时检查，包括特定会话中的当前状态以及有关各个属性的数据的详细信息。 为了获取一个InstanceState对象，公共API是使用inspect()系统：

**>>> from** **sqlalchemy** **import** inspect

**>>>** insp = inspect(some\_mapped\_object)

**See also**

[Runtime Inspection API](http://docs.sqlalchemy.org/en/rel_1_1/core/inspection.html)

**attrs**

Return a namespace representing each attribute on the mapped object, including its current value and history.

The returned object is an instance of [AttributeState](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.AttributeState" \o "sqlalchemy.orm.state.AttributeState). This object allows inspection of the current data within an attribute as well as attribute history since the last flush.

**callables***= ()*

A namespace where a per-state loader callable can be associated.

In SQLAlchemy 1.0, this is only used for lazy loaders / deferred loaders that were set up via query option.

Previously, callables was used also to indicate expired attributes by storing a link to the InstanceState itself in this dictionary. This role is now handled by the expired\_attributes set.

**deleted**

Return true if the object is [deleted](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-deleted).

An object that is in the deleted state is guaranteed to not be within the [Session.identity\_map](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.identity_map" \o "sqlalchemy.orm.session.Session.identity_map) of its parent [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session); however if the session's transaction is rolled back, the object will be restored to the persistent state and the identity map.

**Note**

The [InstanceState.deleted](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState.deleted" \o "sqlalchemy.orm.state.InstanceState.deleted) attribute refers to a specific state of the object that occurs between the "persistent" and "detached" states; once the object is [detached](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-detached), the [InstanceState.deleted](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState.deleted" \o "sqlalchemy.orm.state.InstanceState.deleted) attribute ****no longer returns True****; in order to detect that a state was deleted, regardless of whether or not the object is associated with a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session), use the [InstanceState.was\_deleted](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState.was_deleted" \o "sqlalchemy.orm.state.InstanceState.was_deleted) accessor.

**See also**

[Quickie Intro to Object States](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_state_management.html" \l "session-object-states)

**detached**

Return true if the object is [detached](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-detached).

**See also**

[Quickie Intro to Object States](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_state_management.html" \l "session-object-states)

**dict**

Return the instance dict used by the object.

Under normal circumstances, this is always synonymous with the \_\_dict\_\_ attribute of the mapped object, unless an alternative instrumentation system has been configured.

In the case that the actual object has been garbage collected, this accessor returns a blank dictionary.

**expired\_attributes***= None*

The set of keys which are 'expired' to be loaded by the manager's deferred scalar loader, assuming no pending changes.

see also the unmodified collection which is intersected against this set when a refresh operation occurs.

**has\_identity**

Return True if this object has an identity key.

This should always have the same value as the expression state.persistent or state.detached.

**identity**

Return the mapped identity of the mapped object. This is the primary key identity as persisted by the ORM which can always be passed directly to[Query.get()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.get" \o "sqlalchemy.orm.query.Query.get).

Returns None if the object has no primary key identity.

**Note**

An object which is [transient](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-transient) or [pending](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-pending) does ****not**** have a mapped identity until it is flushed, even if its attributes include primary key values.

**identity\_key**

Return the identity key for the mapped object.

This is the key used to locate the object within the [Session.identity\_map](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.identity_map" \o "sqlalchemy.orm.session.Session.identity_map) mapping. It contains the identity as returned by [identity](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState.identity" \o "sqlalchemy.orm.state.InstanceState.identity) within it.

**mapper**

Return the [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) used for this mapepd object.

**object**

Return the mapped object represented by this [InstanceState](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState" \o "sqlalchemy.orm.state.InstanceState).

**pending**

Return true if the object is[pending](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-pending).

**See also**

[Quickie Intro to Object States](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_state_management.html" \l "session-object-states)

**persistent**

Return true if the object is[persistent](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-persistent).

An object that is in the persistent state is guaranteed to be within the [Session.identity\_map](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session.identity_map" \o "sqlalchemy.orm.session.Session.identity_map) of its parent [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session).

*Changed in version 1.1:*The [InstanceState.persistent](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState.persistent" \o "sqlalchemy.orm.state.InstanceState.persistent) accessor no longer returns True for an object that was "deleted" within a flush; use the [InstanceState.deleted](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState.deleted" \o "sqlalchemy.orm.state.InstanceState.deleted) accessor to detect this state. This allows the "persistent" state to guarantee membership in the identity map.

**See also**

[Quickie Intro to Object States](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_state_management.html" \l "session-object-states)

**session**

Return the owning [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) for this instance, or None if none available.

Note that the result here can in some cases be *different* from that of obj in session; an object that's been deleted will report as not in session, however if the transaction is still in progress, this attribute will still refer to that session. Only when the transaction is completed does the object become fully detached under normal circumstances.

**transient**

Return true if the object is [transient](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-transient).

**See also**

[Quickie Intro to Object States](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_state_management.html" \l "session-object-states)

**unloaded**

Return the set of keys which do not have a loaded value.

This includes expired attributes and any other attribute that was never populated or modified.

**unmodified**

Return the set of keys which have no uncommitted changes

**unmodified\_intersection**(*keys*)

Return self.unmodified.intersection(keys).

**was\_deleted**

Return True if this object is or was previously in the "deleted" state and has not been reverted to persistent.

This flag returns True once the object was deleted in flush. When the object is expunged from the session either explicitly or via transaction commit and enters the "detached" state, this flag will continue to report True.

*New in version 1.1:*- added a local method form of [orm.util.was\_deleted()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.util.was_deleted" \o "sqlalchemy.orm.util.was_deleted).

**See also**

[InstanceState.deleted](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.state.InstanceState.deleted" \o "sqlalchemy.orm.state.InstanceState.deleted) - refers to the "deleted" state

[orm.util.was\_deleted()](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.util.was_deleted" \o "sqlalchemy.orm.util.was_deleted) - standalone function

[Quickie Intro to Object States](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_state_management.html" \l "session-object-states)

*class*sqlalchemy.orm.attributes.**InstrumentedAttribute**(*class\_*, *key*, *impl=None*, *comparator=None*, *parententity=None*, *of\_type=None*)

Bases: [sqlalchemy.orm.attributes.QueryableAttribute](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.attributes.QueryableAttribute" \o "sqlalchemy.orm.attributes.QueryableAttribute)

Class bound instrumented attribute which adds basic [descriptor](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-descriptor) methods.

See [QueryableAttribute](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.attributes.QueryableAttribute" \o "sqlalchemy.orm.attributes.QueryableAttribute) for a description of most features.

**\_\_delete\_\_**(*instance*)

**\_\_get\_\_**(*instance*, *owner*)

**\_\_set\_\_**(*instance*, *value*)

sqlalchemy.orm.interfaces.**MANYTOONE***= symbol('MANYTOONE')*

Indicates the many-to-one direction for a [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship).

This symbol is typically used by the internals but may be exposed within certain API features.

sqlalchemy.orm.interfaces.**MANYTOMANY***= symbol('MANYTOMANY')*

Indicates the many-to-many direction for a [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship).

This symbol is typically used by the internals but may be exposed within certain API features.

*class*sqlalchemy.orm.interfaces.**MapperProperty**

Bases: sqlalchemy.orm.base.\_MappedAttribute, [sqlalchemy.orm.base.InspectionAttr](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.base.InspectionAttr" \o "sqlalchemy.orm.base.InspectionAttr), sqlalchemy.util.langhelpers.MemoizedSlots

Represent a particular class attribute mapped by [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper).

The most common occurrences of [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty) are the mapped [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column), which is represented in a mapping as an instance of [ColumnProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.ColumnProperty" \o "sqlalchemy.orm.properties.ColumnProperty), and a reference to another class produced by [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship), represented in the mapping as an instance of [RelationshipProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty" \o "sqlalchemy.orm.properties.RelationshipProperty).

**info**

Info dictionary associated with the object, allowing user-defined data to be associated with this [InspectionAttr](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.base.InspectionAttr" \o "sqlalchemy.orm.base.InspectionAttr).

The dictionary is generated when first accessed. Alternatively, it can be specified as a constructor argument to the [column\_property()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_columns.html" \l "sqlalchemy.orm.column_property" \o "sqlalchemy.orm.column_property), [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship), or [composite()](http://docs.sqlalchemy.org/en/rel_1_1/orm/composites.html" \l "sqlalchemy.orm.composite" \o "sqlalchemy.orm.composite) functions.

*New in version 0.8:*Added support for .info to all [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty) subclasses.

*Changed in version 1.0.0:*InspectionAttr.info moved from [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty) so that it can apply to a wider variety of ORM and extension constructs.

**See also**

[QueryableAttribute.info](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.attributes.QueryableAttribute.info" \o "sqlalchemy.orm.attributes.QueryableAttribute.info)

[SchemaItem.info](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem.info" \o "sqlalchemy.schema.SchemaItem.info)

**cascade***= frozenset([])*

The set of 'cascade' attribute names.

This collection is checked before the 'cascade\_iterator' method is called.

The collection typically only applies to a RelationshipProperty.

**cascade\_iterator**(*type\_*, *state*, *visited\_instances=None*, *halt\_on=None*)

Iterate through instances related to the given instance for a particular 'cascade', starting with this MapperProperty.

Return an iterator3-tuples (instance, mapper, state).

Note that the 'cascade' collection on this MapperProperty is checked first for the given type before cascade\_iterator is called.

This method typically only applies to RelationshipProperty.

**class\_attribute**

Return the class-bound descriptor corresponding to this [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty).

This is basically a getattr() call:

**return** getattr(self.parent.class\_, self.key)

I.e. if this [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty) were named addresses, and the class to which it is mapped is User, this sequence is possible:

**>>> from** **sqlalchemy** **import** inspect

**>>>** mapper = inspect(User)

**>>>** addresses\_property = mapper.attrs.addresses

**>>>** addresses\_property.class\_attribute **is** User.addresses

True

**>>>** User.addresses.property **is** addresses\_property

True

**create\_row\_processor**(*context*, *path*, *mapper*, *result*, *adapter*, *populators*)

Produce row processing functions and append to the given set of populators lists.

**do\_init**()

Perform subclass-specific initialization post-mapper-creation steps.

This is a template method called by the MapperProperty object's init() method.

**init**()

Called after all mappers are created to assemble relationships between mappers and perform other post-mapper-creation initialization steps.

**instrument\_class**(*mapper*)

Hook called by the Mapper to the property to initiate instrumentation of the class attribute managed by this MapperProperty.

Hook通过Mapper调用该属性来启动由此MapperProperty管理的类属性的检测。

The MapperProperty here will typically call out to the attributes module to set up an InstrumentedAttribute.

这里的MapperProperty通常会调用属性模块来设置InstrumentedAttribute。

This step is the first of two steps to set up an InstrumentedAttribute, and is called early in the mapper setup process.

此步骤是设置InstrumentedAttribute的两个步骤中的第一步，并且在映射器设置过程的早期被称为。

The second step is typically the init\_class\_attribute step, called from StrategizedProperty via the post\_instrument\_class() hook. This step assigns additional state to the InstrumentedAttribute (specifically the "impl") which has been determined after the MapperProperty has determined what kind of persistence management it needs to do (e.g. scalar, object, collection, etc).

第二步通常是init\_class\_attribute步骤，通过post\_instrument\_class()钩子从StrategizedProperty调用。 此步骤为MapperProperty已经确定需要做什么样的持久性管理(例如标量，对象，集合等)之后已经确定的InstrumentedAttribute(具体为“impl”)分配附加状态。

**is\_property***= True*

Part of the InspectionAttr interface; states this object is a mapper property.

**merge**(*session*, *source\_state*, *source\_dict*, *dest\_state*, *dest\_dict*, *load*, *\_recursive*, *\_resolve\_conflict\_map*)

Merge the attribute represented by this MapperProperty from source to destination object.

**post\_instrument\_class**(*mapper*)

Perform instrumentation adjustments that need to occur after init() has completed.

The given Mapper is the Mapper invoking the operation, which may not be the same Mapper as self.parent in an inheritance scenario; however, Mapper will always at least be a sub-mapper of self.parent.

This method is typically used by StrategizedProperty, which delegates it to LoaderStrategy.init\_class\_attribute() to perform final setup on the class-bound InstrumentedAttribute.

**set\_parent**(*parent*, *init*)

Set the parent mapper that references this MapperProperty.

This method is overridden by some subclasses to perform extra setup when the mapper is first known.

**setup**(*context*, *entity*, *path*, *adapter*, *\*\*kwargs*)

Called by Query for the purposes of constructing a SQL statement.

Each MapperProperty associated with the target mapper processes the statement referenced by the query context, adding columns and/or criterion as appropriate.

sqlalchemy.orm.interfaces.**NOT\_EXTENSION***= symbol('NOT\_EXTENSION')*

Symbol indicating an InspectionAttr that's not part of sqlalchemy.ext.

Is assigned to the [InspectionAttr.extension\_type](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.base.InspectionAttr.extension_type" \o "sqlalchemy.orm.base.InspectionAttr.extension_type) attibute.

sqlalchemy.orm.interfaces.**ONETOMANY***= symbol('ONETOMANY')*

Indicates the one-to-many direction for a [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship).

This symbol is typically used by the internals but may be exposed within certain API features.

*class*sqlalchemy.orm.interfaces.**PropComparator**(*prop*, *parentmapper*, *adapt\_to\_entity=None*)

Bases: [sqlalchemy.sql.operators.ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Defines SQL operators for [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty) objects.

SQLAlchemy allows for operators to be redefined at both the Core and ORM level. [PropComparator](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator" \o "sqlalchemy.orm.interfaces.PropComparator) is the base class of operator redefinition for ORM-level operations, including those of [ColumnProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.ColumnProperty" \o "sqlalchemy.orm.properties.ColumnProperty), [RelationshipProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty" \o "sqlalchemy.orm.properties.RelationshipProperty), and [CompositeProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.descriptor_props.CompositeProperty" \o "sqlalchemy.orm.descriptor_props.CompositeProperty).

**Note**

With the advent of Hybrid properties introduced in SQLAlchemy 0.7, as well as Core-level operator redefinition in SQLAlchemy 0.8, the use case for user-defined [PropComparator](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator" \o "sqlalchemy.orm.interfaces.PropComparator) instances is extremely rare. See [Hybrid Attributes](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/hybrid.html) as well as [Redefining and Creating New Operators](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "types-operators).

User-defined subclasses of [PropComparator](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator" \o "sqlalchemy.orm.interfaces.PropComparator) may be created. The built-in Python comparison and math operator methods, such as[operators.ColumnOperators.\_\_eq\_\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.__eq__" \o "sqlalchemy.sql.operators.ColumnOperators.__eq__), [operators.ColumnOperators.\_\_lt\_\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.__lt__" \o "sqlalchemy.sql.operators.ColumnOperators.__lt__), and [operators.ColumnOperators.\_\_add\_\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.__add__" \o "sqlalchemy.sql.operators.ColumnOperators.__add__), can be overridden to provide new operator behavior. The custom [PropComparator](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator" \o "sqlalchemy.orm.interfaces.PropComparator) is passed to the [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty) instance via the comparator\_factoryargument. In each case, the appropriate subclass of [PropComparator](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator" \o "sqlalchemy.orm.interfaces.PropComparator) should be used:

*# definition of custom PropComparator subclasses*

**from** **sqlalchemy.orm.properties** **import** \

ColumnProperty,\

CompositeProperty,\

RelationshipProperty

**class** **MyColumnComparator**(ColumnProperty.Comparator):

**def** \_\_eq\_\_(self, other):

**return** self.\_\_clause\_element\_\_() == other

**class** **MyRelationshipComparator**(RelationshipProperty.Comparator):

**def** any(self, expression):

"define the 'any' operation"

*# ...*

**class** **MyCompositeComparator**(CompositeProperty.Comparator):

**def** \_\_gt\_\_(self, other):

"redefine the 'greater than' operation"

**return** sql.and\_(\*[a>b **for** a, b **in**

zip(self.\_\_clause\_element\_\_().clauses,

other.\_\_composite\_values\_\_())])

*# application of custom PropComparator subclasses*

**from** **sqlalchemy.orm** **import** column\_property, relationship, composite

**from** **sqlalchemy** **import** Column, String

**class** **SomeMappedClass**(Base):

some\_column = column\_property(Column("some\_column", String),

comparator\_factory=MyColumnComparator)

some\_relationship = relationship(SomeOtherClass,

comparator\_factory=MyRelationshipComparator)

some\_composite = composite(

Column("a", String), Column("b", String),

comparator\_factory=MyCompositeComparator

)

Note that for column-level operator redefinition, it's usually simpler to define the operators at the Core level, using the [TypeEngine.comparator\_factory](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.comparator_factory" \o "sqlalchemy.types.TypeEngine.comparator_factory)attribute. See [Redefining and Creating New Operators](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "types-operators) for more detail.

See also:

[ColumnProperty.Comparator](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.ColumnProperty.Comparator" \o "sqlalchemy.orm.properties.ColumnProperty.Comparator)

[RelationshipProperty.Comparator](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator)

[CompositeProperty.Comparator](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.descriptor_props.CompositeProperty.Comparator" \o "sqlalchemy.orm.descriptor_props.CompositeProperty.Comparator)

[ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

[Redefining and Creating New Operators](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "types-operators)

[TypeEngine.comparator\_factory](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.comparator_factory" \o "sqlalchemy.types.TypeEngine.comparator_factory)

**\_\_eq\_\_**(*other*)

*inherited from the* [\_\_eq\_\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.__eq__" \o "sqlalchemy.sql.operators.ColumnOperators.__eq__) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the == operator.

In a column context, produces the clause a = b. If the target is None, produces a IS NULL.

**\_\_le\_\_**(*other*)

*inherited from the* [\_\_le\_\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.__le__" \o "sqlalchemy.sql.operators.ColumnOperators.__le__) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the <= operator.

In a column context, produces the clause a <= b.

**\_\_lt\_\_**(*other*)

*inherited from the* [\_\_lt\_\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.__lt__" \o "sqlalchemy.sql.operators.ColumnOperators.__lt__) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the < operator.

In a column context, produces the clause a < b.

**\_\_ne\_\_**(*other*)

*inherited from the* [\_\_ne\_\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.__ne__" \o "sqlalchemy.sql.operators.ColumnOperators.__ne__) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the != operator.

In a column context, produces the clause a != b. If the target is None, produces a IS NOT NULL.

**adapt\_to\_entity**(*adapt\_to\_entity*)

Return a copy of this PropComparator which will use the given [AliasedInsp](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.util.AliasedInsp" \o "sqlalchemy.orm.util.AliasedInsp) to produce corresponding expressions.

**adapter**

Produce a callable that adapts column expressions to suit an aliased version of this comparator.

**all\_**()

*inherited from the* [all\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.all_" \o "sqlalchemy.sql.operators.ColumnOperators.all_) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Produce a [all\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.all_" \o "sqlalchemy.sql.expression.all_) clause against the parent object.

*New in version 1.1.*

**any**(*criterion=None*, *\*\*kwargs*)

Return true if this collection contains any member that meets the given criterion.

The usual implementation of any() is [RelationshipProperty.Comparator.any()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator.any" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator.any).

|  |  |
| --- | --- |
| **Parameters:** | * ****criterion**** – an optional ClauseElement formulated against the member class' table or attributes. * ****\*\*kwargs**** – key/value pairs corresponding to member class attribute names which will be compared via equality to the corresponding values. |

**any\_**()

*inherited from the* [any\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.any_" \o "sqlalchemy.sql.operators.ColumnOperators.any_) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Produce a [any\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.any_" \o "sqlalchemy.sql.expression.any_) clause against the parent object.

*New in version 1.1.*

**asc**()

*inherited from the* [asc()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.asc" \o "sqlalchemy.sql.operators.ColumnOperators.asc) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Produce a [asc()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.asc" \o "sqlalchemy.sql.expression.asc) clause against the parent object.

**between**(*cleft*, *cright*, *symmetric=False*)

*inherited from the* [between()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.between" \o "sqlalchemy.sql.operators.ColumnOperators.between) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Produce a [between()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.between" \o "sqlalchemy.sql.expression.between) clause against the parent object, given the lower and upper range.

**collate**(*collation*)

*inherited from the* [collate()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.collate" \o "sqlalchemy.sql.operators.ColumnOperators.collate) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Produce a [collate()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.collate" \o "sqlalchemy.sql.expression.collate) clause against the parent object, given the collation string.

**concat**(*other*)

*inherited from the* [concat()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.concat" \o "sqlalchemy.sql.operators.ColumnOperators.concat) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the 'concat' operator.

In a column context, produces the clause a || b, or uses the concat() operator on MySQL.

**contains**(*other*, *\*\*kwargs*)

*inherited from the* [contains()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.contains" \o "sqlalchemy.sql.operators.ColumnOperators.contains) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the 'contains' operator.

In a column context, produces the clause LIKE '%<other>%'

**desc**()

*inherited from the* [desc()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.desc" \o "sqlalchemy.sql.operators.ColumnOperators.desc) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Produce a [desc()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.desc" \o "sqlalchemy.sql.expression.desc) clause against the parent object.

**distinct**()

*inherited from the* [distinct()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.distinct" \o "sqlalchemy.sql.operators.ColumnOperators.distinct) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Produce a [distinct()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.distinct" \o "sqlalchemy.sql.expression.distinct) clause against the parent object.

**endswith**(*other*, *\*\*kwargs*)

*inherited from the* [endswith()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.endswith" \o "sqlalchemy.sql.operators.ColumnOperators.endswith) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the 'endswith' operator.

In a column context, produces the clause LIKE '%<other>'

**has**(*criterion=None*, *\*\*kwargs*)

Return true if this element references a member which meets the given criterion.

The usual implementation of has() is [RelationshipProperty.Comparator.has()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator.has" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator.has).

|  |  |
| --- | --- |
| **Parameters:** | * ****criterion**** – an optional ClauseElement formulated against the member class' table or attributes. * ****\*\*kwargs**** – key/value pairs corresponding to member class attribute names which will be compared via equality to the corresponding values. |

**ilike**(*other*, *escape=None*)

*inherited from the* [ilike()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.ilike" \o "sqlalchemy.sql.operators.ColumnOperators.ilike) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the ilike operator, e.g. case insensitive LIKE.

In a column context, produces an expression either of the form:

lower(a) LIKE lower(other)

Or on backends that support the ILIKE operator:

a ILIKE other

E.g.:

stmt = select([sometable]).\

where(sometable.c.column.ilike("*%f*oobar%"))

|  |  |
| --- | --- |
| **Parameters:** | * ****other**** – expression to be compared * ****escape –****optional escape character, renders the ESCAPE keyword, e.g.:   somecolumn.ilike("foo/%bar", escape="/") |

**See also**

[ColumnOperators.like()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.like" \o "sqlalchemy.sql.operators.ColumnOperators.like)

**in\_**(*other*)

*inherited from the* [in\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.in_" \o "sqlalchemy.sql.operators.ColumnOperators.in_) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the in operator.

In a column context, produces the clause a IN other. "other" may be a tuple/list of column expressions, or a [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) construct.

**is\_**(*other*)

*inherited from the* [is\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.is_" \o "sqlalchemy.sql.operators.ColumnOperators.is_) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the IS operator.

Normally, IS is generated automatically when comparing to a value of None, which resolves to NULL. However, explicit usage of IS may be desirable if comparing to boolean values on certain platforms.

*New in version 0.7.9.*

**See also**

[ColumnOperators.isnot()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.isnot" \o "sqlalchemy.sql.operators.ColumnOperators.isnot)

**is\_distinct\_from**(*other*)

*inherited from the* [is\_distinct\_from()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.is_distinct_from" \o "sqlalchemy.sql.operators.ColumnOperators.is_distinct_from) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the IS DISTINCT FROM operator.

Renders "a IS DISTINCT FROM b" on most platforms; on some such as SQLite may render "a IS NOT b".

*New in version 1.1.*

**isnot**(*other*)

*inherited from the* [isnot()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.isnot" \o "sqlalchemy.sql.operators.ColumnOperators.isnot) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the IS NOT operator.

Normally, IS NOT is generated automatically when comparing to a value of None, which resolves to NULL. However, explicit usage of IS NOT may be desirable if comparing to boolean values on certain platforms.

*New in version 0.7.9.*

**See also**

[ColumnOperators.is\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.is_" \o "sqlalchemy.sql.operators.ColumnOperators.is_)

**isnot\_distinct\_from**(*other*)

*inherited from the* [isnot\_distinct\_from()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.isnot_distinct_from" \o "sqlalchemy.sql.operators.ColumnOperators.isnot_distinct_from) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the IS NOT DISTINCT FROM operator.

Renders "a IS NOT DISTINCT FROM b" on most platforms; on some such as SQLite may render "a IS b".

*New in version 1.1.*

**like**(*other*, *escape=None*)

*inherited from the* [like()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.like" \o "sqlalchemy.sql.operators.ColumnOperators.like) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the like operator.

In a column context, produces the expression:

a LIKE other

E.g.:

stmt = select([sometable]).\

where(sometable.c.column.like("*%f*oobar%"))

|  |  |
| --- | --- |
| **Parameters:** | * ****other**** – expression to be compared * ****escape –****optional escape character, renders the ESCAPE keyword, e.g.:   somecolumn.like("foo/%bar", escape="/") |

**See also**

[ColumnOperators.ilike()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.ilike" \o "sqlalchemy.sql.operators.ColumnOperators.ilike)

**match**(*other*, *\*\*kwargs*)

*inherited from the* [match()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.match" \o "sqlalchemy.sql.operators.ColumnOperators.match) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implements a database-specific 'match' operator.

[match()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.match" \o "sqlalchemy.sql.operators.ColumnOperators.match) attempts to resolve to a MATCH-like function or operator provided by the backend. Examples include:

* PostgreSQL - renders x @@ to\_tsquery(y)
* MySQL - renders MATCH (x) AGAINST (y IN BOOLEAN MODE)
* Oracle - renders CONTAINS(x, y)
* other backends may provide special implementations.
* Backends without any special implementation will emit the operator as "MATCH". This is compatible with SQlite, for example.

**notilike**(*other*, *escape=None*)

*inherited from the* [notilike()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.notilike" \o "sqlalchemy.sql.operators.ColumnOperators.notilike) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

implement the NOT ILIKE operator.

This is equivalent to using negation with [ColumnOperators.ilike()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.ilike" \o "sqlalchemy.sql.operators.ColumnOperators.ilike), i.e. ~x.ilike(y).

*New in version 0.8.*

**See also**

[ColumnOperators.ilike()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.ilike" \o "sqlalchemy.sql.operators.ColumnOperators.ilike)

**notin\_**(*other*)

*inherited from the* [notin\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.notin_" \o "sqlalchemy.sql.operators.ColumnOperators.notin_) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

implement the NOT IN operator.

This is equivalent to using negation with [ColumnOperators.in\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.in_" \o "sqlalchemy.sql.operators.ColumnOperators.in_), i.e. ~x.in\_(y).

*New in version 0.8.*

**See also**

[ColumnOperators.in\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.in_" \o "sqlalchemy.sql.operators.ColumnOperators.in_)

**notlike**(*other*, *escape=None*)

*inherited from the* [notlike()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.notlike" \o "sqlalchemy.sql.operators.ColumnOperators.notlike) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

implement the NOT LIKE operator.

This is equivalent to using negation with [ColumnOperators.like()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.like" \o "sqlalchemy.sql.operators.ColumnOperators.like), i.e. ~x.like(y).

*New in version 0.8.*

**See also**

[ColumnOperators.like()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.like" \o "sqlalchemy.sql.operators.ColumnOperators.like)

**nullsfirst**()

*inherited from the* [nullsfirst()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.nullsfirst" \o "sqlalchemy.sql.operators.ColumnOperators.nullsfirst) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Produce a [nullsfirst()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.nullsfirst" \o "sqlalchemy.sql.expression.nullsfirst) clause against the parent object.

**nullslast**()

*inherited from the* [nullslast()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.nullslast" \o "sqlalchemy.sql.operators.ColumnOperators.nullslast) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Produce a [nullslast()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.nullslast" \o "sqlalchemy.sql.expression.nullslast) clause against the parent object.

**of\_type**(*class\_*)

Redefine this object in terms of a polymorphic subclass.

Returns a new PropComparator from which further criterion can be evaluated.

e.g.:

query.join(Company.employees.of\_type(Engineer)).\

filter(Engineer.name=='foo')

|  |  |
| --- | --- |
| **Parameters:** | ****class\_**** – a class or mapper indicating that criterion will be against this specific subclass. |

**op**(*opstring*, *precedence=0*, *is\_comparison=False*)

*inherited from the* [op()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.Operators.op" \o "sqlalchemy.sql.operators.Operators.op) *method of* [Operators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.Operators" \o "sqlalchemy.sql.operators.Operators)

produce a generic operator function.

e.g.:

somecolumn.op("\*")(5)

produces:

somecolumn \* 5

This function can also be used to make bitwise operators explicit. For example:

somecolumn.op('&')(0xff)

is a bitwise AND of the value in somecolumn.

|  |  |
| --- | --- |
| **Parameters:** | * ****operator**** – a string which will be output as the infix operator between this element and the expression passed to the generated function.将作为该元素和传递给生成函数的表达式之间的中缀运算符输出的字符串。 * ****precedence –****precedence to apply to the operator, when parenthesizing expressions. A lower number will cause the expression to be parenthesized when applied against another operator with higher precedence. The default value of 0 is lower than all operators except for the comma (,) and AS operators. A value of 100 will be higher or equal to all operators, and -100 will be lower than or equal to all operators. * 优先级应用于运算符，括号中表达式。 较低的数字会导致表达式被对应于较高优先级的另一个运算符时被括号。 默认值为0，除逗号(，)和AS运算符之外的所有运算符都低。 值100将高于或等于所有运算符，-100将低于或等于所有运算符。   *New in version 0.8:*- added the 'precedence' argument.   * ****is\_comparison –****if True, the operator will be considered as a "comparison" operator, that is which evaluates to a boolean true/false value, like ==, >, etc. This flag should be set so that ORM relationships can establish that the operator is a comparison operator when used in a custom join condition.如果为True，则运算符将被视为“比较”运算符，即运算符的值为布尔值true / false，如==，>等。此标志应设置为使ORM关系可以确定运算符 一个比较运算符，用于自定义连接条件。   *New in version 0.9.2:*- added the [Operators.op.is\_comparison](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.Operators.op.params.is_comparison" \o "sqlalchemy.sql.operators.Operators.op) flag. |

**See also**

[Redefining and Creating New Operators](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "types-operators)

[Using custom operators in join conditions](http://docs.sqlalchemy.org/en/rel_1_1/orm/join_conditions.html" \l "relationship-custom-operator)

**operate**(*op*, *\*other*, *\*\*kwargs*)

*inherited from the* [operate()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.Operators.operate" \o "sqlalchemy.sql.operators.Operators.operate) *method of* [Operators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.Operators" \o "sqlalchemy.sql.operators.Operators)

Operate on an argument.

This is the lowest level of operation, raises NotImplementedError by default.

Overriding this on a subclass can allow common behavior to be applied to all operations. For example, overriding [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators) to apply func.lower() to the left and right side:

**class** **MyComparator**(ColumnOperators):

**def** operate(self, op, other):

**return** op(func.lower(self), func.lower(other))

|  |  |
| --- | --- |
| **Parameters:** | * ****op**** – Operator callable. * ****\*other**** – the 'other' side of the operation. Will be a single scalar for most operations. * ****\*\*kwargs**** – modifiers. These may be passed by special operators such as ColumnOperators.contains(). |

**reverse\_operate**(*op*, *other*, *\*\*kwargs*)

*inherited from the* [reverse\_operate()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.Operators.reverse_operate" \o "sqlalchemy.sql.operators.Operators.reverse_operate) *method of* [Operators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.Operators" \o "sqlalchemy.sql.operators.Operators)

Reverse operate on an argument.

Usage is the same as operate().

**startswith**(*other*, *\*\*kwargs*)

*inherited from the* [startswith()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.startswith" \o "sqlalchemy.sql.operators.ColumnOperators.startswith) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the startwith operator.

In a column context, produces the clause LIKE '<other>%'

*class*sqlalchemy.orm.properties.**RelationshipProperty**(*argument*, *secondary=None*, *primaryjoin=None*, *secondaryjoin=None*, *foreign\_keys=None*, *uselist=None*, *order\_by=False*, *backref=None*, *back\_populates=None*, *post\_update=False*, *cascade=False*, *extension=None*, *viewonly=False*, *lazy=True*, *collection\_class=None*, *passive\_deletes=False*, *passive\_updates=True*, *remote\_side=None*, *enable\_typechecks=True*, *join\_depth=None*, *comparator\_factory=None*, *single\_parent=False*, *innerjoin=False*, *distinct\_target\_key=None*, *doc=None*, *active\_history=False*, *cascade\_backrefs=True*, *load\_on\_pending=False*, *bake\_queries=True*, *\_local\_remote\_pairs=None*, *query\_class=None*, *info=None*)

Bases: sqlalchemy.orm.interfaces.StrategizedProperty

Describes an object property that holds a single item or list of items that correspond to a related database table.

Public constructor is the [orm.relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) function.

See also:

[Relationship Configuration](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationships.html)

*class***Comparator**(*prop*, *parentmapper*, *adapt\_to\_entity=None*, *of\_type=None*)

Bases: [sqlalchemy.orm.interfaces.PropComparator](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator" \o "sqlalchemy.orm.interfaces.PropComparator)

Produce boolean, comparison, and other operators for [RelationshipProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty" \o "sqlalchemy.orm.properties.RelationshipProperty) attributes.

See the documentation for [PropComparator](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator" \o "sqlalchemy.orm.interfaces.PropComparator) for a brief overview of ORM level operator definition.

See also:

[PropComparator](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator" \o "sqlalchemy.orm.interfaces.PropComparator)

[ColumnProperty.Comparator](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.ColumnProperty.Comparator" \o "sqlalchemy.orm.properties.ColumnProperty.Comparator)

[ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

[Redefining and Creating New Operators](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "types-operators)

[TypeEngine.comparator\_factory](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.comparator_factory" \o "sqlalchemy.types.TypeEngine.comparator_factory)

**\_\_eq\_\_**(*other*)

Implement the == operator.

In a many-to-one context, such as:

MyClass.some\_prop == <some object>

this will typically produce a clause such as:

mytable.related\_id == <some id>

Where <some id> is the primary key of the given object.

The == operator provides partial functionality for non- many-to-one comparisons:

* Comparisons against collections are not supported. Use [contains()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator.contains" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator.contains).
* Compared to a scalar one-to-many, will produce a clause that compares the target columns in the parent to the given target.
* Compared to a scalar many-to-many, an alias of the association table will be rendered as well, forming a natural join that is part of the main body of the query. This will not work for queries that go beyond simple AND conjunctions of comparisons, such as those which use OR. Use explicit joins, outerjoins, or [has()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator.has" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator.has) for more comprehensive non-many-to-one scalar membership tests.
* Comparisons against None given in a one-to-many or many-to-many context produce a NOT EXISTS clause.

**\_\_init\_\_**(*prop*, *parentmapper*, *adapt\_to\_entity=None*, *of\_type=None*)

Construction of [RelationshipProperty.Comparator](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator) is internal to the ORM's attribute mechanics.

**\_\_le\_\_**(*other*)

*inherited from the* [\_\_le\_\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.__le__" \o "sqlalchemy.sql.operators.ColumnOperators.__le__) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the <= operator.

In a column context, produces the clause a <= b.

**\_\_lt\_\_**(*other*)

*inherited from the* [\_\_lt\_\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.__lt__" \o "sqlalchemy.sql.operators.ColumnOperators.__lt__) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the < operator.

In a column context, produces the clause a < b.

**\_\_ne\_\_**(*other*)

Implement the != operator.

In a many-to-one context, such as:

MyClass.some\_prop != <some object>

This will typically produce a clause such as:

mytable.related\_id != <some id>

Where <some id> is the primary key of the given object.

The != operator provides partial functionality for non- many-to-one comparisons:

* Comparisons against collections are not supported. Use [contains()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator.contains" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator.contains) in conjunction with [not\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.not_" \o "sqlalchemy.sql.expression.not_).
* Compared to a scalar one-to-many, will produce a clause that compares the target columns in the parent to the given target.
* Compared to a scalar many-to-many, an alias of the association table will be rendered as well, forming a natural join that is part of the main body of the query. This will not work for queries that go beyond simple AND conjunctions of comparisons, such as those which use OR. Use explicit joins, outerjoins, or [has()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator.has" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator.has) in conjunction with [not\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.not_" \o "sqlalchemy.sql.expression.not_) for more comprehensive non-many-to-one scalar membership tests.
* Comparisons against None given in a one-to-many or many-to-many context produce an EXISTS clause.

**adapter**

*inherited from the* [adapter](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator.adapter" \o "sqlalchemy.orm.interfaces.PropComparator.adapter) *attribute of* [PropComparator](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator" \o "sqlalchemy.orm.interfaces.PropComparator)

Produce a callable that adapts column expressions to suit an aliased version of this comparator.

**all\_**()

*inherited from the* [all\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.all_" \o "sqlalchemy.sql.operators.ColumnOperators.all_) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Produce a [all\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.all_" \o "sqlalchemy.sql.expression.all_) clause against the parent object.

*New in version 1.1.*

**any**(*criterion=None*, *\*\*kwargs*)

Produce an expression that tests a collection against particular criterion, using EXISTS.

An expression like:

session.query(MyClass).filter(

MyClass.somereference.any(SomeRelated.x==2))

Will produce a query like:

SELECT \* FROM my\_table WHEREEXISTS (SELECT 1 FROM related WHERE related.my\_id=my\_table.idAND related.x=2)

Because [any()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator.any" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator.any) uses a correlated subquery, its performance is not nearly as good when compared against large target tables as that of using a join.

[any()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator.any" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator.any) is particularly useful for testing for empty collections:

session.query(MyClass).filter(

~MyClass.somereference.any())

will produce:

SELECT \* FROM my\_table WHERENOT EXISTS (SELECT 1 FROM related WHERErelated.my\_id=my\_table.id)

[any()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator.any" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator.any) is only valid for collections, i.e. a [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) that has uselist=True. For scalar references, use [has()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator.has" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator.has).

**any\_**()

*inherited from the* [any\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.any_" \o "sqlalchemy.sql.operators.ColumnOperators.any_) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Produce a [any\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.any_" \o "sqlalchemy.sql.expression.any_) clause against the parent object.

*New in version 1.1.*

**asc**()

*inherited from the* [asc()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.asc" \o "sqlalchemy.sql.operators.ColumnOperators.asc) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Produce a [asc()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.asc" \o "sqlalchemy.sql.expression.asc) clause against the parent object.

**between**(*cleft*, *cright*, *symmetric=False*)

*inherited from the* [between()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.between" \o "sqlalchemy.sql.operators.ColumnOperators.between) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Produce a [between()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.between" \o "sqlalchemy.sql.expression.between) clause against the parent object, given the lower and upper range.

**collate**(*collation*)

*inherited from the* [collate()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.collate" \o "sqlalchemy.sql.operators.ColumnOperators.collate) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Produce a [collate()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.collate" \o "sqlalchemy.sql.expression.collate) clause against the parent object, given the collation string.

**concat**(*other*)

*inherited from the* [concat()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.concat" \o "sqlalchemy.sql.operators.ColumnOperators.concat) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the 'concat' operator.

In a column context, produces the clause a || b, or uses the concat() operator on MySQL.

**contains**(*other*, *\*\*kwargs*)

Return a simple expression that tests a collection for containment of a particular item.

[contains()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator.contains" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator.contains) is only valid for a collection, i.e. a [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) that implements one-to-many or many-to-many with uselist=True.

When used in a simple one-to-many context, an expression like:

MyClass.contains(other)

Produces a clause like:

mytable.id == <some id>

Where <some id> is the value of the foreign key attribute on other which refers to the primary key of its parent object. From this it follows that[contains()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator.contains" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator.contains) is very useful when used with simple one-to-many operations.

For many-to-many operations, the behavior of [contains()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator.contains" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator.contains) has more caveats. The association table will be rendered in the statement, producing an "implicit" join, that is, includes multiple tables in the FROM clause which are equated in the WHERE clause:

query(MyClass).filter(MyClass.contains(other))

Produces a query like:

SELECT \* FROM my\_table, my\_association\_table ASmy\_association\_table\_1 WHEREmy\_table.id = my\_association\_table\_1.parent\_idAND my\_association\_table\_1.child\_id = <some id>

Where <some id> would be the primary key of other. From the above, it is clear that [contains()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator.contains" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator.contains) will ****not**** work with many-to-many collections when used in queries that move beyond simple AND conjunctions, such as multiple [contains()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator.contains" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator.contains) expressions joined by OR. In such cases subqueries or explicit "outer joins" will need to be used instead. See [any()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator.any" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator.any) for a less-performant alternative using EXISTS, or refer to [Query.outerjoin()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.outerjoin" \o "sqlalchemy.orm.query.Query.outerjoin) as well as [Querying with Joins](http://docs.sqlalchemy.org/en/rel_1_1/orm/tutorial.html" \l "ormtutorial-joins) for more details on constructing outer joins.

**desc**()

*inherited from the* [desc()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.desc" \o "sqlalchemy.sql.operators.ColumnOperators.desc) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Produce a [desc()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.desc" \o "sqlalchemy.sql.expression.desc) clause against the parent object.

**distinct**()

*inherited from the* [distinct()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.distinct" \o "sqlalchemy.sql.operators.ColumnOperators.distinct) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Produce a [distinct()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.distinct" \o "sqlalchemy.sql.expression.distinct) clause against the parent object.

**endswith**(*other*, *\*\*kwargs*)

*inherited from the* [endswith()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.endswith" \o "sqlalchemy.sql.operators.ColumnOperators.endswith) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the 'endswith' operator.

In a column context, produces the clause LIKE '%<other>'

**has**(*criterion=None*, *\*\*kwargs*)

Produce an expression that tests a scalar reference against particular criterion, using EXISTS.

An expression like:

session.query(MyClass).filter(

MyClass.somereference.has(SomeRelated.x==2))

Will produce a query like:

SELECT \* FROM my\_table WHEREEXISTS (SELECT 1 FROM related WHERErelated.id==my\_table.related\_id AND related.x=2)

Because [has()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator.has" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator.has) uses a correlated subquery, its performance is not nearly as good when compared against large target tables as that of using a join.

[has()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator.has" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator.has) is only valid for scalar references, i.e. a [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) that has uselist=False. For collection references, use [any()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator.any" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator.any).

**ilike**(*other*, *escape=None*)

*inherited from the* [ilike()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.ilike" \o "sqlalchemy.sql.operators.ColumnOperators.ilike) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the ilike operator, e.g. case insensitive LIKE.

In a column context, produces an expression either of the form:

lower(a) LIKE lower(other)

Or on backends that support the ILIKE operator:

a ILIKE other

E.g.:

stmt = select([sometable]).\

where(sometable.c.column.ilike("*%f*oobar%"))

|  |  |
| --- | --- |
| **Parameters:** | * ****other**** – expression to be compared * ****escape**** –   optional escape character, renders the ESCAPE keyword, e.g.:  somecolumn.ilike("foo/%bar", escape="/") |

**See also**

[ColumnOperators.like()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.like" \o "sqlalchemy.sql.operators.ColumnOperators.like)

**in\_**(*other*)

Produce an IN clause - this is not implemented for [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)-based attributes at this time.

**is\_**(*other*)

*inherited from the* [is\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.is_" \o "sqlalchemy.sql.operators.ColumnOperators.is_) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the IS operator.

Normally, IS is generated automatically when comparing to a value of None, which resolves to NULL. However, explicit usage of IS may be desirable if comparing to boolean values on certain platforms.

*New in version 0.7.9.*

**See also**

[ColumnOperators.isnot()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.isnot" \o "sqlalchemy.sql.operators.ColumnOperators.isnot)

**is\_distinct\_from**(*other*)

*inherited from the* [is\_distinct\_from()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.is_distinct_from" \o "sqlalchemy.sql.operators.ColumnOperators.is_distinct_from) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the IS DISTINCT FROM operator.

Renders "a IS DISTINCT FROM b" on most platforms; on some such as SQLite may render "a IS NOT b".

*New in version 1.1.*

**isnot**(*other*)

*inherited from the* [isnot()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.isnot" \o "sqlalchemy.sql.operators.ColumnOperators.isnot) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the IS NOT operator.

Normally, IS NOT is generated automatically when comparing to a value of None, which resolves to NULL. However, explicit usage of IS NOT may be desirable if comparing to boolean values on certain platforms.

*New in version 0.7.9.*

**See also**

[ColumnOperators.is\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.is_" \o "sqlalchemy.sql.operators.ColumnOperators.is_)

**isnot\_distinct\_from**(*other*)

*inherited from the* [isnot\_distinct\_from()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.isnot_distinct_from" \o "sqlalchemy.sql.operators.ColumnOperators.isnot_distinct_from) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the IS NOT DISTINCT FROM operator.

Renders "a IS NOT DISTINCT FROM b" on most platforms; on some such as SQLite may render "a IS b".

*New in version 1.1.*

**like**(*other*, *escape=None*)

*inherited from the* [like()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.like" \o "sqlalchemy.sql.operators.ColumnOperators.like) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the like operator.

In a column context, produces the expression:

a LIKE other

E.g.:

stmt = select([sometable]).\

where(sometable.c.column.like("*%f*oobar%"))

|  |  |
| --- | --- |
| **Parameters:** | * ****other**** – expression to be compared * ****escape**** –   optional escape character, renders the ESCAPE keyword, e.g.:  somecolumn.like("foo/%bar", escape="/") |

**See also**

[ColumnOperators.ilike()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.ilike" \o "sqlalchemy.sql.operators.ColumnOperators.ilike)

**mapper**

The target [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) referred to by this [RelationshipProperty.Comparator](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator).

This is the "target" or "remote" side of the [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship).

**match**(*other*, *\*\*kwargs*)

*inherited from the* [match()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.match" \o "sqlalchemy.sql.operators.ColumnOperators.match) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implements a database-specific 'match' operator.

[match()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.match" \o "sqlalchemy.sql.operators.ColumnOperators.match) attempts to resolve to a MATCH-like function or operator provided by the backend. Examples include:

* PostgreSQL - renders x @@ to\_tsquery(y)
* MySQL - renders MATCH (x) AGAINST (y IN BOOLEAN MODE)
* Oracle - renders CONTAINS(x, y)
* other backends may provide special implementations.
* Backends without any special implementation will emit the operator as "MATCH". This is compatible with SQlite, for example.

**notilike**(*other*, *escape=None*)

*inherited from the* [notilike()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.notilike" \o "sqlalchemy.sql.operators.ColumnOperators.notilike) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

implement the NOT ILIKE operator.

This is equivalent to using negation with [ColumnOperators.ilike()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.ilike" \o "sqlalchemy.sql.operators.ColumnOperators.ilike), i.e. ~x.ilike(y).

*New in version 0.8.*

**See also**

[ColumnOperators.ilike()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.ilike" \o "sqlalchemy.sql.operators.ColumnOperators.ilike)

**notin\_**(*other*)

*inherited from the* [notin\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.notin_" \o "sqlalchemy.sql.operators.ColumnOperators.notin_) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

implement the NOT IN operator.

This is equivalent to using negation with [ColumnOperators.in\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.in_" \o "sqlalchemy.sql.operators.ColumnOperators.in_), i.e. ~x.in\_(y).

*New in version 0.8.*

**See also**

[ColumnOperators.in\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.in_" \o "sqlalchemy.sql.operators.ColumnOperators.in_)

**notlike**(*other*, *escape=None*)

*inherited from the* [notlike()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.notlike" \o "sqlalchemy.sql.operators.ColumnOperators.notlike) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

implement the NOT LIKE operator.

This is equivalent to using negation with [ColumnOperators.like()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.like" \o "sqlalchemy.sql.operators.ColumnOperators.like), i.e. ~x.like(y).

*New in version 0.8.*

**See also**

[ColumnOperators.like()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.like" \o "sqlalchemy.sql.operators.ColumnOperators.like)

**nullsfirst**()

*inherited from the* [nullsfirst()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.nullsfirst" \o "sqlalchemy.sql.operators.ColumnOperators.nullsfirst) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Produce a [nullsfirst()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.nullsfirst" \o "sqlalchemy.sql.expression.nullsfirst) clause against the parent object.

**nullslast**()

*inherited from the* [nullslast()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.nullslast" \o "sqlalchemy.sql.operators.ColumnOperators.nullslast) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Produce a [nullslast()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.nullslast" \o "sqlalchemy.sql.expression.nullslast) clause against the parent object.

**of\_type**(*cls*)

Produce a construct that represents a particular 'subtype' of attribute for the parent class.

Currently this is usable in conjunction with [Query.join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join) and [Query.outerjoin()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.outerjoin" \o "sqlalchemy.orm.query.Query.outerjoin).

**op**(*opstring*, *precedence=0*, *is\_comparison=False*)

*inherited from the* [op()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.Operators.op" \o "sqlalchemy.sql.operators.Operators.op) *method of* [Operators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.Operators" \o "sqlalchemy.sql.operators.Operators)

produce a generic operator function.

e.g.:

somecolumn.op("\*")(5)

produces:

somecolumn \* 5

This function can also be used to make bitwise operators explicit. For example:

somecolumn.op('&')(0xff)

is a bitwise AND of the value in somecolumn.

|  |  |
| --- | --- |
| **Parameters:** | * ****operator**** – a string which will be output as the infix operator between this element and the expression passed to the generated function. * ****precedence –****precedence to apply to the operator, when parenthesizing expressions. A lower number will cause the expression to be parenthesized when applied against another operator with higher precedence. The default value of 0 is lower than all operators except for the comma (,) and ASoperators. A value of 100 will be higher or equal to all operators, and -100 will be lower than or equal to all operators.   优先级应用于运算符，括号中表达式。 较低的数字会导致表达式被对应于较高优先级的另一个运算符时被括号。 默认值为0，除逗号(，)和ASoperators之外的所有运算符都低。 值100将高于或等于所有运算符，-100将低于或等于所有运算符。  *New in version 0.8:*- added the 'precedence' argument.   * ****is\_comparison –****if True, the operator will be considered as a "comparison" operator, that is which evaluates to a boolean true/false value, like ==, >, etc. This flag should be set so that ORM relationships can establish that the operator is a comparison operator when used in a custom join condition.   如果为True，则运算符将被视为“比较”运算符，即运算符的值为布尔值true / false，如==，>等。此标志应设置为使ORM关系可以确定运算符 一个比较运算符，用于自定义连接条件。  *New in version 0.9.2:*- added the [Operators.op.is\_comparison](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.Operators.op.params.is_comparison" \o "sqlalchemy.sql.operators.Operators.op) flag. |

**See also**

[Redefining and Creating New Operators](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "types-operators)

[Using custom operators in join conditions](http://docs.sqlalchemy.org/en/rel_1_1/orm/join_conditions.html" \l "relationship-custom-operator)

**operate**(*op*, *\*other*, *\*\*kwargs*)

*inherited from the* [operate()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.Operators.operate" \o "sqlalchemy.sql.operators.Operators.operate) *method of* [Operators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.Operators" \o "sqlalchemy.sql.operators.Operators)

Operate on an argument.

This is the lowest level of operation, raises NotImplementedError by default.

Overriding this on a subclass can allow common behavior to be applied to all operations. For example, overriding [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators) to apply func.lower() to the left and right side:

**class** **MyComparator**(ColumnOperators):

**def** operate(self, op, other):

**return** op(func.lower(self), func.lower(other))

|  |  |
| --- | --- |
| **Parameters:** | * ****op**** – Operator callable. * ****\*other**** – the 'other' side of the operation. Will be a single scalar for most operations. * ****\*\*kwargs**** – modifiers. These may be passed by special operators such as ColumnOperators.contains(). |

**reverse\_operate**(*op*, *other*, *\*\*kwargs*)

*inherited from the* [reverse\_operate()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.Operators.reverse_operate" \o "sqlalchemy.sql.operators.Operators.reverse_operate) *method of* [Operators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.Operators" \o "sqlalchemy.sql.operators.Operators)

Reverse operate on an argument.

Usage is the same as operate().

**startswith**(*other*, *\*\*kwargs*)

*inherited from the* [startswith()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.startswith" \o "sqlalchemy.sql.operators.ColumnOperators.startswith) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the startwith operator.

In a column context, produces the clause LIKE '<other>%'

**\_\_init\_\_**(*argument*, *secondary=None*, *primaryjoin=None*, *secondaryjoin=None*, *foreign\_keys=None*, *uselist=None*, *order\_by=False*, *backref=None*, *back\_populates=None*, *post\_update=False*, *cascade=False*, *extension=None*, *viewonly=False*, *lazy=True*, *collection\_class=None*, *passive\_deletes=False*, *passive\_updates=True*, *remote\_side=None*, *enable\_typechecks=True*, *join\_depth=None*, *comparator\_factory=None*, *single\_parent=False*, *innerjoin=False*, *distinct\_target\_key=None*, *doc=None*, *active\_history=False*, *cascade\_backrefs=True*, *load\_on\_pending=False*, *bake\_queries=True*, *\_local\_remote\_pairs=None*, *query\_class=None*, *info=None*)

Construct a new [RelationshipProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty" \o "sqlalchemy.orm.properties.RelationshipProperty) object.

