[CHAPTER 1 SQL Expression Language Tutorial 5](#_Toc20161)

[1.1 Version Check 6](#_Toc31911)

[1.2 Connecting 6](#_Toc28355)

[1.3 Define and Create Tables 7](#_Toc19826)

[1.4 Insert Expressions 10](#_Toc22799)

[1.5 Executing 11](#_Toc11049)

[1.6 Executing Multiple Statements 13](#_Toc21047)

[1.7 Selecting 15](#_Toc20585)

[1.8 Operators 21](#_Toc22896)

[Operator Customization 23](#_Toc23156)

[1.9 Conjunctions 24](#_Toc20046)

[1.10.1 Specifying Bound Parameter Behaviors 28](#_Toc27776)

[1.10.2 Specifying Result-Column Behaviors 28](#_Toc18776)

[1.10.3 Using text() fragments inside bigger statements 31](#_Toc24187)

[1.10.4 Using More Specific Text with table(), literal\_column(), and column() 32](#_Toc26171)

[1.10.5 Ordering or Grouping by a Label 34](#_Toc23278)

[1.11 Using Aliases 35](#_Toc29869)

[1.12 Using Joins 38](#_Toc11486)

[1.13 Everything Else 41](#_Toc27957)

[1.13.1 Bind Parameter Objects 42](#_Toc17889)

[1.13.2 Functions 44](#_Toc12666)

[1.13.3 Window Functions 46](#_Toc1478)

[1.13.4 Unions and Other Set Operations 47](#_Toc7113)

[1.13.5 Scalar Selects 50](#_Toc22650)

[1.13.6 Correlated Subqueries 52](#_Toc16359)

[1.13.7 Ordering, Grouping, Limiting, Offset…ing… 57](#_Toc9232)

[1.14 Inserts, Updates and Deletes 60](#_Toc3956)

[1.14.1 Correlated Updates 63](#_Toc4121)

[1.14.2 Multiple Table Updates 63](#_Toc26217)

[1.14.3 Parameter-Ordered Updates 65](#_Toc14511)

[1.14.4 Deletes 66](#_Toc26737)

[1.14.5 Matched Row Counts 67](#_Toc13020)

[1.15 Further Reference 67](#_Toc7734)

[Chapter 2 SQL Statements and Expressions API 68](#_Toc13643)

[2.1 Column Elements and Expressions 68](#_Toc31184)

[2.2 Selectables, Tables, FROM objects 154](#_Toc3938)

[2.3 Insert, Updates, Deletes 312](#_Toc25596)

[2.4 SQL and Generic Functions 379](#_Toc10798)

[2.6 Custom SQL Constructs and Compilation Extension 395](#_Toc16328)

[2.6.1 Synopsis 395](#_Toc27093)

[2.6.2 Dialect-specific compilation rules 396](#_Toc210)

[2.6.3 Compiling sub-elements of a custom expression construct 396](#_Toc15201)

[Cross Compiling between SQL and DDL compilers 398](#_Toc4174)

[2.6.4 Enabling Autocommit on a Construct 398](#_Toc14887)

[2.6.5 Changing the default compilation of existing constructs 399](#_Toc24431)

[2.6.6 Changing Compilation of Types 400](#_Toc28946)

[2.6.7 Subclassing Guidelines 401](#_Toc14822)

[2.6.8 Further Examples 402](#_Toc8091)

["UTC timestamp" function 403](#_Toc16793)

["GREATEST" function 404](#_Toc26425)

["false" expression 405](#_Toc1210)

[2.7 Expression Serializer Extension 406](#_Toc11656)

[Chapter 3 Schema Definition Language 408](#_Toc27105)

[3.1 Describing Databases with MetaData 409](#_Toc8316)

[3.1.1 Accessing Tables and Columns 410](#_Toc18921)

[3.1.2 Creating and Dropping Database Tables 412](#_Toc30042)

[3.1.3 Altering Schemas through Migrations 415](#_Toc3470)

[3.1.4 Specifying the Schema Name 416](#_Toc79)

[3.1.5 Backend-Specific Options 417](#_Toc5409)

[3.1.6 Column, Table, MetaData API 417](#_Toc29145)

[3.2 Reflecting Database Objects 468](#_Toc8919)

[3.2.1 Overriding Reflected Columns 469](#_Toc12383)

[3.2.2 Reflecting Views 470](#_Toc1136)

[3.2.3 Reflecting All Tables at Once 470](#_Toc14254)

[3.2.4 Fine Grained Reflection with Inspector 471](#_Toc24405)

[3.2.5 Limitations of Reflection 478](#_Toc30001)

[3.3 Column Insert/Update Defaults 479](#_Toc23127)

[3.3.1 Scalar Defaults 479](#_Toc3874)

[3.3.2 Python-Executed Functions 480](#_Toc31415)

[Context-Sensitive Default Functions 481](#_Toc13830)

[3.3.3 SQL Expressions 482](#_Toc30241)

[3.3.4 Server Side Defaults 484](#_Toc23338)

[3.3.5 Triggered Columns 484](#_Toc2488)

[3.3.6 Defining Sequences 485](#_Toc15363)

[Associating a Sequence on a SERIAL column 486](#_Toc9842)

[Executing a Sequence Standalone 487](#_Toc29089)

[Associating a Sequence with the MetaData 487](#_Toc1358)

[Associating a Sequence as the Server Side Default 489](#_Toc27790)

[3.3.7 Default Objects API 491](#_Toc2595)

[3.4 Defining Constraints and Indexes 497](#_Toc18644)

[Creating/Dropping Foreign Key Constraints via ALTER 499](#_Toc21028)

[ON UPDATE and ON DELETE 503](#_Toc763)

[3.4.2 UNIQUE Constraint 504](#_Toc29517)

[3.4.3 CHECK Constraint 505](#_Toc7747)

[3.4.4 PRIMARY KEY Constraint 506](#_Toc23627)

[3.4.5 Setting up Constraints when using the Declarative ORM Extension 506](#_Toc3432)

[3.4.6 Configuring Constraint Naming Conventions 507](#_Toc164)

[Naming CHECK Constraints 511](#_Toc31398)

[Configuring Naming for Boolean, Enum, and other schema types 512](#_Toc6957)

[3.4.7 Constraints API 514](#_Toc12517)

[3.4.8 Indexes 542](#_Toc24839)

[Functional Indexes 544](#_Toc14127)

[3.4.9 Index API 544](#_Toc4546)

[3.5 Customizing DDL 550](#_Toc30937)

[3.5.1 Custom DDL 550](#_Toc30037)

[3.5.2 Controlling DDL Sequences 551](#_Toc26639)

[3.5.3 Using the built-in DDLElement Classes 553](#_Toc7942)

[3.5.4 DDL Expression Constructs API 555](#_Toc13729)

[Chapter 4 Column and Data Types 569](#_Toc17537)

[4.1 Column and Data Types 569](#_Toc20539)

[4.1.1Generic Types 569](#_Toc12970)

[4.1.2 SQL Standard and Multiple Vendor Types 584](#_Toc11897)

[4.1.3 Vendor-Specific Types 597](#_Toc30185)

[Custom Types 598](#_Toc17008)

[4.2.1 Overriding Type Compilation 598](#_Toc14857)

[4.2.2 Augmenting Existing Types 599](#_Toc10912)

[4.2.3 TypeDecorator Recipes 612](#_Toc17891)

[4.2.3.1 Coercing Encoded Strings to Unicode 612](#_Toc4980)

[4.2.3.2 Rounding Numerics 613](#_Toc22879)

[4.2.3.3 Backend-agnostic GUID Type 614](#_Toc5559)

[4.2.3.4 Marshal JSON Strings 615](#_Toc10393)

[4.2.4 Replacing the Bind/Result Processing of Existing Types 618](#_Toc416)

[4.2.5 Applying SQL-level Bind/Result Processing 619](#_Toc17717)

[4.2.6 Redefining and Creating New Operators 622](#_Toc17258)

[4.2.7 Creating New Types 625](#_Toc3864)

[4.3 Base Type API 626](#_Toc13723)

[Chapter 5 Engine and Connection Use 635](#_Toc20512)

[5.1 Engine Configuration 635](#_Toc5715)

[5.1.1 Supported Databases 636](#_Toc675)

[5.1.2 Database Urls 636](#_Toc16488)

[PostgreSQL 637](#_Toc171)

[MySQL 637](#_Toc3611)

[Oracle 637](#_Toc2637)

[Microsoft SQL Server 638](#_Toc15011)

[SQLite 638](#_Toc17298)

[Others 639](#_Toc227)

[5.1.3 Engine Creation API 639](#_Toc3783)

[5.1.4 Pooling 647](#_Toc26887)

[5.1.5 Custom DBAPI connect() arguments 647](#_Toc3633)

[5.1.6 Configuring Logging 648](#_Toc994)

[5.2 Working with Engines and Connections 650](#_Toc12406)

[5.2.1 Basic Usage 650](#_Toc2612)

[5.2.2 Using Transactions 653](#_Toc1339)

[Nesting of Transaction Blocks 654](#_Toc18255)

[5.2.3 Understanding Autocommit 655](#_Toc17352)

[5.2.4 Connectionless Execution, Implicit Execution 656](#_Toc12719)

[5.2.5 Translation of Schema Names 659](#_Toc31663)

[5.2.6 Engine Disposal 660](#_Toc27009)

[5.2.7 Using the Threadlocal Execution Strategy 662](#_Toc26569)

[5.2.8 Working with Raw DBAPI Connections 664](#_Toc21547)

[Calling Stored Procedures 665](#_Toc24072)

[Multiple Result Sets 665](#_Toc1365)

[5.2.9 Registering New Dialects 666](#_Toc27377)

[Registering Dialects In-Process 666](#_Toc23989)

[5.2.10 Connection / Engine API 667](#_Toc32720)

[5.3 Connection Pooling 703](#_Toc29248)

[5.3.1 Connection Pool Configuration 704](#_Toc17687)

[5.3.2 Switching Pool Implementations 705](#_Toc28872)

[5.3.3 Using a Custom Connection Function 706](#_Toc22789)

[5.3.4 Constructing a Pool 707](#_Toc5774)

[3.4.5 Pool Events 708](#_Toc4775)

[3.4.6 Dealing with Disconnects 708](#_Toc31218)

[Disconnect Handling - Optimistic 708](#_Toc1081)

[Disconnect Handling - Pessimistic 710](#_Toc27676)

[More on Invalidation 713](#_Toc2749)

[5.3.7 Using Connection Pools with Multiprocessing 714](#_Toc11439)

[5.3.8 API Documentation - Available Pool Implementations 716](#_Toc2886)

[5.3.9 Pooling Plain DB-API Connections 726](#_Toc10350)

[5.4 Core Events 728](#_Toc21407)

[5.4.1 Connection Pool Events 728](#_Toc446)

[5.4.2 SQL Execution and Connection Events 735](#_Toc317)

[5.4.3 Schema Events 759](#_Toc23133)

# 

# CHAPTER 1 SQL Expression Language Tutorial

The SQLAlchemy Expression Language presents a system of representing relational database structures and expressions using Python constructs. These constructs are modeled to resemble those of the underlying database as closely as possible, while providing a modicum of abstraction of the various implementation differences between database backends. While the constructs attempt to represent equivalent concepts between backends with consistent structures, they do not conceal useful concepts that are unique to particular subsets of backends. The Expression Language therefore presents a method of writing backend-neutral SQL expressions, but does not attempt to enforce that expressions are backend-neutral.

SQLAlchemy表达式语言提供了一种使用Python构造表示关系数据库结构和表达式的系统。 这些构造被建模为尽可能接近于基础数据库的构造，同时提供数据库后端之间的各种实现差异的一小部分抽象。 虽然这些构造尝试在具有一致结构的后端之间表示等效的概念，但它们不隐藏对于后端的特定子集是唯一的有用概念。 因此，表达式语言提供了一种编写后端中性SQL表达式的方法，但不会尝试强制该表达式是后端中立的。

The Expression Language is in contrast to the Object Relational Mapper, which is a distinct API that builds on top of the Expression Language. Whereas the ORM, introduced in [Object Relational Tutorial](http://docs.sqlalchemy.org/en/rel_1_1/orm/tutorial.html), presents a high level and abstracted pattern of usage, which itself is an example of applied usage of the Expression Language, the Expression Language presents a system of representing the primitive constructs of the relational database directly without opinion.

表达式语言与对象关系映射器形成对比，对象关系映射器是一种独特的API，构建在表达式语言之上。 而在对象关系教程中引入的ORM呈现了高级和抽象的使用模式，其本身就是表达式语言的应用使用的示例，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，而不直接 意见。

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://en.wikipedia.org/wiki/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其底层存储模型中刷新。 另一个从文字模式和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耗的消息。

A successful application may be constructed using the Expression Language exclusively, though the application will need to define its own system of translating application concepts into individual database messages and from individual database result sets. Alternatively, an application constructed with the ORM may, in advanced scenarios, make occasional usage of the Expression Language directly in certain areas where specific database interactions are required.

可以使用表达式语言专门构建成功的应用程序，尽管应用程序需要定义自己的将应用程序概念转换为单个数据库消息和单个数据库结果集的系统。 或者，使用ORM构建的应用程序可以在高级情况下，直接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. The tutorial has no prerequisites.

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。 本教程没有先决条件。

1.1 Version Check

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

**>>> import** **sqlalchemy**

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

1.1.0

### 1.2 Connecting

For this tutorial we will use an in-memory-only SQLite database. This is an easy way to test things without needing to have an actual database defined anywhere. 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。 这是一种简单的测试方法，无需在任何地方定义实际的数据库。 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的标准日志记录模块完成。 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，以免我们的方式; 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() 的返回值是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，该方法处理数据库和正在使用的DBAPI的详细信息。 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.

引擎首次被create\_engine（）返回时，实际上还没有试图连接到数据库; 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.

第一次调用像Engine.execute（）或Engine.connect（）这样的方法时，引擎会建立到数据库的真实DBAPI连接，然后用于发出SQL。

**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.

数据库网址 - 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 Define and Create Tables

The SQL Expression Language constructs its expressions in most cases against table columns. In SQLAlchemy, a column is most often represented by an object called [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column), and in all cases a [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) is associated with a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table). A collection of [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects and their associated child objects is referred to as ****database metadata****. In this tutorial we will explicitly lay out several [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects, but note that SA can also "import" whole sets of [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects automatically from an existing database (this process is called ****table reflection****).

在大多数情况下，SQL表达式语言在表列中构造其表达式。 在SQLAlchemy中，列通常由一个名为[Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column)的对象表示，在所有情况下，一[Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column)与一个[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)相关联。 [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)对象及其关联子对象的集合称为数据库元数据。 在本教程中，我们将明确地列出几个[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)对象，但请注意，SA还可以从现有数据库中自动导入整组[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)对象（此过程称为表反射）。

We define our tables all within a catalog called [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData), using the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) construct, which resembles regular SQL CREATE TABLE statements. We'll make two tables, one of which represents "users" in an application, and another which represents zero or more "email addresses" for each row in the "users" table:

我们使用表结构定义我们的表全部在名为[MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData)的目录中，它类似于常规的SQL CREATE [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)语句。 我们将创建两个表，其中一个表示应用程序中的"用户"，另一个表示"users"表中每一行的零个或多个"电子邮件地址"：

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

**>>>** metadata = MetaData()

**>>>** users = Table('users', metadata,

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

**...**  Column('name', String)

,**...**  Column('fullname', String),

**...** )

**>>>** addresses = Table('addresses', metadata,

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

**...**  Column('user\_id', None, ForeignKey('users.id')),

**...**  Column('email\_address', String, nullable=False)

**...**  )

All about how to define [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects, as well as how to create them from an existing database automatically, is described in [Describing Databases with MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html).

关于如何定义Table对象，以及如何从现有的数据库自动创建它们，在“使用MetaData描述数据库”中进行了描述。

Next, to tell the [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) we'd actually like to create our selection of tables for real inside the SQLite database, we use [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), passing it the engineinstance which points to our database. This will check for the presence of each table first before creating, so it's safe to call multiple times:

接下来，为了告诉MetaData我们实际上希望在SQLite数据库中创建我们选择的表，我们使用create\_all（），并将它指向我们的数据库的engineinstance。 这将在创建之前检查每个表的存在，因此多次调用是安全的：

**>>>** metadata.create\_all(engine)SE...

CREATE TABLE users (

id INTEGER NOT NULL,

name VARCHAR,

fullname VARCHAR,

PRIMARY KEY (id)

)

()

COMMIT

CREATE TABLE addresses (

id INTEGER NOT NULL,

user\_id INTEGER,

email\_address VARCHAR NOT NULL,

PRIMARY KEY (id),

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

)

()

COMMIT

**Note**

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列的生成没有长度; 在SQLite和PostgreSQL上，这是一个有效的数据类型，但在其他情况下，这是不允许的。 因此，如果在其中一个数据库上运行本教程，并且希望使用SQLAlchemy发出CREATE TABLE，则可以为String类型提供“length”，如下所示：

Column('name', 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.

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:

除了创建表格之外，String上的length字段以及Integer，Numeric等上可用的类似的precision / scale字段不会被SQLAlchemy引用。

此外，Firebird和Oracle需要序列来生成新的主键标识符，并且SQLAlchemy不会在未经指示的情况下生成或假定这些标识符。 为此，您使用Sequence构造：

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

Column('id', 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) is therefore:

users = Table('users', metadata,

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

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

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

Column('password', String(12)))

We include this more verbose [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) construct 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.

1.4 Insert Expressions

The first SQL expression we'll create is the [Insert](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert" \o "sqlalchemy.sql.expression.Insert) construct, which represents an INSERT statement. This is typically created relative to its target table:

**>>>** ins = users.insert()

To see a sample of the SQL this construct produces, use the str() function:

**>>>** str(ins)

'INSERT INTO users (id, name, fullname) VALUES (:id, :name, :fullname)'

Notice above that the INSERT statement names every column in the users table. This can be limited by using the values() method, which establishes the VALUES clause of the INSERT explicitly:

**>>>** ins = users.insert().values(name='jack', fullname='Jack Jones')

**>>>** str(ins)'INSERT INTO users (name, fullname) VALUES (:name, :fullname)'

Above, while the values method limited the VALUES clause to just two columns, the actual data we placed in values didn't get rendered into the string; instead we got named bind parameters. As it turns out, our data *is* stored within our [Insert](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert" \o "sqlalchemy.sql.expression.Insert) construct, but it typically only comes out when the statement is actually executed; since the data consists of literal values, SQLAlchemy automatically generates bind parameters for them. We can peek at this data for now by looking at the compiled form of the statement:

以上，当values方法将VALUES子句限制为只有两列时，我们放在values中的实际数据没有被渲染到字符串中; 而是我们得到了命名绑定参数。 事实证明，我们的数据存储在我们的Insert结构中，但通常只有当语句实际执行时才会出现; 由于数据由文字值组成，SQLAlchemy会自动为其生成绑定参数。 我们现在可以通过查看语句的编译形式来窥探这些数据：

**>>>** ins.compile().params

{'fullname': 'Jack Jones', 'name': 'jack'}

### 1.5 Executing

The interesting part of an [Insert](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert" \o "sqlalchemy.sql.expression.Insert) is executing it. In this tutorial, we will generally focus on the most explicit method of executing a SQL construct, and later touch upon some "shortcut" ways to do it. The engine object we created is a repository for database connections capable of issuing SQL to the database. To acquire a connection, we use the connect() method:

[Insert](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert" \o "sqlalchemy.sql.expression.Insert)的有趣部分就是执行它。 在本教程中，我们通常将重点介绍执行SQL构造的最显而易见的方法，然后再介绍一些"快捷方式"来实现。 我们创建的engine对象是一个能够向数据库发出SQL的数据库连接的存储库。 要获取连接，我们使用connect()方法：

**>>>** conn = engine.connect()

**>>>** conn

<sqlalchemy.engine.base.Connection object at 0x...>

The [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) object represents an actively checked out DBAPI connection resource. Lets feed it our [Insert](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert" \o "sqlalchemy.sql.expression.Insert) object and see what happens:

Connection对象表示主动签出的DBAPI连接资源。 让我们给我们Insert对象，看看会发生什么：

**>>>** result = conn.execute(ins)

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

('jack', 'Jack Jones')

COMMIT

So the INSERT statement was now issued to the database. Although we got positional "qmark" bind parameters instead of "named" bind parameters in the output. How come ? Because when executed, the [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) used the SQLite ****dialect**** to help generate the statement; when we use the str() function, the statement isn't aware of this dialect, and falls back onto a default which uses named parameters. We can view this manually as follows:

所以INSERT语句现在被发布到数据库。 虽然我们在输出中获得了位置"qmark"绑定参数，而不是"命名"绑定参数。 怎么来的 ？ 因为执行时，[Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection)使用SQLite方言来帮助生成语句; 当我们使用str() 函数时，语句不知道这个方言，并且返回到使用命名参数的默认值。 我们可以手动查看如下：

**>>>** ins.bind = engine

**>>>** str(ins)'INSERT INTO users (name, fullname) VALUES (?, ?)'

What about the result variable we got when we called execute() ? As the SQLAlchemy [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) object references a DBAPI connection, the result, known as a [ResultProxy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy" \o "sqlalchemy.engine.ResultProxy) object, is analogous to the DBAPI cursor object. In the case of an INSERT, we can get important information from it, such as the primary key values which were generated from our statement using [ResultProxy.inserted\_primary\_key](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.inserted_primary_key" \o "sqlalchemy.engine.ResultProxy.inserted_primary_key):

当我们调用execute() 时，我们得到的result变量呢？ 由于SQLAlchemy [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection)对象引用了DBAPI连接，所以称为ResultProxy对象的结果类似于DBAPI游标对象。 在INSERT的情况下，我们可以从中获取重要信息，例如使用[ResultProxy.inserted\_primary\_key](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.inserted_primary_key" \o "sqlalchemy.engine.ResultProxy.inserted_primary_key)从我们的语句生成的主键值：

**>>>** result.inserted\_primary\_key[1]

The value of 1 was automatically generated by SQLite, but only because we did not specify the id column in our [Insert](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert" \o "sqlalchemy.sql.expression.Insert) statement; otherwise, our explicit value would have been used. In either case, SQLAlchemy always knows how to get at a newly generated primary key value, even though the method of generating them is different across different databases; each database's [Dialect](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Dialect" \o "sqlalchemy.engine.interfaces.Dialect) knows the specific steps needed to determine the correct value (or values; note that[ResultProxy.inserted\_primary\_key](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.inserted_primary_key" \o "sqlalchemy.engine.ResultProxy.inserted_primary_key) returns a list so that it supports composite primary keys). Methods here range from using cursor.lastrowid, to selecting from a database-specific function, to using INSERT..RETURNING syntax; this all occurs transparently.

值为1由SQLite自动生成，但仅因为我们未在[Insert](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert" \o "sqlalchemy.sql.expression.Insert)语句中指定id列; 否则，我们的显式值将被使用。 在任一情况下，SQLAlchemy总是知道如何获取新生成的主键值，即使生成它们的方法在不同数据库之间是不同的; 每个数据库的Dialect知道确定正确值所需的具体步骤;（或多个值;请注意，[ResultProxy.inserted\_primary\_key](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.inserted_primary_key" \o "sqlalchemy.engine.ResultProxy.inserted_primary_key)返回一个列表，以支持复合主键）。 这里的方法范围从使用cursor.lastrowid，从数据库特定的功能中选择，使用INSERT..RETURNING语法; 这一切都是透明的。

1.6 Executing Multiple Statements

Our insert example above was intentionally a little drawn out to show some various behaviors of expression language constructs. In the usual case, an [Insert](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert" \o "sqlalchemy.sql.expression.Insert) statement is usually compiled against the parameters sent to the execute() method on [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection), so that there's no need to use the values keyword with [Insert](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert" \o "sqlalchemy.sql.expression.Insert). Lets create a generic [Insert](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert" \o "sqlalchemy.sql.expression.Insert) statement again and use it in the "normal" way:

我们上面插入的例子是有意引导出来表达语言结构的各种行为。 在通常的情况下，通常会根据[Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection)上发送给execute()方法的参数来编译[Insert](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert" \o "sqlalchemy.sql.expression.Insert)语句，因此不需要使用带有[Insert](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert" \o "sqlalchemy.sql.expression.Insert)键的values关键字。 让我们再次创建一个通用的[Insert](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert" \o "sqlalchemy.sql.expression.Insert)语句，并以"正常"的方式使用它：

**>>>** ins = users.insert()

**>>>** conn.execute(ins, id=2, name='wendy', fullname='Wendy Williams')

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

(2, 'wendy', 'Wendy Williams')

COMMIT

<sqlalchemy.engine.result.ResultProxy object at 0x...>

Above, because we specified all three columns in the execute() method, the compiled [Insert](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert" \o "sqlalchemy.sql.expression.Insert) included all three columns. The [Insert](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert" \o "sqlalchemy.sql.expression.Insert) statement is compiled at execution time based on the parameters we specified; if we specified fewer parameters, the [Insert](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert" \o "sqlalchemy.sql.expression.Insert) would have fewer entries in its VALUES clause.

以上，因为我们在execute()方法中指定了所有三列，所以编译的Insert包含了所有三列。 Insert语句根据我们指定的参数在执行时编译; 如果我们指定了较少的参数，那么Insert在其VALUES子句中将具有较少的条目。

To issue many inserts using DBAPI's executemany() method, we can send in a list of dictionaries each containing a distinct set of parameters to be inserted, as we do here to add some email addresses:

要使用DBAPI的executemany()方法发出许多插入，我们可以发送一个字典的列表，每个字典包含一个不同的参数集，如我们在这里添加一些电子邮件地址：

**>>>** conn.execute(addresses.insert(), [

**...**  {'user\_id': 1, 'email\_address' : ['jack@yahoo.com'}](mailto:'jack@yahoo.com'})

,**...**  {'user\_id': 1, 'email\_address' : ['jack@msn.com'},](mailto:'jack@msn.com'},)

**...**  {'user\_id': 2, 'email\_address' : ['www@www.org'},](mailto:'www@www.org'},)

**...**  {'user\_id': 2, 'email\_address' : ['wendy@aol.com'},](mailto:'wendy@aol.com'},)

**...** ])

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

((1, 'jack@yahoo.com'), (1, 'jack@msn.com'), (2, 'www@www.org'), (2, 'wendy@aol.com'))

COMMIT

<sqlalchemy.engine.result.ResultProxy object at 0x...>

Above, we again relied upon SQLite's automatic generation of primary key identifiers for each addresses row.

以上，我们再次依赖于SQLite自动生成每个addresses行的主键标识符。

When executing multiple sets of parameters, each dictionary must have the ****same**** set of keys; i.e. you cant have fewer keys in some dictionaries than others. This is because the [Insert](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert" \o "sqlalchemy.sql.expression.Insert) statement is compiled against the ****first**** dictionary in the list, and it's assumed that all subsequent argument dictionaries are compatible with that statement.

The "executemany" style of invocation is available for each of the [insert()](http://docs.sqlalchemy.org/en/rel_1_1/dialects/postgresql.html" \l "sqlalchemy.dialects.postgresql.dml.insert" \o "sqlalchemy.dialects.postgresql.dml.insert), [update()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.update" \o "sqlalchemy.sql.expression.update) and [delete()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.delete" \o "sqlalchemy.sql.expression.delete) constructs.

当执行多组参数时，每个字典必须具有相同的一组键; 也就是说，在某些字典中，你不能拥有比其他字典更少的键。 这是因为[Insert](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert" \o "sqlalchemy.sql.expression.Insert)语句是针对列表中的第一个字典编译的，并且假设所有后续的参数字典都与该语句兼容。

每个[insert()](http://docs.sqlalchemy.org/en/rel_1_1/dialects/postgresql.html" \l "sqlalchemy.dialects.postgresql.dml.insert" \o "sqlalchemy.dialects.postgresql.dml.insert)，[update()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.update" \o "sqlalchemy.sql.expression.update)和[delete()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.delete" \o "sqlalchemy.sql.expression.delete)结构都可以使用"executemany"的调用方式。

1.7 Selecting

We began with inserts just so that our test database had some data in it. The more interesting part of the data is selecting it! We'll cover UPDATE and DELETE statements later. The primary construct used to generate SELECT statements is the [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) function:

我们从插入开始，以便我们的测试数据库中有一些数据。 数据越有趣的部分就是选择它！ 稍后我们将介绍UPDATE和DELETE语句。 用于生成SELECT语句的主要构造是[select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select)函数：

**>>> from** **sqlalchemy.sql** **import** select

**>>>** s = select([users])

**>>>** result = conn.execute(s)

SELECT users.id, users.name, users.fullname

FROM users

()

Above, we issued a basic [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) call, placing the users table within the COLUMNS clause of the select, and then executing. SQLAlchemy expanded the users table into the set of each of its columns, and also generated a FROM clause for us. The result returned is again a [ResultProxy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy" \o "sqlalchemy.engine.ResultProxy) object, which acts much like a DBAPI cursor, including methods such as [fetchone()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.fetchone" \o "sqlalchemy.engine.ResultProxy.fetchone) and [fetchall()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.fetchall" \o "sqlalchemy.engine.ResultProxy.fetchall). The easiest way to get rows from it is to just iterate:

以上，我们发出了一个基本的[select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select)调用，将users表放在select的COLUMNS子句中，然后执行。 SQLAlchemy将users 表扩展为每个列的集合，并为我们生成了一个FROM子句。 返回的结果又是一个[ResultProxy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy" \o "sqlalchemy.engine.ResultProxy)对象，它的行为非常像一个DBAPI游标，包括诸如[fetchone()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.fetchone" \o "sqlalchemy.engine.ResultProxy.fetchone)和[fetchall()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.fetchall" \o "sqlalchemy.engine.ResultProxy.fetchall)之类的方法。 从它获取行的最简单的方法是迭代：

**>>> for** row **in** result:

**...**  **print**(row)

(1, u'jack', u'Jack Jones')(2, u'wendy', u'Wendy Williams')

Above, we see that printing each row produces a simple tuple-like result. We have more options at accessing the data in each row. One very common way is through dictionary access, using the string names of columns:

以上，我们看到打印每行会产生一个简单的元组结果。 我们有更多的选择来访问每一行的数据。 一个很常见的方式是通过字典访问，使用字符串名称列：

**>>>** result = conn.execute(s)

SELECT users.id, users.name, users.fullname

FROM users

()

**>>>** row = result.fetchone()

**>>> print**("name:", row['name'], "; fullname:", row['fullname'])

name: jack ; fullname: Jack Jones

Integer indexes work as well:

**>>>** row = result.fetchone()

**>>> print**("name:", row[1], "; fullname:", row[2])

name: wendy ; fullname: Wendy Williams

But another way, whose usefulness will become apparent later on, is to use the [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects directly as keys:

**>>> for** row **in** conn.execute(s):

**...**  **print**("name:", row[users.c.name], "; fullname:", row[users.c.fullname])

SELECT users.id, users.name, users.fullname

FROM users

()

name: jack ; fullname: Jack Jones

name: wendy ; fullname: Wendy Williams

Result sets which have pending rows remaining should be explicitly closed before discarding. While the cursor and connection resources referenced by the[ResultProxy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy" \o "sqlalchemy.engine.ResultProxy) will be respectively closed and returned to the connection pool when the object is garbage collected, it's better to make it explicit as some database APIs are very picky about such things:

剩余待处理行的结果集应在丢弃之前明确关闭。 虽然当对象被垃圾回收时，由[ResultProxy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy" \o "sqlalchemy.engine.ResultProxy)引用的游标和连接资源将被分别关闭并返回到连接池，最好使它显式，因为一些数据库API非常挑剔这样的事情：

**>>>** result.close()

If we'd like to more carefully control the columns which are placed in the COLUMNS clause of the select, we reference individual [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects from our [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table). These are available as named attributes off the c attribute of the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object:

如果我们想要更仔细地控制放置在select的COLUMNS子句中的列，那么我们从[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)中引用各个[Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column)对象。 这些可用作为[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)对象的c属性的命名属性：

**>>>** s = select([users.c.name, users.c.fullname])

**>>>** result = conn.execute(s)

SELECT users.name, users.fullname

FROM users

()

**>>> for** row **in** result:

**...**  **print**(row)

(u'jack', u'Jack Jones')

(u'wendy', u'Wendy Williams')

Lets observe something interesting about the FROM clause. Whereas the generated statement contains two distinct sections, a "SELECT columns" part and a "FROM table" part, our [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) construct only has a list containing columns. How does this work ? Let's try putting *two* tables into our [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) statement:

让我们观察一下有关FROM子句的内容。 而生成的语句包含两个不同的部分，一个"SELECT columns"部分和一个"FROM table"部分，我们的[select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select)结构只有一个包含列的列表。 这个怎么用 ？ 我们尝试将两个表放入我们的[select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select)语句中：

**>>> for** row **in** conn.execute(select([users, addresses])):

**...**  **print**(row)

SELECT users.id, users.name, users.fullname, addresses.id, addresses.user\_id, addresses.email\_address

FROM users, addresses

()

(1, u'jack', u'Jack Jones', 1, 1, [u'jack@yahoo.com')](mailto:u'jack@yahoo.com'))

(1, u'jack', u'Jack Jones', 2, 1, [u'jack@msn.com')](mailto:u'jack@msn.com'))

(1, u'jack', u'Jack Jones', 3, 2, [u'www@www.org')](mailto:u'www@www.org'))

(1, u'jack', u'Jack Jones', 4, 2, [u'wendy@aol.com')](mailto:u'wendy@aol.com'))

(2, u'wendy', u'Wendy Williams', 1, 1, [u'jack@yahoo.com')](mailto:u'jack@yahoo.com'))

(2, u'wendy', u'Wendy Williams', 2, 1, [u'jack@msn.com')](mailto:u'jack@msn.com'))

(2, u'wendy', u'Wendy Williams', 3, 2, [u'www@www.org')](mailto:u'www@www.org'))

(2, u'wendy', u'Wendy Williams', 4, 2, u'wendy@aol.com')

It placed ****both**** tables into the FROM clause. But also, it made a real mess. Those who are familiar with SQL joins know that this is a ****Cartesian product****; each row from the users table is produced against each row from the addresses table. So to put some sanity into this statement, we need a WHERE clause. We do that using [Select.where()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.where" \o "sqlalchemy.sql.expression.Select.where):

它将两个表放入FROM子句中。 但也是一个真正的混乱。 熟悉SQL连接的人知道这是一个笛卡尔积分产品; 来自users表的每一行都是针对addresses表中的每一行生成的。 所以为了在这个语句中放一些理智，我们需要一个WHERE子句。 我们使用[Select.where()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.where" \o "sqlalchemy.sql.expression.Select.where)：

**>>>** s = select([users, addresses]).where(users.c.id == addresses.c.user\_id)

**>>> for** row **in** conn.execute(s):

**...**  **print**(row)

SELECT users.id, users.name, users.fullname, addresses.id,

addresses.user\_id, addresses.email\_address

FROM users, addresses

WHERE users.id = addresses.user\_id

()

(1, u'jack', u'Jack Jones', 1, 1, [u'jack@yahoo.com')](mailto:u'jack@yahoo.com'))

(1, u'jack', u'Jack Jones', 2, 1, [u'jack@msn.com')](mailto:u'jack@msn.com'))

(2, u'wendy', u'Wendy Williams', 3, 2, [u'www@www.org')](mailto:u'www@www.org'))

(2, u'wendy', u'Wendy Williams', 4, 2, u'wendy@aol.com')

So that looks a lot better, we added an expression to our [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) which had the effect of adding WHERE users.id = addresses.user\_id to our statement, and our results were managed down so that the join of users and addresses rows made sense. But let's look at that expression? It's using just a Python equality operator between two different [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects. It should be clear that something is up. Saying 1 == 1 produces True, and 1 == 2 produces False, not a WHERE clause. So lets see exactly what that expression is doing:

所以看起来好多了，我们向我们的[select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select)添加了一个表达式，这个表达式有效地将WHERE users.id = addresses.user\_id添加到我们的语句中，并且我们的结果被管理，以便使users和addresses行的连接合理。 但是我们来看看那个表达？ 它在两个不同的Column对象之间只使用一个Python相等运算符。 应该清楚的是，事情已经到了。 说1 == 1产生True，1 == 2产生False，而不是WHERE子句。 所以让我们看看这个表达式正在做什么：

**>>>** users.c.id == addresses.c.user\_id

<sqlalchemy.sql.elements.BinaryExpression object at 0x...>

Wow, surprise ! This is neither a True nor a False. Well what is it ?

**>>>** str(users.c.id == addresses.c.user\_id)

'users.id = addresses.user\_id'

As you can see, the == operator is producing an object that is very much like the [Insert](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert" \o "sqlalchemy.sql.expression.Insert) and [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) objects we've made so far, thanks to Python's \_\_eq\_\_()builtin; you call str() on it and it produces SQL. By now, one can see that everything we are working with is ultimately the same type of object. SQLAlchemy terms the base class of all of these expressions as [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement).

如你所见，==操作符正在生成一个非常类似于我们迄今为止所做的[Insert](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert" \o "sqlalchemy.sql.expression.Insert)和[select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select)的对象，这要归功于Python的\_\_eq\_\_()内置函数; 你在其上调用了str()，并生成SQL。 到目前为止，我们可以看到，我们正在使用的一切都是最终相同类型的对象。 SQLAlchemy将所有这些表达式的基类称为[ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement).。

1.8 Operators

Since we've stumbled upon SQLAlchemy's operator paradigm, let's go through some of its capabilities. We've seen how to equate two columns to each other:

由于我们绊倒了SQLAlchemy的运算符范例，我们来看看它的一些功能。 我们已经看到了如何让两列相互等价：

**>>> print**(users.c.id == addresses.c.user\_id)

users.id = addresses.user\_id

If we use a literal value (a literal meaning, not a SQLAlchemy clause object), we get a bind parameter:

**>>> print**(users.c.id == 7)

users.id = :id\_1

The 7 literal is embedded the resulting [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement); we can use the same trick we did with the [Insert](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert" \o "sqlalchemy.sql.expression.Insert) object to see it:

**>>>** (users.c.id == 7).compile().params

{u'id\_1': 7}

Most Python operators, as it turns out, produce a SQL expression here, like equals, not equals, etc.:

**>>> print**(users.c.id != 7)

users.id != :id\_1

**>>>** *# None converts to IS NULL*

**>>> print**(users.c.name == None)

users.name IS NULL

**>>>** *# reverse works too*

**>>> print**('fred' > users.c.name)

users.name < :name\_1

If we add two integer columns together, we get an addition expression:

**>>> print**(users.c.id + addresses.c.id)

users.id + addresses.id

Interestingly, the type of the [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) is important! If we use + with two string based columns (recall we put types like [Integer](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Integer" \o "sqlalchemy.types.Integer) and [String](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.String" \o "sqlalchemy.types.String) on our [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects at the beginning), we get something different:

有趣的是，[Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column)的类型很重要！ 如果我们使用两个基于字符串的列（回想起来，我们在开始时将类型像[Integer](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Integer" \o "sqlalchemy.types.Integer)和[String](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.String" \o "sqlalchemy.types.String)放在[Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column)对象上），我们得到了一些不同的东西：

**>>> print**(users.c.name + users.c.fullname)

users.name || users.fullname

Where || is the string concatenation operator used on most databases. But not all of them. MySQL users, fear not:

其中|| 是大多数数据库使用的字符串连接运算符。 但不是全部。 MySQL用户不怕：

**>>> print**((users.c.name + users.c.fullname).

**...**  compile(bind=create\_engine('mysql://')))

concat(users.name, users.fullname)

The above illustrates the SQL that's generated for an [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) that's connected to a MySQL database; the || operator now compiles as MySQL's concat() function.

以上说明了为连接到MySQL数据库的引擎生成的SQL;该|| 运算符现在编译为MySQL的concat()函数。

If you have come across an operator which really isn't available, you can always 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) method; this generates whatever operator you need:

如果您遇到真正不可用的操作员，则可以随时使用[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)方法; 这将产生您需要的任何操作符：

**>>> print**(users.c.name.op('tiddlywinks')('foo'))

users.name tiddlywinks :name\_1

This function can also be used to make bitwise operators explicit. For example:

此功能也可显式用于使按位运算符。 例如：

somecolumn.op('&')(0xff)

is a bitwise AND of the value in somecolumn.

是一个在somecolumn值按位的AND。

When using [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), the return type of the expression may be important, especialy when the operator is used in an expression that will be sent as a result column. For this case, be sure to make the type explicit, if not what's normally expected, using [type\_coerce()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.type_coerce" \o "sqlalchemy.sql.expression.type_coerce):

当使用[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)时，表达式的返回类型可能很重要，特别是在将作为结果列发送的表达式中使用运算符时。 对于这种情况，请确保使用[type\_coerce()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.type_coerce" \o "sqlalchemy.sql.expression.type_coerce)使类型显式（如果不是通常期望的）：

**from** **sqlalchemy** **import** type\_coerce

expr = type\_coerce(somecolumn.op('-%>')('foo'), MySpecialType())

stmt = select([expr])

### Operator Customization

While [ColumnOperators.op()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.op" \o "sqlalchemy.sql.operators.ColumnOperators.op) is handy to get at a custom operator in a hurry, the Core supports fundamental customization and extension of the operator system at the type level. The behavior of existing operators can be modified on a per-type basis, and new operations can be defined which become available for all column expressions that are part of that particular type. 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.

虽然[ColumnOperators.op()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.op" \o "sqlalchemy.sql.operators.ColumnOperators.op)方便快捷地获取定制操作员，但Core支持在类型级别的操作员系统的基本定制和扩展。 现有操作符的行为可以在每种类型的基础上进行修改，并且可以定义新的操作，这些操作对于属于该特定类型的所有列表达式都可用。 有关说明，请参阅重新定义和创建新操作符。

1.9 Conjunctions

We'd like to show off some of our operators inside of [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) constructs. But we need to lump them together a little more, so let's first introduce some conjunctions. Conjunctions are those little words like AND and OR that put things together. We'll also hit upon NOT. [and\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.and_" \o "sqlalchemy.sql.expression.and_), [or\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.or_" \o "sqlalchemy.sql.expression.or_), and [not\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.not_" \o "sqlalchemy.sql.expression.not_) can work from the corresponding functions SQLAlchemy provides (notice we also throw in a [like()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.like" \o "sqlalchemy.sql.operators.ColumnOperators.like)):

我们希望在select() 结构中炫耀我们的一些操作符。 但是我们需要把它们聚在一起，所以让我们先来介绍一些连词。 连词是AND和OR把小东西放在一起的小字。 我们也打了NOT。 and\_() ，or\_() 和not\_() 可以从SQLAlchemy提供的相应函数中工作（请注意，我们还抛出一个like() ）：

**>>> from** **sqlalchemy.sql** **import** and\_, or\_, not\_

**>>> print**(and\_(

**...**  users.c.name.like('j%'),

**...**  users.c.id == addresses.c.user\_id,

**...**  or\_(

**...**  addresses.c.email\_address == ['wendy@aol.com',](mailto:'wendy@aol.com',)

**...**  addresses.c.email\_address == ['jack@yahoo.com'](mailto:'jack@yahoo.com')

**...**  ),

**...**  not\_(users.c.id > 5)

**...**  )

**...**  )users.name LIKE :name\_1 AND users.id = addresses.user\_id AND(addresses.email\_address = :email\_address\_1 OR addresses.email\_address = :email\_address\_2)AND users.id <= :id\_1

And you can also use the re-jiggered bitwise AND, OR and NOT operators, although because of Python operator precedence you have to watch your parenthesis:

你也可以使用重复的按位AND，OR和NOT运算符，但是由于Python运算符的优先级，你必须注意括号：

**>>> print**(users.c.name.like('j%') & (users.c.id == addresses.c.user\_id) &

**...**  (

**...**  (addresses.c.email\_address == 'wendy@aol.com') | \

**...**  (addresses.c.email\_address == ['jack@yahoo.com')](mailto:'jack@yahoo.com'))

**...**  ) \

**...**  & ~(users.c.id>5)

**...** )

users.name LIKE :name\_1 AND users.id = addresses.user\_id AND(addresses.email\_address = :email\_address\_1 OR addresses.email\_address = :email\_address\_2)AND users.id <= :id\_1

So with all of this vocabulary, let's select all users who have an email address at AOL or MSN, whose name starts with a letter between "m" and "z", and we'll also generate a column containing their full name combined with their email address. We will add two new constructs to this statement, [between()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.between" \o "sqlalchemy.sql.operators.ColumnOperators.between) and [label()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement.label" \o "sqlalchemy.sql.expression.ColumnElement.label).[between()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.between" \o "sqlalchemy.sql.operators.ColumnOperators.between) produces a BETWEEN clause, and [label()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement.label" \o "sqlalchemy.sql.expression.ColumnElement.label) is used in a column expression to produce labels using the AS keyword; it's recommended when selecting from expressions that otherwise would not have a name:

所有这些词汇，让我们选择所有的用户，他们的姓名以"m"和"z"之间的一个字母开头的AOL或MSN的电子邮件地址，我们还将生成一个包含全名的列 他们的邮箱地址。 我们添加两个新结构，[between()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.between" \o "sqlalchemy.sql.operators.ColumnOperators.between)和[label()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement.label" \o "sqlalchemy.sql.expression.ColumnElement.label).[between()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.between" \o "sqlalchemy.sql.operators.ColumnOperators.between)产生一个BETWEEN子句，而在一个列表达式中使用了[label()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement.label" \o "sqlalchemy.sql.expression.ColumnElement.label) 来产生使用AS关键字的标签; 建议从表达式中选择否则不具有名称：

**>>>** s = select([(users.c.fullname +

**...**  ", " + addresses.c.email\_address).

**...**  label('title')]).\

**...**  where(

**...**  and\_(

**...**  users.c.id == addresses.c.user\_id,

**...**  users.c.name.between('m', 'z'),

**...**  or\_(**...**  [addresses.c.email\_address.like('%@aol.com'),](mailto:addresses.c.email_address.like('%@aol.com'),)

**...**  [addresses.c.email\_address.like('%@msn.com')](mailto:addresses.c.email_address.like('%@msn.com'))

**...**  )

**...**  )

**...**  )

**>>>** conn.execute(s).fetchall()

SELECT users.fullname || ? || addresses.email\_address AS title

FROM users, addresses

WHERE users.id = addresses.user\_id

AND users.name BETWEEN ? AND ? AND(addresses.email\_address LIKE ? OR addresses.email\_address LIKE ?)(', ', 'm', 'z', '%@aol.com', '%@msn.com')[(u'Wendy Williams, wendy@aol.com',)]

Once again, SQLAlchemy figured out the FROM clause for our statement. In fact it will determine the FROM clause based on all of its other bits; the columns clause, the where clause, and also some other elements which we haven't covered yet, which include ORDER BY, GROUP BY, and HAVING.

再次，SQLAlchemy为我们的语句找出了FROM子句。 事实上，它将根据其所有其他位确定FROM子句; columns子句，where子句以及我们尚未涵盖的其他元素，其中包括ORDER BY，GROUP BY和HAVING。

A shortcut to using [and\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.and_" \o "sqlalchemy.sql.expression.and_) is to chain together multiple [where()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.where" \o "sqlalchemy.sql.expression.Select.where) clauses. The above can also be written as:

使用[and\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.and_" \o "sqlalchemy.sql.expression.and_) 的快捷方式是将多个[where()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.where" \o "sqlalchemy.sql.expression.Select.where)子句链接在一起。 以上也可以写成：

**>>>** s = select([(users.c.fullname +

**...**  ", " + addresses.c.email\_address).

**...**  label('title')]).\

**...**  where(users.c.id == addresses.c.user\_id).\

**...**  where(users.c.name.between('m', 'z')).\

**...**  where(**...**  or\_(

**...**  [addresses.c.email\_address.like('%@aol.com'),](mailto:addresses.c.email_address.like('%@aol.com'),)

**...**  [addresses.c.email\_address.like('%@msn.com')](mailto:addresses.c.email_address.like('%@msn.com'))

**...**  )

**...**  )

**>>>** conn.execute(s).fetchall()

SELECT users.fullname || ? || addresses.email\_address AS title

FROM users, addresses

WHERE users.id = addresses.user\_id AND users.name BETWEEN ? AND ? AND(addresses.email\_address LIKE ? OR addresses.email\_address LIKE ?)(', ', 'm', 'z', '%@aol.com', ['%@msn.com')](mailto:'%@msn.com'))

[(u'Wendy Williams, wendy@aol.com',)]

The way that we can build up a [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) construct through successive method calls is called [method chaining](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-method-chaining).

我们可以通过连续的方法调用来构建select（）构造的方式称为方法链。

1.10 Using Textual SQL

Our last example really became a handful to type. Going from what one understands to be a textual SQL expression into a Python construct which groups components together in a programmatic style can be hard. That's why SQLAlchemy lets you just use strings, for those cases when the SQL is already known and there isn't a strong need for the statement to support dynamic features. The [text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text) construct is used to compose a textual statement that is passed to the database mostly unchanged. Below, we create a [text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text) object and execute it:

我们最后一个例子真的成了一个很少的类型。 从一个人所理解的，将一个文本的SQL表达式变成一个将程序化风格组合在一起的Python构造可能很难。 这就是为什么SQLAlchemy允许你只是使用字符串，对于那些情况，当SQL已经知道，并没有强烈需要语句来支持动态特性。 text() 结构用于组合一个文本语句，该文本语句几乎不改变。 下面我们创建一个text() 对象并执行它：

**>>> from** **sqlalchemy.sql** **import** text

**>>>** s = text(

**...**  "SELECT users.fullname || ', ' || addresses.email\_address AS title "

**...**  "FROM users, addresses "

**...**  "WHERE users.id = addresses.user\_id "

**...**  "AND users.name BETWEEN :x AND :y "

**...**  "AND (addresses.email\_address LIKE :e1 "

**...**  "OR addresses.email\_address LIKE :e2)")

**>>>** conn.execute(s, x='m', y='z', e1='%@aol.com', e2='%@msn.com').fetchall()

SELECT users.fullname || ', ' || addresses.email\_address AS title

FROM users, addresses

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

(addresses.email\_address LIKE ? OR addresses.email\_address LIKE ?)

('m', 'z', '%@aol.com', '%@msn.com')

[(u'Wendy Williams, wendy@aol.com',)]

Above, we can see that bound parameters are specified in [text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text) using the named colon format; this format is consistent regardless of database backend. To send values in for the parameters, we passed them into the [execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.execute" \o "sqlalchemy.engine.Connection.execute) method as additional arguments.

以上，我们可以看到使用命名冒号格式在[text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text)中指定了绑定的参数; 无论数据库后端如何，此格式都是一致的。 要为参数发送值，我们将它们作为附加参数传递给[execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.execute" \o "sqlalchemy.engine.Connection.execute)方法。

### 1.10.1 Specifying Bound Parameter Behaviors

The [text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text) construct supports pre-established bound values using the [TextClause.bindparams()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.TextClause.bindparams" \o "sqlalchemy.sql.expression.TextClause.bindparams) method:

stmt = text("SELECT \* FROM users WHERE users.name BETWEEN :x AND :y")

stmt = stmt.bindparams(x="m", y="z")

The parameters can also be explicitly typed:

stmt = stmt.bindparams(bindparam("x", String), bindparam("y", String))

result = conn.execute(stmt, {"x": "m", "y": "z"})

Typing for bound parameters is necessary when the type requires Python-side or special SQL-side processing provided by the datatype.

当类型需要Python边或由数据类型提供的特殊SQL边处理时，键入绑定参数是必需的。

**See also**

[TextClause.bindparams()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.TextClause.bindparams" \o "sqlalchemy.sql.expression.TextClause.bindparams) - full method description

### 1.10.2 Specifying Result-Column Behaviors

We may also specify information about the result columns using 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; this method can be used to specify the return types, based on name:

我们也可以使用TextClause.columns() 方法来指定关于结果列的信息; 此方法可用于指定返回类型，基于名称：

stmt = stmt.columns(id=Integer, name=String)

or it can be passed full column expressions positionally, either typed or untyped. In this case it's a good idea to list out the columns explicitly within our textual SQL, since the correlation of our column expressions to the SQL will be done positionally:

或者可以通过字段传递完整的列表达式，无论是键入还是非类型化。 在这种情况下，最好在文本SQL中显式列出列，因为我们的列表达式与SQL的关联将在位置上完成：

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

stmt = stmt.columns(users.c.id, users.c.name)

When we call 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, we get back a [TextAsFrom](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TextAsFrom" \o "sqlalchemy.sql.expression.TextAsFrom) object that supports the full suite of [TextAsFrom.c](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TextAsFrom.c" \o "sqlalchemy.sql.expression.TextAsFrom.c) and other "selectable" operations:

j = stmt.join(addresses, stmt.c.id == addresses.c.user\_id)

new\_stmt = select([stmt.c.id, addresses.c.id]).\

select\_from(j).where(stmt.c.name == 'x')

The positional form of [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) is particularly useful when relating textual SQL to existing Core or ORM models, because we can use column expressions directly without worrying about name conflicts or other issues with the result column names in the textual SQL:

当将文本SQL与现有的Core或ORM模型相关联时，TextClause.columns() 的位置形式特别有用，因为我们可以直接使用列表达式，而不必担心文本SQL中的结果列名称的名称冲突或其他问题：

**>>>** stmt = text("SELECT users.id, addresses.id, users.id, "

**...**  "users.name, addresses.email\_address AS email "

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

**...**  "WHERE users.id = 1").columns(

**...**  users.c.id,

**...**  addresses.c.id,

**...**  addresses.c.user\_id,

**...**  users.c.name,

**...**  addresses.c.email\_address

**...**  )

**>>>** result = conn.execute(stmt)

SELECT users.id, addresses.id, users.id, users.name,

addresses.email\_address AS email

FROM users JOIN addresses ON users.id=addresses.user\_id WHERE users.id = 1

()

Above, there's three columns in the result that are named "id", but since we've associated these with column expressions positionally, the names aren't an issue when the result-columns are fetched using the actual column object as a key. Fetching the email\_address column would be:

以上，结果中有三列名为"id"，但是由于我们已经将这些列与位置相关联，所以当使用实际列对象作为关键字获取结果列时，名称不是问题。 获取email\_address列将是：

**>>>** row = result.fetchone()

**>>>** row[addresses.c.email\_address]'jack@yahoo.com'

If on the other hand we used a string column key, the usual rules of name- based matching still apply, and we'd get an ambiguous column error for the id value:

如果另一方面我们使用了一个字符串列键，通常的基于名称的匹配规则仍然适用，我们会得到一个模糊的列错误的id值：

**>>>** row["id"]

Traceback (most recent call last):

*...*InvalidRequestError: Ambiguous column name 'id' in result set column descriptions

It's important to note that while accessing columns from a result set using [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects may seem unusual, it is in fact the only system used by the ORM, which occurs transparently beneath the facade of the [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) object; in this way, 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 is typically very applicable to textual statements to be used in an ORM context. The example at [Using Textual SQL](http://docs.sqlalchemy.org/en/rel_1_1/orm/tutorial.html" \l "orm-tutorial-literal-sql) illustrates a simple usage.

重要的是要注意，使用Column对象从结果集中访问列可能看起来是不寻常的，它实际上是由ORM使用的唯一系统，它在Query对象的立面下透明地出现; 以这种方式，TextClause.columns() 方法通常非常适用于要在ORM上下文中使用的文本语句。 使用文本SQL的示例说明了一个简单的用法。

*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 when matching table metadata or ORM models to textual SQL.

1.1版中的新功能：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，消除了在匹配表元数据时SQL列中匹配或甚至唯一的列名称， ORM模型到文本SQL。

**See also**

[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) - full method description

[Using Textual SQL](http://docs.sqlalchemy.org/en/rel_1_1/orm/tutorial.html" \l "orm-tutorial-literal-sql) - integrating ORM-level queries with [text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text)

### 1.10.3 Using text() fragments inside bigger statements

[text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text) can also be used to produce fragments of SQL that can be freely within a [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) object, which accepts [text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text) objects as an argument for most of its builder functions. Below, we combine the usage of [text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text) within a [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) object. The [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) construct provides the "geometry" of the statement, and the [text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text) construct provides the textual content within this form. We can build a statement without the need to refer to any pre-established [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) metadata:

text() 也可以用于生成可以在[select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) 对象内自由的SQL片段，它可以接受[text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text) 对象作为大部分构建器函数的参数。 下面我们将[select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) 对象中的[text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text) 的使用相结合。 [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) 构造提供了语句的"几何"，而[text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text) 结构提供了此表单中的文本内容。 我们可以构建一个语句，而无需参考任何预先建立的表元数据：

**>>>** s = select([

**...**  text("users.fullname || ', ' || addresses.email\_address AS title")

**...**  ]).\

**...**  where(

**...**  and\_(

**...**  text("users.id = addresses.user\_id"),

**...**  text("users.name BETWEEN 'm' AND 'z'"),

**...**  text(

**...**  "(addresses.email\_address LIKE :x "

**...**  "OR addresses.email\_address LIKE :y)")

**...**  )

**...**  ).select\_from(text('users, addresses'))

**>>>** conn.execute(s, x='%@aol.com', y='%@msn.com').fetchall()

SELECT users.fullname || ', ' || addresses.email\_address AS title

FROM users, addresses

WHERE users.id = addresses.user\_id AND users.name BETWEEN 'm' AND 'z'

AND (addresses.email\_address LIKE ? OR addresses.email\_address LIKE ?)

('%@aol.com', '%@msn.com')

[(u'Wendy Williams, wendy@aol.com',)]

*Changed in version 1.0.0:*The [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) construct emits warnings when string SQL fragments are coerced to [text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text), and [text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text) should be used explicitly. See [Warnings emitted when coercing full SQL fragments into text()](http://docs.sqlalchemy.org/en/rel_1_1/changelog/migration_10.html" \l "migration-2992) for background.

### 1.10.4 Using More Specific Text with [table()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.table" \o "sqlalchemy.sql.expression.table), [literal\_column()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.literal_column" \o "sqlalchemy.sql.expression.literal_column), and [column()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.column" \o "sqlalchemy.sql.expression.column)

We can move our level of structure back in the other direction too, by using [column()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.column" \o "sqlalchemy.sql.expression.column), [literal\_column()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.literal_column" \o "sqlalchemy.sql.expression.literal_column), and [table()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.table" \o "sqlalchemy.sql.expression.table) for some of the key elements of our statement. Using these constructs, we can get some more expression capabilities than if we used [text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text) directly, as they provide to the Core more information about how the strings they store are to be used, but still without the need to get into full [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) based metadata. Below, we also specify the [String](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.String" \o "sqlalchemy.types.String) datatype for two of the key [literal\_column()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.literal_column" \o "sqlalchemy.sql.expression.literal_column) objects, so that the string-specific concatenation operator becomes available. We also use [literal\_column()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.literal_column" \o "sqlalchemy.sql.expression.literal_column) in order to use table-qualified expressions, e.g. users.fullname, that will be rendered as is; using [column()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.column" \o "sqlalchemy.sql.expression.column) implies an individual column name that may be quoted:

我们可以通过对我们语句的一些关键元素使用column() ，literal\_column() 和table() 来将我们的结构层次推向另一个方向。 使用这些结构，我们可以获得更多的表达能力，而不是直接使用text() ，因为它们向Core提供了有关如何使用它们存储的字符串的更多信息，但仍然无需进入完整的基于表元数据。 下面，我们还为两个关键的literal\_column() 对象指定了String数据类型，以便字符串特定的级联运算符可用。 我们还使用literal\_column() 来使用表格限定的表达式，例如 users.fullname，将被呈现为; 使用column() 表示可能引用的单个列名称：

**>>> from** **sqlalchemy** **import** select, and\_, text, String

**>>> from** **sqlalchemy.sql** **import** table, literal\_column

**>>>** s = select([

**...**  literal\_column("users.fullname", String) +

**...**  ', ' +

**...**  literal\_column("addresses.email\_address").label("title")

**...** ]).\

**...**  where(

**...**  and\_(

**...**  literal\_column("users.id") == literal\_column("addresses.user\_id"),

**...**  text("users.name BETWEEN 'm' AND 'z'"),

**...**  text(

**...**  "(addresses.email\_address LIKE :x OR "

**...**  "addresses.email\_address LIKE :y)")

**...**  )

**...**  ).select\_from(table('users')).select\_from(table('addresses'))

**>>>** conn.execute(s, x='%@aol.com', y='%@msn.com').fetchall()

SELECT users.fullname || ? || addresses.email\_address AS anon\_1

FROM users, addresses

WHERE users.id = addresses.user\_id

AND users.name BETWEEN 'm' AND 'z'

AND (addresses.email\_address LIKE ? OR addresses.email\_address LIKE ?)

(', ', '%@aol.com', '%@msn.com')

[(u'Wendy Williams, wendy@aol.com',)]

### 1.10.5 Ordering or Grouping by a Label

One place where we sometimes want to use a string as a shortcut is when our statement has some labeled column element that we want to refer to in a place such as the "ORDER BY" or "GROUP BY" clause; other candidates include fields within an "OVER" or "DISTINCT" clause. If we have such a label in our [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) construct, we can refer to it directly by passing the string straight into select.order\_by() or select.group\_by(), among others. This will refer to the named label and also prevent the expression from being rendered twice:

我们有时希望使用字符串作为捷径的一个地方是当我们的语句有一些我们想在诸如"ORDER BY"或"GROUP BY"子句之类的地方引用的标签列元素时; 其他候选人包括"OVER"或"DISTINCT"子句中的字段。 如果我们在[select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select)结构中有这样一个标签，我们可以直接通过将字符串直接传递给select.order\_by()或select.group\_by()等来引用它。 这将引用命名的标签，并且也阻止表达式被渲染两次：

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

**>>>** stmt = select([

**...**  addresses.c.user\_id,

**...**  func.count(addresses.c.id).label('num\_addresses')]).\

**...**  order\_by("num\_addresses")

**>>>** conn.execute(stmt).fetchall()

SELECT addresses.user\_id, count(addresses.id) AS num\_addresses

FROM addresses ORDER BY num\_addresses

()

[(2, 4)]

We can use modifiers like [asc()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.asc" \o "sqlalchemy.sql.expression.asc) or [desc()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.desc" \o "sqlalchemy.sql.expression.desc) by passing the string name:

**>>> from** **sqlalchemy** **import** func, desc

**>>>** stmt = select([

**...**  addresses.c.user\_id,

**...**  func.count(addresses.c.id).label('num\_addresses')]).\

**...**  order\_by(desc("num\_addresses"))

**>>>** conn.execute(stmt).fetchall()

SELECT addresses.user\_id, count(addresses.id) AS num\_addresses

FROM addresses ORDER BY num\_addresses DESC

()

[(2, 4)]

Note that the string feature here is very much tailored to when we have already used 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) method to create a specifically-named label. In other cases, we always want to refer to the [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) object directly so that the expression system can make the most effective choices for rendering. Below, we illustrate how using the [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) eliminates ambiguity when we want to order by a column name that appears more than once:

请注意，这里的字符串特征非常适合当我们已经使用[label()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement.label" \o "sqlalchemy.sql.expression.ColumnElement.label)方法创建一个特定命名的标签时。 在其他情况下，我们总是直接引用[ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement)对象，以使表达式系统能够最有效地进行渲染。 下面我们将说明当我们想通过不止一次出现的列名进行排序时，如何使用[ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement)消除歧义：

**>>>** u1a, u1b = users.alias(), users.alias()

**>>>** stmt = select([u1a, u1b]).\

**...**  where(u1a.c.name > u1b.c.name).\

**...**  order\_by(u1a.c.name) *# using "name" here would be ambiguous*

**>>>** conn.execute(stmt).fetchall()

SELECT users\_1.id, users\_1.name, users\_1.fullname, users\_2.id,

users\_2.name, users\_2.fullname

FROM users AS users\_1, users AS users\_2

WHERE users\_1.name > users\_2.name ORDER BY users\_1.name

()

[(2, u'wendy', u'Wendy Williams', 1, u'jack', u'Jack Jones')]

## 1.11 Using Aliases

The alias in SQL corresponds to a "renamed" version of a table or SELECT statement, which occurs anytime you say "SELECT .. FROM sometable AS someothername". The AS creates a new name for the table. Aliases are a key construct as they allow any table or subquery to be referenced by a unique name. In the case of a table, this allows the same table to be named in the FROM clause multiple times. In the case of a SELECT statement, it provides a parent name for the columns represented by the statement, allowing them to be referenced relative to this name.

SQL中的别名对应于表或SELECT语句的"重命名"版本，只要您说"SELECT .. FROM sometable AS someothername"，就会发生。 AS为表创建一个新名称。别名是一个关键结构，因为它们允许任何表或子查询被唯一的名称引用。在表的情况下，这允许在FROM子句中多次命名相同的表。在SELECT语句的情况下，它为由语句表示的列提供父名称，允许相对于此名称引用它们。

In SQLAlchemy, any [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table), [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) construct, or other selectable can be turned into an alias 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, which produces a [Alias](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Alias" \o "sqlalchemy.sql.expression.Alias) construct. As an example, suppose we know that our user jack has two particular email addresses. How can we locate jack based on the combination of those two addresses? To accomplish this, we'd use a join to the addresses table, once for each address. We create two [Alias](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Alias" \o "sqlalchemy.sql.expression.Alias) constructs against addresses, and then use them both within 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SQLAlchemy中，可以使用[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)方法将任何一个[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)，[select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select)结构或其他可选项转换为别名，该方法生成一个[Alias](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Alias" \o "sqlalchemy.sql.expression.Alias)构造。例如，假设我们知道我们的用户jack有两个特定的电子邮件地址。我们如何根据这两个地址的组合来定位jack？为了实现这一点，我们将使用一个addresses表的连接，每个地址一次。我们针对地址创建两个Alias构造，然后在一个select() 结构中使用它们：

**>>>** a1 = addresses.alias()

**>>>** a2 = addresses.alias()

**>>>** s = select([users]).\

**...**  where(and\_(

**...**  users.c.id == a1.c.user\_id,

**...**  users.c.id == a2.c.user\_id,

**...**  a1.c.email\_address == ['jack@msn.com',](mailto:'jack@msn.com',)

**...**  a2.c.email\_address == ['jack@yahoo.com'](mailto:'jack@yahoo.com')

**...**  ))

**>>>** conn.execute(s).fetchall()

SELECT users.id, users.name, users.fullname

FROM users, addresses AS addresses\_1, addresses AS addresses\_2

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

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

AND addresses\_1.email\_address = ?

AND addresses\_2.email\_address = ?

('jack@msn.com', 'jack@yahoo.com')

[(1, u'jack', u'Jack Jones')]

Note that 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 the names addresses\_1 and addresses\_2 in the final SQL result. The generation of these names is determined by the position of the construct within the statement. If we created a query using only the second a2 alias, the name would come out as addresses\_1. The generation of the names is also *deterministic*, meaning the same SQLAlchemy statement construct will produce the identical SQL string each time it is rendered for a particular dialect.

请注意，Alias构造在最终SQL结果中生成名称addresses\_1和addresses\_2。 这些名称的生成由语句中的构造的位置决定。 如果我们仅使用第二个a2别名创建了一个查询，则该名称将作为addresses\_1出现。 名称的生成也是确定性的，这意味着相同的SQLAlchemy语句构造将在每次为特定方言呈现时生成相同的SQL字符串。

Since on the outside, we refer to the alias using the [Alias](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Alias" \o "sqlalchemy.sql.expression.Alias) construct itself, we don't need to be concerned about the generated name. However, for the purposes of debugging, it can be specified by passing a string name 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) method:

因为在外面，我们使用Alias构造本身引用别名，我们不需要关心生成的名称。 但是，为了进行调试，可以通过将字符串名称传递给FromClause.alias() 方法来指定：

**>>>** a1 = addresses.alias('a1')

Aliases can of course be used for anything which you can SELECT from, including SELECT statements themselves. We can self-join the users table back to the [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) we've created by making an alias of the entire statement. The correlate(None) directive is to avoid SQLAlchemy's attempt to "correlate" the inner users table with the outer one:

别名当然可以用于您可以从中选择的任何内容，包括SELECT语句本身。 我们可以通过创建整个语句的别名，将用户表自己加入到我们创建的select() 中。 相关（无）指令是避免SQLAlchemy尝试将内部用户表与外部表进行"关联"：

**>>>** a1 = s.correlate(None).alias()

**>>>** s = select([users.c.name]).where(users.c.id == a1.c.id)

**>>>** conn.execute(s).fetchall()

SELECT users.name

FROM users,

(SELECT users.id AS id, users.name AS name, users.fullname AS fullname

FROM users, addresses AS addresses\_1, addresses AS addresses\_2

WHERE users.id = addresses\_1.user\_id AND users.id = addresses\_2.user\_id

AND addresses\_1.email\_address = ?

AND addresses\_2.email\_address = ?) AS anon\_1

WHERE users.id = anon\_1.id

('jack@msn.com', 'jack@yahoo.com')

[(u'jack',)]

## 1.12 Using Joins

We're halfway along to being able to construct any SELECT expression. The next cornerstone of the SELECT is the JOIN expression. We've already been doing joins in our examples, by just placing two tables in either the columns clause or the where clause of the [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) construct. But if we want to make a real "JOIN" or "OUTERJOIN" construct, we use the [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 [outerjoin()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.outerjoin" \o "sqlalchemy.sql.expression.FromClause.outerjoin) methods, most commonly accessed from the left table in the join:

我们中途可以构建任何SELECT表达式。 SELECT的下一个基石是JOIN表达式。 我们已经在我们的示例中进行了连接，只需将两个表放在SELECT() 结构的columns子句或where子句中即可。 但是，如果我们想要创建一个真正的"JOIN"或"OUTERJOIN"结构，我们使用join() 和outerjoin() 方法，最常见的是从连接的左侧表中访问：

**>>> print**(users.join(addresses))

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

The alert reader will see more surprises; SQLAlchemy figured out how to JOIN the two tables ! The ON condition of the join, as it's called, was automatically generated based on the [ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey) object which we placed on the addresses table way at the beginning of this tutorial. Already the join() construct is looking like a much better way to join tables.

警报读者会看到更多的惊喜; SQLAlchemy想出如何加入两个表！ 根据本教程开头，我们在地址表上放置的ForeignKey对象自动生成连接的ON条件，因为它被称为。 已经是join() 结构看起来像是一个更好的连接表的方式。

Of course you can join on whatever expression you want, such as if we want to join on all users who use the same name in their email address as their username:

当然，你可以加入任何你想要的表达方式，例如我们想要加入他们的电子邮件地址中使用相同名称的所有用户的用户名：

**>>> print**(users.join(addresses,

**...**  addresses.c.email\_address.like(users.c.name + '%')

**...**  )

**...**  )users JOIN addresses ON addresses.email\_address LIKE (users.name || :name\_1)

When we create a [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) construct, SQLAlchemy looks around at the tables we've mentioned and then places them in the FROM clause of the statement. When we use JOINs however, we know what FROM clause we want, so here we make use of the [select\_from()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.select_from" \o "sqlalchemy.sql.expression.Select.select_from) method:

当我们创建一个select() 结构时，SQLAlchemy会看到我们提到的表，然后将它们放在语句的FROM子句中。 当我们使用JOIN时，我们知道我们想要什么FROM子句，所以这里我们使用select\_from() 方法：

**>>>** s = select([users.c.fullname]).select\_from(

**...**  users.join(addresses,

**...**  addresses.c.email\_address.like(users.c.name + '%'))

**...**  )

**>>>** conn.execute(s).fetchall()

SELECT users.fullname

FROM users JOIN addresses ON addresses.email\_address LIKE (users.name || ?)