This constructor is mirrored as a public API function; see [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) for a full usage and argument description.

**cascade**

Return the current cascade setting for this [RelationshipProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty" \o "sqlalchemy.orm.properties.RelationshipProperty).

**class\_attribute**

*inherited from the* [class\_attribute](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty.class_attribute" \o "sqlalchemy.orm.interfaces.MapperProperty.class_attribute) *attribute of* [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty)

Return the class-bound descriptor corresponding to this [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty).

This is basically a getattr() call:

**return** getattr(self.parent.class\_, self.key)

I.e. if this [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty) were named addresses, and the class to which it is mapped is User, this sequence is possible:

**>>> from** **sqlalchemy** **import** inspect

**>>>** mapper = inspect(User)

**>>>** addresses\_property = mapper.attrs.addresses

**>>>** addresses\_property.class\_attribute **is** User.addressesTrue

**>>>** User.addresses.property **is** addresses\_propertyTrue

**extension\_type***= symbol('NOT\_EXTENSION')*

**init**()

*inherited from the* [init()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty.init" \o "sqlalchemy.orm.interfaces.MapperProperty.init) *method of* [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty)

Called after all mappers are created to assemble relationships between mappers and perform other post-mapper-creation initialization steps.

**mapper**

Return the targeted [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) for this [RelationshipProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty" \o "sqlalchemy.orm.properties.RelationshipProperty).

This is a lazy-initializing static attribute.

**set\_parent**(*parent*, *init*)

*inherited from the* [set\_parent()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty.set_parent" \o "sqlalchemy.orm.interfaces.MapperProperty.set_parent) *method of* [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty)

Set the parent mapper that references this MapperProperty.

This method is overridden by some subclasses to perform extra setup when the mapper is first known.

**table**

Return the selectable linked to this [RelationshipProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty" \o "sqlalchemy.orm.properties.RelationshipProperty) object's target [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper).

*Deprecated since version 0.7:*Use .target

*class*sqlalchemy.orm.descriptor\_props.**SynonymProperty**(*name*, *map\_column=None*, *descriptor=None*, *comparator\_factory=None*, *doc=None*, *info=None*)

Bases: sqlalchemy.orm.descriptor\_props.DescriptorProperty

**\_\_init\_\_**(*name*, *map\_column=None*, *descriptor=None*, *comparator\_factory=None*, *doc=None*, *info=None*)

Construct a new [SynonymProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.descriptor_props.SynonymProperty" \o "sqlalchemy.orm.descriptor_props.SynonymProperty) object.

This constructor is mirrored as a public API function; see [synonym()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapped_attributes.html" \l "sqlalchemy.orm.synonym" \o "sqlalchemy.orm.synonym) for a full usage and argument description.

**cascade\_iterator**(*type\_*, *state*, *visited\_instances=None*, *halt\_on=None*)

*inherited from the* [cascade\_iterator()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty.cascade_iterator" \o "sqlalchemy.orm.interfaces.MapperProperty.cascade_iterator) *method of* [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty)

Iterate through instances related to the given instance for a particular 'cascade', starting with this MapperProperty.

通过与给定实例相关的实例，针对特定的“级联”迭代，从此MapperProperty开始。

Return an iterator3-tuples (instance, mapper, state).

返回一个迭代器3元组(实例，映射器，状态)。

Note that the 'cascade' collection on this MapperProperty is checked first for the given type before cascade\_iterator is called.

请注意，在调用cascade\_iterator之前，首先检查此MapperProperty上的“cascade”集合给定类型。

This method typically only applies to RelationshipProperty.

此方法通常仅适用于RelationshipProperty。

**class\_attribute**

*inherited from the* [class\_attribute](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty.class_attribute" \o "sqlalchemy.orm.interfaces.MapperProperty.class_attribute) *attribute of* [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty)

Return the class-bound descriptor corresponding to this [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty).

This is basically a getattr() call:

返回与此MapperProperty对应的类绑定描述符。

这基本上是一个getattr()调用：

**return** getattr(self.parent.class\_, self.key)

I.e. if this [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty) were named addresses, and the class to which it is mapped is User, this sequence is possible:

**>>> from** **sqlalchemy** **import** inspect

**>>>** mapper = inspect(User)

**>>>** addresses\_property = mapper.attrs.addresses

**>>>** addresses\_property.class\_attribute **is** User.addressesTrue

**>>>** User.addresses.property **is** addresses\_propertyTrue

**create\_row\_processor**(*context*, *path*, *mapper*, *result*, *adapter*, *populators*)

*inherited from the* [create\_row\_processor()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty.create_row_processor" \o "sqlalchemy.orm.interfaces.MapperProperty.create_row_processor) *method of* [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty)

Produce row processing functions and append to the given set of populators lists.

**do\_init**()

*inherited from the* [do\_init()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty.do_init" \o "sqlalchemy.orm.interfaces.MapperProperty.do_init) *method of* [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty)

Perform subclass-specific initialization post-mapper-creation steps.

This is a template method called by the MapperProperty object's init() method.

**extension\_type***= symbol('NOT\_EXTENSION')*

**init**()

*inherited from the* [init()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty.init" \o "sqlalchemy.orm.interfaces.MapperProperty.init) *method of* [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty)

Called after all mappers are created to assemble relationships between mappers and perform other post-mapper-creation initialization steps.

**merge**(*session*, *source\_state*, *source\_dict*, *dest\_state*, *dest\_dict*, *load*, *\_recursive*, *\_resolve\_conflict\_map*)

*inherited from the* [merge()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty.merge" \o "sqlalchemy.orm.interfaces.MapperProperty.merge) *method of* [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty)

Merge the attribute represented by this MapperProperty from source to destination object.

**post\_instrument\_class**(*mapper*)

*inherited from the* [post\_instrument\_class()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty.post_instrument_class" \o "sqlalchemy.orm.interfaces.MapperProperty.post_instrument_class) *method of* [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty)

Perform instrumentation adjustments that need to occur after init() has completed.

The given Mapper is the Mapper invoking the operation, which may not be the same Mapper as self.parent in an inheritance scenario; however, Mapper will always at least be a sub-mapper of self.parent.

This method is typically used by StrategizedProperty, which delegates it to LoaderStrategy.init\_class\_attribute() to perform final setup on the class-bound InstrumentedAttribute.

**setup**(*context*, *entity*, *path*, *adapter*, *\*\*kwargs*)

*inherited from the* [setup()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty.setup" \o "sqlalchemy.orm.interfaces.MapperProperty.setup) *method of* [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty)

Called by Query for the purposes of constructing a SQL statement.

Each MapperProperty associated with the target mapper processes the statement referenced by the query context, adding columns and/or criterion as appropriate.

*class*sqlalchemy.orm.query.**QueryContext**(*query*)

*class*sqlalchemy.orm.attributes.**QueryableAttribute**(*class\_*, *key*, *impl=None*, *comparator=None*, *parententity=None*, *of\_type=None*)

Bases: sqlalchemy.orm.base.\_MappedAttribute, [sqlalchemy.orm.base.InspectionAttr](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.base.InspectionAttr" \o "sqlalchemy.orm.base.InspectionAttr), [sqlalchemy.orm.interfaces.PropComparator](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator" \o "sqlalchemy.orm.interfaces.PropComparator)

Base class for [descriptor](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-descriptor) objects that intercept attribute events on behalf of a [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty) object. The actual [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty) is accessible via the [QueryableAttribute.property](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.attributes.QueryableAttribute.property" \o "sqlalchemy.orm.attributes.QueryableAttribute.property) attribute.

**See also**

[InstrumentedAttribute](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.attributes.InstrumentedAttribute" \o "sqlalchemy.orm.attributes.InstrumentedAttribute)

[MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty)

[Mapper.all\_orm\_descriptors](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper.all_orm_descriptors" \o "sqlalchemy.orm.mapper.Mapper.all_orm_descriptors)

[Mapper.attrs](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper.attrs" \o "sqlalchemy.orm.mapper.Mapper.attrs)

**\_\_eq\_\_**(*other*)

*inherited from the* [\_\_eq\_\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.__eq__" \o "sqlalchemy.sql.operators.ColumnOperators.__eq__) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the == operator.

In a column context, produces the clause a = b. If the target is None, produces a IS NULL.

**\_\_le\_\_**(*other*)

*inherited from the* [\_\_le\_\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.__le__" \o "sqlalchemy.sql.operators.ColumnOperators.__le__) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the <= operator.

In a column context, produces the clause a <= b.

**\_\_lt\_\_**(*other*)

*inherited from the* [\_\_lt\_\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.__lt__" \o "sqlalchemy.sql.operators.ColumnOperators.__lt__) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the < operator.

In a column context, produces the clause a < b.

**\_\_ne\_\_**(*other*)

*inherited from the* [\_\_ne\_\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.__ne__" \o "sqlalchemy.sql.operators.ColumnOperators.__ne__) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the != operator.

In a column context, produces the clause a != b. If the target is None, produces a IS NOT NULL.

**adapter**

*inherited from the* [adapter](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator.adapter" \o "sqlalchemy.orm.interfaces.PropComparator.adapter) *attribute of* [PropComparator](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator" \o "sqlalchemy.orm.interfaces.PropComparator)

Produce a callable that adapts column expressions to suit an aliased version of this comparator.

**all\_**()

*inherited from the* [all\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.all_" \o "sqlalchemy.sql.operators.ColumnOperators.all_) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Produce a [all\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.all_" \o "sqlalchemy.sql.expression.all_) clause against the parent object.

*New in version 1.1.*

**any**(*criterion=None*, *\*\*kwargs*)

*inherited from the* [any()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator.any" \o "sqlalchemy.orm.interfaces.PropComparator.any) *method of* [PropComparator](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator" \o "sqlalchemy.orm.interfaces.PropComparator)

Return true if this collection contains any member that meets the given criterion.

The usual implementation of any() is [RelationshipProperty.Comparator.any()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator.any" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator.any).

|  |  |
| --- | --- |
| **Parameters:** | * ****criterion**** – an optional ClauseElement formulated against the member class' table or attributes. * ****\*\*kwargs**** – key/value pairs corresponding to member class attribute names which will be compared via equality to the corresponding values. |

**any\_**()

*inherited from the* [any\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.any_" \o "sqlalchemy.sql.operators.ColumnOperators.any_) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Produce a [any\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.any_" \o "sqlalchemy.sql.expression.any_) clause against the parent object.

*New in version 1.1.*

**asc**()

*inherited from the* [asc()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.asc" \o "sqlalchemy.sql.operators.ColumnOperators.asc) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Produce a [asc()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.asc" \o "sqlalchemy.sql.expression.asc) clause against the parent object.

**between**(*cleft*, *cright*, *symmetric=False*)

*inherited from the* [between()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.between" \o "sqlalchemy.sql.operators.ColumnOperators.between) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Produce a [between()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.between" \o "sqlalchemy.sql.expression.between) clause against the parent object, given the lower and upper range.

**collate**(*collation*)

*inherited from the* [collate()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.collate" \o "sqlalchemy.sql.operators.ColumnOperators.collate) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Produce a [collate()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.collate" \o "sqlalchemy.sql.expression.collate) clause against the parent object, given the collation string.

**concat**(*other*)

*inherited from the* [concat()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.concat" \o "sqlalchemy.sql.operators.ColumnOperators.concat) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the 'concat' operator.

In a column context, produces the clause a || b, or uses the concat() operator on MySQL.

**contains**(*other*, *\*\*kwargs*)

*inherited from the* [contains()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.contains" \o "sqlalchemy.sql.operators.ColumnOperators.contains) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the 'contains' operator.

In a column context, produces the clause LIKE '%<other>%'

**desc**()

*inherited from the* [desc()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.desc" \o "sqlalchemy.sql.operators.ColumnOperators.desc) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Produce a [desc()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.desc" \o "sqlalchemy.sql.expression.desc) clause against the parent object.

**distinct**()

*inherited from the* [distinct()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.distinct" \o "sqlalchemy.sql.operators.ColumnOperators.distinct) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Produce a [distinct()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.distinct" \o "sqlalchemy.sql.expression.distinct) clause against the parent object.

**endswith**(*other*, *\*\*kwargs*)

*inherited from the* [endswith()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.endswith" \o "sqlalchemy.sql.operators.ColumnOperators.endswith) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the 'endswith' operator.

In a column context, produces the clause LIKE '%<other>'

**extension\_type***= symbol('NOT\_EXTENSION')*

**has**(*criterion=None*, *\*\*kwargs*)

*inherited from the* [has()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator.has" \o "sqlalchemy.orm.interfaces.PropComparator.has) *method of* [PropComparator](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator" \o "sqlalchemy.orm.interfaces.PropComparator)

Return true if this element references a member which meets the given criterion.

The usual implementation of has() is [RelationshipProperty.Comparator.has()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator.has" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator.has).

|  |  |
| --- | --- |
| **Parameters:** | * ****criterion**** – an optional ClauseElement formulated against the member class' table or attributes. * ****\*\*kwargs**** – key/value pairs corresponding to member class attribute names which will be compared via equality to the corresponding values. |

**ilike**(*other*, *escape=None*)

*inherited from the* [ilike()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.ilike" \o "sqlalchemy.sql.operators.ColumnOperators.ilike) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the ilike operator, e.g. case insensitive LIKE.

In a column context, produces an expression either of the form:

lower(a) LIKE lower(other)

Or on backends that support the ILIKE operator:

a ILIKE other

E.g.:

stmt = select([sometable]).\

where(sometable.c.column.ilike("*%f*oobar%"))

|  |  |
| --- | --- |
| **Parameters:** | * ****other**** – expression to be compared * ****escape**** –   optional escape character, renders the ESCAPE keyword, e.g.:  somecolumn.ilike("foo/%bar", escape="/") |

**See also**

[ColumnOperators.like()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.like" \o "sqlalchemy.sql.operators.ColumnOperators.like)

**in\_**(*other*)

*inherited from the* [in\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.in_" \o "sqlalchemy.sql.operators.ColumnOperators.in_) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the in operator.

In a column context, produces the clause a IN other. "other" may be a tuple/list of column expressions, or a [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) construct.

**info**

Return the 'info' dictionary for the underlying SQL element.

The behavior here is as follows:

* If the attribute is a column-mapped property, i.e. [ColumnProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.ColumnProperty" \o "sqlalchemy.orm.properties.ColumnProperty), which is mapped directly to a schema-level [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) object, this attribute will return the [SchemaItem.info](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem.info" \o "sqlalchemy.schema.SchemaItem.info) dictionary associated with the core-level [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) object.
* If the attribute is a [ColumnProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.ColumnProperty" \o "sqlalchemy.orm.properties.ColumnProperty) but is mapped to any other kind of SQL expression other than a [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column), the attribute will refer to the [MapperProperty.info](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "MapperProperty.info" \o "MapperProperty.info) dictionary associated directly with the [ColumnProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.ColumnProperty" \o "sqlalchemy.orm.properties.ColumnProperty), assuming the SQL expression itself does not have its own .info attribute (which should be the case, unless a user-defined SQL construct has defined one).
* If the attribute refers to any other kind of [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty), including [RelationshipProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty" \o "sqlalchemy.orm.properties.RelationshipProperty), the attribute will refer to the [MapperProperty.info](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "MapperProperty.info" \o "MapperProperty.info) dictionary associated with that [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty).
* To access the [MapperProperty.info](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "MapperProperty.info" \o "MapperProperty.info) dictionary of the [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty) unconditionally, including for a [ColumnProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.ColumnProperty" \o "sqlalchemy.orm.properties.ColumnProperty) that's associated directly with a [schema.Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column), the attribute can be referred to using [QueryableAttribute.property](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.attributes.QueryableAttribute.property" \o "sqlalchemy.orm.attributes.QueryableAttribute.property) attribute, asMyClass.someattribute.property.info.

*New in version 0.8.0.*

**See also**

[SchemaItem.info](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem.info" \o "sqlalchemy.schema.SchemaItem.info)

[MapperProperty.info](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "MapperProperty.info" \o "MapperProperty.info)

**is\_**(*other*)

*inherited from the* [is\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.is_" \o "sqlalchemy.sql.operators.ColumnOperators.is_) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the IS operator.

Normally, IS is generated automatically when comparing to a value of None, which resolves to NULL. However, explicit usage of IS may be desirable if comparing to boolean values on certain platforms.

*New in version 0.7.9.*

**See also**

[ColumnOperators.isnot()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.isnot" \o "sqlalchemy.sql.operators.ColumnOperators.isnot)

**is\_distinct\_from**(*other*)

*inherited from the* [is\_distinct\_from()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.is_distinct_from" \o "sqlalchemy.sql.operators.ColumnOperators.is_distinct_from) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the IS DISTINCT FROM operator.

Renders "a IS DISTINCT FROM b" on most platforms; on some such as SQLite may render "a IS NOT b".

*New in version 1.1.*

**isnot**(*other*)

*inherited from the* [isnot()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.isnot" \o "sqlalchemy.sql.operators.ColumnOperators.isnot) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the IS NOT operator.

Normally, IS NOT is generated automatically when comparing to a value of None, which resolves to NULL. However, explicit usage of IS NOT may be desirable if comparing to boolean values on certain platforms.

*New in version 0.7.9.*

**See also**

[ColumnOperators.is\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.is_" \o "sqlalchemy.sql.operators.ColumnOperators.is_)

**isnot\_distinct\_from**(*other*)

*inherited from the* [isnot\_distinct\_from()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.isnot_distinct_from" \o "sqlalchemy.sql.operators.ColumnOperators.isnot_distinct_from) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the IS NOT DISTINCT FROM operator.

Renders "a IS NOT DISTINCT FROM b" on most platforms; on some such as SQLite may render "a IS b".

*New in version 1.1.*

**like**(*other*, *escape=None*)

*inherited from the* [like()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.like" \o "sqlalchemy.sql.operators.ColumnOperators.like) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the like operator.

In a column context, produces the expression:

a LIKE other

E.g.:

stmt = select([sometable]).\

where(sometable.c.column.like("*%f*oobar%"))

|  |  |
| --- | --- |
| **Parameters:** | * ****other**** – expression to be compared * ****escape**** –   optional escape character, renders the ESCAPE keyword, e.g.:  somecolumn.like("foo/%bar", escape="/") |

**See also**

[ColumnOperators.ilike()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.ilike" \o "sqlalchemy.sql.operators.ColumnOperators.ilike)

**match**(*other*, *\*\*kwargs*)

*inherited from the* [match()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.match" \o "sqlalchemy.sql.operators.ColumnOperators.match) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implements a database-specific 'match' operator.

[match()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.match" \o "sqlalchemy.sql.operators.ColumnOperators.match) attempts to resolve to a MATCH-like function or operator provided by the backend. Examples include:

* PostgreSQL - renders x @@ to\_tsquery(y)
* MySQL - renders MATCH (x) AGAINST (y IN BOOLEAN MODE)
* Oracle - renders CONTAINS(x, y)
* other backends may provide special implementations.
* Backends without any special implementation will emit the operator as "MATCH". This is compatible with SQlite, for example.

**notilike**(*other*, *escape=None*)

*inherited from the* [notilike()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.notilike" \o "sqlalchemy.sql.operators.ColumnOperators.notilike) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

implement the NOT ILIKE operator.

This is equivalent to using negation with [ColumnOperators.ilike()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.ilike" \o "sqlalchemy.sql.operators.ColumnOperators.ilike), i.e. ~x.ilike(y).

*New in version 0.8.*

**See also**

[ColumnOperators.ilike()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.ilike" \o "sqlalchemy.sql.operators.ColumnOperators.ilike)

**notin\_**(*other*)

*inherited from the* [notin\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.notin_" \o "sqlalchemy.sql.operators.ColumnOperators.notin_) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

implement the NOT IN operator.

This is equivalent to using negation with [ColumnOperators.in\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.in_" \o "sqlalchemy.sql.operators.ColumnOperators.in_), i.e. ~x.in\_(y).

*New in version 0.8.*

**See also**

[ColumnOperators.in\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.in_" \o "sqlalchemy.sql.operators.ColumnOperators.in_)

**notlike**(*other*, *escape=None*)

*inherited from the* [notlike()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.notlike" \o "sqlalchemy.sql.operators.ColumnOperators.notlike) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

implement the NOT LIKE operator.

This is equivalent to using negation with [ColumnOperators.like()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.like" \o "sqlalchemy.sql.operators.ColumnOperators.like), i.e. ~x.like(y).

*New in version 0.8.*

**See also**

[ColumnOperators.like()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.like" \o "sqlalchemy.sql.operators.ColumnOperators.like)

**nullsfirst**()

*inherited from the* [nullsfirst()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.nullsfirst" \o "sqlalchemy.sql.operators.ColumnOperators.nullsfirst) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Produce a [nullsfirst()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.nullsfirst" \o "sqlalchemy.sql.expression.nullsfirst) clause against the parent object.

**nullslast**()

*inherited from the* [nullslast()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.nullslast" \o "sqlalchemy.sql.operators.ColumnOperators.nullslast) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Produce a [nullslast()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.nullslast" \o "sqlalchemy.sql.expression.nullslast) clause against the parent object.

**op**(*opstring*, *precedence=0*, *is\_comparison=False*)

*inherited from the* [op()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.Operators.op" \o "sqlalchemy.sql.operators.Operators.op) *method of* [Operators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.Operators" \o "sqlalchemy.sql.operators.Operators)

produce a generic operator function.

e.g.:

somecolumn.op("\*")(5)

produces:

somecolumn \* 5

This function can also be used to make bitwise operators explicit. For example:

somecolumn.op('&')(0xff)

is a bitwise AND of the value in somecolumn.

|  |  |
| --- | --- |
| **Parameters:** | * ****operator**** – a string which will be output as the infix operator between this element and the expression passed to the generated function.该字符串将作为该元素和传递给生成函数的表达式之间的中缀运算符输出。 * ****precedence –****precedence to apply to the operator, when parenthesizing expressions. A lower number will cause the expression to be parenthesized when applied against another operator with higher precedence. The default value of 0 is lower than all operators except for the comma (,) and AS operators. A value of 100 will be higher or equal to all operators, and -100 will be lower than or equal to all operators.优先级应用于运算符，括号中表达式。 较低的数字会导致表达式被对应于较高优先级的另一个运算符时被括号。 默认值为0，除逗号(，)和AS运算符之外的所有运算符都低。 值100将高于或等于所有运算符，-100将低于或等于所有运算符。   *New in version 0.8:*- added the 'precedence' argument.   * ****is\_comparison –****if True, the operator will be considered as a "comparison" operator, that is which evaluates to a boolean true/false value, like ==, >, etc. This flag should be set so that ORM relationships can establish that the operator is a comparison operator when used in a custom join condition.如果为True，则运算符将被视为“比较”运算符，即运算符的值为布尔值true / false，如==，>等。此标志应设置为使ORM关系可以确定运算符 一个比较运算符，用于自定义连接条件。   *New in version 0.9.2:*- added the [Operators.op.is\_comparison](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.Operators.op.params.is_comparison" \o "sqlalchemy.sql.operators.Operators.op) flag. |

**See also**

[Redefining and Creating New Operators](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "types-operators)

[Using custom operators in join conditions](http://docs.sqlalchemy.org/en/rel_1_1/orm/join_conditions.html" \l "relationship-custom-operator)

**parent**

Return an inspection instance representing the parent.

This will be either an instance of [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) or [AliasedInsp](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.util.AliasedInsp" \o "sqlalchemy.orm.util.AliasedInsp), depending upon the nature of the parent entity which this attribute is associated with.

**property**

Return the [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty) associated with this [QueryableAttribute](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.attributes.QueryableAttribute" \o "sqlalchemy.orm.attributes.QueryableAttribute).

Return values here will commonly be instances of [ColumnProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.ColumnProperty" \o "sqlalchemy.orm.properties.ColumnProperty) or [RelationshipProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty" \o "sqlalchemy.orm.properties.RelationshipProperty).

**startswith**(*other*, *\*\*kwargs*)

*inherited from the* [startswith()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.startswith" \o "sqlalchemy.sql.operators.ColumnOperators.startswith) *method of* [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators)

Implement the startwith operator.

In a column context, produces the clause LIKE '<other>%'

*class*sqlalchemy.orm.session.**UOWTransaction**(*session*)

**filter\_states\_for\_dep**(*dep*, *states*)

Filter the given list of InstanceStates to those relevant to the given DependencyProcessor.

**finalize\_flush\_changes**()

mark processed objects as clean / deleted after a successful flush().

this method is called within the flush() method after the execute() method has succeeded and the transaction has been committed.

**get\_attribute\_history**(*state*, *key*, *passive=symbol('PASSIVE\_NO\_INITIALIZE')*)

facade to attributes.get\_state\_history(), including caching of results.

**is\_deleted**(*state*)

return true if the given state is marked as deleted within this uowtransaction.

**remove\_state\_actions**(*state*)

remove pending actions for a state from the uowtransaction.

**was\_already\_deleted**(*state*)

return true if the given state is expired and was deleted previously.

## 6.3 ORM Exceptions

SQLAlchemy ORM exceptions.

sqlalchemy.orm.exc.**ConcurrentModificationError**

alias of [StaleDataError](http://docs.sqlalchemy.org/en/rel_1_1/orm/exceptions.html" \l "sqlalchemy.orm.exc.StaleDataError" \o "sqlalchemy.orm.exc.StaleDataError)

*exception*sqlalchemy.orm.exc.**DetachedInstanceError**

An attempt to access unloaded attributes on a mapped instance that is detached.

*exception*sqlalchemy.orm.exc.**FlushError**

A invalid condition was detected during flush().

*exception*sqlalchemy.orm.exc.**MultipleResultsFound**

A single database result was required but more than one were found.

sqlalchemy.orm.exc.**NO\_STATE***= (<type 'exceptions.AttributeError'>, <type 'exceptions.KeyError'>)*

Exception types that may be raised by instrumentation implementations.

*exception*sqlalchemy.orm.exc.**NoResultFound**

A database result was required but none was found.

*exception*sqlalchemy.orm.exc.**ObjectDeletedError**(*state*, *msg=None*)

A refresh operation failed to retrieve the database row corresponding to an object's known primary key identity.

刷新操作无法检索对应于对象的已知主键标识的数据库行。

A refresh operation proceeds when an expired attribute is accessed on an object, or when [Query.get()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.get" \o "sqlalchemy.orm.query.Query.get) is used to retrieve an object which is, upon retrieval, detected as expired. A SELECT is emitted for the target row based on primary key; if no row is returned, this exception is raised.

当在对象上访问过期的属性时，或者当使用Query.get()来检索检索到已过期的对象时，刷新操作继续。 基于主键对目标行发出SELECT; 如果没有返回行，则会引发此异常。

The true meaning of this exception is simply that no row exists for the primary key identifier associated with a persistent object. The row may have been deleted, or in some cases the primary key updated to a new value, outside of the ORM's management of the target object.

这个异常的真正含义是简单地说，与持久对象关联的主键标识符不存在任何行。 该行可能已被删除，或者在某些情况下将主键更新为新值，而不在目标对象的ORM管理之外。

*exception*sqlalchemy.orm.exc.**ObjectDereferencedError**

An operation cannot complete due to an object being garbage collected.

*exception*sqlalchemy.orm.exc.**StaleDataError**

An operation encountered database state that is unaccounted for.

Conditions which cause this to happen include:

A flush may have attempted to update or delete rows and an unexpected number of rows were matched during the UPDATE or DELETE statement. Note that when version\_id\_col is used, rows in UPDATE or DELETE statements are also matched against the current known version identifier.

刷新可能已尝试更新或删除行，并且在UPDATE或DELETE语句期间匹配了意外数量的行。 请注意，当使用version\_id\_col时，UPDATE或DELETE语句中的行也与当前已知的版本标识符进行匹配。

A mapped object with version\_id\_col was refreshed, and the version number coming back from the database does not match that of the object itself.

已刷新具有version\_id\_col的映射对象，并从数据库返回的版本号与对象本身不匹配。

A object is detached from its parent object, however the object was previously attached to a different parent identity which was garbage collected, and a decision cannot be made if the new parent was really the most recent "parent".

一个对象与其父对象分离，但是该对象之前已附加到垃圾回收的不同父标识，如果新父级真的是最新的“父”，则无法做出决定。

*New in version 0.7.4.*

*exception*sqlalchemy.orm.exc.**UnmappedClassError**(*cls*, *msg=None*)

An mapping operation was requested for an unknown class.

*exception*sqlalchemy.orm.exc.**UnmappedColumnError**

Mapping operation was requested on an unknown column.

*exception*sqlalchemy.orm.exc.**UnmappedError**

Base for exceptions that involve expected mappings not present.

*exception*sqlalchemy.orm.exc.**UnmappedInstanceError**(*obj*, *msg=None*)

An mapping operation was requested for an unknown instance.

## ~~6.4 Deprecated ORM Event Interfaces~~

~~This section describes the class-based ORM event interface which first existed in SQLAlchemy 0.1, which progressed with more kinds of events up until SQLAlchemy 0.5. The non-ORM analogue is described at [Deprecated Event Interfaces](http://docs.sqlalchemy.org/en/rel_1_1/core/interfaces.html).~~

*~~Deprecated since version 0.7:~~*~~As of SQLAlchemy 0.7, the new event system described in [Events](http://docs.sqlalchemy.org/en/rel_1_1/core/event.html) replaces the extension/proxy/listener system, providing a consistent interface to all events without the need for subclassing.~~

~~6.4.1 Mapper Events~~

*~~class~~*~~sqlalchemy.orm.interfaces.~~**~~MapperExtension~~**

~~Base implementation for [Mapper](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper.Mapper" \o "sqlalchemy.orm.mapper.Mapper) event hooks.~~

**~~Note~~**

~~[MapperExtension](http://docs.sqlalchemy.org/en/rel_1_1/orm/deprecated.html" \l "sqlalchemy.orm.interfaces.MapperExtension" \o "sqlalchemy.orm.interfaces.MapperExtension) is deprecated. Please refer to [event.listen()](http://docs.sqlalchemy.org/en/rel_1_1/core/event.html" \l "sqlalchemy.event.listen" \o "sqlalchemy.event.listen) as well as [MapperEvents](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents" \o "sqlalchemy.orm.events.MapperEvents).~~

~~New extension classes subclass [MapperExtension](http://docs.sqlalchemy.org/en/rel_1_1/orm/deprecated.html" \l "sqlalchemy.orm.interfaces.MapperExtension" \o "sqlalchemy.orm.interfaces.MapperExtension) and are specified using the extension mapper() argument, which is a single [MapperExtension](http://docs.sqlalchemy.org/en/rel_1_1/orm/deprecated.html" \l "sqlalchemy.orm.interfaces.MapperExtension" \o "sqlalchemy.orm.interfaces.MapperExtension) or a list of such:~~

**~~from~~****~~sqlalchemy.orm.interfaces~~****~~import~~** ~~MapperExtension~~

**~~class~~****~~MyExtension~~**~~(MapperExtension):~~

**~~def~~** ~~before\_insert(self, mapper, connection, instance):~~

~~print "instance~~ *~~%s~~* ~~before insert !" % instance~~

~~m = mapper(User, users\_table, extension=MyExtension())~~

~~A single mapper can maintain a chain of MapperExtension objects. When a particular mapping event occurs, the corresponding method on each MapperExtension is invoked serially, and each method has the ability to halt the chain from proceeding further:~~

~~m = mapper(User, users\_table, extension=[ext1, ext2, ext3])~~

~~Each MapperExtension method returns the symbol EXT\_CONTINUE by default. This symbol generally means "move to the next MapperExtension for processing". For methods that return objects like translated rows or new object instances, EXT\_CONTINUE means the result of the method should be ignored. In some cases it's required for a default mapper activity to be performed, such as adding a new instance to a result list.~~

~~The symbol EXT\_STOP has significance within a chain of MapperExtension objects that the chain will be stopped when this symbol is returned. Like EXT\_CONTINUE, it also has additional significance in some cases that a default mapper activity will not be performed.~~

**~~after\_delete~~**~~(~~*~~mapper~~*~~,~~*~~connection~~*~~,~~*~~instance~~*~~)~~

~~Receive an object instance after that instance is deleted.~~

~~The return value is only significant within the MapperExtension chain; the parent mapper's behavior isn't modified by this method.~~

**~~after\_insert~~**~~(~~*~~mapper~~*~~,~~*~~connection~~*~~,~~*~~instance~~*~~)~~

~~Receive an object instance after that instance is inserted.~~

~~The return value is only significant within the MapperExtension chain; the parent mapper's behavior isn't modified by this method.~~

**~~after\_update~~**~~(~~*~~mapper~~*~~,~~*~~connection~~*~~,~~*~~instance~~*~~)~~

~~Receive an object instance after that instance is updated.~~

~~The return value is only significant within the MapperExtension chain; the parent mapper's behavior isn't modified by this method.~~

**~~before\_delete~~**~~(~~*~~mapper~~*~~,~~*~~connection~~*~~,~~*~~instance~~*~~)~~

~~Receive an object instance before that instance is deleted.~~

~~Note that~~*~~no~~*~~changes to the overall flush plan can be made here; and manipulation of the Session will not have the desired effect. To manipulate the Session within an extension, use SessionExtension.~~

~~The return value is only significant within the MapperExtension chain; the parent mapper's behavior isn't modified by this method.~~

**~~before\_insert~~**~~(~~*~~mapper~~*~~,~~*~~connection~~*~~,~~*~~instance~~*~~)~~

~~Receive an object instance before that instance is inserted into its table.~~

~~This is a good place to set up primary key values and such that aren't handled otherwise.~~

~~Column-based attributes can be modified within this method which will result in the new value being inserted. However~~*~~no~~*~~changes to the overall flush plan can be made, and manipulation of the Session will not have the desired effect. To manipulate the Session within an extension, useSessionExtension.~~

~~The return value is only significant within the MapperExtension chain; the parent mapper's behavior isn't modified by this method.~~

**~~before\_update~~**~~(~~*~~mapper~~*~~,~~*~~connection~~*~~,~~*~~instance~~*~~)~~

~~Receive an object instance before that instance is updated.~~

~~Note that this method is called for all instances that are marked as "dirty", even those which have no net changes to their column-based attributes. An object is marked as dirty when any of its column-based attributes have a "set attribute" operation called or when any of its collections are modified. If, at update time, no column-based attributes have any net changes, no UPDATE statement will be issued. This means that an instance being sent to before\_update is~~*~~not~~*~~a guarantee that an UPDATE statement will be issued (although you can affect the outcome here).~~

~~To detect if the column-based attributes on the object have net changes, and will therefore generate an UPDATE statement, useobject\_session(instance).is\_modified(instance, include\_collections=False).~~

~~Column-based attributes can be modified within this method which will result in the new value being updated. However~~*~~no~~*~~changes to the overall flush plan can be made, and manipulation of the Session will not have the desired effect. To manipulate the Session within an extension, useSessionExtension.~~

~~The return value is only significant within the MapperExtension chain; the parent mapper's behavior isn't modified by this method.~~

**~~init\_failed~~**~~(~~*~~mapper~~*~~,~~*~~class\_~~*~~,~~*~~oldinit~~*~~,~~*~~instance~~*~~,~~*~~args~~*~~,~~*~~kwargs~~*~~)~~

~~Receive an instance when its constructor has been called, and raised an exception.~~

~~This method is only called during a userland construction of an object. It is not called when an object is loaded from the database.~~

~~The return value is only significant within the MapperExtension chain; the parent mapper's behavior isn't modified by this method.~~

**~~init\_instance~~**~~(~~*~~mapper~~*~~,~~*~~class\_~~*~~,~~*~~oldinit~~*~~,~~*~~instance~~*~~,~~*~~args~~*~~,~~*~~kwargs~~*~~)~~

~~Receive an instance when its constructor is called.~~

~~This method is only called during a userland construction of an object. It is not called when an object is loaded from the database.~~

~~The return value is only significant within the MapperExtension chain; the parent mapper's behavior isn't modified by this method.~~

**~~instrument\_class~~**~~(~~*~~mapper~~*~~,~~*~~class\_~~*~~)~~

~~Receive a class when the mapper is first constructed, and has applied instrumentation to the mapped class.~~

~~The return value is only significant within the MapperExtension chain; the parent mapper's behavior isn't modified by this method.~~

**~~reconstruct\_instance~~**~~(~~*~~mapper~~*~~,~~*~~instance~~*~~)~~

~~Receive an object instance after it has been created via \_\_new\_\_, and after initial attribute population has occurred.~~

~~This typically occurs when the instance is created based on incoming result rows, and is only called once for that instance's lifetime.~~

~~Note that during a result-row load, this method is called upon the first row received for this instance. Note that some attributes and collections may or may not be loaded or even initialized, depending on what's present in the result rows.~~

~~The return value is only significant within the MapperExtension chain; the parent mapper's behavior isn't modified by this method.~~

~~6.4.2 Session Events~~

*~~class~~*~~sqlalchemy.orm.interfaces.~~**~~SessionExtension~~**

~~Base implementation for [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) event hooks.~~

**~~Note~~**

~~[SessionExtension](http://docs.sqlalchemy.org/en/rel_1_1/orm/deprecated.html" \l "sqlalchemy.orm.interfaces.SessionExtension" \o "sqlalchemy.orm.interfaces.SessionExtension) is deprecated. Please refer to [event.listen()](http://docs.sqlalchemy.org/en/rel_1_1/core/event.html" \l "sqlalchemy.event.listen" \o "sqlalchemy.event.listen) as well as [SessionEvents](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.SessionEvents" \o "sqlalchemy.orm.events.SessionEvents).~~

~~Subclasses may be installed into a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) (or [sessionmaker](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.sessionmaker" \o "sqlalchemy.orm.session.sessionmaker)) using the extension keyword argument:~~

**~~from~~****~~sqlalchemy.orm.interfaces~~****~~import~~** ~~SessionExtension~~

**~~class~~****~~MySessionExtension~~**~~(SessionExtension):~~

**~~def~~** ~~before\_commit(self, session):~~

~~print "before commit!"~~

~~Session = sessionmaker(extension=MySessionExtension())~~

~~The same [SessionExtension](http://docs.sqlalchemy.org/en/rel_1_1/orm/deprecated.html" \l "sqlalchemy.orm.interfaces.SessionExtension" \o "sqlalchemy.orm.interfaces.SessionExtension) instance can be used with any number of sessions.~~

**~~after\_attach~~**~~(~~*~~session~~*~~,~~*~~instance~~*~~)~~

~~Execute after an instance is attached to a session.~~

~~This is called after an add, delete or merge.~~

**~~after\_begin~~**~~(~~*~~session~~*~~,~~*~~transaction~~*~~,~~*~~connection~~*~~)~~

~~Execute after a transaction is begun on a connection~~

~~transaction is the SessionTransaction. This method is called after an engine level transaction is begun on a connection.~~

**~~after\_bulk\_delete~~**~~(~~*~~session~~*~~,~~*~~query~~*~~,~~*~~query\_context~~*~~,~~*~~result~~*~~)~~

~~Execute after a bulk delete operation to the session.~~

~~This is called after a session.query(…).delete()~~

~~query is the query object that this delete operation was called on. query\_context was the query context object. result is the result object returned from the bulk operation.~~

**~~after\_bulk\_update~~**~~(~~*~~session~~*~~,~~*~~query~~*~~,~~*~~query\_context~~*~~,~~*~~result~~*~~)~~

~~Execute after a bulk update operation to the session.~~

~~This is called after a session.query(…).update()~~

~~query is the query object that this update operation was called on. query\_context was the query context object. result is the result object returned from the bulk operation.~~

**~~after\_commit~~**~~(~~*~~session~~*~~)~~

~~Execute after a commit has occurred.~~

~~Note that this may not be per-flush if a longer running transaction is ongoing.~~

**~~after\_flush~~**~~(~~*~~session~~*~~,~~*~~flush\_context~~*~~)~~

~~Execute after flush has completed, but before commit has been called.~~

~~Note that the session's state is still in pre-flush, i.e. 'new', 'dirty', and 'deleted' lists still show pre-flush state as well as the history settings on instance attributes.~~

**~~after\_flush\_postexec~~**~~(~~*~~session~~*~~,~~*~~flush\_context~~*~~)~~

~~Execute after flush has completed, and after the post-exec state occurs.~~

~~This will be when the 'new', 'dirty', and 'deleted' lists are in their final state. An actual commit() may or may not have occurred, depending on whether or not the flush started its own transaction or participated in a larger transaction.~~

**~~after\_rollback~~**~~(~~*~~session~~*~~)~~

~~Execute after a rollback has occurred.~~

~~Note that this may not be per-flush if a longer running transaction is ongoing.~~

**~~before\_commit~~**~~(~~*~~session~~*~~)~~

~~Execute right before commit is called.~~

~~Note that this may not be per-flush if a longer running transaction is ongoing.~~

**~~before\_flush~~**~~(~~*~~session~~*~~,~~*~~flush\_context~~*~~,~~*~~instances~~*~~)~~

~~Execute before flush process has started.~~

~~instances is an optional list of objects which were passed to the flush() method.~~

### ~~6.4.3 Attribute Events~~

*~~class~~*~~sqlalchemy.orm.interfaces.~~**~~AttributeExtension~~**

~~Base implementation for AttributeImpl event hooks, events that fire upon attribute mutations in user code.~~

**~~Note~~**

~~[AttributeExtension](http://docs.sqlalchemy.org/en/rel_1_1/orm/deprecated.html" \l "sqlalchemy.orm.interfaces.AttributeExtension" \o "sqlalchemy.orm.interfaces.AttributeExtension) is deprecated. Please refer to [event.listen()](http://docs.sqlalchemy.org/en/rel_1_1/core/event.html" \l "sqlalchemy.event.listen" \o "sqlalchemy.event.listen) as well as [AttributeEvents](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.AttributeEvents" \o "sqlalchemy.orm.events.AttributeEvents).~~

~~[AttributeExtension](http://docs.sqlalchemy.org/en/rel_1_1/orm/deprecated.html" \l "sqlalchemy.orm.interfaces.AttributeExtension" \o "sqlalchemy.orm.interfaces.AttributeExtension) is used to listen for set, remove, and append events on individual mapped attributes. It is established on an individual mapped attribute using the extension argument, available on [column\_property()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_columns.html" \l "sqlalchemy.orm.column_property" \o "sqlalchemy.orm.column_property), [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship), and others:~~

**~~from~~****~~sqlalchemy.orm.interfaces~~****~~import~~** ~~AttributeExtension~~**~~from~~****~~sqlalchemy.orm~~****~~import~~** ~~mapper, relationship, column\_property~~

**~~class~~****~~MyAttrExt~~**~~(AttributeExtension):~~

**~~def~~** ~~append(self, state, value, initiator):~~

~~print "append event !"~~

**~~return~~** ~~value~~

**~~def~~** ~~set(self, state, value, oldvalue, initiator):~~

~~print "set event !"~~

**~~return~~** ~~value~~

~~mapper(SomeClass, sometable, properties={~~

~~'foo':column\_property(sometable.c.foo, extension=MyAttrExt()),~~

~~'bar':relationship(Bar, extension=MyAttrExt())})~~

~~Note that the [AttributeExtension](http://docs.sqlalchemy.org/en/rel_1_1/orm/deprecated.html" \l "sqlalchemy.orm.interfaces.AttributeExtension" \o "sqlalchemy.orm.interfaces.AttributeExtension) methods [append()](http://docs.sqlalchemy.org/en/rel_1_1/orm/deprecated.html" \l "sqlalchemy.orm.interfaces.AttributeExtension.append" \o "sqlalchemy.orm.interfaces.AttributeExtension.append) and [set()](http://docs.sqlalchemy.org/en/rel_1_1/orm/deprecated.html" \l "sqlalchemy.orm.interfaces.AttributeExtension.set" \o "sqlalchemy.orm.interfaces.AttributeExtension.set) need to return the value parameter. The returned value is used as the effective value, and allows the extension to change what is ultimately persisted.~~

~~AttributeExtension is assembled within the descriptors associated with a mapped class.~~

**~~active\_history~~***~~= True~~*

~~indicates that the set() method would like to receive the 'old' value, even if it means firing lazy callables.~~

~~Note that active\_history can also be set directly via [column\_property()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_columns.html" \l "sqlalchemy.orm.column_property" \o "sqlalchemy.orm.column_property) and [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship).~~

**~~append~~**~~(~~*~~state~~*~~,~~*~~value~~*~~,~~*~~initiator~~*~~)~~

~~Receive a collection append event.~~

~~The returned value will be used as the actual value to be appended.~~

**~~remove~~**~~(~~*~~state~~*~~,~~*~~value~~*~~,~~*~~initiator~~*~~)~~

~~Receive a remove event.~~

~~No return value is defined.~~

**~~set~~**~~(~~*~~state~~*~~,~~*~~value~~*~~,~~*~~oldvalue~~*~~,~~*~~initiator~~*~~)~~

~~Receive a set event.~~

~~The returned value will be used as the actual value to be set.~~

# Chapter 7 ORM Extensions

SQLAlchemy has a variety of ORM extensions available, which add additional functionality to the core behavior.

SQLAlchemy提供了多种可用的ORM扩展，它们为核心行为添加了额外的功能。

The extensions build almost entirely on public core and ORM APIs and users should be encouraged to read their source code to further their understanding of their behavior. In particular the "Horizontal Sharding", "Hybrid Attributes", and "Mutation Tracking" extensions are very succinct.

扩展构建几乎完全基于公共核心和ORM API，应鼓励用户阅读其源代码，以进一步了解其行为。 特别是“水平分片”，“混合属性”和“变形跟踪”扩展非常简洁。

## 7.1 Association Proxy

associationproxy is used to create a read/write view of a target attribute across a relationship. It essentially conceals the usage of a "middle" attribute between two endpoints, and can be used to cherry-pick fields from a collection of related objects or to reduce the verbosity of using the association object pattern. Applied creatively, the association proxy allows the construction of sophisticated collections and dictionary views of virtually any geometry, persisted to the database using standard, transparently configured relational patterns.

associationproxy用于在关系中创建目标属性的读/写视图。 它基本上掩盖了两个端点之间的“中间”属性的使用，并且可以用于从相关对象的集合中挑选字段，或者减少使用关联对象模式的冗长度。 应用创造性地，关联代理允许构建几乎任何几何的复杂集合和字典视图，使用标准的，透明配置的关系模式持久到数据库。

### 7.1.1 Simplifying Scalar Collections

Consider a many-to-many mapping between two classes, User and Keyword. Each User can have any number of Keyword objects, and vice-versa (the many-to-many pattern is described at [Many To Many](http://docs.sqlalchemy.org/en/rel_1_1/orm/basic_relationships.html" \l "relationships-many-to-many)):

考虑两个类(User和Keyword)之间的多对多映射。 每个用户可以有任意数量的关键字对象，反之亦然(多对多模式在多对多描述中)：

**from** **sqlalchemy** **import** Column, Integer, String, ForeignKey, Table

**from** **sqlalchemy.orm** **import** relationship

**from** **sqlalchemy.ext.declarative** **import** declarative\_base

Base = declarative\_base()

**class** **User**(Base):

\_\_tablename\_\_ = 'user'

id = Column(Integer, primary\_key=**True**)

name = Column(String(64))

kw = relationship("Keyword", secondary=**lambda**: userkeywords\_table)

**def** \_\_init\_\_(self, name):

self.name = name

**class** **Keyword**(Base):

\_\_tablename\_\_ = 'keyword'

id = Column(Integer, primary\_key=**True**)

keyword = Column('keyword', String(64))

**def** \_\_init\_\_(self, keyword):

self.keyword = keyword

userkeywords\_table = Table('userkeywords', Base.metadata,

Column('user\_id', Integer, ForeignKey("user.id"),

primary\_key=**True**),

Column('keyword\_id', Integer, ForeignKey("keyword.id"),

primary\_key=**True**))

Reading and manipulating the collection of "keyword" strings associated with User requires traversal from each collection element to the .keyword attribute, which can be awkward:

读取和操作与User关联的“关键字”字符串的集合需要从每个集合元素遍历到.keyword属性，这可能是尴尬的：

**>>>** user = User('jek')

**>>>** user.kw.append(Keyword('cheese inspector'))

**>>>** print(user.kw)

[<\_\_main\_\_.Keyword object at 0x12bf830>]

**>>>** print(user.kw[0].keyword)cheese inspector

**>>>** print([keyword.keyword **for** keyword **in** user.kw])['cheese inspector']

The association\_proxy is applied to the User class to produce a "view" of the kw relationship, which only exposes the string value of .keyword associated with each Keyword object:

association\_proxy应用于User类以产生kw关系的“视图”，该关系仅显示与每个Keyword对象关联的.keyword的字符串值：

**from** **sqlalchemy.ext.associationproxy** **import** association\_proxy

**class** **User**(Base):

\_\_tablename\_\_ = 'user'

id = Column(Integer, primary\_key=**True**)

name = Column(String(64))

kw = relationship("Keyword", secondary=**lambda**: userkeywords\_table)

**def** \_\_init\_\_(self, name):

self.name = name

*# proxy the 'keyword' attribute from the 'kw' relationship*

keywords = association\_proxy('kw', 'keyword')

We can now reference the .keywords collection as a listing of strings, which is both readable and writable. New Keyword objects are created for us transparently:

现在我们可以将.keywords集合引用为可读写的字符串列表。 新的关键字对象是透明地为我们创建的：

**>>>** user = User('jek')

**>>>** user.keywords.append('cheese inspector')

**>>>** user.keywords['cheese inspector']

**>>>** user.keywords.append('snack ninja')

**>>>** user.kw[<\_\_main\_\_.Keyword object at 0x12cdd30>, <\_\_main\_\_.Keyword object at 0x12cde30>]

The [AssociationProxy](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.AssociationProxy" \o "sqlalchemy.ext.associationproxy.AssociationProxy) object produced by the [association\_proxy()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.association_proxy" \o "sqlalchemy.ext.associationproxy.association_proxy) function is an instance of a [Python descriptor](http://docs.python.org/howto/descriptor.html). It is always declared with the user-defined class being mapped, regardless of whether Declarative or classical mappings via the [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper) function are used.

由association\_proxy()函数生成的AssociationProxy对象是Python描述符的一个实例。 无论是否使用通过mapper()函数的声明或经典映射，都始终声明用户定义的类被映射。

The proxy functions by operating upon the underlying mapped attribute or collection in response to operations, and changes made via the proxy are immediately apparent in the mapped attribute, as well as vice versa. The underlying attribute remains fully accessible.

代理通过对基于映射的属性或集合进行操作以响应操作进行操作，并且通过代理进行的更改在映射属性中立即显现，反之亦然。 底层属性保持完全可访问。

When first accessed, the association proxy performs introspection operations on the target collection so that its behavior corresponds correctly. Details such as if the locally proxied attribute is a collection (as is typical) or a scalar reference, as well as if the collection acts like a set, list, or dictionary is taken into account, so that the proxy should act just like the underlying collection or attribute does.

当首次访问时，关联代理对目标集合执行内省操作，以使其行为正确对应。 详细信息，例如本地代理的属性是否是一个集合(如典型)或标量引用，以及如果集合的行为类似于集合，列表或字典，则代理应该像 底层集合或属性。

7.1.2 Creation of New Values

When a list append() event (or set add(), dictionary \_\_setitem\_\_(), or scalar assignment event) is intercepted by the association proxy, it instantiates a new instance of the "intermediary" object using its constructor, passing as a single argument the given value. In our example above, an operation like:

当关联代理拦截列表append()事件(或set add()，字典\_\_setitem \_\_()或标量赋值事件)时，它使用其构造函数实例化“中间”对象的新实例，将其作为单个 参数给定值。 在上面的例子中，

user.keywords.append('cheese inspector')

Is translated by the association proxy into the operation:

user.kw.append(Keyword('cheese inspector'))

The example works here because we have designed the constructor for Keyword to accept a single positional argument, keyword. For those cases where a single-argument constructor isn't feasible, the association proxy's creational behavior can be customized using the creator argument, which references a callable (i.e. Python function) that will produce a new object instance given the singular argument. Below we illustrate this using a lambda as is typical:

该示例在这里工作，因为我们设计了Keyword的构造函数来接受单个位置参数，keyword。 对于单参数构造函数不可行的情况，可以使用creator参数来定制关联代理的创建行为，该引用参数引用了一个可调用(即Python函数)，它将产生一个给定单个参数的新对象实例。 下面我们用一个lambda来说明这一点：

**class** **User**(Base):

*# ...*

*# use Keyword(keyword=kw) on append() events*

keywords = association\_proxy('kw', 'keyword',

creator=**lambda** kw: Keyword(keyword=kw))

The creator function accepts a single argument in the case of a list- or set- based collection, or a scalar attribute. In the case of a dictionary-based collection, it accepts two arguments, "key" and "value". An example of this is below in [Proxying to Dictionary Based Collections](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "proxying-dictionaries).

在基于列表或集合集合或标量属性的情况下，creator函数接受单个参数。 在基于字典的集合的情况下，它接受两个参数“key”和“value”。 下面是代理到基于字典的集合的一个例子。

**7.1.3 Simplifying Association Objects**

The "association object" pattern is an extended form of a many-to-many relationship, and is described at [Association Object](http://docs.sqlalchemy.org/en/rel_1_1/orm/basic_relationships.html" \l "association-pattern). Association proxies are useful for keeping "association objects" out of the way during regular use.

“关联对象”模式是多对多关系的扩展形式，并在关联对象中进行了描述。 关联代理对于在常规使用过程中保持“关联对象”非常有用。

Suppose our userkeywords table above had additional columns which we'd like to map explicitly, but in most cases we don't require direct access to these attributes. Below, we illustrate a new mapping which introduces the UserKeyword class, which is mapped to the userkeywords table illustrated earlier. This class adds an additional column special\_key, a value which we occasionally want to access, but not in the usual case. We create an association proxy on the User class called keywords, which will bridge the gap from the user\_keywords collection of User to the .keyword attribute present on each UserKeyword:

假设我们上面的用户关键字表具有我们要显式映射的其他列，但在大多数情况下，我们不需要直接访问这些属性。 下面，我们说明一个新的映射，它介绍了UserKeyword类，映射到前面说明的userkeywords表中。 这个类添加了一个额外的列special\_key，这是一个我们偶尔想要访问的值，但不是在通常的情况下。 我们在User类中创建一个关联代理叫做keywords，这将将用户的userkeywords集合与每个UserKeyword上存在的.keyword属性的差距缩小：

**from** **sqlalchemy** **import** Column, Integer, String, ForeignKey

**from** **sqlalchemy.orm** **import** relationship, backref

**from** **sqlalchemy.ext.associationproxy** **import** association\_proxy

**from** **sqlalchemy.ext.declarative** **import** declarative\_base

Base = declarative\_base()

**class** **User**(Base):

\_\_tablename\_\_ = 'user'

id = Column(Integer, primary\_key=**True**)

name = Column(String(64))

*# association proxy of "user\_keywords" collection*

*# to "keyword" attribute*

keywords = association\_proxy('user\_keywords', 'keyword')

**def** \_\_init\_\_(self, name):

self.name = name

**class** **UserKeyword**(Base):

\_\_tablename\_\_ = 'user\_keyword'

user\_id = Column(Integer, ForeignKey('user.id'), primary\_key=**True**)

keyword\_id = Column(Integer, ForeignKey('keyword.id'), primary\_key=**True**)

special\_key = Column(String(50))

*# bidirectional attribute/collection of "user"/"user\_keywords"*

user = relationship(User,

backref=backref("user\_keywords",

cascade="all, delete-orphan")

)

*# reference to the "Keyword" object*

keyword = relationship("Keyword")

**def** \_\_init\_\_(self, keyword=**None**, user=**None**, special\_key=**None**):

self.user = user

self.keyword = keyword

self.special\_key = special\_key

**class** **Keyword**(Base):

\_\_tablename\_\_ = 'keyword'

id = Column(Integer, primary\_key=**True**)

keyword = Column('keyword', String(64))

**def** \_\_init\_\_(self, keyword):

self.keyword = keyword

**def** \_\_repr\_\_(self):

**return** 'Keyword(*%s*)' % repr(self.keyword)

With the above configuration, we can operate upon the .keywords collection of each User object, and the usage of UserKeyword is concealed:

通过上述配置，我们可以对每个User对象的.keywords集合进行操作，隐藏UserKeyword的用法：

**>>>** user = User('log')

**>>> for** kw **in** (Keyword('new\_from\_blammo'), Keyword('its\_big')):

**...**  user.keywords.append(kw)

**...**

**>>>** print(user.keywords)[Keyword('new\_from\_blammo'), Keyword('its\_big')]

Where above, each .keywords.append() operation is equivalent to:

在上面的每个.keywords.append()操作相当于：

**>>>** user.user\_keywords.append(UserKeyword(Keyword('its\_heavy')))

The UserKeyword association object has two attributes here which are populated; the .keyword attribute is populated directly as a result of passing the Keyword object as the first argument. The .user argument is then assigned as the UserKeyword object is appended to the User.user\_keywords collection, where the bidirectional relationship configured between User.user\_keywords and UserKeyword.user results in a population of the UserKeyword.user attribute. The special\_key argument above is left at its default value of None.

UserKeyword关联对象在这里已经填充了两个属性; 作为第一个参数传递Keyword 对象的结果，直接填充.keyword属性。 然后分配.user参数，因为UserKeyword对象附加到User.user\_keywords集合中，其中User.user\_keywords和UserKeyword.user之间配置的双向关系导致UserKeyword.user属性的总体。 上面的special\_key参数的默认值为None。

For those cases where we do want special\_key to have a value, we create the UserKeyword object explicitly. Below we assign all three attributes, where the assignment of .user has the effect of the UserKeyword being appended to the User.user\_keywords collection:

对于那些我们希望special\_key有一个值的情况，我们明确地创建了UserKeyword对象。 下面我们分配所有三个属性，其中.user的赋值对UserKeyword的作用被附加到User.user\_keywords集合中：

**>>>** UserKeyword(Keyword('its\_wood'), user, special\_key='my special key')

The association proxy returns to us a collection of Keyword objects represented by all these operations:

关联代理向我们返回由所有这些操作表示的Keyword对象的集合：

**>>>** user.keywords[Keyword('new\_from\_blammo'), Keyword('its\_big'), Keyword('its\_heavy'), Keyword('its\_wood')]

### **7.1.4 Proxying to Dictionary Based Collections**

The association proxy can proxy to dictionary based collections as well. SQLAlchemy mappings usually use the [attribute\_mapped\_collection()](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "sqlalchemy.orm.collections.attribute_mapped_collection" \o "sqlalchemy.orm.collections.attribute_mapped_collection) collection type to create dictionary collections, as well as the extended techniques described in [Custom Dictionary-Based Collections](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "id1).

关联代理也可以代理到基于字典的集合。 SQLAlchemy映射通常使用[attribute\_mapped\_collection()](http://docs.sqlalchemy.org/en/rel_1_1/orm/collections.html" \l "sqlalchemy.orm.collections.attribute_mapped_collection" \o "sqlalchemy.orm.collections.attribute_mapped_collection)集合类型来创建字典集合，以及在基于字典的自定义集合中描述的扩展技术。

The association proxy adjusts its behavior when it detects the usage of a dictionary-based collection. When new values are added to the dictionary, the association proxy instantiates the intermediary object by passing two arguments to the creation function instead of one, the key and the value. As always, this creation function defaults to the constructor of the intermediary class, and can be customized using the creator argument.

当它检测到基于字典的集合的使用时，关联代理调整其行为。当将新值添加到字典中时，关联代理通过将两个参数传递给创建函数而不是一个键和值来实例化中间对象。和往常一样，这个创建函数默认为中间类的构造函数，并且可以使用creator参数进行自定义。

Below, we modify our UserKeyword example such that the User.user\_keywords collection will now be mapped using a dictionary, where the UserKeyword.special\_key argument will be used as the key for the dictionary. We then apply a creator argument to the User.keywords proxy so that these values are assigned appropriately when new elements are added to the dictionary:

下面我们修改我们的UserKeyword示例，现在将使用一个字典映射User.user\_keywords集合，其中UserKeyword.special\_key参数将被用作字典的键。然后，我们将一个creator参数应用于User.keywords代理，以便在将新元素添加到字典时适当地分配这些值：

**from** **sqlalchemy** **import** Column, Integer, String, ForeignKey

**from** **sqlalchemy.orm** **import** relationship, backref

**from** **sqlalchemy.ext.associationproxy** **import** association\_proxy

**from** **sqlalchemy.ext.declarative** **import** declarative\_base

**from** **sqlalchemy.orm.collections** **import** attribute\_mapped\_collection

Base = declarative\_base()

**class** **User**(Base):

\_\_tablename\_\_ = 'user'

id = Column(Integer, primary\_key=**True**)

name = Column(String(64))

*# proxy to 'user\_keywords', instantiating UserKeyword*

*# assigning the new key to 'special\_key', values to*

*# 'keyword'.*

keywords = association\_proxy('user\_keywords', 'keyword',

creator=**lambda** k, v:

UserKeyword(special\_key=k, keyword=v)

)

**def** \_\_init\_\_(self, name):

self.name = name

**class** **UserKeyword**(Base):

\_\_tablename\_\_ = 'user\_keyword'

user\_id = Column(Integer, ForeignKey('user.id'), primary\_key=**True**)

keyword\_id = Column(Integer, ForeignKey('keyword.id'), primary\_key=**True**)

special\_key = Column(String)

*# bidirectional user/user\_keywords relationships, mapping*

*# user\_keywords with a dictionary against "special\_key" as key.*

user = relationship(User, backref=backref(

"user\_keywords",

collection\_class=attribute\_mapped\_collection("special\_key"),

cascade="all, delete-orphan"

)

)

keyword = relationship("Keyword")

**class** **Keyword**(Base):

\_\_tablename\_\_ = 'keyword'

id = Column(Integer, primary\_key=**True**)

keyword = Column('keyword', String(64))

**def** \_\_init\_\_(self, keyword):

self.keyword = keyword

**def** \_\_repr\_\_(self):

**return** 'Keyword(*%s*)' % repr(self.keyword)

We illustrate the .keywords collection as a dictionary, mapping the UserKeyword.string\_key value to Keyword objects:

我们将.keywords集合说明为字典，将UserKeyword.string\_key值映射到关键字对象：

**>>>** user = User('log')

**>>>** user.keywords['sk1'] = Keyword('kw1')

**>>>** user.keywords['sk2'] = Keyword('kw2')

**>>>** print(user.keywords){'sk1': Keyword('kw1'), 'sk2': Keyword('kw2')}

### **7.1.5 Composite Association Proxies**

Given our previous examples of proxying from relationship to scalar attribute, proxying across an association object, and proxying dictionaries, we can combine all three techniques together to give User a keywords dictionary that deals strictly with the string value of special\_key mapped to the string keyword. Both the UserKeyword and Keyword classes are entirely concealed. This is achieved by building an association proxy on User that refers to an association proxy present on UserKeyword:

鉴于我们以前从关系到标量属性的代理示例，代理关联对象和代理字典，我们可以将所有三种技术组合在一起，为User提供严格处理映射到字符串keyword的special\_key的字符串值的keywords字典。 UserKeyword和Keyword类都完全隐藏。 这通过在User上建立关联代理来实现，该关联代理引用UserKeyword上存在的关联代理：

**from** **sqlalchemy** **import** Column, Integer, String, ForeignKey

**from** **sqlalchemy.orm** **import** relationship, backref

**from** **sqlalchemy.ext.associationproxy** **import** association\_proxy

**from** **sqlalchemy.ext.declarative** **import** declarative\_base

**from** **sqlalchemy.orm.collections** **import** attribute\_mapped\_collection

Base = declarative\_base()

**class** **User**(Base):

\_\_tablename\_\_ = 'user'

id = Column(Integer, primary\_key=**True**)

name = Column(String(64))

*# the same 'user\_keywords'->'keyword' proxy as in*

*# the basic dictionary example*

keywords = association\_proxy(

'user\_keywords',

'keyword',

creator=**lambda** k, v:

UserKeyword(special\_key=k, keyword=v)

)

**def** \_\_init\_\_(self, name):

self.name = name

**class** **UserKeyword**(Base):

\_\_tablename\_\_ = 'user\_keyword'

user\_id = Column(Integer, ForeignKey('user.id'), primary\_key=**True**)

keyword\_id = Column(Integer, ForeignKey('keyword.id'),

primary\_key=**True**)

special\_key = Column(String)

user = relationship(User, backref=backref(

"user\_keywords",

collection\_class=attribute\_mapped\_collection("special\_key"),

cascade="all, delete-orphan"

)

)

*# the relationship to Keyword is now called*

*# 'kw'*

kw = relationship("Keyword")

*# 'keyword' is changed to be a proxy to the*

*# 'keyword' attribute of 'Keyword'*

keyword = association\_proxy('kw', 'keyword')

**class** **Keyword**(Base):

\_\_tablename\_\_ = 'keyword'

id = Column(Integer, primary\_key=**True**)

keyword = Column('keyword', String(64))

**def** \_\_init\_\_(self, keyword):

self.keyword = keyword

User.keywords is now a dictionary of string to string, where UserKeyword and Keyword objects are created and removed for us transparently using the association proxy. In the example below, we illustrate usage of the assignment operator, also appropriately handled by the association proxy, to apply a dictionary value to the collection at once:

User.keywords现在是字符串到字符串的字典，其中为我们透明地使用关联代理创建和删除了UserKeyword和Keyword对象。 在下面的例子中，我们说明了由关联代理适当处理的赋值运算符的使用，可以一次将字典值应用于集合：

**>>>** user = User('log')

**>>>** user.keywords = {

**...**  'sk1':'kw1',

**...**  'sk2':'kw2'

**...** }

**>>>** print(user.keywords){'sk1': 'kw1', 'sk2': 'kw2'}

**>>>** user.keywords['sk3'] = 'kw3'

**>>> del** user.keywords['sk2']

**>>>** print(user.keywords){'sk1': 'kw1', 'sk3': 'kw3'}

**>>>** *# illustrate un-proxied usage*

**...**

print(user.user\_keywords['sk3'].kw)

<\_\_main\_\_.Keyword object at 0x12ceb90>

One caveat with our example above is that because Keyword objects are created for each dictionary set operation, the example fails to maintain uniqueness for the Keyword objects on their string name, which is a typical requirement for a tagging scenario such as this one. For this use case the recipe [UniqueObject](http://www.sqlalchemy.org/trac/wiki/UsageRecipes/UniqueObject), or a comparable creational strategy, is recommended, which will apply a "lookup first, then create" strategy to the constructor of the Keyword class, so that an already existing Keyword is returned if the given name is already present.

以上我们的示例的一个注意事项是，由于为每个字典集操作创建了Keyword对象，因此该示例无法保持Keyword对象在其字符串名称上的唯一性，这对于像这样的标记场景是典型的要求。 对于这种用例，建议使用UniqueObject方法或类似的创建策略，这将对Keyword类的构造函数应用“首先查找，然后创建”策略，因此已经存在的返回Keyword,如果给定的名称为已经存在

7.1.6 Querying with Association Proxies

The [AssociationProxy](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.AssociationProxy" \o "sqlalchemy.ext.associationproxy.AssociationProxy) features simple SQL construction capabilities which relate down to the underlying [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) in use as well as the target attribute. For example, the [RelationshipProperty.Comparator.any()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator.any" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator.any) and [RelationshipProperty.Comparator.has()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator.has" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator.has) operations are available, and will produce a "nested" EXISTS clause, such as in our basic association object example:

[AssociationProxy](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.AssociationProxy" \o "sqlalchemy.ext.associationproxy.AssociationProxy)具有简单的SQL构造功能，它与使用的基础[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)以及目标属性有关。 例如，[RelationshipProperty.Comparator.any()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator.any" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator.any)和[RelationshipProperty.Comparator.has()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator.has" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator.has)操作可用，并将生成“嵌套”EXISTS子句，例如在我们的基本关联对象示例中：

**>>>** print(session.query(User).filter(User.keywords.any(keyword='jek')))

SELECT user.id AS user\_id, user.name AS user\_name

FROM userWHERE EXISTS (SELECT 1FROM user\_keyword

WHERE user.id = user\_keyword.user\_id

AND (EXISTS (SELECT 1

FROM keyword

WHERE keyword.id = user\_keyword.keyword\_id

AND keyword.keyword = :keyword\_1)))

For a proxy to a scalar attribute, \_\_eq\_\_() is supported:

对于标量属性的代理，支持\_\_eq\_\_()：

**>>>** print(session.query(UserKeyword).filter(UserKeyword.keyword == 'jek'))

SELECT user\_keyword.\*FROM user\_keyword

WHERE EXISTS (SELECT 1 FROM keyword WHERE keyword.id = user\_keyword.keyword\_id AND keyword.keyword = :keyword\_1)

and .contains() is available for a proxy to a scalar collection:

**>>>** print(session.query(User).filter(User.keywords.contains('jek')))

SELECT user.\*FROM user

WHERE EXISTS (SELECT 1FROM userkeywords, keyword

WHERE user.id = userkeywords.user\_id

AND keyword.id = userkeywords.keyword\_id

AND keyword.keyword = :keyword\_1)

[AssociationProxy](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.AssociationProxy" \o "sqlalchemy.ext.associationproxy.AssociationProxy) can be used with [Query.join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join) somewhat manually using the [attr](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.AssociationProxy.attr" \o "sqlalchemy.ext.associationproxy.AssociationProxy.attr) attribute in a star-args context:

[AssociationProxy](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.AssociationProxy" \o "sqlalchemy.ext.associationproxy.AssociationProxy)可以与[Query.join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join)一起使用star-args上下文中的[attr](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.AssociationProxy.attr" \o "sqlalchemy.ext.associationproxy.AssociationProxy.attr)属性手动使用：

q = session.query(User).join(\*User.keywords.attr)

*New in version 0.7.3:*[attr](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.AssociationProxy.attr" \o "sqlalchemy.ext.associationproxy.AssociationProxy.attr) attribute in a star-args context.

0.7.3版本中的新功能：star-args上下文中的[attr](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.AssociationProxy.attr" \o "sqlalchemy.ext.associationproxy.AssociationProxy.attr)属性。

[attr](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.AssociationProxy.attr" \o "sqlalchemy.ext.associationproxy.AssociationProxy.attr) is composed of [AssociationProxy.local\_attr](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.AssociationProxy.local_attr" \o "sqlalchemy.ext.associationproxy.AssociationProxy.local_attr) and [AssociationProxy.remote\_attr](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.AssociationProxy.remote_attr" \o "sqlalchemy.ext.associationproxy.AssociationProxy.remote_attr), which are just synonyms for the actual proxied attributes, and can also be used for querying:

[attr](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.AssociationProxy.attr" \o "sqlalchemy.ext.associationproxy.AssociationProxy.attr)由[AssociationProxy.local\_attr](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.AssociationProxy.local_attr" \o "sqlalchemy.ext.associationproxy.AssociationProxy.local_attr)和[AssociationProxy.remote\_attr](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.AssociationProxy.remote_attr" \o "sqlalchemy.ext.associationproxy.AssociationProxy.remote_attr)组成，它们只是实际代理属性的同义词，也可用于查询：

uka = aliased(UserKeyword)ka = aliased(Keyword)q = session.query(User).\

join(uka, User.keywords.local\_attr).\

join(ka, User.keywords.remote\_attr)

*New in version 0.7.3:*[AssociationProxy.local\_attr](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.AssociationProxy.local_attr" \o "sqlalchemy.ext.associationproxy.AssociationProxy.local_attr) and [AssociationProxy.remote\_attr](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.AssociationProxy.remote_attr" \o "sqlalchemy.ext.associationproxy.AssociationProxy.remote_attr), synonyms for the actual proxied attributes, and usable for querying.

新版本0.7.3：[AssociationProxy.local\_attr](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.AssociationProxy.local_attr" \o "sqlalchemy.ext.associationproxy.AssociationProxy.local_attr)和[AssociationProxy.remote\_attr](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.AssociationProxy.remote_attr" \o "sqlalchemy.ext.associationproxy.AssociationProxy.remote_attr)，实际代理属性的同义词，可用于查询。

7.1.7 API Documentation

sqlalchemy.ext.associationproxy.**association\_proxy**(*target\_collection*, *attr*, *\*\*kw*)

Return a Python property implementing a view of a target attribute which references an attribute on members of the target.

返回一个Python属性，实现一个目标属性的视图，它引用了目标成员的属性。

The returned value is an instance of [AssociationProxy](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.AssociationProxy" \o "sqlalchemy.ext.associationproxy.AssociationProxy).

返回的值是AssociationProxy的一个实例。

Implements a Python property representing a relationship as a collection of simpler values, or a scalar value. The proxied property will mimic the collection type of the target (list, dict or set), or, in the case of a one to one relationship, a simple scalar value.

实现一个表示一个关系的Python属性作为一个更简单的值或一个标量值的集合。 代理的属性将模拟目标(list，dict或set)的集合类型，或者在一对一关系的情况下，将是一个简单的标量值。

|  |  |
| --- | --- |
| **Parameters:** | * ****target\_collection**** – Name of the attribute we'll proxy to. This attribute is typically mapped by [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) to link to a target collection, but can also be a many-to-one or non-scalar relationship.我们要代理的属性的名称。 此属性通常由[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)映射以链接到目标集合，但也可以是多对一或非标量关系。 * ****attr –****Attribute on the associated instance or instances we'll proxy for.在我们将要代理的关联实例或实例上的属性。   For example, given a target collection of [obj1, obj2], a list created by this proxy property would look like [getattr(obj1, *attr*), getattr(obj2, *attr*)]  例如，给定目标集合[obj1，obj2]，由此代理创建的列表将如[getattr(obj1，attr)，getattr(obj2，attr)])  If the relationship is one-to-one or otherwise uselist=False, then simply: getattr(obj, *attr*)  如果关系是一对一的，或者其他的uselist = False，那么简单地说：getattr(obj，attr)   * ****creator –****optional.   When new items are added to this proxied collection, new instances of the class collected by the target collection will be created. For list and set collections, the target class constructor will be called with the 'value' for the new instance. For dict types, two arguments are passed: key and value.将新项目添加到此代理集合中时，将创建目标集合收集的类的新实例。 对于列表和集合集合，将使用新实例的“值”来调用目标类构造函数。 对于dict类型，将传递两个参数：key和value。  If you want to construct instances differently, supply a *creator* function that takes arguments as above and returns instances.  For scalar relationships, creator() will be called if the target is None. If the target is present, set operations are proxied to setattr() on the associated object.  If you have an associated object with multiple attributes, you may set up multiple association proxies mapping to different attributes. See the unit tests for examples, and for examples of how creator() functions can be used to construct the scalar relationship on-demand in this situation.   * ****\*\*kw**** – Passes along any other keyword arguments to [AssociationProxy](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.AssociationProxy" \o "sqlalchemy.ext.associationproxy.AssociationProxy). |

*class*sqlalchemy.ext.associationproxy.**AssociationProxy**(*target\_collection*, *attr*, *creator=None*, *getset\_factory=None*, *proxy\_factory=None*, *proxy\_bulk\_set=None*, *info=None*)

Bases: [sqlalchemy.orm.base.InspectionAttrInfo](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.base.InspectionAttrInfo" \o "sqlalchemy.orm.base.InspectionAttrInfo)

A descriptor that presents a read/write view of an object attribute.

呈现对象属性的读/写视图的描述符。

**\_\_init\_\_**(*target\_collection*, *attr*, *creator=None*, *getset\_factory=None*, *proxy\_factory=None*, *proxy\_bulk\_set=None*, *info=None*)

Construct a new [AssociationProxy](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.AssociationProxy" \o "sqlalchemy.ext.associationproxy.AssociationProxy).

构建一个新的AssociationProxy。

The [association\_proxy()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.association_proxy" \o "sqlalchemy.ext.associationproxy.association_proxy) function is provided as the usual entrypoint here, though [AssociationProxy](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.AssociationProxy" \o "sqlalchemy.ext.associationproxy.AssociationProxy) can be instantiated and/or subclassed directly.

association\_proxy()函数作为通常的entrypoint在这里提供，尽管AssociationProxy可以直接实例化和/或子类化。

|  |  |
| --- | --- |
| **Parameters:** | * ****target\_collection**** – Name of the collection we'll proxy to, usually created with [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship). * ****attr**** – Attribute on the collected instances we'll proxy for. For example, given a target collection of [obj1, obj2], a list created by this proxy property would look like [getattr(obj1, attr), getattr(obj2, attr)] * ****creator –****Optional. When new items are added to this proxied collection, new instances of the class collected by the target collection will be created. For list and set collections, the target class constructor will be called with the 'value' for the new instance. For dict types, two arguments are passed: key and value.   If you want to construct instances differently, supply a 'creator' function that takes arguments as above and returns instances.   * ****getset\_factory –****Optional. Proxied attribute access is automatically handled by routines that get and set values based on the attr argument for this proxy.   If you would like to customize this behavior, you may supply a getset\_factory callable that produces a tuple of getter and setter functions. The factory is called with two arguments, the abstract type of the underlying collection and this proxy instance.   * ****proxy\_factory**** – Optional. The type of collection to emulate is determined by sniffing the target collection. If your collection type can't be determined by duck typing or you'd like to use a different collection implementation, you may supply a factory function to produce those collections. Only applicable to non-scalar relationships. * ****proxy\_bulk\_set**** – Optional, use with proxy\_factory. See the \_set() method for details. * ****info –****optional, will be assigned to [AssociationProxy.info](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.AssociationProxy.info" \o "sqlalchemy.ext.associationproxy.AssociationProxy.info) if present.   *New in version 1.0.9.* |

**any**(*criterion=None*, *\*\*kwargs*)

Produce a proxied 'any' expression using EXISTS.

This expression will be a composed product using the [RelationshipProperty.Comparator.any()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator.any" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator.any) and/or [RelationshipProperty.Comparator.has()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator.has" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator.has) operators of the underlying proxied attributes.

**attr**

Return a tuple of (local\_attr, remote\_attr).

This attribute is convenient when specifying a join using [Query.join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join) across two relationships:

sess.query(Parent).join(\*Parent.proxied.attr)

*New in version 0.7.3.*

See also:

[AssociationProxy.local\_attr](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.AssociationProxy.local_attr" \o "sqlalchemy.ext.associationproxy.AssociationProxy.local_attr)

[AssociationProxy.remote\_attr](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.AssociationProxy.remote_attr" \o "sqlalchemy.ext.associationproxy.AssociationProxy.remote_attr)

**contains**(*obj*)

Produce a proxied 'contains' expression using EXISTS.

This expression will be a composed product using the [RelationshipProperty.Comparator.any()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator.any" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator.any) , [RelationshipProperty.Comparator.has()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator.has" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator.has), and/or [RelationshipProperty.Comparator.contains()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator.contains" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator.contains) operators of the underlying proxied attributes.

**extension\_type***= symbol('ASSOCIATION\_PROXY')*

**has**(*criterion=None*, *\*\*kwargs*)

Produce a proxied 'has' expression using EXISTS.

This expression will be a composed product using the [RelationshipProperty.Comparator.any()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator.any" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator.any) and/or [RelationshipProperty.Comparator.has()](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.properties.RelationshipProperty.Comparator.has" \o "sqlalchemy.orm.properties.RelationshipProperty.Comparator.has) operators of the underlying proxied attributes.

**info**

*inherited from the* [info](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.base.InspectionAttrInfo.info" \o "sqlalchemy.orm.base.InspectionAttrInfo.info) *attribute of* [InspectionAttrInfo](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.base.InspectionAttrInfo" \o "sqlalchemy.orm.base.InspectionAttrInfo)

Info dictionary associated with the object, allowing user-defined data to be associated with this [InspectionAttr](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.base.InspectionAttr" \o "sqlalchemy.orm.base.InspectionAttr).

The dictionary is generated when first accessed. Alternatively, it can be specified as a constructor argument to the [column\_property()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_columns.html" \l "sqlalchemy.orm.column_property" \o "sqlalchemy.orm.column_property), [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship), or [composite()](http://docs.sqlalchemy.org/en/rel_1_1/orm/composites.html" \l "sqlalchemy.orm.composite" \o "sqlalchemy.orm.composite) functions.

*New in version 0.8:*Added support for .info to all [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty) subclasses.

*Changed in version 1.0.0:*[MapperProperty.info](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "MapperProperty.info" \o "MapperProperty.info) is also available on extension types via the [InspectionAttrInfo.info](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.base.InspectionAttrInfo.info" \o "sqlalchemy.orm.base.InspectionAttrInfo.info) attribute, so that it can apply to a wider variety of ORM and extension constructs.

**See also**

[QueryableAttribute.info](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.attributes.QueryableAttribute.info" \o "sqlalchemy.orm.attributes.QueryableAttribute.info)

[SchemaItem.info](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem.info" \o "sqlalchemy.schema.SchemaItem.info)

**is\_aliased\_class***= False*

**is\_attribute***= False*

**is\_clause\_element***= False*

**is\_instance***= False*

**is\_mapper***= False*

**is\_property***= False*

**is\_selectable***= False*

**local\_attr**

The 'local' [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty) referenced by this [AssociationProxy](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.AssociationProxy" \o "sqlalchemy.ext.associationproxy.AssociationProxy).

*New in version 0.7.3.*

See also:

[AssociationProxy.attr](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.AssociationProxy.attr" \o "sqlalchemy.ext.associationproxy.AssociationProxy.attr)

[AssociationProxy.remote\_attr](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.AssociationProxy.remote_attr" \o "sqlalchemy.ext.associationproxy.AssociationProxy.remote_attr)

**remote\_attr**

The 'remote' [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty) referenced by this [AssociationProxy](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.AssociationProxy" \o "sqlalchemy.ext.associationproxy.AssociationProxy).

*New in version 0.7.3.*

See also:

[AssociationProxy.attr](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.AssociationProxy.attr" \o "sqlalchemy.ext.associationproxy.AssociationProxy.attr)

[AssociationProxy.local\_attr](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.AssociationProxy.local_attr" \o "sqlalchemy.ext.associationproxy.AssociationProxy.local_attr)

**scalar**

Return True if this [AssociationProxy](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.AssociationProxy" \o "sqlalchemy.ext.associationproxy.AssociationProxy) proxies a scalar relationship on the local side.

**target\_class**

The intermediary class handled by this [AssociationProxy](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.AssociationProxy" \o "sqlalchemy.ext.associationproxy.AssociationProxy).

Intercepted append/set/assignment events will result in the generation of new instances of this class.

sqlalchemy.ext.associationproxy.**ASSOCIATION\_PROXY***= symbol('ASSOCIATION\_PROXY')*

## 7.2 Automap

Define an extension to the [sqlalchemy.ext.declarative](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "module-sqlalchemy.ext.declarative" \o "sqlalchemy.ext.declarative) system which automatically generates mapped classes and relationships from a database schema, typically though not necessarily one which is reflected.

定义[sqlalchemy.ext.declarative](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "module-sqlalchemy.ext.declarative" \o "sqlalchemy.ext.declarative)系统的扩展，它自动从数据库模式生成映射的类和关系，通常不一定反映出来。

*New in version 0.9.1:*Added [sqlalchemy.ext.automap](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "module-sqlalchemy.ext.automap" \o "sqlalchemy.ext.automap).

0.9.1版本中新增了[sqlalchemy.ext.automap](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "module-sqlalchemy.ext.automap" \o "sqlalchemy.ext.automap)。

It is hoped that the [AutomapBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase" \o "sqlalchemy.ext.automap.AutomapBase) system provides a quick and modernized solution to the problem that the very famous [SQLSoup](https://sqlsoup.readthedocs.io/en/latest/) also tries to solve, that of generating a quick and rudimentary object model from an existing database on the fly. By addressing the issue strictly at the mapper configuration level, and integrating fully with existing Declarative class techniques, [AutomapBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase" \o "sqlalchemy.ext.automap.AutomapBase) seeks to provide a well-integrated approach to the issue of expediently auto-generating ad-hoc mappings.

希望[AutomapBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase" \o "sqlalchemy.ext.automap.AutomapBase)系统能够为这个问题提供一个快速而现代化的解决方案，著名的SQLSoup也试图解决，即从现有数据库中快速生成基本的对象模型。 通过在映射器配置级别严格地解决这个问题，并完全集成了现有的Declarative类技术，[AutomapBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase" \o "sqlalchemy.ext.automap.AutomapBase)旨在为自适应自动生成特殊映射的问题提供一个良好集成的方法。

### 7.2.1 Basic Use

The simplest usage is to reflect an existing database into a new model. We create a new [AutomapBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase" \o "sqlalchemy.ext.automap.AutomapBase) class in a similar manner as to how we create a declarative base class, using [automap\_base()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.automap_base" \o "sqlalchemy.ext.automap.automap_base). We then call [AutomapBase.prepare()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase.prepare" \o "sqlalchemy.ext.automap.AutomapBase.prepare) on the resulting base class, asking it to reflect the schema and produce mappings:

最简单的用法是将现有的数据库反映到新的模型中。 我们以类似于使用[automap\_base()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.automap_base" \o "sqlalchemy.ext.automap.automap_base)创建声明性基类的方式创建一个新的[AutomapBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase" \o "sqlalchemy.ext.automap.AutomapBase)类。 然后我们在生成的基类上调用[AutomapBase.prepare()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase.prepare" \o "sqlalchemy.ext.automap.AutomapBase.prepare)，要求它反映模式并产生映射：

**from** **sqlalchemy.ext.automap** **import** automap\_base

**from** **sqlalchemy.orm** **import** Session

**from** **sqlalchemy** **import** create\_engine

Base = automap\_base()

*# engine, suppose it has two tables 'user' and 'address' set up*

engine = create\_engine("sqlite:///mydatabase.db")

*# reflect the tables*

Base.prepare(engine, reflect=**True**)

*# mapped classes are now created with names by default*

*# matching that of the table name.*

User = Base.classes.user

Address = Base.classes.address

session = Session(engine)

*# rudimentary relationships are produced*

session.add(Address(email\_address="foo@bar.com", user=User(name="foo")))session.commit()

*# collection-based relationships are by default named*

*# "<classname>\_collection"*

print (u1.address\_collection)

Above, calling [AutomapBase.prepare()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase.prepare" \o "sqlalchemy.ext.automap.AutomapBase.prepare) while passing along the [AutomapBase.prepare.reflect](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase.prepare.params.reflect" \o "sqlalchemy.ext.automap.AutomapBase.prepare) parameter indicates that the [MetaData.reflect()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData.reflect" \o "sqlalchemy.schema.MetaData.reflect)method will be called on this declarative base classes' [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) collection; then, each ****viable**** [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) within the [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) will get a new mapped class generated automatically.

以上，在传递[AutomapBase.prepare.reflect](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase.prepare.params.reflect" \o "sqlalchemy.ext.automap.AutomapBase.prepare)参数时调用[AutomapBase.prepare()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase.prepare" \o "sqlalchemy.ext.automap.AutomapBase.prepare)，表示将在此声明性基类[MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData)集合上调用[MetaData.reflect()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData.reflect" \o "sqlalchemy.schema.MetaData.reflect)方法; 那么，[MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData)中的每个可行的[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)将会自动获得一个新的映射类。

The [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint) objects which link the various tables together will be used to produce new, bidirectional [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) objects between classes. The classes and relationships follow along a default naming scheme that we can customize. At this point, our basic mapping consisting of related User and Address classes is ready to use in the traditional way.

将各种表链接在一起的[ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint)对象将用于在类之间生成新的双向[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)对象。 类和关系遵循我们可以自定义的默认命名方案。 在这一点上，我们的基本映射包括相关的User和Address类可以传统的方式使用。

**Note**

By ****viable****, we mean that for a table to be mapped, it must specify a primary key. Additionally, if the table is detected as being a pure association table between two other tables, it will not be directly mapped and will instead be configured as a many-to-many table between the mappings for the two referring tables.

通过变量，我们的意思是对于要映射的表，它必须指定一个主键。 另外，如果表被检测为两个其他表之间的纯关联表，则不会直接映射表，而是将其配置为两个引用表的映射之间的多对多表。

### 7.2.2 Generating Mappings from an Existing MetaData

We can pass a pre-declared [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) object to [automap\_base()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.automap_base" \o "sqlalchemy.ext.automap.automap_base). This object can be constructed in any way, including programmatically, from a serialized file, or from itself being reflected using [MetaData.reflect()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData.reflect" \o "sqlalchemy.schema.MetaData.reflect). Below we illustrate a combination of reflection and explicit table declaration:

我们可以将预先声明的[MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData)对象传递给[automap\_base()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.automap_base" \o "sqlalchemy.ext.automap.automap_base)。 该对象可以以任何方式构建，包括以编程方式，从序列化的文件，或从本身使用[MetaData.reflect()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData.reflect" \o "sqlalchemy.schema.MetaData.reflect)反映。 下面我们来说明反射和显式表声明的组合：

**from** **sqlalchemy** **import** create\_engine, MetaData, Table, Column, ForeignKey

**from** **sqlalchemy.ext.automap** **import** automap\_base

engine = create\_engine("sqlite:///mydatabase.db")

*# produce our own MetaData object*

metadata = MetaData()

*# we can reflect it ourselves from a database, using options# such as 'only' to limit what tables we look at...*

metadata.reflect(engine, only=['user', 'address'])

*# ... or just define our own Table objects with it (or combine both)*

Table('user\_order', metadata,

Column('id', Integer, primary\_key=**True**),

Column('user\_id', ForeignKey('user.id'))

)

*# we can then produce a set of mappings from this MetaData.*

Base = automap\_base(metadata=metadata)

*# calling prepare() just sets up mapped classes and relationships.*

Base.prepare()

*# mapped classes are ready*

User, Address, Order = Base.classes.user, Base.classes.address,\

Base.classes.user\_order

### 7.2.3 Specifying Classes Explicitly

The [sqlalchemy.ext.automap](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "module-sqlalchemy.ext.automap" \o "sqlalchemy.ext.automap) extension allows classes to be defined explicitly, in a way similar to that of the [DeferredReflection](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.DeferredReflection" \o "sqlalchemy.ext.declarative.DeferredReflection) class. Classes that extend from [AutomapBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase" \o "sqlalchemy.ext.automap.AutomapBase) act like regular declarative classes, but are not immediately mapped after their construction, and are instead mapped when we call [AutomapBase.prepare()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase.prepare" \o "sqlalchemy.ext.automap.AutomapBase.prepare). The [AutomapBase.prepare()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase.prepare" \o "sqlalchemy.ext.automap.AutomapBase.prepare) method will make use of the classes we've established based on the table name we use. If our schema contains tables user and address, we can define one or both of the classes to be used:

[sqlalchemy.ext.automap](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "module-sqlalchemy.ext.automap" \o "sqlalchemy.ext.automap)扩展允许以类似于[DeferredReflection](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.DeferredReflection" \o "sqlalchemy.ext.declarative.DeferredReflection)类的方式显式定义类。 从[AutomapBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase" \o "sqlalchemy.ext.automap.AutomapBase)扩展的类似于常规声明类，但在构造之后不会立即映射，而是在调用[AutomapBase.prepare()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase.prepare" \o "sqlalchemy.ext.automap.AutomapBase.prepare)时映射。该[AutomapBase.prepare()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase.prepare" \o "sqlalchemy.ext.automap.AutomapBase.prepare)方法将利用我们基于我们使用的表名建立的类。 如果我们的模式包含user和address，我们可以定义要使用的一个或两个类：

**from** **sqlalchemy.ext.automap** **import** automap\_base

**from** **sqlalchemy** **import** create\_engine

*# automap base*

Base = automap\_base()

*# pre-declare User for the 'user' table*

**class** **User**(Base):

\_\_tablename\_\_ = 'user'

*# override schema elements like Columns*

user\_name = Column('name', String)

*# override relationships too, if desired.*

*# we must use the same name that automap would use for the*

*# relationship, and also must refer to the class name that automap will*

*# generate for "address"*

address\_collection = relationship("address", collection\_class=set)

*# reflect*

engine = create\_engine("sqlite:///mydatabase.db")

Base.prepare(engine, reflect=**True**)

*# we still have Address generated from the tablename "address",# but User is the same as Base.classes.User now*

Address = Base.classes.address

u1 = session.query(User).first()

print (u1.address\_collection)

*# the backref is still there:*

a1 = session.query(Address).first()

print (a1.user)

Above, one of the more intricate details is that we illustrated overriding one of the [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) objects that automap would have created. To do this, we needed to make sure the names match up with what automap would normally generate, in that the relationship name would be User.address\_collection and the name of the class referred to, from automap's perspective, is called address, even though we are referring to it as Address within our usage of this class.

以上，更复杂的细节之一是我们说明了覆盖automap创建的[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)对象之一。 为了做到这一点，我们需要确保这些名称与automap通常会生成什么匹配，因为关系名称将是User.address\_collection，并且从automap的角度来看，类的名称称为address，即使我们 在我们使用这个类的时候把它称为Address。

### 7.2.4 Overriding Naming Schemes

[sqlalchemy.ext.automap](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "module-sqlalchemy.ext.automap" \o "sqlalchemy.ext.automap) is tasked with producing mapped classes and relationship names based on a schema, which means it has decision points in how these names are determined. These three decision points are provided using functions which can be passed to the [AutomapBase.prepare()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase.prepare" \o "sqlalchemy.ext.automap.AutomapBase.prepare) method, and are known as [classname\_for\_table()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.classname_for_table" \o "sqlalchemy.ext.automap.classname_for_table), [name\_for\_scalar\_relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.name_for_scalar_relationship" \o "sqlalchemy.ext.automap.name_for_scalar_relationship), and [name\_for\_collection\_relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.name_for_collection_relationship" \o "sqlalchemy.ext.automap.name_for_collection_relationship). Any or all of these functions are provided as in the example below, where we use a "camel case" scheme for class names and a "pluralizer" for collection names using the [Inflect](https://pypi.python.org/pypi/inflect) package:

[sqlalchemy.ext.automap](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "module-sqlalchemy.ext.automap" \o "sqlalchemy.ext.automap)的任务是根据模式生成映射的类和关系名称，这意味着它具有如何确定这些名称的决策点。 使用可以传递给[AutomapBase.prepare()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase.prepare" \o "sqlalchemy.ext.automap.AutomapBase.prepare)方法的函数提供这三个决策点，并且被称为[classname\_for\_table()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.classname_for_table" \o "sqlalchemy.ext.automap.classname_for_table)，[name\_for\_scalar\_relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.name_for_scalar_relationship" \o "sqlalchemy.ext.automap.name_for_scalar_relationship)和[name\_for\_collection\_relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.name_for_collection_relationship" \o "sqlalchemy.ext.automap.name_for_collection_relationship)。 这些功能中的任何一个或全部是在下面的例子中提供的，我们使用“类型名称”的“camel case”方案，使用Inflect包为集合名称使用“pluralizer”：

**import** **re**

**import** **inflect**

**def** camelize\_classname(base, tablename, table):

"Produce a 'camelized' class name, e.g. "

"'words\_and\_underscores' -> 'WordsAndUnderscores'"

**return** str(tablename[0].upper() + \

re.sub(r'\_([a-z])', **lambda** m: m.group(1).upper(), tablename[1:]))

\_pluralizer = inflect.engine()

**def** pluralize\_collection(base, local\_cls, referred\_cls, constraint):

"Produce an 'uncamelized', 'pluralized' class name, e.g. "

"'SomeTerm' -> 'some\_terms'"

referred\_name = referred\_cls.\_\_name\_\_

uncamelized = re.sub(r'[A-Z]',

**lambda** m: "\_*%s*" % m.group(0).lower(),

referred\_name)[1:]

pluralized = \_pluralizer.plural(uncamelized)

**return** pluralized

**from** **sqlalchemy.ext.automap** **import** automap\_base

Base = automap\_base()

engine = create\_engine("sqlite:///mydatabase.db")

Base.prepare(engine, reflect=**True**,

classname\_for\_table=camelize\_classname,

name\_for\_collection\_relationship=pluralize\_collection

)

From the above mapping, we would now have classes User and Address, where the collection from User to Address is called User.addresses:

从上面的映射，我们现在将有User和Address类，其中从User到Address的集合称为User.addresses：

User, Address = Base.classes.User, Base.classes.Address

u1 = User(addresses=[Address(email="foo@bar.com")])

### 7.2.5 Relationship Detection

The vast majority of what automap accomplishes is the generation of [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) structures based on foreign keys. The mechanism by which this works for many-to-one and one-to-many relationships is as follows:

自动完成的绝大多数是基于外键的[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)结构的生成。这对于多对一和一对多关系的作用机制如下：

A given [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table), known to be mapped to a particular class, is examined for [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint) objects.

对于[ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint)对象，将检查已知映射到特定类的给定[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)。

From each [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint), the remote [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object present is matched up to the class to which it is to be mapped, if any, else it is skipped.

从每个[ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint)，远程[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)对象存在与其映射到的类(如果有的话)匹配，否则跳过。

As the [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint) we are examining corresponds to a reference from the immediate mapped class, the relationship will be set up as a many-to-one referring to the referred class; a corresponding one-to-many backref will be created on the referred class referring to this class.

正如我们正在检查的[ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint)对应于来自立即映射类的引用，该关系将被设置为多对一引用引用类;将引用此类的引用类将创建一个对应的一对多backref。

If any of the columns that are part of the [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint) are not nullable (e.g. nullable=False), a [cascade](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.cascade" \o "sqlalchemy.orm.relationship) keyword argument of all,delete-orphan will be added to the keyword arguments to be passed to the relationship or backref. If the [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint) reports that [ForeignKeyConstraint.ondelete](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint.params.ondelete" \o "sqlalchemy.schema.ForeignKeyConstraint) is set to CASCADE for a not null or SET NULL for a nullable set of columns, the option [passive\_deletes](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.passive_deletes" \o "sqlalchemy.orm.relationship) flag is set to True in the set of relationship keyword arguments. Note that not all backends support reflection of ON DELETE.

如果作为[ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint)的一部分的任何列不可为空(例如，nullable=False)，则将all，delete-orphan的[cascade](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.cascade" \o "sqlalchemy.orm.relationship)关键字参数添加到要传递给关系或backref的关键字参数中。如果[ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint)报告[ForeignKeyConstraint.ondelete](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint.params.ondelete" \o "sqlalchemy.schema.ForeignKeyConstraint)为可空的一组列设置为CASCADE为非空值或SET NULL，则在关系关键字参数集中将“[passive\_deletes](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.passive_deletes" \o "sqlalchemy.orm.relationship)”选项设置为True。请注意，并非所有后端都支持ON DELETE的反射。

*New in version 1.0.0:*- automap will detect non-nullable foreign key constraints when producing a one-to-many relationship and establish a default cascade of all, delete-orphan if so; additionally, if the constraint specifies [ForeignKeyConstraint.ondelete](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint.params.ondelete" \o "sqlalchemy.schema.ForeignKeyConstraint) of CASCADE for non-nullable or SET NULL for nullable columns, the passive\_deletes=True option is also added.

版本1.0.0中的新功能： - 在生成一对多关系时，automap将检测到不可空的外键约束，并建立一个默认的所有级别，删除 - 孤立，如果是这样;另外，如果约束指定CASCADE的ForeignKeyConstraint.ondelete为不可为空，或者为可空列设置为NULL，那么也会添加passive\_deletes = True选项。

The names of the relationships are determined using the [AutomapBase.prepare.name\_for\_scalar\_relationship](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase.prepare.params.name_for_scalar_relationship" \o "sqlalchemy.ext.automap.AutomapBase.prepare) and [AutomapBase.prepare.name\_for\_collection\_relationship](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase.prepare.params.name_for_collection_relationship" \o "sqlalchemy.ext.automap.AutomapBase.prepare) callable functions. It is important to note that the default relationship naming derives the name from the ****the actual class name****. If you've given a particular class an explicit name by declaring it, or specified an alternate class naming scheme, that's the name from which the relationship name will be derived.

关系的名称使用[AutomapBase.prepare.name\_for\_scalar\_relationship](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase.prepare.params.name_for_scalar_relationship" \o "sqlalchemy.ext.automap.AutomapBase.prepare)和[AutomapBase.prepare.name\_for\_collection\_relationship](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase.prepare.params.name_for_collection_relationship" \o "sqlalchemy.ext.automap.AutomapBase.prepare)可调用函数确定。重要的是要注意，默认关系命名从实际的类名称中导出名称。如果您通过声明某个特定类来给出明确的名称，或者指定了一个替代类命名方案，则该名称将从其中导出关系名称。

The classes are inspected for an existing mapped property matching these names. If one is detected on one side, but none on the other side, [AutomapBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase" \o "sqlalchemy.ext.automap.AutomapBase) attempts to create a relationship on the missing side, then uses the [relationship.back\_populates](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.back_populates" \o "sqlalchemy.orm.relationship) parameter in order to point the new relationship to the other side.

检查与这些名称匹配的现有映射属性的类。如果一方检测到一个，但另一侧没有检测到，则[AutomapBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase" \o "sqlalchemy.ext.automap.AutomapBase)试图在缺失的一方创建关系，然后使用[relationship.back\_populates](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.back_populates" \o "sqlalchemy.orm.relationship)参数将新关系指向另一方。

In the usual case where no relationship is on either side, [AutomapBase.prepare()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase.prepare" \o "sqlalchemy.ext.automap.AutomapBase.prepare) produces a [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) on the "many-to-one" side and matches it to the other using the [relationship.backref](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.backref" \o "sqlalchemy.orm.relationship) parameter.

在任何一方没有关系的通常情况下，[AutomapBase.prepare()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase.prepare" \o "sqlalchemy.ext.automap.AutomapBase.prepare)在“多对一”一边产生一个[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)，并使用[relationship.backref](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.backref" \o "sqlalchemy.orm.relationship)参数将其与另一个相匹配。

Production of the [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) and optionally the [backref()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.backref" \o "sqlalchemy.orm.backref) is handed off to the [AutomapBase.prepare.generate\_relationship](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase.prepare.params.generate_relationship" \o "sqlalchemy.ext.automap.AutomapBase.prepare) function, which can be supplied by the end-user in order to augment the arguments passed to [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) or [backref()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.backref" \o "sqlalchemy.orm.backref) or to make use of custom implementations of these functions.

将[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)和可选的[backref()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.backref" \o "sqlalchemy.orm.backref)的生成交给[AutomapBase.prepare.generate\_relationship](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase.prepare.params.generate_relationship" \o "sqlalchemy.ext.automap.AutomapBase.prepare)函数，该函数可由最终用户提供，以便扩展传递给[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)或[backref()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.backref" \o "sqlalchemy.orm.backref)的参数或利用这些功能的自定义实现。

#### 7.2.5.1 Custom Relationship Arguments

The [AutomapBase.prepare.generate\_relationship](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase.prepare.params.generate_relationship" \o "sqlalchemy.ext.automap.AutomapBase.prepare) hook can be used to add parameters to relationships. For most cases, we can make use of the existing [automap.generate\_relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.generate_relationship" \o "sqlalchemy.ext.automap.generate_relationship) function to return the object, after augmenting the given keyword dictionary with our own arguments.

[AutomapBase.prepare.generate\_relationship](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase.prepare.params.generate_relationship" \o "sqlalchemy.ext.automap.AutomapBase.prepare)钩子可用于向关系添加参数。 在大多数情况下，我们可以利用现有的[automap.generate\_relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.generate_relationship" \o "sqlalchemy.ext.automap.generate_relationship)函数返回对象，然后使用我们自己的参数来增加给定的关键字字典。

Below is an illustration of how to send [relationship.cascade](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.cascade" \o "sqlalchemy.orm.relationship) and [relationship.passive\_deletes](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.passive_deletes" \o "sqlalchemy.orm.relationship) options along to all one-to-many relationships:

以下是关于如何向所有一对多关系发送[relationship.cascade](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.cascade" \o "sqlalchemy.orm.relationship)和[relationship.passive\_deletes](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.passive_deletes" \o "sqlalchemy.orm.relationship)选项的说明：

**from** **sqlalchemy.ext.automap** **import** generate\_relationship

**def** \_gen\_relationship(base, direction, return\_fn,

attrname, local\_cls, referred\_cls, \*\*kw):

**if** direction **is** interfaces.ONETOMANY:

kw['cascade'] = 'all, delete-orphan'

kw['passive\_deletes'] = **True**

*# make use of the built-in function to actually return*

*# the result.*

**return** generate\_relationship(base, direction, return\_fn,

attrname, local\_cls, referred\_cls, \*\*kw)

**from** **sqlalchemy.ext.automap** **import** automap\_base

**from** **sqlalchemy** **import** create\_engine

*# automap base*

Base = automap\_base()

engine = create\_engine("sqlite:///mydatabase.db")

Base.prepare(engine, reflect=**True**,

generate\_relationship=\_gen\_relationship)

### **7.2.5.2 Many-to-Many relationships**

[sqlalchemy.ext.automap](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "module-sqlalchemy.ext.automap" \o "sqlalchemy.ext.automap) will generate many-to-many relationships, e.g. those which contain a secondary argument. The process for producing these is as follows:

[sqlalchemy.ext.automap](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "module-sqlalchemy.ext.automap" \o "sqlalchemy.ext.automap)将生成多对多关系，例如 那些包含secondary参数。 生成过程如下：

1. A given [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) is examined for [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint) objects, before any mapped class has been assigned to it.在给任何映射类分配给它之前，将给出一个给定的[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)来检查[ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint)对象。
2. If the table contains two and exactly two [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint) objects, and all columns within this table are members of these two [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint) objects, the table is assumed to be a "secondary" table, and will ****not be mapped directly****.如果该表包含两个并且正好两个[ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint)对象，并且此表中的所有列都是这两个[ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint)对象的成员，则该表被假定为“辅助”表，不会直接映射。
3. The two (or one, for self-referential) external tables to which the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) refers to are matched to the classes to which they will be mapped, if any.[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)中引用的两个(或一个，用于自引用的)外部表与它们将被映射的类匹配(如果有的话)。
4. If mapped classes for both sides are located, a many-to-many bi-directional [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) / [backref()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.backref" \o "sqlalchemy.orm.backref) pair is created between the two classes.如果双方的映射类位于两个类之间，则会在两个类之间创建一对多对双向[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) / [backref()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.backref" \o "sqlalchemy.orm.backref)对。
5. The override logic for many-to-many works the same as that of one-to-many/ many-to-one; the [generate\_relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.generate_relationship" \o "sqlalchemy.ext.automap.generate_relationship) function is called upon to generate the strucures and existing attributes will be maintained.多对多作品的超越逻辑与一对多/多对一的作品相同; 调用[generate\_relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.generate_relationship" \o "sqlalchemy.ext.automap.generate_relationship)函数来生成结构，并维护现有的属性。

### **7.2.5.3 Relationships with Inheritance**

[sqlalchemy.ext.automap](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "module-sqlalchemy.ext.automap" \o "sqlalchemy.ext.automap) will not generate any relationships between two classes that are in an inheritance relationship. That is, with two classes given as follows:

[sqlalchemy.ext.automap](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "module-sqlalchemy.ext.automap" \o "sqlalchemy.ext.automap)不会在存在继承关系的两个类之间生成任何关系。 也就是说，两个类别如下给出：

**class** **Employee**(Base):

\_\_tablename\_\_ = 'employee'

id = Column(Integer, primary\_key=**True**)

type = Column(String(50))

\_\_mapper\_args\_\_ = {

'polymorphic\_identity':'employee', 'polymorphic\_on': type

}

**class** **Engineer**(Employee):

\_\_tablename\_\_ = 'engineer'

id = Column(Integer, ForeignKey('employee.id'), primary\_key=**True**)

\_\_mapper\_args\_\_ = {

'polymorphic\_identity':'engineer',

}

The foreign key from Engineer to Employee is used not for a relationship, but to establish joined inheritance between the two classes.