('%',)

[(u'Jack Jones',), (u'Jack Jones',), (u'Wendy Williams',)]

The [outerjoin()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.outerjoin" \o "sqlalchemy.sql.expression.FromClause.outerjoin) method creates LEFT OUTER JOIN constructs, and is used in the same way as [join()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.join" \o "sqlalchemy.sql.expression.FromClause.join):

**>>>** s = select([users.c.fullname]).select\_from(users.outerjoin(addresses))

**>>> print**(s)

SELECT users.fullname

FROM users

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

That's the output outerjoin() produces, unless, of course, you're stuck in a gig using Oracle prior to version 9, and you've set up your engine (which would be using OracleDialect) to use Oracle-specific SQL:

这就是输出outerjoin() 产生的，除非你在9版之前使用Oracle，而且你已经设置了引擎（它将使用OracleDialect）来使用Oracle特定的SQL：

**>>> from** **sqlalchemy.dialects.oracle** **import** dialect **as** OracleDialect

**>>> print**(s.compile(dialect=OracleDialect(use\_ansi=False)))

SELECT users.fullname

FROM users, addresses

WHERE users.id = addresses.user\_id(+)

If you don't know what that SQL means, don't worry ! The secret tribe of Oracle DBAs don't want their black magic being found out ;).

如果你不知道这是什么意思，不用担心！ Oracle DBA的秘密部落不希望发现他们的黑魔法;）。

**See also**

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

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

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

1.13 Everything Else

The concepts of creating SQL expressions have been introduced. What's left are more variants of the same themes. So now we'll catalog the rest of the important things we'll need to know.

已经介绍了创建SQL表达式的概念。 剩下的是相同主题的更多变体。 所以现在我们将列出我们需要知道的其他重要事项。

### 1.13.1 Bind Parameter Objects

Throughout all these examples, SQLAlchemy is busy creating bind parameters wherever literal expressions occur. You can also specify your own bind parameters with your own names, and use the same statement repeatedly. The [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) construct is used to produce a bound parameter with a given name. While SQLAlchemy always refers to bound parameters by name on the API side, the database dialect converts to the appropriate named or positional style at execution time, as here where it converts to positional for SQLite:

在所有这些示例中，SQLAlchemy正忙于在文字表达式发生的任何地方创建绑定参数。 您还可以使用自己的名称指定自己的绑定参数，并重复使用相同的语句。 bindparam() 构造用于产生具有给定名称的绑定参数。 虽然SQLAlchemy总是在API端指定绑定的参数，但数据库方言在执行时转换为适当的命名或位置样式，就像在SQLite中将其转换为位置一样：

**>>> from** **sqlalchemy.sql** **import** bindparam

**>>>** s = users.select(users.c.name == bindparam('username'))

**>>>** conn.execute(s, username='wendy').fetchall()

SELECT users.id, users.name, users.fullname

FROM users

WHERE users.name = ?

('wendy',)

[(2, u'wendy', u'Wendy Williams')]

Another important aspect of [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) is that it may be assigned a type. The type of the bind parameter will determine its behavior within expressions and also how the data bound to it is processed before being sent off to the database:

bindparam() 的另一个重要方面是它可以被分配一个类型。 绑定参数的类型将确定其在表达式中的行为，以及在发送到数据库之前如何处理绑定到它的数据：

**>>>** s = users.select(users.c.name.like(bindparam('username', type\_=String) + text("'%'")))

**>>>** conn.execute(s, username='wendy').fetchall()

SELECT users.id, users.name, users.fullname

FROM users

WHERE users.name LIKE (? || '%')

('wendy',)

[(2, u'wendy', u'Wendy Williams')]

[bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) constructs of the same name can also be used multiple times, where only a single named value is needed in the execute parameters:

相同名称的bindparam() 构造也可以多次使用，在execute参数中只需要一个命名值：

**>>>** s = select([users, addresses]).\

**...**  where(

**...**  or\_(

**...**  users.c.name.like(

**...**  bindparam('name', type\_=String) + text("'%'")),

**...**  addresses.c.email\_address.like(

**...**  bindparam('name', type\_=String) + text("'@%'"))

**...**  )

**...**  ).\

**...**  select\_from(users.outerjoin(addresses)).\

**...**  order\_by(addresses.c.id)

**>>>** conn.execute(s, name='jack').fetchall()

SELECT users.id, users.name, users.fullname, addresses.id,

addresses.user\_id, addresses.email\_address

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

WHERE users.name LIKE (? || '%') OR addresses.email\_address LIKE (? || '@%')

ORDER BY addresses.id

('jack', 'jack')

[(1, u'jack', u'Jack Jones', 1, 1, u'jack@yahoo.com'), (1, u'jack', u'Jack Jones', 2, 1, u'jack@msn.com')]

**See also**

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

### 1.13.2 Functions

SQL functions are created using the [func](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.func" \o "sqlalchemy.sql.expression.func) keyword, which generates functions using attribute access:

使用func关键字创建SQL函数，该函数使用属性访问生成函数：

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

**>>> print**(func.now())

now()

**>>> print**(func.concat('x', 'y'))

concat(:concat\_1, :concat\_2)

By "generates", we mean that ****any**** SQL function is created based on the word you choose:

通过“生成”，我们意味着任何SQL函数都是基于您选择的单词创建的：

**>>>** print(func.xyz\_my\_goofy\_function())

xyz\_my\_goofy\_function()

Certain function names are known by SQLAlchemy, allowing special behavioral rules to be applied. Some for example are "ANSI" functions, which mean they don't get the parenthesis added after them, such as CURRENT\_TIMESTAMP:

SQLAlchemy已知某些函数名称，允许应用特殊的行为规则。 一些例如是"ANSI"函数，这意味着它们不会在它们之后添加括号，例如CURRENT\_TIMESTAMP：

**>>> print**(func.current\_timestamp())

CURRENT\_TIMESTAMP

Functions are most typically used in the columns clause of a select statement, and can also be labeled as well as given a type. Labeling a function is recommended so that the result can be targeted in a result row based on a string name, and assigning it a type is required when you need result-set processing to occur, such as for Unicode conversion and date conversions. Below, we use the result function scalar() to just read the first column of the first row and then close the result; the label, even though present, is not important in this case:

函数最常用于select语句的columns子句，也可以给出类型。 建议使用标签功能，以便根据字符串名称将结果定位到结果行中，并且在需要执行结果集处理（例如Unicode转换和日期转换）时，需要分配一个类型。 下面我们用结果函数scalar() 来读取第一行的第一列，然后关闭结果; 标签即使是现在，在这种情况下也不重要：

**>>>** conn.execute(

**...**  select([

**...**  func.max(addresses.c.email\_address, type\_=String).

**...**  label('maxemail')

**...**  ])

**...**  ).scalar()

SELECT max(addresses.email\_address) AS maxemail

FROM addresses

()

u'www@www.org'

Databases such as PostgreSQL and Oracle which support functions that return whole result sets can be assembled into selectable units, which can be used in statements. Such as, a database function calculate() which takes the parameters x and y, and returns three columns which we'd like to name q, z and r, we can construct using "lexical" column objects as well as bind parameters:

支持返回整个结果集的函数的PostgreSQL和Oracle等数据库可以组合成可选择的单元，可以在语句中使用。 例如，一个数据库函数calculate() ，它接受参数x和y，并返回三个我们想命名为q，z和r的列，我们可以使用"lexical"列对象和绑定参数来构造：

**>>> from** **sqlalchemy.sql** **import** column

**>>>** calculate = select([column('q'), column('z'), column('r')]).\

**...**  select\_from(

**...**  func.calculate(

**...**  bindparam('x'),

**...**  bindparam('y')

**...**  )

**...**  )

**>>>** calc = calculate.alias()

**>>> print**(select([users]).where(users.c.id > calc.c.z))

SELECT users.id, users.name, users.fullname

FROM users, (

SELECT q, z, r

FROM calculate(:x, :y)

) AS anon\_1

WHERE users.id > anon\_1.z

If we wanted to use our calculate statement twice with different bind parameters, the [unique\_params()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.unique_params" \o "sqlalchemy.sql.expression.ClauseElement.unique_params) function will create copies for us, and mark the bind parameters as "unique" so that conflicting names are isolated. Note we also make two separate aliases of our selectable:

如果我们想使用不同绑定参数的两次计算语句，unique\_params() 函数将为我们创建副本，并将绑定参数标记为“唯一”，以便隔离冲突的名称。 请注意，我们还可以选择两个独立的别名：

**>>>** calc1 = calculate.alias('c1').unique\_params(x=17, y=45)

**>>>** calc2 = calculate.alias('c2').unique\_params(x=5, y=12)

**>>>** s = select([users]).\

**...**  where(users.c.id.between(calc1.c.z, calc2.c.z))

**>>> print**(s)SELECT users.id, users.name, users.fullname

FROM users,

(SELECT q, z, r FROM calculate(:x\_1, :y\_1)) AS c1,

(SELECT q, z, r FROM calculate(:x\_2, :y\_2)) AS c2

WHERE users.id BETWEEN c1.z AND c2.z

**>>>** s.compile().params

{u'x\_2': 5, u'y\_2': 12, u'y\_1': 45, u'x\_1': 17}

**See also**

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

### 1.13.3 Window Functions

Any [FunctionElement](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.FunctionElement" \o "sqlalchemy.sql.functions.FunctionElement), including functions generated by [func](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.func" \o "sqlalchemy.sql.expression.func), can be turned into a "window function", that is an OVER clause, using the [FunctionElement.over()](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.FunctionElement.over" \o "sqlalchemy.sql.functions.FunctionElement.over) method:

任何FunctionElement，包括由func生成的函数，都可以使用FunctionElement.over() 方法转换成一个“窗口函数”，即一个OVER子句。

**>>>** s = select([

**...**  users.c.id,

**...**  func.row\_number().over(order\_by=users.c.name)

**...**  ])

**>>>** print(s)

SELECT users.id, row\_number()

OVER (ORDER BY users.name) AS anon\_1

FROM users

[FunctionElement.over()](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.FunctionElement.over" \o "sqlalchemy.sql.functions.FunctionElement.over) also supports range specification using either the [expression.over.rows](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.over.params.rows" \o "sqlalchemy.sql.expression.over) or [expression.over.range](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.over.params.range" \o "sqlalchemy.sql.expression.over) parameters:

Function Element.over() 也支持使用expression.over.rows或expression.over.range参数的范围规范：

**>>>** s = select([

**...**  users.c.id,

**...**  func.row\_number().over(

**...**  order\_by=users.c.name,

**...**  rows=(-2, **None**))

**...**  ])

**>>>** print(s)

SELECT users.id, row\_number()

OVER(ORDER BY users.name ROWS BETWEEN :param\_1 PRECEDING AND UNBOUNDED FOLLOWING)

AS anon\_1FROM users

[expression.over.rows](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.over.params.rows" \o "sqlalchemy.sql.expression.over) and [expression.over.range](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.over.params.range" \o "sqlalchemy.sql.expression.over) each accept a two-tuple which contains a combination of negative and positive integers for ranges, zero to indicate "CURRENT ROW" and None to indicate "UNBOUNDED". See the examples at [over()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.over" \o "sqlalchemy.sql.expression.over) for more detail.

expression.over.rows和expression.over.range都接受一个两元组，它包含范围为负整数和正整数的组合，零表示"CURRENT ROW"，无表示"UNBOUNDED"。 有关更多详细信息，请参阅over() 的示例。

*New in version 1.1:*support for "rows" and "range" specification for window functions

版本1.1中的新功能：支持窗口功能的"行"和"范围"规范

**See also**

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

[FunctionElement.over()](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.FunctionElement.over" \o "sqlalchemy.sql.functions.FunctionElement.over)

### 1.13.4 Unions and Other Set Operations

Unions come in two flavors, UNION and UNION ALL, which are available via module level functions [union()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.union" \o "sqlalchemy.sql.expression.union) and [union\_all()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.union_all" \o "sqlalchemy.sql.expression.union_all):

联盟有两种方式，UNION和UNION ALL，可通过模块级别的函数union() 和union\_all() 获得：

**>>> from** **sqlalchemy.sql** **import** union

**>>>** u = union(

**...**  addresses.select().

**...**  where(addresses.c.email\_address == ['foo@bar.com'),](mailto:'foo@bar.com'),)

**...**  addresses.select().

**...**  [where(addresses.c.email\_address.like('%@yahoo.com')),](mailto:where(addresses.c.email_address.like('%@yahoo.com')),)

**...** ).order\_by(addresses.c.email\_address)

**>>>** conn.execute(u).fetchall()

SELECT addresses.id, addresses.user\_id, addresses.email\_address

FROM addresses

WHERE addresses.email\_address = ?

UNION

SELECT addresses.id, addresses.user\_id, addresses.email\_address

FROM addresses

WHERE addresses.email\_address LIKE ? ORDER BY addresses.email\_address

('foo@bar.com', '%@yahoo.com')

[(1, 1, u'jack@yahoo.com')]

Also available, though not supported on all databases, are [intersect()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.intersect" \o "sqlalchemy.sql.expression.intersect), [intersect\_all()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.intersect_all" \o "sqlalchemy.sql.expression.intersect_all), [except\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.except_" \o "sqlalchemy.sql.expression.except_), and [except\_all()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.except_all" \o "sqlalchemy.sql.expression.except_all):

虽然在所有数据库中都不支持，但也可以使用intersect() ，intersect\_all() ，except\_() 和except\_all() ：

**>>> from** **sqlalchemy.sql** **import** except\_

**>>>** u = except\_(

**...**  addresses.select().

**...**  [where(addresses.c.email\_address.like('%@%.com')),](mailto:where(addresses.c.email_address.like('%@%.com')),)

**...**  addresses.select().

**...**  [where(addresses.c.email\_address.like('%@msn.com'))](mailto:where(addresses.c.email_address.like('%@msn.com')))

**...** )

**>>>** conn.execute(u).fetchall(

)[(1, 1, u'jack@yahoo.com'), (4, 2, u'wendy@aol.com')]

A common issue with so-called "compound" selectables arises due to the fact that they nest with parenthesis. SQLite in particular doesn't like a statement that starts with parenthesis. So when nesting a "compound" inside a "compound", it's often necessary to apply .alias().select() to the first element of the outermost compound, if that element is also a compound. For example, to nest a "union" and a "select" inside of "except\_", SQLite will want the "union" to be stated as a subquery:

所谓"复合"可选项的常见问题是由于它们以括号嵌套的事实。 SQLite特别不喜欢以括号开头的语句。 因此，当将"化合物"嵌套在"化合物"中时，通常需要将.alias() 。select() 应用于最外层化合物的第一个元素，如果该元素也是化合物。 例如，要在"except\_"内嵌入"union"和"select"，SQLite将要将"union"声明为子查询：

**>>>** u = except\_(

**...**  union(

**...**  addresses.select().

**...**  [where(addresses.c.email\_address.like('%@yahoo.com')),](mailto:where(addresses.c.email_address.like('%@yahoo.com')),)

**...**  addresses.select().

**...**  [where(addresses.c.email\_address.like('%@msn.com'))](mailto:where(addresses.c.email_address.like('%@msn.com')))

**...**  ).alias().select(), *# apply subquery here*

**...**  [addresses.select(addresses.c.email\_address.like('%@msn.com'))](mailto:addresses.select(addresses.c.email_address.like('%@msn.com')))

**...** )

**>>>** conn.execute(u).fetchall()

SELECT anon\_1.id, anon\_1.user\_id, anon\_1.email\_address

FROM (SELECT addresses.id AS id, addresses.user\_id AS user\_id,

addresses.email\_address AS email\_address

FROM addresses

WHERE addresses.email\_address LIKE ?

UNION

SELECT addresses.id AS id,

addresses.user\_id AS user\_id,

addresses.email\_address AS email\_address

FROM addresses

WHERE addresses.email\_address LIKE ?) AS anon\_1

EXCEPT

SELECT addresses.id, addresses.user\_id, addresses.email\_address

FROM addresses

WHERE addresses.email\_address LIKE ?

('%@yahoo.com', '%@msn.com', '%@msn.com')

[(1, 1, u'jack@yahoo.com')]

**See also**

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

[union\_all()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.union_all" \o "sqlalchemy.sql.expression.union_all)

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

[intersect\_all()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.intersect_all" \o "sqlalchemy.sql.expression.intersect_all)

[except\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.except_" \o "sqlalchemy.sql.expression.except_)

[except\_all()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.except_all" \o "sqlalchemy.sql.expression.except_all)

### 1.13.5 Scalar Selects

A scalar select is a SELECT that returns exactly one row and one column. It can then be used as a column expression. A scalar select is often a [correlated subquery](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-correlated-subquery), which relies upon the enclosing SELECT statement in order to acquire at least one of its FROM clauses.

标量选择是一个SELECT，它只返回一行和一列。 然后可以将其用作列表达式。 标量选择通常是相关的子查询，它依赖于包含的SELECT语句，以获取其FROM子句中的至少一个。

The [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) construct can be modified to act as a column expression by calling either the [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) or [label()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.SelectBase.label" \o "sqlalchemy.sql.expression.SelectBase.label) method:

可以通过调用as\_scalar() 或label() 方法来修改select() 结构作为列表达式：

**>>>** stmt = select([func.count(addresses.c.id)]).\

**...**  where(users.c.id == addresses.c.user\_id).\

**...**  as\_scalar()

The above construct is now a [ScalarSelect](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.ScalarSelect" \o "sqlalchemy.sql.expression.ScalarSelect) object, and is no longer part of the [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) hierarchy; it instead is within the [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) family of expression constructs. We can place this construct the same as any other column within another [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select):

上述结构现在是ScalarSelect对象，不再是FromClause层次结构的一部分; 而是在ColumnElement系列的表达式构造之内。 我们可以将这个构造与另一个select() 中的任何其他列相同：

**>>>** conn.execute(select([users.c.name, stmt])).fetchall()

SELECT users.name, (SELECT count(addresses.id) AS count\_1

FROM addresses

WHERE users.id = addresses.user\_id) AS anon\_1

FROM users

()

[(u'jack', 2), (u'wendy', 2)]

To apply a non-anonymous column name to our scalar select, we create it using [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) instead:

要将非匿名列名应用于我们的标量选择，我们使用[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) 来创建它：

**>>>** stmt = select([func.count(addresses.c.id)]).\

**...**  where(users.c.id == addresses.c.user\_id).\

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

**>>>** conn.execute(select([users.c.name, stmt])).fetchall()

SELECT users.name, (SELECT count(addresses.id) AS count\_1

FROM addresses

WHERE users.id = addresses.user\_id) AS address\_count

FROM users

()

[(u'jack', 2), (u'wendy', 2)]

**See also**

[Select.as\_scalar()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.as_scalar" \o "sqlalchemy.sql.expression.Select.as_scalar)

[Select.label()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.label" \o "sqlalchemy.sql.expression.Select.label)

### 1.13.6 Correlated Subqueries

Notice in the examples on [Scalar Selects](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html" \l "scalar-selects), the FROM clause of each embedded select did not contain the users table in its FROM clause. This is because SQLAlchemy automatically [correlates](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-correlates) embedded FROM objects to that of an enclosing query, if present, and if the inner SELECT statement would still have at least one FROM clause of its own. For example:

注意在Scalar Selects的例子中，每个嵌入式select的FROM子句在其FROM子句中都不包含users表。 这是因为SQLAlchemy自动将嵌入的FROM对象与封闭查询的INS对象（如果存在）相关联，如果内部SELECT语句仍然至少具有一个自己的FROM子句。 例如：

**>>>** stmt = select([addresses.c.user\_id]).\

**...**  where(addresses.c.user\_id == users.c.id).\

**...**  where(addresses.c.email\_address == ['jack@yahoo.com')](mailto:'jack@yahoo.com'))

**>>>** enclosing\_stmt = select([users.c.name]).where(users.c.id == stmt)

**>>>** conn.execute(enclosing\_stmt).fetchall()

SELECT users.name

FROM users

WHERE users.id = (SELECT addresses.user\_id

FROM addresses

WHERE addresses.user\_id = users.id

AND addresses.email\_address = ?)

('jack@yahoo.com',)

[(u'jack',)]

Auto-correlation will usually do what's expected, however it can also be controlled. For example, if we wanted a statement to correlate only to the addresses table but not the users table, even if both were present in the enclosing SELECT, we use the [correlate()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.correlate" \o "sqlalchemy.sql.expression.Select.correlate) method to specify those FROM clauses that may be correlated:

自动相关通常会做预期的，但也可以控制。 例如，如果我们想要一个语句仅与地址表相关联，而不与users表相关联，即使两者都存在于包含的SELECT中，我们使用correlate() 方法来指定可能相关的那些FROM子句：

**>>>** stmt = select([users.c.id]).\

**...**  where(users.c.id == addresses.c.user\_id).\

**...**  where(users.c.name == 'jack').\

**...**  correlate(addresses)

**>>>** enclosing\_stmt = select(

**...**  [users.c.name, addresses.c.email\_address]).\

**...**  select\_from(users.join(addresses)).\

**...**  where(users.c.id == stmt)

**>>>** conn.execute(enclosing\_stmt).fetchall()

SELECT users.name, addresses.email\_address

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

WHERE users.id = (SELECT users.id

FROM users

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

('jack',)

[(u'jack', u'jack@yahoo.com'), (u'jack', u'jack@msn.com')]

To entirely disable a statement from correlating, we can pass None as the argument:

要完全禁用相关的语句，我们可以传递None作为参数：

**>>>** stmt = select([users.c.id]).\

**...**  where(users.c.name == 'wendy').\

**...**  correlate(None)

**>>>** enclosing\_stmt = select([users.c.name]).\

**...**  where(users.c.id == stmt)

**>>>** conn.execute(enclosing\_stmt).fetchall()

SELECT users.name

FROM users

WHERE users.id = (SELECT users.id

FROM users

WHERE users.name = ?)

('wendy',)

[(u'wendy',)]

We can also control correlation via exclusion, using the [Select.correlate\_except()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.correlate_except" \o "sqlalchemy.sql.expression.Select.correlate_except) method. Such as, we can write our SELECT for the users table by telling it to correlate all FROM clauses except for users:

我们也可以使用[Select.correlate\_except()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.correlate_except" \o "sqlalchemy.sql.expression.Select.correlate_except) 方法通过排除来控制关联。 例如，我们可以通过告诉它关联除users之外的所有FROM子句来为users表写入我们的SELECT：

**>>>** stmt = select([users.c.id]).\

**...**  where(users.c.id == addresses.c.user\_id).\

**...**  where(users.c.name == 'jack').\

**...**  correlate\_except(users)

**>>>** enclosing\_stmt = select(

**...**  [users.c.name, addresses.c.email\_address]).\

**...**  select\_from(users.join(addresses)).\

**...**  where(users.c.id == stmt)

**>>>** conn.execute(enclosing\_stmt).fetchall()

SELECT users.name, addresses.email\_address

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

WHERE users.id = (SELECT users.id

FROM users

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

('jack',)

[(u'jack', u'jack@yahoo.com'), (u'jack', u'jack@msn.com')]

#### LATERAL correlation

LATERAL correlation is a special sub-category of SQL correlation which allows a selectable unit to refer to another selectable unit within a single FROM clause. This is an extremely special use case which, while part of the SQL standard, is only known to be supported by recent versions of PostgreSQL.

LATERAL相关性是SQL相关的特殊子类别，允许可选单元引用单个FROM子句中的另一个可选单元。 这是一个非常特殊的用例，尽管SQL标准的一部分，仅被PostgreSQL的最新版本所支持。

Normally, if a SELECT statement refers to table1 JOIN (some SELECT) AS subquery in its FROM clause, the subquery on the right side may not refer to the "table1" expression from the left side; correlation may only refer to a table that is part of another SELECT that entirely encloses this SELECT. The LATERAL keyword allows us to turn this behavior around, allowing an expression such as:

通常，如果SELECT语句引用其FROM子句中的table1 JOIN（某些SELECT）AS子查询，则右侧的子查询可能不会从左侧引用“table1”表达式; 相关性可能只是指完全包围此SELECT的另一个SELECT的一部分的表。 LATERAL关键字可以让我们围绕这个行为，允许一个表达式，如：

**SELECT** people.people\_id, people.age, people.name

**FROM** people **JOIN** **LATERAL** (**SELECT** books.book\_id **AS** book\_id

**FROM** books **WHERE** books.owner\_id = people.people\_id)

**AS** book\_subq **ON**

**true**

Where above, the right side of the JOIN contains a subquery that refers not just to the "books" table but also the "people" table, correlating to the left side of the JOIN. SQLAlchemy Core supports a statement like the above using the [Select.lateral()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.lateral" \o "sqlalchemy.sql.expression.Select.lateral) method as follows:

在上面，JOIN的右侧包含一个子查询，它不仅指"书"表，还涉及"人"表，与JOIN的左侧相关。 SQLAlchemy Core支持如上所述的使用Select.lateral() 方法的语句如下：

**>>> from** **sqlalchemy** **import** table, column, select, true

**>>>** people = table('people', column('people\_id'), column('age'), column('name'))

**>>>** books = table('books', column('book\_id'), column('owner\_id'))

**>>>** subq = select([books.c.book\_id]).\

**...**  where(books.c.owner\_id == people.c.people\_id).lateral("book\_subq")

**>>>** print(select([people]).select\_from(people.join(subq, true())))

SELECT people.people\_id, people.age, people.name

FROM people JOIN LATERAL (SELECT books.book\_id AS book\_id

FROM books WHERE books.owner\_id = people.people\_id)

AS book\_subq ON true

Above, we can see that the [Select.lateral()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.lateral" \o "sqlalchemy.sql.expression.Select.lateral) method acts a lot like the [Select.alias()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.alias" \o "sqlalchemy.sql.expression.Select.alias) method, including that we can specify an optional name. However the construct is the [Lateral](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Lateral" \o "sqlalchemy.sql.expression.Lateral) construct instead of an [Alias](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Alias" \o "sqlalchemy.sql.expression.Alias) which provides for the LATERAL keyword as well as special instructions to allow correlation from inside the FROM clause of the enclosing statement.

以上，我们可以看到Select.lateral() 方法的行为非常像Select.alias() 方法，包括我们可以指定一个可选的名称。 然而，构造是Lateral构造而不是Alias，它提供了LATERAL关键字以及特殊的指令，允许在封闭语句的FROM子句内部进行相关。

The [Select.lateral()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.lateral" \o "sqlalchemy.sql.expression.Select.lateral) method interacts normally with the [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) and [Select.correlate\_except()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.correlate_except" \o "sqlalchemy.sql.expression.Select.correlate_except) methods, except that the correlation rules also apply to any other tables present in the enclosing statement's FROM clause. Correlation is "automatic" to these tables by default, is explicit if the table is specified 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), and is explicit to all tables except those specified to [Select.correlate\_except()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.correlate_except" \o "sqlalchemy.sql.expression.Select.correlate_except).

[Select.lateral()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.lateral" \o "sqlalchemy.sql.expression.Select.lateral)方法与[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)和[Select.correlate\_except()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.correlate_except" \o "sqlalchemy.sql.expression.Select.correlate_except)方法正常交互，除了相关规则也适用于封闭语句FROM子句中存在的任何其他表。 默认情况下，这些表相关是"自动的"，如果表被指定为[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)，则是显式的，并且除了为[Select.correlate\_except()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.correlate_except" \o "sqlalchemy.sql.expression.Select.correlate_except).指定的所有表之外，所有表都是显式的。

*New in version 1.1:*Support for the LATERAL keyword and lateral correlation.

**See also**

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

[Select.lateral()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.lateral" \o "sqlalchemy.sql.expression.Select.lateral)

### 1.13.7 Ordering, Grouping, Limiting, Offset…ing…

Ordering is done by passing column expressions to the order\_by() method:

通过将列表达式传递给order\_by()方法来完成排序：

**>>>** stmt = select([users.c.name]).order\_by(users.c.name)

**>>>** conn.execute(stmt).fetchall()

SELECT users.name

FROM users ORDER BY users.name

()

[(u'jack',), (u'wendy',)]

Ascending or descending can be controlled using the [asc()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement.asc" \o "sqlalchemy.sql.expression.ColumnElement.asc) and [desc()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement.desc" \o "sqlalchemy.sql.expression.ColumnElement.desc) modifiers:

可以使用[asc()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement.asc" \o "sqlalchemy.sql.expression.ColumnElement.asc)和[desc()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement.desc" \o "sqlalchemy.sql.expression.ColumnElement.desc)修饰符来控制升序或降序：

**>>>** stmt = select([users.c.name]).order\_by(users.c.name.desc())

**>>>** conn.execute(stmt).fetchall()

SELECT users.name

FROM users ORDER BY users.name DESC

()

[(u'wendy',), (u'jack',)]

Grouping refers to the GROUP BY clause, and is usually used in conjunction with aggregate functions to establish groups of rows to be aggregated. This is provided via the group\_by() method:

分组是指GROUP BY子句，通常与聚合函数一起使用以建立要聚合的行组。 这是通过group\_by()方法提供的：

**>>>** stmt = select([users.c.name, func.count(addresses.c.id)]).\

**...**  select\_from(users.join(addresses)).\

**...**  group\_by(users.c.name)

**>>>** conn.execute(stmt).fetchall()

SELECT users.name, count(addresses.id) AS count\_1

FROM users JOIN addresses

ON users.id = addresses.user\_id

GROUP BY users.name

()

[(u'jack', 2), (u'wendy', 2)]

HAVING can be used to filter results on an aggregate value, after GROUP BY has been applied. It's available here via the [having()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.having" \o "sqlalchemy.sql.expression.Select.having) method:

在应用GROUP BY之后，可以使用HAVING过滤汇总值上的结果。 这里可以通过[having()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.having" \o "sqlalchemy.sql.expression.Select.having)方法：

**>>>** stmt = select([users.c.name, func.count(addresses.c.id)]).\

**...**  select\_from(users.join(addresses)).\

**...**  group\_by(users.c.name).\

**...**  having(func.length(users.c.name) > 4)

**>>>** conn.execute(stmt).fetchall()

SELECT users.name, count(addresses.id) AS count\_1

FROM users JOIN addresses

ON users.id = addresses.user\_id

GROUP BY users.name

HAVING length(users.name) > ?

(4,)

[(u'wendy', 2)]

A common system of dealing with duplicates in composed SELECT statements is the DISTINCT modifier. A simple DISTINCT clause can be added using the [Select.distinct()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.distinct" \o "sqlalchemy.sql.expression.Select.distinct) method:

在组合的SELECT语句中处理复制的常见系统是DISTINCT修饰符。 可以使用[Select.distinct()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.distinct" \o "sqlalchemy.sql.expression.Select.distinct)方法添加一个简单的DISTINCT子句：

**>>>** stmt = select([users.c.name]).\

**...**  where(addresses.c.email\_address

.**...**  contains(users.c.name)).\

**...**  distinct()

**>>>** conn.execute(stmt).fetchall()

SELECT DISTINCT users.name

FROM users, addresses

WHERE (addresses.email\_address LIKE '%%' || users.name || '%%')

()

[(u'jack',), (u'wendy',)]

Most database backends support a system of limiting how many rows are returned, and the majority also feature a means of starting to return rows after a given "offset". While common backends like PostgreSQL, MySQL and SQLite support LIMIT and OFFSET keywords, other backends need to refer to more esoteric features such as "window functions" and row ids to achieve the same effect. The [limit()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.limit" \o "sqlalchemy.sql.expression.Select.limit) and [offset()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.offset" \o "sqlalchemy.sql.expression.Select.offset) methods provide an easy abstraction into the current backend's methodology:

大多数数据库后端支持限制返回多少行的系统，并且大多数还具有在给定"偏移量"之后开始返回行的方法。 虽然像PostgreSQL这样的常见后端，MySQL和SQLite支持LIMIT和OFFSET关键字，但其他后端需要引用更加深奥的功能，例如"窗口功能"和行ids来实现相同的效果。 [limit()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.limit" \o "sqlalchemy.sql.expression.Select.limit)和[offset()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.offset" \o "sqlalchemy.sql.expression.Select.offset)方法为当前后端的方法提供了一个简单的抽象：

**>>>** stmt = select([users.c.name, addresses.c.email\_address]).\

**...**  select\_from(users.join(addresses)).\

**...**  limit(1).offset(1)

**>>>** conn.execute(stmt).fetchall()

SELECT users.name, addresses.email\_address

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

LIMIT ? OFFSET ?

(1, 1)

[(u'jack', u'jack@msn.com')]

### 1.14 Inserts, Updates and Deletes

We've seen [insert()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableClause.insert" \o "sqlalchemy.sql.expression.TableClause.insert) demonstrated earlier in this tutorial. Where [insert()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableClause.insert" \o "sqlalchemy.sql.expression.TableClause.insert) produces INSERT, the [update()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableClause.update" \o "sqlalchemy.sql.expression.TableClause.update) method produces UPDATE. Both of these constructs feature a method called [values()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase.values" \o "sqlalchemy.sql.expression.ValuesBase.values) which specifies the VALUES or SET clause of the statement.

我们已经看到了本教程前面介绍的insert() 。 在insert() 生成INSERT的情况下，update() 方法生成UPDATE。 这两个构造都具有一个名为values() 的方法，它指定语句的VALUES或SET子句。

The [values()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase.values" \o "sqlalchemy.sql.expression.ValuesBase.values) method accommodates any column expression as a value:

values() 方法将任何列表达式适用于值：

**>>>** stmt = users.update().\

**...**  values(fullname="Fullname: " + users.c.name)

**>>>** conn.execute(stmt)

UPDATE users SET fullname=(? || users.name)

('Fullname: ',)

COMMIT

<sqlalchemy.engine.result.ResultProxy object at 0x...>

When using [insert()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableClause.insert" \o "sqlalchemy.sql.expression.TableClause.insert) or [update()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableClause.update" \o "sqlalchemy.sql.expression.TableClause.update) in an "execute many" context, we may also want to specify named bound parameters which we can refer to in the argument list. The two constructs will automatically generate bound placeholders for any column names passed in the dictionaries sent to [execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.execute" \o "sqlalchemy.engine.Connection.execute) at execution time. However, if we wish to use explicitly targeted named parameters with composed expressions, we need to use the [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) construct. When using [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) with[insert()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableClause.insert" \o "sqlalchemy.sql.expression.TableClause.insert) or [update()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableClause.update" \o "sqlalchemy.sql.expression.TableClause.update), the names of the table's columns themselves are reserved for the "automatic" generation of bind names. We can combine the usage of implicitly available bind names and explicitly named parameters as in the example below:

当在"执行许多"上下文中使用insert() 或update() 时，我们可能还需要指定参数列表中可以引用的命名绑定参数。 这两个结构将在执行时自动生成在发送到execute() 的字典中传递的任何列名的绑定占位符。 但是，如果我们希望使用明确的目标命名参数与组合表达式，我们需要使用bindparam() 构造。 当使用bindparam() insert() 或update() 时，表的列本身的名称将保留用于"自动"生成绑定名称。 我们可以结合使用隐式可用的绑定名称和明确命名的参数，如下例所示：

**>>>** stmt = users.insert().\

**...**  values(name=bindparam('\_name') + " .. name")

**>>>** conn.execute(stmt, [

**...**  {'id':4, '\_name':'name1'},

**...**  {'id':5, '\_name':'name2'},

**...**  {'id':6, '\_name':'name3'},

**...**  ])

INSERT INTO users (id, name) VALUES (?, (? || ?))

((4, 'name1', ' .. name'), (5, 'name2', ' .. name'), (6, 'name3', ' .. name'))

COMMIT

<sqlalchemy.engine.result.ResultProxy object at 0x...>

An UPDATE statement is emitted using the [update()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableClause.update" \o "sqlalchemy.sql.expression.TableClause.update) construct. This works much like an INSERT, except there is an additional WHERE clause that can be specified:

使用update() 构造发出UPDATE语句。 这非常像INSERT，除了可以指定的另外一个WHERE子句：

**>>>** stmt = users.update().\

**...**  where(users.c.name == 'jack').\

**...**  values(name='ed')

**>>>** conn.execute(stmt)

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

('ed', 'jack')

COMMIT

<sqlalchemy.engine.result.ResultProxy object at 0x...>

When using [update()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableClause.update" \o "sqlalchemy.sql.expression.TableClause.update) in an "executemany" context, we may wish to also use explicitly named bound parameters in the WHERE clause. Again, [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) is the construct used to achieve this:

当在"executemany"上下文中使用update() 时，我们也可能希望在WHERE子句中使用明确命名的bound参数。 再次，bindparam() 是用于实现这个的构造：

**>>>** stmt = users.update().\

**...**  where(users.c.name == bindparam('oldname')).\

**...**  values(name=bindparam('newname'))

**>>>** conn.execute(stmt, [

**...**  {'oldname':'jack', 'newname':'ed'},

**...**  {'oldname':'wendy', 'newname':'mary'},

**...**  {'oldname':'jim', 'newname':'jake'},

**...**  ])

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

(('ed', 'jack'), ('mary', 'wendy'), ('jake', 'jim'))

COMMIT

<sqlalchemy.engine.result.ResultProxy object at 0x...>

### 1.14.1 Correlated Updates

A correlated update lets you update a table using selection from another table, or the same table:

**>>>** stmt = select([addresses.c.email\_address]).\**...**  where(addresses.c.user\_id == users.c.id).\**...**  limit(1)

**>>>** conn.execute(users.update().values(fullname=stmt))

UPDATE users SET fullname=(SELECT addresses.email\_address

FROM addresses

WHERE addresses.user\_id = users.id

LIMIT ? OFFSET ?)