Engineer到Employee的外键不是用于关系，而是建立两类之间的joined inheritance。

Note that this means automap will not generate *any* relationships for foreign keys that link from a subclass to a superclass. If a mapping has actual relationships from subclass to superclass as well, those need to be explicit. Below, as we have two separate foreign keys from Engineer to Employee, we need to set up both the relationship we want as well as the inherit\_condition, as these are not things SQLAlchemy can guess:

请注意，这意味着automap不会为从一个子类链接到一个超类的外键生成任何关系。 如果映射也具有从子类到超类的实际关系，则这些关系必须是明确的。 下面，由于我们有两个独立的外部Engineer到Employee的外键，所以我们需要设置我们想要的关系以及inherit\_condition，因为这些不是SQLAlchemy可以猜到的：

**class** **Employee**(Base):

\_\_tablename\_\_ = 'employee'

id = Column(Integer, primary\_key=**True**)

type = Column(String(50))

\_\_mapper\_args\_\_ = {

'polymorphic\_identity':'employee', 'polymorphic\_on':type

}

**class** **Engineer**(Employee):

\_\_tablename\_\_ = 'engineer'

id = Column(Integer, ForeignKey('employee.id'), primary\_key=**True**)

favorite\_employee\_id = Column(Integer, ForeignKey('employee.id'))

favorite\_employee = relationship(Employee,

foreign\_keys=favorite\_employee\_id)

\_\_mapper\_args\_\_ = {

'polymorphic\_identity':'engineer',

'inherit\_condition': id == Employee.id

}

### **7.2.5.4 Handling Simple Naming Conflicts**

In the case of naming conflicts during mapping, override any of [classname\_for\_table()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.classname_for_table" \o "sqlalchemy.ext.automap.classname_for_table), [name\_for\_scalar\_relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.name_for_scalar_relationship" \o "sqlalchemy.ext.automap.name_for_scalar_relationship), and [name\_for\_collection\_relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.name_for_collection_relationship" \o "sqlalchemy.ext.automap.name_for_collection_relationship) as needed. For example, if automap is attempting to name a many-to-one relationship the same as an existing column, an alternate convention can be conditionally selected. Given a schema:

在映射时命名冲突的情况下，根据需要覆盖[classname\_for\_table()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.classname_for_table" \o "sqlalchemy.ext.automap.classname_for_table)，[name\_for\_scalar\_relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.name_for_scalar_relationship" \o "sqlalchemy.ext.automap.name_for_scalar_relationship)和[name\_for\_collection\_relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.name_for_collection_relationship" \o "sqlalchemy.ext.automap.name_for_collection_relationship)中的任何一个。 例如，如果automap尝试命名与现有列相同的多对一关系，则可以有条件地选择备用约定。 给定一个模式：

**CREATE** **TABLE** table\_a (

id INTEGER **PRIMARY** **KEY**);

**CREATE** **TABLE** table\_b (

id INTEGER **PRIMARY** **KEY**,

table\_a INTEGER,

**FOREIGN** **KEY**(table\_a) **REFERENCES** table\_a(id));

The above schema will first automap the table\_a table as a class named table\_a; it will then automap a relationship onto the class for table\_b with the same name as this related class, e.g. table\_a. This relationship name conflicts with the mapping column table\_b.table\_a, and will emit an error on mapping.

上述模式将首先将table\_a表自动化为名为table\_a的类; 然后它会将与该类相关联的关系自动关联到与此相关类相同名称的table\_b，即table\_a。 此关系名称与映射列table\_b.table\_a冲突，并将在映射时发出错误。

We can resolve this conflict by using an underscore as follows:

我们可以使用下划线来解决这个冲突，如下所示：

**def** name\_for\_scalar\_relationship(base, local\_cls, referred\_cls, constraint):

name = referred\_cls.\_\_name\_\_.lower()

local\_table = local\_cls.\_\_table\_\_

**if** name **in** local\_table.columns:

newname = name + "\_"

warnings.warn(

"Already detected name *%s* present. using *%s*" %

(name, newname))

**return** newname

**return** name

Base.prepare(engine, reflect=**True**,

name\_for\_scalar\_relationship=name\_for\_scalar\_relationship)

Alternatively, we can change the name on the column side. The columns that are mapped can be modified using the technique described at [Naming Columns Distinctly from Attribute Names](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_columns.html" \l "mapper-column-distinct-names), by assigning the column explicitly to a new name:

或者，我们可以更改列侧的名称。 可以使用命名列不同于属性名称的技术，通过将列明确分配给新名称来修改映射的列：

Base = automap\_base()

**class** **TableB**(Base):

\_\_tablename\_\_ = 'table\_b'

\_table\_a = Column('table\_a', ForeignKey('table\_a.id'))

Base.prepare(engine, reflect=**True**)

### 7.2.6 Using Automap with Explicit Declarations

As noted previously, automap has no dependency on reflection, and can make use of any collection of [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects within a [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) collection. From this, it follows that automap can also be used generate missing relationships given an otherwise complete model that fully defines table metadata:

如前所述，automap不依赖于反射，并且可以利用[MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData)集合中的任何[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)对象集合。 由此可见，还可以使用automap来生成缺少的关系，给定一个完全定义表元数据的完整模型：

**from** **sqlalchemy.ext.automap** **import** automap\_base

**from** **sqlalchemy** **import** Column, Integer, String, ForeignKey

Base = automap\_base()

**class** **User**(Base):

\_\_tablename\_\_ = 'user'

id = Column(Integer, primary\_key=**True**)

name = Column(String)

**class** **Address**(Base):

\_\_tablename\_\_ = 'address'

id = Column(Integer, primary\_key=**True**)

email = Column(String)

user\_id = Column(ForeignKey('user.id'))

*# produce relationships*Base.prepare()

*# mapping is complete, with "address\_collection" and# "user" relationships*

a1 = Address(email='u1')

a2 = Address(email='u2')

u1 = User(address\_collection=[a1, a2])

**assert** a1.user **is** u1

Above, given mostly complete User and Address mappings, the [ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey) which we defined on Address.user\_id allowed a bidirectional relationship pair Address.user and User.address\_collection to be generated on the mapped classes.

以上，给出大部分完整的User和Address映射，我们在Address.user\_id上定义的[ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey)允许在映射类上生成双向关系对Address.user和User.address\_collection。

Note that when subclassing [AutomapBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase" \o "sqlalchemy.ext.automap.AutomapBase), the [AutomapBase.prepare()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase.prepare" \o "sqlalchemy.ext.automap.AutomapBase.prepare) method is required; if not called, the classes we've declared are in an un-mapped state.

请注意，当对[AutomapBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase" \o "sqlalchemy.ext.automap.AutomapBase)进行子类化时，需要使用[AutomapBase.prepare()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase.prepare" \o "sqlalchemy.ext.automap.AutomapBase.prepare)方法; 如果没有被调用，我们已经声明的类处于未映射状态。

7.2.7 API Reference

sqlalchemy.ext.automap.**automap\_base**(*declarative\_base=None*, *\*\*kw*)

Produce a declarative automap base.

生成一个声明性的automap基础。

This function produces a new base class that is a product of the [AutomapBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase" \o "sqlalchemy.ext.automap.AutomapBase) class as well a declarative base produced by[declarative.declarative\_base()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.declarative_base" \o "sqlalchemy.ext.declarative.declarative_base).

该函数产生一个新的基类，它是[AutomapBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase" \o "sqlalchemy.ext.automap.AutomapBase)类的一个产品，以及由[declarative.declarative\_base()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.declarative_base" \o "sqlalchemy.ext.declarative.declarative_base)生成的声明性基类。

All parameters other than declarative\_base are keyword arguments that are passed directly to the [declarative.declarative\_base()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.declarative_base" \o "sqlalchemy.ext.declarative.declarative_base) function.

除declarative\_base之外的所有参数都是直接传递给[declarative.declarative\_base()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.declarative_base" \o "sqlalchemy.ext.declarative.declarative_base)函数的关键字参数。

|  |  |
| --- | --- |
| **Parameters:** | * ****declarative\_base**** – an existing class produced by [declarative.declarative\_base()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.declarative_base" \o "sqlalchemy.ext.declarative.declarative_base). When this is passed, the function no longer invokes [declarative.declarative\_base()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.declarative_base" \o "sqlalchemy.ext.declarative.declarative_base) itself, and all other keyword arguments are ignored. * ****\*\*kw**** – keyword arguments are passed along to [declarative.declarative\_base()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.declarative_base" \o "sqlalchemy.ext.declarative.declarative_base). |

*class*sqlalchemy.ext.automap.**AutomapBase**

Base class for an "automap" schema.

“automap”模式的基类。

The [AutomapBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase" \o "sqlalchemy.ext.automap.AutomapBase) class can be compared to the "declarative base" class that is produced by the [declarative.declarative\_base()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.declarative_base" \o "sqlalchemy.ext.declarative.declarative_base) function. In practice, the [AutomapBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase" \o "sqlalchemy.ext.automap.AutomapBase) class is always used as a mixin along with an actual declarative base.

[AutomapBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase" \o "sqlalchemy.ext.automap.AutomapBase)类可以与由[declarative.declarative\_base()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.declarative_base" \o "sqlalchemy.ext.declarative.declarative_base)函数生成的“声明性基础”类进行比较。 实际上，[AutomapBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase" \o "sqlalchemy.ext.automap.AutomapBase)类始终用作一个mixin以及一个实际的声明基础。

A new subclassable [AutomapBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase" \o "sqlalchemy.ext.automap.AutomapBase) is typically instantiated using the [automap\_base()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.automap_base" \o "sqlalchemy.ext.automap.automap_base) function.

通常使用[automap\_base()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.automap_base" \o "sqlalchemy.ext.automap.automap_base)函数来实例化一个新的可分类的[AutomapBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase" \o "sqlalchemy.ext.automap.AutomapBase)。

**See also**

[Automap](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html)

**classes***= None*

An instance of util.Properties containing classes.

包含类的util.Properties的一个实例。

This object behaves much like the .c collection on a table. Classes are present under the name they were given, e.g.:

此对象的行为与表上的.c集合非常相似。 课程以他们所提供的名义存在，例如：

Base = automap\_base()

Base.prepare(engine=some\_engine, reflect=**True**)

User, Address = Base.classes.User, Base.classes.Address

*classmethod***prepare**(*engine=None*, *reflect=False*, *schema=None*, *classname\_for\_table=<function classname\_for\_table>*, *collection\_class=<type 'list'>*, *name\_for\_scalar\_relationship=<function name\_for\_scalar\_relationship>*, *name\_for\_collection\_relationship=<function name\_for\_collection\_relationship>*, *generate\_relationship=<function generate\_relationship>*)

Extract mapped classes and relationships from the [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) and perform mappings.

从[MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData)中提取映射的类和关系并执行映射。

|  |  |
| --- | --- |
| **Parameters:** | * ****engine**** – an [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) or [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) with which to perform schema reflection, if specified. If the [AutomapBase.prepare.reflect](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase.prepare.params.reflect" \o "sqlalchemy.ext.automap.AutomapBase.prepare) argument is False, this object is not used.一个[Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine)或[Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection)，用于执行模式反射(如果指定)。 如果[AutomapBase.prepare.reflect](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase.prepare.params.reflect" \o "sqlalchemy.ext.automap.AutomapBase.prepare)参数为False，则不使用此对象。 * ****reflect**** – if True, the [MetaData.reflect()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData.reflect" \o "sqlalchemy.schema.MetaData.reflect) method is called on the [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) associated with this [AutomapBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase" \o "sqlalchemy.ext.automap.AutomapBase). The [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) passed via[AutomapBase.prepare.engine](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase.prepare.params.engine" \o "sqlalchemy.ext.automap.AutomapBase.prepare) will be used to perform the reflection if present; else, the [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) should already be bound to some engine else the operation will fail.如果为True，则在与此AutomapBase相关联的MetaData上调用MetaData.reflect()方法。 通过AutomapBase.prepare.engine传递的引擎将用于执行反射(如果存在) 否则，MetaData应该已经绑定到某些引擎，否则操作将失败。 * ****classname\_for\_table**** – callable function which will be used to produce new class names, given a table name. Defaults to[classname\_for\_table()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.classname_for_table" \o "sqlalchemy.ext.automap.classname_for_table).可用函数，用于生成新的类名，给定一个表名。 默认为classname\_for\_table()。 * ****name\_for\_scalar\_relationship**** – callable function which will be used to produce relationship names for scalar relationships. Defaults to [name\_for\_scalar\_relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.name_for_scalar_relationship" \o "sqlalchemy.ext.automap.name_for_scalar_relationship).可用函数，用于产生标量关系的关系名称。 默认为name\_for\_scalar\_relationship()。 * ****name\_for\_collection\_relationship**** – callable function which will be used to produce relationship names for collection-oriented relationships. Defaults to [name\_for\_collection\_relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.name_for_collection_relationship" \o "sqlalchemy.ext.automap.name_for_collection_relationship).可用函数，用于产生面向集合的关系的关系名称。 默认为name\_for\_collection\_relationship()。 * ****generate\_relationship**** – callable function which will be used to actually generate [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) and [backref()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.backref" \o "sqlalchemy.orm.backref) constructs. Defaults to [generate\_relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.generate_relationship" \o "sqlalchemy.ext.automap.generate_relationship).可调用函数将用于实际生成relationship()和backref()构造。 默认为generate\_relationship()。 * ****collection\_class**** – the Python collection class that will be used when a new [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) object is created that represents a collection. Defaults to list.将在创建代表集合的新[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)对象时使用的Python集合类。 默认为列表。 * ****schema –****When present in conjunction with the [AutomapBase.prepare.reflect](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase.prepare.params.reflect" \o "sqlalchemy.ext.automap.AutomapBase.prepare) flag, is passed to [MetaData.reflect()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData.reflect" \o "sqlalchemy.schema.MetaData.reflect) to indicate the primary schema where tables should be reflected from. When omitted, the default schema in use by the database connection is used.当与[AutomapBase.prepare.reflect](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase.prepare.params.reflect" \o "sqlalchemy.ext.automap.AutomapBase.prepare)标志一起使用时，将传递给[MetaData.reflect()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData.reflect" \o "sqlalchemy.schema.MetaData.reflect)以指示应从其中反映表的主模式。 当省略时，使用数据库连接使用的默认模式。   *New in version 1.1.* |

sqlalchemy.ext.automap.**classname\_for\_table**(*base*, *tablename*, *table*)

Return the class name that should be used, given the name of a table.

返回应该使用的类名，给定一个表的名称。

The default implementation is:

默认实现是：

**return** str(tablename)

Alternate implementations can be specified using the [AutomapBase.prepare.classname\_for\_table](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase.prepare.params.classname_for_table" \o "sqlalchemy.ext.automap.AutomapBase.prepare) parameter.

|  |  |
| --- | --- |
| **Parameters:** | * ****base**** – the [AutomapBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase" \o "sqlalchemy.ext.automap.AutomapBase) class doing the prepare. * ****tablename**** – string name of the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table). * ****table**** – the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object itself. |
| **Returns:** | a string class name.  **Note**  In Python 2, the string used for the class name ****must**** be a non-Unicode object, e.g. a str() object. The .name attribute of [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) is typically a Python unicode subclass, so the str() function should be applied to this name, after accounting for any non-ASCII characters.  在Python 2中，用于类名的字符串必须是非Unicode对象，例如一个str()对象。 [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)的.name属性通常是Python unicode子类，所以在计算任何非ASCII字符后，应该将str()函数应用于此名称。 |

sqlalchemy.ext.automap.**name\_for\_scalar\_relationship**(*base*, *local\_cls*, *referred\_cls*, *constraint*)

Return the attribute name that should be used to refer from one class to another, for a scalar object reference.

The default implementation is:

**return** referred\_cls.\_\_name\_\_.lower()

Alternate implementations can be specified using the [AutomapBase.prepare.name\_for\_scalar\_relationship](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase.prepare.params.name_for_scalar_relationship" \o "sqlalchemy.ext.automap.AutomapBase.prepare) parameter.

|  |  |
| --- | --- |
| **Parameters:** | * ****base**** – the [AutomapBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase" \o "sqlalchemy.ext.automap.AutomapBase) class doing the prepare. * ****local\_cls**** – the class to be mapped on the local side. * ****referred\_cls**** – the class to be mapped on the referring side. * ****constraint**** – the [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint) that is being inspected to produce this relationship. |

sqlalchemy.ext.automap.**name\_for\_collection\_relationship**(*base*, *local\_cls*, *referred\_cls*, *constraint*)

Return the attribute name that should be used to refer from one class to another, for a collection reference.

The default implementation is:

**return** referred\_cls.\_\_name\_\_.lower() + "\_collection"

Alternate implementations can be specified using the [AutomapBase.prepare.name\_for\_collection\_relationship](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase.prepare.params.name_for_collection_relationship" \o "sqlalchemy.ext.automap.AutomapBase.prepare) parameter.

|  |  |
| --- | --- |
| **Parameters:** | * ****base**** – the [AutomapBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase" \o "sqlalchemy.ext.automap.AutomapBase) class doing the prepare. * ****local\_cls**** – the class to be mapped on the local side. * ****referred\_cls**** – the class to be mapped on the referring side. * ****constraint**** – the [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint) that is being inspected to produce this relationship. |

sqlalchemy.ext.automap.**generate\_relationship**(*base*, *direction*, *return\_fn*, *attrname*, *local\_cls*, *referred\_cls*, *\*\*kw*)

Generate a [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) or [backref()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.backref" \o "sqlalchemy.orm.backref) on behalf of two mapped classes.

An alternate implementation of this function can be specified using the [AutomapBase.prepare.generate\_relationship](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase.prepare.params.generate_relationship" \o "sqlalchemy.ext.automap.AutomapBase.prepare) parameter.

The default implementation of this function is as follows:

**if** return\_fn **is** backref:

**return** return\_fn(attrname, \*\*kw)**elif** return\_fn **is** relationship:

**return** return\_fn(referred\_cls, \*\*kw)**else**:

**raise** TypeError("Unknown relationship function: *%s*" % return\_fn)

|  |  |
| --- | --- |
| **Parameters:** | * ****base**** – the [AutomapBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.AutomapBase" \o "sqlalchemy.ext.automap.AutomapBase) class doing the prepare. * ****direction**** – indicate the "direction" of the relationship; this will be one of [ONETOMANY](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.ONETOMANY" \o "sqlalchemy.orm.interfaces.ONETOMANY), [MANYTOONE](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MANYTOONE" \o "sqlalchemy.orm.interfaces.MANYTOONE), [MANYTOMANY](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MANYTOMANY" \o "sqlalchemy.orm.interfaces.MANYTOMANY). * ****return\_fn**** – the function that is used by default to create the relationship. This will be either [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) or [backref()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.backref" \o "sqlalchemy.orm.backref). The [backref()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.backref" \o "sqlalchemy.orm.backref) function's result will be used to produce a new [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) in a second step, so it is critical that user-defined implementations correctly differentiate between the two functions, if a custom relationship function is being used. * ****attrname**** – the attribute name to which this relationship is being assigned. If the value of [generate\_relationship.return\_fn](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.generate_relationship.params.return_fn" \o "sqlalchemy.ext.automap.generate_relationship) is the [backref()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.backref" \o "sqlalchemy.orm.backref)function, then this name is the name that is being assigned to the backref. * ****local\_cls**** – the "local" class to which this relationship or backref will be locally present. * ****referred\_cls**** – the "referred" class to which the relationship or backref refers to. * ****\*\*kw**** – all additional keyword arguments are passed along to the function. |
| **Returns:** | a [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) or [backref()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.backref" \o "sqlalchemy.orm.backref) construct, as dictated by the [generate\_relationship.return\_fn](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/automap.html" \l "sqlalchemy.ext.automap.generate_relationship.params.return_fn" \o "sqlalchemy.ext.automap.generate_relationship) parameter. |

## 7.3 Baked Queries

baked provides an alternative creational pattern for [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) objects, which allows for caching of the object's construction and string-compilation steps. This means that for a particular [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) building scenario that is used more than once, all of the Python function invocation involved in building the query from its initial construction up through generating a SQL string will only occur ****once****, rather than for each time that query is built up and executed.

baked为[Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) 对象提供了替代的创建模式，这允许缓存对象的构造和字符串编译步骤。 这意味着对于使用不止一次的特定[Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) 构建场景，通过生成SQL字符串构建查询所涉及的所有Python函数调用将仅发生一次，而不是每次查询都创建并执行。

The rationale for this system is to greatly reduce Python interpreter overhead for everything that occurs ****before the SQL is emitted****. The caching of the "baked" system does ****not**** in any way reduce SQL calls or cache the ****return results**** from the database. A technique that demonstrates the caching of the SQL calls and result sets themselves is available in [Dogpile Caching](http://docs.sqlalchemy.org/en/rel_1_1/orm/examples.html" \l "examples-caching).

该系统的基本原理是大大减少Python解释器开销，以便在发出SQL之前发生的所有事务。 “baked”系统的缓存不会以任何方式减少SQL调用或缓存数据库中的返回结果。 显示缓存SQL调用和结果集本身的技术在Dogpile Caching中可用。

*New in version 1.0.0.*

**Note**

The [sqlalchemy.ext.baked](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/baked.html" \l "module-sqlalchemy.ext.baked" \o "sqlalchemy.ext.baked) extension is ****not for beginners****. Using it correctly requires a good high level understanding of how SQLAlchemy, the database driver, and the backend database interact with each other. This extension presents a very specific kind of optimization that is not ordinarily needed. As noted above, it ****does not cache queries****, only the string formulation of the SQL itself.

[sqlalchemy.ext.baked](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/baked.html" \l "module-sqlalchemy.ext.baked" \o "sqlalchemy.ext.baked)扩展不适用于初学者。 正确使用它需要对SQLAlchemy，数据库驱动程序和后端数据库如何相互交互的高级了解。 此扩展提供了非常特殊的优化，通常不需要。 如上所述，它不缓存查询，只是SQL本身的字符串公式。

7.3.1 Synopsis摘要

Usage of the baked system starts by producing a so-called "bakery", which represents storage for a particular series of query objects:

baked 系统的使用开始于生产所谓的“bakery”，其代表特定系列查询对象的存储：

**from** **sqlalchemy.ext** **import** baked

bakery = baked.bakery()

The above "bakery" will store cached data in an LRU cache that defaults to 200 elements, noting that an ORM query will typically contain one entry for the ORM query as invoked, as well as one entry per database dialect for the SQL string.

上述“bakery”将缓存的数据存储在默认为200个元素的LRU缓存中，注意到ORM查询通常将包含一个ORM查询作为调用条目，以及SQL字符串每个数据库方言一个条目。

The bakery allows us to build up a [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) object by specifying its construction as a series of Python callables, which are typically lambdas. For succinct usage, it overrides the += operator so that a typical query build-up looks like the following:

bakery允许我们通过将其构造指定为一系列Python调用，通常是lambdas来构建一个[Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query)对象。 为了简洁的使用，它将覆盖+=运算符，因此典型的查询构建如下所示：

**from** **sqlalchemy** **import** bindparam

**def** search\_for\_user(session, username, email=**None**):

baked\_query = bakery(**lambda** session: session.query(User))

baked\_query += **lambda** q: q.filter(User.name == bindparam('username'))

baked\_query += **lambda** q: q.order\_by(User.id)

**if** email:

baked\_query += **lambda** q: q.filter(User.email == bindparam('email'))

result = baked\_query(session).params(username=username, email=email).all()

**return** result

Following are some observations about the above code:

以下是有关上述代码的一些意见：

1. The baked\_query object is an instance of [BakedQuery](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/baked.html" \l "sqlalchemy.ext.baked.BakedQuery" \o "sqlalchemy.ext.baked.BakedQuery). This object is essentially the "builder" for a real orm [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) object, but it is not itself the *actual* [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) object. baked\_query对象是[BakedQuery](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/baked.html" \l "sqlalchemy.ext.baked.BakedQuery" \o "sqlalchemy.ext.baked.BakedQuery)的一个实例。 该对象本质上是一个真正的orm [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query)对象的“构造器”，但它本身不是实际的[Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query)对象。
2. The actual [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) object is not built at all, until the very end of the function when [Result.all()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/baked.html" \l "sqlalchemy.ext.baked.Result.all" \o "sqlalchemy.ext.baked.Result.all) is called.实际的[Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query)对象根本就没有建立，直到调用[Result.all()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/baked.html" \l "sqlalchemy.ext.baked.Result.all" \o "sqlalchemy.ext.baked.Result.all)函数的结尾。
3. The steps that are added to the baked\_query object are all expressed as Python functions, typically lambdas. The first lambda given to the [bakery()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/baked.html" \l "sqlalchemy.ext.baked.bakery" \o "sqlalchemy.ext.baked.bakery)function receives a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) as its argument. The remaining lambdas each receive a [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) as their argument.添加到baked\_query对象的步骤都表示为Python函数，通常是lambdas。 给bakery()函数的第一个lambda接收一个[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)作为参数。 剩下的lambdas 每个都收到一个查询作为他们的参数。
4. In the above code, even though our application may call upon search\_for\_user() many times, and even though within each invocation we build up an entirely new [BakedQuery](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/baked.html" \l "sqlalchemy.ext.baked.BakedQuery" \o "sqlalchemy.ext.baked.BakedQuery) object, *all of the lambdas are only called once*. Each lambda is ****never**** called a second time for as long as this query is cached in the bakery.在上面的代码中，即使我们的应用程序可能会多次调用search\_for\_user()，即使在每个调用之内，我们构建了一个全新的[BakedQuery](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/baked.html" \l "sqlalchemy.ext.baked.BakedQuery" \o "sqlalchemy.ext.baked.BakedQuery)对象，所有的lambdas只被调用一次。 只要这个查询被缓存在面包店中，每个lambda从不被称为第二次。
5. The caching is achieved by storing references to the ****lambda objects themselves**** in order to formulate a cache key; that is, the fact that the Python interpreter assigns an in-Python identity to these functions is what determines how to identify the query on successive runs. For those invocations of search\_for\_user() where the email parameter is specified, the callable lambda q: q.filter(User.email == bindparam('email')) will be part of the cache key that's retrieved; when email is None, this callable is not part of the cache key.缓存是通过存储对lambda对象本身的引用来实现的，以便制定缓存密钥; 也就是说，Python解释器为这些函数分配一个Python标识是什么决定了如何在连续的运行中识别查询。 对于那些指定了email参数的search\_for\_user()的这些调用，可调用的lambda q: q.filter(User.email == bindparam('email'))将作为检索的缓存键的一部分; 当email 为None时，此可调用不是缓存密钥的一部分。
6. Because the lambdas are all called only once, it is essential that no variables which may change across calls are referenced ****within**** the lambdas; instead, assuming these are values to be bound into the SQL string, we use [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) to construct named parameters, where we apply their actual values later using [Result.params()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/baked.html" \l "sqlalchemy.ext.baked.Result.params" \o "sqlalchemy.ext.baked.Result.params).因为lambdas只被调用一次，所以必须在lambdas中引用可能在调用中变化的变量; 相反，假设这些值被绑定到SQL字符串中，我们使用[bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam)构造命名参数，我们稍后使用[Result.params()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/baked.html" \l "sqlalchemy.ext.baked.Result.params" \o "sqlalchemy.ext.baked.Result.params)应用它们的实际值。

7.3.2 Performance

The baked query probably looks a little odd, a little bit awkward and a little bit verbose. However, the savings in Python performance for a query which is invoked lots of times in an application are very dramatic. The example suite short\_selects demonstrated in [Performance](http://docs.sqlalchemy.org/en/rel_1_1/orm/examples.html" \l "examples-performance) illustrates a comparison of queries which each return only one row, such as the following regular query:

baked的查询可能看起来有点奇怪，有点尴尬，有点冗长。 然而，在应用程序中调用了很多次的查询的Python性能的节省是非常显着的。 Performance中演示的示例套件short\_selects说明了每个仅返回一行的查询的比较，例如以下常规查询：

session = Session(bind=engine)

**for** id\_ **in** random.sample(ids, n):

session.query(Customer).filter(Customer.id == id\_).one()

compared to the equivalent "baked" query:

相当于相当于“baked”的查询：

bakery = baked.bakery()

s = Session(bind=engine)

**for** id\_ **in** random.sample(ids, n):

q = bakery(**lambda** s: s.query(Customer))

q += **lambda** q: q.filter(Customer.id == bindparam('id'))

q(s).params(id=id\_).one()

The difference in Python function call count for an iteration of 10000 calls to each block are:

每个块迭代10000次调用的Python函数调用计数的区别是：

test\_baked\_query : test a baked query of the full entity.

(10000 iterations); total fn calls 1951294

test\_orm\_query : test a straight ORM query of the full entity.

(10000 iterations); total fn calls 7900535

In terms of number of seconds on a powerful laptop, this comes out as:

在强大的笔记本电脑上的秒数，这出来是：

test\_baked\_query : test a baked query of the full entity.

(10000 iterations); total time 2.174126 sec

test\_orm\_query : test a straight ORM query of the full entity.

(10000 iterations); total time 7.958516 sec

Note that this test very intentionally features queries that only return one row. For queries that return many rows, the performance advantage of the baked query will have less and less of an impact, proportional to the time spent fetching rows. It is critical to keep in mind that the ****baked query feature only applies to building the query itself, not the fetching of results****. Using the baked feature is by no means a guarantee to a much faster application; it is only a potentially useful feature for those applications that have been measured as being impacted by this particular form of overhead.

请注意，此测试非常有意地提供仅返回一行的查询。 对于返回许多行的查询，****baked**** 查询的表现的性能优势将越来越少，与获取行花费的时间成比例。 请务必记住，****baked**** 的查询功能仅适用于构建查询本身，而不是提取结果。 使用****baked**** 功能绝对不能保证更快的应用程序; 那些已被测量为受特定形式的开销影响的应用程序只是一个潜在有用的功能。

**Measure twice, cut once**

For background on how to profile a SQLAlchemy application, please see the section [Performance](http://docs.sqlalchemy.org/en/rel_1_1/faq/performance.html" \l "faq-performance). It is essential that performance measurement techniques are used when attempting to improve the performance of an application.

有关如何配置SQLAlchemy应用程序的背景信息，请参阅“性能”一节。 当尝试提高应用程序的性能时，必须使用性能测量技术。

7.3.3 Rationale

The "lambda" approach above is a superset of what would be a more traditional "parameterized" approach. Suppose we wished to build a simple system where we build a [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) just once, then store it in a dictionary for re-use. This is possible right now by just building up the query, and removing its [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) by callingmy\_cached\_query = query.with\_session(None):

上面的“lambda”方法是一个更传统的“参数化”方法的超集。 假设我们希望构建一个简单的系统，我们构建一个[Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query)只需一次，然后将其存储在一个字典中以便重新使用。 现在可以通过建立查询，并通过调用my\_cached\_query = query.with\_session(None)来删除其[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)：

my\_simple\_cache = {}

**def** lookup(session, id\_argument):

**if** "my\_key" **not** **in** my\_simple\_cache:

query = session.query(Model).filter(Model.id == bindparam('id'))

my\_simple\_cache["my\_key"] = query.with\_session(**None**)

**else**:

query = my\_simple\_cache["my\_key"].with\_session(session)

**return** query.params(id=id\_argument).all()

The above approach gets us a very minimal performance benefit. By re-using a [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query), we save on the Python work within the session.query(Model)constructor as well as calling upon filter(Model.id == bindparam('id')), which will skip for us the building up of the Core expression as well as sending it to [Query.filter()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.filter" \o "sqlalchemy.orm.query.Query.filter). However, the approach still regenerates the full [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select) object every time when [Query.all()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.all" \o "sqlalchemy.orm.query.Query.all) is called and additionally this brand new [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select) is sent off to the string compilation step every time, which for a simple case like the above is probably about 70% of the overhead.

上述方法使我们获得了极小的性能优势。通过重新使用[Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query)，我们在session.query(Model)构造函数中保存Python工作，并调用过滤器filter(Model.id == bindparam('id'))，这将跳过我们构建的Core表达式，并将其发送到[Query.filter()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.filter" \o "sqlalchemy.orm.query.Query.filter)。但是，每次调用[Query.all()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.all" \o "sqlalchemy.orm.query.Query.all)时，该方法仍然会重新生成完整的[Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select)对象，此外，每次都会将此全新[Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select)选择发送到字符串编译步骤，对于上述类似的简单情况，大概约为70％的开销。

To reduce the additional overhead, we need some more specialized logic, some way to memoize the construction of the select object and the construction of the SQL. There is an example of this on the wiki in the section [BakedQuery](https://bitbucket.org/zzzeek/sqlalchemy/wiki/UsageRecipes/BakedQuery), a precursor to this feature, however in that system, we aren't caching the *construction* of the query. In order to remove all the overhead, we need to cache both the construction of the query as well as the SQL compilation. Let's assume we adapted the recipe in this way and made ourselves a method .bake() that pre-compiles the SQL for the query, producing a new object that can be invoked with minimal overhead. Our example becomes:

为了减少额外的开销，我们需要一些更专业的逻辑，一些方法来记录select对象的构造和SQL的构造。在BakedQuery部分的一个维基中有一个例子，这是该功能的前身，然而在该系统中，我们并没有缓存查询的构造。为了消除所有开销，我们需要缓存查询的构造以及SQL编译。让我们假设我们以这种方式调整了配方，并使自己成为一种预编译查询的SQL的方法.bake()，生成一个可以以最小开销调用的新对象。我们的例子变成：

my\_simple\_cache = {}

**def** lookup(session, id\_argument):

**if** "my\_key" **not** **in** my\_simple\_cache:

query = session.query(Model).filter(Model.id == bindparam('id'))

my\_simple\_cache["my\_key"] = query.with\_session(**None**).bake()

**else**:

query = my\_simple\_cache["my\_key"].with\_session(session)

**return** query.params(id=id\_argument).all()

Above, we've fixed the performance situation, but we still have this string cache key to deal with.

We can use the "bakery" approach to re-frame the above in a way that looks less unusual than the "building up lambdas" approach, and more like a simple improvement upon the simple "reuse a query" approach:

以上，我们已经修复了性能情况，但是我们仍然有这个字符串缓存键来处理。

我们可以使用“bakery”方法重新构建上述方法，其看起来比“建立lambdas”方法更不寻常，更像简单的“重用查询”方法的简单改进：

bakery = baked.bakery()

**def** lookup(session, id\_argument):

**def** create\_model\_query(session):

**return** session.query(Model).filter(Model.id == bindparam('id'))

parameterized\_query = bakery.bake(create\_model\_query)

**return** parameterized\_query(session).params(id=id\_argument).all()

Above, we use the "baked" system in a manner that is very similar to the simplistic "cache a query" system. However, it uses two fewer lines of code, does not need to manufacture a cache key of "my\_key", and also includes the same feature as our custom "bake" function that caches 100% of the Python invocation work from the constructor of the query, to the filter call, to the production of the [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select) object, to the string compilation step.

以上，我们以与简单的“缓存查询”系统非常相似的方式使用“baked”系统。 然而，它使用两行代码，不需要制造“my\_key”的缓存键，并且还包括与我们的自定义“baked”功能相同的功能，该功能从100％的Python调用工作缓存 查询，到过滤器调用，生成Select对象，到字符串编译步骤。

From the above, if we ask ourselves, "what if lookup needs to make conditional decisions as to the structure of the query?", this is where hopefully it becomes apparent why "baked" is the way it is. Instead of a parameterized query building off from exactly one function (which is how we thought baked might work originally), we can build it from *any number* of functions. Consider our naive example, if we needed to have an additional clause in our query on a conditional basis:

如上所述，如果我们问自己，“如果查找需要对查询的结构作出有条件的决定呢”，那么这是希望明白为什么“baked”是这样的。 我们可以从任何数量的函数中构建一个完全一个函数的参数化查询(这是我们以为baked 可能最初工作的)。 考虑我们的天真的例子，如果我们需要在条件的基础上在我们的查询中添加一个附加条款：

my\_simple\_cache = {}

**def** lookup(session, id\_argument, include\_frobnizzle=**False**):

**if** include\_frobnizzle:

cache\_key = "my\_key\_with\_frobnizzle"

**else**:

cache\_key = "my\_key\_without\_frobnizzle"

**if** cache\_key **not** **in** my\_simple\_cache:

query = session.query(Model).filter(Model.id == bindparam('id'))

**if** include\_frobnizzle:

query = query.filter(Model.frobnizzle == **True**)

my\_simple\_cache[cache\_key] = query.with\_session(**None**).bake()

**else**:

query = my\_simple\_cache[cache\_key].with\_session(session)

**return** query.params(id=id\_argument).all()

Our "simple" parameterized system must now be tasked with generating cache keys which take into account whether or not the "include\_frobnizzle" flag was passed, as the presence of this flag means that the generated SQL would be entirely different. It should be apparent that as the complexity of query building goes up, the task of caching these queries becomes burdensome very quickly. We can convert the above example into a direct use of "bakery" as follows:

现在，我们的“简单”参数化系统现在需要生成高速缓存键，这些缓存键考虑到是否传递了“include\_frobnizzle”标志，因为这个标志的存在意味着生成的SQL将完全不同。 显而易见的是，随着查询建立的复杂性上升，缓存这些查询的任务变得非常快。 我们可以将上述示例转换为直接使用“bakery”，如下所示：

bakery = baked.bakery()

**def** lookup(session, id\_argument, include\_frobnizzle=**False**):

**def** create\_model\_query(session):

**return** session.query(Model).filter(Model.id == bindparam('id'))

parameterized\_query = bakery.bake(create\_model\_query)

**if** include\_frobnizzle:

**def** include\_frobnizzle\_in\_query(query):

**return** query.filter(Model.frobnizzle == **True**)

parameterized\_query = parameterized\_query.with\_criteria(

include\_frobnizzle\_in\_query)

**return** parameterized\_query(session).params(id=id\_argument).all()

Above, we again cache not just the query object but all the work it needs to do in order to generate SQL. We also no longer need to deal with making sure we generate a cache key that accurately takes into account all of the structural modifications we've made; this is now handled automatically and without the chance of mistakes.

以上，我们再次不仅要查询查询对象，而且还要缓存生成SQL需要做的所有工作。 我们也不再需要处理，确保我们生成一个准确地考虑到我们所做的所有结构修改的缓存密钥; 这是现在自动处理，没有错误的机会。

This code sample is a few lines shorter than the naive example, removes the need to deal with cache keys, and has the vast performance benefits of the full so-called "baked" feature. But still a little verbose! Hence we take methods like [BakedQuery.add\_criteria()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/baked.html" \l "sqlalchemy.ext.baked.BakedQuery.add_criteria" \o "sqlalchemy.ext.baked.BakedQuery.add_criteria) and [BakedQuery.with\_criteria()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/baked.html" \l "sqlalchemy.ext.baked.BakedQuery.with_criteria" \o "sqlalchemy.ext.baked.BakedQuery.with_criteria) and shorten them into operators, and encourage (though certainly not require!) using simple lambdas, only as a means to

reduce verbosity:

这个代码示例比天真的例子短了几行，消除了处理缓存键的需要，并且具有完全所谓的“烘烤”功能的巨大的性能优势。 但还是有点冗长！ 因此，我们采取像[BakedQuery.add\_criteria()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/baked.html" \l "sqlalchemy.ext.baked.BakedQuery.add_criteria" \o "sqlalchemy.ext.baked.BakedQuery.add_criteria)和[BakedQuery.with\_criteria()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/baked.html" \l "sqlalchemy.ext.baked.BakedQuery.with_criteria" \o "sqlalchemy.ext.baked.BakedQuery.with_criteria)这样的方法，并将它们缩短为运算符，并鼓励(尽管当然不需要！)使用简单的lambdas，只是减少冗长度的一种手段：

bakery = baked.bakery()

**def** lookup(session, id\_argument, include\_frobnizzle=**False**):

parameterized\_query = bakery.bake(

**lambda** s: s.query(Model).filter(Model.id == bindparam('id'))

)

**if** include\_frobnizzle:

parameterized\_query += **lambda** q: q.filter(Model.frobnizzle == **True**)

**return** parameterized\_query(session).params(id=id\_argument).all()

Where above, the approach is simpler to implement and much more similar in code flow to what a non-cached querying function would look like, hence making code easier to port.

在上面，该方法在代码流中更容易实现，并且与非缓存的查询功能类似，因此使代码更易于端口。

The above description is essentially a summary of the design process used to arrive at the current "baked" approach. Starting from the "normal" approaches, the additional issues of cache key construction and management, removal of all redundant Python execution, and queries built up with conditionals needed to be addressed, leading to the final approach.

以上描述本质上是用于达到当前“baked”方法的设计过程的总结。 从“正常”方法开始，缓存关键字构建和管理的附加问题，所有冗余Python执行的删除以及使用条件化建立的查询都需要解决，从而导致最终的方法。

7.3.4 Lazy Loading Integration

The baked query can be integrated with SQLAlchemy's lazy loader feature transparently. A future release of SQLAlchemy may enable this by default, as its use within lazy loading is completely transparent. For now, to enable baked lazyloading for all lazyloaders systemwide, call upon the [bake\_lazy\_loaders()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/baked.html" \l "sqlalchemy.ext.baked.bake_lazy_loaders" \o "sqlalchemy.ext.baked.bake_lazy_loaders) function. This will impact all relationships that use the lazy='select' strategy as well as all use of the [lazyload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.lazyload" \o "sqlalchemy.orm.lazyload) per-query strategy.

baked 查询可以透明地与SQLAlchemy的延迟加载器功能集成。 SQLAlchemy的未来版本可能默认启用，因为它在延迟加载中的使用是完全透明的。 现在，为了在全系统范围内为所有的懒惰加载器启用烘焙懒惰，请调用[bake\_lazy\_loaders()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/baked.html" \l "sqlalchemy.ext.baked.bake_lazy_loaders" \o "sqlalchemy.ext.baked.bake_lazy_loaders)函数。 这将影响使用lazy='select'策略的所有关系以及所有使用[lazyload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_relationships.html" \l "sqlalchemy.orm.lazyload" \o "sqlalchemy.orm.lazyload)每个查询策略。

"Baked" lazy loading may be enabled on a per-[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) basis using the baked\_select loader strategy:

可以使用baked\_select加载程序策略在每个[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)基础上启用“baked”延迟加载：

**class** **MyClass**(Base):

*# ...*

widgets = relationship("Widget", lazy="baked\_select")

The baked\_select strategy is available once any part of the application has imported the sqlalchemy.ext.baked module. The "bakery" used by this feature is local to the mapper for MyClass.

一旦应用程序的任何部分导入了sqlalchemy.ext.baked模块，就可以使用baked\_select策略。 此功能使用的“面包店”对于MyClass的映射程序是本地的。

For per-query use, the [baked\_lazyload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/baked.html" \l "sqlalchemy.ext.baked.baked_lazyload" \o "sqlalchemy.ext.baked.baked_lazyload) strategy may be used, which works like any other loader option.

对于每个查询使用，可以使用[baked\_lazyload()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/baked.html" \l "sqlalchemy.ext.baked.baked_lazyload" \o "sqlalchemy.ext.baked.baked_lazyload)策略，其工作方式与任何其他加载程序选项一样。

### Opting out with the bake\_queries flag

The [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) construct includes a flag [relationship.bake\_queries](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.bake_queries" \o "sqlalchemy.orm.relationship) which when set to False will cause that relationship to opt out of the baked query system, when the application-wide [bake\_lazy\_loaders()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/baked.html" \l "sqlalchemy.ext.baked.bake_lazy_loaders" \o "sqlalchemy.ext.baked.bake_lazy_loaders) function has been called to enable baked query loaders by default.

[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)构造包含一个标志[relationship.bake\_queries](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.bake_queries" \o "sqlalchemy.orm.relationship)，当设置为False时，当该应用程序范围的[bake\_lazy\_loaders()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/baked.html" \l "sqlalchemy.ext.baked.bake_lazy_loaders" \o "sqlalchemy.ext.baked.bake_lazy_loaders)函数被调用以默认启用烘焙查询加载程序时，它将导致该关系退出烘焙查询系统。

7.3.5 API Documentation

sqlalchemy.ext.baked.**bakery**(*cls*, *size=200*)

Construct a new bakery.

构建新的bakery。

*class*sqlalchemy.ext.baked.**BakedQuery**(*bakery*, *initial\_fn*, *args=()*)

A builder object for [query.Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) objects.

用于query.Query对象的构建器对象。

**add\_criteria**(*fn*, *\*args*)

Add a criteria function to this [BakedQuery](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/baked.html" \l "sqlalchemy.ext.baked.BakedQuery" \o "sqlalchemy.ext.baked.BakedQuery).

This is equivalent to using the += operator to modify a [BakedQuery](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/baked.html" \l "sqlalchemy.ext.baked.BakedQuery" \o "sqlalchemy.ext.baked.BakedQuery) in-place.

为此BakedQuery添加条件函数。

这相当于使用+ =运算符就地修改BakedQuery。

*classmethod***bakery**(*size=200*)

Construct a new bakery.

**for\_session**(*session*)

Return a [Result](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/baked.html" \l "sqlalchemy.ext.baked.Result" \o "sqlalchemy.ext.baked.Result) object for this [BakedQuery](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/baked.html" \l "sqlalchemy.ext.baked.BakedQuery" \o "sqlalchemy.ext.baked.BakedQuery).

返回此[BakedQuery](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/baked.html" \l "sqlalchemy.ext.baked.BakedQuery" \o "sqlalchemy.ext.baked.BakedQuery)的[Result](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/baked.html" \l "sqlalchemy.ext.baked.Result" \o "sqlalchemy.ext.baked.Result)对象。

This is equivalent to calling the [BakedQuery](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/baked.html" \l "sqlalchemy.ext.baked.BakedQuery" \o "sqlalchemy.ext.baked.BakedQuery) as a Python callable, e.g. result = my\_baked\_query(session).

这相当于将[BakedQuery](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/baked.html" \l "sqlalchemy.ext.baked.BakedQuery" \o "sqlalchemy.ext.baked.BakedQuery)称为Python可调用，例如result = my\_baked\_query(session)。

**spoil**(*full=False*)

Cancel any query caching that will occur on this BakedQuery object.

取消此BakedQuery对象上发生的任何查询缓存。

The BakedQuery can continue to be used normally, however additional creational functions will not be cached; they will be called on every invocation.

BakedQuery可以继续正常使用，但是不会缓存附加的创建功能; 每次调用都会被调用。

This is to support the case where a particular step in constructing a baked query disqualifies the query from being cacheable, such as a variant that relies upon some uncacheable value.

这是为了支持构建baked 查询的特定步骤取消对可查询的查询资格的情况，例如依赖于某些不可缓存值的变体。

|  |  |
| --- | --- |
| **Parameters:** | ****full**** – if False, only functions added to this [BakedQuery](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/baked.html" \l "sqlalchemy.ext.baked.BakedQuery" \o "sqlalchemy.ext.baked.BakedQuery) object subsequent to the spoil step will be non-cached; the state of the [BakedQuery](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/baked.html" \l "sqlalchemy.ext.baked.BakedQuery" \o "sqlalchemy.ext.baked.BakedQuery) up until this point will be pulled from the cache. If True, then the entire [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) object is built from scratch each time, with all creational functions being called on each invocation.  如果为False，则在破坏步骤之后只添加到此[BakedQuery](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/baked.html" \l "sqlalchemy.ext.baked.BakedQuery" \o "sqlalchemy.ext.baked.BakedQuery)对象的函数将不被缓存; [BakedQuery](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/baked.html" \l "sqlalchemy.ext.baked.BakedQuery" \o "sqlalchemy.ext.baked.BakedQuery)的状态直到这一点将从缓存中拉出。 如果为True，则每次都从头开始构建整个[Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query)对象，每次调用都调用所有创建函数。 |

**with\_criteria**(*fn*, *\*args*)

Add a criteria function to a [BakedQuery](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/baked.html" \l "sqlalchemy.ext.baked.BakedQuery" \o "sqlalchemy.ext.baked.BakedQuery) cloned from this one.

This is equivalent to using the + operator to produce a new [BakedQuery](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/baked.html" \l "sqlalchemy.ext.baked.BakedQuery" \o "sqlalchemy.ext.baked.BakedQuery) with modifications.

*class*sqlalchemy.ext.baked.**Result**(*bq*, *session*)

Invokes a [BakedQuery](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/baked.html" \l "sqlalchemy.ext.baked.BakedQuery" \o "sqlalchemy.ext.baked.BakedQuery) against a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session).

The [Result](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/baked.html" \l "sqlalchemy.ext.baked.Result" \o "sqlalchemy.ext.baked.Result) object is where the actual [query.Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) object gets created, or retrieved from the cache, against a target [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session), and is then invoked for results.

**all**()

Return all rows.

Equivalent to [Query.all()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.all" \o "sqlalchemy.orm.query.Query.all).

**count**()

return the 'count'.

Equivalent to [Query.count()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.count" \o "sqlalchemy.orm.query.Query.count).

Note this uses a subquery to ensure an accurate count regardless of the structure of the original statement.

*New in version 1.1.6.*

**first**()

Return the first row.

Equivalent to [Query.first()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.first" \o "sqlalchemy.orm.query.Query.first).

**get**(*ident*)

Retrieve an object based on identity.

Equivalent to [Query.get()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.get" \o "sqlalchemy.orm.query.Query.get).

**one**()

Return exactly one result or raise an exception.

Equivalent to [Query.one()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.one" \o "sqlalchemy.orm.query.Query.one).

**one\_or\_none**()

Return one or zero results, or raise an exception for multiple rows.

Equivalent to [Query.one\_or\_none()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.one_or_none" \o "sqlalchemy.orm.query.Query.one_or_none).

*New in version 1.0.9.*

**params**(*\*args*, *\*\*kw*)

Specify parameters to be replaced into the string SQL statement.

**scalar**()

Return the first element of the first result or None if no rows present. If multiple rows are returned, raises MultipleResultsFound.

Equivalent to [Query.scalar()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.scalar" \o "sqlalchemy.orm.query.Query.scalar).

*New in version 1.1.6.*

sqlalchemy.ext.baked.**bake\_lazy\_loaders**()

Enable the use of baked queries for all lazyloaders systemwide.

This operation should be safe for all lazy loaders, and will reduce Python overhead for these operations.

sqlalchemy.ext.baked.**unbake\_lazy\_loaders**()

Disable the use of baked queries for all lazyloaders systemwide.

This operation reverts the changes produced by [bake\_lazy\_loaders()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/baked.html" \l "sqlalchemy.ext.baked.bake_lazy_loaders" \o "sqlalchemy.ext.baked.bake_lazy_loaders).

sqlalchemy.ext.baked.**baked\_lazyload**(*\*keys*)

Indicate that the given attribute should be loaded using "lazy" loading with a "baked" query used in the load.

指示使用加载中使用的“baked”查询“懒”加载给定属性。

sqlalchemy.ext.baked.**baked\_lazyload\_all**(*\*keys*)

Produce a standalone "all" option for orm.baked\_lazyload().

为orm.baked\_lazyload()生成独立的“全部”选项。

*Deprecated since version 0.9.0:*The "\_all()" style is replaced by method chaining, e.g.:

session.query(MyClass).options(

baked\_lazyload("someattribute").baked\_lazyload("anotherattribute"))

## 7.4 Declarative

The Declarative system is the typically used system provided by the SQLAlchemy ORM in order to define classes mapped to relational database tables. However, as noted in [Classical Mappings](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_styles.html" \l "classical-mapping), Declarative is in fact a series of extensions that ride on top of the SQLAlchemy [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper) construct.

While the documentation typically refers to Declarative for most examples, the following sections will provide detailed information on how the Declarative API interacts with the basic [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper) and Core [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) systems, as well as how sophisticated patterns can be built using systems such as mixins.