(1, 0)

COMMIT

<sqlalchemy.engine.result.ResultProxy object at 0x...>

### 1.14.2 Multiple Table Updates

*New in version 0.7.4.*

The PostgreSQL, Microsoft SQL Server, and MySQL backends all support UPDATE statements that refer to multiple tables. For PG and MSSQL, this is the "UPDATE FROM" syntax, which updates one table at a time, but can reference additional tables in an additional "FROM" clause that can then be referenced in the WHERE clause directly. On MySQL, multiple tables can be embedded into a single UPDATE statement separated by a comma. The SQLAlchemy [update()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.update" \o "sqlalchemy.sql.expression.update) construct supports both of these modes implicitly, by specifying multiple tables in the WHERE clause:

PostgreSQL，Microsoft SQL Server和MySQL后端都支持引用多个表的UPDATE语句。 对于PG和MSSQL，这是"UPDATE FROM"语法，它一次更新一个表，但可以引用额外的"FROM"子句中的其他表，然后可以直接在WHERE子句中引用它们。 在MySQL上，可以将多个表嵌入到用逗号分隔的单个UPDATE语句中。 SQLAlchemy update() 构造通过在WHERE子句中指定多个表来隐含地支持这两种模式：

stmt = users.update().\

values(name='ed wood').\

where(users.c.id == addresses.c.id).\

where(addresses.c.email\_address.startswith('ed%'))

conn.execute(stmt)

The resulting SQL from the above statement would render as:

UPDATE users SET name=:name FROM addresses

WHERE users.id = addresses.id AND

addresses.email\_address LIKE :email\_address\_1 || '*%%*'

When using MySQL, columns from each table can be assigned to in the SET clause directly, using the dictionary form passed to [Update.values()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Update.values" \o "sqlalchemy.sql.expression.Update.values):

stmt = users.update().\

values({

users.c.name:'ed wood',

addresses.c.email\_address:'ed.wood@foo.com'

}).\

where(users.c.id == addresses.c.id).\

where(addresses.c.email\_address.startswith('ed%'))

The tables are referenced explicitly in the SET clause:

UPDATE users, addresses SET addresses.email\_address=%s,

users.name=%s WHERE users.id = addresses.id

AND addresses.email\_address LIKE concat(%s, '*%%*')

SQLAlchemy doesn't do anything special when these constructs are used on a non-supporting database. The UPDATE FROM syntax generates by default when multiple tables are present, and the statement will be rejected by the database if this syntax is not supported.

### 1.14.3 Parameter-Ordered Updates

The default behavior of the [update()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.update" \o "sqlalchemy.sql.expression.update) construct when rendering the SET clauses is to render them using the column ordering given in the originating [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object. This is an important behavior, since it means that the rendering of a particular UPDATE statement with particular columns will be rendered the same each time, which has an impact on query caching systems that rely on the form of the statement, either client side or server side. Since the parameters themselves are passed to the [Update.values()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Update.values" \o "sqlalchemy.sql.expression.Update.values) method as Python dictionary keys, there is no other fixed ordering available.

在渲染SET子句时，update() 构造的默认行为是使用始发表对象中给定的列排序来呈现它们。 这是一个重要的行为，因为这意味着使用特定列的特定UPDATE语句的呈现将每次呈现相同，这对依赖语句形式的查询缓存系统（客户端或服务器）都有影响 侧。 由于参数本身作为Python字典键传递给Update.values() 方法，因此没有其他固定的排序可用。

However in some cases, the order of parameters rendered in the SET clause of an UPDATE statement can be significant. The main example of this is when using MySQL and providing updates to column values based on that of other column values. The end result of the following statement:

但是在某些情况下，在UPDATE语句的SET子句中呈现的参数的顺序可能很大。 其主要的例子是使用MySQL并根据其他列值的列值提供更新。 最后结果如下：

UPDATE some\_table SET x = y + 10, y = 20

Will have a different result than:

UPDATE some\_table SET y = 20, x = y + 10

This because on MySQL, the individual SET clauses are fully evaluated on a per-value basis, as opposed to on a per-row basis, and as each SET clause is evaluated, the values embedded in the row are changing.

To suit this specific use case, 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 may be used. When using this flag, we supply a ****Python list of 2-tuples**** as the argument to the [Update.values()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Update.values" \o "sqlalchemy.sql.expression.Update.values) method:

stmt = some\_table.update(preserve\_parameter\_order=**True**).\

values([(some\_table.c.y, 20), (some\_table.c.x, some\_table.c.y + 10)])

The list of 2-tuples is essentially the same structure as a Python dictionary except it is ordered. Using the above form, we are assured that the "y" column's SET clause will render first, then the "x" column's SET clause.

*New in version 1.0.10:*Added support for explicit ordering of UPDATE parameters using 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.

### 1.14.4 Deletes

Finally, a delete. This is accomplished easily enough using the [delete()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableClause.delete" \o "sqlalchemy.sql.expression.TableClause.delete) construct:

最后是删除。 使用[delete()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableClause.delete" \o "sqlalchemy.sql.expression.TableClause.delete)构造很容易实现：

**>>>** conn.execute(addresses.delete())

DELETE FROM addresses

()

COMMIT

<sqlalchemy.engine.result.ResultProxy object at 0x...>

**>>>** conn.execute(users.delete().where(users.c.name > 'm'))

DELETE FROM users WHERE users.name > ?

('m',)

COMMIT

<sqlalchemy.engine.result.ResultProxy object at 0x...>

### 1.14.5 Matched Row Counts

Both of [update()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableClause.update" \o "sqlalchemy.sql.expression.TableClause.update) and [delete()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableClause.delete" \o "sqlalchemy.sql.expression.TableClause.delete) are associated with *matched row counts*. This is a number indicating the number of rows that were matched by the WHERE clause. Note that by "matched", this includes rows where no UPDATE actually took place. The value is available as [rowcount](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.rowcount" \o "sqlalchemy.engine.ResultProxy.rowcount):

[update()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableClause.update" \o "sqlalchemy.sql.expression.TableClause.update)和[delete()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableClause.delete" \o "sqlalchemy.sql.expression.TableClause.delete)都与匹配的行数相关联。 这是一个数字，表示WHERE子句匹配的行数。 请注意，通过“匹配”，这包括没有更新实际发生的行。 该值可作为[rowcount](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.rowcount" \o "sqlalchemy.engine.ResultProxy.rowcount)使用：

**>>>** result = conn.execute(users.delete())

DELETE FROM users

()

COMMIT

**>>>** result.rowcount1

## 1.15 Further Reference

Expression Language Reference: [SQL Statements and Expressions API](http://docs.sqlalchemy.org/en/rel_1_1/core/expression_api.html)

Database Metadata Reference: [Describing Databases with MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html)

Engine Reference: [Engine Configuration](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html)

Connection Reference: [Working with Engines and Connections](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html)

Types Reference: [Column and Data Types](http://docs.sqlalchemy.org/en/rel_1_1/core/types.html)

窗体底端

# Chapter 2 SQL Statements and Expressions API

This section presents the API reference for the SQL Expression Language. For a full introduction to its usage, see [SQL Expression Language Tutorial](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html).

本节介绍SQL表达式语言的API参考。 有关其用法的全面介绍，请参阅SQL表达语言教程。

# **2.1 Column Elements and Expressions**

The expression API consists of a series of classes that each represent a specific lexical element within a SQL string. Composed together into a larger structure, they form a statement construct that may be *compiled* into a string representation that can be passed to a database. The classes are organized into a hierarchy that begins at the basemost ClauseElement class. Key subclasses include ColumnElement, which represents the role of any column-based expression in a SQL statement, such as in the columns clause, WHERE clause, and ORDER BY clause, and FromClause, which represents the role of a token that is placed in the FROM clause of a SELECT statement.

表达式API由一系列类组成，每个类表示SQL字符串中的特定词法元素。 它们组合成一个较大的结构，它们形成一个可以编译成可以传递给数据库的字符串表示形式的语句结构。 这些类被组织成一个层次结构，从一个基本的ClauseElement类开始。 关键子类包括ColumnElement，它表示SQL语句中任何基于列的表达式的作用，例如在columns子句WHERE子句和ORDER BY子句中，以及FromClause，它表示放在 SELECT语句的FROM子句。

sqlalchemy.sql.expression.**all\_**(*expr*)

Produce an ALL expression.

This may apply to an array type for some dialects (e.g. postgresql), or to a subquery for others (e.g. mysql). e.g.:

这可能适用于某些方言（例如postgresql）的阵列类型或其他（例如mysql）的子查询。 例如。：

*# postgresql '5 = ALL (somearray)'*

expr = 5 == all\_(mytable.c.somearray)

*# mysql '5 = ALL (SELECT value FROM table)'*

expr = 5 == all\_(select([table.c.value]))

*New in version 1.1.*

**See also**

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

sqlalchemy.sql.expression.**and\_**(*\*clauses*)

Produce a conjunction of expressions joined by AND.

E.g.:

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

stmt = select([users\_table]).where(

and\_(

users\_table.c.name == 'wendy',

users\_table.c.enrolled == **True**

)

)

The [and\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.and_" \o "sqlalchemy.sql.expression.and_) conjunction is also available using the Python & operator (though note that compound expressions need to be parenthesized in order to function with Python operator precedence behavior):

and\_() 连接也可以使用Python和运算符（虽然注意复合表达式需要括号，以便与Python运算符优先行为）

stmt = select([users\_table]).where(

(users\_table.c.name == 'wendy') &

(users\_table.c.enrolled == **True**)

)

The [and\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.and_" \o "sqlalchemy.sql.expression.and_) operation is also implicit in some cases; the [Select.where()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.where" \o "sqlalchemy.sql.expression.Select.where) method for example can be invoked multiple times against a statement, which will have the effect of each clause being combined using [and\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.and_" \o "sqlalchemy.sql.expression.and_):

在某些情况下，[and\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.and_" \o "sqlalchemy.sql.expression.and_)操作也是隐含的; 可以针对一个语句多次调用[Select.where()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.where" \o "sqlalchemy.sql.expression.Select.where)方法，这将使用[and\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.and_" \o "sqlalchemy.sql.expression.and_)来组合每个子句的效果：

stmt = select([users\_table]). where(users\_table.c.name == 'wendy'). where(users\_table.c.enrolled == **True**)

**See also**

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

sqlalchemy.sql.expression.**any\_**(*expr*)

Produce an ANY expression.

产生ANY 表达式。

This may apply to an array type for some dialects (e.g. postgresql), or to a subquery for others (e.g. mysql). e.g.:

这可能适用于某些方言（例如postgresql）的阵列类型或其他（例如mysql）的子查询。 例如。：

*# postgresql '5 = ANY (somearray)'*expr = 5 == any\_(mytable.c.somearray)

*# mysql '5 = ANY (SELECT value FROM table)'*expr = 5 == any\_(select([table.c.value]))

*New in version 1.1.*

**See also**

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

sqlalchemy.sql.expression.**asc**(*column*)

Produce an ascending ORDER BY clause element.

e.g.:

**from** **sqlalchemy** **import** ascstmt = select([users\_table]).order\_by(asc(users\_table.c.name))

will produce SQL as:

SELECT id, name FROM user ORDER BY name ASC

The [asc()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.asc" \o "sqlalchemy.sql.expression.asc) function is a standalone version of the [ColumnElement.asc()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement.asc" \o "sqlalchemy.sql.expression.ColumnElement.asc) method available on all SQL expressions, e.g.:

stmt = select([users\_table]).order\_by(users\_table.c.name.asc())

|  |  |
| --- | --- |
| **Parameters:** | ****column**** – A [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) (e.g. scalar SQL expression) with which to apply the [asc()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.asc" \o "sqlalchemy.sql.expression.asc) operation. |

**See also**

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

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

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

[Select.order\_by()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.order_by" \o "sqlalchemy.sql.expression.Select.order_by)

sqlalchemy.sql.expression.**between**(*expr*, *lower\_bound*, *upper\_bound*, *symmetric=False*)

Produce a BETWEEN predicate clause.

E.g.:

**from** **sqlalchemy** **import** between

stmt = select([users\_table]).where(between(users\_table.c.id, 5, 7))

Would produce SQL resembling:

SELECT id, name FROM user WHERE id BETWEEN :id\_1 AND :id\_2

The [between()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.between" \o "sqlalchemy.sql.expression.between) function is a standalone version of the [ColumnElement.between()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement.between" \o "sqlalchemy.sql.expression.ColumnElement.between) method available on all SQL expressions, as in:

stmt = select([users\_table]).where(users\_table.c.id.between(5, 7))

All arguments passed to [between()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.between" \o "sqlalchemy.sql.expression.between), including the left side column expression, are coerced from Python scalar values if a the value is not a [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement)subclass. For example, three fixed values can be compared as in:

传递给between() 的所有参数，包括左侧列表达式，如果值不是ColumnElementsubclass，则会从Python标量值强制执行。 例如，可以比较三个固定值，如：

print(between(5, 3, 7))

Which would produce:

:param\_1 BETWEEN :param\_2 AND :param\_3

|  |  |
| --- | --- |
| **Parameters:** | * ****expr**** – a column expression, typically a [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) instance or alternatively a Python scalar expression to be coerced into a column expression, serving as the left side of the BETWEEN expression. * ****lower\_bound**** – a column or Python scalar expression serving as the lower bound of the right side of the BETWEEN expression. * ****upper\_bound**** – a column or Python scalar expression serving as the upper bound of the right side of the BETWEEN expression. * ****symmetric –****if True, will render " BETWEEN SYMMETRIC ". Note that not all databases support this syntax.   *New in version 0.9.5.* |

**See also**

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

sqlalchemy.sql.expression.**bindparam**(*key*, *value=symbol('NO\_ARG')*, *type\_=None*, *unique=False*, *required=symbol('NO\_ARG')*, *quote=None*, *callable\_=None*, *isoutparam=False*, *\_compared\_to\_operator=None*, *\_compared\_to\_type=None*)

Produce a "bound expression".

产生"bound expression"。

The return value is an instance of [BindParameter](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.BindParameter" \o "sqlalchemy.sql.expression.BindParameter); this is a [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) subclass which represents a so-called "placeholder" value in a SQL expression, the value of which is supplied at the point at which the statement in executed against a database connection.

返回值是BindParameter的一个实例; 这是一个ColumnElement子类，它表示SQL表达式中所谓的"占位符"值，其值在根据数据库连接执行的语句点提供。

In SQLAlchemy, the [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) construct has the ability to carry along the actual value that will be ultimately used at expression time. In this way, it serves not just as a "placeholder" for eventual population, but also as a means of representing so-called "unsafe" values which should not be rendered directly in a SQL statement, but rather should be passed along to the [DBAPI](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-dbapi) as values which need to be correctly escaped and potentially handled for type-safety.

在SQLAlchemy中，bindparam() 结构具有携带在表达式最终将被使用的实际值的能力。 这样，它不仅可以作为最终人口的"占位符"，还可以作为一种表示所谓"不安全"值的方法，而不应该直接在SQL语句中呈现，而应该传递给 DBAPI作为需要正确转义并可能处理类型安全的值。

When using [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) explicitly, the use case is typically one of traditional deferment of parameters; the [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) construct accepts a name which can then be referred to at execution time:

当显式使用bindparam() 时，用例通常是传统延迟参数之一; bindparam() 构造接受一个可以在执行时被引用的名称：

**from** **sqlalchemy** **import** bindparam

stmt = select([users\_table]).\

where(users\_table.c.name == bindparam('username'))

The above statement, when rendered, will produce SQL similar to:

上述语句在渲染时将产生类似于以下内容的SQL：

SELECT id, name FROM user WHERE name = :username

In order to populate the value of :username above, the value would typically be applied at execution time to a method like [Connection.execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.execute" \o "sqlalchemy.engine.Connection.execute):

为了填充:username的值，通常会在执行时将该值应用于[Connection.execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.execute" \o "sqlalchemy.engine.Connection.execute) 之类的方法：

result = connection.execute(stmt, username='wendy')

Explicit use of [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) is also common when producing UPDATE or DELETE statements that are to be invoked multiple times, where the WHERE criterion of the statement is to change on each invocation, such as:

当生成要多次调用的UPDATE或DELETE语句时，显式使用bindparam() 也是常见的，其中语句的WHERE标准将在每次调用时更改，例如：

stmt = (users\_table.update().

where(user\_table.c.name == bindparam('username')).

values(fullname=bindparam('fullname'))

)

connection.execute(

stmt, [{"username": "wendy", "fullname": "Wendy Smith"},

{"username": "jack", "fullname": "Jack Jones"},

])

SQLAlchemy's Core expression system makes wide use of [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) in an implicit sense. It is typical that Python literal values passed to virtually all SQL expression functions are coerced into fixed [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) constructs. For example, given a comparison operation such as:

SQLAlchemy的核心表达系统在隐含的意义上广泛使用了bindparam() 。 典型地，传递给几乎所有SQL表达式函数的Python文字值被强制转换为固定的bindparam() 结构。 例如，给出比较操作，如：

expr = users\_table.c.name == 'Wendy'

The above expression will produce a [BinaryExpression](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.BinaryExpression" \o "sqlalchemy.sql.expression.BinaryExpression) construct, where the left side is the [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) object representing the name column, and the right side is a [BindParameter](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.BindParameter" \o "sqlalchemy.sql.expression.BindParameter) representing the literal value:

上述表达式将生成一个[BinaryExpression](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.BinaryExpression" \o "sqlalchemy.sql.expression.BinaryExpression)构造，其中左侧是表示name列的[Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column)对象，右侧是表示文字值的[BindParameter](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.BindParameter" \o "sqlalchemy.sql.expression.BindParameter)：

print(repr(expr.right))

BindParameter('%(4327771088 name)s', 'Wendy', type\_=String())

The expression above will render SQL such as:

user.name = :name\_1

Where the :name\_1 parameter name is an anonymous name. The actual string Wendy is not in the rendered string, but is carried along where it is later used within statement execution. If we invoke a statement like the following:

其中：name\_1参数名称是匿名名称。 实际的字符串Wendy不在渲染的字符串中，而是随后在语句执行中被使用。 如果我们调用如下的语句：

stmt = select([users\_table]).where(users\_table.c.name == 'Wendy')

result = connection.execute(stmt)

We would see SQL logging output as:

SELECT "user".id, "user".name FROM "user"

WHERE "user".name = %(name\_1)

s{'name\_1': 'Wendy'}

Above, we see that Wendy is passed as a parameter to the database, while the placeholder :name\_1 is rendered in the appropriate form for the target database, in this case the PostgreSQL database.

以上，我们看到Wendy作为一个参数传递给数据库，而占位符：name\_1以适当的形式呈现给目标数据库，在这种情况下是PostgreSQL数据库。

Similarly, [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) is invoked automatically when working with [CRUD](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-crud) statements as far as the "VALUES" portion is concerned. The [insert()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.insert" \o "sqlalchemy.sql.expression.insert) construct produces an INSERT expression which will, at statement execution time, generate bound placeholders based on the arguments passed, as in:

类似地，在使用CRUD语句时，就会自动调用[bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) ，就"VALUES"部分而言。 [insert()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.insert" \o "sqlalchemy.sql.expression.insert) 结构生成一个INSERT表达式，它将在语句执行时根据传递的参数生成绑定的占位符，如：

stmt = users\_table.insert()result = connection.execute(stmt, name='Wendy')

The above will produce SQL output as:

INSERT INTO "user" (name) VALUES (%(name)s){'name': 'Wendy'}

The [Insert](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert" \o "sqlalchemy.sql.expression.Insert) construct, at compilation/execution time, rendered a single [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) mirroring the column name name as a result of the single nameparameter we passed to the [Connection.execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.execute" \o "sqlalchemy.engine.Connection.execute) method.

作为编译/执行时间的Insert构造，渲染了一个单一的bindparam() ，作为我们传递给Connection.execute() 方法的单个nameparameter结果的镜像列名称。

|  |  |
| --- | --- |
| **Parameters:** | * ****key**** – the key (e.g. the name) for this bind param. Will be used in the generated SQL statement for dialects that use named parameters. This value may be modified when part of a compilation operation, if other [BindParameter](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.BindParameter" \o "sqlalchemy.sql.expression.BindParameter) objects exist with the same key, or if its length is too long and truncation is required.这个绑定参数的关键（例如名称）。 将在生成的SQL语句中用于使用命名参数的方言。 如果其他BindParameter对象存在相同的键，或者如果其长度太长并且需要截断，则可能会在编译操作的一部分时修改该值。 * ****value**** – Initial value for this bind param. Will be used at statement execution time as the value for this parameter passed to the DBAPI, if no other value is indicated to the statement execution method for this particular parameter name. Defaults to None.此绑定参数的初始值。 将在语句执行时使用该参数的值传递给DBAPI，如果该特定参数名称的语句执行方法中没有其他值。 默认为无。 * ****callable\_**** – A callable function that takes the place of "value". The function will be called at statement execution time to determine the ultimate value. Used for scenarios where the actual bind value cannot be determined at the point at which the clause construct is created, but embedded bind values are still desirable.代替"值"的可调用函数。 该函数将在语句执行时调用以确定最终值。 用于在创建子句构造时无法确定实际绑定值的情况，但仍然需要嵌入的绑定值。 * ****type\_ –****A [TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine) class or instance representing an optional datatype for this [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam). If not passed, a type may be determined automatically for the bind, based on the given value; for example, trivial Python types such as str, int, bool may result in the [String](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.String" \o "sqlalchemy.types.String), [Integer](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Integer" \o "sqlalchemy.types.Integer) or [Boolean](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Boolean" \o "sqlalchemy.types.Boolean) types being automatically selected.表示此bindparam() 的可选数据类型的TypeEngine类或实例。 如果没有通过，可以根据给定的值自动确定绑定的类型; 例如，琐碎的Python类型，如str，int，bool可能会导致String，Integer或Boolean类型被自动选择。   The type of a [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) is significant especially in that the type will apply pre-processing to the value before it is passed to the database. For example, a [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) which refers to a datetime value, and is specified as holding the [DateTime](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.DateTime" \o "sqlalchemy.types.DateTime) type, may apply conversion needed to the value (such as stringification on SQLite) before passing the value to the database.  bindparam() 的类型很重要，特别是在该类型将传递给数据库之前，该类型将对值应用预处理。 例如，引用datetime值并被指定为保留DateTime类型的bindparam() 可以在将值传递给数据库之前将该值（例如SQLite上的字符串化）所需的转换应用于应用程序。   * ****unique**** – if True, the key name of this [BindParameter](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.BindParameter" \o "sqlalchemy.sql.expression.BindParameter) will be modified if another [BindParameter](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.BindParameter" \o "sqlalchemy.sql.expression.BindParameter) of the same name already has been located within the containing expression. This flag is used generally by the internals when producing so-called "anonymous" bound expressions, it isn't generally applicable to explicitly-named [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) constructs.如果为True，则如果在包含表达式中已经找到另一个同名的BindParameter，则将修改此BindParameter的键名称。 当生成所谓的“匿名”绑定表达式时，这个标志一般由内部使用，它通常不适用于明确命名的bindparam（）构造。 * ****required –****If True, a value is required at execution time. If not passed, it defaults to True if neither [bindparam.value](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam.params.value" \o "sqlalchemy.sql.expression.bindparam) or [bindparam.callable](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam.params.callable" \o "sqlalchemy.sql.expression.bindparam) were passed. If either of these parameters are present, then [bindparam.required](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam.params.required" \o "sqlalchemy.sql.expression.bindparam) defaults to False.如果为True，则在执行时需要一个值。 如果没有通过，则默认为True，如果bindparam.value或bindparam.callable都没有通过。 如果这些参数中的任何一个都存在，那么bindparam.required默认为False。   *Changed in version 0.8:*If the required flag is not specified, it will be set automatically to True or False depending on whether or not the value or callableparameters were specified.   * ****quote**** – True if this parameter name requires quoting and is not currently known as a SQLAlchemy reserved word; this currently only applies to the Oracle backend, where bound names must sometimes be quoted.如果此参数名称需要引用，并且当前不被称为SQLAlchemy保留字，则为true; 这目前只适用于Oracle后端，有时必须引用绑定名称。 * ****isoutparam**** – if True, the parameter should be treated like a stored procedure "OUT" parameter. This applies to backends such as Oracle which support OUT parameters.如果为True，则该参数应该像存储过程“OUT”参数一样处理。 这适用于支持OUT参数的后端，如Oracle。 |

**See also**

[Bind Parameter Objects](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html" \l "coretutorial-bind-param)

[Insert Expressions](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html" \l "coretutorial-insert-expressions)

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

sqlalchemy.sql.expression.**case**(*whens*, *value=None*, *else\_=None*)

Produce a CASE expression.

产生一个CASE表达式。

The CASE construct in SQL is a conditional object that acts somewhat analogously to an "if/then" construct in other languages. It returns an instance of [Case](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.Case" \o "sqlalchemy.sql.expression.Case).

SQL中的CASE构造是一种条件对象，其类似于其他语言中的"if / then"结构。 它返回Case的一个实例。

[case()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.case" \o "sqlalchemy.sql.expression.case) in its usual form is passed a list of "when" constructs, that is, a list of conditions and results as tuples:

其通常形式的case() 通过一个"when"结构的列表，即一个条件和结果列表作为元组：

**from** **sqlalchemy** **import** case

stmt = select([users\_table]).\

where(

case(

[

(users\_table.c.name == 'wendy', 'W'),

(users\_table.c.name == 'jack', 'J')

],

else\_='E'

)

)

The above statement will produce SQL resembling:

上面的语句会产生类似于以下的SQL：

SELECT id, name FROM user

WHERE CASE

WHEN (name = :name\_1) THEN :param\_1

WHEN (name = :name\_2) THEN :param\_2

ELSE :param\_3

END

When simple equality expressions of several values against a single parent column are needed, [case()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.case" \o "sqlalchemy.sql.expression.case) also has a "shorthand" format used via the[case.value](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.case.params.value" \o "sqlalchemy.sql.expression.case) parameter, which is passed a column expression to be compared. In this form, the [case.whens](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.case.params.whens" \o "sqlalchemy.sql.expression.case) parameter is passed as a dictionary containing expressions to be compared against keyed to result expressions. The statement below is equivalent to the preceding statement:

当需要针对单个父列的几个值的简单等式表达式时，case() 也具有通过thecase.value参数使用的“速记”格式，该格式传递要比较的列表达式。 在这种形式中，case.whens参数作为字典传递，包含要与键控到结果表达式进行比较的表达式。 以下声明等同于上述声明：

stmt = select([users\_table]).\

where(

case(

{"wendy": "W", "jack": "J"},

value=users\_table.c.name,

else\_='E'

)

)

The values which are accepted as result values in [case.whens](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.case.params.whens" \o "sqlalchemy.sql.expression.case) as well as with [case.else\_](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.case.params.else_" \o "sqlalchemy.sql.expression.case) are coerced from Python literals into [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) constructs. SQL expressions, e.g. [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) constructs, are accepted as well. To coerce a literal string expression into a constant expression rendered inline, use the [literal\_column()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.literal_column" \o "sqlalchemy.sql.expression.literal_column) construct, as in:

在case.when以及case.else\_中接受为结果值的值被强制从Python文字转换成bindparam() 结构。 SQL表达式，例如 ColumnElement构造也被接受。 要将文字字符串表达式强制转换为内联呈现的常量表达式，请使用literal\_column() 构造，如：

**from** **sqlalchemy** **import** case, literal\_column

case(

[

(

orderline.c.qty > 100,

literal\_column("'greaterthan100'")

),

(

orderline.c.qty > 10,

literal\_column("'greaterthan10'")

)

],

else\_=literal\_column("'lessthan10'"))

The above will render the given constants without using bound parameters for the result values (but still for the comparison values), as in:

CASE

WHEN (orderline.qty > :qty\_1) THEN 'greaterthan100'

WHEN (orderline.qty > :qty\_2) THEN 'greaterthan10'

ELSE 'lessthan10'END

|  |  |
| --- | --- |
| **Parameters:** | * ****whens –****The criteria to be compared against, [case.whens](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.case.params.whens" \o "sqlalchemy.sql.expression.case) accepts two different forms, based on whether or not [case.value](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.case.params.value" \o "sqlalchemy.sql.expression.case) is used.要比较的标准，case.when接受两种不同的形式，根据是否使用case.value。   In the first form, it accepts a list of 2-tuples; each 2-tuple consists of (<sql expression>, <value>), where the SQL expression is a boolean expression and "value" is a resulting value, e.g.:  case([  (users\_table.c.name == 'wendy', 'W'),  (users\_table.c.name == 'jack', 'J')])  In the second form, it accepts a Python dictionary of comparison values mapped to a resulting value; this form requires [case.value](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.case.params.value" \o "sqlalchemy.sql.expression.case) to be present, and values will be compared using the == operator, e.g.:  case(  {"wendy": "W", "jack": "J"},  value=users\_table.c.name)   * ****value**** – An optional SQL expression which will be used as a fixed "comparison point" for candidate values within a dictionary passed to [case.whens](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.case.params.whens" \o "sqlalchemy.sql.expression.case). * ****else\_**** – An optional SQL expression which will be the evaluated result of the CASE construct if all expressions within [case.whens](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.case.params.whens" \o "sqlalchemy.sql.expression.case) evaluate to false. When omitted, most databases will produce a result of NULL if none of the "when" expressions evaluate to true. |

sqlalchemy.sql.expression.**cast**(*expression*, *type\_*)

Produce a CAST expression.

[cast()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.cast" \o "sqlalchemy.sql.expression.cast) returns an instance of [Cast](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.Cast" \o "sqlalchemy.sql.expression.Cast).

E.g.:

**from** **sqlalchemy** **import** cast, Numeric

stmt = select([

cast(product\_table.c.unit\_price, Numeric(10, 4))

])

The above statement will produce SQL resembling:

SELECT CAST(unit\_price AS NUMERIC(10, 4)) FROM product

The [cast()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.cast" \o "sqlalchemy.sql.expression.cast) function performs two distinct functions when used. The first is that it renders the CAST expression within the resulting SQL string. The second is that it associates the given type (e.g. [TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine) class or instance) with the column expression on the Python side, which means the expression will take on the expression operator behavior associated with that type, as well as the bound-value handling and result-row-handling behavior of the type.

使用时，cast() 函数执行两个不同的功能。 第一个是它在生成的SQL字符串中呈现CAST表达式。 第二个是它将给定类型（例如TypeEngine类或实例）与Python端的列表达式相关联，这意味着表达式将接受与该类型相关联的表达式运算符行为以及绑定值处理和 该类型的结果行处理行为。

*Changed in version 0.9.0:*[cast()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.cast" \o "sqlalchemy.sql.expression.cast) now applies the given type to the expression such that it takes effect on the bound-value, e.g. the Python-to-database direction, in addition to the result handling, e.g. database-to-Python, direction.

在版本0.9.0中更改：cast() 现在将给定的类型应用于表达式，使其在绑定值上生效，例如。 除了结果处理之外，Python到数据库方向也是如此。 数据库到Python，方向。

An alternative to [cast()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.cast" \o "sqlalchemy.sql.expression.cast) is the [type\_coerce()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.type_coerce" \o "sqlalchemy.sql.expression.type_coerce) function. This function performs the second task of associating an expression with a specific type, but does not render the CAST expression in SQL.

cast() 的替代方法是type\_coerce() 函数。 此函数执行将表达式与特定类型相关联的第二个任务，但不会在SQL中呈现CAST表达式。

|  |  |
| --- | --- |
| **Parameters:** | * ****expression**** – A SQL expression, such as a [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) expression or a Python string which will be coerced into a bound literal value. * ****type\_**** – A [TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine) class or instance indicating the type to which the CAST should apply. |

**See also**

[type\_coerce()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.type_coerce" \o "sqlalchemy.sql.expression.type_coerce) - Python-side type coercion without emitting CAST.

sqlalchemy.sql.expression.**column**(*text*, *type\_=None*, *is\_literal=False*, *\_selectable=None*)

Produce a [ColumnClause](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnClause" \o "sqlalchemy.sql.expression.ColumnClause) object.

生成ColumnClause对象。

The [ColumnClause](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnClause" \o "sqlalchemy.sql.expression.ColumnClause) is a lightweight analogue to the [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) class. The [column()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.column" \o "sqlalchemy.sql.expression.column) function can be invoked with just a name alone, as in:

ColumnClause是一个轻量级的类与Column类。 只能使用一个名称来调用column() 函数，如：

**from** **sqlalchemy** **import** column

id, name = column("id"), column("name")stmt = select([id, name]).select\_from("user")

The above statement would produce SQL like:

上面的语句会产生SQL，如：

SELECT id, name FROM user

Once constructed, [column()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.column" \o "sqlalchemy.sql.expression.column) may be used like any other SQL expression element such as within [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) constructs:

一旦构造，column() 可以像任何其他SQL表达式元素一样使用，例如在select() 结构中：

**from** **sqlalchemy.sql** **import** column

id, name = column("id"), column("name")

stmt = select([id, name]).select\_from("user")

The text handled by [column()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.column" \o "sqlalchemy.sql.expression.column) is assumed to be handled like the name of a database column; if the string contains mixed case, special characters, or matches a known reserved word on the target backend, the column expression will render using the quoting behavior determined by the backend. To produce a textual SQL expression that is rendered exactly without any quoting, use [literal\_column()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.literal_column" \o "sqlalchemy.sql.expression.literal_column) instead, or pass True as the value of [column.is\_literal](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.column.params.is_literal" \o "sqlalchemy.sql.expression.Select.column). Additionally, full SQL statements are best handled using the [text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text) construct.

由[column()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.column" \o "sqlalchemy.sql.expression.column)处理的文本被假定为像数据库列的名称一样处理; 如果字符串包含混合大小写，特殊字符或匹配目标后端上的已知保留字，则列表达式将使用由后端确定的引用行为来呈现。 要产生完全没有引用的文本SQL表达式，请改用[literal\_column()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.literal_column" \o "sqlalchemy.sql.expression.literal_column)，或者将True作为[column.is\_literal](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.column.params.is_literal" \o "sqlalchemy.sql.expression.Select.column)的值。 另外，最好使用[text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text)构造处理完整的SQL语句。

[column()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.column" \o "sqlalchemy.sql.expression.column) can be used in a table-like fashion by combining it with the [table()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.table" \o "sqlalchemy.sql.expression.table) function (which is the lightweight analogue to [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)) to produce a working table construct with minimal boilerplate:

将column() 函数与table() 函数（它是与表的轻量级类似）结合使用，可以以table-like方式使用column() 来生成一个带有最小样板的工作表构造：

**from** **sqlalchemy** **import** table, column, select

user = table("user",

column("id"),

column("name"),

column("description"),)

stmt = select([user.c.description]).where(user.c.name == 'wendy')

A [column()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.column" \o "sqlalchemy.sql.expression.column) / [table()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.table" \o "sqlalchemy.sql.expression.table) construct like that illustrated above can be created in an ad-hoc fashion and is not associated with any [schema.MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData), DDL, or events, unlike its [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) counterpart.

可以以ad-hoc方式创建一个像上面所示的列() / table() 结构，并且与其对应的表不同，它不与任何schema.MetaData，DDL或事件相关联。

*Changed in version 1.0.0:*[expression.column()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.column" \o "sqlalchemy.sql.expression.column) can now be imported from the plain sqlalchemy namespace like any other SQL element.

|  |  |
| --- | --- |
| **Parameters:** | * ****text**** – the text of the element. * ****type**** – [types.TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine) object which can associate this [ColumnClause](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnClause" \o "sqlalchemy.sql.expression.ColumnClause) with a type. * ****is\_literal**** – if True, the [ColumnClause](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnClause" \o "sqlalchemy.sql.expression.ColumnClause) is assumed to be an exact expression that will be delivered to the output with no quoting rules applied regardless of case sensitive settings. the [literal\_column()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.literal_column" \o "sqlalchemy.sql.expression.literal_column) function essentially invokes [column()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.column" \o "sqlalchemy.sql.expression.column) while passing is\_literal=True. * 如果为True，则ColumnClause被假定为将被传递到输出的精确表达式，不使用引号规则，而不管区分大小写设置。 literal\_column() 函数在传递is\_literal = True时基本上调用column() 。 |

**See also**

[Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column)

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

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

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

[Using More Specific Text with table(), literal\_column(), and column()](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html" \l "sqlexpression-literal-column)

sqlalchemy.sql.expression.**collate**(*expression*, *collation*)

Return the clause expression COLLATE collation.

e.g.:

collate(mycolumn, 'utf8\_bin')

produces:

mycolumn COLLATE utf8\_bin

sqlalchemy.sql.expression.**desc**(*column*)

Produce a descending ORDER BY clause element.

e.g.:

**from** **sqlalchemy** **import** desc

stmt = select([users\_table]).order\_by(desc(users\_table.c.name))

will produce SQL as:

SELECT id, name FROM user ORDER BY name DESC

The [desc()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.desc" \o "sqlalchemy.sql.expression.desc) function is a standalone version of the [ColumnElement.desc()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement.desc" \o "sqlalchemy.sql.expression.ColumnElement.desc) method available on all SQL expressions, e.g.:

desc() 函数是所有SQL表达式上可用的ColumnElement.desc() 方法的独立版本，例如：

stmt = select([users\_table]).order\_by(users\_table.c.name.desc())

|  |  |
| --- | --- |
| **Parameters:** | ****column**** – A [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) (e.g. scalar SQL expression) with which to apply the [desc()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.desc" \o "sqlalchemy.sql.expression.desc) operation.  用于应用desc() 操作的ColumnElement（例如标量SQL表达式）。 |

**See also**

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

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

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

[Select.order\_by()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.order_by" \o "sqlalchemy.sql.expression.Select.order_by)

sqlalchemy.sql.expression.**distinct**(*expr*)

Produce an column-expression-level unary DISTINCT clause.

生成一个列表达式的一元DISTINCT子句。

This applies the DISTINCT keyword to an individual column expression, and is typically contained within an aggregate function, as in:

这将DISTINCT关键字应用于单个列表达式，通常包含在聚合函数中，如：

**from** **sqlalchemy** **import** distinct, func

stmt = select([func.count(distinct(users\_table.c.name))])

The above would produce an expression resembling:

以上将产生一个类似于：

SELECT COUNT(DISTINCT name) FROM user

The [distinct()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.distinct" \o "sqlalchemy.sql.expression.distinct) function is also available as a column-level method, e.g. [ColumnElement.distinct()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement.distinct" \o "sqlalchemy.sql.expression.ColumnElement.distinct), as in:

distinct() 函数也可用作列级方法，例如。 ColumnElement.distinct() ，如：

stmt = select([func.count(users\_table.c.name.distinct())])

The [distinct()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.distinct" \o "sqlalchemy.sql.expression.distinct) operator is different from the [Select.distinct()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.distinct" \o "sqlalchemy.sql.expression.Select.distinct) method of [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select), which produces a SELECT statement with DISTINCT applied to the result set as a whole, e.g. a SELECT DISTINCT expression. See that method for further information.

distinct() 运算符与Select的Select.distinct() 方法不同，该方法产生一个SELECT语句，其中DISTINCT作为整体应用于结果集，例如。一个SELECT DISTINCT表达。 请参阅该方法获取更多信息。

**See also**

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

[Select.distinct()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.distinct" \o "sqlalchemy.sql.expression.Select.distinct)

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

sqlalchemy.sql.expression.**extract**(*field*, *expr*, *\*\*kwargs*)

Return a [Extract](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.Extract" \o "sqlalchemy.sql.expression.Extract) construct.

This is typically available as [extract()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.extract" \o "sqlalchemy.sql.expression.extract) as well as func.extract from the [func](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.func" \o "sqlalchemy.sql.expression.func) namespace.

sqlalchemy.sql.expression.**false**()

Return a [False\_](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.elements.False_" \o "sqlalchemy.sql.elements.False_) construct.

E.g.:

**>>> from** **sqlalchemy** **import** false

**>>>** print select([t.c.x]).where(false())SELECT x FROM t WHERE false

A backend which does not support true/false constants will render as an expression against 1 or 0:

不支持true / false常量的后端将呈现为1或0的表达式：

**>>>** print select([t.c.x]).where(false())SELECT x FROM t WHERE 0 = 1

The [true()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.true" \o "sqlalchemy.sql.expression.true) and [false()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.false" \o "sqlalchemy.sql.expression.false) constants also feature "short circuit" operation within an [and\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.and_" \o "sqlalchemy.sql.expression.and_) or [or\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.or_" \o "sqlalchemy.sql.expression.or_) conjunction:

true() 和false() 常量也在and\_() 或or\_() 连接中具有“短路”操作：

**>>>** print select([t.c.x]).where(or\_(t.c.x > 5, true()))

SELECT x FROM t WHERE true

**>>>** print select([t.c.x]).where(and\_(t.c.x > 5, false()))

SELECT x FROM t WHERE false

*Changed in version 0.9:*[true()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.true" \o "sqlalchemy.sql.expression.true) and [false()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.false" \o "sqlalchemy.sql.expression.false) feature better integrated behavior within conjunctions and on dialects that don't support true/false constants.

**See also**

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

sqlalchemy.sql.expression.**func***= <sqlalchemy.sql.functions.\_FunctionGenerator object>*

Generate SQL function expressions.

[func](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.func" \o "sqlalchemy.sql.expression.func) is a special object instance which generates SQL functions based on name-based attributes, e.g.:

**>>>** print(func.count(1))count(:param\_1)

The element is a column-oriented SQL element like any other, and is used in that way:

**>>>** print(select([func.count(table.c.id)]))

SELECT count(sometable.id) FROM sometable

Any name can be given to [func](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.func" \o "sqlalchemy.sql.expression.func). If the function name is unknown to SQLAlchemy, it will be rendered exactly as is. For common SQL functions which SQLAlchemy is aware of, the name may be interpreted as a *generic function* which will be compiled appropriately to the target database:

任何名字都可以发给func。 如果函数名称对于SQLAlchemy是未知的，则它将被完全呈现。 对于SQLAlchemy所知道的常见SQL函数，该名称可能会被解释为一个通用函数，它将被适当地编译到目标数据库中：

**>>>** print(func.current\_timestamp())

CURRENT\_TIMESTAMP

To call functions which are present in dot-separated packages, specify them in the same manner:

要调用以点分隔的包中存在的函数，请以相同的方式指定它们：

**>>>** print(func.stats.yield\_curve(5, 10))

stats.yield\_curve(:yield\_curve\_1, :yield\_curve\_2)

SQLAlchemy can be made aware of the return type of functions to enable type-specific lexical and result-based behavior. For example, to ensure that a string-based function returns a Unicode value and is similarly treated as a string in expressions, specify [Unicode](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Unicode" \o "sqlalchemy.types.Unicode) as the type:

可以使SQLAlchemy知道返回类型的函数，以启用类型特定的词法和基于结果的行为。 例如，为了确保基于字符串的函数返回Unicode值，并且在表达式中也被视为一个字符串，请指定Unicode作为类型：

**>>>** print(func.my\_string(u'hi', type\_=Unicode) + ' ' +**...**  func.my\_string(u'there', type\_=Unicode))my\_string(:my\_string\_1) || :my\_string\_2 || my\_string(:my\_string\_3)

The object returned by a [func](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.func" \o "sqlalchemy.sql.expression.func) call is usually an instance of [Function](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.Function" \o "sqlalchemy.sql.functions.Function). This object meets the "column" interface, including comparison and labeling functions. The object can also be passed the [execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connectable.execute" \o "sqlalchemy.engine.Connectable.execute) method of a [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) or [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine), where it will be wrapped inside of a SELECT statement first:

func调用返回的对象通常是Function的一个实例。 此对象符合“列”界面，包括比较和标注功能。 该对象也可以传递一个连接或引擎的execute() 方法，它将被首先包含在SELECT语句之内：

print(connection.execute(func.current\_timestamp()).scalar())

In a few exception cases, the [func](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.func" \o "sqlalchemy.sql.expression.func) accessor will redirect a name to a built-in expression such as [cast()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.cast" \o "sqlalchemy.sql.expression.cast) or [extract()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.extract" \o "sqlalchemy.sql.expression.extract), as these names have well-known meaning but are not exactly the same as "functions" from a SQLAlchemy perspective.

在一些异常情况下，func访问器将将名称重定向到内置表达式，如cast() 或extract() ，因为这些名称具有众所周知的含义，但与SQLAlchemy的“函数”不完全相同 透视。

*New in version 0.8:*[func](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.func" \o "sqlalchemy.sql.expression.func) can return non-function expression constructs for common quasi-functional names like [cast()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.cast" \o "sqlalchemy.sql.expression.cast) and [extract()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.extract" \o "sqlalchemy.sql.expression.extract).

Functions which are interpreted as "generic" functions know how to calculate their return type automatically. For a listing of known generic functions, see [SQL and Generic Functions](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "generic-functions).

被解释为“通用”功能的函数知道如何自动计算其返回类型。 有关已知通用函数的列表，请参阅SQL和泛函数。

**Note**

The [func](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.func" \o "sqlalchemy.sql.expression.func) construct has only limited support for calling standalone "stored procedures", especially those with special parameterization concerns.

func构造对于调用独立的“存储过程”，特别是那些具有特殊参数化问题的调用来说只能得到有限的支持。

See the section [Calling Stored Procedures](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "stored-procedures) for details on how to use the DBAPI-level callproc() method for fully traditional stored procedures.

有关如何对完全传统的存储过程使用DBAPI级别的callproc() 方法的详细信息，请参阅调用存储过程一节。

sqlalchemy.sql.expression.**funcfilter**(*func*, *\*criterion*)

Produce a [FunctionFilter](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.FunctionFilter" \o "sqlalchemy.sql.expression.FunctionFilter) object against a function.

针对函数生成一个FunctionFilter对象。

Used against aggregate and window functions, for database backends that support the "FILTER" clause.

用于聚合和窗口函数，用于支持“FILTER”子句的数据库后端。

E.g.:

**from** **sqlalchemy** **import** funcfilter

funcfilter(func.count(1), MyClass.name == 'some name')

Would produce "COUNT(1) FILTER (WHERE myclass.name = 'some name')".

This function is also 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 itself via the [FunctionElement.filter()](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.FunctionElement.filter" \o "sqlalchemy.sql.functions.FunctionElement.filter) method.

会产生“COUNT（1）FILTER（WHERE myclass.name ='some name'）”。

该函数也可以通过FunctionElement.filter() 方法从func结构本身获得。

*New in version 1.0.0.*

**See also**

[FunctionElement.filter()](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.FunctionElement.filter" \o "sqlalchemy.sql.functions.FunctionElement.filter)

sqlalchemy.sql.expression.**label**(*name*, *element*, *type\_=None*)

Return a [Label](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.Label" \o "sqlalchemy.sql.expression.Label) object for the given [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement).

返回给定ColumnElement的Label对象。

A label changes the name of an element in the columns clause of a SELECT statement, typically via the AS SQL keyword.

标签通常通过AS SQL关键字更改SELECT语句的columns子句中的元素的名称。

This functionality is more conveniently available via the [ColumnElement.label()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement.label" \o "sqlalchemy.sql.expression.ColumnElement.label) method on [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement).

ColumnElement上的ColumnElement.label() 方法可以更方便地使用此功能。

|  |  |
| --- | --- |
| **Parameters:** | * ****name**** – label name * ****obj**** – a [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement). |

sqlalchemy.sql.expression.**literal**(*value*, *type\_=None*)

Return a literal clause, bound to a bind parameter.

返回一个文字子句，绑定到一个绑定参数。

Literal clauses are created automatically when non- [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement) objects (such as strings, ints, dates, etc.) are used in a comparison operation with a [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) subclass, such as a [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) object. Use this function to force the generation of a literal clause, which will be created as a [BindParameter](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.BindParameter" \o "sqlalchemy.sql.expression.BindParameter)with a bound value.

当与ColumnElement子类（如Column对象）的比较操作中使用非ClauseElement对象（如字符串，int，日期等）时，会自动创建文字子句。 使用此函数强制生成一个文字子句，该子句将被创建为具有绑定值的BindParameter。

|  |  |
| --- | --- |
| **Parameters:** | * ****value**** – the value to be bound. Can be any Python object supported by the underlying DB-API, or is translatable via the given type argument. * ****type\_**** – an optional [TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine) which will provide bind-parameter translation for this literal. |

sqlalchemy.sql.expression.**literal\_column**(*text*, *type\_=None*)

Produce a [ColumnClause](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnClause" \o "sqlalchemy.sql.expression.ColumnClause) object that has the [column.is\_literal](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.column.params.is_literal" \o "sqlalchemy.sql.expression.Select.column) flag set to True.

生成一个Column.is\_literal标志设置为True的ColumnClause对象。

[literal\_column()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.literal_column" \o "sqlalchemy.sql.expression.literal_column) is similar to [column()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.column" \o "sqlalchemy.sql.expression.column), except that it is more often used as a "standalone" column expression that renders exactly as stated; while [column()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.column" \o "sqlalchemy.sql.expression.column) stores a string name that will be assumed to be part of a table and may be quoted as such, [literal\_column()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.literal_column" \o "sqlalchemy.sql.expression.literal_column) can be that, or any other arbitrary column-oriented expression.

literal\_column() 类似于column() ，除了它更常被用作一个“独立”列表达式，正如所陈述的一样; 而column() 存储将被假定为表的一部分的字符串名称，并且可以这样引用，literal\_column() 可以是或任何其他任意的面向列的表达式。

|  |  |
| --- | --- |
| **Parameters:** | * ****text**** – the text of the expression; can be any SQL expression. Quoting rules will not be applied. To specify a column-name expression which should be subject to quoting rules, use the [column()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.column" \o "sqlalchemy.sql.expression.column) function. * ****type\_**** – an optional [TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine) object which will provide result-set translation and additional expression semantics for this column. If left as None the type will be NullType. |

**See also**

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

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

[Using More Specific Text with table(), literal\_column(), and column()](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html" \l "sqlexpression-literal-column)

sqlalchemy.sql.expression.**not\_**(*clause*)

Return a negation of the given clause, i.e. NOT(clause).

The ~ operator is also overloaded on all [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) subclasses to produce the same result.

sqlalchemy.sql.expression.**null**()

Return a constant [Null](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.elements.Null" \o "sqlalchemy.sql.elements.Null) construct.

sqlalchemy.sql.expression.**nullsfirst**(*column*)

Produce the NULLS FIRST modifier for an ORDER BY expression.

[nullsfirst()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.nullsfirst" \o "sqlalchemy.sql.expression.nullsfirst) is intended to modify the expression produced by [asc()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.asc" \o "sqlalchemy.sql.expression.asc) or [desc()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.desc" \o "sqlalchemy.sql.expression.desc), and indicates how NULL values should be handled when they are encountered during ordering:

**from** **sqlalchemy** **import** desc, nullsfirst

stmt = select([users\_table]). order\_by(nullsfirst(desc(users\_table.c.name)))

The SQL expression from the above would resemble:

SELECT id, name FROM user ORDER BY name DESC NULLS FIRST

Like [asc()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.asc" \o "sqlalchemy.sql.expression.asc) and [desc()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.desc" \o "sqlalchemy.sql.expression.desc), [nullsfirst()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.nullsfirst" \o "sqlalchemy.sql.expression.nullsfirst) is typically invoked from the column expression itself using [ColumnElement.nullsfirst()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement.nullsfirst" \o "sqlalchemy.sql.expression.ColumnElement.nullsfirst), rather than as its standalone function version, as in:

stmt = (select([users\_table]).

order\_by(users\_table.c.name.desc().nullsfirst())

)

**See also**

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

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

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

[Select.order\_by()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.order_by" \o "sqlalchemy.sql.expression.Select.order_by)

sqlalchemy.sql.expression.**nullslast**(*column*)

Produce the NULLS LAST modifier for an ORDER BY expression.

为ORDER BY表达式生成NULLS LAST修饰符。

[nullslast()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.nullslast" \o "sqlalchemy.sql.expression.nullslast) is intended to modify the expression produced by [asc()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.asc" \o "sqlalchemy.sql.expression.asc) or [desc()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.desc" \o "sqlalchemy.sql.expression.desc), and indicates how NULL values should be handled when they are encountered during ordering:

nullslast() 旨在修改由asc() 或desc() 生成的表达式，并指示在排序期间遇到NULL值时应如何处理：

**from** **sqlalchemy** **import** desc, nullslast

stmt = select([users\_table]). order\_by(nullslast(desc(users\_table.c.name)))

The SQL expression from the above would resemble:

上面的SQL表达式将类似于：

SELECT id, name FROM user ORDER BY name DESC NULLS LAST

Like [asc()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.asc" \o "sqlalchemy.sql.expression.asc) and [desc()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.desc" \o "sqlalchemy.sql.expression.desc), [nullslast()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.nullslast" \o "sqlalchemy.sql.expression.nullslast) is typically invoked from the column expression itself using [ColumnElement.nullslast()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement.nullslast" \o "sqlalchemy.sql.expression.ColumnElement.nullslast), rather than as its standalone function version, as in:

像asc() 和desc() 一样，nullslast() 通常使用ColumnElement.nullslast() 从列表达式本身调用，而不是作为其独立的函数版本，如：

stmt = select([users\_table]). order\_by(users\_table.c.name.desc().nullslast())

**See also**

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

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

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

[Select.order\_by()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.order_by" \o "sqlalchemy.sql.expression.Select.order_by)

sqlalchemy.sql.expression.**or\_**(*\*clauses*)

Produce a conjunction of expressions joined by OR.

E.g.:

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

stmt = select([users\_table]).where(

or\_(

users\_table.c.name == 'wendy',

users\_table.c.name == 'jack'

)

)

The [or\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.or_" \o "sqlalchemy.sql.expression.or_) conjunction is also available using the Python | operator (though note that compound expressions need to be parenthesized in order to function with Python operator precedence behavior):

or\_() 连接也可以使用Python | 运算符（虽然注意复合表达式需要括起来才能使用Python运算符优先行为）：

stmt = select([users\_table]).where(

(users\_table.c.name == 'wendy') |

(users\_table.c.name == 'jack')

)

**See also**

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

sqlalchemy.sql.expression.**outparam**(*key*, *type\_=None*)

Create an 'OUT' parameter for usage in functions (stored procedures), for databases which support them.

创建一个“OUT”参数，用于函数（存储过程）中，用于支持它们的数据库。

The outparam can be used like a regular function parameter. The "output" value will be available from the [ResultProxy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy" \o "sqlalchemy.engine.ResultProxy) object via its out\_parametersattribute, which returns a dictionary containing the values.

outparam可以像常规函数参数一样使用。 “输出”值将通过其out\_parametersattribute从ResultProxy对象中可用，该对象返回包含值的字典。

sqlalchemy.sql.expression.**over**(*element*, *partition\_by=None*, *order\_by=None*, *range\_=None*, *rows=None*)

Produce an [Over](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.Over" \o "sqlalchemy.sql.expression.Over) object against a function.

针对一个函数产生一个Over对象。

Used against aggregate or so-called "window" functions, for database backends that support window functions.

[over()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.over" \o "sqlalchemy.sql.expression.over) is usually called using the [FunctionElement.over()](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.FunctionElement.over" \o "sqlalchemy.sql.functions.FunctionElement.over) method, e.g.:

用于聚合或所谓的“窗口”功能，用于支持窗口功能的数据库后端。

over() 通常使用FunctionElement.over() 方法调用，例如：

func.row\_number().over(order\_by=mytable.c.some\_column)

Would produce:

ROW\_NUMBER() OVER(ORDER BY some\_column)

Ranges are also possible using the [expression.over.range\_](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.over.params.range_" \o "sqlalchemy.sql.expression.over) and [expression.over.rows](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.over.params.rows" \o "sqlalchemy.sql.expression.over) parameters. These mutually-exclusive parameters each accept a 2-tuple, which contains a combination of integers and None:

范围也可以使用expression.over.range\_和expression.over.rows参数。 这些相互排斥的参数都接受一个2元组，它包含整数和无数组合：

func.row\_number().over(order\_by=my\_table.c.some\_column, range\_=(**None**, 0))

The above would produce:

以上将产生：

ROW\_NUMBER() OVER(ORDER BY some\_column RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)

A value of None indicates "unbounded", a value of zero indicates "current row", and negative / positive integers indicate "preceding" and "following":

值“None”表示“无限制”，零值表示“当前行”，负/正整数表示“前”和“后”：

RANGE BETWEEN 5 PRECEDING AND 10 FOLLOWING:

func.row\_number().over(order\_by='x', range\_=(-5, 10))

ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW:

func.row\_number().over(order\_by='x', rows=(**None**, 0))

RANGE BETWEEN 2 PRECEDING AND UNBOUNDED FOLLOWING:

func.row\_number().over(order\_by='x', range\_=(-2, **None**))

RANGE BETWEEN 1 FOLLOWING AND 3 FOLLOWING:

func.row\_number().over(order\_by='x', range\_=(1, 3))

*New in version 1.1:*support for RANGE / ROWS within a window

|  |  |
| --- | --- |
| **Parameters:** | * ****element**** – a [FunctionElement](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.FunctionElement" \o "sqlalchemy.sql.functions.FunctionElement), [WithinGroup](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.WithinGroup" \o "sqlalchemy.sql.expression.WithinGroup), or other compatible construct.一个FunctionElement，WithinGroup或其他兼容的构造。 * ****partition\_by**** – a column element or string, or a list of such, that will be used as the PARTITION BY clause of the OVER construct.一个列元素或字符串，或这样的列表，将用作OVER构造的PARTITION BY子句。 * ****order\_by**** – a column element or string, or a list of such, that will be used as the ORDER BY clause of the OVER construct.列元素或字符串，或列表，将用作OVER构造的ORDER BY子句。 * ****range\_ –****optional range clause for the window. This is a tuple value which can contain integer values or None, and will render a RANGE BETWEEN PRECEDING / FOLLOWING clause窗口的可选范围子句。 这是一个可以包含整数值的元组值，也可以是无，并将在PRECEDING / FOLLOWING子句之前呈现RANGE   *New in version 1.1.*   * ****rows –****optional rows clause for the window. This is a tuple value which can contain integer values or None, and will render a ROWS BETWEEN PRECEDING / FOLLOWING clause.窗口的可选rows子句。 这是一个可以包含整数值的元组值，也可以是None，并将渲染一个ROWS BETWEEN PRECEDING / FOLLOWING子句。   *New in version 1.1.* |

This function is also 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 itself via the [FunctionElement.over()](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.FunctionElement.over" \o "sqlalchemy.sql.functions.FunctionElement.over) method.

该函数也可以通过FunctionElement.over() 方法从func结构本身中获得。

**See also**

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

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

sqlalchemy.sql.expression.**text**(*text*, *bind=None*, *bindparams=None*, *typemap=None*, *autocommit=None*)

Construct a new [TextClause](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.TextClause" \o "sqlalchemy.sql.expression.TextClause) clause, representing a textual SQL string directly.

E.g.:

构造一个新的TextClause子句，直接表示文本SQL字符串。

例如。：

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

t = text("SELECT \* FROM users")result = connection.execute(t)

The advantages [text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text) provides over a plain string are backend-neutral support for bind parameters, per-statement execution options, as well as bind parameter and result-column typing behavior, allowing SQLAlchemy type constructs to play a role when executing a statement that is specified literally. The construct can also be provided with a .c collection of column elements, allowing it to be embedded in other SQL expression constructs as a subquery.

text() 通过一个纯粹的字符串提供的优点是对绑定参数，每个语句执行选项以及绑定参数和结果列类型行为的后端中立支持，允许SQLAlchemy类型结构在执行语句时发挥作用 是字面上的。 该结构还可以提供一个列的元素的.c集合，允许它作为子查询嵌入其他SQL表达式构造。

Bind parameters are specified by name, using the format :name. E.g.:

绑定参数由名称指定，格式为：name。 例如。：

t = text("SELECT \* FROM users WHERE id=:user\_id")result = connection.execute(t, user\_id=12)

For SQL statements where a colon is required verbatim, as within an inline string, use a backslash to escape:

对于SQL语句，其中必须逐字进行冒号，如同内联字符串一样，请使用反斜杠进行转义：

t = text("SELECT \* FROM users WHERE name='\:username'")

The [TextClause](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.TextClause" \o "sqlalchemy.sql.expression.TextClause) construct includes methods which can provide information about the bound parameters as well as the column values which would be returned from the textual statement, assuming it's an executable SELECT type of statement. The [TextClause.bindparams()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.TextClause.bindparams" \o "sqlalchemy.sql.expression.TextClause.bindparams) method is used to provide bound parameter detail, and [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 allows specification of return columns including names and types:

TextClause结构包括可以提供关于绑定参数的信息的方法以及从文本语句返回的列值，假设它是一个可执行的SELECT类型的语句。 TextClause.bindparams() 方法用于提供绑定的参数详细信息，TextClause.columns() 方法允许指定返回列，包括名称和类型：

t = text("SELECT \* FROM users WHERE id=:user\_id").\

bindparams(user\_id=7).\

columns(id=Integer, name=String)

**for** id, name **in** connection.execute(t):

print(id, name)

The [text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text) construct is used in cases when a literal string SQL fragment is specified as part of a larger query, such as for the WHERE clause of a SELECT statement:

当文本字符串SQL片段被指定为较大查询的一部分（例如SELECT语句的WHERE子句）时，将使用text() 结构：

s = select([users.c.id, users.c.name]).where(text("id=:user\_id"))result = connection.execute(s, user\_id=12)

[text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text) is also used for the construction of a full, standalone statement using plain text. As such, SQLAlchemy refers to it as an [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable) object, and it supports the [Executable.execution\_options()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable.execution_options" \o "sqlalchemy.sql.expression.Executable.execution_options) method. For example, 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 should be subject to "autocommit" can be set explicitly so using the [Connection.execution\_options.autocommit](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.execution_options.params.autocommit" \o "sqlalchemy.engine.Connection.execution_options) option:

text() 也用于使用纯文本构建完整的独立语句。 因此，SQLAlchemy将其称为可执行对象，并支持Executable.execution\_options() 方法。 例如，可以使用Connection.execution\_options.autocommit选项显式设置一个应该“autocommit”的text() 结构，

t = text("EXEC my\_procedural\_thing()").\

execution\_options(autocommit=**True**)

Note that SQLAlchemy's usual "autocommit" behavior applies to [text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text) constructs implicitly - that is, statements which begin with a phrase such as INSERT, UPDATE, DELETE, or a variety of other phrases specific to certain backends, will be eligible for autocommit if no transaction is in progress.

请注意，SQLAlchemy通常的“自动提交”行为隐式应用于text() 结构 - 即以诸如INSERT，UPDATE，DELETE或某些后端特定的各种其他短语开头的语句将有资格自动提交 没有交易正在进行中。

|  |  |
| --- | --- |
| **Parameters:** | * ****text**** – the text of the SQL statement to be created. use :<param> to specify bind parameters; they will be compiled to their engine-specific format. * ****~~autocommit~~****~~– Deprecated. Use .execution\_options(autocommit=<True|False>) to set the autocommit option~~. * ****bind**** – an optional connection or engine to be used for this text query. * ****~~bindparams –~~****~~Deprecated. A list of [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) instances used to provide information about parameters embedded in the statement. This argument now invokes the [TextClause.bindparams()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.TextClause.bindparams" \o "sqlalchemy.sql.expression.TextClause.bindparams) method on the construct before returning it. E.g.:~~   stmt = text("SELECT \* FROM table WHERE id=:id",  bindparams=[bindparam('id', value=5, type\_=Integer)])  Is equivalent to:  stmt = text("SELECT \* FROM table WHERE id=:id").\  bindparams(bindparam('id', value=5, type\_=Integer))  *Deprecated since version 0.9.0:*the [TextClause.bindparams()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.TextClause.bindparams" \o "sqlalchemy.sql.expression.TextClause.bindparams) method supersedes the bindparams argument to [text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text).   * ****typemap –****Deprecated. A dictionary mapping the names of columns represented in the columns clause of a SELECT statement to type objects, which will be used to perform post-processing on columns within the result set. This parameter now invokes 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, which returns a[TextAsFrom](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TextAsFrom" \o "sqlalchemy.sql.expression.TextAsFrom) construct that gains a .c collection and can be embedded in other expressions. E.g.:   stmt = text("SELECT \* FROM table",  typemap={'id': Integer, 'name': String},  )  Is equivalent to:  stmt = text("SELECT \* FROM table").columns(id=Integer,  name=String)  Or alternatively:  **from** **sqlalchemy.sql** **import** columnstmt = text("SELECT \* FROM table").columns(  column('id', Integer),  column('name', String)  )  *Deprecated since version 0.9.0:*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 supersedes the typemap argument to [text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text). |

**See also**

[Using Textual SQL](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html" \l "sqlexpression-text) - in the Core tutorial

[Using Textual SQL](http://docs.sqlalchemy.org/en/rel_1_1/orm/tutorial.html" \l "orm-tutorial-literal-sql) - in the ORM tutorial

sqlalchemy.sql.expression.**true**()

Return a constant [True\_](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.elements.True_" \o "sqlalchemy.sql.elements.True_) construct.

E.g.:

**>>> from** **sqlalchemy** **import** true

**>>>** print select([t.c.x]).where(true())SELECT x FROM t WHERE true

A backend which does not support true/false constants will render as an expression against 1 or 0:

不支持true / false常量的后端将呈现为1或0的表达式：

**>>>** print select([t.c.x]).where(true())SELECT x FROM t WHERE 1 = 1

The [true()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.true" \o "sqlalchemy.sql.expression.true) and [false()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.false" \o "sqlalchemy.sql.expression.false) constants also feature "short circuit" operation within an [and\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.and_" \o "sqlalchemy.sql.expression.and_) or [or\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.or_" \o "sqlalchemy.sql.expression.or_) conjunction:

true() 和false() 常量也在and\_() 或or\_() 连接中具有“短路”操作：

**>>>** print select([t.c.x]).where(or\_(t.c.x > 5, true()))

SELECT x FROM t WHERE true

**>>>** print select([t.c.x]).where(and\_(t.c.x > 5, false()))

SELECT x FROM t WHERE false

*Changed in version 0.9:*[true()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.true" \o "sqlalchemy.sql.expression.true) and [false()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.false" \o "sqlalchemy.sql.expression.false) feature better integrated behavior within conjunctions and on dialects that don't support true/false constants.

在版本0.9中更改：true() 和false() 在连接中和不支持true / false常量的方言上具有更好的集成行为。

**See also**

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

sqlalchemy.sql.expression.**tuple\_**(*\*clauses*, *\*\*kw*)

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

Main usage is to produce a composite IN construct:

主要用途是生成一个复合IN结构：

**from** **sqlalchemy** **import** tuple\_

tuple\_(table.c.col1, table.c.col2).in\_(

[(1, 2), (5, 12), (10, 19)])

**Warning**

The composite IN construct is not supported by all backends, and is currently known to work on PostgreSQL and MySQL, but not SQLite. Unsupported backends will raise a subclass of [DBAPIError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.DBAPIError" \o "sqlalchemy.exc.DBAPIError) when such an expression is invoked.

所有后端都不支持复合IN结构，目前已知它可以在PostgreSQL和MySQL上工作，而不是SQLite。 当调用这样的表达式时，不支持的后端将引发DBAPIError的子类。

sqlalchemy.sql.expression.**type\_coerce**(*expression*, *type\_*)

Associate a SQL expression with a particular type, without rendering CAST.

E.g.:

将SQL表达式与特定类型相关联，而无需呈现CAST。

例如。：

**from** **sqlalchemy** **import** type\_coerce

stmt = select([

type\_coerce(log\_table.date\_string, StringDateTime())])

The above construct will produce a [TypeCoerce](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.TypeCoerce" \o "sqlalchemy.sql.expression.TypeCoerce) object, which renders SQL that labels the expression, but otherwise does not modify its value on the SQL side:

上述构造将生成一个TypeCoerce对象，该对象将呈现标记表达式的SQL，否则不会在SQL端修改其值：

SELECT date\_string AS anon\_1 FROM log

When result rows are fetched, the StringDateTime type will be applied to result rows on behalf of the date\_string column. The rationale for the "anon\_1" label is so that the type-coerced column remains separate in the list of result columns vs. other type-coerced or direct values of the target column. In order to provide a named label for the expression, use [ColumnElement.label()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement.label" \o "sqlalchemy.sql.expression.ColumnElement.label):

当获取结果行时，StringDateTime类型将代表date\_string列应用于结果行。 “anon\_1”标签的理由是，类型强制列在结果列列表中与目标列的其他类型强制或直接值保持分离。 为了提供表达式的命名标签，请使用ColumnElement.label() ：

stmt = select([

type\_coerce(

log\_table.date\_string, StringDateTime()).label('date')])

A type that features bound-value handling will also have that behavior take effect when literal values or [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) constructs are passed to [type\_coerce()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.type_coerce" \o "sqlalchemy.sql.expression.type_coerce) as targets. For example, if a type implements the [TypeEngine.bind\_expression()](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.bind_expression" \o "sqlalchemy.types.TypeEngine.bind_expression) method or [TypeEngine.bind\_processor()](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.bind_processor" \o "sqlalchemy.types.TypeEngine.bind_processor)method or equivalent, these functions will take effect at statement compilation/execution time when a literal value is passed, as in:

当将文字值或bindparam() 构造作为目标传递给type\_coerce() 时，具有绑定值处理功能的类型也将有效。 例如，如果一个类型实现了TypeEngine.bind\_expression() 方法或TypeEngine.bind\_processor() 方法或等效项，这些函数将在传递文字值时在语句编译/执行时生效，如下所示：

*# bound-value handling of MyStringType will be applied to the# literal value "some string"*

stmt = select([type\_coerce("some string", MyStringType)])

[type\_coerce()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.type_coerce" \o "sqlalchemy.sql.expression.type_coerce) is similar to the [cast()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.cast" \o "sqlalchemy.sql.expression.cast) function, except that it does not render the CAST expression in the resulting statement.

type\_coerce() 类似于cast() 函数，但它不会在生成的语句中呈现CAST表达式。

|  |  |
| --- | --- |
| **Parameters:** | * ****expression**** – A SQL expression, such as a [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) expression or a Python string which will be coerced into a bound literal value.SQL表达式，例如ColumnElement表达式或将被强制转换为绑定文字值的Python字符串。 * ****type\_**** – A [TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine) class or instance indicating the type to which the expression is coerced.一个TypeEngine类或实例，指示表达式被胁迫的类型。 |

**See also**

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

sqlalchemy.sql.expression.**within\_group**(*element*, *\*order\_by*)

Produce a [WithinGroup](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.WithinGroup" \o "sqlalchemy.sql.expression.WithinGroup) object against a function.

针对函数生成WithinGroup对象。

Used against so-called "ordered set aggregate" and "hypothetical set aggregate" functions, including [percentile\_cont](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.percentile_cont" \o "sqlalchemy.sql.functions.percentile_cont), [rank](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.rank" \o "sqlalchemy.sql.functions.rank), [dense\_rank](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.dense_rank" \o "sqlalchemy.sql.functions.dense_rank), etc.

用于所谓的“有序集合”和“假设集合”功能，包括percentile\_cont，rank，dense\_rank等。

[within\_group()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.within_group" \o "sqlalchemy.sql.expression.within_group) is usually called using the [FunctionElement.within\_group()](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.FunctionElement.within_group" \o "sqlalchemy.sql.functions.FunctionElement.within_group) method, e.g.:

within\_group() 通常使用FunctionElement.within\_group() 方法调用，例如：

**from** **sqlalchemy** **import** within\_group

stmt = select([

department.c.id,

func.percentile\_cont(0.5).within\_group(

department.c.salary.desc()

)])

The above statement would produce SQL similar to SELECT department.id, percentile\_cont(0.5) WITHIN GROUP (ORDER BYdepartment.salary DESC).