Declarative系统是由SQLAlchemy ORM提供的通常使用的系统，用于定义映射到关系数据库表的类。 然而，正如经典映射中所指出的那样，Declarative实际上是一系列扩展，它们搭载在SQLAlchemy mapper()结构之上。

虽然文档通常引用大多数示例的Declarative，以下部分将提供关于Declarative API如何与基本mapper()和Core Table系统交互的详细信息，以及可以使用诸如mixins之类的系统构建复杂的模式。

# 7.4.1 Basic Use

SQLAlchemy object-relational configuration involves the combination of [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table), [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper), and class objects to define a mapped class. [declarative](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "module-sqlalchemy.ext.declarative" \o "sqlalchemy.ext.declarative) allows all three to be expressed at once within the class declaration. As much as possible, regular SQLAlchemy schema and ORM constructs are used directly, so that configuration between "classical" ORM usage and declarative remain highly similar.

As a simple example:

SQLAlchemy对象关系配置涉及到Table，mapper()和类对象的组合来定义一个映射类。 声明式允许在类声明中立即表达所有三个。 尽可能地使用常规的SQLAlchemy架构和ORM构造，因此“经典”ORM使用和声明性之间的配置仍然非常相似。

作为一个简单的例子：

**from** **sqlalchemy** **import** Column, Integer, String

**from** **sqlalchemy.ext.declarative** **import** declarative\_base

Base = declarative\_base()

**class** **SomeClass**(Base):

\_\_tablename\_\_ = 'some\_table'

id = Column(Integer, primary\_key=**True**)

name = Column(String(50))

Above, the declarative\_base() callable returns a new base class from which all mapped classes should inherit. When the class definition is completed, a new [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) and [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper) will have been generated.

The resulting table and mapper are accessible via \_\_table\_\_ and \_\_mapper\_\_ attributes on the SomeClass class:

*# access the mapped Table*

SomeClass.\_\_table\_\_

*# access the Mapper*

SomeClass.\_\_mapper\_\_

#### 7.4.1.1 Defining Attributes

In the previous example, the [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects are automatically named with the name of the attribute to which they are assigned.

在上一个示例中，Column对象将自动使用分配给它们的属性的名称命名。

To name columns explicitly with a name distinct from their mapped attribute, just give the column a name. Below, column "some\_table\_id" is mapped to the "id" attribute of SomeClass, but in SQL will be represented as "some\_table\_id":

要使用与其映射属性不同的名称显式指定列，只需将列命名。 下面列“some\_table\_id”映射到SomeClass的“id”属性，但在SQL中将被表示为“some\_table\_id”：

**class** **SomeClass**(Base):

\_\_tablename\_\_ = 'some\_table'

id = Column("some\_table\_id", Integer, primary\_key=**True**)

Attributes may be added to the class after its construction, and they will be added to the underlying [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) and [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper) definitions as appropriate:

SomeClass.data = Column('data', Unicode)SomeClass.related = relationship(RelatedInfo)

Classes which are constructed using declarative can interact freely with classes that are mapped explicitly with [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper).

It is recommended, though not required, that all tables share the same underlying [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) object, so that string-configured [ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey) references can be resolved without issue.

7.4.1.2 Accessing the MetaData

The declarative\_base() base class contains a [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) object where newly defined [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects are collected. This object is intended to be accessed directly for [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData)-specific operations. Such as, to issue CREATE statements for all tables:

engine = create\_engine('sqlite://')Base.metadata.create\_all(engine)

declarative\_base() can also receive a pre-existing [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) object, which allows a declarative setup to be associated with an already existing traditional collection of [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects:

mymetadata = MetaData()

Base = declarative\_base(metadata=mymetadata)

#### 7.4.1.3 Class Constructor

As a convenience feature, the declarative\_base() sets a default constructor on classes which takes keyword arguments, and assigns them to the named attributes:

e = Engineer(primary\_language='python')

#### 7.4.1.4 Mapper Configuration

Declarative makes use of the [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper) function internally when it creates the mapping to the declared table. The options for [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper) are passed directly through via the \_\_mapper\_args\_\_ class attribute. As always, arguments which reference locally mapped columns can reference them directly from within the class declaration:

**from** **datetime** **import** datetime

**class** **Widget**(Base):

\_\_tablename\_\_ = 'widgets'

id = Column(Integer, primary\_key=**True**)

timestamp = Column(DateTime, nullable=**False**)

\_\_mapper\_args\_\_ = {

'version\_id\_col': timestamp,

'version\_id\_generator': **lambda** v:datetime.now()

}

#### 7.4.1.5 Defining SQL Expressions

See [SQL Expressions as Mapped Attributes](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapped_sql_expr.html" \l "mapper-sql-expressions) for examples on declaratively mapping attributes to SQL expressions.

# 7.4.2 Configuring Relationships

Relationships to other classes are done in the usual way, with the added feature that the class specified to [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) may be a string name. The "class registry" associated with Base is used at mapper compilation time to resolve the name into the actual class object, which is expected to have been defined once the mapper configuration is used:

与其他类的关系以通常的方式完成，增加的功能是指定为relationship()的类可以是字符串名称。 与Base关联的“类注册表”在映射程序编译时使用，将名称解析为实际的类对象，一旦使用映射器配置，该对象将被定义;

**class** **User**(Base):

\_\_tablename\_\_ = 'users'

id = Column(Integer, primary\_key=**True**)

name = Column(String(50))

addresses = relationship("Address", backref="user")

**class** **Address**(Base):

\_\_tablename\_\_ = 'addresses'

id = Column(Integer, primary\_key=**True**)

email = Column(String(50))

user\_id = Column(Integer, ForeignKey('users.id'))

Column constructs, since they are just that, are immediately usable, as below where we define a primary join condition on the Address class using them:

列结构，因为它们只是这些，可立即使用，如下所示，我们使用它们在Address类定义一个主连接条件：

**class** **Address**(Base):

\_\_tablename\_\_ = 'addresses'

id = Column(Integer, primary\_key=**True**)

email = Column(String(50))

user\_id = Column(Integer, ForeignKey('users.id'))

user = relationship(User, primaryjoin=user\_id == User.id)

In addition to the main argument for [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship), other arguments which depend upon the columns present on an as-yet undefined class may also be specified as strings. These strings are evaluated as Python expressions. The full namespace available within this evaluation includes all classes mapped for this declarative base, as well as the contents of the sqlalchemy package, including expression functions like [desc()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.desc" \o "sqlalchemy.sql.expression.desc) and [func](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.func" \o "sqlalchemy.sql.expression.func):

除了[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)的主要参数外，依赖于尚未定义类的列的其他参数也可以被指定为字符串。 这些字符串被评估为Python表达式。 此评估中可用的完整命名空间包括为此声明性基础映射的所有类，以及sqlalchemy软件包的内容，包括表达式函数，如[desc()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.desc" \o "sqlalchemy.sql.expression.desc)和[func](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.func" \o "sqlalchemy.sql.expression.func)：

**class** **User**(Base):

*# ....*

addresses = relationship("Address",

order\_by="desc(Address.email)",

primaryjoin="Address.user\_id==User.id")

For the case where more than one module contains a class of the same name, string class names can also be specified as module-qualified paths within any of these string expressions:

对于多个模块包含相同名称的类的情况，字符串类名称也可以指定为任何这些字符串表达式中的模块限定路径：

**class** **User**(Base):

*# ....*

addresses = relationship("myapp.model.address.Address",

order\_by="desc(myapp.model.address.Address.email)",

primaryjoin="myapp.model.address.Address.user\_id=="

"myapp.model.user.User.id")

The qualified path can be any partial path that removes ambiguity between the names. For example, to disambiguate between myapp.model.address.Address and myapp.model.lookup.Address, we can specify address.Address or lookup.Address:

合格路径可以是消除名称之间的歧义的任何部分路径。 例如，要消除myapp.model.address.Address和myapp.model.lookup.Address之间的歧义，我们可以指定address.Address或lookup.Address：

**class** **User**(Base):

*# ....*

addresses = relationship("address.Address",

order\_by="desc(address.Address.email)",

primaryjoin="address.Address.user\_id=="

"User.id")

*New in version 0.8:*module-qualified paths can be used when specifying string arguments with Declarative, in order to specify specific modules.

在版本0.8中新增：使用声明式指定字符串参数时可以使用模块限定路径，以指定特定的模块。

Two alternatives also exist to using string-based attributes. A lambda can also be used, which will be evaluated after all mappers have been configured:

使用基于字符串的属性也有两种选择。 也可以使用lambda，在配置完所有映射器之后，它将被评估：

**class** **User**(Base):

*# ...*

addresses = relationship(**lambda**: Address,

order\_by=**lambda**: desc(Address.email),

primaryjoin=**lambda**: Address.user\_id==User.id)

Or, the relationship can be added to the class explicitly after the classes are available:

User.addresses = relationship(Address,

primaryjoin=Address.user\_id==User.id)

#### 7.4.2.1 Configuring Many-to-Many Relationships

Many-to-many relationships are also declared in the same way with declarative as with traditional mappings. The secondary argument to [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) is as usual passed a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object, which is typically declared in the traditional way. The [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) usually shares the [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) object used by the declarative base:

多对多关系也以与传统映射的声明相同的方式声明。 关系()的二次参数是一样的传递一个Table对象，它通常以传统方式声明。 表通常共享声明库使用的MetaData对象：

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)

Like other [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) arguments, a string is accepted as well, passing the string name of the table as defined in the Base.metadata.tables collection:

像其他的relationship()参数一样，也接受一个字符串，传递Base.metadata.tables集合中定义的表的字符串名称：

**class** **Author**(Base):

\_\_tablename\_\_ = 'authors'

id = Column(Integer, primary\_key=**True**)

keywords = relationship("Keyword", secondary="keywords")

As with traditional mapping, its generally not a good idea to use a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) as the "secondary" argument which is also mapped to a class, unless the [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)is declared with viewonly=True. Otherwise, the unit-of-work system may attempt duplicate INSERT and DELETE statements against the underlying table.

与传统的映射一样，它一般不是一个好主意，它使用一个表作为“次”参数，它也被映射到一个类，除非关系()被声明为viewonly = True。 否则，工作单位系统可能会对底层表进行重复的INSERT和DELETE语句。

# 7.4.3 Table Configuration

Table arguments other than the name, metadata, and mapped Column arguments are specified using the \_\_table\_args\_\_ class attribute. This attribute accommodates both positional as well as keyword arguments that are normally sent to the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) constructor. The attribute can be specified in one of two forms. One is as a dictionary:

使用\_\_table\_args\_\_类属性指定名称，元数据和映射列参数之外的表参数。 此属性适用于通常发送到表构造函数的位置以及关键字参数。 属性可以用两种形式之一来指定。 一个是字典：

**class** **MyClass**(Base):

\_\_tablename\_\_ = 'sometable'

\_\_table\_args\_\_ = {'mysql\_engine':'InnoDB'}

The other, a tuple, where each argument is positional (usually constraints):

另一个，一个元组，每个参数是位置(通常是约束)：

**class** **MyClass**(Base):

\_\_tablename\_\_ = 'sometable'

\_\_table\_args\_\_ = (

ForeignKeyConstraint(['id'], ['remote\_table.id']),

UniqueConstraint('foo'),

)

Keyword arguments can be specified with the above form by specifying the last argument as a dictionary:

通过将最后一个参数指定为字典，可以使用上述表单指定关键字参数：

**class** **MyClass**(Base):

\_\_tablename\_\_ = 'sometable'

\_\_table\_args\_\_ = (

ForeignKeyConstraint(['id'], ['remote\_table.id']),

UniqueConstraint('foo'),

{'autoload':**True**}

)

#### 7.4.3.1 Using a Hybrid Approach with \_\_table\_\_

As an alternative to \_\_tablename\_\_, a direct [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) construct may be used. The [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects, which in this case require their names, will be added to the mapping just like a regular mapping to a table:

作为\_\_tablename\_\_的替代方法，可以使用直接的Table构造。 Column对象(在这种情况下需要其名称)将被添加到映射中，就像对表的常规映射：

**class** **MyClass**(Base):

\_\_table\_\_ = Table('my\_table', Base.metadata,

Column('id', Integer, primary\_key=**True**),

Column('name', String(50))

)

\_\_table\_\_ provides a more focused point of control for establishing table metadata, while still getting most of the benefits of using declarative. An application that uses reflection might want to load table metadata elsewhere and pass it to declarative classes:

\_\_table\_\_为创建表元数据提供了更为集中的控制点，同时仍然获得使用声明式的最大优势。 使用反射的应用程序可能希望将表元数据加载到其他位置，并将其传递给声明类：

**from** **sqlalchemy.ext.declarative** **import** declarative\_base

Base = declarative\_base()Base.metadata.reflect(some\_engine)

**class** **User**(Base):

\_\_table\_\_ = metadata.tables['user']

**class** **Address**(Base):

\_\_table\_\_ = metadata.tables['address']

Some configuration schemes may find it more appropriate to use \_\_table\_\_, such as those which already take advantage of the data-driven nature of [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) to customize and/or automate schema definition.

某些配置方案可能会发现更适合使用\_\_table\_\_，例如已经利用了Table的数据驱动特性来定制和/或自动化模式定义的配置方案。

Note that when the \_\_table\_\_ approach is used, the object is immediately usable as a plain [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) within the class declaration body itself, as a Python class is only another syntactical block. Below this is illustrated by using the id column in the primaryjoin condition of a [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship):

请注意，当使用\_\_table\_\_方法时，对象可以立即用作类声明体本身中的普通表，因为Python类只是另一个语法块。 下面通过使用关系()的primaryjoin条件中的id列来说明：

**class** **MyClass**(Base):

\_\_table\_\_ = Table('my\_table', Base.metadata,

Column('id', Integer, primary\_key=**True**),

Column('name', String(50))

)

widgets = relationship(Widget,

primaryjoin=Widget.myclass\_id==\_\_table\_\_.c.id)

Similarly, mapped attributes which refer to \_\_table\_\_ can be placed inline, as below where we assign the name column to the attribute \_name, generating a synonym for name:

类似地，引用\_\_table\_\_的映射属性可以内联放置，如下所示，我们将name列分配给属性\_name，生成名称的同义词：

**from** **sqlalchemy.ext.declarative** **import** synonym\_for

**class** **MyClass**(Base):

\_\_table\_\_ = Table('my\_table', Base.metadata,

Column('id', Integer, primary\_key=**True**),

Column('name', String(50))

)

\_name = \_\_table\_\_.c.name

**@synonym\_for**("\_name")

**def** name(self):

**return** "Name: *%s*" % \_name

#### 7.2.3.2 Using Reflection with Declarative

It's easy to set up a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) that uses autoload=True in conjunction with a mapped class:

很容易设置一个使用autoload = True与映射类结合使用的表：

**class** **MyClass**(Base):

\_\_table\_\_ = Table('mytable', Base.metadata,

autoload=**True**, autoload\_with=some\_engine)

However, one improvement that can be made here is to not require the [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) to be available when classes are being first declared. To achieve this, use the[DeferredReflection](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.DeferredReflection" \o "sqlalchemy.ext.declarative.DeferredReflection) mixin, which sets up mappings only after a special prepare(engine) step is called:

然而，这里可以做出的一个改进是在第一次声明类时不要求引擎可用。 为了实现这一点，使用DeferredReflection mixin，它仅在调用特殊的准备(引擎)步骤之后才能设置映射：

**from** **sqlalchemy.ext.declarative** **import** declarative\_base, DeferredReflection

Base = declarative\_base(cls=DeferredReflection)

**class** **Foo**(Base):

\_\_tablename\_\_ = 'foo'

bars = relationship("Bar")

**class** **Bar**(Base):

\_\_tablename\_\_ = 'bar'

*# illustrate overriding of "bar.foo\_id" to have*

*# a foreign key constraint otherwise not*

*# reflected, such as when using MySQL*

foo\_id = Column(Integer, ForeignKey('foo.id'))

Base.prepare(e)

*New in version 0.8:*Added [DeferredReflection](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.DeferredReflection" \o "sqlalchemy.ext.declarative.DeferredReflection).

### 7.4.4 Inheritance Configuration

Declarative supports all three forms of inheritance as intuitively as possible. The inherits mapper keyword argument is not needed as declarative will determine this from the class itself. The various "polymorphic" keyword arguments are specified using \_\_mapper\_args\_\_.

声明式支持所有三种形式的继承，尽可能直观。 继承mapper关键字参数是不需要的，因为声明将从类本身确定这一点。 使用\_\_mapper\_args\_\_指定各种“多态”关键字参数。

**See also**

[Mapping Class Inheritance Hierarchies](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance.html) - general introduction to inheritance mapping with Declarative.

映射类继承层次结构 - 具有声明性的继承映射的一般介绍。

7.4.4.1 Joined Table Inheritance

Joined table inheritance is defined as a subclass that defines its own table:

连接表继承被定义为定义自己的表的子类：

**class** **Person**(Base):

\_\_tablename\_\_ = 'people'

id = Column(Integer, primary\_key=**True**)

discriminator = Column('type', String(50))

\_\_mapper\_args\_\_ = {'polymorphic\_on': discriminator}

**class** **Engineer**(Person):

\_\_tablename\_\_ = 'engineers'

\_\_mapper\_args\_\_ = {'polymorphic\_identity': 'engineer'}

id = Column(Integer, ForeignKey('people.id'), primary\_key=**True**)

primary\_language = Column(String(50))

Note that above, the Engineer.id attribute, since it shares the same attribute name as the Person.id attribute, will in fact represent the people.id and engineers.id columns together, with the "Engineer.id" column taking precedence if queried directly. To provide the Engineer class with an attribute that represents only the engineers.id column, give it a different attribute name:

注意，上面的Engineer.id属性，由于它与Person.id属性共享相同的属性名称，实际上将people.id和engineers.id列表示在一起，“Engineer.id”列优先于if 直接询问 为工程师类提供一个仅代表workers.id列的属性，给它一个不同的属性名称：

**class** **Engineer**(Person):

\_\_tablename\_\_ = 'engineers'

\_\_mapper\_args\_\_ = {'polymorphic\_identity': 'engineer'}

engineer\_id = Column('id', Integer, ForeignKey('people.id'),

primary\_key=**True**)

primary\_language = Column(String(50))

#### 7.4.4.2 Single Table Inheritance

Single table inheritance is defined as a subclass that does not have its own table; you just leave out the \_\_table\_\_ and \_\_tablename\_\_ attributes:

单表继承被定义为没有自己的表的子类; 你只是省略\_\_table\_\_和\_\_tablename\_\_属性：

**class** **Person**(Base):

\_\_tablename\_\_ = 'people'

id = Column(Integer, primary\_key=**True**)

discriminator = Column('type', String(50))

\_\_mapper\_args\_\_ = {'polymorphic\_on': discriminator}

**class** **Engineer**(Person):

\_\_mapper\_args\_\_ = {'polymorphic\_identity': 'engineer'}

primary\_language = Column(String(50))

When the above mappers are configured, the Person class is mapped to the people table *before* the primary\_language column is defined, and this column will not be included in its own mapping. When Engineer then defines the primary\_language column, the column is added to the people table so that it is included in the mapping for Engineer and is also part of the table's full set of columns. Columns which are not mapped to Person are also excluded from any other single or joined inheriting classes using the exclude\_properties mapper argument. Below, Manager will have all the attributes of Person and Manager but *not* theprimary\_language attribute of Engineer:

当配置上述映射器时，在定义primary\_language列之前，将Person类映射到people表，并且此列不会包含在其自己的映射中。 当工程师然后定义primary\_language列时，列将添加到people表中，以便它被包含在工程师的映射中，并且也是表的完整列的一部分。 未映射到Person的列也使用exclude\_properties mapper参数从任何其他单个或连接的继承类中排除。 以下管理员将具有Person和Manager的所有属性，但不具有工程师的primary\_language属性：

**class** **Manager**(Person):

\_\_mapper\_args\_\_ = {'polymorphic\_identity': 'manager'}

golf\_swing = Column(String(50))

The attribute exclusion logic is provided by the exclude\_properties mapper argument, and declarative's default behavior can be disabled by passing an explicit exclude\_properties collection (empty or otherwise) to the \_\_mapper\_args\_\_.

属性排除逻辑由exclude\_properties mapper参数提供，并且可以通过将显式的exclude\_properties集合(空或其他方式)传递给\_\_mapper\_args\_\_来禁用声明性的默认行为。

### Resolving Column Conflicts

Note above that the primary\_language and golf\_swing columns are "moved up" to be applied to Person.\_\_table\_\_, as a result of their declaration on a subclass that has no table of its own. A tricky case comes up when two subclasses want to specify *the same* column, as below:

请注意，primary\_language和golf\_swing列“向上移动”应用于Person .\_\_ table\_\_，因为它们在没有自己的表的子类上声明。 当两个子类想要指定相同的列时，会出现一个棘手的情况，如下所示：

**class** **Person**(Base):

\_\_tablename\_\_ = 'people'

id = Column(Integer, primary\_key=**True**)

discriminator = Column('type', String(50))

\_\_mapper\_args\_\_ = {'polymorphic\_on': discriminator}

**class** **Engineer**(Person):

\_\_mapper\_args\_\_ = {'polymorphic\_identity': 'engineer'}

start\_date = Column(DateTime)

**class** **Manager**(Person):

\_\_mapper\_args\_\_ = {'polymorphic\_identity': 'manager'}

start\_date = Column(DateTime)

Above, the start\_date column declared on both Engineer and Manager will result in an error:

上面，在工程师和管理器上声明的start\_date列会导致错误：

sqlalchemy.exc.ArgumentError: Column 'start\_date' on class

<class '\_\_main\_\_.Manager'> conflicts with existing

column 'people.start\_date'

In a situation like this, Declarative can't be sure of the intent, especially if the start\_date columns had, for example, different types. A situation like this can be resolved by using [declared\_attr](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.declared_attr" \o "sqlalchemy.ext.declarative.declared_attr) to define the [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) conditionally, taking care to return the ****existing column**** via the parent \_\_table\_\_ if it already exists:

在这种情况下，Declarative不能确定意图，特别是如果start\_date列具有例如不同类型。 可以通过使用declared\_attr有条件地定义Column来解决这种情况，注意通过父\_\_table\_\_(如果已经存在)返回现有列：

**from** **sqlalchemy.ext.declarative** **import** declared\_attr

**class** **Person**(Base):

\_\_tablename\_\_ = 'people'

id = Column(Integer, primary\_key=**True**)

discriminator = Column('type', String(50))

\_\_mapper\_args\_\_ = {'polymorphic\_on': discriminator}

**class** **Engineer**(Person):

\_\_mapper\_args\_\_ = {'polymorphic\_identity': 'engineer'}

**@declared\_attr**

**def** start\_date(cls):

"Start date column, if not present already."

**return** Person.\_\_table\_\_.c.get('start\_date', Column(DateTime))

**class** **Manager**(Person):

\_\_mapper\_args\_\_ = {'polymorphic\_identity': 'manager'}

**@declared\_attr**

**def** start\_date(cls):

"Start date column, if not present already."

**return** Person.\_\_table\_\_.c.get('start\_date', Column(DateTime))

Above, when Manager is mapped, the start\_date column is already present on the Person class. Declarative lets us return that [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) as a result in this case, where it knows to skip re-assigning the same column. If the mapping is mis-configured such that the start\_date column is accidentally re-assigned to a different table (such as, if we changed Manager to be joined inheritance without fixing start\_date), an error is raised which indicates an existing [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) is trying to be re-assigned to a different owning [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table).

以上，当管理器被映射时，start\_date列已经存在于Person类上。 在这种情况下，声明式使我们能够返回该列，因为它知道跳过重新分配相同的列。 如果映射配置错误，以致start\_date列被意外地重新分配给不同的表(例如，如果我们将Manager修改为继承而不修改start\_date)，则会出现一个错误，表示现有的列正在尝试 被重新分配到一个不同的拥有表。

*New in version 0.8:*[declared\_attr](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.declared_attr" \o "sqlalchemy.ext.declarative.declared_attr) can be used on a non-mixin class, and the returned [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) or other mapped attribute will be applied to the mapping as any other attribute. Previously, the resulting attribute would be ignored, and also result in a warning being emitted when a subclass was created.

版本0.8中的新增：declared\_attr可以在非混合类中使用，并且返回的列或其他映射属性将作为任何其他属性应用于映射。 以前，结果属性将被忽略，并且还会在创建子类时导致发出警告。

*New in version 0.8:*[declared\_attr](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.declared_attr" \o "sqlalchemy.ext.declarative.declared_attr), when used either with a mixin or non-mixin declarative class, can return an existing [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) already assigned to the parent [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table), to indicate that the re-assignment of the [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) should be skipped, however should still be mapped on the target class, in order to resolve duplicate column conflicts.

新版本0.8：declared\_attr，当与mixin或non-mixin声明类一起使用时，可以返回已分配给父表的现有列，以表明应重新分配列应该被跳过，但是应该仍然是 映射到目标类，以解决重复的列冲突。

The same concept can be used with mixin classes (see [Mixin and Custom Base Classes](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/mixins.html" \l "declarative-mixins)):

Mixin类可以使用相同的概念(参见Mixin和Custom Base Classes)：

**class** **Person**(Base):

\_\_tablename\_\_ = 'people'

id = Column(Integer, primary\_key=**True**)

discriminator = Column('type', String(50))

\_\_mapper\_args\_\_ = {'polymorphic\_on': discriminator}

**class** **HasStartDate**(object):

**@declared\_attr**

**def** start\_date(cls):

**return** cls.\_\_table\_\_.c.get('start\_date', Column(DateTime))

**class** **Engineer**(HasStartDate, Person):

\_\_mapper\_args\_\_ = {'polymorphic\_identity': 'engineer'}

**class** **Manager**(HasStartDate, Person):

\_\_mapper\_args\_\_ = {'polymorphic\_identity': 'manager'}

The above mixin checks the local \_\_table\_\_ attribute for the column. Because we're using single table inheritance, we're sure that in this case, cls.\_\_table\_\_refers to Person.\_\_table\_\_. If we were mixing joined- and single-table inheritance, we might want our mixin to check more carefully if cls.\_\_table\_\_ is really the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) we're looking for.

上面的mixin会检查列的本地\_\_table\_\_属性。 因为我们使用单表继承，我们确定在这种情况下，cls .\_\_ table\_\_引用到Person .\_\_ table\_\_。 如果我们混合连接和单表继承，我们可能希望我们的mixin更仔细地检查，如果cls .\_\_ table\_\_真的是我们正在寻找的表。

7.4.4.3 Concrete Table Inheritance

Concrete is defined as a subclass which has its own table and sets the concrete keyword argument to True:

具体定义为具有自己的表并将具体的关键字参数设置为True的子类：

**class** **Person**(Base):

\_\_tablename\_\_ = 'people'

id = Column(Integer, primary\_key=**True**)

name = Column(String(50))

**class** **Engineer**(Person):

\_\_tablename\_\_ = 'engineers'

\_\_mapper\_args\_\_ = {'concrete':**True**}

id = Column(Integer, primary\_key=**True**)

primary\_language = Column(String(50))

name = Column(String(50))

Usage of an abstract base class is a little less straightforward as it requires usage of [polymorphic\_union()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.util.polymorphic_union" \o "sqlalchemy.orm.util.polymorphic_union), which needs to be created with the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects before the class is built:

抽象基类的使用不太简单，因为它需要使用polymorphic\_union()，它需要在构建类之前使用Table对象创建：

engineers = Table('engineers', Base.metadata,

Column('id', Integer, primary\_key=**True**),

Column('name', String(50)),

Column('primary\_language', String(50))

)managers = Table('managers', Base.metadata,

Column('id', Integer, primary\_key=**True**),

Column('name', String(50)),

Column('golf\_swing', String(50))

)

punion = polymorphic\_union({

'engineer':engineers,

'manager':managers}, 'type', 'punion')

**class** **Person**(Base):

\_\_table\_\_ = punion

\_\_mapper\_args\_\_ = {'polymorphic\_on':punion.c.type}

**class** **Engineer**(Person):

\_\_table\_\_ = engineers

\_\_mapper\_args\_\_ = {'polymorphic\_identity':'engineer', 'concrete':**True**}

**class** **Manager**(Person):

\_\_table\_\_ = managers

\_\_mapper\_args\_\_ = {'polymorphic\_identity':'manager', 'concrete':**True**}

The helper classes [AbstractConcreteBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.AbstractConcreteBase" \o "sqlalchemy.ext.declarative.AbstractConcreteBase) and [ConcreteBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.ConcreteBase" \o "sqlalchemy.ext.declarative.ConcreteBase) provide automation for the above system of creating a polymorphic union. See the documentation for these helpers as well as the main ORM documentation on concrete inheritance for details.

帮助类AbstractConcreteBase和ConcreteBase为上述创建多态联合的系统提供了自动化。 有关这些帮助者的文档以及有关具体继承的主要ORM文档，请参见详细信息。

**See also**

[Concrete Table Inheritance](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance.html" \l "concrete-inheritance)

inheritance\_concrete\_helpers

# 7.4.5 Mixin and Custom Base Classes

A common need when using [declarative](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "module-sqlalchemy.ext.declarative" \o "sqlalchemy.ext.declarative) is to share some functionality, such as a set of common columns, some common table options, or other mapped properties, across many classes. The standard Python idioms for this is to have the classes inherit from a base which includes these common features.

使用声明性的常见需求是在许多类中共享一些功能，例如一组公共列，一些常用表选项或其他映射属性。 标准的Python成语是让类继承自包含这些常见功能的基础。

When using [declarative](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "module-sqlalchemy.ext.declarative" \o "sqlalchemy.ext.declarative), this idiom is allowed via the usage of a custom declarative base class, as well as a "mixin" class which is inherited from in addition to the primary base. Declarative includes several helper features to make this work in terms of how mappings are declared. An example of some commonly mixed-in idioms is below:

当使用声明式时，通过使用自定义声明性基类，以及除主基础之外继承的“mixin”类允许此成语。 声明式包括几个帮助功能，使得它能够如何声明映射。 一些常见的混合成语的例子如下：

**from** **sqlalchemy.ext.declarative** **import** declared\_attr

**class** **MyMixin**(object):

**@declared\_attr**

**def** \_\_tablename\_\_(cls):

**return** cls.\_\_name\_\_.lower()

\_\_table\_args\_\_ = {'mysql\_engine': 'InnoDB'}

\_\_mapper\_args\_\_= {'always\_refresh': **True**}

id = Column(Integer, primary\_key=**True**)

**class** **MyModel**(MyMixin, Base):

name = Column(String(1000))

Where above, the class MyModel will contain an "id" column as the primary key, a \_\_tablename\_\_ attribute that derives from the name of the class itself, as well as \_\_table\_args\_\_ and \_\_mapper\_args\_\_ defined by the MyMixin mixin class.

上面的类，MyModel类将包含一个“id”列作为主键，派生自类名称的\_\_tablename\_\_属性以及由MyMixin mixin类定义的\_\_table\_args\_\_和\_\_mapper\_args\_\_。

There's no fixed convention over whether MyMixin precedes Base or not. Normal Python method resolution rules apply, and the above example would work just as well with:

没有关于MyMixin是否在Base之前的固定的约定。 正常的Python方法解析规则适用，上面的例子同样适用于：

**class** **MyModel**(Base, MyMixin):

name = Column(String(1000))

This works because Base here doesn't define any of the variables that MyMixin defines, i.e. \_\_tablename\_\_, \_\_table\_args\_\_, id, etc. If the Base did define an attribute of the same name, the class placed first in the inherits list would determine which attribute is used on the newly defined class.

这是因为Base这里没有定义MyMixin定义的任何变量，即\_\_tablename\_\_，\_\_table\_args\_\_，id等。如果Base确实定义了同名的属性，则首先放在继承列表中的类将决定哪个属性 在新定义的类上使用。

7.4.5.1 Augmenting the Base

In addition to using a pure mixin, most of the techniques in this section can also be applied to the base class itself, for patterns that should apply to all classes derived from a particular base. This is achieved using the cls argument of the [declarative\_base()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.declarative_base" \o "sqlalchemy.ext.declarative.declarative_base) function:

除了使用纯mixin之外，本节中的大部分技术也可以应用于基类本身，适用于应用于从特定基础派生的所有类的模式。 这是使用declarative\_base()函数的cls参数实现的：

**from** **sqlalchemy.ext.declarative** **import** declared\_attr

**class** **Base**(object):

**@declared\_attr**

**def** \_\_tablename\_\_(cls):

**return** cls.\_\_name\_\_.lower()

\_\_table\_args\_\_ = {'mysql\_engine': 'InnoDB'}

id = Column(Integer, primary\_key=**True**)

**from** **sqlalchemy.ext.declarative** **import** declarative\_base

Base = declarative\_base(cls=Base)

**class** **MyModel**(Base):

name = Column(String(1000))

Where above, MyModel and all other classes that derive from Base will have a table name derived from the class name, an id primary key column, as well as the "InnoDB" engine for MySQL.

在上面，MyModel和从Base派生的所有其他类将具有派生自类名称，id主键列以及MySQL的“InnoDB”引擎的表名称。

7.4.5.2 Mixing in Columns

The most basic way to specify a column on a mixin is by simple declaration:

在mixin上指定列的最基本方法是简单的声明：

**class** **TimestampMixin**(object):

created\_at = Column(DateTime, default=func.now())

**class** **MyModel**(TimestampMixin, Base):

\_\_tablename\_\_ = 'test'

id = Column(Integer, primary\_key=**True**)

name = Column(String(1000))

Where above, all declarative classes that include TimestampMixin will also have a column created\_at that applies a timestamp to all row insertions.

在上面的位置，包括TimestampMixin的所有声明性类也将有一个create\_at列，将时间戳应用于所有行插入。

Those familiar with the SQLAlchemy expression language know that the object identity of clause elements defines their role in a schema. Two Table objects a and bmay both have a column called id, but the way these are differentiated is that a.c.id and b.c.id are two distinct Python objects, referencing their parent tables aand b respectively.

熟悉SQLAlchemy表达式语言的人们知道子句元素的对象标识定义了它们在模式中的作用。两个表对象a和bmay都有一个名为id的列，但这些区别的方式是a.c.id和b.c.id是两个不同的Python对象，分别引用它们的父表a和b。

In the case of the mixin column, it seems that only one [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) object is explicitly created, yet the ultimate created\_at column above must exist as a distinct Python object for each separate destination class. To accomplish this, the declarative extension creates a ****copy**** of each [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) object encountered on a class that is detected as a mixin.

在mixin列的情况下，似乎只有一个Column对象被明确地创建，但是上面的最终的create\_at列必须作为每个单独的目标类的一个独特的Python对象存在。为了实现这一点，声明性扩展将创建一个被检测为mixin的类上遇到的每个Column对象的副本。

This copy mechanism is limited to simple columns that have no foreign keys, as a [ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey) itself contains references to columns which can't be properly recreated at this level. For columns that have foreign keys, as well as for the variety of mapper-level constructs that require destination-explicit context, the [declared\_attr](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.declared_attr" \o "sqlalchemy.ext.declarative.declared_attr)decorator is provided so that patterns common to many classes can be defined as callables:

此复制机制仅限于没有外键的简单列，因为ForeignKey本身包含对在此级别无法正确重新创建的列的引用。对于具有外键的列以及需要目标显式上下文的各种映射程序级结构，提供了declared\_attrdecorator，以便许多类通用的模式可以被定义为可调用：

**from** **sqlalchemy.ext.declarative** **import** declared\_attr

**class** **ReferenceAddressMixin**(object):

**@declared\_attr**

**def** address\_id(cls):

**return** Column(Integer, ForeignKey('address.id'))

**class** **User**(ReferenceAddressMixin, Base):

\_\_tablename\_\_ = 'user'

id = Column(Integer, primary\_key=**True**)

Where above, the address\_id class-level callable is executed at the point at which the User class is constructed, and the declarative extension can use the resulting [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) object as returned by the method without the need to copy it.

在上面的地方，address\_id类级别可调用在构造User类的点执行，声明式扩展可以使用方法返回的结果Column对象，而不需要复制它。

*Changed in version 0.6.5:*Rename sqlalchemy.util.classproperty into [declared\_attr](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.declared_attr" \o "sqlalchemy.ext.declarative.declared_attr).

Columns generated by [declared\_attr](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.declared_attr" \o "sqlalchemy.ext.declarative.declared_attr) can also be referenced by \_\_mapper\_args\_\_ to a limited degree, currently by polymorphic\_on and version\_id\_col; the declarative extension will resolve them at class construction time:

由declared\_attr生成的列也可以由\_\_mapper\_args\_\_引用到目前有限的位置，目前由polymorphic\_on和version\_id\_col引用; 声明性扩展将在类构建时解析它们：

**class** **MyMixin**:

**@declared\_attr**

**def** type\_(cls):

**return** Column(String(50))

\_\_mapper\_args\_\_= {'polymorphic\_on':type\_}

**class** **MyModel**(MyMixin, Base):

\_\_tablename\_\_='test'

id = Column(Integer, primary\_key=**True**)

#### 7.4.5.3 Mixing in Relationships

Relationships created by [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) are provided with declarative mixin classes exclusively using the [declared\_attr](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.declared_attr" \o "sqlalchemy.ext.declarative.declared_attr) approach, eliminating any ambiguity which could arise when copying a relationship and its possibly column-bound contents. Below is an example which combines a foreign key column and a relationship so that two classes Foo and Bar can both be configured to reference a common target class via many-to-one:

由relationship()创建的关系仅使用declared\_attr方法提供声明性mixin类，从而消除了复制关系及其可能的列内容时可能出现的任何歧义。 下面是一个结合外键列和关系的示例，使得两个类Foo和Bar都可以配置为通过多对一引用通用的目标类：

**class** **RefTargetMixin**(object):

**@declared\_attr**

**def** target\_id(cls):

**return** Column('target\_id', ForeignKey('target.id'))

**@declared\_attr**

**def** target(cls):

**return** relationship("Target")

**class** **Foo**(RefTargetMixin, Base):

\_\_tablename\_\_ = 'foo'

id = Column(Integer, primary\_key=**True**)

**class** **Bar**(RefTargetMixin, Base):

\_\_tablename\_\_ = 'bar'

id = Column(Integer, primary\_key=**True**)

**class** **Target**(Base):

\_\_tablename\_\_ = 'target'

id = Column(Integer, primary\_key=**True**)

### Using Advanced Relationship Arguments (e.g. primaryjoin, etc.)

[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) definitions which require explicit primaryjoin, order\_by etc. expressions should in all but the most simplistic cases use ****late bound**** forms for these arguments, meaning, using either the string form or a lambda. The reason for this is that the related [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects which are to be configured using @declared\_attr are not available to another @declared\_attr attribute; while the methods will work and return new [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects, those are not the [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column)objects that Declarative will be using as it calls the methods on its own, thus using *different* [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects.

除了最简单的情况外，需要明确的primaryjoin，order\_by等表达式的[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)定义应该使用这些参数的后期绑定形式，这意味着使用字符串形式或lambda。 原因是使用@declared\_attr进行配置的相关列对象不可用于另一个@declared\_attr属性; 而这些方法将工作并返回新的Column对象，那些不是Declared将使用的ColumnObject，因为它自己调用方法，因此使用不同的Column对象。

The canonical example is the primaryjoin condition that depends upon another mixed-in column:

典型的例子是依赖于另一个混合列的primary join条件：

**class** **RefTargetMixin**(object):

**@declared\_attr**

**def** target\_id(cls):

**return** Column('target\_id', ForeignKey('target.id'))

**@declared\_attr**

**def** target(cls):

**return** relationship(Target,

primaryjoin=Target.id==cls.target\_id *# this is \*incorrect\**

)

Mapping a class using the above mixin, we will get an error like:

使用上面的mixin映射一个类，我们会得到如下错误：

sqlalchemy.exc.InvalidRequestError:

this ForeignKey's parent column is not

yet associated **with** a Table.

This is because the target\_id [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) we've called upon in our target() method is not the same [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) that declarative is actually going to map to our table.

The condition above is resolved using a lambda:

这是因为我们在target()方法中调用的target\_id列与声明式实际上映射到我们的表不同。

上面的条件是使用lambda来解决的：

**class** **RefTargetMixin**(object):

**@declared\_attr**

**def** target\_id(cls):

**return** Column('target\_id', ForeignKey('target.id'))

**@declared\_attr**

**def** target(cls):

**return** relationship(Target,

primaryjoin=**lambda**: Target.id==cls.target\_id

)

or alternatively, the string form (which ultimately generates a lambda):

或者替代地，字符串形式(其最终生成lambda)：

**class** **RefTargetMixin**(object):

**@declared\_attr**

**def** target\_id(cls):

**return** Column('target\_id', ForeignKey('target.id'))

**@declared\_attr**

**def** target(cls):

**return** relationship("Target",

primaryjoin="Target.id==*%s*.target\_id" % cls.\_\_name\_\_

)

#### 7.4.5.4 Mixing in deferred(), column\_property(), and other MapperProperty classes

Like [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship), all [MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty) subclasses such as [deferred()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "sqlalchemy.orm.deferred" \o "sqlalchemy.orm.deferred), [column\_property()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_columns.html" \l "sqlalchemy.orm.column_property" \o "sqlalchemy.orm.column_property), etc. ultimately involve references to columns, and therefore, when used with declarative mixins, have the [declared\_attr](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.declared_attr" \o "sqlalchemy.ext.declarative.declared_attr) requirement so that no reliance on copying is needed:

像[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)一样，所有[MapperProperty](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.MapperProperty" \o "sqlalchemy.orm.interfaces.MapperProperty)子类(如[deferred()](http://docs.sqlalchemy.org/en/rel_1_1/orm/loading_columns.html" \l "sqlalchemy.orm.deferred" \o "sqlalchemy.orm.deferred)，[column\_property()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_columns.html" \l "sqlalchemy.orm.column_property" \o "sqlalchemy.orm.column_property)等)最终都涉及到列的引用，因此当与声明式混合使用时，具有[declared\_attr](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.declared_attr" \o "sqlalchemy.ext.declarative.declared_attr)要求，因此不需要依靠复制。

**class** **SomethingMixin**(object):

**@declared\_attr**

**def** dprop(cls):

**return** deferred(Column(Integer))

**class** **Something**(SomethingMixin, Base):

\_\_tablename\_\_ = "something"

The [column\_property()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_columns.html" \l "sqlalchemy.orm.column_property" \o "sqlalchemy.orm.column_property) or other construct may refer to other columns from the mixin. These are copied ahead of time before the [declared\_attr](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.declared_attr" \o "sqlalchemy.ext.declarative.declared_attr) is invoked:

[column\_property()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_columns.html" \l "sqlalchemy.orm.column_property" \o "sqlalchemy.orm.column_property)或其他构造可以引用mixin中的其他列。 这些在调用[declared\_attr](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.declared_attr" \o "sqlalchemy.ext.declarative.declared_attr)之前提前复制：

**class** **SomethingMixin**(object):

x = Column(Integer)

y = Column(Integer)

**@declared\_attr**

**def** x\_plus\_y(cls):

**return** column\_property(cls.x + cls.y)

*Changed in version 1.0.0:*mixin columns are copied to the final mapped class so that [declared\_attr](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.declared_attr" \o "sqlalchemy.ext.declarative.declared_attr) methods can access the actual column that will be mapped.

在版本1.0.0中更改：将mixin列复制到最终映射的类，以便declared\_attr方法可以访问将映射的实际列。

#### 7.4.5.5 Mixing in Association Proxy and Other Attributes

Mixins can specify user-defined attributes as well as other extension units such as [association\_proxy()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.association_proxy" \o "sqlalchemy.ext.associationproxy.association_proxy). The usage of [declared\_attr](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.declared_attr" \o "sqlalchemy.ext.declarative.declared_attr) is required in those cases where the attribute must be tailored specifically to the target subclass. An example is when constructing multiple [association\_proxy()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.association_proxy" \o "sqlalchemy.ext.associationproxy.association_proxy) attributes which each target a different type of child object. Below is an [association\_proxy()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.association_proxy" \o "sqlalchemy.ext.associationproxy.association_proxy) / mixin example which provides a scalar list of string values to an implementing class:

Mixins可以指定用户定义的属性以及其他扩展单元，如[association\_proxy()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.association_proxy" \o "sqlalchemy.ext.associationproxy.association_proxy)。 在属性必须专门针对目标子类的情况下，必须使用[declared\_attr](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.declared_attr" \o "sqlalchemy.ext.declarative.declared_attr)。 一个例子是构造多个[association\_proxy()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.association_proxy" \o "sqlalchemy.ext.associationproxy.association_proxy)属性时，每个对象的目标是一个不同类型的子对象。 下面是一个[association\_proxy()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.association_proxy" \o "sqlalchemy.ext.associationproxy.association_proxy)/ mixin示例，它向实现类提供了一个字符串值的标量列表：

**from** **sqlalchemy** **import** Column, Integer, ForeignKey, String

**from** **sqlalchemy.orm** **import** relationship

**from** **sqlalchemy.ext.associationproxy** **import** association\_proxy

**from** **sqlalchemy.ext.declarative** **import** declarative\_base, declared\_attr

Base = declarative\_base()

**class** **HasStringCollection**(object):

**@declared\_attr**

**def** \_strings(cls):

**class** **StringAttribute**(Base):

\_\_tablename\_\_ = cls.string\_table\_name

id = Column(Integer, primary\_key=**True**)

value = Column(String(50), nullable=**False**)

parent\_id = Column(Integer,

ForeignKey('*%s*.id' % cls.\_\_tablename\_\_),

nullable=**False**)

**def** \_\_init\_\_(self, value):

self.value = value

**return** relationship(StringAttribute)

**@declared\_attr**

**def** strings(cls):

**return** association\_proxy('\_strings', 'value')

**class** **TypeA**(HasStringCollection, Base):

\_\_tablename\_\_ = 'type\_a'

string\_table\_name = 'type\_a\_strings'

id = Column(Integer(), primary\_key=**True**)

**class** **TypeB**(HasStringCollection, Base):

\_\_tablename\_\_ = 'type\_b'

string\_table\_name = 'type\_b\_strings'

id = Column(Integer(), primary\_key=**True**)

Above, the HasStringCollection mixin produces a [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) which refers to a newly generated class called StringAttribute. TheStringAttribute class is generated with its own [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) definition which is local to the parent class making usage of the HasStringCollection mixin. It also produces an [association\_proxy()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.association_proxy" \o "sqlalchemy.ext.associationproxy.association_proxy) object which proxies references to the strings attribute onto the value attribute of each StringAttribute instance.

以上，HasStringCollectionmixin 生成一个[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)，它引用了一个新生成的名为StringAttribute的类。 StringAttribute类是使用自己的[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)定义生成的，该定义是使用HasStringCollection mixin的父类的本地。 它还生成一个[association\_proxy()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.association_proxy" \o "sqlalchemy.ext.associationproxy.association_proxy)对象，它将strings属性引用到每个StringAttribute实例的value属性。

TypeA or TypeB can be instantiated given the constructor argument strings, a list of strings:

给定构造函数参数字符串，字符串列表可以实例化TypeA或TypeB：

ta = TypeA(strings=['foo', 'bar'])

tb = TypeA(strings=['bat', 'bar'])

This list will generate a collection of StringAttribute objects, which are persisted into a table that's local to either the type\_a\_strings or type\_b\_stringstable:

**>>>** print(ta.\_strings)

[<\_\_main\_\_.StringAttribute object at 0x10151cd90>, <\_\_main\_\_.StringAttribute object at 0x10151ce10>]

When constructing the [association\_proxy()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.association_proxy" \o "sqlalchemy.ext.associationproxy.association_proxy), the [declared\_attr](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.declared_attr" \o "sqlalchemy.ext.declarative.declared_attr) decorator must be used so that a distinct [association\_proxy()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.association_proxy" \o "sqlalchemy.ext.associationproxy.association_proxy) object is created for each of the TypeA and TypeB classes.

当构造[association\_proxy()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.association_proxy" \o "sqlalchemy.ext.associationproxy.association_proxy)时，必须使用[declared\_attr](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.declared_attr" \o "sqlalchemy.ext.declarative.declared_attr)装饰器，以便为每个TypeA和TypeB类创建一个有区别的[association\_proxy()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.association_proxy" \o "sqlalchemy.ext.associationproxy.association_proxy)对象。

*New in version 0.8:*[declared\_attr](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.declared_attr" \o "sqlalchemy.ext.declarative.declared_attr) is usable with non-mapped attributes, including user-defined attributes as well as [association\_proxy()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.association_proxy" \o "sqlalchemy.ext.associationproxy.association_proxy).

版本0.8中的新增：[declared\_attr](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.declared_attr" \o "sqlalchemy.ext.declarative.declared_attr)可用于非映射属性，包括用户定义的属性以及[association\_proxy()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.association_proxy" \o "sqlalchemy.ext.associationproxy.association_proxy)。

7.4.5.6 Controlling table inheritance with mixins

The \_\_tablename\_\_ attribute may be used to provide a function that will determine the name of the table used for each class in an inheritance hierarchy, as well as whether a class has its own distinct table.

\_\_tablename\_\_属性可用于提供一个函数，该函数将确定用于继承层次结构中每个类的表的名称，以及类是否具有其自己的不同表。

This is achieved using the [declared\_attr](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.declared_attr" \o "sqlalchemy.ext.declarative.declared_attr) indicator in conjunction with a method named \_\_tablename\_\_(). Declarative will always invoke [declared\_attr](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.declared_attr" \o "sqlalchemy.ext.declarative.declared_attr) for the special names \_\_tablename\_\_, \_\_mapper\_args\_\_ and \_\_table\_args\_\_ function ****for each mapped class in the hierarchy****. The function therefore needs to expect to receive each class individually and to provide the correct answer for each.

这是使用declared\_attr指示符与名为\_\_tablename \_\_()的方法结合使用的。 声明式将始终为层次结构中每个映射类的特殊名称\_\_tablename\_\_，\_\_mapper\_args\_\_和\_\_table\_args\_\_函数调用declared\_attr。 因此，该功能需要期望单独接收每个课程，并为每个课程提供正确的答案。

For example, to create a mixin that gives every class a simple table name based on class name:

例如，要创建一个mixin，它为每个类提供了一个基于类名的简单表名：

**from** **sqlalchemy.ext.declarative** **import** declared\_attr

**class** **Tablename**:

**@declared\_attr**

**def** \_\_tablename\_\_(cls):

**return** cls.\_\_name\_\_.lower()

**class** **Person**(Tablename, Base):

id = Column(Integer, primary\_key=**True**)

discriminator = Column('type', String(50))

\_\_mapper\_args\_\_ = {'polymorphic\_on': discriminator}

**class** **Engineer**(Person):

\_\_tablename\_\_ = **None**

\_\_mapper\_args\_\_ = {'polymorphic\_identity': 'engineer'}

primary\_language = Column(String(50))

Alternatively, we can modify our \_\_tablename\_\_ function to return None for subclasses, using [has\_inherited\_table()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.has_inherited_table" \o "sqlalchemy.ext.declarative.has_inherited_table). This has the effect of those subclasses being mapped with single table inheritance against the parent:

或者，我们可以使用has\_inherited\_table()修改我们的\_\_tablename\_\_函数以返回None类。 这具有这些子类与父表单对表进行映射的效果：

**from** **sqlalchemy.ext.declarative** **import** declared\_attr

**from** **sqlalchemy.ext.declarative** **import** has\_inherited\_table

**class** **Tablename**(object):

**@declared\_attr**

**def** \_\_tablename\_\_(cls):

**if** has\_inherited\_table(cls):

**return** **None**

**return** cls.\_\_name\_\_.lower()

**class** **Person**(Tablename, Base):

id = Column(Integer, primary\_key=**True**)

discriminator = Column('type', String(50))

\_\_mapper\_args\_\_ = {'polymorphic\_on': discriminator}

**class** **Engineer**(Person):

primary\_language = Column(String(50))

\_\_mapper\_args\_\_ = {'polymorphic\_identity': 'engineer'}

#### 7.4.5.7 Mixing in Columns in Inheritance Scenarios

In contrast to how \_\_tablename\_\_ and other special names are handled when used with [declared\_attr](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.declared_attr" \o "sqlalchemy.ext.declarative.declared_attr), when we mix in columns and properties (e.g. relationships, column properties, etc.), the function is invoked for the ****base class only**** in the hierarchy. Below, only the Person class will receive a column called id; the mapping will fail on Engineer, which is not given a primary key:

与[declared\_attr](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.declared_attr" \o "sqlalchemy.ext.declarative.declared_attr)一起使用时，如何处理\_\_tablename\_\_和其他特殊名称相反，当我们混合列和属性(例如关系，列属性等)时，仅在层次结构中为基类调用该函数。 下面只有Person类将收到一个名为id的列; Engineer的映射将失败，这不是主键：

**class** **HasId**(object):

**@declared\_attr**

**def** id(cls):

**return** Column('id', Integer, primary\_key=**True**)

**class** **Person**(HasId, Base):

\_\_tablename\_\_ = 'person'

discriminator = Column('type', String(50))

\_\_mapper\_args\_\_ = {'polymorphic\_on': discriminator}

**class** **Engineer**(Person):

\_\_tablename\_\_ = 'engineer'

primary\_language = Column(String(50))

\_\_mapper\_args\_\_ = {'polymorphic\_identity': 'engineer'}

It is usually the case in joined-table inheritance that we want distinctly named columns on each subclass. However in this case, we may want to have an id column on every table, and have them refer to each other via foreign key. We can achieve this as a mixin by using the [declared\_attr.cascading](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.declared_attr.cascading" \o "sqlalchemy.ext.declarative.declared_attr.cascading) modifier, which indicates that the function should be invoked ****for each class in the hierarchy****, just like it does for \_\_tablename\_\_:

在连接表继承中通常情况下，我们需要在每个子类上明确命名列。 但是在这种情况下，我们可能希望在每个表上都有一个id列，并通过外键引用它们。 我们可以通过使用declared\_attr.cascading修饰符来实现此作为一个mixin，这表明该层次结构中的每个类都应该调用该函数，就像\_\_tablename\_\_一样：

**class** **HasIdMixin**(object):

**@declared\_attr**.cascading

**def** id(cls):

**if** has\_inherited\_table(cls):

**return** Column(ForeignKey('person.id'), primary\_key=**True**)

**else**:

**return** Column(Integer, primary\_key=**True**)

**class** **Person**(HasIdMixin, Base):

\_\_tablename\_\_ = 'person'

discriminator = Column('type', String(50))

\_\_mapper\_args\_\_ = {'polymorphic\_on': discriminator}

**class** **Engineer**(Person):

\_\_tablename\_\_ = 'engineer'

primary\_language = Column(String(50))

\_\_mapper\_args\_\_ = {'polymorphic\_identity': 'engineer'}

*New in version 1.0.0:*added [declared\_attr.cascading](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.declared_attr.cascading" \o "sqlalchemy.ext.declarative.declared_attr.cascading).