上述语句将产生类似于SELECT department.id，percentile\_cont（0.5）WITHIN GROUP（ORDER BYDepartment.salary DESC）的SQL。

|  |  |
| --- | --- |
| **Parameters:** | * ****element**** – a [FunctionElement](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.FunctionElement" \o "sqlalchemy.sql.functions.FunctionElement) construct, typically generated by [func](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.func" \o "sqlalchemy.sql.expression.func). * ****\*order\_by**** – one or more column elements that will be used as the ORDER BY clause of the WITHIN GROUP construct. |

*New in version 1.1.*

**See also**

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

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

*class*sqlalchemy.sql.expression.**BinaryExpression**(*left*, *right*, *operator*, *type\_=None*, *negate=None*, *modifiers=None*)

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

Represent an expression that is LEFT <operator> RIGHT.

表示一个LEFT <operator> RIGHT的表达式。

A [BinaryExpression](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.BinaryExpression" \o "sqlalchemy.sql.expression.BinaryExpression) is generated automatically whenever two column expressions are used in a Python binary expression:

当Python二进制表达式中使用两列表达式时，将自动生成BinaryExpression：

**>>> from** **sqlalchemy.sql** **import** column

**>>>** column('a') + column('b')

<sqlalchemy.sql.expression.BinaryExpression object at 0x101029dd0>

**>>>** print column('a') + column('b')

a + b

**compare**(*other*, *\*\*kw*)

Compare this [BinaryExpression](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.BinaryExpression" \o "sqlalchemy.sql.expression.BinaryExpression) against the given [BinaryExpression](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.BinaryExpression" \o "sqlalchemy.sql.expression.BinaryExpression).

*class*sqlalchemy.sql.expression.**BindParameter**(*key*, *value=symbol('NO\_ARG')*, *type\_=None*, *unique=False*, *required=symbol('NO\_ARG')*, *quote=None*, *callable\_=None*, *isoutparam=False*, *\_compared\_to\_operator=None*, *\_compared\_to\_type=None*)

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

Represent a "bound expression".

表示“绑定表达式”。

[BindParameter](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.BindParameter" \o "sqlalchemy.sql.expression.BindParameter) is invoked explicitly using the [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) function, as in:

使用bindparam() 函数显式调用BindParameter，如：

**from** **sqlalchemy** **import** bindparam

stmt = select([users\_table]).\

where(users\_table.c.name == bindparam('username'))

Detailed discussion of how [BindParameter](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.BindParameter" \o "sqlalchemy.sql.expression.BindParameter) is used is at [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam).

如何使用BindParameter的详细讨论是在bindparam() 。

**See also**

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

**\_\_init\_\_**(*key*, *value=symbol('NO\_ARG')*, *type\_=None*, *unique=False*, *required=symbol('NO\_ARG')*, *quote=None*, *callable\_=None*, *isoutparam=False*, *\_compared\_to\_operator=None*, *\_compared\_to\_type=None*)

Construct a new [BindParameter](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.BindParameter" \o "sqlalchemy.sql.expression.BindParameter) object.

构造一个新的BindParameter对象。

This constructor is mirrored as a public API function; see [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) for a full usage and argument description.

该构造函数作为公共API函数进行镜像; 有关完整用法和参数描述，请参阅bindparam() 。

**compare**(*other*, *\*\*kw*)

Compare this [BindParameter](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.BindParameter" \o "sqlalchemy.sql.expression.BindParameter) to the given clause.

**effective\_value**

Return the value of this bound parameter, taking into account if the callable parameter was set.

返回此绑定参数的值，同时考虑到是否设置了可调用参数。

The callable value will be evaluated and returned if present, else value.

可调用值将被评估并返回（如果存在），否则返回值。

*class*sqlalchemy.sql.expression.**Case**(*whens*, *value=None*, *else\_=None*)

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

Represent a CASE expression.

[Case](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.Case" \o "sqlalchemy.sql.expression.Case) is produced using the [case()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.case" \o "sqlalchemy.sql.expression.case) factory function, as in:

**from** **sqlalchemy** **import** case

stmt = select([users\_table]). where(

case(

[

(users\_table.c.name == 'wendy', 'W'),

(users\_table.c.name == 'jack', 'J')

],

else\_='E'

)

)

Details on [Case](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.Case" \o "sqlalchemy.sql.expression.Case) usage is at [case()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.case" \o "sqlalchemy.sql.expression.case).

**See also**

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

**\_\_init\_\_**(*whens*, *value=None*, *else\_=None*)

Construct a new [Case](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.Case" \o "sqlalchemy.sql.expression.Case) object.

构造一个新的Case对象。

This constructor is mirrored as a public API function; see [case()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.case" \o "sqlalchemy.sql.expression.case) 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case() 以获取完整的用法和参数描述。

*class*sqlalchemy.sql.expression.**Cast**(*expression*, *type\_*)

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

Represent a CAST expression.

代表CAST表达式。

[Cast](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.Cast" \o "sqlalchemy.sql.expression.Cast) is produced using the [cast()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.cast" \o "sqlalchemy.sql.expression.cast) factory function, as in:

Cast是使用cast() 工厂函数生成的，如：

**from** **sqlalchemy** **import** cast, Numeric

stmt = select([

cast(product\_table.c.unit\_price, Numeric(10, 4))

])

Details on [Cast](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.Cast" \o "sqlalchemy.sql.expression.Cast) usage is at [cast()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.cast" \o "sqlalchemy.sql.expression.cast).

**See also**

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

**\_\_init\_\_**(*expression*, *type\_*)

Construct a new [Cast](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.Cast" \o "sqlalchemy.sql.expression.Cast) object.

构造一个新的Cast对象。

This constructor is mirrored as a public API function; see [cast()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.cast" \o "sqlalchemy.sql.expression.cast) 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cast() 以获取完整的用法和参数说明。

*class*sqlalchemy.sql.expression.**ClauseElement**

Bases: sqlalchemy.sql.visitors.Visitable

Base class for elements of a programmatically constructed SQL expression.

用于以编程方式构造的SQL表达式的元素的基类。

**compare**(*other*, *\*\*kw*)

Compare this ClauseElement to the given ClauseElement.

将此条款元素与给定的条款元素进行比较。

Subclasses should override the default behavior, which is a straight identity comparison.

子类应该覆盖默认行为，这是一个直接的身份比较。

\*\*kw are arguments consumed by subclass compare() methods and may be used to modify the criteria for comparison. (see [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement))

\*\* kw是由subclass compare（）方法消耗的参数，可用于修改比较条件。（见ColumnElement）

**compile**(*bind=None*, *dialect=None*, *\*\*kw*)

Compile this SQL expression.

编译此SQL表达式。

The return value is a [Compiled](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Compiled" \o "sqlalchemy.engine.interfaces.Compiled) object. Calling str() or unicode() on the returned value will yield a string representation of the result. The[Compiled](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Compiled" \o "sqlalchemy.engine.interfaces.Compiled) object also can return a dictionary of bind parameter names and values using the params accessor.

返回值是一个编译对象。 对返回的值调用str（）或unicode（）将产生结果的字符串表示形式。 Compiled对象还可以使用params访问器返回绑定参数名称和值的字典。

|  |  |
| --- | --- |
| **Parameters:** | * ****bind**** – An Engine or Connection from which a Compiled will be acquired. This argument takes precedence over this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)'s bound engine, if any. * ****column\_keys**** – Used for INSERT and UPDATE statements, a list of column names which should be present in the VALUES clause of the compiled statement. If None, all columns from the target table object are rendered. * ****dialect**** – A Dialect instance from which a Compiled will be acquired. This argument takes precedence over the bind argument as well as this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)'s bound engine, if any. * ****inline**** – Used for INSERT statements, for a dialect which does not support inline retrieval of newly generated primary key columns, will force the expression used to create the new primary key value to be rendered inline within the INSERT statement's VALUES clause. This typically refers to Sequence execution but may also refer to any server-side default generation function associated with a primary key Column. * ****compile\_kwargs –****optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the literal\_binds flag through:   **from** **sqlalchemy.sql** **import** table, column, select  t = table('t', column('x'))  s = select([t]).where(t.c.x == 5)  print s.compile(compile\_kwargs={"literal\_binds": **True**})  *New in version 0.9.0.* |

**See also**

[How do I render SQL expressions as strings, possibly with bound parameters inlined?](http://docs.sqlalchemy.org/en/rel_1_1/faq/sqlexpressions.html" \l "faq-sql-expression-string)

**get\_children**(*\*\*kwargs*)

Return immediate child elements of this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement).

This is used for visit traversal.

\*\*kwargs may contain flags that change the collection that is returned, for example to return a subset of items in order to cut down on larger traversals, or to return child items from a different context (such as schema-level collections instead of clause-level).

**params**(*\*optionaldict*, *\*\*kwargs*)

Return a copy with [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) elements replaced.

Returns a copy of this ClauseElement with [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) elements replaced with values taken from the given dictionary:

**>>>** clause = column('x') + bindparam('foo')**>>>** print clause.compile().params{'foo':None}**>>>** print clause.params({'foo':7}).compile().params{'foo':7}

**self\_group**(*against=None*)

Apply a 'grouping' to this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement).

This method is overridden by subclasses to return a "grouping" construct, i.e. parenthesis. In particular it's used by "binary" expressions to provide a grouping around themselves when placed into a larger expression, as well as by [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) constructs when placed into the FROM clause of another [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select). (Note that subqueries should be normally created using the [Select.alias()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.alias" \o "sqlalchemy.sql.expression.Select.alias) method, as many platforms require nested SELECT statements to be named).

As expressions are composed together, the application of [self\_group()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.self_group" \o "sqlalchemy.sql.expression.ClauseElement.self_group) is automatic - end-user code should never need to use this method directly. Note that SQLAlchemy's clause constructs take operator precedence into account - so parenthesis might not be needed, for example, in an expression like x OR (y AND z) - AND takes precedence over OR.

The base [self\_group()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.self_group" \o "sqlalchemy.sql.expression.ClauseElement.self_group) method of [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement) just returns self.

**unique\_params**(*\*optionaldict*, *\*\*kwargs*)

Return a copy with [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) elements replaced.

Same functionality as params(), except adds unique=True to affected bind parameters so that multiple statements can be used.

*class*sqlalchemy.sql.expression.**ClauseList**(*\*clauses*, *\*\*kwargs*)

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

Describe a list of clauses, separated by an operator.

By default, is comma-separated, such as a column listing.

**compare**(*other*, *\*\*kw*)

Compare this [ClauseList](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseList" \o "sqlalchemy.sql.expression.ClauseList) to the given [ClauseList](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseList" \o "sqlalchemy.sql.expression.ClauseList), including a comparison of all the clause items.

*class*sqlalchemy.sql.expression.**ColumnClause**(*text*, *type\_=None*, *is\_literal=False*, *\_selectable=None*)

Bases: sqlalchemy.sql.expression.Immutable, [sqlalchemy.sql.expression.ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement)

Represents a column expression from any textual string.

The [ColumnClause](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnClause" \o "sqlalchemy.sql.expression.ColumnClause), a lightweight analogue to the [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) class, is typically invoked using the [column()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.column" \o "sqlalchemy.sql.expression.column) function, as in:

**from** **sqlalchemy** **import** column

id, name = column("id"), column("name")stmt = select([id, name]).select\_from("user")

The above statement would produce SQL like:

SELECT id, name FROM user

[ColumnClause](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnClause" \o "sqlalchemy.sql.expression.ColumnClause) is the immediate superclass of the schema-specific [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) object. While the [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) class has all the same capabilities as [ColumnClause](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnClause" \o "sqlalchemy.sql.expression.ColumnClause), the [ColumnClause](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnClause" \o "sqlalchemy.sql.expression.ColumnClause) class is usable by itself in those cases where behavioral requirements are limited to simple SQL expression generation. The object has none of the associations with schema-level metadata or with execution-time behavior that [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) does, so in that sense is a "lightweight" version of [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column).

Full details on [ColumnClause](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnClause" \o "sqlalchemy.sql.expression.ColumnClause) usage is at [column()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.column" \o "sqlalchemy.sql.expression.column).

**See also**

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

[Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column)

**\_\_init\_\_**(*text*, *type\_=None*, *is\_literal=False*, *\_selectable=None*)

Construct a new [ColumnClause](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnClause" \o "sqlalchemy.sql.expression.ColumnClause) object.

This constructor is mirrored as a public API function; see [column()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.column" \o "sqlalchemy.sql.expression.column) for a full usage and argument description.

*class*sqlalchemy.sql.expression.**ColumnCollection**(*\*columns*)

Bases: sqlalchemy.util.\_collections.OrderedProperties

An ordered dictionary that stores a list of ColumnElement instances.

Overrides the \_\_eq\_\_() method to produce SQL clauses between sets of correlated columns.

**add**(*column*)

Add a column to this collection.

The key attribute of the column will be used as the hash key for this dictionary.

**replace**(*column*)

add the given column to this collection, removing unaliased versions of this column as well as existing columns with the same key.

e.g.:

t = Table('sometable', metadata, Column('col1', Integer))t.columns.replace(Column('col1', Integer, key='columnone'))

will remove the original 'col1' from the collection, and add the new column under the name 'columnname'.

Used by schema.Column to override columns during table reflection.

*class*sqlalchemy.sql.expression.**ColumnElement**

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

Represent a column-oriented SQL expression suitable for usage in the "columns" clause, WHERE clause etc. of a statement.

While the most familiar kind of [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) is the [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) object, [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) serves as the basis for any unit that may be present in a SQL expression, including the expressions themselves, SQL functions, bound parameters, literal expressions, keywords such as NULL, etc. [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) is the ultimate base class for all such elements.

A wide variety of SQLAlchemy Core functions work at the SQL expression level, and are intended to accept instances of [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) as arguments. These functions will typically document that they accept a "SQL expression" as an argument. What this means in terms of SQLAlchemy usually refers to an input which is either already in the form of a [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) object, or a value which can be ****coerced**** into one. The coercion rules followed by most, but not all, SQLAlchemy Core functions with regards to SQL expressions are as follows:

* a literal Python value, such as a string, integer or floating point value, boolean, datetime, Decimal object, or virtually any other Python object, will be coerced into a "literal bound value". This generally means that a [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) will be produced featuring the given value embedded into the construct; the resulting [BindParameter](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.BindParameter" \o "sqlalchemy.sql.expression.BindParameter) object is an instance of [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement). The Python value will ultimately be sent to the DBAPI at execution time as a parameterized argument to the execute() or executemany() methods, after SQLAlchemy type-specific converters (e.g. those provided by any associated [TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine) objects) are applied to the value.
* any special object value, typically ORM-level constructs, which feature a method called \_\_clause\_element\_\_(). The Core expression system looks for this method when an object of otherwise unknown type is passed to a function that is looking to coerce the argument into a [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) expression. The \_\_clause\_element\_\_() method, if present, should return a [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) instance. The primary use of \_\_clause\_element\_\_() within SQLAlchemy is that of class-bound attributes on ORM-mapped classes; a User class which contains a mapped attribute named .name will have a method User.name.\_\_clause\_element\_\_() which when invoked returns the[Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) called name associated with the mapped table.
* The Python None value is typically interpreted as NULL, which in SQLAlchemy Core produces an instance of [null()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.null" \o "sqlalchemy.sql.expression.null).

A [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) provides the ability to generate new [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) objects using Python expressions. This means that Python operators such as ==, != and < are overloaded to mimic SQL operations, and allow the instantiation of further [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) instances which are composed from other, more fundamental [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) objects. For example, two [ColumnClause](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnClause" \o "sqlalchemy.sql.expression.ColumnClause) objects can be added together with the addition operator + to produce a [BinaryExpression](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.BinaryExpression" \o "sqlalchemy.sql.expression.BinaryExpression). Both [ColumnClause](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnClause" \o "sqlalchemy.sql.expression.ColumnClause) and [BinaryExpression](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.BinaryExpression" \o "sqlalchemy.sql.expression.BinaryExpression) are subclasses of [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement):

**>>> from** **sqlalchemy.sql** **import** column**>>>** column('a') + column('b')<sqlalchemy.sql.expression.BinaryExpression object at 0x101029dd0>**>>>** print column('a') + column('b')a + b

**See also**

[Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column)

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

**\_\_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.

**\_\_init\_\_**

*inherited from the* \_\_init\_\_ *attribute of* object

x.\_\_init\_\_(…) initializes x; see help(type(x)) for signature

**\_\_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.

**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.*

**anon\_label**

provides a constant 'anonymous label' for this ColumnElement.

This is a label() expression which will be named at compile time. The same label() is returned each time anon\_label is called so that expressions can reference anon\_label multiple times, producing the same label name at compile time.

the compiler uses this function automatically at compile time for expressions that are known to be 'unnamed' like binary expressions and function calls.

**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.

**base\_columns**

**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.

**bind***= None*

**cast**(*type\_*)

Produce a type cast, i.e. CAST(<expression> AS <type>).

This is a shortcut to the [cast()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.cast" \o "sqlalchemy.sql.expression.cast) function.

*New in version 1.0.7.*

**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.

**comparator**

**compare**(*other*, *use\_proxies=False*, *equivalents=None*, *\*\*kw*)

Compare this ColumnElement to another.

Special arguments understood:

|  |  |
| --- | --- |
| **Parameters:** | * ****use\_proxies**** – when True, consider two columns that share a common base column as equivalent (i.e. shares\_lineage())当为True时，考虑两个共享同一基本列的列(即shares\_lineage()) * ****equivalents**** – a dictionary of columns as keys mapped to sets of columns. If the given "other" column is present in this dictionary, if any of the columns in the corresponding set() pass the comparison test, the result is True. This is used to expand the comparison to other columns that may be known to be equivalent to this one via foreign key or other criterion.作为映射到列集的键的列字典。 如果给定的“其他”列出现在这个字典中，如果对应的set()中的任何一列通过了比较测试，结果为真。 这被用来扩展比较到其他列可能被认为是相当于这个通过外键或其他标准。 |

**compile**(*bind=None*, *dialect=None*, *\*\*kw*)

*inherited from the* [compile()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.compile" \o "sqlalchemy.sql.expression.ClauseElement.compile) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Compile this SQL expression.

编译这个SQL表达式。

The return value is a [Compiled](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Compiled" \o "sqlalchemy.engine.interfaces.Compiled) object. Calling str() or unicode() on the returned value will yield a string representation of the result. The[Compiled](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Compiled" \o "sqlalchemy.engine.interfaces.Compiled) object also can return a dictionary of bind parameter names and values using the params accessor.

返回值是一个编译对象。 在返回的值上调用str（）或unicode（）将产生结果的字符串表示形式。 编译对象也可以使用params访问器返回一个绑定参数名称和值的字典。

|  |  |
| --- | --- |
| **Parameters:** | * ****bind**** – An Engine or Connection from which a Compiled will be acquired. This argument takes precedence over this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)'s bound engine, if any.将从中获取编译的引擎或连接。 这个参数优先于这个ClauseElement的绑定引擎，如果有的话。 * ****column\_keys**** – Used for INSERT and UPDATE statements, a list of column names which should be present in the VALUES clause of the compiled statement. If None, all columns from the target table object are rendered.用于INSERT和UPDATE语句，在编译语句的VALUES子句中应该出现的列名列表。 如果None，则渲染目标表格对象中的所有列 * ****dialect**** – A Dialect instance from which a Compiled will be acquired. This argument takes precedence over the bind argument as well as this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)'s bound engine, if any.将从中获取Compiled 的Dialect 实例。 这个参数优先于绑定参数以及这个[ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)绑定的引擎（如果有的话）。 * ****inline**** – Used for INSERT statements, for a dialect which does not support inline retrieval of newly generated primary key columns, will force the expression used to create the new primary key value to be rendered inline within the INSERT statement's VALUES clause. This typically refers to Sequence execution but may also refer to any server-side default generation function associated with a primary key Column.用于INSERT语句，对于不支持内联检索新生成的主键列的方言，将强制将用于创建新主键值的表达式内联呈现在INSERT语句的VALUES子句中。 这通常指的是序列执行，但也可能涉及任何与主键列相关联的服务器端默认生成函数。 * ****compile\_kwargs –****optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the literal\_binds flag through:在所有“访问”方法中将传递给编译器的附加参数的可选字典。 例如，这允许将自定义标志传递给自定义编译结构。 它也用于传递literal\_binds标志的情况：   **from** **sqlalchemy.sql** **import** table, column, select  t = table('t', column('x'))  s = select([t]).where(t.c.x == 5)  print s.compile(compile\_kwargs={"literal\_binds": **True**})  *New in version 0.9.0.* |

**See also**

[How do I render SQL expressions as strings, possibly with bound parameters inlined?](http://docs.sqlalchemy.org/en/rel_1_1/faq/sqlexpressions.html" \l "faq-sql-expression-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.

实现'concat'操作符。

In a column context, produces the clause a || b, or uses the concat() operator on MySQL.

在列上下文中，生成子句a || b，或者在MySQL上使用concat（）运算符。

**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.

**description***= None*

**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>'

**expression**

Return a column expression.

Part of the inspection interface; returns self.

**foreign\_keys***= []*

**get\_children**(*\*\*kwargs*)

*inherited from the* [get\_children()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.get_children" \o "sqlalchemy.sql.expression.ClauseElement.get_children) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Return immediate child elements of this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement).

返回此[ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)的直接子元素。

This is used for visit traversal.

这用于访问遍历。

\*\*kwargs may contain flags that change the collection that is returned, for example to return a subset of items in order to cut down on larger traversals, or to return child items from a different context (such as schema-level collections instead of clause-level).

\*\* kwargs可能包含标志，这些标志会改变返回的集合，例如为了减少较大的遍历而返回一个项目子集，或者从不同的上下文中返回子项目（比如模式级集合而不是子句 -水平）。

**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\_clause\_element***= True*

**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.*

**is\_selectable***= False*

**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.*

**key***= None*

the 'key' that in some circumstances refers to this object in a Python namespace.

This typically refers to the "key" of the column as present in the .c collection of a selectable, e.g. sometable.c["somekey"] would return a Column with a .key of "somekey".

**label**(*name*)

Produce a column label, i.e. <columnname> AS <name>.

This is a shortcut to the [label()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.label" \o "sqlalchemy.sql.expression.label) function.

if 'name' is None, an anonymous label name will be generated.

**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. * ****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)

**operate**(*op*, *\*other*, *\*\*kwargs*)

**params**(*\*optionaldict*, *\*\*kwargs*)

*inherited from the* [params()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.params" \o "sqlalchemy.sql.expression.ClauseElement.params) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Return a copy with [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) elements replaced.

Returns a copy of this ClauseElement with [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) elements replaced with values taken from the given dictionary:

**>>>** clause = column('x') + bindparam('foo')

**>>>** print clause.compile().params{'foo':None}

**>>>** print clause.params({'foo':7}).compile().params{'foo':7}

**primary\_key***= False*

**proxy\_set**

**reverse\_operate**(*op*, *other*, *\*\*kwargs*)

**self\_group**(*against=None*)

**shares\_lineage**(*othercolumn*)

Return True if the given [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) has a common ancestor to this [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement).

**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>%'

**supports\_execution***= False*

**timetuple***= None*

**type**

**unique\_params**(*\*optionaldict*, *\*\*kwargs*)

*inherited from the* [unique\_params()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.unique_params" \o "sqlalchemy.sql.expression.ClauseElement.unique_params) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Return a copy with [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) elements replaced.

Same functionality as params(), except adds unique=True to affected bind parameters so that multiple statements can be used.

*class*sqlalchemy.sql.operators.**ColumnOperators**

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

Defines boolean, comparison, and other operators for [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) expressions.

By default, all methods call down to [operate()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.Operators.operate" \o "sqlalchemy.sql.operators.Operators.operate) or [reverse\_operate()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement.reverse_operate" \o "sqlalchemy.sql.expression.ColumnElement.reverse_operate), passing in the appropriate operator function from the Python builtin operatormodule or a SQLAlchemy-specific operator function from sqlalchemy.expression.operators. For example the \_\_eq\_\_ function:

**def** \_\_eq\_\_(self, other):

**return** self.operate(operators.eq, other)

Where operators.eq is essentially:

**def** eq(a, b):

**return** a == b

The core column expression unit [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) overrides [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) and others to return further [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) constructs, so that the == operation above is replaced by a clause construct.

See also:

[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)

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

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

**\_\_add\_\_**(*other*)

Implement the + operator.

In a column context, produces the clause a + b if the parent object has non-string affinity. If the parent object has a string affinity, produces the concatenation operator, a || b - see [ColumnOperators.concat()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.concat" \o "sqlalchemy.sql.operators.ColumnOperators.concat).

**\_\_and\_\_**(*other*)

*inherited from the* [\_\_and\_\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.Operators.__and__" \o "sqlalchemy.sql.operators.Operators.__and__) *method of* [Operators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.Operators" \o "sqlalchemy.sql.operators.Operators)

Implement the & operator.

When used with SQL expressions, results in an AND operation, equivalent to [and\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.and_" \o "sqlalchemy.sql.expression.and_), that is:

a & b

is equivalent to:

**from** **sqlalchemy** **import** and\_and\_(a, b)

Care should be taken when using & regarding operator precedence; the & operator has the highest precedence. The operands should be enclosed in parenthesis if they contain further sub expressions:

(a == 2) & (b == 4)

**\_\_delattr\_\_**

*inherited from the* \_\_delattr\_\_ *attribute of* object

x.\_\_delattr\_\_('name') <==> del x.name

**\_\_div\_\_**(*other*)

Implement the / operator.

In a column context, produces the clause a / b.

**\_\_eq\_\_**(*other*)

Implement the == operator.

In a column context, produces the clause a = b. If the target is None, produces a IS NULL.

**\_\_format\_\_**()

*inherited from the* \_\_format\_\_() *method of* object

default object formatter

**\_\_ge\_\_**(*other*)

Implement the >= operator.

In a column context, produces the clause a >= b.

**\_\_getattribute\_\_**

*inherited from the* \_\_getattribute\_\_ *attribute of* object

x.\_\_getattribute\_\_('name') <==> x.name

**\_\_getitem\_\_**(*index*)

Implement the [] operator.

This can be used by some database-specific types such as PostgreSQL ARRAY and HSTORE.

**\_\_gt\_\_**(*other*)

Implement the > operator.

In a column context, produces the clause a > b.

**\_\_hash\_\_**

**\_\_init\_\_**

*inherited from the* \_\_init\_\_ *attribute of* object

x.\_\_init\_\_(…) initializes x; see help(type(x)) for signature

**\_\_invert\_\_**()

*inherited from the* [\_\_invert\_\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.Operators.__invert__" \o "sqlalchemy.sql.operators.Operators.__invert__) *method of* [Operators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.Operators" \o "sqlalchemy.sql.operators.Operators)

Implement the ~ operator.

When used with SQL expressions, results in a NOT operation, equivalent to [not\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.not_" \o "sqlalchemy.sql.expression.not_), that is:

~a

is equivalent to:

**from** **sqlalchemy** **import** not\_

not\_(a)

**\_\_le\_\_**(*other*)

Implement the <= operator.

In a column context, produces the clause a <= b.

**\_\_lshift\_\_**(*other*)

implement the << operator.

Not used by SQLAlchemy core, this is provided for custom operator systems which want to use << as an extension point.

**\_\_lt\_\_**(*other*)

Implement the < operator.

In a column context, produces the clause a < b.

**\_\_mod\_\_**(*other*)

Implement the % operator.

In a column context, produces the clause a % b.

**\_\_mul\_\_**(*other*)

Implement the \* operator.

In a column context, produces the clause a \* b.

**\_\_ne\_\_**(*other*)

Implement the != operator.

In a column context, produces the clause a != b. If the target is None, produces a IS NOT NULL.

**\_\_neg\_\_**()

Implement the - operator.

In a column context, produces the clause -a.

**\_\_new\_\_**(*S*, *...*) → a new object with type S, a subtype of T

*inherited from the* \_\_new\_\_() *method of* object

**\_\_or\_\_**(*other*)

*inherited from the* [\_\_or\_\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.Operators.__or__" \o "sqlalchemy.sql.operators.Operators.__or__) *method of* [Operators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.Operators" \o "sqlalchemy.sql.operators.Operators)

Implement the | operator.

When used with SQL expressions, results in an OR operation, equivalent to [or\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.or_" \o "sqlalchemy.sql.expression.or_), that is:

a | b

is equivalent to:

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

or\_(a, b)

Care should be taken when using | regarding operator precedence; the | operator has the highest precedence. The operands should be enclosed in parenthesis if they contain further sub expressions:

(a == 2) | (b == 4)

**\_\_radd\_\_**(*other*)

Implement the + operator in reverse.

See [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__).

**\_\_rdiv\_\_**(*other*)

Implement the / operator in reverse.

See [ColumnOperators.\_\_div\_\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.__div__" \o "sqlalchemy.sql.operators.ColumnOperators.__div__).

**\_\_reduce\_\_**()

*inherited from the* \_\_reduce\_\_() *method of* object

helper for pickle

**\_\_reduce\_ex\_\_**()

*inherited from the* \_\_reduce\_ex\_\_() *method of* object

helper for pickle

**\_\_repr\_\_**

*inherited from the* \_\_repr\_\_ *attribute of* object

**\_\_rmod\_\_**(*other*)

Implement the % operator in reverse.

See [ColumnOperators.\_\_mod\_\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.__mod__" \o "sqlalchemy.sql.operators.ColumnOperators.__mod__).

**\_\_rmul\_\_**(*other*)

Implement the \* operator in reverse.

See [ColumnOperators.\_\_mul\_\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.__mul__" \o "sqlalchemy.sql.operators.ColumnOperators.__mul__).

**\_\_rshift\_\_**(*other*)

implement the >> operator.

Not used by SQLAlchemy core, this is provided for custom operator systems which want to use >> as an extension point.

**\_\_rsub\_\_**(*other*)

Implement the - operator in reverse.

See [ColumnOperators.\_\_sub\_\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.__sub__" \o "sqlalchemy.sql.operators.ColumnOperators.__sub__).

**\_\_rtruediv\_\_**(*other*)

Implement the // operator in reverse.

See [ColumnOperators.\_\_truediv\_\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.__truediv__" \o "sqlalchemy.sql.operators.ColumnOperators.__truediv__).

**\_\_setattr\_\_**

*inherited from the* \_\_setattr\_\_ *attribute of* object

x.\_\_setattr\_\_('name', value) <==> x.name = value

**\_\_sizeof\_\_**() → int

*inherited from the* \_\_sizeof\_\_() *method of* object

size of object in memory, in bytes

**\_\_str\_\_**

*inherited from the* \_\_str\_\_ *attribute of* object

**\_\_sub\_\_**(*other*)

Implement the - operator.

In a column context, produces the clause a - b.

**\_\_subclasshook\_\_**()

*inherited from the* \_\_subclasshook\_\_() *method of* object

Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.\_\_subclasscheck\_\_(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

**\_\_truediv\_\_**(*other*)

Implement the // operator.

In a column context, produces the clause a / b.

**all\_**()

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\_**()

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**()

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*)

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*)

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*)

Implement the 'concat' operator.

In a column context, produces the clause a || b, or uses the concat() operator on MySQL.

**contains**(*other*, *\*\*kwargs*)

Implement the 'contains' operator.

In a column context, produces the clause LIKE '%<other>%'

**desc**()

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**()

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*)

Implement the 'endswith' operator.

In a column context, produces the clause LIKE '%<other>'

**ilike**(*other*, *escape=None*)

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*)

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*)

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*)

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".

实现IS DISTINCT FROM运算符。

在大多数平台上渲染“一个不同于B” 在一些如SQLite可能会渲染“a不是b”。

*New in version 1.1.*

**isnot**(*other*)

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*)

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*)

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*)

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*)

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*)

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*)

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**()

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**()

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. * ****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)

**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*)

Implement the startwith operator.

In a column context, produces the clause LIKE '<other>%'

**timetuple***= None*

Hack, allows datetime objects to be compared on the LHS.

*class*sqlalchemy.sql.base.**DialectKWArgs**

Establish the ability for a class to have dialect-specific arguments with defaults and constructor validation.

The [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs) interacts with the [DefaultDialect.construct\_arguments](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.default.DefaultDialect.construct_arguments" \o "sqlalchemy.engine.default.DefaultDialect.construct_arguments) present on a dialect.

**See also**

[DefaultDialect.construct\_arguments](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.default.DefaultDialect.construct_arguments" \o "sqlalchemy.engine.default.DefaultDialect.construct_arguments)

*classmethod***argument\_for**(*dialect\_name*, *argument\_name*, *default*)

Add a new kind of dialect-specific keyword argument for this class.

E.g.:

Index.argument\_for("mydialect", "length", **None**)

some\_index = Index('a', 'b', mydialect\_length=5)

The [DialectKWArgs.argument\_for()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.argument_for" \o "sqlalchemy.sql.base.DialectKWArgs.argument_for) method is a per-argument way adding extra arguments to the[DefaultDialect.construct\_arguments](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.default.DefaultDialect.construct_arguments" \o "sqlalchemy.engine.default.DefaultDialect.construct_arguments) dictionary. This dictionary provides a list of argument names accepted by various schema-level constructs on behalf of a dialect.

New dialects should typically specify this dictionary all at once as a data member of the dialect class. The use case for ad-hoc addition of argument names is typically for end-user code that is also using a custom compilation scheme which consumes the additional arguments.

|  |  |
| --- | --- |
| **Parameters:** | * ****dialect\_name**** – name of a dialect. The dialect must be locatable, else a [NoSuchModuleError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.NoSuchModuleError" \o "sqlalchemy.exc.NoSuchModuleError) is raised. The dialect must also include an existing[DefaultDialect.construct\_arguments](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.default.DefaultDialect.construct_arguments" \o "sqlalchemy.engine.default.DefaultDialect.construct_arguments) collection, indicating that it participates in the keyword-argument validation and default system, else [ArgumentError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.ArgumentError" \o "sqlalchemy.exc.ArgumentError) is raised. If the dialect does not include this collection, then any keyword argument can be specified on behalf of this dialect already. All dialects packaged within SQLAlchemy include this collection, however for third party dialects, support may vary. * ****argument\_name**** – name of the parameter. * ****default**** – default value of the parameter. |

*New in version 0.9.4.*

**dialect\_kwargs**

A collection of keyword arguments specified as dialect-specific options to this construct.

The arguments are present here in their original <dialect>\_<kwarg> format. Only arguments that were actually passed are included; unlike the [DialectKWArgs.dialect\_options](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_options" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_options) collection, which contains all options known by this dialect including defaults.

The collection is also writable; keys are accepted of the form <dialect>\_<kwarg> where the value will be assembled into the list of options.

*New in version 0.9.2.*

*Changed in version 0.9.4:*The [DialectKWArgs.dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs) collection is now writable.

**See also**

[DialectKWArgs.dialect\_options](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_options" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_options) - nested dictionary form

**dialect\_options**

A collection of keyword arguments specified as dialect-specific options to this construct.

This is a two-level nested registry, keyed to <dialect\_name> and <argument\_name>. For example, the postgresql\_where argument would be locatable as:

arg = my\_object.dialect\_options['postgresql']['where']

*New in version 0.9.2.*

**See also**

[DialectKWArgs.dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs) - flat dictionary form

**kwargs**

A synonym for [DialectKWArgs.dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs).

*class*sqlalchemy.sql.expression.**Extract**(*field*, *expr*, *\*\*kwargs*)

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

Represent a SQL EXTRACT clause, extract(field FROM expr).

**\_\_init\_\_**(*field*, *expr*, *\*\*kwargs*)

Construct a new [Extract](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.Extract" \o "sqlalchemy.sql.expression.Extract) object.

This constructor is mirrored as a public API function; see [extract()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.extract" \o "sqlalchemy.sql.expression.extract) for a full usage and argument description.

*class*sqlalchemy.sql.elements.**False\_**

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

Represent the false keyword, or equivalent, in a SQL statement.