#### 7.4.5.8 Combining Table/Mapper Arguments from Multiple Mixins

In the case of \_\_table\_args\_\_ or \_\_mapper\_args\_\_ specified with declarative mixins, you may want to combine some parameters from several mixins with those you wish to define on the class iteself. The [declared\_attr](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.declared_attr" \o "sqlalchemy.ext.declarative.declared_attr) decorator can be used here to create user-defined collation routines that pull from multiple collections:

在\_\_table\_args\_\_或\_\_mapper\_args\_\_用声明式混合指定的情况下，您可能需要将几个混合中的某些参数与您希望在类本身定义的参数组合。 declared\_attr装饰器可用于创建从多个集合中提取的用户定义的归类例程：

**from** **sqlalchemy.ext.declarative** **import** declared\_attr

**class** **MySQLSettings**(object):

\_\_table\_args\_\_ = {'mysql\_engine':'InnoDB'}

**class** **MyOtherMixin**(object):

\_\_table\_args\_\_ = {'info':'foo'}

**class** **MyModel**(MySQLSettings, MyOtherMixin, Base):

\_\_tablename\_\_='my\_model'

**@declared\_attr**

**def** \_\_table\_args\_\_(cls):

args = dict()

args.update(MySQLSettings.\_\_table\_args\_\_)

args.update(MyOtherMixin.\_\_table\_args\_\_)

**return** args

id = Column(Integer, primary\_key=**True**)

#### 7.4.5.10 Creating Indexes with Mixins

To define a named, potentially multicolumn [Index](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.Index" \o "sqlalchemy.schema.Index) that applies to all tables derived from a mixin, use the "inline" form of [Index](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.Index" \o "sqlalchemy.schema.Index) and establish it as part of \_\_table\_args\_\_:

**class** **MyMixin**(object):

a = Column(Integer)

b = Column(Integer)

**@declared\_attr**

**def** \_\_table\_args\_\_(cls):

**return** (Index('test\_idx\_*%s*' % cls.\_\_tablename\_\_, 'a', 'b'),)

**class** **MyModel**(MyMixin, Base):

\_\_tablename\_\_ = 'atable'

c = Column(Integer,primary\_key=**True**)

7.4.6 Declarative API

7.4.6.1 API Reference

sqlalchemy.ext.declarative.**declarative\_base**(*bind=None*, *metadata=None*, *mapper=None*, *cls=<type 'object'>*, *name='Base'*, *constructor=<function \_\_init\_\_>*, *class\_registry=None*, *metaclass=<class 'sqlalchemy.ext.declarative.api.DeclarativeMeta'>*)

Construct a base class for declarative class definitions.

The new base class will be given a metaclass that produces appropriate [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects and makes the appropriate [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper) calls based on the information provided declaratively in the class and any subclasses of the class.

|  |  |
| --- | --- |
| **Parameters:** | * ****bind**** – An optional [Connectable](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connectable" \o "sqlalchemy.engine.Connectable), will be assigned the bind attribute on the [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) instance. * ****metadata**** – An optional [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) instance. All [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects implicitly declared by subclasses of the base will share this MetaData. A MetaData instance will be created if none is provided. The [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) instance will be available via the metadata attribute of the generated declarative base class. * ****mapper**** – An optional callable, defaults to [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper). Will be used to map subclasses to their Tables. * ****cls**** – Defaults to object. A type to use as the base for the generated declarative base class. May be a class or tuple of classes. * ****name**** – Defaults to Base. The display name for the generated class. Customizing this is not required, but can improve clarity in tracebacks and debugging. * ****constructor**** – Defaults to \_declarative\_constructor(), an \_\_init\_\_ implementation that assigns \*\*kwargs for declared fields and relationships to an instance. If None is supplied, no \_\_init\_\_ will be provided and construction will fall back to cls.\_\_init\_\_ by way of the normal Python semantics. * ****class\_registry**** – optional dictionary that will serve as the registry of class names-> mapped classes when string names are used to identify classes inside of [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) and others. Allows two or more declarative base classes to share the same registry of class names for simplified inter-base relationships. * ****metaclass**** – Defaults to DeclarativeMeta. A metaclass or \_\_metaclass\_\_ compatible callable to use as the meta type of the generated declarative base class. |

*Changed in version 1.1:*if [declarative\_base.cls](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.declarative_base.params.cls" \o "sqlalchemy.ext.declarative.declarative_base) is a single class (rather than a tuple), the constructed base class will inherit its docstring.

**See also**

[as\_declarative()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.as_declarative" \o "sqlalchemy.ext.declarative.as_declarative)

sqlalchemy.ext.declarative.**as\_declarative**(*\*\*kw*)

Class decorator for [declarative\_base()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.declarative_base" \o "sqlalchemy.ext.declarative.declarative_base).

Provides a syntactical shortcut to the cls argument sent to [declarative\_base()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.declarative_base" \o "sqlalchemy.ext.declarative.declarative_base), allowing the base class to be converted in-place to a "declarative" base:

**from** **sqlalchemy.ext.declarative** **import** as\_declarative

**@as\_declarative**()

**class** **Base**(object):

**@declared\_attr**

**def** \_\_tablename\_\_(cls):

**return** cls.\_\_name\_\_.lower()

id = Column(Integer, primary\_key=**True**)

**class** **MyMappedClass**(Base):

*# ...*

All keyword arguments passed to [as\_declarative()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.as_declarative" \o "sqlalchemy.ext.declarative.as_declarative) are passed along to [declarative\_base()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.declarative_base" \o "sqlalchemy.ext.declarative.declarative_base).

*New in version 0.8.3.*

**See also**

[declarative\_base()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.declarative_base" \o "sqlalchemy.ext.declarative.declarative_base)

*class*sqlalchemy.ext.declarative.**declared\_attr**(*fget*, *cascading=False*)

Bases: sqlalchemy.orm.base.\_MappedAttribute, \_\_builtin\_\_.property

Mark a class-level method as representing the definition of a mapped property or special declarative member name.

@declared\_attr turns the attribute into a scalar-like property that can be invoked from the uninstantiated class. Declarative treats attributes specifically marked with @declared\_attr as returning a construct that is specific to mapping or declarative table configuration. The name of the attribute is that of what the non-dynamic version of the attribute would be.

@declared\_attr is more often than not applicable to mixins, to define relationships that are to be applied to different implementors of the class:

**class** **ProvidesUser**(object):

"A mixin that adds a 'user' relationship to classes."

**@declared\_attr**

**def** user(self):

**return** relationship("User")

It also can be applied to mapped classes, such as to provide a "polymorphic" scheme for inheritance:

**class** **Employee**(Base):

id = Column(Integer, primary\_key=**True**)

type = Column(String(50), nullable=**False**)

**@declared\_attr**

**def** \_\_tablename\_\_(cls):

**return** cls.\_\_name\_\_.lower()

**@declared\_attr**

**def** \_\_mapper\_args\_\_(cls):

**if** cls.\_\_name\_\_ == 'Employee':

**return** {

"polymorphic\_on":cls.type,

"polymorphic\_identity":"Employee"

}

**else**:

**return** {"polymorphic\_identity":cls.\_\_name\_\_}

*Changed in version 0.8:*[declared\_attr](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.declared_attr" \o "sqlalchemy.ext.declarative.declared_attr) can be used with non-ORM or extension attributes, such as user-defined attributes or [association\_proxy()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "sqlalchemy.ext.associationproxy.association_proxy" \o "sqlalchemy.ext.associationproxy.association_proxy) objects, which will be assigned to the class at class construction time.

**cascading**

Mark a [declared\_attr](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.declared_attr" \o "sqlalchemy.ext.declarative.declared_attr) as cascading.

This is a special-use modifier which indicates that a column or MapperProperty-based declared attribute should be configured distinctly per mapped subclass, within a mapped-inheritance scenario.

这是一个特殊用途修饰符，它指示在映射继承方案中，应该在每个映射的子类中明确配置一个基于或者基于MapperProperty的声明属性。

Below, both MyClass as well as MySubClass will have a distinct id Column object established:

下面，MyClass以及MySubClass都将建立一个独特的id列对象：

**class** **HasIdMixin**(object):

**@declared\_attr**.cascading

**def** id(cls):

**if** has\_inherited\_table(cls):

**return** Column(ForeignKey('myclass.id'), primary\_key=**True**)

**else**:

**return** Column(Integer, primary\_key=**True**)

**class** **MyClass**(HasIdMixin, Base):

\_\_tablename\_\_ = 'myclass'

*# ...*

**class** **MySubClass**(MyClass):

""

*# ...*

The behavior of the above configuration is that MySubClass will refer to both its own id column as well as that of MyClass underneath the attribute named some\_id.

上述配置的行为是MySubClass将引用其自己的id列以及名为some\_id的属性下的MyClass。

**See also**

[Inheritance Configuration](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/inheritance.html" \l "declarative-inheritance)

[Mixing in Columns in Inheritance Scenarios](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/mixins.html" \l "mixin-inheritance-columns)

sqlalchemy.ext.declarative.api.**\_declarative\_constructor**(*self*, *\*\*kwargs*)

A simple constructor that allows initialization from kwargs.

一个简单的构造函数，允许从kwargs初始化。

Sets attributes on the constructed instance using the names and values in kwargs.

使用kwargs中的名称和值设置构造的实例上的属性。

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

只允许作为实例类的属性存在的键。 这些可以是例如任何映射的列或关系。

sqlalchemy.ext.declarative.**has\_inherited\_table**(*cls*)

Given a class, return True if any of the classes it inherits from has a mapped table, otherwise return False.

给定一个类，如果其继承的任何类具有映射表，则返回True，否则返回False。

This is used in declarative mixins to build attributes that behave differently for the base class vs. a subclass in an inheritance hierarchy.

这在声明性mixins中用于构建对于基类与继承层次结构中的子类的行为不同的属性。

**See also**

[Controlling table inheritance with mixins](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/mixins.html" \l "decl-mixin-inheritance)

sqlalchemy.ext.declarative.**synonym\_for**(*name*, *map\_column=False*)

Decorator, make a Python @property a query synonym for a column.

装饰器，使Python @property成为列的查询同义词。

A decorator version of [synonym()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapped_attributes.html" \l "sqlalchemy.orm.synonym" \o "sqlalchemy.orm.synonym). The function being decorated is the 'descriptor', otherwise passes its arguments through to [synonym()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapped_attributes.html" \l "sqlalchemy.orm.synonym" \o "sqlalchemy.orm.synonym):

[synonym()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapped_attributes.html" \l "sqlalchemy.orm.synonym" \o "sqlalchemy.orm.synonym)的装饰版本。 装饰的功能是“描述符”，否则将其参数传递给[synonym()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapped_attributes.html" \l "sqlalchemy.orm.synonym" \o "sqlalchemy.orm.synonym)：

**@synonym\_for**('col')

**@property**

**def** prop(self):

**return** 'special sauce'

The regular synonym() is also usable directly in a declarative setting and may be convenient for read/write properties:

常规synonym()也可以直接在声明性设置中使用，并且可以方便读/写属性：

prop = synonym('col', descriptor=property(\_read\_prop, \_write\_prop))

sqlalchemy.ext.declarative.**comparable\_using**(*comparator\_factory*)

Decorator, allow a Python @property to be used in query criteria.

This is a decorator front end to comparable\_property() that passes through the comparator\_factory and the function being decorated:

装饰器，允许在查询条件中使用Python @property。

这是通过comparator\_factory和正在装饰的函数的comparable\_property()的装饰器前端：

**@comparable\_using**(MyComparatorType)

**@property**

**def** prop(self):

**return** 'special sauce'

The regular comparable\_property() is also usable directly in a declarative setting and may be convenient for read/write properties:

comparable\_property()也可以直接在声明性设置中使用，并且可以方便读/写属性：

prop = comparable\_property(MyComparatorType)

sqlalchemy.ext.declarative.**instrument\_declarative**(*cls*, *registry*, *metadata*)

Given a class, configure the class declaratively, using the given registry, which can be any dictionary, and MetaData object.

给定一个类，使用给定的注册表(可以是任何字典)和MetaData对象来声明地配置该类。

*class*sqlalchemy.ext.declarative.**AbstractConcreteBase**

Bases: sqlalchemy.ext.declarative.api.ConcreteBase

A helper class for 'concrete' declarative mappings.

[AbstractConcreteBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.AbstractConcreteBase" \o "sqlalchemy.ext.declarative.AbstractConcreteBase) will use the [polymorphic\_union()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.util.polymorphic_union" \o "sqlalchemy.orm.util.polymorphic_union) function automatically, against all tables mapped as a subclass to this class. The function is called via the \_\_declare\_last\_\_() function, which is essentially a hook for the [after\_configured()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents.after_configured" \o "sqlalchemy.orm.events.MapperEvents.after_configured) event.

“具体”声明映射的辅助类。

[AbstractConcreteBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.AbstractConcreteBase" \o "sqlalchemy.ext.declarative.AbstractConcreteBase)将自动使用[polymorphic\_union()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.util.polymorphic_union" \o "sqlalchemy.orm.util.polymorphic_union)函数，对所有映射为该类的子类的表。 该函数通过\_\_declare\_last\_\_()函数调用，该函数本质上是一个用于[after\_configured()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents.after_configured" \o "sqlalchemy.orm.events.MapperEvents.after_configured)事件的钩子。

[AbstractConcreteBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.AbstractConcreteBase" \o "sqlalchemy.ext.declarative.AbstractConcreteBase) does produce a mapped class for the base class, however it is not persisted to any table; it is instead mapped directly to the "polymorphic" selectable directly and is only used for selecting. Compare to [ConcreteBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.ConcreteBase" \o "sqlalchemy.ext.declarative.ConcreteBase), which does create a persisted table for the base class.

[AbstractConcreteBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.AbstractConcreteBase" \o "sqlalchemy.ext.declarative.AbstractConcreteBase)确实为基类生成了一个映射类，但它并没有保留到任何表中; 它直接映射到直接选择的“多态”，仅用于选择。 与[ConcreteBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.ConcreteBase" \o "sqlalchemy.ext.declarative.ConcreteBase)进行比较，它为基类创建一个持久化的表。

Example:

**from** **sqlalchemy.ext.declarative** **import** AbstractConcreteBase

**class** **Employee**(AbstractConcreteBase, Base):

**pass**

**class** **Manager**(Employee):

\_\_tablename\_\_ = 'manager'

employee\_id = Column(Integer, primary\_key=**True**)

name = Column(String(50))

manager\_data = Column(String(40))

\_\_mapper\_args\_\_ = {

'polymorphic\_identity':'manager',

'concrete':**True**}

The abstract base class is handled by declarative in a special way; at class configuration time, it behaves like a declarative mixin or an \_\_abstract\_\_ base class. Once classes are configured and mappings are produced, it then gets mapped itself, but after all of its decscendants. This is a very unique system of mapping not found in any other SQLAlchemy system.

抽象基类以特殊方式由声明式处理; 在类配置时，它的行为就像一个声明性mixin或\_\_abstract\_\_基类。 一旦配置了类并且生成了映射，它就会被映射到本身，但是在它所有的暗杀之后。 这是一个非常独特的映射系统，在任何其他SQLAlchemy系统中都没有找到。

Using this approach, we can specify columns and properties that will take place on mapped subclasses, in the way that we normally do as in [Mixin and Custom Base Classes](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/mixins.html" \l "declarative-mixins):

使用这种方法，我们可以按照我们通常在Mixin和Custom Base Class中进行的方式指定将在映射子类上进行的列和属性：

**class** **Company**(Base):

\_\_tablename\_\_ = 'company'

id = Column(Integer, primary\_key=**True**)

**class** **Employee**(AbstractConcreteBase, Base):

employee\_id = Column(Integer, primary\_key=**True**)

**@declared\_attr**

**def** company\_id(cls):

**return** Column(ForeignKey('company.id'))

**@declared\_attr**

**def** company(cls):

**return** relationship("Company")

**class** **Manager**(Employee):

\_\_tablename\_\_ = 'manager'

name = Column(String(50))

manager\_data = Column(String(40))

\_\_mapper\_args\_\_ = {

'polymorphic\_identity':'manager',

'concrete':**True**}

When we make use of our mappings however, both Manager and Employee will have an independently usable .company attribute:

然而，当我们使用我们的映射时，Manager和Employee将具有独立可用的.company属性：

session.query(Employee).filter(Employee.company.has(id=5))

*Changed in version 1.0.0:*- The mechanics of [AbstractConcreteBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.AbstractConcreteBase" \o "sqlalchemy.ext.declarative.AbstractConcreteBase) have been reworked to support relationships established directly on the abstract base, without any special configurational steps.

在版本1.0.0中更改： - AbstractConcreteBase的机制已经重做，以支持直接在抽象基础上建立的关系，而无需任何特殊的配置步骤。

**See also**

[ConcreteBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.ConcreteBase" \o "sqlalchemy.ext.declarative.ConcreteBase)

[Concrete Table Inheritance](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance.html" \l "concrete-inheritance)

inheritance\_concrete\_helpers

*class*sqlalchemy.ext.declarative.**ConcreteBase**

A helper class for 'concrete' declarative mappings.

“具体”声明映射的辅助类。

[ConcreteBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.ConcreteBase" \o "sqlalchemy.ext.declarative.ConcreteBase) will use the [polymorphic\_union()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.util.polymorphic_union" \o "sqlalchemy.orm.util.polymorphic_union) function automatically, against all tables mapped as a subclass to this class. The function is called via the \_\_declare\_last\_\_() function, which is essentially a hook for the [after\_configured()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents.after_configured" \o "sqlalchemy.orm.events.MapperEvents.after_configured) event.

ConcreteBase将自动使用polymorphic\_union()函数，对所有映射为该类的子类的表。 该函数通过\_\_declare\_last \_\_()函数调用，该函数本质上是一个用于after\_configured()事件的钩子。

[ConcreteBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.ConcreteBase" \o "sqlalchemy.ext.declarative.ConcreteBase) produces a mapped table for the class itself. Compare to [AbstractConcreteBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.AbstractConcreteBase" \o "sqlalchemy.ext.declarative.AbstractConcreteBase), which does not.

ConcreteBase为类本身产生映射表。 与AbstractConcreteBase进行比较，而不是。

Example:

**from** **sqlalchemy.ext.declarative** **import** ConcreteBase

**class** **Employee**(ConcreteBase, Base):

\_\_tablename\_\_ = 'employee'

employee\_id = Column(Integer, primary\_key=**True**)

name = Column(String(50))

\_\_mapper\_args\_\_ = {

'polymorphic\_identity':'employee',

'concrete':**True**}

**class** **Manager**(Employee):

\_\_tablename\_\_ = 'manager'

employee\_id = Column(Integer, primary\_key=**True**)

name = Column(String(50))

manager\_data = Column(String(40))

\_\_mapper\_args\_\_ = {

'polymorphic\_identity':'manager',

'concrete':**True**}

**See also**

[AbstractConcreteBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.AbstractConcreteBase" \o "sqlalchemy.ext.declarative.AbstractConcreteBase)

[Concrete Table Inheritance](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance.html" \l "concrete-inheritance)

inheritance\_concrete\_helpers

*class*sqlalchemy.ext.declarative.**DeferredReflection**

A helper class for construction of mappings based on a deferred reflection step.

基于延迟反射步骤构建映射的帮助类。

Normally, declarative can be used with reflection by setting a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object using autoload=True as the \_\_table\_\_ attribute on a declarative class. The caveat is that the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) must be fully reflected, or at the very least have a primary key column, at the point at which a normal declarative mapping is constructed, meaning the [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) must be available at class declaration time.

通常，声明式可以通过使用autoload = True作为声明类上的\_\_table\_\_属性设置Table对象来反映。 需要注意的是，在构建正常声明映射的时候，表必须被完全反映，或者至少有一个主键列，这意味着引擎必须在类声明时可用。

The [DeferredReflection](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.DeferredReflection" \o "sqlalchemy.ext.declarative.DeferredReflection) mixin moves the construction of mappers to be at a later point, after a specific method is called which first reflects all [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)objects created so far. Classes can define it as such:

DeferredReflection mixin将映射器的构造移动到稍后的位置，在调用了一个首先反映到目前为止创建的所有Tableobjects的特定方法之后。 类可以定义为：

**from** **sqlalchemy.ext.declarative** **import** declarative\_base

**from** **sqlalchemy.ext.declarative** **import** Deferred

ReflectionBase = declarative\_base()

**class** **MyClass**(DeferredReflection, Base):

\_\_tablename\_\_ = 'mytable'

Above, MyClass is not yet mapped. After a series of classes have been defined in the above fashion, all tables can be reflected and mappings created using[prepare()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.DeferredReflection.prepare" \o "sqlalchemy.ext.declarative.DeferredReflection.prepare):

以上，MyClass尚未映射。 在以上方式定义了一系列类后，可以反映所有表格，并使用[prepare()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.DeferredReflection.prepare" \o "sqlalchemy.ext.declarative.DeferredReflection.prepare)创建映射：

engine = create\_engine("someengine://...")

DeferredReflection.prepare(engine)

The [DeferredReflection](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.DeferredReflection" \o "sqlalchemy.ext.declarative.DeferredReflection) mixin can be applied to individual classes, used as the base for the declarative base itself, or used in a custom abstract class. Using an abstract base allows that only a subset of classes to be prepared for a particular prepare step, which is necessary for applications that use more than one engine. For example, if an application has two engines, you might use two bases, and prepare each separately, e.g.:

[DeferredReflection](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.DeferredReflection" \o "sqlalchemy.ext.declarative.DeferredReflection) mixin可以应用于单独的类，用作声明性基础本身的基础，或者用于自定义抽象类。 使用抽象基础允许仅为一个特定的准备步骤准备一个类的子集，这对于使用多个引擎的应用程序是必需的。 例如，如果应用程序有两个引擎，则可以使用两个基础，并单独准备，例如：

**class** **ReflectedOne**(DeferredReflection, Base):

\_\_abstract\_\_ = **True**

**class** **ReflectedTwo**(DeferredReflection, Base):

\_\_abstract\_\_ = **True**

**class** **MyClass**(ReflectedOne):

\_\_tablename\_\_ = 'mytable'

**class** **MyOtherClass**(ReflectedOne):

\_\_tablename\_\_ = 'myothertable'

**class** **YetAnotherClass**(ReflectedTwo):

\_\_tablename\_\_ = 'yetanothertable'

*# ... etc.*

Above, the class hierarchies for ReflectedOne and ReflectedTwo can be configured separately:

以上，可以单独配置ReflectedOne和ReflectedTwo的类层次结构：

ReflectedOne.prepare(engine\_one)

ReflectedTwo.prepare(engine\_two)

*New in version 0.8.*

*classmethod***prepare**(*engine*)

Reflect all [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects for all current [DeferredReflection](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "sqlalchemy.ext.declarative.DeferredReflection" \o "sqlalchemy.ext.declarative.DeferredReflection) subclasses

### Special Directives

#### \_\_declare\_last\_\_()

The \_\_declare\_last\_\_() hook allows definition of a class level function that is automatically called by the [MapperEvents.after\_configured()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents.after_configured" \o "sqlalchemy.orm.events.MapperEvents.after_configured) event, which occurs after mappings are assumed to be completed and the 'configure' step has finished:

\_\_declare\_last \_\_()钩子允许定义由MapperEvents.after\_configured()事件自动调用的类级别函数，这是在映射映射完成并且“configure”步骤完成之后发生的：

**class** **MyClass**(Base):

**@classmethod**

**def** \_\_declare\_last\_\_(cls):

""

*# do something with mappings*

*New in version 0.7.3.*

#### \_\_declare\_first\_\_()

Like \_\_declare\_last\_\_(), but is called at the beginning of mapper configuration via the [MapperEvents.before\_configured()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.MapperEvents.before_configured" \o "sqlalchemy.orm.events.MapperEvents.before_configured) event:

像\_\_declare\_last \_\_()，但是通过MapperEvents.before\_configured()事件在映射器配置开始时调用：

**class** **MyClass**(Base):

**@classmethod**

**def** \_\_declare\_first\_\_(cls):

""

*# do something before mappings are configured*

*New in version 0.9.3.*

#### \_\_abstract\_\_

\_\_abstract\_\_ causes declarative to skip the production of a table or mapper for the class entirely. A class can be added within a hierarchy in the same way as mixin (see [Mixin and Custom Base Classes](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/mixins.html" \l "declarative-mixins)), allowing subclasses to extend just from the special class:

\_\_abstract\_\_导致声明性完全跳过该类生成表或映射器。 一个类可以以与mixin相同的方式添加到层次结构中(参见Mixin和Custom Base Classes)，允许子类从特殊类扩展：

**class** **SomeAbstractBase**(Base):

\_\_abstract\_\_ = **True**

**def** some\_helpful\_method(self):

""

**@declared\_attr**

**def** \_\_mapper\_args\_\_(cls):

**return** {"helpful mapper arguments":**True**}

**class** **MyMappedClass**(SomeAbstractBase):

""

One possible use of \_\_abstract\_\_ is to use a distinct [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) for different bases:

\_\_abstract\_\_的一个可能的用途是使用不同的MetaData作为不同的基础：

Base = declarative\_base()

**class** **DefaultBase**(Base):

\_\_abstract\_\_ = **True**

metadata = MetaData()

**class** **OtherBase**(Base):

\_\_abstract\_\_ = **True**

metadata = MetaData()

Above, classes which inherit from DefaultBase will use one [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) as the registry of tables, and those which inherit from OtherBase will use a different one. The tables themselves can then be created perhaps within distinct databases:

以上，从DefaultBase继承的类将使用一个MetaData作为表的注册表，而继承自OtherBase的那些将使用不同的。 然后可以在不同的数据库中创建表本身：

DefaultBase.metadata.create\_all(some\_engine)

OtherBase.metadata\_create\_all(some\_other\_engine)

*New in version 0.7.3.*

# 7.5 Mutation Tracking

Provide support for tracking of in-place changes to scalar values, which are propagated into ORM change events on owning parent objects.

*New in version 0.7:*[sqlalchemy.ext.mutable](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "module-sqlalchemy.ext.mutable" \o "sqlalchemy.ext.mutable) replaces SQLAlchemy's legacy approach to in-place mutations of scalar values; see [Mutation event extension, supersedes "mutable=True"](http://docs.sqlalchemy.org/en/rel_1_1/changelog/migration_07.html" \l "migration-mutation-extension).

提供跟踪标量值的就地更改的支持，标量值将传播到拥有父对象的ORM更改事件中。

版本0.7中的新功能：sqlalchemy.ext.mutable将SQLAlchemy的遗留方法替换为标量值的就地突变; 参见Mutation事件扩展，取代“mutable = True”。

7.5.1 Establishing Mutability on Scalar Column Values

A typical example of a "mutable" structure is a Python dictionary. Following the example introduced in [Column and Data Types](http://docs.sqlalchemy.org/en/rel_1_1/core/types.html), we begin with a custom type that marshals Python dictionaries into JSON strings before being persisted:

“可变”结构的典型示例是Python字典。 按照列和数据类型中介绍的例子，我们从一个自定义类型开始，在保持Python之前将Python字典编译成JSON字符串：

**from** **sqlalchemy.types** **import** TypeDecorator, VARCHAR

**import** **json**

**class** **JSONEncodedDict**(TypeDecorator):

"Represents an immutable structure as a json-encoded string."

impl = VARCHAR

**def** process\_bind\_param(self, value, dialect):

**if** value **is** **not** **None**:

value = json.dumps(value)

**return** value

**def** process\_result\_value(self, value, dialect):

**if** value **is** **not** **None**:

value = json.loads(value)

**return** value

The usage of json is only for the purposes of example. The [sqlalchemy.ext.mutable](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "module-sqlalchemy.ext.mutable" \o "sqlalchemy.ext.mutable) extension can be used with any type whose target Python type may be mutable, including [PickleType](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.PickleType" \o "sqlalchemy.types.PickleType), [postgresql.ARRAY](http://docs.sqlalchemy.org/en/rel_1_1/dialects/postgresql.html" \l "sqlalchemy.dialects.postgresql.ARRAY" \o "sqlalchemy.dialects.postgresql.ARRAY), etc.

json的用法只是为了示例的目的。 sqlalchemy.ext.mutable扩展可以用于目标Python类型可能是可变的任何类型，包括PickleType，postgresql.ARRAY等。

When using the [sqlalchemy.ext.mutable](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "module-sqlalchemy.ext.mutable" \o "sqlalchemy.ext.mutable) extension, the value itself tracks all parents which reference it. Below, we illustrate a simple version of the [MutableDict](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableDict" \o "sqlalchemy.ext.mutable.MutableDict) dictionary object, which applies the [Mutable](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.Mutable" \o "sqlalchemy.ext.mutable.Mutable) mixin to a plain Python dictionary:

当使用sqlalchemy.ext.mutable扩展名时，值本身会跟踪引用它的所有父类。 下面我们来说明一个简单版本的MutableDict字典对象，它将Mutable mixin应用到一个简单的Python字典：

**from** **sqlalchemy.ext.mutable** **import** Mutable

**class** **MutableDict**(Mutable, dict):

**@classmethod**

**def** coerce(cls, key, value):

"Convert plain dictionaries to MutableDict."

**if** **not** isinstance(value, MutableDict):

**if** isinstance(value, dict):

**return** MutableDict(value)

*# this call will raise ValueError*

**return** Mutable.coerce(key, value)

**else**:

**return** value

**def** \_\_setitem\_\_(self, key, value):

"Detect dictionary set events and emit change events."

dict.\_\_setitem\_\_(self, key, value)

self.changed()

**def** \_\_delitem\_\_(self, key):

"Detect dictionary del events and emit change events."

dict.\_\_delitem\_\_(self, key)

self.changed()

The above dictionary class takes the approach of subclassing the Python built-in dict to produce a dict subclass which routes all mutation events through \_\_setitem\_\_. There are variants on this approach, such as subclassing UserDict.UserDict or collections.MutableMapping; the part that's important to this example is that the [Mutable.changed()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.Mutable.changed" \o "sqlalchemy.ext.mutable.Mutable.changed) method is called whenever an in-place change to the datastructure takes place.

上述字典类采用对Python内置dict进行子类化的方法来生成一个dict子类，它通过\_\_setitem\_\_路由所有突变事件。这种方法有变体，例如子类化UserDict.UserDict或collections.MutableMapping;对这个例子很重要的部分是每当对数据结构进行就地更改时，都会调用Mutable.changed()方法。

We also redefine the [Mutable.coerce()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.Mutable.coerce" \o "sqlalchemy.ext.mutable.Mutable.coerce) method which will be used to convert any values that are not instances of MutableDict, such as the plain dictionaries returned by the json module, into the appropriate type. Defining this method is optional; we could just as well created our JSONEncodedDict such that it always returns an instance of MutableDict, and additionally ensured that all calling code uses MutableDict explicitly. When [Mutable.coerce()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.Mutable.coerce" \o "sqlalchemy.ext.mutable.Mutable.coerce) is not overridden, any values applied to a parent object which are not instances of the mutable type will raise a ValueError.

我们还重新定义了Mutable.coerce()方法，它将用于将不是MutableDict实例的任何值(如json模块返回的简单字典)转换为适当的类型。定义此方法是可选的;我们也可以创建我们的JSONEncodedDict，这样它总是返回一个MutableDict的实例，另外确保所有的调用代码使用MutableDict显式。当Mutable.coerce()不被覆盖时，应用于不可变类型实例的父对象的任何值都将引发ValueError。

Our new MutableDict type offers a class method [as\_mutable()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.Mutable.as_mutable" \o "sqlalchemy.ext.mutable.Mutable.as_mutable) which we can use within column metadata to associate with types. This method grabs the given type object or class and associates a listener that will detect all future mappings of this type, applying event listening instrumentation to the mapped attribute. Such as, with classical table metadata:

我们的新的MutableDict类型提供了一个类方法as\_mutable()，我们可以在列元数据中使用类方法与类型相关联。此方法可以捕获给定的类型对象或类，并将侦听器关联，该侦听器将检测此类型的所有未来映射，将事件侦听侦听应用于映射属性。例如，与古典表元数据：

**from** **sqlalchemy** **import** Table, Column, Integer

my\_data = Table('my\_data', metadata,

Column('id', Integer, primary\_key=**True**),

Column('data', MutableDict.as\_mutable(JSONEncodedDict)))

Above, [as\_mutable()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.Mutable.as_mutable" \o "sqlalchemy.ext.mutable.Mutable.as_mutable) returns an instance of JSONEncodedDict (if the type object was not an instance already), which will intercept any attributes which are mapped against this type. Below we establish a simple mapping against the my\_data table:

以上，as\_mutable()返回一个JSONEncodedDict的实例(如果类型对象不是一个实例)，它将拦截任何根据此类型映射的属性。 下面我们根据my\_data表建立一个简单的映射：

**from** **sqlalchemy** **import** mapper

**class** **MyDataClass**(object):

**pass**

*# associates mutation listeners with MyDataClass.data*mapper(MyDataClass, my\_data)

The MyDataClass.data member will now be notified of in place changes to its value.

现在将通知MyDataClass.data成员对其值的更改。

There's no difference in usage when using declarative:

使用声明时使用方面没有区别：

**from** **sqlalchemy.ext.declarative** **import** declarative\_base

Base = declarative\_base()

**class** **MyDataClass**(Base):

\_\_tablename\_\_ = 'my\_data'

id = Column(Integer, primary\_key=**True**)

data = Column(MutableDict.as\_mutable(JSONEncodedDict))

Any in-place changes to the MyDataClass.data member will flag the attribute as "dirty" on the parent object:

对MyDataClass.data成员的任何就地更改将在父对象上将属性标记为“dirty”：

**>>> from** **sqlalchemy.orm** **import** Session

**>>>** sess = Session()

**>>>** m1 = MyDataClass(data={'value1':'foo'})

**>>>** sess.add(m1)

**>>>** sess.commit()

**>>>** m1.data['value1'] = 'bar'

**>>> assert** m1 **in** sess.dirty

True

The MutableDict can be associated with all future instances of JSONEncodedDict in one step, using [associate\_with()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.Mutable.associate_with" \o "sqlalchemy.ext.mutable.Mutable.associate_with). This is similar to [as\_mutable()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.Mutable.as_mutable" \o "sqlalchemy.ext.mutable.Mutable.as_mutable)except it will intercept all occurrences of MutableDict in all mappings unconditionally, without the need to declare it individually:

可以使用[associate\_with()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.Mutable.associate_with" \o "sqlalchemy.ext.mutable.Mutable.associate_with)在一个步骤中将MutableDict与所有将来的JSONEncodedDict实例相关联。 这类似于[as\_mutable()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.Mutable.as_mutable" \o "sqlalchemy.ext.mutable.Mutable.as_mutable)，除了它将无条件地拦截所有映射中所有出现的MutableDict，而不需要单独声明它：

MutableDict.associate\_with(JSONEncodedDict)

**class** **MyDataClass**(Base):

\_\_tablename\_\_ = 'my\_data'

id = Column(Integer, primary\_key=**True**)

data = Column(JSONEncodedDict)

### Supporting Pickling

The key to the [sqlalchemy.ext.mutable](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "module-sqlalchemy.ext.mutable" \o "sqlalchemy.ext.mutable) extension relies upon the placement of a weakref.WeakKeyDictionary upon the value object, which stores a mapping of parent mapped objects keyed to the attribute name under which they are associated with this value. WeakKeyDictionary objects are not picklable, due to the fact that they contain weakrefs and function callbacks. In our case, this is a good thing, since if this dictionary were picklable, it could lead to an excessively large pickle size for our value objects that are pickled by themselves outside of the context of the parent. The developer responsibility here is only to provide a \_\_getstate\_\_ method that excludes the [\_parents()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableBase._parents" \o "sqlalchemy.ext.mutable.MutableBase._parents) collection from the pickle stream:

[sqlalchemy.ext.mutable](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "module-sqlalchemy.ext.mutable" \o "sqlalchemy.ext.mutable)扩展的关键依赖于将weakref.WeakKeyDictionary放置在值对象上，该对象存储一个父对象对象的映射，该对象映射对象与该值相关联的属性名称。 WeakKeyDictionary对象不可挑剔，因为它们包含weakrefs和函数回调。 在我们的情况下，这是一件好事，因为如果这本字典是可拾取的，那么对于我们的价值对象来说，它可能会导致一个过大的pickle大小，它们在父母背景之外被自己腌制。 这里的开发人员责任只是提供一个\_\_getstate\_\_方法，排除pickle流中的[\_parents()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableBase._parents" \o "sqlalchemy.ext.mutable.MutableBase._parents)集合：

**class** **MyMutableType**(Mutable):

**def** \_\_getstate\_\_(self):

d = self.\_\_dict\_\_.copy()

d.pop('\_parents', **None**)

**return** d

With our dictionary example, we need to return the contents of the dict itself (and also restore them on \_\_setstate\_\_):

使用我们的字典示例，我们需要返回dict本身的内容(也可以在\_\_setstate\_\_上还原它们)：

**class** **MutableDict**(Mutable, dict):

*# ....*

**def** \_\_getstate\_\_(self):

**return** dict(self)

**def** \_\_setstate\_\_(self, state):

self.update(state)

In the case that our mutable value object is pickled as it is attached to one or more parent objects that are also part of the pickle, the [Mutable](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.Mutable" \o "sqlalchemy.ext.mutable.Mutable) mixin will re-establish the [Mutable.\_parents](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.Mutable._parents" \o "sqlalchemy.ext.mutable.Mutable._parents) collection on each value object as the owning parents themselves are unpickled.

在我们的可变值对象被连接到一个或多个也是pickle的一部分的父对象的情况下，[Mutable](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.Mutable" \o "sqlalchemy.ext.mutable.Mutable) mixin将重新建立每个值对象上的[Mutable.\_parents](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.Mutable._parents" \o "sqlalchemy.ext.mutable.Mutable._parents)集合，因为所有父对象本身都是拆封。

## Establishing Mutability on Composites

Composites are a special ORM feature which allow a single scalar attribute to be assigned an object value which represents information "composed" from one or more columns from the underlying mapped table. The usual example is that of a geometric "point", and is introduced in [Composite Column Types](http://docs.sqlalchemy.org/en/rel_1_1/orm/composites.html" \l "mapper-composite).

复合材料是一种特殊的ORM特征，允许为单个标量属性分配一个对象值，该对象值表示从底层映射表的一个或多个列组成的信息。通常的例子是几何“点”，并在“复合列类型”中引入。

*Changed in version 0.7:*The internals of [orm.composite()](http://docs.sqlalchemy.org/en/rel_1_1/orm/composites.html" \l "sqlalchemy.orm.composite" \o "sqlalchemy.orm.composite) have been greatly simplified and in-place mutation detection is no longer enabled by default; instead, the user-defined value must detect changes on its own and propagate them to all owning parents. The [sqlalchemy.ext.mutable](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "module-sqlalchemy.ext.mutable" \o "sqlalchemy.ext.mutable) extension provides the helper class [MutableComposite](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableComposite" \o "sqlalchemy.ext.mutable.MutableComposite), which is a slight variant on the [Mutable](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.Mutable" \o "sqlalchemy.ext.mutable.Mutable) class.

版本0.7更改：orm.composite()的内部版本已被大大简化，默认情况下不再启用就地突变检测;相反，用户定义的值必须自己检测更改，并将其传播给所有拥有的父母。 sqlalchemy.ext.mutable扩展提供了类MutableComposite，这是Mutable类的一个轻微的变体。

As is the case with [Mutable](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.Mutable" \o "sqlalchemy.ext.mutable.Mutable), the user-defined composite class subclasses [MutableComposite](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableComposite" \o "sqlalchemy.ext.mutable.MutableComposite) as a mixin, and detects and delivers change events to its parents via the [MutableComposite.changed()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableComposite.changed" \o "sqlalchemy.ext.mutable.MutableComposite.changed) method. In the case of a composite class, the detection is usually via the usage of Python descriptors (i.e. @property), or alternatively via the special Python method \_\_setattr\_\_(). Below we expand upon the Point class introduced in [Composite Column Types](http://docs.sqlalchemy.org/en/rel_1_1/orm/composites.html" \l "mapper-composite) to subclass [MutableComposite](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableComposite" \o "sqlalchemy.ext.mutable.MutableComposite) and to also route attribute set events via \_\_setattr\_\_ to the [MutableComposite.changed()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableComposite.changed" \o "sqlalchemy.ext.mutable.MutableComposite.changed) method:

与Mutable的情况一样，用户定义的复合类将MutableComposite作为mixin进行子类化，并通过MutableComposite.changed()方法检测并将更改事件传递给其父项。在复合类的情况下，检测通常是通过使用Python描述符(即@property)，或者通过特殊的Python方法\_\_setattr \_\_()来实现的。下面我们展开复合列类型中引入的Point类，以便将MutableComposite子类化，并通过\_\_setattr\_\_将属性集事件路由到MutableComposite.changed()方法：

**from** **sqlalchemy.ext.mutable** **import** MutableComposite

**class** **Point**(MutableComposite):

**def** \_\_init\_\_(self, x, y):

self.x = x

self.y = y

**def** \_\_setattr\_\_(self, key, value):

"Intercept set events"

*# set the attribute*

object.\_\_setattr\_\_(self, key, value)

*# alert all parents to the change*

self.changed()

**def** \_\_composite\_values\_\_(self):

**return** self.x, self.y

**def** \_\_eq\_\_(self, other):

**return** isinstance(other, Point) **and** \

other.x == self.x **and** \

other.y == self.y

**def** \_\_ne\_\_(self, other):

**return** **not** self.\_\_eq\_\_(other)

The [MutableComposite](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableComposite" \o "sqlalchemy.ext.mutable.MutableComposite) class uses a Python metaclass to automatically establish listeners for any usage of [orm.composite()](http://docs.sqlalchemy.org/en/rel_1_1/orm/composites.html" \l "sqlalchemy.orm.composite" \o "sqlalchemy.orm.composite) that specifies our Point type. Below, when Point is mapped to the Vertex class, listeners are established which will route change events from Point objects to each of the Vertex.start and Vertex.end attributes:

MutableComposite类使用Python元类自动建立监听器，以指定我们的Point类型的orm.composite()的任何用法。 下面，当Point映射到顶点类时，将建立侦听器，将将Point对象的更改事件路由到每个Vertex.start和Vertex.end属性：

**from** **sqlalchemy.orm** **import** composite, mapper**from** **sqlalchemy** **import** Table, Column

vertices = Table('vertices', metadata,

Column('id', Integer, primary\_key=**True**),

Column('x1', Integer),

Column('y1', Integer),

Column('x2', Integer),

Column('y2', Integer),

)

**class** **Vertex**(object):

**pass**

mapper(Vertex, vertices, properties={

'start': composite(Point, vertices.c.x1, vertices.c.y1),

'end': composite(Point, vertices.c.x2, vertices.c.y2)})

Any in-place changes to the Vertex.start or Vertex.end members will flag the attribute as "dirty" on the parent object:

Vertex.start或Vertex.end成员的任何就地更改将在父对象上将该属性标记为“dirty”：

**>>> from** **sqlalchemy.orm** **import** Session

**>>>** sess = Session()

**>>>** v1 = Vertex(start=Point(3, 4), end=Point(12, 15))

**>>>** sess.add(v1)

**>>>** sess.commit()

**>>>** v1.end.x = 8

**>>> assert** v1 **in** sess.dirtyTrue

### Coercing Mutable Composites

The [MutableBase.coerce()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableBase.coerce" \o "sqlalchemy.ext.mutable.MutableBase.coerce) method is also supported on composite types. In the case of [MutableComposite](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableComposite" \o "sqlalchemy.ext.mutable.MutableComposite), the [MutableBase.coerce()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableBase.coerce" \o "sqlalchemy.ext.mutable.MutableBase.coerce) method is only called for attribute set operations, not load operations. Overriding the [MutableBase.coerce()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableBase.coerce" \o "sqlalchemy.ext.mutable.MutableBase.coerce) method is essentially equivalent to using a [validates()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapped_attributes.html" \l "sqlalchemy.orm.validates" \o "sqlalchemy.orm.validates)validation routine for all attributes which make use of the custom composite type:

复合类型也支持[MutableBase.coerce()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableBase.coerce" \o "sqlalchemy.ext.mutable.MutableBase.coerce)方法。 在[MutableComposite](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableComposite" \o "sqlalchemy.ext.mutable.MutableComposite)的情况下，[MutableBase.coerce()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableBase.coerce" \o "sqlalchemy.ext.mutable.MutableBase.coerce)方法仅用于属性集操作，而不是加载操作。 覆盖[MutableBase.coerce()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableBase.coerce" \o "sqlalchemy.ext.mutable.MutableBase.coerce)方法基本上等同于对使用自定义复合类型的所有属性使用[validates()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapped_attributes.html" \l "sqlalchemy.orm.validates" \o "sqlalchemy.orm.validates)验证例程：

**class** **Point**(MutableComposite):

*# other Point methods*

*# ...*

**def** coerce(cls, key, value):

**if** isinstance(value, tuple):

value = Point(\*value)

**elif** **not** isinstance(value, Point):

**raise** ValueError("tuple or Point expected")

**return** value

*New in version 0.7.10,0.8.0b2:*Support for the [MutableBase.coerce()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableBase.coerce" \o "sqlalchemy.ext.mutable.MutableBase.coerce) method in conjunction with objects of type [MutableComposite](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableComposite" \o "sqlalchemy.ext.mutable.MutableComposite).

### Supporting Pickling

As is the case with [Mutable](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.Mutable" \o "sqlalchemy.ext.mutable.Mutable), the [MutableComposite](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableComposite" \o "sqlalchemy.ext.mutable.MutableComposite) helper class uses a weakref.WeakKeyDictionary available via the [MutableBase.\_parents()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableBase._parents" \o "sqlalchemy.ext.mutable.MutableBase._parents)attribute which isn't picklable. If we need to pickle instances of Point or its owning class Vertex, we at least need to define a \_\_getstate\_\_ that doesn't include the \_parents dictionary. Below we define both a \_\_getstate\_\_ and a \_\_setstate\_\_ that package up the minimal form of our Point class:

与[Mutable](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.Mutable" \o "sqlalchemy.ext.mutable.Mutable)的情况一样，[MutableComposite](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableComposite" \o "sqlalchemy.ext.mutable.MutableComposite)帮助器类使用一个weakref.WeakKeyDictionary，可通过MutableBase.\_parents()属性获得，该属性不可取。 如果我们需要挑选Point或其所属类Vertex的实例，我们至少需要定义一个不包含\_parents字典的\_\_getstate\_\_。 下面我们定义一个\_\_getstate\_\_和一个\_\_setstate\_\_来打包我们的Point类的最小形式：

**class** **Point**(MutableComposite):

*# ...*

**def** \_\_getstate\_\_(self):

**return** self.x, self.y

**def** \_\_setstate\_\_(self, state):

self.x, self.y = state

As with [Mutable](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.Mutable" \o "sqlalchemy.ext.mutable.Mutable), the [MutableComposite](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableComposite" \o "sqlalchemy.ext.mutable.MutableComposite) augments the pickling process of the parent's object-relational state so that the [MutableBase.\_parents()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableBase._parents" \o "sqlalchemy.ext.mutable.MutableBase._parents)collection is restored to all Point objects.

与[Mutable](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.Mutable" \o "sqlalchemy.ext.mutable.Mutable)一样，[MutableComposite](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableComposite" \o "sqlalchemy.ext.mutable.MutableComposite)会增加父对象关系状态的酸洗过程，以便将[MutableBase.\_parents()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableBase._parents" \o "sqlalchemy.ext.mutable.MutableBase._parents)集合还原到所有Point对象。

## API Reference

*class*sqlalchemy.ext.mutable.**MutableBase**

Common base class to [Mutable](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.Mutable" \o "sqlalchemy.ext.mutable.Mutable) and [MutableComposite](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableComposite" \o "sqlalchemy.ext.mutable.MutableComposite).

Mutable和MutableComposite的常用基类。

**\_parents**

Dictionary of parent object->attribute name on the parent.

父对象的字典 - >父属的属性名称。

This attribute is a so-called "memoized" property. It initializes itself with a new weakref.WeakKeyDictionary the first time it is accessed, returning the same object upon subsequent access.

这个属性是所谓的“记忆”属性。 它首次使用新的weakref.WeakKeyDictionary初始化自身，并在后续访问时返回相同的对象。

*classmethod***coerce**(*key*, *value*)

Given a value, coerce it into the target type.

给定一个值，将其强制为目标类型。

Can be overridden by custom subclasses to coerce incoming data into a particular type.

可以被自定义子类覆盖，以将输入的数据强制为特定类型。

By default, raises ValueError.

默认情况下，引发ValueError。

This method is called in different scenarios depending on if the parent class is of type [Mutable](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.Mutable" \o "sqlalchemy.ext.mutable.Mutable) or of type [MutableComposite](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableComposite" \o "sqlalchemy.ext.mutable.MutableComposite). In the case of the former, it is called for both attribute-set operations as well as during ORM loading operations. For the latter, it is only called during attribute-set operations; the mechanics of the [composite()](http://docs.sqlalchemy.org/en/rel_1_1/orm/composites.html" \l "sqlalchemy.orm.composite" \o "sqlalchemy.orm.composite) construct handle coercion during load operations.

根据父类是Mutable类型还是MutableComposite类型，在不同的方案中调用此方法。 在前者的情况下，在ORM加载操作期间也调用这两个属性集操作。 对于后者，仅在属性集操作期间调用; 复合()构造的力学在加载操作期间处理强制。

|  |  |
| --- | --- |
| **Parameters:** | * ****key**** – string name of the ORM-mapped attribute being set. * ****value**** – the incoming value. |
| **Returns:** | the method should return the coerced value, or raise ValueError if the coercion cannot be completed.  该方法应返回强制值，如果强制无法完成，则会引发ValueError。 |

*class*sqlalchemy.ext.mutable.**Mutable**

Bases: [sqlalchemy.ext.mutable.MutableBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableBase" \o "sqlalchemy.ext.mutable.MutableBase)

Mixin that defines transparent propagation of change events to a parent object.

See the example in [Establishing Mutability on Scalar Column Values](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "mutable-scalars) for usage information.

将更改事件的透明传播定义为父对象的Mixin。

有关使用信息，请参阅建立标量列变量的示例。

**\_\_init\_\_**

*inherited from the* \_\_init\_\_ *attribute of* object

x.\_\_init\_\_(…) initializes x; see help(type(x)) for signature

**\_get\_listen\_keys**(*attribute*)

*inherited from the* \_get\_listen\_keys() *method of* [MutableBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableBase" \o "sqlalchemy.ext.mutable.MutableBase)

Given a descriptor attribute, return a set() of the attribute keys which indicate a change in the state of this attribute.

给定一个描述符属性，返回一个属性键的set()，表示该属性状态的变化。

This is normally just set([attribute.key]), but can be overridden to provide for additional keys. E.g. a [MutableComposite](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableComposite" \o "sqlalchemy.ext.mutable.MutableComposite) augments this set with the attribute keys associated with the columns that comprise the composite value.

这通常只是设置([attribute.key])，但可以覆盖以提供额外的键。 例如。 mutableComposite使用与构成复合值的列相关联的属性键来增加此集合。

This collection is consulted in the case of intercepting the [InstanceEvents.refresh()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.InstanceEvents.refresh" \o "sqlalchemy.orm.events.InstanceEvents.refresh) and [InstanceEvents.refresh\_flush()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.InstanceEvents.refresh_flush" \o "sqlalchemy.orm.events.InstanceEvents.refresh_flush) events, which pass along a list of attribute names that have been refreshed; the list is compared against this set to determine if action needs to be taken.