[False\_](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.elements.False_" \o "sqlalchemy.sql.elements.False_) is accessed as a constant via the [false()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.false" \o "sqlalchemy.sql.expression.false) function.

*class*sqlalchemy.sql.expression.**FunctionFilter**(*func*, *\*criterion*)

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

Represent a function FILTER clause.

This is a special operator against aggregate and window functions, which controls which rows are passed to it. It's supported only by certain database backends.

Invocation of [FunctionFilter](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.FunctionFilter" \o "sqlalchemy.sql.expression.FunctionFilter) is via [FunctionElement.filter()](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.FunctionElement.filter" \o "sqlalchemy.sql.functions.FunctionElement.filter):

func.count(1).filter(**True**)

*New in version 1.0.0.*

**See also**

[FunctionElement.filter()](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.FunctionElement.filter" \o "sqlalchemy.sql.functions.FunctionElement.filter)

**\_\_init\_\_**(*func*, *\*criterion*)

Construct a new [FunctionFilter](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.FunctionFilter" \o "sqlalchemy.sql.expression.FunctionFilter) object.

This constructor is mirrored as a public API function; see [funcfilter()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.funcfilter" \o "sqlalchemy.sql.expression.funcfilter) for a full usage and argument description.

**filter**(*\*criterion*)

Produce an additional FILTER against the function.

This method adds additional criteria to the initial criteria set up by [FunctionElement.filter()](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.FunctionElement.filter" \o "sqlalchemy.sql.functions.FunctionElement.filter).

Multiple criteria are joined together at SQL render time via AND.

**over**(*partition\_by=None*, *order\_by=None*)

Produce an OVER clause against this filtered function.

Used against aggregate or so-called "window" functions, for database backends that support window functions.

The expression:

func.rank().filter(MyClass.y > 5).over(order\_by='x')

is shorthand for:

**from** **sqlalchemy** **import** over, funcfilterover(funcfilter(func.rank(), MyClass.y > 5), order\_by='x')

See [over()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.over" \o "sqlalchemy.sql.expression.over) for a full description.

*class*sqlalchemy.sql.expression.**Label**(*name*, *element*, *type\_=None*)

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

Represents a column label (AS).

Represent a label, as typically applied to any column-level element using the AS sql keyword.

**\_\_init\_\_**(*name*, *element*, *type\_=None*)

Construct a new [Label](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.Label" \o "sqlalchemy.sql.expression.Label) object.

This constructor is mirrored as a public API function; see [label()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.label" \o "sqlalchemy.sql.expression.label) for a full usage and argument description.

*class*sqlalchemy.sql.elements.**Null**

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

Represent the NULL keyword in a SQL statement.

[Null](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.elements.Null" \o "sqlalchemy.sql.elements.Null) is accessed as a constant via the [null()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.null" \o "sqlalchemy.sql.expression.null) function.

*class*sqlalchemy.sql.expression.**Over**(*element*, *partition\_by=None*, *order\_by=None*, *range\_=None*, *rows=None*)

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

Represent an OVER clause.

This is a special operator against a so-called "window" function, as well as any aggregate function, which produces results relative to the result set itself. It's supported only by certain database backends.

**\_\_init\_\_**(*element*, *partition\_by=None*, *order\_by=None*, *range\_=None*, *rows=None*)

Construct a new [Over](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.Over" \o "sqlalchemy.sql.expression.Over) object.

This constructor is mirrored as a public API function; see [over()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.over" \o "sqlalchemy.sql.expression.over) for a full usage and argument description.

**func**

the element referred to by this [Over](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.Over" \o "sqlalchemy.sql.expression.Over) clause.

*Deprecated since version 1.1:*the func element has been renamed to .element. The two attributes are synonymous though .func is read-only.

*class*sqlalchemy.sql.expression.**TextClause**(*text*, *bind=None*)

Bases: [sqlalchemy.sql.expression.Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable), [sqlalchemy.sql.expression.ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Represent a literal SQL text fragment.

E.g.:

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

t = text("SELECT \* FROM users")

result = connection.execute(t)

The [Text](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Text" \o "sqlalchemy.types.Text) construct is produced using the [text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text) function; see that function for full documentation.

**See also**

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

**bindparams**(*\*binds*, *\*\*names\_to\_values*)

Establish the values and/or types of bound parameters within this [TextClause](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.TextClause" \o "sqlalchemy.sql.expression.TextClause) construct.

Given a text construct such as:

**from** **sqlalchemy** **import** textstmt = text("SELECT id, name FROM user WHERE name=:name "

"AND timestamp=:timestamp")

the [TextClause.bindparams()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.TextClause.bindparams" \o "sqlalchemy.sql.expression.TextClause.bindparams) method can be used to establish the initial value of :name and :timestamp, using simple keyword arguments:

stmt = stmt.bindparams(name='jack',

timestamp=datetime.datetime(2012, 10, 8, 15, 12, 5))

Where above, new [BindParameter](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.BindParameter" \o "sqlalchemy.sql.expression.BindParameter) objects will be generated with the names name and timestamp, and values of jack and datetime.datetime(2012, 10, 8, 15, 12, 5), respectively. The types will be inferred from the values given, in this case [String](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.String" \o "sqlalchemy.types.String) and[DateTime](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.DateTime" \o "sqlalchemy.types.DateTime).

When specific typing behavior is needed, the positional \*binds argument can be used in which to specify [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) constructs directly. These constructs must include at least the key argument, then an optional value and type:

**from** **sqlalchemy** **import** bindparamstmt = stmt.bindparams(

bindparam('name', value='jack', type\_=String),

bindparam('timestamp', type\_=DateTime)

)

Above, we specified the type of [DateTime](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.DateTime" \o "sqlalchemy.types.DateTime) for the timestamp bind, and the type of [String](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.String" \o "sqlalchemy.types.String) for the name bind. In the case of name we also set the default value of "jack".

Additional bound parameters can be supplied at statement execution time, e.g.:

result = connection.execute(stmt,

timestamp=datetime.datetime(2012, 10, 8, 15, 12, 5))

The [TextClause.bindparams()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.TextClause.bindparams" \o "sqlalchemy.sql.expression.TextClause.bindparams) method can be called repeatedly, where it will re-use existing [BindParameter](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.BindParameter" \o "sqlalchemy.sql.expression.BindParameter) objects to add new information. For example, we can call [TextClause.bindparams()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.TextClause.bindparams" \o "sqlalchemy.sql.expression.TextClause.bindparams) first with typing information, and a second time with value information, and it will be combined:

stmt = text("SELECT id, name FROM user WHERE name=:name "

"AND timestamp=:timestamp")stmt = stmt.bindparams(

bindparam('name', type\_=String),

bindparam('timestamp', type\_=DateTime))stmt = stmt.bindparams(

name='jack',

timestamp=datetime.datetime(2012, 10, 8, 15, 12, 5))

*New in version 0.9.0:*The [TextClause.bindparams()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.TextClause.bindparams" \o "sqlalchemy.sql.expression.TextClause.bindparams) method supersedes the argument bindparams passed to [text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text).

**columns**(*\*cols*, *\*\*types*)

Turn this [TextClause](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.TextClause" \o "sqlalchemy.sql.expression.TextClause) object into a [TextAsFrom](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TextAsFrom" \o "sqlalchemy.sql.expression.TextAsFrom) object that can be embedded into another statement.

This function essentially bridges the gap between an entirely textual SELECT statement and the SQL expression language concept of a "selectable":

**from** **sqlalchemy.sql** **import** column, text

stmt = text("SELECT id, name FROM some\_table")stmt = stmt.columns(column('id'), column('name')).alias('st')

stmt = select([mytable]). select\_from(

mytable.join(stmt, mytable.c.name == stmt.c.name)

).where(stmt.c.id > 5)

Above, we pass a series of [column()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.column" \o "sqlalchemy.sql.expression.column) elements 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 positionally. These [column()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.column" \o "sqlalchemy.sql.expression.column) elements now become first class elements upon the [TextAsFrom.c](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TextAsFrom.c" \o "sqlalchemy.sql.expression.TextAsFrom.c) column collection, just like any other selectable.

以上，我们将一系列的column() 元素定位到TextClause.columns() 方法。 这些column() 元素现在成为TextAsFrom.c列集合的第一类元素，就像任何其他可选择的。

The column expressions we pass to [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) may also be typed; when we do so, these [TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine) objects become the effective return type of the column, so that SQLAlchemy's result-set-processing systems may be used on the return values. This is often needed for types such as date or boolean types, as well as for unicode processing on some dialect configurations:

我们传递给TextClause.columns() 的列表达式也可以键入; 当我们这样做时，这些TypeEngine对象成为列的有效返回类型，因此可以在返回值上使用SQLAlchemy的结果集处理系统。 对于诸如日期或布尔类型的类型，以及对某些方言配置的Unicode编码处理，通常需要这样做：

stmt = text("SELECT id, name, timestamp FROM some\_table")stmt = stmt.columns(

column('id', Integer),

column('name', Unicode),

column('timestamp', DateTime)

)

**for** id, name, timestamp **in** connection.execute(stmt):

print(id, name, timestamp)

As a shortcut to the above syntax, keyword arguments referring to types alone may be used, if only type conversion is needed:

作为上述语法的捷径，可以使用仅引用类型转换的关键字参数，

stmt = text("SELECT id, name, timestamp FROM some\_table")stmt = stmt.columns(

id=Integer,

name=Unicode,

timestamp=DateTime

)

**for** id, name, timestamp **in** connection.execute(stmt):

print(id, name, timestamp)

The positional form of [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) also provides the unique feature of ****positional column targeting****, which is particularly useful when using the ORM with complex textual queries. If we specify the columns from our model to [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), the result set will match to those columns positionally, meaning the name or origin of the column in the textual SQL doesn't matter:

[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) 的位置形式还提供了定位列定位的独特功能，这在使用复杂的文本查询的ORM时特别有用。 如果我们将模型中的列指定为[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) ，则结果集将以这些位置的方式匹配，这意味着文本SQL中列的名称或原点并不重要：

stmt = text("SELECT users.id, addresses.id, users.id, "

"users.name, addresses.email\_address AS email "

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

"WHERE users.id = 1").columns(

User.id,

Address.id,

Address.user\_id,

User.name,

Address.email\_address

)

query = session.query(User).from\_statement(stmt).options(

contains\_eager(User.addresses))

*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 offers positional column targeting in the result set when the column expressions are passed purely positionally.

版本1.1中的新功能：当列表达式纯粹通过位置传递时，TextClause.columns() 方法现在会在结果集中提供位置列定位。

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 provides a direct route to calling [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) as well as [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) against a textual SELECT statement:

[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) 方法提供了一种直接路由来调用[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) 以及[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) 对文本的SELECT语句：

stmt = stmt.columns(id=Integer, name=String).cte('st')

stmt = select([sometable]).where(sometable.c.id == stmt.c.id)

*New in version 0.9.0:*[text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text) can now be converted into a fully featured "selectable" construct using 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. This method supersedes the typemap argument to [text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text).

0.9.0版本中的新功能现在可以使用[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) 方法将[text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text) 转换为功能齐全的“可选”结构。 此方法取代了[text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text) 的typemap参数。

*class*sqlalchemy.sql.expression.**Tuple**(*\*clauses*, *\*\*kw*)

Bases: [sqlalchemy.sql.expression.ClauseList](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseList" \o "sqlalchemy.sql.expression.ClauseList), [sqlalchemy.sql.expression.ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement)

Represent a SQL tuple.

代表SQL元组。

**\_\_init\_\_**(*\*clauses*, *\*\*kw*)

Construct a new [Tuple](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.Tuple" \o "sqlalchemy.sql.expression.Tuple) object.

This constructor is mirrored as a public API function; see [tuple\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.tuple_" \o "sqlalchemy.sql.expression.tuple_) 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tuple\_() 以获取完整的用法和参数说明。

*class*sqlalchemy.sql.expression.**WithinGroup**(*element*, *\*order\_by*)

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

Represent a WITHIN GROUP (ORDER BY) clause.

代表一个WITHIN GROUP（ORDER BY）子句。

This is a special operator against so-called so-called "ordered set aggregate" and "hypothetical set aggregate" functions, including percentile\_cont(),rank(), dense\_rank(), etc.

这是一个特殊的操作符，可以使用所谓的“有序集合”和“假设集合”函数，包括percentile\_cont() ，rank() ，dense\_rank() 等）。

It's supported only by certain database backends, such as PostgreSQL, Oracle and MS SQL Server.

它仅由某些数据库后端支持，如PostgreSQL，Oracle和MS SQL Server。

The [WithinGroup](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.WithinGroup" \o "sqlalchemy.sql.expression.WithinGroup) construct extracts its type from the method [FunctionElement.within\_group\_type()](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.FunctionElement.within_group_type" \o "sqlalchemy.sql.functions.FunctionElement.within_group_type). If this returns None, the function's .typeis used.

WithinGroup构造从FunctionElement.within\_group\_type() 方法中提取其类型。 如果这返回None，则使用该函数的.typeis。

**\_\_init\_\_**(*element*, *\*order\_by*)

Construct a new [WithinGroup](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.WithinGroup" \o "sqlalchemy.sql.expression.WithinGroup) object.

This constructor is mirrored as a public API function; see [within\_group()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.within_group" \o "sqlalchemy.sql.expression.within_group) for a full usage and argument description.

该构造函数作为公共API函数进行镜像; 有关完整的用法和参数描述，请参阅within\_group() 。

**over**(*partition\_by=None*, *order\_by=None*)

Produce an OVER clause against this [WithinGroup](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.WithinGroup" \o "sqlalchemy.sql.expression.WithinGroup) construct.

This function has the same signature as that of [FunctionElement.over()](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.FunctionElement.over" \o "sqlalchemy.sql.functions.FunctionElement.over).

针对此WithinGroup结构生成OVER子句。

该函数具有与FunctionElement.over() 相同的签名。

*class*sqlalchemy.sql.elements.**True\_**

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

Represent the true keyword, or equivalent, in a SQL statement.

在SQL语句中表示true关键字或等效项。

[True\_](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.elements.True_" \o "sqlalchemy.sql.elements.True_) is accessed as a constant via the [true()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.true" \o "sqlalchemy.sql.expression.true) function.

True通过true() 函数作为常量访问。

*class*sqlalchemy.sql.expression.**TypeCoerce**(*expression*, *type\_*)

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

Represent a Python-side type-coercion wrapper.

代表一个Python侧的强制包装。

[TypeCoerce](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.TypeCoerce" \o "sqlalchemy.sql.expression.TypeCoerce) supplies the [expression.type\_coerce()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.type_coerce" \o "sqlalchemy.sql.expression.type_coerce) function; see that function for usage details.

TypeCoerce提供expression.type\_coerce() 函数; 请参阅该功能的使用细节。

*Changed in version 1.1:*The [type\_coerce()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.type_coerce" \o "sqlalchemy.sql.expression.type_coerce) function now produces a persistent [TypeCoerce](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.TypeCoerce" \o "sqlalchemy.sql.expression.TypeCoerce) wrapper object rather than translating the given object in place.

在版本1.1中更改：type\_coerce() 函数现在生成一个持久的TypeCoerce包装器对象，而不是将给定对象翻译到位。

**See also**

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

**\_\_init\_\_**(*expression*, *type\_*)

Construct a new [TypeCoerce](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.TypeCoerce" \o "sqlalchemy.sql.expression.TypeCoerce) object.

This constructor is mirrored as a public API function; see [type\_coerce()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.type_coerce" \o "sqlalchemy.sql.expression.type_coerce) for a full usage and argument description.

该构造函数作为公共API函数进行镜像; 有关完整的用法和参数描述，请参阅type\_coerce() 。

*class*sqlalchemy.sql.operators.**custom\_op**(*opstring*, *precedence=0*, *is\_comparison=False*, *natural\_self\_precedent=False*, *eager\_grouping=False*)

Represent a 'custom' operator.

表示“自定义”运算符。

[custom\_op](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.custom_op" \o "sqlalchemy.sql.operators.custom_op) is normally instantitated when the [ColumnOperators.op()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.op" \o "sqlalchemy.sql.operators.ColumnOperators.op) method is used to create a custom operator callable. The class can also be used directly when programmatically constructing expressions. E.g. to represent the "factorial" operation:

当使用ColumnOperators.op() 方法创建自定义操作符callable时，custom\_op通常被实例化。 该类也可以直接使用编程方式构造表达式。 例如。 代表“阶乘”操作：

**from** **sqlalchemy.sql** **import** UnaryExpression

**from** **sqlalchemy.sql** **import** operators

**from** **sqlalchemy** **import** Numeric

unary = UnaryExpression(table.c.somecolumn,

modifier=operators.custom\_op("!"),

type\_=Numeric)

*class*sqlalchemy.sql.operators.**Operators**

Base of comparison and logical operators.

比较基础和逻辑运算符。

Implements base methods [operate()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.Operators.operate" \o "sqlalchemy.sql.operators.Operators.operate) and [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), as well as [\_\_and\_\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.Operators.__and__" \o "sqlalchemy.sql.operators.Operators.__and__), [\_\_or\_\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.Operators.__or__" \o "sqlalchemy.sql.operators.Operators.__or__), [\_\_invert\_\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.Operators.__invert__" \o "sqlalchemy.sql.operators.Operators.__invert__).

实现基本方法operate() 和reverse\_operate() ，以及\_\_and \_\_() ，\_\_or \_\_() ，\_\_invert \_\_() 。

Usually is used via its most common subclass [ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators).

通常通过其最常见的子类ColumnOperators使用。

**\_\_and\_\_**(*other*)

Implement the & operator.

When used with SQL expressions, results in an AND operation, equivalent to [and\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.and_" \o "sqlalchemy.sql.expression.and_), that is:

当与SQL表达式一起使用时，会导致AND操作，等效于and\_() ，即：

a & b

is equivalent to:

**from** **sqlalchemy** **import** and\_and\_(a, b)

Care should be taken when using & regarding operator precedence; the & operator has the highest precedence. The operands should be enclosed in parenthesis if they contain further sub expressions:

使用＆关于运算符优先时应注意; ＆运算符具有最高优先级。 操作数应包含在括号中，如果它们包含进一步的子表达式：

(a == 2) & (b == 4)

**\_\_invert\_\_**()

Implement the ~ operator.

实现〜运算符。

When used with SQL expressions, results in a NOT operation, equivalent to [not\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.not_" \o "sqlalchemy.sql.expression.not_), that is:

当与SQL表达式一起使用时，会导致NOT操作，相当于not\_() ，即：

~a

is equivalent to:

**from** **sqlalchemy** **import** not\_not\_(a)

**\_\_or\_\_**(*other*)

Implement the | operator.

实施|运营商。

When used with SQL expressions, results in an OR operation, equivalent to [or\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.or_" \o "sqlalchemy.sql.expression.or_), that is:

当与SQL表达式一起使用时，会导致OR操作，等效于or\_() ，即：

a | b

is equivalent to:

**from** **sqlalchemy** **import** or\_or\_(a, b)

Care should be taken when using | regarding operator precedence; the | operator has the highest precedence. The operands should be enclosed in parenthesis if they contain further sub expressions:

使用|时请注意 关于运营商优先;该| 运营商具有最高优先级。 操作数应包含在括号中，如果它们包含进一步的子表达式：

(a == 2) | (b == 4)

**op**(*opstring*, *precedence=0*, *is\_comparison=False*)

produce a generic operator function.

产生通用的运算符函数。

e.g.:

somecolumn.op("\*")(5)

produces:

somecolumn \* 5

This function can also be used to make bitwise operators explicit. For example:

此功能也可用于使按位运算符显式。 例如：

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)

**operate**(*op*, *\*other*, *\*\*kwargs*)

Operate on an argument.

操作参数。

This is the lowest level of operation, raises NotImplementedError by default.

这是最低级别的操作，默认情况下会引发NotImplementedError。

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:

在子类上覆盖它可以允许将常见行为应用于所有操作。 例如，覆盖ColumnOperators以将func.lower() 应用于左侧和右侧：

**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*)

Reverse operate on an argument.

Usage is the same as operate().

*class*sqlalchemy.sql.elements.**quoted\_name**

Bases: sqlalchemy.util.langhelpers.MemoizedSlots, \_\_builtin\_\_.unicode

Represent a SQL identifier combined with quoting preferences.

表示SQL标识符和引用偏好设置。

[quoted\_name](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.elements.quoted_name" \o "sqlalchemy.sql.elements.quoted_name) is a Python unicode/str subclass which represents a particular identifier name along with a quote flag. This quote flag, when set to True or False, overrides automatic quoting behavior for this identifier in order to either unconditionally quote or to not quote the name. If left at its default of None, quoting behavior is applied to the identifier on a per-backend basis based on an examination of the token itself.

quoted\_name是一个Python unicode / str子类，它表示一个特定的标识符名称以及一个引号标志。此引号标志设置为True或False时，会覆盖此标识符的自动引用行为，以便无条件引用或不引用该名称。如果默认为“否”，则基于令牌本身的检查，每个后端将引用行为应用于该标识符。

A [quoted\_name](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.elements.quoted_name" \o "sqlalchemy.sql.elements.quoted_name) object with quote=True is also prevented from being modified in the case of a so-called "name normalize" option. Certain database backends, such as Oracle, Firebird, and DB2 "normalize" case-insensitive names as uppercase. The SQLAlchemy dialects for these backends convert from SQLAlchemy's lower-case-means-insensitive convention to the upper-case-means-insensitive conventions of those backends. The quote=True flag here will prevent this conversion from occurring to support an identifier that's quoted as all lower case against such a backend.

在所谓的“名称规范化”选项的情况下，也可以防止在引号= True的quoted\_name对象被修改。某些数据库后端（如Oracle，Firebird和DB2）将大小写不区分大小写的名称“归一化”。这些后端的SQLAlchemy方言将从SQLAlchemy的小写意义不敏感的约定转换为这些后端的大写无关的约定。这里的quote = True标志将阻止这种转换发生，以支持引用为所有小写的标识符对此后台。

The [quoted\_name](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.elements.quoted_name" \o "sqlalchemy.sql.elements.quoted_name) object is normally created automatically when specifying the name for key schema constructs such as [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table), [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column), and others. The class can also be passed explicitly as the name to any function that receives a name which can be quoted. Such as to use the [Engine.has\_table()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.has_table" \o "sqlalchemy.engine.Engine.has_table) method with an unconditionally quoted name:

在指定关键模式构造的名称（如表，列等）时，通常会自动创建quoted\_name对象。该类也可以作为名称显式地传递给接收可引用的名称的任何函数。例如使用Engine.has\_table() 方法和无条件引用的名称：

**from** **sqlaclchemy** **import** create\_engine

**from** **sqlalchemy.sql.elements** **import** quoted\_name

engine = create\_engine("oracle+cx\_oracle://some\_dsn")

engine.has\_table(quoted\_name("some\_table", **True**))

The above logic will run the "has table" logic against the Oracle backend, passing the name exactly as "some\_table" without converting to upper case.

上述逻辑将对Oracle后端运行“has table”逻辑，将名称完全按“some\_table”传递，而不转换为大写。

*New in version 0.9.0.*

*class*sqlalchemy.sql.expression.**UnaryExpression**(*element*, *operator=None*, *modifier=None*, *type\_=None*, *negate=None*, *wraps\_column\_expression=False*)

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

Define a 'unary' expression.

定义“一元”表达。

A unary expression has a single column expression and an operator. The operator can be placed on the left (where it is called the 'operator') or right (where it is called the 'modifier') of the column expression.

一元表达式具有单列表达式和运算符。 操作符可以放置在列表达式的左侧（称为“操作符”）或右侧（称为“修饰符”））。

[UnaryExpression](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.UnaryExpression" \o "sqlalchemy.sql.expression.UnaryExpression) is the basis for several unary operators including those used by [desc()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.desc" \o "sqlalchemy.sql.expression.desc), [asc()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.asc" \o "sqlalchemy.sql.expression.asc), [distinct()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.distinct" \o "sqlalchemy.sql.expression.distinct), [nullsfirst()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.nullsfirst" \o "sqlalchemy.sql.expression.nullsfirst) and [nullslast()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.nullslast" \o "sqlalchemy.sql.expression.nullslast).

UnaryExpression是几个一元运算符的基础，包括desc() ，asc() ，distinct() ，nullsfirst() 和nullslast() 所使用的运算符。

**compare**(*other*, *\*\*kw*)

Compare this [UnaryExpression](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.UnaryExpression" \o "sqlalchemy.sql.expression.UnaryExpression) against the given [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement).

将此UnaryExpression与给定的ClauseElement进行比较。

## 2.2 Selectables, Tables, FROM objects

The term "selectable" refers to any object that rows can be selected from; in SQLAlchemy, these objects descend from [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) and their distinguishing feature is their [FromClause.c](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.c" \o "sqlalchemy.sql.expression.FromClause.c) attribute, which is a namespace of all the columns contained within the FROM clause (these elements are themselves [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement)subclasses).

术语“可选择”是指可以选择行的任何对象; 在SQLAlchemy中，这些对象来自FromClause，它们的区别特征是它们的FromClause.c属性，它是FROM子句中包含的所有列的名称空间（这些元素本身是ColumnElement子类）。

sqlalchemy.sql.expression.**alias**(*selectable*, *name=None*, *flat=False*)

Return an [Alias](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Alias" \o "sqlalchemy.sql.expression.Alias) object.

返回Alias对象。

An [Alias](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Alias" \o "sqlalchemy.sql.expression.Alias) represents any [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) with an alternate name assigned within SQL, typically using the AS clause when generated, e.g. SELECT \* FROMtable AS aliasname.

别名代表任何FromClause，并在SQL中分配一个备用名称，通常在生成时使用AS子句，例如。SELECT \* FROMtable AS aliasname的。

Similar functionality is available via the [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 available on all [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) subclasses.

所有FromClause子类都可以使用alias() 方法提供类似的功能。

When an [Alias](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Alias" \o "sqlalchemy.sql.expression.Alias) is created from a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object, this has the effect of the table being rendered as tablename AS aliasname in a SELECT statement.

当从一个表对象创建一个别名时，这样做会在SELECT语句中将该表的形式呈现为tablename AS aliasname。

For [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) objects, the effect is that of creating a named subquery, i.e. (select ...) AS aliasname.

对于select() 对象，效果是创建一个命名的子查询，即（选择...）AS别名。

The name parameter is optional, and provides the name to use in the rendered SQL. If blank, an "anonymous" name will be deterministically generated at compile time. Deterministic means the name is guaranteed to be unique against other constructs used in the same statement, and will also be the same name for each successive compilation of the same statement object.

name参数是可选的，并提供了在呈现的SQL中使用的名称。 如果为空，则在编译时将确定性地生成“匿名”名称。 确定性意味着该名称保证对于同一语句中使用的其他结构是唯一的，并且对于相同语句对象的每个连续编译也将是相同的名称。

|  |  |
| --- | --- |
| **Parameters:** | * ****selectable**** – any [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) subclass, such as a table, select statement, etc. * ****name**** – string name to be assigned as the alias. If None, a name will be deterministically generated at compile time. * ****flat –****Will be passed through to if the given selectable is an instance of [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) - see [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) for details.   *New in version 0.9.0.* |

sqlalchemy.sql.expression.**except\_**(*\*selects*, *\*\*kwargs*)

Return an EXCEPT of multiple selectables.

返回一个EXCEPT的多个可选项。

The returned object is an instance of [CompoundSelect](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.CompoundSelect" \o "sqlalchemy.sql.expression.CompoundSelect).

返回的对象是CompoundSelect的一个实例。

\*selects

a list of [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select) instances.

\*\*kwargs

available keyword arguments are the same as those of [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select).

sqlalchemy.sql.expression.**except\_all**(*\*selects*, *\*\*kwargs*)

Return an EXCEPT ALL of multiple selectables.

返回一个除了所有的多个可选项。

The returned object is an instance of [CompoundSelect](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.CompoundSelect" \o "sqlalchemy.sql.expression.CompoundSelect).

返回的对象是复合选择的一个实例。

\*selects

a list of [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select) instances.

\*\*kwargs

available keyword arguments are the same as those of [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select).

sqlalchemy.sql.expression.**exists**(*\*args*, *\*\*kwargs*)

Construct a new Exists against an existing [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select) object.

根据现有的Select对象构造新的Exists。

Calling styles are of the following forms:

呼叫样式有以下形式：

*# use on an existing select()*

s = select([table.c.col1]).where(table.c.col2==5)s = exists(s)

*# construct a select() at once*

exists(['\*'], \*\*select\_arguments).where(criterion)

*# columns argument is optional, generates "EXISTS (SELECT \*)"# by default.*

exists().where(table.c.col2==5)

sqlalchemy.sql.expression.**intersect**(*\*selects*, *\*\*kwargs*)

Return an INTERSECT of multiple selectables.

The returned object is an instance of [CompoundSelect](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.CompoundSelect" \o "sqlalchemy.sql.expression.CompoundSelect).

\*selects

a list of [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select) instances.

\*\*kwargs

available keyword arguments are the same as those of [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select).

sqlalchemy.sql.expression.**intersect\_all**(*\*selects*, *\*\*kwargs*)

Return an INTERSECT ALL of multiple selectables.

The returned object is an instance of [CompoundSelect](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.CompoundSelect" \o "sqlalchemy.sql.expression.CompoundSelect).

\*selects

a list of [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select) instances.

\*\*kwargs

available keyword arguments are the same as those of [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select).

sqlalchemy.sql.expression.**join**(*left*, *right*, *onclause=None*, *isouter=False*, *full=False*)

Produce a [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) object, given two [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) expressions.

E.g.:

j = join(user\_table, address\_table,

user\_table.c.id == address\_table.c.user\_id)stmt = select([user\_table]).select\_from(j)

would emit SQL along the lines of:

SELECT user.id, user.name FROM userJOIN address ON user.id = address.user\_id

Similar functionality is available given any [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) object (e.g. such as a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)) using the [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) method.

|  |  |
| --- | --- |
| **Parameters:** | * ****left**** – The left side of the join. * ****right**** – the right side of the join; this is any [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) object such as a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object, and may also be a selectable-compatible object such as an ORM-mapped class. * ****onclause**** – a SQL expression representing the ON clause of the join. If left at None, [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) will attempt to join the two tables based on a foreign key relationship. * ****isouter**** – if True, render a LEFT OUTER JOIN, instead of JOIN. * ****full –****if True, render a FULL OUTER JOIN, instead of JOIN.   *New in version 1.1.* |

**See also**

[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) - method form, based on a given left side

[Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) - the type of object produced

sqlalchemy.sql.expression.**lateral**(*selectable*, *name=None*)

Return a [Lateral](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Lateral" \o "sqlalchemy.sql.expression.Lateral) object.

返回一个横向对象。

[Lateral](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Lateral" \o "sqlalchemy.sql.expression.Lateral) is an [Alias](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Alias" \o "sqlalchemy.sql.expression.Alias) subclass that represents a subquery with the LATERAL keyword applied to it.

Lateral是一个Alias子类，表示使用了LATERAL关键字的子查询。

The special behavior of a LATERAL subquery is that it appears in the FROM clause of an enclosing SELECT, but may correlate to other FROM clauses of that SELECT. It is a special case of subquery only supported by a small number of backends, currently more recent PostgreSQL versions.

LATERAL子查询的特殊行为是它出现在封闭SELECT的FROM子句中，但可能与该SELECT的其他FROM子句相关。 这是一个特殊情况的子查询，只有少量的后端支持，目前更新的PostgreSQL版本。

*New in version 1.1.*

**See also**

[LATERAL correlation](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html" \l "lateral-selects) - overview of usage.

sqlalchemy.sql.expression.**outerjoin**(*left*, *right*, *onclause=None*, *full=False*)

Return an OUTER JOIN clause element.

返回一个OUTER JOIN子句元素。

The returned object is an instance of [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join).

返回的对象是Join的一个实例。

Similar functionality is also available via the [outerjoin()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.outerjoin" \o "sqlalchemy.sql.expression.FromClause.outerjoin) method on any [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

任何FromClause都可以通过outerjoin() 方法获得类似的功能。

|  |  |
| --- | --- |
| **Parameters:** | * ****left**** – The left side of the join. * ****right**** – The right side of the join. * ****onclause**** – Optional criterion for the ON clause, is derived from foreign key relationships established between left and right otherwise. |

To chain joins together, use the [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) or [FromClause.outerjoin()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.outerjoin" \o "sqlalchemy.sql.expression.FromClause.outerjoin) methods on the resulting [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) object.

要将连接链接在一起，请在生成的Join对象上使用From Clause.join() 或From Clause.outer join() 方法。

sqlalchemy.sql.expression.**select**(*columns=None*, *whereclause=None*, *from\_obj=None*, *distinct=False*, *having=None*, *correlate=True*, *prefixes=None*, *suffixes=None*, *\*\*kwargs*)

Construct a new [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select).

构建新的选择。

Similar functionality is also available via the [FromClause.select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.select" \o "sqlalchemy.sql.expression.FromClause.select) method on any [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

任何FromClause都可以通过FromClause.select() 方法获得类似的功能。

All arguments which accept [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement) arguments also accept string arguments, which will be converted as appropriate into either [text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text) or [literal\_column()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.literal_column" \o "sqlalchemy.sql.expression.literal_column) constructs.