在截取InstanceEvents.refresh()和InstanceEvents.refresh\_flush()事件的情况下，将查询此集合，该事件会传递已刷新的属性名称列表; 将该列表与该集合进行比较以确定是否需要采取行动。

*New in version 1.0.5.*

**\_listen\_on\_attribute**(*attribute*, *coerce*, *parent\_cls*)

*inherited from the* \_listen\_on\_attribute() *method of* [MutableBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableBase" \o "sqlalchemy.ext.mutable.MutableBase)

Establish this type as a mutation listener for the given mapped descriptor.

建立此类型作为给定映射描述符的突变侦听器。

**\_parents**

*inherited from the* [\_parents](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableBase._parents" \o "sqlalchemy.ext.mutable.MutableBase._parents) *attribute of* [MutableBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableBase" \o "sqlalchemy.ext.mutable.MutableBase)

Dictionary of parent object->attribute name on the parent.

父对象的字典 - >父属的属性名称。

This attribute is a so-called "memoized" property. It initializes itself with a new weakref.WeakKeyDictionary the first time it is accessed, returning the same object upon subsequent access.

这个属性是所谓的“记忆”属性。 它首次使用新的weakref.WeakKeyDictionary初始化自身，并在后续访问时返回相同的对象。

*classmethod***as\_mutable**(*sqltype*)

Associate a SQL type with this mutable Python type.

将SQL类型与此可变的Python类型相关联。

This establishes listeners that will detect ORM mappings against the given type, adding mutation event trackers to those mappings.

这将建立将根据给定类型检测ORM映射的侦听器，将突变事件跟踪器添加到这些映射。

The type is returned, unconditionally as an instance, so that [as\_mutable()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.Mutable.as_mutable" \o "sqlalchemy.ext.mutable.Mutable.as_mutable) can be used inline:

该类型作为一个实例无条件地返回，因此as\_mutable()可以使用inline：

Table('mytable', metadata,

Column('id', Integer, primary\_key=**True**),

Column('data', MyMutableType.as\_mutable(PickleType)))

Note that the returned type is always an instance, even if a class is given, and that only columns which are declared specifically with that type instance receive additional instrumentation.

请注意，返回的类型始终是一个实例，即使给定了一个类，并且只有与该类型实例特定声明的列才会接收到其他工具。

To associate a particular mutable type with all occurrences of a particular type, use the [Mutable.associate\_with()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.Mutable.associate_with" \o "sqlalchemy.ext.mutable.Mutable.associate_with) classmethod of the particular [Mutable](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.Mutable" \o "sqlalchemy.ext.mutable.Mutable) subclass to establish a global association.

要将特定可变类型与特定类型的所有出现相关联，请使用特定Mutable子类的Mutable.associate\_with()类方法来建立全局关联。

**Warning**

The listeners established by this method are *global* to all mappers, and are *not* garbage collected. Only use [as\_mutable()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.Mutable.as_mutable" \o "sqlalchemy.ext.mutable.Mutable.as_mutable) for types that are permanent to an application, not with ad-hoc types else this will cause unbounded growth in memory usage.

通过此方法建立的听众对所有映射器是全局的，不是垃圾回收。 只对应用程序永久性的类型使用as\_mutable()，而不是ad-hoc类型，否则这将导致内存使用率无限增长。

*classmethod***associate\_with**(*sqltype*)

Associate this wrapper with all future mapped columns of the given type.

将此包装器与给定类型的所有未来映射列相关联。

This is a convenience method that calls associate\_with\_attribute automatically.

这是一个方便的方法，自动调用associate\_with\_attribute。

**Warning**

The listeners established by this method are *global* to all mappers, and are *not* garbage collected. Only use [associate\_with()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.Mutable.associate_with" \o "sqlalchemy.ext.mutable.Mutable.associate_with) for types that are permanent to an application, not with ad-hoc types else this will cause unbounded growth in memory usage.

通过此方法建立的听众对所有映射器是全局的，不是垃圾回收。 只有对于应用程序永久性的类型使用associate\_with()，而不是ad-hoc类型，否则这将导致内存使用率无限增长。

*classmethod***associate\_with\_attribute**(*attribute*)

Establish this type as a mutation listener for the given mapped descriptor.

建立此类型作为给定映射描述符的突变监听器。

**changed**()

Subclasses should call this method whenever change events occur.

每当发生更改事件时，子类应调用此方法。

**coerce**(*key*, *value*)

*inherited from the* [coerce()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableBase.coerce" \o "sqlalchemy.ext.mutable.MutableBase.coerce) *method of* [MutableBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableBase" \o "sqlalchemy.ext.mutable.MutableBase)

Given a value, coerce it into the target type.

给定一个值，将其强制为目标类型。

Can be overridden by custom subclasses to coerce incoming data into a particular type.

可以被自定义子类覆盖，以将输入的数据强制为特定类型。

By default, raises ValueError.

默认情况下，引发ValueError。

This method is called in different scenarios depending on if the parent class is of type [Mutable](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.Mutable" \o "sqlalchemy.ext.mutable.Mutable) or of type [MutableComposite](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableComposite" \o "sqlalchemy.ext.mutable.MutableComposite). In the case of the former, it is called for both attribute-set operations as well as during ORM loading operations. For the latter, it is only called during attribute-set operations; the mechanics of the [composite()](http://docs.sqlalchemy.org/en/rel_1_1/orm/composites.html" \l "sqlalchemy.orm.composite" \o "sqlalchemy.orm.composite) construct handle coercion during load operations.

根据父类是Mutable类型还是MutableComposite类型，在不同的方案中调用此方法。 在前者的情况下，在ORM加载操作期间也调用这两个属性集操作。 对于后者，仅在属性集操作期间调用; 复合()构造的力学在加载操作期间处理强制。

|  |  |
| --- | --- |
| **Parameters:** | * ****key**** – string name of the ORM-mapped attribute being set. * ****value**** – the incoming value. |
| **Returns:** | the method should return the coerced value, or raise ValueError if the coercion cannot be completed. |

*class*sqlalchemy.ext.mutable.**MutableComposite**

Bases: [sqlalchemy.ext.mutable.MutableBase](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableBase" \o "sqlalchemy.ext.mutable.MutableBase)

Mixin that defines transparent propagation of change events on a SQLAlchemy "composite" object to its owning parent or parents.

Mixin将SQLAlchemy“复合”对象上的更改事件的透明传播定义为其父或父。

See the example in [Establishing Mutability on Composites](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "mutable-composites) for usage information.

有关使用信息，请参阅建立复合材料的可变性的示例。

**changed**()

Subclasses should call this method whenever change events occur.

*class*sqlalchemy.ext.mutable.**MutableDict**

Bases: [sqlalchemy.ext.mutable.Mutable](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.Mutable" \o "sqlalchemy.ext.mutable.Mutable), \_\_builtin\_\_.dict

A dictionary type that implements [Mutable](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.Mutable" \o "sqlalchemy.ext.mutable.Mutable).

实现Mutable的字典类型。

The [MutableDict](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableDict" \o "sqlalchemy.ext.mutable.MutableDict) object implements a dictionary that will emit change events to the underlying mapping when the contents of the dictionary are altered, including when values are added or removed.

MutableDict对象实现一个字典，当字典的内容被更改时，包括添加或删除值时，会将更改事件发送到底层映射。

Note that [MutableDict](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableDict" \o "sqlalchemy.ext.mutable.MutableDict) does ****not**** apply mutable tracking to the *values themselves* inside the dictionary. Therefore it is not a sufficient solution for the use case of tracking deep changes to a *recursive* dictionary structure, such as a JSON structure. To support this use case, build a subclass of [MutableDict](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableDict" \o "sqlalchemy.ext.mutable.MutableDict) that provides appropriate coersion to the values placed in the dictionary so that they too are "mutable", and emit events up to their parent structure.

请注意，MutableDict不会对字典中的值本身应用可变跟踪。 因此，对于追溯深度改变递归字典结构(如JSON结构)的用例来说，这不是一个充分的解决方案。 为了支持这种用例，构建一个MutableDict的子类，它为放置在字典中的值提供了适当的修饰，使得它们也是“可变的”，并且发出直到它们的父结构的事件。

*New in version 0.8.*

**See also**

[MutableList](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableList" \o "sqlalchemy.ext.mutable.MutableList)

[MutableSet](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableSet" \o "sqlalchemy.ext.mutable.MutableSet)

**clear**()

*classmethod***coerce**(*key*, *value*)

Convert plain dictionary to instance of this class.

**pop**(*\*arg*)

**popitem**()

**setdefault**(*key*, *value*)

**update**(*\*a*, *\*\*kw*)

*class*sqlalchemy.ext.mutable.**MutableList**

Bases: [sqlalchemy.ext.mutable.Mutable](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.Mutable" \o "sqlalchemy.ext.mutable.Mutable), \_\_builtin\_\_.list

A list type that implements [Mutable](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.Mutable" \o "sqlalchemy.ext.mutable.Mutable).

实现Mutable的列表类型。

The [MutableList](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableList" \o "sqlalchemy.ext.mutable.MutableList) object implements a list that will emit change events to the underlying mapping when the contents of the list are altered, including when values are added or removed.

MutableList对象实现一个列表，当列表的内容被更改时，包括添加或删除值时，会将更改事件发送到底层映射。

Note that [MutableList](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableList" \o "sqlalchemy.ext.mutable.MutableList) does ****not**** apply mutable tracking to the *values themselves* inside the list. Therefore it is not a sufficient solution for the use case of tracking deep changes to a *recursive* mutable structure, such as a JSON structure. To support this use case, build a subclass of [MutableList](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableList" \o "sqlalchemy.ext.mutable.MutableList) that provides appropriate coersion to the values placed in the dictionary so that they too are "mutable", and emit events up to their parent structure.

请注意，MutableList不会对列表中的值本身应用可变跟踪。 因此，跟踪对递归可变结构(例如JSON结构)的深度变化的用例并不是一个充分的解决方案。 为了支持这种用例，构建一个MutableList的子类，为字典中放置的值提供适当的泛型，以使它们也是“可变的”，并且发出直到它们的父结构的事件。

*New in version 1.1.*

**See also**

[MutableDict](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableDict" \o "sqlalchemy.ext.mutable.MutableDict)

[MutableSet](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableSet" \o "sqlalchemy.ext.mutable.MutableSet)

**append**(*x*)

**clear**()

*classmethod***coerce**(*index*, *value*)

Convert plain list to instance of this class.

**extend**(*x*)

**insert**(*i*, *x*)

**pop**(*\*arg*)

**remove**(*i*)

**reverse**()

**sort**()

*class*sqlalchemy.ext.mutable.**MutableSet**

Bases: [sqlalchemy.ext.mutable.Mutable](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.Mutable" \o "sqlalchemy.ext.mutable.Mutable), \_\_builtin\_\_.set

A set type that implements [Mutable](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.Mutable" \o "sqlalchemy.ext.mutable.Mutable).

一种实现Mutable的集合类型。

The [MutableSet](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableSet" \o "sqlalchemy.ext.mutable.MutableSet) object implements a set that will emit change events to the underlying mapping when the contents of the set are altered, including when values are added or removed.

MutableSet对象实现一个集合，当集合的内容被更改时，包括添加或删除值时，将发送更改事件到底层映射的集合。

Note that [MutableSet](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableSet" \o "sqlalchemy.ext.mutable.MutableSet) does ****not**** apply mutable tracking to the *values themselves* inside the set. Therefore it is not a sufficient solution for the use case of tracking deep changes to a *recursive* mutable structure. To support this use case, build a subclass of [MutableSet](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableSet" \o "sqlalchemy.ext.mutable.MutableSet) that provides appropriate coersion to the values placed in the dictionary so that they too are "mutable", and emit events up to their parent structure.

请注意，MutableSet不会对集合中的值本身应用可变跟踪。 因此，跟踪递归可变结构深度变化的用例并不是一个充分的解决方案。 为了支持这种用例，构建一个MutableSet的子类，它为放置在字典中的值提供了适当的泛型，使得它们也是“可变的”，并且发出直到它们的父结构的事件。

*New in version 1.1.*

**See also**

[MutableDict](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableDict" \o "sqlalchemy.ext.mutable.MutableDict)

[MutableList](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableList" \o "sqlalchemy.ext.mutable.MutableList)

**add**(*elem*)

**clear**()

*classmethod***coerce**(*index*, *value*)

Convert plain set to instance of this class.

**difference\_update**(*\*arg*)

**discard**(*elem*)

**intersection\_update**(*\*arg*)

**pop**(*\*arg*)

**remove**(*elem*)

**symmetric\_difference\_update**(*\*arg*)

**update**(*\*arg*)

# **7.6 Ordering List**

A custom list that manages index/position information for contained elements.

|  |  |
| --- | --- |
| **author:** | Jason Kirtland |

orderinglist is a helper for mutable ordered relationships. It will intercept list operations performed on a [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)-managed collection and automatically synchronize changes in list position onto a target scalar attribute.

订购单是可变订单关系的帮手。 它将拦截对关系()管理集合执行的列表操作，并自动将列表位置的更改同步到目标标量属性。

Example: A slide table, where each row refers to zero or more entries in a related bullet table. The bullets within a slide are displayed in order based on the value of the position column in the bullet table. As entries are reordered in memory, the value of the position attribute should be updated to reflect the new sort order:

示例：幻灯片表，其中每行指向相关项目符号表中的零个或多个条目。 幻灯片中的项目符号将根据项目符号表中位置列的值顺序显示。 当条目在内存中重新排序时，应该更新position属性的值以反映新的排序顺序：

Base = declarative\_base()

**class** **Slide**(Base):

\_\_tablename\_\_ = 'slide'

id = Column(Integer, primary\_key=**True**)

name = Column(String)

bullets = relationship("Bullet", order\_by="Bullet.position")

**class** **Bullet**(Base):

\_\_tablename\_\_ = 'bullet'

id = Column(Integer, primary\_key=**True**)

slide\_id = Column(Integer, ForeignKey('slide.id'))

position = Column(Integer)

text = Column(String)

The standard relationship mapping will produce a list-like attribute on each Slide containing all related Bullet objects, but coping with changes in ordering is not handled automatically. When appending a Bullet into Slide.bullets, the Bullet.position attribute will remain unset until manually assigned. When the Bullet is inserted into the middle of the list, the following Bullet objects will also need to be renumbered.

标准关系映射将在每个包含所有相关“项目符号”对象的“幻灯片”中生成类似列表的属性，但不会自动处理排序中的更改。 当将Bullet附加到Slide.bullets中时，Bullet.position属性将保持未设置，直到手动分配。 当子弹插入列表的中间时，以下项目符号对象也需要重新编号。

The [OrderingList](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/orderinglist.html" \l "sqlalchemy.ext.orderinglist.OrderingList" \o "sqlalchemy.ext.orderinglist.OrderingList) object automates this task, managing the position attribute on all Bullet objects in the collection. It is constructed using the [ordering\_list()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/orderinglist.html" \l "sqlalchemy.ext.orderinglist.ordering_list" \o "sqlalchemy.ext.orderinglist.ordering_list) factory:

OrderingList对象自动执行此任务，管理集合中所有Bullet对象上的position属性。 它使用ordering\_list()工厂构建：

**from** **sqlalchemy.ext.orderinglist** **import** ordering\_list

Base = declarative\_base()

**class** **Slide**(Base):

\_\_tablename\_\_ = 'slide'

id = Column(Integer, primary\_key=**True**)

name = Column(String)

bullets = relationship("Bullet", order\_by="Bullet.position",

collection\_class=ordering\_list('position'))

**class** **Bullet**(Base):

\_\_tablename\_\_ = 'bullet'

id = Column(Integer, primary\_key=**True**)

slide\_id = Column(Integer, ForeignKey('slide.id'))

position = Column(Integer)

text = Column(String)

With the above mapping the Bullet.position attribute is managed:

通过上述映射，管理Bullet.position属性：

s = Slide()s.bullets.append(Bullet())

s.bullets.append(Bullet())

s.bullets[1].position

>>> 1

s.bullets.insert(1, Bullet())s.bullets[2].position

>>> 2

The [OrderingList](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/orderinglist.html" \l "sqlalchemy.ext.orderinglist.OrderingList" \o "sqlalchemy.ext.orderinglist.OrderingList) construct only works with ****changes**** to a collection, and not the initial load from the database, and requires that the list be sorted when loaded. Therefore, be sure to specify order\_by on the [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) against the target ordering attribute, so that the ordering is correct when first loaded.

[OrderingList](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/orderinglist.html" \l "sqlalchemy.ext.orderinglist.OrderingList" \o "sqlalchemy.ext.orderinglist.OrderingList)结构仅适用于对集合的更改，而不适用于数据库的初始加载，并且要求在加载时对列表进行排序。 因此，请确保在与目标排序属性的[relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship)上指定order\_by，以便在首次加载时排序正确。

**Warning**

[OrderingList](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/orderinglist.html" \l "sqlalchemy.ext.orderinglist.OrderingList" \o "sqlalchemy.ext.orderinglist.OrderingList) only provides limited functionality when a primary key column or unique column is the target of the sort. Operations that are unsupported or are problematic include:

当主键列或唯一列是排序的目标时，OrderingList仅提供有限的功能。 不支持或有问题的操作包括：

two entries must trade values. This is not supported directly in the case of a primary key or unique constraint because it means at least one row would need to be temporarily removed first, or changed to a third, neutral value while the switch occurs.两个条目必须交易价值。 在主键或唯一约束的情况下不直接支持这一点，因为这意味着首先需要暂时删除至少一行，或者在交换机发生时更改为第三个中性值。

an entry must be deleted in order to make room for a new entry. SQLAlchemy's unit of work performs all INSERTs before DELETEs within a single flush. In the case of a primary key, it will trade an INSERT/DELETE of the same primary key for an UPDATE statement in order to lessen the impact of this limitation, however this does not take place for a UNIQUE column. A future feature will allow the "DELETE before INSERT" behavior to be possible, allevating this limitation, though this feature will require explicit configuration at the mapper level for sets of columns that are to be handled in this way.必须删除一个条目，以便为新条目腾出空间。 SQLAlchemy的工作单元在单次刷新之前执行DELETE之前的所有INSERT。 在主键的情况下，它将交换一个UPDATE语句的相同主键的INSERT / DELETE，以减轻此限制的影响，但这不会发生在UNIQUE列。 未来的功能将允许“删除之前的INSERT”行为是可行的，减轻了这一限制，尽管此功能将需要在映射器级别显式配置要以这种方式处理的列集。

[ordering\_list()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/orderinglist.html" \l "sqlalchemy.ext.orderinglist.ordering_list" \o "sqlalchemy.ext.orderinglist.ordering_list) takes the name of the related object's ordering attribute as an argument. By default, the zero-based integer index of the object's position in the [ordering\_list()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/orderinglist.html" \l "sqlalchemy.ext.orderinglist.ordering_list" \o "sqlalchemy.ext.orderinglist.ordering_list) is synchronized with the ordering attribute: index 0 will get position 0, index 1 position 1, etc. To start numbering at 1 or some other integer, provide count\_from=1.

ordering\_list()将相关对象的排序属性的名称作为参数。 默认情况下，在sequence\_list()中对象的位置的零基整数索引与排序属性同步：索引0将获取位置0，索引1位置1等。要开始编号为1或其他整数，请提供count\_from= 1。

## API Reference

sqlalchemy.ext.orderinglist.**ordering\_list**(*attr*, *count\_from=None*, *\*\*kw*)

Prepares an [OrderingList](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/orderinglist.html" \l "sqlalchemy.ext.orderinglist.OrderingList" \o "sqlalchemy.ext.orderinglist.OrderingList) factory for use in mapper definitions.

Returns an object suitable for use as an argument to a Mapper relationship's collection\_class option. e.g.:

准备一个用于映射器定义的OrderingList工厂。

返回适合用作Mapper关系的collection\_class选项的参数的对象。 例如。：

**from** **sqlalchemy.ext.orderinglist** **import** ordering\_list

**class** **Slide**(Base):

\_\_tablename\_\_ = 'slide'

id = Column(Integer, primary\_key=**True**)

name = Column(String)

bullets = relationship("Bullet", order\_by="Bullet.position",

collection\_class=ordering\_list('position'))

|  |  |
| --- | --- |
| **Parameters:** | * ****attr**** – Name of the mapped attribute to use for storage and retrieval of ordering information * ****count\_from**** – Set up an integer-based ordering, starting at count\_from. For example, ordering\_list('pos', count\_from=1) would create a 1-based list in SQL, storing the value in the 'pos' column. Ignored if ordering\_func is supplied. |

Additional arguments are passed to the [OrderingList](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/orderinglist.html" \l "sqlalchemy.ext.orderinglist.OrderingList" \o "sqlalchemy.ext.orderinglist.OrderingList) constructor.

sqlalchemy.ext.orderinglist.**count\_from\_0**(*index*, *collection*)

Numbering function: consecutive integers starting at 0.

sqlalchemy.ext.orderinglist.**count\_from\_1**(*index*, *collection*)

Numbering function: consecutive integers starting at 1.

sqlalchemy.ext.orderinglist.**count\_from\_n\_factory**(*start*)

Numbering function: consecutive integers starting at arbitrary start.

*class*sqlalchemy.ext.orderinglist.**OrderingList**(*ordering\_attr=None*, *ordering\_func=None*, *reorder\_on\_append=False*)

Bases: \_\_builtin\_\_.list

A custom list that manages position information for its children.

The [OrderingList](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/orderinglist.html" \l "sqlalchemy.ext.orderinglist.OrderingList" \o "sqlalchemy.ext.orderinglist.OrderingList) object is normally set up using the [ordering\_list()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/orderinglist.html" \l "sqlalchemy.ext.orderinglist.ordering_list" \o "sqlalchemy.ext.orderinglist.ordering_list) factory function, used in conjunction with the [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) function.

**\_\_init\_\_**(*ordering\_attr=None*, *ordering\_func=None*, *reorder\_on\_append=False*)

A custom list that manages position information for its children.

OrderingList is a collection\_class list implementation that syncs position in a Python list with a position attribute on the mapped objects.

This implementation relies on the list starting in the proper order, so be ****sure**** to put an order\_by on your relationship.

|  |  |
| --- | --- |
| **Parameters:** | * ****ordering\_attr**** – Name of the attribute that stores the object's order in the relationship. * ****ordering\_func –****Optional. A function that maps the position in the Python list to a value to store in the ordering\_attr. Values returned are usually (but need not be!) integers.可选的。 将Python列表中的位置映射到要存储在ordering\_attr中的值的函数。 返回的值通常是(但不一定是！)整数。   An ordering\_func is called with two positional parameters: the index of the element in the list, and the list itself.使用两个位置参数调用ordered\_func：列表中元素的索引和列表本身。  If omitted, Python list indexes are used for the attribute values. Two basic pre-built numbering functions are provided in this module: count\_from\_0and count\_from\_1. For more exotic examples like stepped numbering, alphabetical and Fibonacci numbering, see the unit tests.如果省略，则使用Python列表索引作为属性值。 本模块提供了两个基本的预构建编号功能：count\_from\_0和count\_from\_1。 对于更多奇异的例子，如分步编号，字母和斐波纳契编号，请参阅单元测试。   * ****reorder\_on\_append –****Default False. When appending an object with an existing (non-None) ordering value, that value will be left untouched unless reorder\_on\_append is true. This is an optimization to avoid a variety of dangerous unexpected database writes.默认为假。 当使用现有(非无)排序值附加对象时，该值将保持不变，除非reorder\_on\_append为真。 这是一个优化，以避免各种危险的意外数据库写入。   SQLAlchemy will add instances to the list via append() when your object loads. If for some reason the result set from the database skips a step in the ordering (say, row '1' is missing but you get '2', '3', and '4'), reorder\_on\_append=True would immediately renumber the items to '1', '2', '3'. If you have multiple sessions making changes, any of whom happen to load this collection even in passing, all of the sessions would try to "clean up" the numbering in their commits, possibly causing all but one to fail with a concurrent modification error.SQLAlchemy将在对象加载时通过append()向列表中添加实例。 如果由于某种原因，数据库中的结果设置会跳过排序的一个步骤(比如说，行'1'丢失但是你得到'2'，'3'和'4')，reorder\_on\_append = True会立即重新编号项 到'1'，'2'，'3'。 如果您有多个会话进行更改，其中任何一个偶尔会加载此集合，即使传递，所有会话都将尝试“清理”其提交中的编号，可能会导致所有但失败并发修改错误。  Recommend leaving this with the default of False, and just call reorder() if you're doing append() operations with previously ordered instances or when doing some housekeeping after manual sql operations.建议将默认值设置为False，如果您对以前订购的实例进行了append()操作，或者在手动执行sql操作后进行一些内部处理，则只需调用reorder()。 |

**append**(*entity*)

L.append(object) – append object to end

**insert**(*index*, *entity*)

L.insert(index, object) – insert object before index

**pop**([*index*]) → item -- remove and return item at index (default last).

Raises IndexError if list is empty or index is out of range.

**remove**(*entity*)

L.remove(value) – remove first occurrence of value. Raises ValueError if the value is not present.

**reorder**()

Synchronize ordering for the entire collection.

Sweeps through the list and ensures that each object has accurate ordering information set.

# **7.7 Horizontal Sharding**

Horizontal sharding support.

水平分片支持。

Defines a rudimental 'horizontal sharding' system which allows a Session to distribute queries and persistence operations across multiple databases.

For a usage example, see the [Horizontal Sharding](http://docs.sqlalchemy.org/en/rel_1_1/orm/examples.html" \l "examples-sharding) example included in the source distribution.

定义一个初步的“横向分片”系统，允许会话在多个数据库之间分发查询和持久化操作。

有关使用示例，请参阅源分发中包含的“水平分片”示例。

## API Documentation

*class*sqlalchemy.ext.horizontal\_shard.**ShardedSession**(*shard\_chooser*, *id\_chooser*, *query\_chooser*, *shards=None*, *query\_cls=<class 'sqlalchemy.ext.horizontal\_shard.ShardedQuery'>*, *\*\*kwargs*)

Bases: [sqlalchemy.orm.session.Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session)

**\_\_init\_\_**(*shard\_chooser*, *id\_chooser*, *query\_chooser*, *shards=None*, *query\_cls=<class 'sqlalchemy.ext.horizontal\_shard.ShardedQuery'>*, *\*\*kwargs*)

Construct a ShardedSession.

|  |  |
| --- | --- |
| **Parameters:** | * ****shard\_chooser**** – A callable which, passed a Mapper, a mapped instance, and possibly a SQL clause, returns a shard ID. This id may be based off of the attributes present within the object, or on some round-robin scheme. If the scheme is based on a selection, it should set whatever state on the instance to mark it in the future as participating in that shard. * ****id\_chooser**** – A callable, passed a query and a tuple of identity values, which should return a list of shard ids where the ID might reside. The databases will be queried in the order of this listing. * ****query\_chooser**** – For a given Query, returns the list of shard\_ids where the query should be issued. Results from all shards returned will be combined together into a single listing. * ****shards**** – A dictionary of string shard names to [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) objects. |

*class*sqlalchemy.ext.horizontal\_shard.**ShardedQuery**(*\*args*, *\*\*kwargs*)

Bases: [sqlalchemy.orm.query.Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query)

**set\_shard**(*shard\_id*)

return a new query, limited to a single shard ID.

all subsequent operations with the returned query will be against the single shard regardless of other state.

# **7.8 Hybrid Attributes**

Define attributes on ORM-mapped classes that have "hybrid" behavior.

在具有“hybrid”行为的ORM映射类上定义属性。

"hybrid" means the attribute has distinct behaviors defined at the class level and at the instance level.

“hybrid”表示属性在类级别和实例级别上定义了不同的行为。

The [hybrid](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/hybrid.html" \l "module-sqlalchemy.ext.hybrid" \o "sqlalchemy.ext.hybrid) extension provides a special form of method decorator, is around 50 lines of code and has almost no dependencies on the rest of SQLAlchemy. It can, in theory, work with any descriptor-based expression system.

[hybrid](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/hybrid.html" \l "module-sqlalchemy.ext.hybrid" \o "sqlalchemy.ext.hybrid)扩展提供了一种特殊形式的方法装饰器，大约50行代码，几乎不依赖于SQLAlchemy的其余部分。 在理论上，它可以与任何基于描述符的表达式系统一起工作。

Consider a mapping Interval, representing integer start and end values. We can define higher level functions on mapped classes that produce SQL expressions at the class level, and Python expression evaluation at the instance level. Below, each function decorated with [hybrid\_method](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/hybrid.html" \l "sqlalchemy.ext.hybrid.hybrid_method" \o "sqlalchemy.ext.hybrid.hybrid_method) or [hybrid\_property](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/hybrid.html" \l "sqlalchemy.ext.hybrid.hybrid_property" \o "sqlalchemy.ext.hybrid.hybrid_property) may receive self as an instance of the class, or as the class itself:

考虑映射间隔，表示整数开始和结束值。 我们可以在类级别上生成SQL表达式的映射类定义更高级别的函数，在实例级别定义Python表达式求值。 下面，使用hybrid\_method或hybrid\_property装饰的每个函数可以接收self作为类的一个实例，或者作为类本身：

**from** **sqlalchemy** **import** Column, Integer

**from** **sqlalchemy.ext.declarative** **import** declarative\_base

**from** **sqlalchemy.orm** **import** Session, aliased

**from** **sqlalchemy.ext.hybrid** **import** hybrid\_property, hybrid\_method

Base = declarative\_base()

**class** **Interval**(Base):

\_\_tablename\_\_ = 'interval'

id = Column(Integer, primary\_key=**True**)

start = Column(Integer, nullable=**False**)

end = Column(Integer, nullable=**False**)

**def** \_\_init\_\_(self, start, end):

self.start = start

self.end = end

**@hybrid\_property**

**def** length(self):

**return** self.end - self.start

**@hybrid\_method**

**def** contains(self, point):

**return** (self.start <= point) & (point <= self.end)

**@hybrid\_method**

**def** intersects(self, other):

**return** self.contains(other.start) | self.contains(other.end)

Above, the length property returns the difference between the end and start attributes. With an instance of Interval, this subtraction occurs in Python, using normal Python descriptor mechanics:

以上，length属性返回end和start属性之间的差异。 使用Interval的实例，这种减法发生在Python中，使用普通的Python描述符机制：

**>>>** i1 = Interval(5, 10)

**>>>** i1.length

5

When dealing with the Interval class itself, the [hybrid\_property](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/hybrid.html" \l "sqlalchemy.ext.hybrid.hybrid_property" \o "sqlalchemy.ext.hybrid.hybrid_property) descriptor evaluates the function body given the Interval class as the argument, which when evaluated with SQLAlchemy expression mechanics returns a new SQL expression:

当处理Interval类本身时，hybrid\_property描述符会计算给定Interval类的函数体作为参数，当用SQLAlchemy表达式机制评估时返回一个新的SQL表达式：

**>>>** print Interval.length

interval."end" - interval.start

**>>>** print Session().query(Interval).filter(Interval.length > 10)

SELECT interval.id AS interval\_id, interval.start AS interval\_start,

interval."end" AS interval\_end

FROM interval

WHERE interval."end" - interval.start > :param\_1

ORM methods such as [filter\_by()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.filter_by" \o "sqlalchemy.orm.query.Query.filter_by) generally use getattr() to locate attributes, so can also be used with hybrid attributes:

filter\_by()等ORM方法通常使用getattr()来定位属性，所以也可以使用混合属性：

**>>>** print Session().query(Interval).filter\_by(length=5)

SELECT interval.id AS interval\_id, interval.start AS interval\_start,

interval."end" AS interval\_end

FROM interval

WHERE interval."end" - interval.start = :param\_1

The Interval class example also illustrates two methods, contains() and intersects(), decorated with [hybrid\_method](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/hybrid.html" \l "sqlalchemy.ext.hybrid.hybrid_method" \o "sqlalchemy.ext.hybrid.hybrid_method). This decorator applies the same idea to methods that [hybrid\_property](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/hybrid.html" \l "sqlalchemy.ext.hybrid.hybrid_property" \o "sqlalchemy.ext.hybrid.hybrid_property) applies to attributes. The methods return boolean values, and take advantage of the Python | and & bitwise operators to produce equivalent instance-level and SQL expression-level boolean behavior:

Interval类的例子也说明了使用hybrid\_method装饰的两个方法，contains()和intersects()。 这个装饰器将相同的想法应用于hybrid\_property应用于属性的方法。 该方法返回布尔值，并利用Python | 和按位运算符生成等效的实例级和SQL表达式布尔行为：

**>>>** i1.contains(6)

True

**>>>** i1.contains(15)

False

**>>>** i1.intersects(Interval(7, 18))

True

**>>>** i1.intersects(Interval(25, 29))

False

**>>>** print Session().query(Interval).filter(Interval.contains(15))

SELECT interval.id AS interval\_id, interval.start AS interval\_start,

interval."end" AS interval\_end

FROM interval

WHERE interval.start <= :start\_1 AND interval."end" > :end\_1

**>>>** ia = aliased(Interval)

**>>>** print Session().query(Interval, ia).filter(Interval.intersects(ia))

SELECT interval.id AS interval\_id, interval.start AS interval\_start,

interval."end" AS interval\_end, interval\_1.id AS interval\_1\_id,

interval\_1.start AS interval\_1\_start, interval\_1."end" AS interval\_1\_end

FROM interval, interval AS interval\_1

WHERE interval.start <= interval\_1.start

AND interval."end" > interval\_1.start

OR interval.start <= interval\_1."end"

AND interval."end" > interval\_1."end"

7.8.1 Defining Expression Behavior Distinct from Attribute Behavior

Our usage of the & and | bitwise operators above was fortunate, considering our functions operated on two boolean values to return a new one. In many cases, the construction of an in-Python function and a SQLAlchemy SQL expression have enough differences that two separate Python expressions should be defined. The[hybrid](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/hybrid.html" \l "module-sqlalchemy.ext.hybrid" \o "sqlalchemy.ext.hybrid) decorators define the [hybrid\_property.expression()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/hybrid.html" \l "sqlalchemy.ext.hybrid.hybrid_property.expression" \o "sqlalchemy.ext.hybrid.hybrid_property.expression) modifier for this purpose. As an example we'll define the radius of the interval, which requires the usage of the absolute value function:

我们使用＆和| 上面的位运算符是幸运的，考虑到我们的函数在两个布尔值上运行以返回一个新的。 在许多情况下，构建Python函数和SQLAlchemy SQL表达式有足够的差异，应该定义两个单独的Python表达式。 [hybrid](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/hybrid.html" \l "module-sqlalchemy.ext.hybrid" \o "sqlalchemy.ext.hybrid)装饰器为此定义了[hybrid\_property.expression()](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/hybrid.html" \l "sqlalchemy.ext.hybrid.hybrid_property.expression" \o "sqlalchemy.ext.hybrid.hybrid_property.expression)修饰符。 例如，我们将定义间隔的半径，这需要使用绝对值函数：

**from** **sqlalchemy** **import** func

**class** **Interval**(object):

*# ...*

**@hybrid\_property**

**def** radius(self):

**return** abs(self.length) / 2

**@radius**.expression

**def** radius(cls):

**return** func.abs(cls.length) / 2

Above the Python function abs() is used for instance-level operations, the SQL function ABS() is used via the [func](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.func" \o "sqlalchemy.sql.expression.func) object for class-level expressions:

在Python函数之上，abs()用于实例级操作，SQL函数ABS()通过用于类级表达式的func对象使用：

**>>>** i1.radius2

**>>>** print Session().query(Interval).filter(Interval.radius > 5)

SELECT interval.id AS interval\_id, interval.start AS interval\_start, interval."end" AS interval\_endFROM intervalWHERE abs(interval."end" - interval.start) / :abs\_1 > :param\_1

### 7.8.2 Defining Setters

Hybrid properties can also define setter methods. If we wanted length above, when set, to modify the endpoint value:

混合属性也可以定义setter方法。 如果我们想要上面的长度，当设置时，修改端点值：

**class** **Interval**(object):

*# ...*

**@hybrid\_property**

**def** length(self):

**return** self.end - self.start

**@length**.setter

**def** length(self, value):

self.end = self.start + value

The length(self, value) method is now called upon set:

长度(self，value)方法现在调用set：

**>>>** i1 = Interval(5, 10)

**>>>** i1.length5

**>>>** i1.length = 12

**>>>** i1.end

17

7.8.2 Working with Relationships

There's no essential difference when creating hybrids that work with related objects as opposed to column-based data. The need for distinct expressions tends to be greater. The two variants we'll illustrate are the "join-dependent" hybrid, and the "correlated subquery" hybrid.

创建使用相关对象而不是基于列的数据的混合动画没有本质区别。 对不同表达的需求往往更大。 我们将要说明的两个变体是“连接依赖”混合，以及“相关子查询”混合。

### Join-Dependent Relationship Hybrid

Consider the following declarative mapping which relates a User to a SavingsAccount:

考虑将用户与储蓄账户相关联的以下声明性映射：

**from** **sqlalchemy** **import** Column, Integer, ForeignKey, Numeric, String

**from** **sqlalchemy.orm** **import** relationship

**from** **sqlalchemy.ext.declarative** **import** declarative\_base

**from** **sqlalchemy.ext.hybrid** **import** hybrid\_property

Base = declarative\_base()

**class** **SavingsAccount**(Base):

\_\_tablename\_\_ = 'account'

id = Column(Integer, primary\_key=**True**)

user\_id = Column(Integer, ForeignKey('user.id'), nullable=**False**)

balance = Column(Numeric(15, 5))

**class** **User**(Base):

\_\_tablename\_\_ = 'user'

id = Column(Integer, primary\_key=**True**)

name = Column(String(100), nullable=**False**)

accounts = relationship("SavingsAccount", backref="owner")

**@hybrid\_property**

**def** balance(self):

**if** self.accounts:

**return** self.accounts[0].balance

**else**:

**return** **None**

**@balance**.setter

**def** balance(self, value):

**if** **not** self.accounts:

account = Account(owner=self)

**else**:

account = self.accounts[0]

account.balance = value

**@balance**.expression

**def** balance(cls):

**return** SavingsAccount.balance

The above hybrid property balance works with the first SavingsAccount entry in the list of accounts for this user. The in-Python getter/setter methods can treat accounts as a Python list available on self.

以上混合财产余额与该用户的帐户列表中的第一个SavingsAccount条目配合使用。 Python中的getter / setter方法可以将帐户视为可用于自己的Python列表。

However, at the expression level, it's expected that the User class will be used in an appropriate context such that an appropriate join to SavingsAccount will be present:

然而，在表达式级别，预计User类将在适当的上下文中使用，以便存在适用于SavingsAccount的连接：

**>>>** print Session().query(User, User.balance).\

**...**  join(User.accounts).filter(User.balance > 5000)

SELECT "user".id AS user\_id, "user".name AS user\_name,account.balance AS account\_balance

FROM "user" JOIN account ON "user".id = account.user\_id

WHERE account.balance > :balance\_1

Note however, that while the instance level accessors need to worry about whether self.accounts is even present, this issue expresses itself differently at the SQL expression level, where we basically would use an outer join:

但是请注意，虽然实例级别访问者需要担心self.accounts是否存在，但是在SQL表达式级别上，这个问题本身就有所不同，我们基本上将使用外部连接：

**>>> from** **sqlalchemy** **import** or\_

**>>>** print (Session().query(User, User.balance).outerjoin(User.accounts).

**...**  filter(or\_(User.balance < 5000, User.balance == **None**)))

SELECT "user".id AS user\_id, "user".name AS user\_name,account.balance AS account\_balance

FROM "user" LEFT OUTER JOIN account ON "user".id = account.user\_id

WHERE account.balance < :balance\_1 OR account.balance IS NULL

### Correlated Subquery Relationship Hybrid

We can, of course, forego being dependent on the enclosing query's usage of joins in favor of the correlated subquery, which can portably be packed into a single column expression. A correlated subquery is more portable, but often performs more poorly at the SQL level. Using the same technique illustrated at [Using column\_property](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapped_sql_expr.html" \l "mapper-column-property-sql-expressions), we can adjust our SavingsAccount example to aggregate the balances for *all* accounts, and use a correlated subquery for the column expression:

当然，我们可以放弃依靠封闭的查询使用连接来支持相关的子查询，该子查询可以被可移植地包装到单个列表达式中。 相关子查询更易于移植，但在SQL级别上往往执行得更差。 使用“columns\_property”所示的相同技术，我们可以调整我们的SavingsAccount示例来汇总所有帐户的余额，并对列表达式使用相关的子查询：

**from** **sqlalchemy** **import** Column, Integer, ForeignKey, Numeric, String

**from** **sqlalchemy.orm** **import** relationship

**from** **sqlalchemy.ext.declarative** **import** declarative\_base

**from** **sqlalchemy.ext.hybrid** **import** hybrid\_property

**from** **sqlalchemy** **import** select, func

Base = declarative\_base()

**class** **SavingsAccount**(Base):

\_\_tablename\_\_ = 'account'

id = Column(Integer, primary\_key=**True**)

user\_id = Column(Integer, ForeignKey('user.id'), nullable=**False**)

balance = Column(Numeric(15, 5))

**class** **User**(Base):

\_\_tablename\_\_ = 'user'

id = Column(Integer, primary\_key=**True**)

name = Column(String(100), nullable=**False**)

accounts = relationship("SavingsAccount", backref="owner")

**@hybrid\_property**

**def** balance(self):

**return** sum(acc.balance **for** acc **in** self.accounts)

**@balance**.expression

**def** balance(cls):

**return** select([func.sum(SavingsAccount.balance)]).\

where(SavingsAccount.user\_id==cls.id).\

label('total\_balance')

The above recipe will give us the balance column which renders a correlated SELECT:

**>>>** print s.query(User).filter(User.balance > 400)

SELECT "user".id AS user\_id, "user".name AS user\_name

FROM "user"

WHERE (SELECT sum(account.balance) AS sum\_1FROM account

WHERE account.user\_id = "user".id) > :param\_1

#### 7.8.4 Building Custom Comparators

The hybrid property also includes a helper that allows construction of custom comparators. A comparator object allows one to customize the behavior of each SQLAlchemy expression operator individually. They are useful when creating custom types that have some highly idiosyncratic behavior on the SQL side.

混合属性还包括一个可以构建定制比较器的帮助器。 比较对象允许单独定制每个SQLAlchemy表达式运算符的行为。 它们在创建在SQL端具有一些高度特殊行为的自定义类型时很有用。

The example class below allows case-insensitive comparisons on the attribute named word\_insensitive:

下面的示例类允许对名为word\_insensitive的属性进行不区分大小写的比较：

**from** **sqlalchemy.ext.hybrid** **import** Comparator, hybrid\_property

**from** **sqlalchemy** **import** func, Column, Integer, String

**from** **sqlalchemy.orm** **import** Session

**from** **sqlalchemy.ext.declarative** **import** declarative\_base

Base = declarative\_base()

**class** **CaseInsensitiveComparator**(Comparator):

**def** \_\_eq\_\_(self, other):

**return** func.lower(self.\_\_clause\_element\_\_()) == func.lower(other)

**class** **SearchWord**(Base):

\_\_tablename\_\_ = 'searchword'

id = Column(Integer, primary\_key=**True**)

word = Column(String(255), nullable=**False**)

**@hybrid\_property**

**def** word\_insensitive(self):

**return** self.word.lower()

**@word\_insensitive**.comparator

**def** word\_insensitive(cls):

**return** CaseInsensitiveComparator(cls.word)

Above, SQL expressions against word\_insensitive will apply the LOWER() SQL function to both sides:

**>>>** print Session().query(SearchWord).filter\_by(word\_insensitive="Trucks")

SELECT searchword.id AS searchword\_id, searchword.word AS searchword\_word

FROM searchword

WHERE lower(searchword.word) = lower(:lower\_1)

The CaseInsensitiveComparator above implements part of the [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators) interface. A "coercion" operation like lowercasing can be applied to all comparison operations (i.e. eq, lt, gt, etc.) using [Operators.operate()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.Operators.operate" \o "sqlalchemy.sql.operators.Operators.operate):

**class** **CaseInsensitiveComparator**(Comparator):

**def** operate(self, op, other):

**return** op(func.lower(self.\_\_clause\_element\_\_()), func.lower(other))

#### 7.8.5 Hybrid Value Objects

Note in our previous example, if we were to compare the word\_insensitive attribute of a SearchWord instance to a plain Python string, the plain Python string would not be coerced to lower case - the CaseInsensitiveComparator we built, being returned by @word\_insensitive.comparator, only applies to the SQL side.

在前面的例子中，请注意，如果要将SearchWord实例的word\_insensitive属性与普通Python字符串进行比较，则纯Python字符串不会被强制为小写 - 仅由@word\_insensitive.comparator返回的CaseInsensitiveComparator 适用于SQL端。

A more comprehensive form of the custom comparator is to construct a *Hybrid Value Object*. This technique applies the target value or expression to a value object which is then returned by the accessor in all cases. The value object allows control of all operations upon the value as well as how compared values are treated, both on the SQL expression side as well as the Python value side. Replacing the previous CaseInsensitiveComparator class with a new CaseInsensitiveWord class:

一个更全面的定制比较器形式是构造一个混合值对象。 该技术将目标值或表达式应用于值对象，然后在所有情况下由访问器返回值对象。 该值对象允许对该值进行所有操作的控制，以及在SQL表达式以及Python值方面如何处理对比值。 用新的CaseInsensitiveComparator类替换以前的CaseInsensitiveWord类：

**class** **CaseInsensitiveWord**(Comparator):

"Hybrid value representing a lower case representation of a word."

**def** \_\_init\_\_(self, word):

**if** isinstance(word, basestring):

self.word = word.lower()

**elif** isinstance(word, CaseInsensitiveWord):

self.word = word.word

**else**:

self.word = func.lower(word)

**def** operate(self, op, other):

**if** **not** isinstance(other, CaseInsensitiveWord):

other = CaseInsensitiveWord(other)

**return** op(self.word, other.word)

**def** \_\_clause\_element\_\_(self):

**return** self.word

**def** \_\_str\_\_(self):

**return** self.word

key = 'word'

"Label to apply to Query tuple results"

Above, the CaseInsensitiveWord object represents self.word, which may be a SQL function, or may be a Python native. By overriding operate() and \_\_clause\_element\_\_() to work in terms of self.word, all comparison operations will work against the "converted" form of word, whether it be SQL side or Python side. Our SearchWord class can now deliver the CaseInsensitiveWord object unconditionally from a single hybrid call:

以上，CaseInsensitiveWord对象表示self.word，可以是SQL函数，也可以是Python本机。 通过重写operating()和\_\_clause\_element \_\_()来处理self\_word，所有的比较操作都可以反对“转换”的单词形式，无论是SQL边还是Python边。 我们的SearchWord类现在可以无条件地从单个混合调用中传递CaseInsensitiveWord对象：

**class** **SearchWord**(Base):

\_\_tablename\_\_ = 'searchword'

id = Column(Integer, primary\_key=**True**)

word = Column(String(255), nullable=**False**)

**@hybrid\_property**

**def** word\_insensitive(self):

**return** CaseInsensitiveWord(self.word)

The word\_insensitive attribute now has case-insensitive comparison behavior universally, including SQL expression vs. Python expression (note the Python value is converted to lower case on the Python side here):

word\_insensitive属性现在通常具有不区分大小写的比较行为，包括SQL表达式与Python表达式(请注意，Python值在Python端转换为小写)：

**>>>** print Session().query(SearchWord).filter\_by(word\_insensitive="Trucks")

SELECT searchword.id AS searchword\_id, searchword.word AS searchword\_word

FROM searchword

WHERE lower(searchword.word) = :lower\_1

SQL expression versus SQL expression:

**>>>** sw1 = aliased(SearchWord)

**>>>** sw2 = aliased(SearchWord)

**>>>** print Session().query(

**...**  sw1.word\_insensitive,

**...**  sw2.word\_insensitive).\

**...**  filter(

**...**  sw1.word\_insensitive > sw2.word\_insensitive

**...**  )

SELECT lower(searchword\_1.word) AS lower\_1,lower(searchword\_2.word) AS lower\_2

FROM searchword AS searchword\_1, searchword AS searchword\_2

WHERE lower(searchword\_1.word) > lower(searchword\_2.word)

Python only expression:

**>>>** ws1 = SearchWord(word="SomeWord")

**>>>** ws1.word\_insensitive == "sOmEwOrD"

True

**>>>** ws1.word\_insensitive == "XOmEwOrX"

False

**>>>** print ws1.word\_insensitivesomeword

The Hybrid Value pattern is very useful for any kind of value that may have multiple representations, such as timestamps, time deltas, units of measurement, currencies and encrypted passwords.

**See also**

[Hybrids and Value Agnostic Types](http://techspot.zzzeek.org/2011/10/21/hybrids-and-value-agnostic-types/) - on the techspot.zzzeek.org blog

[Value Agnostic Types, Part II](http://techspot.zzzeek.org/2011/10/29/value-agnostic-types-part-ii/) - on the techspot.zzzeek.org blog

7.8.6 Building Transformers

A *transformer* is an object which can receive a [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) object and return a new one. The [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) object includes a method [with\_transformation()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.with_transformation" \o "sqlalchemy.orm.query.Query.with_transformation) that returns a new [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) transformed by the given function.

We can combine this with the [Comparator](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/hybrid.html" \l "sqlalchemy.ext.hybrid.Comparator" \o "sqlalchemy.ext.hybrid.Comparator) class to produce one type of recipe which can both set up the FROM clause of a query as well as assign filtering criterion.

Consider a mapped class Node, which assembles using adjacency list into a hierarchical tree pattern:

**from** **sqlalchemy** **import** Column, Integer, ForeignKey

**from** **sqlalchemy.orm** **import** relationship

**from** **sqlalchemy.ext.declarative** **import** declarative\_baseBase = declarative\_base()

**class** **Node**(Base):

\_\_tablename\_\_ = 'node'

id = Column(Integer, primary\_key=**True**)

parent\_id = Column(Integer, ForeignKey('node.id'))

parent = relationship("Node", remote\_side=id)

Suppose we wanted to add an accessor grandparent. This would return the parent of Node.parent. When we have an instance of Node, this is simple:

**from** **sqlalchemy.ext.hybrid** **import** hybrid\_property

**class** **Node**(Base):

*# ...*

**@hybrid\_property**

**def** grandparent(self):

**return** self.parent.parent

For the expression, things are not so clear. We'd need to construct a [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) where we [join()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join) twice along Node.parent to get to the grandparent. We can instead return a transforming callable that we'll combine with the [Comparator](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/hybrid.html" \l "sqlalchemy.ext.hybrid.Comparator" \o "sqlalchemy.ext.hybrid.Comparator) class to receive any [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) object, and return a new one that's joined to the Node.parent attribute and filtered based on the given criterion:

对于表达来说，事情并不那么清楚。 我们需要在Node.parent中构造一个我们join()两次的Query来获取祖父母。 我们可以返回一个转换的可调用，我们将与Comparator类结合使用，以接收任何Query对象，并返回一个新的Node.parent属性，并根据给定的标准进行过滤：

**from** **sqlalchemy.ext.hybrid** **import** Comparator

**class** **GrandparentTransformer**(Comparator):

**def** operate(self, op, other):

**def** transform(q):

cls = self.\_\_clause\_element\_\_()

parent\_alias = aliased(cls)

**return** q.join(parent\_alias, cls.parent).\

filter(op(parent\_alias.parent, other))

**return** transform

Base = declarative\_base()

**class** **Node**(Base):

\_\_tablename\_\_ = 'node'

id =Column(Integer, primary\_key=**True**)

parent\_id = Column(Integer, ForeignKey('node.id'))

parent = relationship("Node", remote\_side=id)

**@hybrid\_property**

**def** grandparent(self):

**return** self.parent.parent

**@grandparent**.comparator

**def** grandparent(cls):

**return** GrandparentTransformer(cls)

The GrandparentTransformer overrides the core [Operators.operate()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.Operators.operate" \o "sqlalchemy.sql.operators.Operators.operate) method at the base of the [Comparator](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/hybrid.html" \l "sqlalchemy.ext.hybrid.Comparator" \o "sqlalchemy.ext.hybrid.Comparator) hierarchy to return a query-transforming callable, which then runs the given comparison operation in a particular context. Such as, in the example above, the operate method is called, given the Operators.eq callable as well as the right side of the comparison Node(id=5). A function transform is then returned which will transform a [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) first to join to Node.parent, then to compare parent\_alias using Operators.eq against the left and right sides, passing into Query.filter:

GrandparentTransformer在Comparator层次结构的基础上覆盖核心的Operators.operate()方法，以返回一个查询转换可调用函数，然后在特定上下文中运行给定的比较操作。 例如，在上面的示例中，给定了Operators.eq可调用的操作方法以及比较Node(id = 5)的右侧。 然后返回一个函数变换，这将首先转换一个Query以加入Node.parent，然后使用Operators.eq与左侧和右侧比较parent\_alias，传入Query.filter：

**>>> from** **sqlalchemy.orm** **import** Session

**>>>** session = Session()

**>>>** session.query(Node).\

**...**  with\_transformation(Node.grandparent==Node(id=5)).\

**...**  all()

SELECT node.id AS node\_id, node.parent\_id AS node\_parent\_id

FROM node JOIN node AS node\_1 ON node\_1.id = node.parent\_id

WHERE :param\_1 = node\_1.parent\_id

We can modify the pattern to be more verbose but flexible by separating the "join" step from the "filter" step. The tricky part here is ensuring that successive instances of GrandparentTransformer use the same [AliasedClass](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.util.AliasedClass" \o "sqlalchemy.orm.util.AliasedClass) object against Node. Below we use a simple memoizing approach that associates a GrandparentTransformer with each class:

**class** **Node**(Base):

*# ...*

**@grandparent**.comparator

**def** grandparent(cls):

*# memoize a GrandparentTransformer*

*# per class*

**if** '\_gp' **not** **in** cls.\_\_dict\_\_:

cls.\_gp = GrandparentTransformer(cls)

**return** cls.\_gp

**class** **GrandparentTransformer**(Comparator):

**def** \_\_init\_\_(self, cls):

self.parent\_alias = aliased(cls)

**@property**

**def** join(self):

**def** go(q):

**return** q.join(self.parent\_alias, Node.parent)

**return** go

**def** operate(self, op, other):

**return** op(self.parent\_alias.parent, other)

**>>>** session.query(Node).\

**...**  with\_transformation(Node.grandparent.join).\

**...**  filter(Node.grandparent==Node(id=5))

SELECT node.id AS node\_id, node.parent\_id AS node\_parent\_id

FROM node JOIN node AS node\_1 ON node\_1.id = node.parent\_id

WHERE :param\_1 = node\_1.parent\_id

The "transformer" pattern is an experimental pattern that starts to make usage of some functional programming paradigms. While it's only recommended for advanced and/or patient developers, there's probably a whole lot of amazing things it can be used for.

7.8.5 API Reference

*class*sqlalchemy.ext.hybrid.**hybrid\_method**(*func*, *expr=None*)

Bases: [sqlalchemy.orm.base.InspectionAttrInfo](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.base.InspectionAttrInfo" \o "sqlalchemy.orm.base.InspectionAttrInfo)

A decorator which allows definition of a Python object method with both instance-level and class-level behavior.

**\_\_init\_\_**(*func*, *expr=None*)

Create a new [hybrid\_method](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/hybrid.html" \l "sqlalchemy.ext.hybrid.hybrid_method" \o "sqlalchemy.ext.hybrid.hybrid_method).

Usage is typically via decorator:

**from** **sqlalchemy.ext.hybrid** **import** hybrid\_method

**class** **SomeClass**(object):

**@hybrid\_method**

**def** value(self, x, y):

**return** self.\_value + x + y

**@value**.expression

**def** value(self, x, y):

**return** func.some\_function(self.\_value, x, y)

**expression**(*expr*)

Provide a modifying decorator that defines a SQL-expression producing method.

*class*sqlalchemy.ext.hybrid.**hybrid\_property**(*fget*, *fset=None*, *fdel=None*, *expr=None*)

Bases: [sqlalchemy.orm.base.InspectionAttrInfo](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.base.InspectionAttrInfo" \o "sqlalchemy.orm.base.InspectionAttrInfo)

A decorator which allows definition of a Python descriptor with both instance-level and class-level behavior.

**\_\_init\_\_**(*fget*, *fset=None*, *fdel=None*, *expr=None*)

Create a new [hybrid\_property](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/hybrid.html" \l "sqlalchemy.ext.hybrid.hybrid_property" \o "sqlalchemy.ext.hybrid.hybrid_property).

Usage is typically via decorator:

**from** **sqlalchemy.ext.hybrid** **import** hybrid\_property

**class** **SomeClass**(object):

**@hybrid\_property**

**def** value(self):

**return** self.\_value

**@value**.setter

**def** value(self, value):

self.\_value = value

**comparator**(*comparator*)

Provide a modifying decorator that defines a custom comparator producing method.

The return value of the decorated method should be an instance of [Comparator](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/hybrid.html" \l "sqlalchemy.ext.hybrid.Comparator" \o "sqlalchemy.ext.hybrid.Comparator).

**deleter**(*fdel*)

Provide a modifying decorator that defines a value-deletion method.

**expression**(*expr*)

Provide a modifying decorator that defines a SQL-expression producing method.

**setter**(*fset*)

Provide a modifying decorator that defines a value-setter method.

*class*sqlalchemy.ext.hybrid.**Comparator**(*expression*)

Bases: [sqlalchemy.orm.interfaces.PropComparator](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator" \o "sqlalchemy.orm.interfaces.PropComparator)

A helper class that allows easy construction of custom [PropComparator](http://docs.sqlalchemy.org/en/rel_1_1/orm/internals.html" \l "sqlalchemy.orm.interfaces.PropComparator" \o "sqlalchemy.orm.interfaces.PropComparator) classes for usage with hybrids.

sqlalchemy.ext.hybrid.**HYBRID\_METHOD***= symbol('HYBRID\_METHOD')*

sqlalchemy.ext.hybrid.**HYBRID\_PROPERTY***= symbol('HYBRID\_PROPERTY')*

# **7.9 Indexable**

Define attributes on ORM-mapped classes that have "index" attributes for columns with [Indexable](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.Indexable" \o "sqlalchemy.types.Indexable) types.

"index" means the attribute is associated with an element of an [Indexable](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.Indexable" \o "sqlalchemy.types.Indexable) column with the predefined index to access it. The [Indexable](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.Indexable" \o "sqlalchemy.types.Indexable) types include types such as [ARRAY](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.ARRAY" \o "sqlalchemy.types.ARRAY), [JSON](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.JSON" \o "sqlalchemy.types.JSON) and [HSTORE](http://docs.sqlalchemy.org/en/rel_1_1/dialects/postgresql.html" \l "sqlalchemy.dialects.postgresql.HSTORE" \o "sqlalchemy.dialects.postgresql.HSTORE).

The [indexable](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/indexable.html" \l "module-sqlalchemy.ext.indexable" \o "sqlalchemy.ext.indexable) extension provides [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column)-like interface for any element of an [Indexable](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.Indexable" \o "sqlalchemy.types.Indexable) typed column. In simple cases, it can be treated as a [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) - mapped attribute.

*New in version 1.1.*

7.9.1 Synopsis

Given Person as a model with a primary key and JSON data field. While this field may have any number of elements encoded within it, we would like to refer to the element called name individually as a dedicated attribute which behaves like a standalone column:

给予Person作为具有主键和JSON数据字段的模型。 虽然此字段可能具有在其中编码的任何数量的元素，但我们想将称为名称的元素分别称为专用属性，其行为类似于独立列：

**from** **sqlalchemy** **import** Column, JSON, Integer

**from** **sqlalchemy.ext.declarative** **import** declarative\_base

**from** **sqlalchemy.ext.indexable** **import** index\_property

Base = declarative\_base()

**class** **Person**(Base):

\_\_tablename\_\_ = 'person'

id = Column(Integer, primary\_key=**True**)

data = Column(JSON)

name = index\_property('data', 'name')

Above, the name attribute now behaves like a mapped column. We can compose a new Person and set the value of name:

**>>>** person = Person(name='Alchemist')

The value is now accessible:

**>>>** person.name

'Alchemist'

Behind the scenes, the JSON field was initialized to a new blank dictionary and the field was set:

**>>>** person.data

{"name": "Alchemist'}

The field is mutable in place:

**>>>** person.name = 'Renamed'

**>>>** person.name

'Renamed'

**>>>** person.data

{'name': 'Renamed'}

When using [index\_property](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/indexable.html" \l "sqlalchemy.ext.indexable.index_property" \o "sqlalchemy.ext.indexable.index_property), the change that we make to the indexable structure is also automatically tracked as history; we no longer need to use [MutableDict](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableDict" \o "sqlalchemy.ext.mutable.MutableDict)in order to track this change for the unit of work.

Deletions work normally as well:

**>>> del** person.name

**>>>** person.data

{}

Above, deletion of person.name deletes the value from the dictionary, but not the dictionary itself.

A missing key will produce AttributeError:

**>>>** person = Person()

**>>>** person.name

**...**AttributeError: 'name'

Unless you set a default value:

**>>> class** **Person**(Base):

**>>>**  \_\_tablename\_\_ = 'person'

>>>

**>>>**  id = Column(Integer, primary\_key=**True**)

**>>>**  data = Column(JSON)

>>>

**>>>**  name = index\_property('data', 'name', default=**None**) *# See default*

**>>>** person = Person()

**>>>** print(person.name)

None

The attributes are also accessible at the class level. Below, we illustrate Person.name used to generate an indexed SQL criteria:

**>>> from** **sqlalchemy.orm** **import** Session

**>>>** session = Session()

**>>>** query = session.query(Person).filter(Person.name == 'Alchemist')

The above query is equivalent to:

**>>>** query = session.query(Person).filter(Person.data['name'] == 'Alchemist')

Multiple [index\_property](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/indexable.html" \l "sqlalchemy.ext.indexable.index_property" \o "sqlalchemy.ext.indexable.index_property) objects can be chained to produce multiple levels of indexing:

**from** **sqlalchemy** **import** Column, JSON, Integer

**from** **sqlalchemy.ext.declarative** **import** declarative\_base

**from** **sqlalchemy.ext.indexable** **import** index\_property

Base = declarative\_base()

**class** **Person**(Base):

\_\_tablename\_\_ = 'person'

id = Column(Integer, primary\_key=**True**)

data = Column(JSON)

birthday = index\_property('data', 'birthday')

year = index\_property('birthday', 'year')

month = index\_property('birthday', 'month')

day = index\_property('birthday', 'day')

Above, a query such as:

q = session.query(Person).filter(Person.year == '1980')

On a PostgreSQL backend, the above query will render as:

SELECT person.id, person.data

FROM person

WHERE person.data -> %(data\_1)s -> %(param\_1)s = %(param\_2)s

#### 7.9.2 Default Values

[index\_property](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/indexable.html" \l "sqlalchemy.ext.indexable.index_property" \o "sqlalchemy.ext.indexable.index_property) includes special behaviors for when the indexed data structure does not exist, and a set operation is called:

* For an [index\_property](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/indexable.html" \l "sqlalchemy.ext.indexable.index_property" \o "sqlalchemy.ext.indexable.index_property) that is given an integer index value, the default data structure will be a Python list of None values, at least as long as the index value; the value is then set at its place in the list. This means for an index value of zero, the list will be initialized to [None] before setting the given value, and for an index value of five, the list will be initialized to [None, None, None, None, None] before setting the fifth element to the given value. Note that an existing list is ****not**** extended in place to receive a value.
* for an [index\_property](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/indexable.html" \l "sqlalchemy.ext.indexable.index_property" \o "sqlalchemy.ext.indexable.index_property) that is given any other kind of index value (e.g. strings usually), a Python dictionary is used as the default data structure.
* The default data structure can be set to any Python callable using the [index\_property.datatype](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/indexable.html" \l "sqlalchemy.ext.indexable.index_property.params.datatype" \o "sqlalchemy.ext.indexable.index_property) parameter, overriding the previous rules.

7.9.3 Subclassing

[index\_property](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/indexable.html" \l "sqlalchemy.ext.indexable.index_property" \o "sqlalchemy.ext.indexable.index_property) can be subclassed, in particular for the common use case of providing coercion of values or SQL expressions as they are accessed. Below is a common recipe for use with a PostgreSQL JSON type, where we want to also include automatic casting plus astext():

**class** **pg\_json\_property**(index\_property):

**def** \_\_init\_\_(self, attr\_name, index, cast\_type):

super(pg\_json\_property, self).\_\_init\_\_(attr\_name, index)

self.cast\_type = cast\_type

**def** expr(self, model):

expr = super(pg\_json\_property, self).expr(model)

**return** expr.astext.cast(self.cast\_type)

The above subclass can be used with the PostgreSQL-specific version of [postgresql.JSON](http://docs.sqlalchemy.org/en/rel_1_1/dialects/postgresql.html" \l "sqlalchemy.dialects.postgresql.JSON" \o "sqlalchemy.dialects.postgresql.JSON):

**from** **sqlalchemy** **import** Column, Integer

**from** **sqlalchemy.ext.declarative** **import** declarative\_base

**from** **sqlalchemy.dialects.postgresql** **import** JSON

Base = declarative\_base()

**class** **Person**(Base):

\_\_tablename\_\_ = 'person'

id = Column(Integer, primary\_key=**True**)

data = Column(JSON)

age = pg\_json\_property('data', 'age', Integer)

The age attribute at the instance level works as before; however when rendering SQL, PostgreSQL's ->> operator will be used for indexed access, instead of the usual index opearator of ->:

实例级别的age属性与以前一样工作; 然而，当渲染SQL时，PostgreSQL - >>操作符将被用于索引访问，而不是通常的 - >的索引运算符：

**>>>** query = session.query(Person).filter(Person.age < 20)

The above query will render:

SELECT person.id, person.data

FROM person

WHERE CAST(person.data ->> %(data\_1)s AS INTEGER) < %(param\_1)s

### 7.9.5 API Reference

*class*sqlalchemy.ext.indexable.**index\_property**(*attr\_name*, *index*, *default=<object object>*, *datatype=None*, *mutable=True*, *onebased=True*)

Bases: [sqlalchemy.ext.hybrid.hybrid\_property](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/hybrid.html" \l "sqlalchemy.ext.hybrid.hybrid_property" \o "sqlalchemy.ext.hybrid.hybrid_property)

A property generator. The generated property describes an object attribute that corresponds to an [Indexable](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.Indexable" \o "sqlalchemy.types.Indexable) column.

属性生成器。 生成的属性描述对应于可索引列的对象属性。

*New in version 1.1.*

**See also**

[sqlalchemy.ext.indexable](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/indexable.html" \l "module-sqlalchemy.ext.indexable" \o "sqlalchemy.ext.indexable)

**\_\_init\_\_**(*attr\_name*, *index*, *default=<object object>*, *datatype=None*, *mutable=True*, *onebased=True*)

Create a new [index\_property](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/indexable.html" \l "sqlalchemy.ext.indexable.index_property" \o "sqlalchemy.ext.indexable.index_property).

|  |  |
| --- | --- |
| **Parameters:** | * ****attr\_name**** – An attribute name of an Indexable typed column, or other attribute that returns an indexable structure. * ****index**** – The index to be used for getting and setting this value. This should be the Python-side index value for integers. * ****default**** – A value which will be returned instead of AttributeError when there is not a value at given index. * ****datatype**** – default datatype to use when the field is empty. By default, this is derived from the type of index used; a Python list for an integer index, or a Python dictionary for any other style of index. For a list, the list will be initialized to a list of None values that is at least index elements long. * ****mutable**** – if False, writes and deletes to the attribute will be disallowed. * ****onebased**** – assume the SQL representation of this value is one-based; that is, the first index in SQL is 1, not zero. |

# **7.10 Alternate Class Instrumentation**

Extensible class instrumentation.

The [sqlalchemy.ext.instrumentation](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/instrumentation.html" \l "module-sqlalchemy.ext.instrumentation" \o "sqlalchemy.ext.instrumentation) package provides for alternate systems of class instrumentation within the ORM. Class instrumentation refers to how the ORM places attributes on the class which maintain data and track changes to that data, as well as event hooks installed on the class.

**Note**

The extension package is provided for the benefit of integration with other object management packages, which already perform their own instrumentation. It is not intended for general use.

For examples of how the instrumentation extension is used, see the example [Attribute Instrumentation](http://docs.sqlalchemy.org/en/rel_1_1/orm/examples.html" \l "examples-instrumentation).

*Changed in version 0.8:*The [sqlalchemy.orm.instrumentation](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "module-sqlalchemy.orm.instrumentation" \o "sqlalchemy.orm.instrumentation) was split out so that all functionality having to do with non-standard instrumentation was moved out to [sqlalchemy.ext.instrumentation](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/instrumentation.html" \l "module-sqlalchemy.ext.instrumentation" \o "sqlalchemy.ext.instrumentation). When imported, the module installs itself within [sqlalchemy.orm.instrumentation](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "module-sqlalchemy.orm.instrumentation" \o "sqlalchemy.orm.instrumentation) so that it takes effect, including recognition of\_\_sa\_instrumentation\_manager\_\_ on mapped classes, as well [instrumentation\_finders](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/instrumentation.html" \l "sqlalchemy.ext.instrumentation.instrumentation_finders" \o "sqlalchemy.ext.instrumentation.instrumentation_finders) being used to determine class instrumentation resolution.

7.10.1 API Reference

sqlalchemy.ext.instrumentation.**INSTRUMENTATION\_MANAGER***= '\_\_sa\_instrumentation\_manager\_\_'*

Attribute, elects custom instrumentation when present on a mapped class.

Allows a class to specify a slightly or wildly different technique for tracking changes made to mapped attributes and collections.

Only one instrumentation implementation is allowed in a given object inheritance hierarchy.

The value of this attribute must be a callable and will be passed a class object. The callable must return one of:

* An instance of an InstrumentationManager or subclass
* An object implementing all or some of InstrumentationManager (TODO)
* A dictionary of callables, implementing all or some of the above (TODO)
* An instance of a ClassManager or subclass

This attribute is consulted by SQLAlchemy instrumentation resolution, once the [sqlalchemy.ext.instrumentation](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/instrumentation.html" \l "module-sqlalchemy.ext.instrumentation" \o "sqlalchemy.ext.instrumentation) module has been imported. If custom finders are installed in the global instrumentation\_finders list, they may or may not choose to honor this attribute.

*class*sqlalchemy.orm.instrumentation.**InstrumentationFactory**

Factory for new ClassManager instances.

*class*sqlalchemy.ext.instrumentation.**InstrumentationManager**(*class\_*)

User-defined class instrumentation extension.

[InstrumentationManager](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/instrumentation.html" \l "sqlalchemy.ext.instrumentation.InstrumentationManager" \o "sqlalchemy.ext.instrumentation.InstrumentationManager) can be subclassed in order to change how class instrumentation proceeds. This class exists for the purposes of integration with other object management frameworks which would like to entirely modify the instrumentation methodology of the ORM, and is not intended for regular usage. For interception of class instrumentation events, see [InstrumentationEvents](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.InstrumentationEvents" \o "sqlalchemy.orm.events.InstrumentationEvents).

The API for this class should be considered as semi-stable, and may change slightly with new releases.

*Changed in version 0.8:*[InstrumentationManager](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/instrumentation.html" \l "sqlalchemy.ext.instrumentation.InstrumentationManager" \o "sqlalchemy.ext.instrumentation.InstrumentationManager) was moved from [sqlalchemy.orm.instrumentation](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "module-sqlalchemy.orm.instrumentation" \o "sqlalchemy.orm.instrumentation) to [sqlalchemy.ext.instrumentation](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/instrumentation.html" \l "module-sqlalchemy.ext.instrumentation" \o "sqlalchemy.ext.instrumentation).

**dict\_getter**(*class\_*)

**dispose**(*class\_*, *manager*)

**get\_instance\_dict**(*class\_*, *instance*)

**initialize\_instance\_dict**(*class\_*, *instance*)

**install\_descriptor**(*class\_*, *key*, *inst*)

**install\_member**(*class\_*, *key*, *implementation*)

**install\_state**(*class\_*, *instance*, *state*)

**instrument\_attribute**(*class\_*, *key*, *inst*)

**instrument\_collection\_class**(*class\_*, *key*, *collection\_class*)

**manage**(*class\_*, *manager*)

**manager\_getter**(*class\_*)

**post\_configure\_attribute**(*class\_*, *key*, *inst*)

**remove\_state**(*class\_*, *instance*)

**state\_getter**(*class\_*)

**uninstall\_descriptor**(*class\_*, *key*)

**uninstall\_member**(*class\_*, *key*)

sqlalchemy.ext.instrumentation.**instrumentation\_finders***= [<function find\_native\_user\_instrumentation\_hook>]*

An extensible sequence of callables which return instrumentation implementations

When a class is registered, each callable will be passed a class object. If None is returned, the next finder in the sequence is consulted. Otherwise the return must be an instrumentation factory that follows the same guidelines as sqlalchemy.ext.instrumentation.INSTRUMENTATION\_MANAGER.

By default, the only finder is find\_native\_user\_instrumentation\_hook, which searches for INSTRUMENTATION\_MANAGER. If all finders return None, standard ClassManager instrumentation is used.

*class*sqlalchemy.ext.instrumentation.**ExtendedInstrumentationRegistry**

Bases: [sqlalchemy.orm.instrumentation.InstrumentationFactory](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/instrumentation.html" \l "sqlalchemy.orm.instrumentation.InstrumentationFactory" \o "sqlalchemy.orm.instrumentation.InstrumentationFactory)

Extends [InstrumentationFactory](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/instrumentation.html" \l "sqlalchemy.orm.instrumentation.InstrumentationFactory" \o "sqlalchemy.orm.instrumentation.InstrumentationFactory) with additional bookkeeping, to accommodate multiple types of class managers.

# CHAPTER 8 ORM Examples

The SQLAlchemy distribution includes a variety of code examples illustrating a select set of patterns, some typical and some not so typical. All are runnable and can be found in the /examples directory of the distribution. Descriptions and source code for all can be found here.

Additional SQLAlchemy examples, some user contributed, are available on the wiki at <http://www.sqlalchemy.org/trac/wiki/UsageRecipes>.

8.1 Mapping Recipes

### 8.1.1 Adjacency List

An example of a dictionary-of-dictionaries structure mapped using an adjacency list model.

E.g.:

node = TreeNode('rootnode')

node.append('node1')

node.append('node3')

session.add(node)session.commit()

dump\_tree(node)

Listing of files:

* [adjacency\_list.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/adjacency_list/adjacency_list.html)

### 8.1.2 Associations

Examples illustrating the usage of the "association object" pattern, where an intermediary class mediates the relationship between two classes that are associated in a many-to-many pattern.

Listing of files:

* [dict\_of\_sets\_with\_default.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/association/dict_of_sets_with_default.html) - an advanced association proxy example which illustrates nesting of association proxies to produce multi-level Python collections, in this case a dictionary with string keys and sets of integers as values, which conceal the underlying mapped classes.
* [proxied\_association.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/association/proxied_association.html) - same example as basic\_association, adding in usage of [sqlalchemy.ext.associationproxy](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/associationproxy.html" \l "module-sqlalchemy.ext.associationproxy" \o "sqlalchemy.ext.associationproxy) to make explicit references to OrderItem optional.
* [basic\_association.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/association/basic_association.html) - illustrate a many-to-many relationship between an "Order" and a collection of "Item" objects, associating a purchase price with each via an association object called "OrderItem"

### 8.1.3 Directed Graphs

An example of persistence for a directed graph structure. The graph is stored as a collection of edges, each referencing both a "lower" and an "upper" node in a table of nodes. Basic persistence and querying for lower- and upper- neighbors are illustrated:

n2 = Node(2)

n5 = Node(5)

n2.add\_neighbor(n5)

print n2.higher\_neighbors()

Listing of files:

* [directed\_graph.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/graphs/directed_graph.html) - a directed graph example.

### 8.1.4 Dynamic Relations as Dictionaries

Illustrates how to place a dictionary-like facade on top of a "dynamic" relation, so that dictionary operations (assuming simple string keys) can operate upon a large collection without loading the full collection at once.

Listing of files:

* [dynamic\_dict.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/dynamic_dict/dynamic_dict.html)

### 8.1.5 Generic Associations

Illustrates various methods of associating multiple types of parents with a particular child object.

The examples all use the declarative extension along with declarative mixins. Each one presents the identical use case at the end - two classes, Customer and Supplier, both subclassing the HasAddresses mixin, which ensures that the parent class is provided with an addresses collection which contains Addressobjects.

The [discriminator\_on\_association.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/generic_associations/discriminator_on_association.html) and [generic\_fk.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/generic_associations/generic_fk.html) scripts are modernized versions of recipes presented in the 2007 blog post [Polymorphic Associations with SQLAlchemy](http://techspot.zzzeek.org/2007/05/29/polymorphic-associations-with-sqlalchemy/).

Listing of files:

* [discriminator\_on\_association.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/generic_associations/discriminator_on_association.html) - Illustrates a mixin which provides a generic association using a single target table and a single association table, referred to by all parent tables. The association table contains a "discriminator" column which determines what type of parent object associates to each particular row in the association table.
* [table\_per\_association.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/generic_associations/table_per_association.html) - Illustrates a mixin which provides a generic association via a individually generated association tables for each parent class. The associated objects themselves are persisted in a single table shared among all parents.
* [generic\_fk.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/generic_associations/generic_fk.html) - Illustrates a so-called "generic foreign key", in a similar fashion to that of popular frameworks such as Django, ROR, etc. This approach bypasses standard referential integrity practices, in that the "foreign key" column is not actually constrained to refer to any particular table; instead, in-application logic is used to determine which table is referenced.
* [table\_per\_related.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/generic_associations/table_per_related.html) - Illustrates a generic association which persists association objects within individual tables, each one generated to persist those objects on behalf of a particular parent class.

### 8.1.6 Large Collections

Large collection example.

Illustrates the options to use with [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) when the list of related objects is very large, including:

* "dynamic" relationships which query slices of data as accessed
* how to use ON DELETE CASCADE in conjunction with passive\_deletes=True to greatly improve the performance of related collection deletion.

Listing of files:

* [large\_collection.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/large_collection/large_collection.html)

### 8.1.7 Materialized Paths

Illustrates the "materialized paths" pattern for hierarchical data using the SQLAlchemy ORM.

Listing of files:

* [materialized\_paths.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/materialized_paths/materialized_paths.html) - Illustrates the "materialized paths" pattern.

### 8.1.8 Nested Sets

Illustrates a rudimentary way to implement the "nested sets" pattern for hierarchical data using the SQLAlchemy ORM.

Listing of files:

* [nested\_sets.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/nested_sets/nested_sets.html) - Celko's "Nested Sets" Tree Structure.

### 8.1.9 Performance

A performance profiling suite for a variety of SQLAlchemy use cases.

Each suite focuses on a specific use case with a particular performance profile and associated implications:

* bulk inserts
* individual inserts, with or without transactions
* fetching large numbers of rows
* running lots of short queries

All suites include a variety of use patterns illustrating both Core and ORM use, and are generally sorted in order of performance from worst to greatest, inversely based on amount of functionality provided by SQLAlchemy, greatest to least (these two things generally correspond perfectly).

A command line tool is presented at the package level which allows individual suites to be run:

$ python -m examples.performance --help

usage: python -m examples.performance [-h] [--test TEST] [--dburl DBURL]

[--num NUM] [--profile] [--dump]

[--runsnake] [--echo]

{bulk\_inserts,large\_resultsets,single\_inserts}

positional arguments:

{bulk\_inserts,large\_resultsets,single\_inserts}

suite to run

optional arguments:

-h, --help show this help message and exit

--test TEST run specific test name

--dburl DBURL database URL, default sqlite:///profile.db

--num NUM Number of iterations/items/etc for tests; default is 0

module-specific

--profile run profiling and dump call counts

--dump dump full call profile (implies --profile)

--runsnake invoke runsnakerun (implies --profile)

--echo Echo SQL output

An example run looks like:

$ python -m examples.performance bulk\_inserts

Or with options:

$ python -m examples.performance bulk\_inserts \

--dburl mysql+mysqldb://scott:tiger@localhost/test \

--profile --num 1000

**See also**

[How can I profile a SQLAlchemy powered application?](http://docs.sqlalchemy.org/en/rel_1_1/faq/performance.html" \l "faq-how-to-profile)

#### File Listing

Listing of files:

* [bulk\_updates.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/performance/bulk_updates.html) - This series of tests illustrates different ways to UPDATE a large number of rows in bulk.
* [large\_resultsets.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/performance/large_resultsets.html) - In this series of tests, we are looking at time to load a large number of very small and simple rows.
* [short\_selects.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/performance/short_selects.html) - This series of tests illustrates different ways to SELECT a single record by primary key
* [bulk\_inserts.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/performance/bulk_inserts.html) - This series of tests illustrates different ways to INSERT a large number of rows in bulk.
* [single\_inserts.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/performance/single_inserts.html) - In this series of tests, we're looking at a method that inserts a row within a distinct transaction, and afterwards returns to essentially a "closed" state. This would be analogous to an API call that starts up a database connection, inserts the row, commits and closes.
* [\_\_main\_\_.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/performance/__main__.html) - Allows the examples/performance package to be run as a script.

#### Running all tests with time

This is the default form of run:

$ python -m examples.performance single\_inserts

Tests to run: test\_orm\_commit, test\_bulk\_save,

test\_bulk\_insert\_dictionaries, test\_core,

test\_core\_query\_caching, test\_dbapi\_raw\_w\_connect,

test\_dbapi\_raw\_w\_pool

test\_orm\_commit : Individual INSERT/COMMIT pairs via the

ORM (10000 iterations); total time 13.690218 sec

test\_bulk\_save : Individual INSERT/COMMIT pairs using

the "bulk" API (10000 iterations); total time 11.290371 sec

test\_bulk\_insert\_dictionaries : Individual INSERT/COMMIT pairs using

the "bulk" API with dictionaries (10000 iterations);

total time 10.814626 sec

test\_core : Individual INSERT/COMMIT pairs using Core.

(10000 iterations); total time 9.665620 sec

test\_core\_query\_caching : Individual INSERT/COMMIT pairs using Core

with query caching (10000 iterations); total time 9.209010 sec

test\_dbapi\_raw\_w\_connect : Individual INSERT/COMMIT pairs w/ DBAPI +

connection each time (10000 iterations); total time 9.551103 sec

test\_dbapi\_raw\_w\_pool : Individual INSERT/COMMIT pairs w/ DBAPI +

connection pool (10000 iterations); total time 8.001813 sec

#### Dumping Profiles for Individual Tests

A Python profile output can be dumped for all tests, or more commonly individual tests:

$ python -m examples.performance single\_inserts --test test\_core --num 1000 --dump

Tests to run: test\_core

test\_core : Individual INSERT/COMMIT pairs using Core. (1000 iterations); total fn calls 186109

186109 function calls (186102 primitive calls) in 1.089 seconds

Ordered by: internal time, call count

ncalls tottime percall cumtime percall filename:lineno(function)

1000 0.634 0.001 0.634 0.001 {method 'commit' of 'sqlite3.Connection' objects}

1000 0.154 0.000 0.154 0.000 {method 'execute' of 'sqlite3.Cursor' objects}

1000 0.021 0.000 0.074 0.000 /Users/classic/dev/sqlalchemy/lib/sqlalchemy/sql/compiler.py:1950(\_get\_colparams)

1000 0.015 0.000 0.034 0.000 /Users/classic/dev/sqlalchemy/lib/sqlalchemy/engine/default.py:503(\_init\_compiled)

1 0.012 0.012 1.091 1.091 examples/performance/single\_inserts.py:79(test\_core)

...

#### Using RunSnake

This option requires the [RunSnake](https://pypi.python.org/pypi/RunSnakeRun) command line tool be installed:

此选项需要安装RunSnake命令行工具：

$ python -m examples.performance single\_inserts --test test\_core --num 1000 --runsnake

A graphical RunSnake output will be displayed.

将显示图形RunSnake输出。

#### Writing your Own Suites

The profiler suite system is extensible, and can be applied to your own set of tests. This is a valuable technique to use in deciding upon the proper approach for some performance-critical set of routines. For example, if we wanted to profile the difference between several kinds of loading, we can create a file test\_loads.py, with the following content:

Profiler套件系统是可扩展的，可以应用于您自己的测试集。 这是一个有价值的技术，用于决定一些性能关键的程序集的正确方法。 例如，如果我们想要描述几种加载之间的差异，我们可以创建一个文件test\_loads.py，其中包含以下内容：

**from** **examples.performance** **import** Profiler

**from** **sqlalchemy** **import** Integer, Column, create\_engine, ForeignKey

**from** **sqlalchemy.orm** **import** relationship, joinedload, subqueryload, Session

**from** **sqlalchemy.ext.declarative** **import** declarative\_base

Base = declarative\_base()

engine = **None**

session = **None**

**class** **Parent**(Base):

\_\_tablename\_\_ = 'parent'

id = Column(Integer, primary\_key=**True**)

children = relationship("Child")

**class** **Child**(Base):

\_\_tablename\_\_ = 'child'

id = Column(Integer, primary\_key=**True**)

parent\_id = Column(Integer, ForeignKey('parent.id'))

*# Init with name of file, default number of items*

Profiler.init("test\_loads", 1000)

**@Profiler**.setup\_once

**def** setup\_once(dburl, echo, num):

"setup once. create an engine, insert fixture data"

**global** engine

engine = create\_engine(dburl, echo=echo)

Base.metadata.drop\_all(engine)

Base.metadata.create\_all(engine)

sess = Session(engine)

sess.add\_all([

Parent(children=[Child() **for** j **in** range(100)])

**for** i **in** range(num)

])

sess.commit()

**@Profiler**.setup

**def** setup(dburl, echo, num):

"setup per test. create a new Session."

**global** session

session = Session(engine)

*# pre-connect so this part isn't profiled (if we choose)*

session.connection()

**@Profiler**.profile

**def** test\_lazyload(n):

"load everything, no eager loading."

**for** parent **in** session.query(Parent):

parent.children

**@Profiler**.profile

**def** test\_joinedload(n):

"load everything, joined eager loading."

**for** parent **in** session.query(Parent).options(joinedload("children")):

parent.children

**@Profiler**.profile

**def** test\_subqueryload(n):

"load everything, subquery eager loading."

**for** parent **in** session.query(Parent).options(subqueryload("children")):

parent.children

**if** \_\_name\_\_ == '\_\_main\_\_':

Profiler.main()

We can run our new script directly:

我们可以直接运行我们的新脚本：

$ python test\_loads.py --dburl postgresql+psycopg2://scott:tiger@localhost/test

Running setup once...

Tests to run: test\_lazyload, test\_joinedload, test\_subqueryload

test\_lazyload : load everything, no eager loading. (1000 iterations); total time 11.971159 sec

test\_joinedload : load everything, joined eager loading. (1000 iterations); total time 2.754592 sec

test\_subqueryload : load everything, subquery eager loading. (1000 iterations); total time 2.977696 sec

As well as see RunSnake output for an individual test:

以及单独测试的RunSnake输出：

$ python test\_loads.py --num 100 --runsnake --test test\_joinedload

### 8.1.10 Relationship Join Conditions

Examples of various [orm.relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) configurations, which make use of the primaryjoin argument to compose special types of join conditions.

各种orm.relationship()配置的示例，它们使用primaryjoin参数来组合特殊类型的连接条件。

Listing of files:

文件清单：

* [cast.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/join_conditions/cast.html) - Illustrate a [relationship()](http://docs.sqlalchemy.org/en/rel_1_1/orm/relationship_api.html" \l "sqlalchemy.orm.relationship" \o "sqlalchemy.orm.relationship) that joins two columns where those columns are not of the same type, and a CAST must be used on the SQL side in order to match them.示出了一个关联()，它连接两列不同类型的关系，并且必须在SQL端使用CAST才能匹配它们。
* [threeway.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/join_conditions/threeway.html) - Illustrate a "three way join" - where a primary table joins to a remote table via an association table, but then the primary table also needs to refer to some columns in the remote table directly.说明“三向连接” - 主表通过关联表加入到远程表中，但是主表也需要直接引用远程表中的某些列。

### 8.1.11 XML Persistence

Illustrates three strategies for persisting and querying XML documents as represented by ElementTree in a relational database. The techniques do not apply any mappings to the ElementTree objects directly, so are compatible with the native cElementTree as well as lxml, and can be adapted to suit any kind of DOM representation system. Querying along xpath-like strings is illustrated as well.

说明持久化和查询关系数据库中的ElementTree所表示的XML文档的三种策略。 这些技术不直接对ElementTree对象应用任何映射，因此与本地cElementTree以及lxml兼容，并且可以适应任何种类的DOM表示系统。 同时也说明了像xpath一样的字符串的查询。

E.g.:

*# parse an XML file and persist in the database*

doc = ElementTree.parse("test.xml")session.add(Document(file, doc))

session.commit()

*# locate documents with a certain path/attribute structure*

**for** document **in** find\_document('/somefile/header/field2[@attr=foo]'):

*# dump the XML*

print document

Listing of files:

文件清单：

* [pickle.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/elementtree/pickle.html) - illustrates a quick and dirty way to persist an XML document expressed using ElementTree and pickle.说明了一种使用ElementTree和pickle表达的XML文档的一种快而脏的方法。
* [adjacency\_list.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/elementtree/adjacency_list.html) - Illustrates an explicit way to persist an XML document expressed using ElementTree.说明了一种使用ElementTree表达的XML文档的显式方式。
* [optimized\_al.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/elementtree/optimized_al.html) - Uses the same strategy as adjacency\_list.py, but associates each DOM row with its owning document row, so that a full document of DOM nodes can be loaded using O(1) queries - the construction of the "hierarchy" is performed after the load in a non-recursive fashion and is more efficient.使用与adjacency\_list.py相同的策略，但将每个DOM行与其拥有的文档行相关联，以便可以使用O(1)查询来加载DOM节点的完整文档 - “层次结构”的构造在 以非递归的方式加载并且更有效率。

### 8.1.12 Versioning Objects

#### Versioning with a History Table

版本控制与历史表

Illustrates an extension which creates version tables for entities and stores records for each change. The given extensions generate an anonymous "history" class which represents historical versions of the target object.

说明一个扩展，为实体创建版本表并存储每个更改的记录。 给定的扩展生成一个匿名的“历史”类，表示目标对象的历史版本。

Usage is illustrated via a unit test module test\_versioning.py, which can be run via nose:

通过单元测试模块test\_versioning.py来说明用法，可以通过nose运行：

cd examples/versioningnosetests -v

A fragment of example usage, using declarative:

示例使用的一个片段，使用声明式：

**from** **history\_meta** **import** Versioned, versioned\_session

Base = declarative\_base()

**class** **SomeClass**(Versioned, Base):

\_\_tablename\_\_ = 'sometable'

id = Column(Integer, primary\_key=**True**)

name = Column(String(50))

**def** \_\_eq\_\_(self, other):

**assert** type(other) **is** SomeClass **and** other.id == self.id

Session = sessionmaker(bind=engine)

versioned\_session(Session)

sess = Session()

sc = SomeClass(name='sc1')

sess.add(sc)sess.commit()

sc.name = 'sc1modified'

sess.commit()

**assert** sc.version == 2

SomeClassHistory = SomeClass.\_\_history\_mapper\_\_.class\_

**assert** sess.query(SomeClassHistory).\

filter(SomeClassHistory.version == 1).\

all() \

== [SomeClassHistory(version=1, name='sc1')]

The Versioned mixin is designed to work with declarative. To use the extension with classical mappers, the \_history\_mapper function can be applied:

版本化的mixin旨在使用声明式。 要将扩展与经典映射器配合使用，可以应用\_history\_mapper函数：

**from** **history\_meta** **import** \_history\_mapper

m = mapper(SomeClass, sometable)

\_history\_mapper(m)

SomeHistoryClass = SomeClass.\_\_history\_mapper\_\_.class\_

Listing of files:文件清单：

* [test\_versioning.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/versioned_history/test_versioning.html) - Unit tests illustrating usage of the history\_meta.py module functions.说明使用history\_meta.py模块功能的单元测试。
* [history\_meta.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/versioned_history/history_meta.html) - Versioned mixin class and other utilities.版本的mixin类和其他实用程序。

#### Versioning using Temporal Rows

Illustrates an extension which versions data by storing new rows for each change; that is, what would normally be an UPDATE becomes an INSERT.

说明通过为每次更改存储新行的版本数据的扩展名; 也就是说，通常UPDATE将成为一个INSERT。

Listing of files:

文件清单：

* [versioned\_rows.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/versioned_rows/versioned_rows.html) - Illustrates a method to intercept changes on objects, turning an UPDATE statement on a single row into an INSERT statement, so that a new row is inserted with the new data, keeping the old row intact.说明拦截对象变化的方法，将单个行上的UPDATE语句转换为INSERT语句，以便使用新数据插入新行，保持旧行不变。
* [versioned\_map.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/versioned_rows/versioned_map.html) - A variant of the versioned\_rows example. Here we store a dictionary of key/value pairs, storing the k/v's in a "vertical" fashion where each key gets a row. The value is split out into two separate datatypes, string and int - the range of datatype storage can be adjusted for individual needs.versioned\_rows示例的变体。 这里我们存储一个键/值对的字典，将k / v以“垂直”的方式存储，每个键都有一行。 该值分为两个独立的数据类型，string和int - 可以根据个人需要调整数据类型存储的范围。

### 8.1.13 Vertical Attribute Mapping

Illustrates "vertical table" mappings.

说明“垂直表”映射。

A "vertical table" refers to a technique where individual attributes of an object are stored as distinct rows in a table. The "vertical table" technique is used to persist objects which can have a varied set of attributes, at the expense of simple query control and brevity. It is commonly found in content/document management systems in order to represent user-created structures flexibly.

“垂直表”是指将对象的各个属性作为不同行存储在表中的技术。 “垂直表”技术用于维护可以具有不同的属性集的对象，牺牲简单的查询控制和简洁性。 通常在内容/文档管理系统中找到，以便灵活地表示用户创建的结构。

Two variants on the approach are given. In the second, each row references a "datatype" which contains information about the type of information stored in the attribute, such as integer, string, or date.

给出了两种方法。 在第二行中，每行引用一个“数据类型”，其中包含有关存储在属性中的信息类型的信息，如整数，字符串或日期。

Example:

shrew = Animal(u'shrew')

shrew[u'cuteness'] = 5

shrew[u'weasel-like'] = **False**

shrew[u'poisonous'] = **True**

session.add(shrew)session.flush()

q = (session.query(Animal).

filter(Animal.facts.any(

and\_(AnimalFact.key == u'weasel-like',

AnimalFact.value == **True**))))print 'weasel-like animals', q.all()

Listing of files:

文件清单：

* [dictlike.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/vertical/dictlike.html) - Mapping a vertical table as a dictionary. 将垂直表映射为字典。
* [dictlike-polymorphic.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/vertical/dictlike-polymorphic.html) - Mapping a polymorphic-valued vertical table as a dictionary.将多态值垂直表映射为字典。

8.2 Inheritance Mapping Recipes继承映射方法

### Basic Inheritance Mappings基本的继承映射

Working examples of single-table, joined-table, and concrete-table inheritance as described in [Mapping Class Inheritance Hierarchies](http://docs.sqlalchemy.org/en/rel_1_1/orm/inheritance.html).

单表，连接表和具体表继承的工作示例，如“映射类继承层次结构”中所述。

Listing of files:

* [single.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/inheritance/single.html) - Single-table (table-per-hierarchy) inheritance example.
* single.py - 单表(每层次分层)继承示例。
* [joined.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/inheritance/joined.html) - Joined-table (table-per-subclass) inheritance example.
* joined.py - 加入表(每个子类的表)继承示例。
* [concrete.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/inheritance/concrete.html) - Concrete-table (table-per-class) inheritance example.
* concrete.py - Concrete-table(table-per-class)继承示例。

8.3 Special APIs

### Attribute Instrumentation

Examples illustrating modifications to SQLAlchemy's attribute management system.

说明修改SQLAlchemy属性管理系统的示例。

Listing of files:

文件清单：

* [active\_column\_defaults.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/custom_attributes/active_column_defaults.html) - Illustrates use of the [AttributeEvents.init\_scalar()](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html" \l "sqlalchemy.orm.events.AttributeEvents.init_scalar" \o "sqlalchemy.orm.events.AttributeEvents.init_scalar) event, in conjunction with Core column defaults to provide ORM objects that automatically produce the default value when an un-set attribute is accessed.说明使用AttributeEvents.init\_scalar()事件与Core列默认值一起提供ORM对象，这些对象在访问未设置属性时自动生成默认值。
* active\_column\_defaults.py - 说明使用AttributeEvents.init\_scalar()事件，结合Core列默认值提供在访问未设置属性时自动生成默认值的ORM对象。
* [custom\_management.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/custom_attributes/custom_management.html) - Illustrates customized class instrumentation, using the [sqlalchemy.ext.instrumentation](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/instrumentation.html" \l "module-sqlalchemy.ext.instrumentation" \o "sqlalchemy.ext.instrumentation) extension package.
* custom\_management.py - 使用[sqlalchemy.ext.instrumentation](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/instrumentation.html" \l "module-sqlalchemy.ext.instrumentation" \o "sqlalchemy.ext.instrumentation)扩展包说明自定义的类工具。
* [listen\_for\_events.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/custom_attributes/listen_for_events.html) - Illustrates how to attach events to all instrumented attributes and listen for change events.
* listen\_for\_events.py - 说明如何将事件附加到所有已检测属性并侦听更改事件。

### Horizontal Sharding

A basic example of using the SQLAlchemy Sharding API. Sharding refers to horizontally scaling data across multiple databases.

使用SQLAlchemy Sharding API的基本示例。 分片是指跨多个数据库的水平缩放数据。

The basic components of a "sharded" mapping are:

* multiple databases, each assigned a 'shard id'
* 多个数据库，每个数据库分配一个'shard id'
* a function which can return a single shard id, given an instance to be saved; this is called "shard\_chooser"
* 可以返回单个分片ID的函数，给定要保存的实例; 这被称为“shard\_chooser”
* a function which can return a list of shard ids which apply to a particular instance identifier; this is called "id\_chooser". If it returns all shard ids, all shards will be searched.
* 可以返回适用于特定实例标识符的分片ID列表的功能; 这被称为“id\_chooser”。 如果它返回所有的分片，则会搜索所有分片。
* a function which can return a list of shard ids to try, given a particular Query ("query\_chooser"). If it returns all shard ids, all shards will be queried and the results joined together.
* 一个功能，可以返回一个分片标识列表来尝试，给定一个特定的查询(“query\_chooser”)。 如果它返回所有的分片，所有的分片将被查询，并将结果连接在一起。

In this example, four sqlite databases will store information about weather data on a database-per-continent basis. We provide example shard\_chooser, id\_chooser and query\_chooser functions. The query\_chooser illustrates inspection of the SQL expression element in order to attempt to determine a single shard being requested.

在这个例子中，四个sqlite数据库将在每个数据库的基础上存储有关天气数据的信息。 我们提供示例shard\_chooser，id\_chooser和query\_chooser函数。 query\_chooser说明了对SQL表达式元素的检查，以便尝试确定正在请求的单个分片。

The construction of generic sharding routines is an ambitious approach to the issue of organizing instances among multiple databases. For a more plain-spoken alternative, the "distinct entity" approach is a simple method of assigning objects to different tables (and potentially database nodes) in an explicit way - described on the wiki at [EntityName](http://www.sqlalchemy.org/trac/wiki/UsageRecipes/EntityName).

通用分片程序的构建对于在多个数据库之间组织实例的问题是一个雄心勃勃的方法。 对于一个更简单的替代方案，“独特实体”方法是一种将对象以明确方式分配给不同表(和潜在的数据库节点)的简单方法 - 在EntityName的wiki上描述。

Listing of files:

* [attribute\_shard.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/sharding/attribute_shard.html)

8.3 Extending the ORM

### Dogpile Caching

Illustrates how to embed [dogpile.cache](https://dogpilecache.readthedocs.io/) functionality within the [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) object, allowing full cache control as well as the ability to pull "lazy loaded" attributes from long term cache as well.

说明如何在[Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query)对象中嵌入dogpile.cache功能，允许完全缓存控制以及从长期缓存中拉取“延迟加载”属性的能力。

*Changed in version 0.8:*The example was modernized to use dogpile.cache, replacing Beaker as the caching library in use.

在版本0.8中更改：该示例现代化使用dogpile.cache，将Beaker替换为正在使用的缓存库。

In this demo, the following techniques are illustrated:

在本演示中，说明了以下技术：

* Using custom subclasses of [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query)
* 使用[Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query)的自定义子类
* Basic technique of circumventing Query to pull from a custom cache source instead of the database.
* 避免查询的基本技术从自定义缓存源而不是数据库提取。
* Rudimental caching with dogpile.cache, using "regions" which allow global control over a fixed set of configurations.
* 使用dogpile.cache进行实时缓存，使用“区域”，允许全局控制一组固定的配置。
* Using custom MapperOption objects to configure options on a Query, including the ability to invoke the options deep within an object graph when lazy loads occur.
* 使用自定义MapperOption对象来配置查询上的选项，包括在发生延迟加载时调用对象图形深度的选项的能力。

E.g.:

*# query for Person objects, specifying cache*

q = Session.query(Person).options(FromCache("default"))

*# specify that each Person's "addresses" collection comes from# cache too*

q = q.options(RelationshipCache(Person.addresses, "default"))

*# query*

print q.all()

To run, both SQLAlchemy and dogpile.cache must be installed or on the current PYTHONPATH. The demo will create a local directory for datafiles, insert initial data, and run. Running the demo a second time will utilize the cache files already present, and exactly one SQL statement against two tables will be emitted - the displayed result however will utilize dozens of lazyloads that all pull from cache.

要运行，必须安装SQLAlchemy和dogpile.cache，或者在当前的PYTHONPATH上。 演示将为数据文件创建本地目录，插入初始数据并运行。 第二次运行演示将利用已经存在的缓存文件，并且将发出一个针对两个表的SQL语句 - 显示的结果将会使用数十个从缓存中提取的懒惰载入。

The demo scripts themselves, in order of complexity, are run as Python modules so that relative imports work:

演示脚本本身以复杂性的顺序运行为Python模块，以便相对导入工作：

python -m examples.dogpile\_caching.helloworld

python -m examples.dogpile\_caching.relationship\_caching

python -m examples.dogpile\_caching.advanced

python -m examples.dogpile\_caching.local\_session\_caching

Listing of files:

* [environment.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/dogpile_caching/environment.html) - Establish data / cache file paths, and configurations, bootstrap fixture data if necessary.
* environment.py - 建立数据/缓存文件路径和配置，必要时引导夹具数据。
* [caching\_query.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/dogpile_caching/caching_query.html) - Represent functions and classes which allow the usage of Dogpile caching with SQLAlchemy. Introduces a query option called FromCache.
* caching\_query.py - 表示允许使用SQLAlchemy使用Dogpile缓存的函数和类。引入一个名为FromCache的查询选项。
* [model.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/dogpile_caching/model.html) - The datamodel, which represents Person that has multiple Address objects, each with PostalCode, City, Country.
* model.py - 数据模型，其表示具有多个Address对象的Person，每个对象具有PostalCode，City，Country。
* [fixture\_data.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/dogpile_caching/fixture_data.html) - Installs some sample data. Here we have a handful of postal codes for a few US/ Canadian cities. Then, 100 Person records are installed, each with a randomly selected postal code.
* fixture\_data.py - 安装一些示例数据。在这里，我们有几个美国/加拿大城市的邮政编码。然后，安装100人记录，每个记录随机选择一个邮政编码。
* [helloworld.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/dogpile_caching/helloworld.html) - Illustrate how to load some data, and cache the results.
* helloworld.py - 说明如何加载一些数据，并缓存结果。
* [relationship\_caching.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/dogpile_caching/relationship_caching.html) - Illustrates how to add cache options on relationship endpoints, so that lazyloads load from cache.
* relationship\_caching.py - 说明如何在关系端点上添加缓存选项，以便从缓存中加载lazyloads。
* [advanced.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/dogpile_caching/advanced.html) - Illustrate usage of Query combined with the FromCache option, including front-end loading, cache invalidation and collection caching.
* advanced.py - 说明查询结合FromCache选项的用法，包括前端加载，缓存无效和收集缓存。
* [local\_session\_caching.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/dogpile_caching/local_session_caching.html) - Grok everything so far ? This example creates a new dogpile.cache backend that will persist data in a dictionary which is local to the current session. remove() the session and the cache is gone.
* local\_session\_caching.py - 到目前为止所有的东西？此示例创建一个新的dogpile.cache后端，将在当前会话本地的字典中保留数据。 remove()会话和缓存已经不见了。

### PostGIS Integration

A naive example illustrating techniques to help embed PostGIS functionality.

一个天真的例子，说明了帮助嵌入PostGIS功能的技术。

This example was originally developed in the hopes that it would be extrapolated into a comprehensive PostGIS integration layer. We are pleased to announce that this has come to fruition as [GeoAlchemy](http://www.geoalchemy.org/).

这个例子最初是希望将其推广到一个综合的PostGIS集成层。 我们很高兴地宣布，这已经成为GeoAlchemy的成果。

The example illustrates:

* a DDL extension which allows CREATE/DROP to work in conjunction with AddGeometryColumn/DropGeometryColumn
* 一个DDL扩展，允许CREATE / DROP与AddGeometryColumn / DropGeometryColumn一起使用
* a Geometry type, as well as a few subtypes, which convert result row values to a GIS-aware object, and also integrates with the DDL extension.
* 几何类型，以及将结果行值转换为GIS感知对象的几个子类型，并且还与DDL扩展集成。
* a GIS-aware object which stores a raw geometry value and provides a factory for functions such as AsText().
* 一个GIS感知对象，用于存储原始几何值，并为AsText()等功能提供工厂。
* an ORM comparator which can override standard column methods on mapped objects to produce GIS operators.
* 一个ORM比较器可以覆盖映射对象上的标准列方法来生成GIS运算符。
* an attribute event listener that intercepts strings and converts to GeomFromText().
* 一个拦截字符串并转换为GeomFromText()的属性事件侦听器。
* a standalone operator example.
* 独立运算符示例。

The implementation is limited to only public, well known and simple to use extension points.

实施仅限于公开，众所周知且简单易用的扩展点。

E.g.:

print session.query(Road).filter(Road.road\_geom.intersects(r1.road\_geom)).all()

Listing of files:

* [postgis.py](http://docs.sqlalchemy.org/en/rel_1_1/_modules/examples/postgis/postgis.html)