接受ClauseElement参数的所有参数也接受字符串参数，这些参数将被适当转换为text() 或literal\_column() 结构。

**See also**

[Selecting](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html" \l "coretutorial-selecting) - Core Tutorial description of [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select).

|  |  |
| --- | --- |
| **Parameters:** | * ****columns –****A list of [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) or [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) objects which will form the columns clause of the resulting statement. For those objects that are instances of[FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) (typically [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) objects), the [FromClause.c](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.c" \o "sqlalchemy.sql.expression.FromClause.c) collection is extracted to form a collection of [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) objects.ColumnElement或FromClause对象的列表，它将形成结果语句的columns子句。 对于那些作为FromClause（通常为Table或Alias对象）的实例的对象，将提取FromClause.c集合以形成ColumnElement对象的集合。   This parameter will also accept [Text](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Text" \o "sqlalchemy.types.Text) constructs as given, as well as ORM-mapped classes.  此参数还将接受给定的文本结构以及ORM映射的类。  **Note**  The [select.columns](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableSample.select.params.columns" \o "sqlalchemy.sql.expression.TableSample.select) parameter is not available in the method form of [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select), e.g. [FromClause.select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.select" \o "sqlalchemy.sql.expression.FromClause.select).  **See also**  [Select.column()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.column" \o "sqlalchemy.sql.expression.Select.column)  [Select.with\_only\_columns()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.with_only_columns" \o "sqlalchemy.sql.expression.Select.with_only_columns)   * ****whereclause –****A [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement) expression which will be used to form the WHERE clause. It is typically preferable to add WHERE criterion to an existing [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select) using method chaining with [Select.where()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.where" \o "sqlalchemy.sql.expression.Select.where).将用于形成WHERE子句的ClauseElement表达式。 通常优选将WHERE标准添加到使用Select.where() 链接的现有Select选择方法。   **See also**  [Select.where()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.where" \o "sqlalchemy.sql.expression.Select.where)   * ****from\_obj –****A list of [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement) objects which will be added to the FROM clause of the resulting statement. This is equivalent to calling [Select.select\_from()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.select_from" \o "sqlalchemy.sql.expression.Select.select_from) using method chaining on an existing [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select) object.将添加到结果语句的FROM子句的ClauseElement对象的列表。 这相当于使用现有Select对象上的方法链接调用Select.select\_from() 。   **See also**  [Select.select\_from()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.select_from" \o "sqlalchemy.sql.expression.Select.select_from) - full description of explicit FROM clause specification.   * ****~~autocommit –~~****~~Deprecated. Use .execution\_options(autocommit=<True|False>) to set the autocommit option.~~   **~~See also~~**  ~~[Executable.execution\_options()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable.execution_options" \o "sqlalchemy.sql.expression.Executable.execution_options)~~   * ****bind=None**** – 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) instance to which the resulting [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select) object will be bound. The [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select) object will otherwise automatically bind to whatever Connectable instances can be located within its contained [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement) members.将生成的Select对象绑定到的引擎或连接实例。 否则，Select对象将自动绑定到其所包含的ClauseElement成员中的可连接实例。 * ****correlate=True –****indicates that this [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select) object should have its contained [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) elements "correlated" to an enclosing [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select) object. It is typically preferable to specify correlations on an existing [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select) construct using [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).表示此Select对象应具有包含的FromClause元素“相关”到封闭的Select对象。 通常最好使用Select.correlate() 对现有的Select构造指定相关性。   **See also**  [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) - full description of correlation.   * ****distinct=False –****when True, applies a DISTINCT qualifier to the columns clause of the resulting statement.当为True时，将DISTINCT限定符应用于生成语句的columns子句。   The boolean argument may also be a column expression or list of column expressions - this is a special calling form which is understood by the PostgreSQL dialect to render the DISTINCT ON (<columns>) syntax.  布尔参数也可以是列表达式或列表达式列表 - 这是一个特殊的调用形式，PostgreSQL方言可以理解为呈现DISTINCT ON（<columns>）语法。  distinct is also available on an existing [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select) object via the [distinct()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.distinct" \o "sqlalchemy.sql.expression.Select.distinct) method.  **See also**  [Select.distinct()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.distinct" \o "sqlalchemy.sql.expression.Select.distinct)   * ****for\_update=False –****~~when True, applies FOR UPDATE to the end of the resulting statement.~~   *~~Deprecated since version 0.9.0:~~*~~- use [Select.with\_for\_update()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.with_for_update" \o "sqlalchemy.sql.expression.Select.with_for_update) to specify the structure of the FOR UPDATE clause.~~  ~~for\_update accepts various string values interpreted by specific backends, including:~~   * + ~~"read" - on MySQL, translates to LOCK IN SHARE MODE; on PostgreSQL, translates to FOR SHARE.~~   + ~~"nowait" - on PostgreSQL and Oracle, translates to FOR UPDATE NOWAIT.~~   + ~~"read\_nowait" - on PostgreSQL, translates to FOR SHARE NOWAIT.~~   **~~See also~~**  ~~[Select.with\_for\_update()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.with_for_update" \o "sqlalchemy.sql.expression.Select.with_for_update) - improved API for specifying the FOR UPDATE clause.~~   * ****group\_by –****a list of [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement) objects which will comprise the GROUP BY clause of the resulting select. This parameter is typically specified more naturally using the [Select.group\_by()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.group_by" \o "sqlalchemy.sql.expression.Select.group_by) method on an existing [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select).一个ClauseElement对象的列表，它将包含所生成的select的GROUP BY子句。 此参数通常在现有Select上使用Select.group\_by() 方法更自然地指定。   **See also**  [Select.group\_by()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.group_by" \o "sqlalchemy.sql.expression.Select.group_by)   * ****having –****a [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement) that will comprise the HAVING clause of the resulting select when GROUP BY is used. This parameter is typically specified more naturally using the [Select.having()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.having" \o "sqlalchemy.sql.expression.Select.having) method on an existing [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select).当使用GROUP BY时，将包含结果选择的HAVING子句的条款元素。 通常在现有Select上使用Select.having() 方法更自然地指定此参数。   **See also**  [Select.having()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.having" \o "sqlalchemy.sql.expression.Select.having)   * ****limit=None –****a numerical value which usually renders as a LIMIT expression in the resulting select. Backends that don't support LIMIT will attempt to provide similar functionality. This parameter is typically specified more naturally using the [Select.limit()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.limit" \o "sqlalchemy.sql.expression.Select.limit) method on an existing [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select).通常在所得选择中呈现为LIMIT表达式的数值。 不支持LIMIT的后端将尝试提供类似的功能。 通常使用现有Select中的Select.limit() 方法更自然地指定此参数。   **See also**  [Select.limit()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.limit" \o "sqlalchemy.sql.expression.Select.limit)   * ****offset=None –****a numeric value which usually renders as an OFFSET expression in the resulting select. Backends that don't support OFFSET will attempt to provide similar functionality. This parameter is typically specified more naturally using the [Select.offset()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.offset" \o "sqlalchemy.sql.expression.Select.offset) method on an existing [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select).一个通常在结果选择中呈现为OFFSET表达式的数值。 不支持OFFSET的后端将尝试提供类似的功能。 通常使用现有Select中的Select.offset() 方法更自然地指定此参数。   **See also**  [Select.offset()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.offset" \o "sqlalchemy.sql.expression.Select.offset)   * ****order\_by –****a scalar or list of [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement) objects which will comprise the ORDER BY clause of the resulting select. This parameter is typically specified more naturally using the [Select.order\_by()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.order_by" \o "sqlalchemy.sql.expression.Select.order_by) method on an existing [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select).一个标量或ClauseElement对象的列表，它将包含所得到的选择的ORDER BY子句。 通常使用现有Select中的Select.order\_by() 方法更自然地指定此参数。   **See also**  [Select.order\_by()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.order_by" \o "sqlalchemy.sql.expression.Select.order_by)   * ****use\_labels=False –****when True, the statement will be generated using labels for each column in the columns clause, which qualify each column with its parent table's (or aliases) name so that name conflicts between columns in different tables don't occur. The format of the label is <tablename>\_<column>. The "c" collection of the resulting [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select) object will use these names as well for targeting column members.当为True时，将使用columns子句中每列的标签生成该语句，该列用于将每个列与其父表（或别名）的名称进行限定，以使不同表中的列之间的名称冲突。 标签的格式为<tablename> \_ <column>。 所产生的Select对象的“c”集合也将使用这些名称来定位列成员。   This parameter can also be specified on an existing [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select) object using the [Select.apply\_labels()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.apply_labels" \o "sqlalchemy.sql.expression.Select.apply_labels) method.  **See also**  [Select.apply\_labels()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.apply_labels" \o "sqlalchemy.sql.expression.Select.apply_labels) |

sqlalchemy.sql.expression.**subquery**(*alias*, *\*args*, *\*\*kwargs*)

Return an [Alias](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Alias" \o "sqlalchemy.sql.expression.Alias) object derived from a [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select).

name

alias name

\*args, \*\*kwargs

all other arguments are delivered to the [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) function.

sqlalchemy.sql.expression.**table**(*name*, *\*columns*)

Produce a new [TableClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableClause" \o "sqlalchemy.sql.expression.TableClause).

生成一个新的TableClause。

The object returned is an instance of [TableClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableClause" \o "sqlalchemy.sql.expression.TableClause), which represents the "syntactical" portion of the schema-level [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object. It may be used to construct lightweight table constructs.

返回的对象是TableClause的一个实例，它表示模式级Table对象的“语法”部分。 它可以用于构造轻量级表构造。

*Changed in version 1.0.0:*[expression.table()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.table" \o "sqlalchemy.sql.expression.table) can now be imported from the plain sqlalchemy namespace like any other SQL element.

|  |  |
| --- | --- |
| **Parameters:** | * ****name**** – Name of the table. * ****columns**** – A collection of [expression.column()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.column" \o "sqlalchemy.sql.expression.column) constructs. |

sqlalchemy.sql.expression.**tablesample**(*selectable*, *sampling*, *name=None*, *seed=None*)

Return a [TableSample](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableSample" \o "sqlalchemy.sql.expression.TableSample) object.

[TableSample](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableSample" \o "sqlalchemy.sql.expression.TableSample) is an [Alias](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Alias" \o "sqlalchemy.sql.expression.Alias) subclass that represents a table with the TABLESAMPLE clause applied to it. [tablesample()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.tablesample" \o "sqlalchemy.sql.expression.tablesample) is also available from the [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) class via the [FromClause.tablesample()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.tablesample" \o "sqlalchemy.sql.expression.FromClause.tablesample) method.

TableSample是一个Alias子类，用于表示应用了TABLESAMPLE子句的表。 也可以通过FromClause.tablesample() 方法从FromClause类获得tablesample() 。

The TABLESAMPLE clause allows selecting a randomly selected approximate percentage of rows from a table. It supports multiple sampling methods, most commonly BERNOULLI and SYSTEM.

TABLESAMPLE子句允许从表中选择一个随机选择的大约百分比行。 它支持多种采样方式，最常见的是BERNOULLI和SYSTEM。

e.g.:

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

selectable = people.tablesample(

func.bernoulli(1),

name='alias',

seed=func.random())stmt = select([selectable.c.people\_id])

Assuming people with a column people\_id, the above statement would render as:

SELECT alias.people\_id FROMpeople AS alias TABLESAMPLE bernoulli(:bernoulli\_1)REPEATABLE (random())

*New in version 1.1.*

|  |  |
| --- | --- |
| **Parameters:** | * ****sampling**** – a float percentage between 0 and 100 or [functions.Function](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.Function" \o "sqlalchemy.sql.functions.Function). * ****name**** – optional alias name * ****seed**** – any real-valued SQL expression. When specified, the REPEATABLE sub-clause is also rendered. |

sqlalchemy.sql.expression.**union**(*\*selects*, *\*\*kwargs*)

Return a UNION of multiple selectables.

返回一个UNION的多个可选项。

The returned object is an instance of [CompoundSelect](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.CompoundSelect" \o "sqlalchemy.sql.expression.CompoundSelect).

返回的对象是CompoundSelect的一个实例。

A similar [union()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.union" \o "sqlalchemy.sql.expression.union) method is available on all [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) subclasses.

所有FromClause子类都可以使用类似的union() 方法。

\*selects

a list of [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select) instances.

\*\*kwargs

available keyword arguments are the same as those of [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select).

sqlalchemy.sql.expression.**union\_all**(*\*selects*, *\*\*kwargs*)

Return a UNION ALL of multiple selectables.

返回多个可选项的UNION ALL。

The returned object is an instance of [CompoundSelect](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.CompoundSelect" \o "sqlalchemy.sql.expression.CompoundSelect).

返回的对象是复合选择的一个实例。

A similar [union\_all()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.union_all" \o "sqlalchemy.sql.expression.union_all) method is available on all [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) subclasses.

所有From Clause子类都可以使用类似的union\_all() 方法。

\*selects

a list of [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select) instances.

\*\*kwargs

available keyword arguments are the same as those of [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select).

*class*sqlalchemy.sql.expression.**Alias**(*selectable*, *name=None*)

Bases: [sqlalchemy.sql.expression.FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Represents an table or selectable alias (AS).

表示表或可选别名（AS）。

Represents an alias, as typically applied to any table or sub-select within a SQL statement using the AS keyword (or without the keyword on certain databases such as Oracle).

表示一个别名，通常应用于任何表，或使用AS关键字（或在某些数据库（如Oracle）上没有关键字）的SQL语句中进行子选择。

This object is constructed from the [alias()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.alias" \o "sqlalchemy.sql.expression.alias) module level function as well as 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 available on all [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) subclasses.

该对象由alias() 模块级别函数以及所有FromClause子类中可用的FromClause.alias() 方法构成。

**alias**(*name=None*, *flat=False*)

*inherited from the* [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 of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

return an alias of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

返回此FromClause的别名。

This is shorthand for calling:

这是调用的简写：

**from** **sqlalchemy** **import** alias

a = alias(self, name=name)

See [alias()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.alias" \o "sqlalchemy.sql.expression.alias) for details.

**c**

*inherited from the* [c](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.c" \o "sqlalchemy.sql.expression.FromClause.c) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

An alias for the [columns](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Alias.columns" \o "sqlalchemy.sql.expression.Alias.columns) attribute.

**columns**

*inherited from the* [columns](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.columns" \o "sqlalchemy.sql.expression.FromClause.columns) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

A named-based collection of [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) objects maintained by this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

由此FromClause维护的基于命名的ColumnElement对象集合。

The [columns](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Alias.columns" \o "sqlalchemy.sql.expression.Alias.columns), or [c](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Alias.c" \o "sqlalchemy.sql.expression.Alias.c) collection, is the gateway to the construction of SQL expressions using table-bound or other selectable-bound columns:

列或c集合是使用表格绑定或其他可选择绑定列构建SQL表达式的网关：

select([mytable]).where(mytable.c.somecolumn == 5)

**compare**(*other*, *\*\*kw*)

*inherited from the* [compare()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.compare" \o "sqlalchemy.sql.expression.ClauseElement.compare) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Compare this ClauseElement to the given ClauseElement.

将此条款元素与给定的条款元素进行比较。

Subclasses should override the default behavior, which is a straight identity comparison.

子类应该覆盖默认行为，这是一个直接的身份比较。

\*\*kw are arguments consumed by subclass compare() methods and may be used to modify the criteria for comparison. (see [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement))

\*\* kw是由subclass compare() 方法消耗的参数，可用于修改比较条件。（见ColumnElement）

**compile**(*bind=None*, *dialect=None*, *\*\*kw*)

*inherited from the* [compile()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.compile" \o "sqlalchemy.sql.expression.ClauseElement.compile) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Compile this SQL expression.

The return value is a [Compiled](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Compiled" \o "sqlalchemy.engine.interfaces.Compiled) object. Calling str() or unicode() on the returned value will yield a string representation of the result. The[Compiled](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Compiled" \o "sqlalchemy.engine.interfaces.Compiled) object also can return a dictionary of bind parameter names and values using the params accessor.

返回值是一个编译对象。 对返回的值调用str() 或unicode() 将产生结果的字符串表示形式。 Compiled对象还可以使用params访问器返回绑定参数名称和值的字典。

|  |  |
| --- | --- |
| **Parameters:** | * ****bind**** – An Engine or Connection from which a Compiled will be acquired. This argument takes precedence over this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)'s bound engine, if any.将从其获取编译的引擎或连接。 这个参数优先于这个ClauseElement的绑定引擎（如果有的话）。 * ****column\_keys**** – Used for INSERT and UPDATE statements, a list of column names which should be present in the VALUES clause of the compiled statement. If None, all columns from the target table object are rendered.用于INSERT和UPDATE语句，应在编译语句的VALUES子句中存在列名称列表。 如果为None，则会渲染目标表对象中的所有列。 * ****dialect**** – A Dialect instance from which a Compiled will be acquired. This argument takes precedence over the bind argument as well as this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)'s bound engine, if any.将从其中获取编译的方言实例。 此参数优先于bind参数以及此ClauseElement的绑定引擎（如果有）。 * ****inline**** – Used for INSERT statements, for a dialect which does not support inline retrieval of newly generated primary key columns, will force the expression used to create the new primary key value to be rendered inline within the INSERT statement's VALUES clause. This typically refers to Sequence execution but may also refer to any server-side default generation function associated with a primary key Column.用于INSERT语句，对于不支持新生成的主键列的内联检索的方言，将强制用于创建新主键值的表达式在INSERT语句的VALUES子句中内联呈现。 这通常是指Sequence执行，但也可以指与主键Column相关联的任何服务器端默认生成函数。 * ****compile\_kwargs –****optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the literal\_binds flag through:在所有“访问”方法中将附加参数的可选字典传递给编译器。 例如，这允许将任何自定义标志传递到自定义编译构造。 它也用于通过literal\_binds标志的情况：   **from** **sqlalchemy.sql** **import** table, column, select  t = table('t', column('x'))  s = select([t]).where(t.c.x == 5)  print s.compile(compile\_kwargs={"literal\_binds": **True**})  *New in version 0.9.0.* |

**See also**

[How do I render SQL expressions as strings, possibly with bound parameters inlined?](http://docs.sqlalchemy.org/en/rel_1_1/faq/sqlexpressions.html" \l "faq-sql-expression-string)

**correspond\_on\_equivalents**(*column*, *equivalents*)

*inherited from the* [correspond\_on\_equivalents()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.correspond_on_equivalents" \o "sqlalchemy.sql.expression.FromClause.correspond_on_equivalents) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return corresponding\_column for the given column, or if None search for a match in the given dictionary.

**corresponding\_column**(*column*, *require\_embedded=False*)

*inherited from the* [corresponding\_column()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.corresponding_column" \o "sqlalchemy.sql.expression.FromClause.corresponding_column) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Given a [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement), return the exported [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) object from this [Selectable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Selectable" \o "sqlalchemy.sql.expression.Selectable) which corresponds to that original [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) via a common ancestor column.

|  |  |
| --- | --- |
| **Parameters:** | * ****column**** – the target [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) to be matched * ****require\_embedded**** – only return corresponding columns for the given [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement), if the given [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) is actually present within a sub-element of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause). Normally the column will match if it merely shares a common ancestor with one of the exported columns of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause). |

**~~count~~**~~(~~*~~whereclause=None~~*~~,~~*~~\*\*params~~*~~)~~

*~~inherited from the~~*~~[count()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.count" \o "sqlalchemy.sql.expression.FromClause.count)~~*~~method of~~*~~[FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)~~

~~return a SELECT COUNT generated against this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).~~

*~~Deprecated since version 1.1:~~*~~FromClause.count() is deprecated. Counting rows requires that the correct column expression and accommodations for joins, DISTINCT, etc. must be made, otherwise results may not be what's expected. Please use an appropriate func.count() expression directly.~~

~~The function generates COUNT against the first column in the primary key of the table, or against the first column in the table overall. Explicit use offunc.count() should be preferred:~~

~~row\_count = conn.scalar(~~

~~select([func.count('\*')]).select\_from(table))~~

**~~See also~~**

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

**foreign\_keys**

*inherited from the* [foreign\_keys](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.foreign_keys" \o "sqlalchemy.sql.expression.FromClause.foreign_keys) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return the collection of ForeignKey objects which this FromClause references.

返回此FromClause引用的ForeignKey对象的集合。

**join**(*right*, *onclause=None*, *isouter=False*, *full=False*)

*inherited from the* [join()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.join" \o "sqlalchemy.sql.expression.FromClause.join) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) from this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) to another [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

E.g.:

**from** **sqlalchemy** **import** join

j = user\_table.join(address\_table,

user\_table.c.id == address\_table.c.user\_id)stmt = select([user\_table]).select\_from(j)

would emit SQL along the lines of:

SELECT user.id, user.name FROM userJOIN address ON user.id = address.user\_id

|  |  |
| --- | --- |
| **Parameters:** | * ****right**** – the right side of the join; this is any [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) object such as a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object, and may also be a selectable-compatible object such as an ORM-mapped class. * ****onclause**** – a SQL expression representing the ON clause of the join. If left at None, [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) will attempt to join the two tables based on a foreign key relationship. * ****isouter**** – if True, render a LEFT OUTER JOIN, instead of JOIN. * ****full –****if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN. Implies [FromClause.join.isouter](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.join.params.isouter" \o "sqlalchemy.sql.expression.FromClause.join).   *New in version 1.1.* |

**See also**

[join()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.join" \o "sqlalchemy.sql.expression.join) - standalone function

[Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) - the type of object produced

**lateral**(*name=None*)

*inherited from the* [lateral()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.lateral" \o "sqlalchemy.sql.expression.FromClause.lateral) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a LATERAL alias of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

The return value is the [Lateral](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Lateral" \o "sqlalchemy.sql.expression.Lateral) construct also provided by the top-level [lateral()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.lateral" \o "sqlalchemy.sql.expression.lateral) function.

*New in version 1.1.*

**See also**

[LATERAL correlation](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html" \l "lateral-selects) - overview of usage.

**outerjoin**(*right*, *onclause=None*, *full=False*)

*inherited from the* [outerjoin()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.outerjoin" \o "sqlalchemy.sql.expression.FromClause.outerjoin) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) from this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) to another [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause), with the "isouter" flag set to True.

E.g.:

**from** **sqlalchemy** **import** outerjoin

j = user\_table.outerjoin(address\_table,

user\_table.c.id == address\_table.c.user\_id)

The above is equivalent to:

j = user\_table.join(

address\_table,

user\_table.c.id == address\_table.c.user\_id,

isouter=**True**)

|  |  |
| --- | --- |
| **Parameters:** | * ****right**** – the right side of the join; this is any [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) object such as a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object, and may also be a selectable-compatible object such as an ORM-mapped class. * ****onclause**** – a SQL expression representing the ON clause of the join. If left at None, [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) will attempt to join the two tables based on a foreign key relationship. * ****full –****if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN.   *New in version 1.1.* |

**See also**

[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)

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

**params**(*\*optionaldict*, *\*\*kwargs*)

*inherited from the* [params()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.params" \o "sqlalchemy.sql.expression.ClauseElement.params) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Return a copy with [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) elements replaced.

Returns a copy of this ClauseElement with [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) elements replaced with values taken from the given dictionary:

**>>>** clause = column('x') + bindparam('foo')

**>>>** print clause.compile().params{'foo':None}

**>>>** print clause.params({'foo':7}).compile().params{'foo':7}

**primary\_key**

*inherited from the* [primary\_key](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.primary_key" \o "sqlalchemy.sql.expression.FromClause.primary_key) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return the collection of Column objects which comprise the primary key of this FromClause.

**replace\_selectable**(*old*, *alias*)

*inherited from the* [replace\_selectable()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.replace_selectable" \o "sqlalchemy.sql.expression.FromClause.replace_selectable) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

replace all occurrences of FromClause 'old' with the given Alias object, returning a copy of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

**select**(*whereclause=None*, *\*\*params*)

*inherited from the* [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.select" \o "sqlalchemy.sql.expression.FromClause.select) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

return a SELECT of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

**See also**

[select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) - general purpose method which allows for arbitrary column lists.

**tablesample**(*sampling*, *name=None*, *seed=None*)

*inherited from the* [tablesample()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.tablesample" \o "sqlalchemy.sql.expression.FromClause.tablesample) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a TABLESAMPLE alias of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

The return value is the [TableSample](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableSample" \o "sqlalchemy.sql.expression.TableSample) construct also provided by the top-level [tablesample()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.tablesample" \o "sqlalchemy.sql.expression.tablesample) function.

*New in version 1.1.*

**See also**

[tablesample()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.tablesample" \o "sqlalchemy.sql.expression.tablesample) - usage guidelines and parameters

**unique\_params**(*\*optionaldict*, *\*\*kwargs*)

*inherited from the* [unique\_params()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.unique_params" \o "sqlalchemy.sql.expression.ClauseElement.unique_params) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Return a copy with [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) elements replaced.

Same functionality as params(), except adds unique=True to affected bind parameters so that multiple statements can be used.

*class*sqlalchemy.sql.expression.**CompoundSelect**(*keyword*, *\*selects*, *\*\*kwargs*)

Bases: [sqlalchemy.sql.expression.GenerativeSelect](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect" \o "sqlalchemy.sql.expression.GenerativeSelect)

Forms the basis of UNION, UNION ALL, and other

SELECT-based set operations.

**See also**

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

[union\_all()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.union_all" \o "sqlalchemy.sql.expression.union_all)

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

[intersect\_all()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.intersect_all" \o "sqlalchemy.sql.expression.intersect_all)

except()

[except\_all()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.except_all" \o "sqlalchemy.sql.expression.except_all)

**alias**(*name=None*, *flat=False*)

*inherited from the* [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 of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

return an alias of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

This is shorthand for calling:

**from** **sqlalchemy** **import** aliasa = alias(self, name=name)

See [alias()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.alias" \o "sqlalchemy.sql.expression.alias) for details.

**append\_group\_by**(*\*clauses*)

*inherited from the* [append\_group\_by()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect.append_group_by" \o "sqlalchemy.sql.expression.GenerativeSelect.append_group_by) *method of* [GenerativeSelect](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect" \o "sqlalchemy.sql.expression.GenerativeSelect)

Append the given GROUP BY criterion applied to this selectable.

The criterion will be appended to any pre-existing GROUP BY criterion.

This is an ****in-place**** mutation method; the [group\_by()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect.group_by" \o "sqlalchemy.sql.expression.GenerativeSelect.group_by) method is preferred, as it provides standard [method chaining](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-method-chaining).

附加给该可选择的给定GROUP BY标准。

该标准将附加到任何预先存在的GROUP BY标准。

这是一种就地突变方法; group\_by() 方法是首选的，因为它提供了标准的方法链接。

**append\_order\_by**(*\*clauses*)

*inherited from the* [append\_order\_by()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect.append_order_by" \o "sqlalchemy.sql.expression.GenerativeSelect.append_order_by) *method of* [GenerativeSelect](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect" \o "sqlalchemy.sql.expression.GenerativeSelect)

Append the given ORDER BY criterion applied to this selectable.

The criterion will be appended to any pre-existing ORDER BY criterion.

This is an ****in-place**** mutation method; the [order\_by()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect.order_by" \o "sqlalchemy.sql.expression.GenerativeSelect.order_by) method is preferred, as it provides standard [method chaining](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-method-chaining).

附加适用于该可选择的给定ORDER BY标准。

该标准将附加到任何预先存在的ORDER BY标准。

这是一种就地突变方法; order\_by() 方法是首选的，因为它提供了标准的方法链接。

**apply\_labels**()

*inherited from the* [apply\_labels()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect.apply_labels" \o "sqlalchemy.sql.expression.GenerativeSelect.apply_labels) *method of* [GenerativeSelect](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect" \o "sqlalchemy.sql.expression.GenerativeSelect)

return a new selectable with the 'use\_labels' flag set to True.

返回一个新的可选项，将'use\_labels'标志设置为True。

This will result in column expressions being generated using labels against their table name, such as "SELECT somecolumn AS tablename\_somecolumn". This allows selectables which contain multiple FROM clauses to produce a unique set of column names regardless of name conflicts among the individual FROM clauses.

这将导致使用与其表名称相关的标签生成列表达式，例如"SELECT somecolumn AS tablename\_somecolumn"。 这允许包含多个FROM子句的可选项生成一组唯一的列名称，而不管各个FROM子句之间是否有冲突。

**as\_scalar**()

*inherited from the* [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) *method of* [SelectBase](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.SelectBase" \o "sqlalchemy.sql.expression.SelectBase)

return a 'scalar' representation of this selectable, which can be used as a column expression.

返回此可选择的"标量"表示，可用作列表达式。

Typically, a select statement which has only one column in its columns clause is eligible to be used as a scalar expression.

通常，在其column子句中只有一列的select语句有资格用作标量表达式。

The returned object is an instance of [ScalarSelect](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.ScalarSelect" \o "sqlalchemy.sql.expression.ScalarSelect).

返回的对象是ScalarSelect的一个实例。

**~~autocommit~~**~~()~~

*~~inherited from the~~*~~[autocommit()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.SelectBase.autocommit" \o "sqlalchemy.sql.expression.SelectBase.autocommit)~~*~~method of~~*~~[SelectBase](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.SelectBase" \o "sqlalchemy.sql.expression.SelectBase)~~

~~return a new selectable with the 'autocommit' flag set to True.~~

*~~Deprecated since version 0.6:~~*~~autocommit() is deprecated.~~ Use [Executable.execution\_options()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable.execution_options" \o "sqlalchemy.sql.expression.Executable.execution_options) with the 'autocommit' flag.

**c**

*inherited from the* [c](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.c" \o "sqlalchemy.sql.expression.FromClause.c) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

An alias for the [columns](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.CompoundSelect.columns" \o "sqlalchemy.sql.expression.CompoundSelect.columns) attribute.

**columns**

*inherited from the* [columns](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.columns" \o "sqlalchemy.sql.expression.FromClause.columns) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

A named-based collection of [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) objects maintained by this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

The [columns](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.CompoundSelect.columns" \o "sqlalchemy.sql.expression.CompoundSelect.columns), or [c](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.CompoundSelect.c" \o "sqlalchemy.sql.expression.CompoundSelect.c) collection, is the gateway to the construction of SQL expressions using table-bound or other selectable-bound columns:

select([mytable]).where(mytable.c.somecolumn == 5)

**compare**(*other*, *\*\*kw*)

*inherited from the* [compare()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.compare" \o "sqlalchemy.sql.expression.ClauseElement.compare) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Compare this ClauseElement to the given ClauseElement.

Subclasses should override the default behavior, which is a straight identity comparison.

\*\*kw are arguments consumed by subclass compare() methods and may be used to modify the criteria for comparison. (see [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement))

**compile**(*bind=None*, *dialect=None*, *\*\*kw*)

*inherited from the* [compile()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.compile" \o "sqlalchemy.sql.expression.ClauseElement.compile) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Compile this SQL expression.

The return value is a [Compiled](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Compiled" \o "sqlalchemy.engine.interfaces.Compiled) object. Calling str() or unicode() on the returned value will yield a string representation of the result. The[Compiled](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Compiled" \o "sqlalchemy.engine.interfaces.Compiled) object also can return a dictionary of bind parameter names and values using the params accessor.

|  |  |
| --- | --- |
| **Parameters:** | * ****bind**** – An Engine or Connection from which a Compiled will be acquired. This argument takes precedence over this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)'s bound engine, if any.将从其获取编译的引擎或连接。 这个参数优先于这个ClauseElement的绑定引擎（如果有的话）。 * ****column\_keys**** – Used for INSERT and UPDATE statements, a list of column names which should be present in the VALUES clause of the compiled statement. If None, all columns from the target table object are rendered.用于INSERT和UPDATE语句，应在编译语句的VALUES子句中存在列名称列表。 如果为None，则会渲染目标表对象中的所有列 * ****dialect**** – A Dialect instance from which a Compiled will be acquired. This argument takes precedence over the bind argument as well as this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)'s bound engine, if any.将从其中获取编译的方言实例。 此参数优先于bind参数以及此ClauseElement的绑定引擎（如果有）。 * ****inline**** – Used for INSERT statements, for a dialect which does not support inline retrieval of newly generated primary key columns, will force the expression used to create the new primary key value to be rendered inline within the INSERT statement's VALUES clause. This typically refers to Sequence execution but may also refer to any server-side default generation function associated with a primary key Column.用于INSERT语句，对于不支持新生成的主键列的内联检索的方言，将强制用于创建新主键值的表达式在INSERT语句的VALUES子句中内联呈现。 这通常是指Sequence执行，但也可以指与主键Column相关联的任何服务器端默认生成函数。 * ****compile\_kwargs –****optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the literal\_binds flag through:在所有"访问"方法中将附加参数的可选字典传递给编译器。 例如，这允许将任何自定义标志传递到自定义编译构造。 它也用于通过literal\_binds标志的情况：   **from** **sqlalchemy.sql** **import** table, column, select  t = table('t', column('x'))  s = select([t]).where(t.c.x == 5)  print s.compile(compile\_kwargs={"literal\_binds": **True**})  *New in version 0.9.0.* |

**See also**

[How do I render SQL expressions as strings, possibly with bound parameters inlined?](http://docs.sqlalchemy.org/en/rel_1_1/faq/sqlexpressions.html" \l "faq-sql-expression-string)

**correspond\_on\_equivalents**(*column*, *equivalents*)

*inherited from the* [correspond\_on\_equivalents()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.correspond_on_equivalents" \o "sqlalchemy.sql.expression.FromClause.correspond_on_equivalents) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return corresponding\_column for the given column, or if None search for a match in the given dictionary.

**corresponding\_column**(*column*, *require\_embedded=False*)

*inherited from the* [corresponding\_column()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.corresponding_column" \o "sqlalchemy.sql.expression.FromClause.corresponding_column) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Given a [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement), return the exported [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) object from this [Selectable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Selectable" \o "sqlalchemy.sql.expression.Selectable) which corresponds to that original [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) via a common ancestor column.

|  |  |
| --- | --- |
| **Parameters:** | * ****column**** – the target [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) to be matched * ****require\_embedded**** – only return corresponding columns for the given [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement), if the given [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) is actually present within a sub-element of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause). Normally the column will match if it merely shares a common ancestor with one of the exported columns of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause). |

**count**(*whereclause=None*, *\*\*params*)

*inherited from the* [count()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.count" \o "sqlalchemy.sql.expression.FromClause.count) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

return a SELECT COUNT generated against this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

*Deprecated since version 1.1:*FromClause.count() is deprecated. Counting rows requires that the correct column expression and accommodations for joins, DISTINCT, etc. must be made, otherwise results may not be what's expected. Please use an appropriate func.count() expression directly.

The function generates COUNT against the first column in the primary key of the table, or against the first column in the table overall. Explicit use offunc.count() should be preferred:

row\_count = conn.scalar(

select([func.count('\*')]).select\_from(table))

**See also**

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

**cte**(*name=None*, *recursive=False*)

*inherited from the* [cte()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.HasCTE.cte" \o "sqlalchemy.sql.expression.HasCTE.cte) *method of* [HasCTE](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.HasCTE" \o "sqlalchemy.sql.expression.HasCTE)

Return a new [CTE](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.CTE" \o "sqlalchemy.sql.expression.CTE), or Common Table Expression instance.

返回一个新的CTE或Common Table Expression实例。

Common table expressions are a SQL standard whereby SELECT statements can draw upon secondary statements specified along with the primary statement, using a clause called "WITH". Special semantics regarding UNION can also be employed to allow "recursive" queries, where a SELECT statement can draw upon the set of rows that have previously been selected.

公用表表达式是一种SQL标准，其中SELECT语句可以使用一个称为"WITH"的子句来使用主语句指定的辅助语句。 还可以使用有关UNION的特殊语义来允许"递归"查询，其中SELECT语句可以对先前已选择的行集合进行绘制。

CTEs can also be applied to DML constructs UPDATE, INSERT and DELETE on some databases, both as a source of CTE rows when combined with RETURNING, as well as a consumer of CTE rows.

CTE还可以应用于某些数据库上的DML构造UPDATE，INSERT和DELETE，作为与RETURNING组合的CTE行的源，以及CTE行的使用者。

SQLAlchemy detects [CTE](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.CTE" \o "sqlalchemy.sql.expression.CTE) objects, which are treated similarly to [Alias](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Alias" \o "sqlalchemy.sql.expression.Alias) objects, as special elements to be delivered to the FROM clause of the statement as well as to a WITH clause at the top of the statement.

SQLAlchemy检测到与Alias对象类似的CTE对象作为要传递到语句的FROM子句的特殊元素以及语句顶部的WITH子句。

*Changed in version 1.1:*Added support for UPDATE/INSERT/DELETE as CTE, CTEs added to UPDATE/INSERT/DELETE.

|  |  |
| --- | --- |
| **Parameters:** | * ****name**** – name given to the common table expression. Like \_FromClause.alias(), the name can be left as None in which case an anonymous symbol will be used at query compile time. * ****recursive**** – if True, will render WITH RECURSIVE. A recursive common table expression is intended to be used in conjunction with UNION ALL in order to derive rows from those already selected. |

The following examples include two from PostgreSQL's documentation at <http://www.postgresql.org/docs/current/static/queries-with.html>, as well as additional examples.

Example 1, non recursive:

**from** **sqlalchemy** **import** (Table, Column, String, Integer,

MetaData, select, func)

metadata = MetaData()

orders = Table('orders', metadata,

Column('region', String),

Column('amount', Integer),

Column('product', String),

Column('quantity', Integer))

regional\_sales = select([

orders.c.region,

func.sum(orders.c.amount).label('total\_sales')

]).group\_by(orders.c.region).cte("regional\_sales")

top\_regions = select([regional\_sales.c.region]).\

where(

regional\_sales.c.total\_sales >

select([

func.sum(regional\_sales.c.total\_sales)/10

])

).cte("top\_regions")

statement = select([

orders.c.region,

orders.c.product,

func.sum(orders.c.quantity).label("product\_units"),

func.sum(orders.c.amount).label("product\_sales")

]).where(orders.c.region.in\_(

select([top\_regions.c.region])

)).group\_by(orders.c.region, orders.c.product)

result = conn.execute(statement).fetchall()

Example 2, WITH RECURSIVE:

**from** **sqlalchemy** **import** (Table, Column, String, Integer,

MetaData, select, func)

metadata = MetaData()

parts = Table('parts', metadata,

Column('part', String),

Column('sub\_part', String),

Column('quantity', Integer),)

included\_parts = select([

parts.c.sub\_part,

parts.c.part,

parts.c.quantity]).\

where(parts.c.part=='our part').\

cte(recursive=**True**)

incl\_alias = included\_parts.alias()parts\_alias = parts.alias()included\_parts = included\_parts.union\_all(

select([

parts\_alias.c.sub\_part,

parts\_alias.c.part,

parts\_alias.c.quantity

]).

where(parts\_alias.c.part==incl\_alias.c.sub\_part))

statement = select([

included\_parts.c.sub\_part,

func.sum(included\_parts.c.quantity).

label('total\_quantity')

]).\

group\_by(included\_parts.c.sub\_part)

result = conn.execute(statement).fetchall()

Example 3, an upsert using UPDATE and INSERT with CTEs:

**from** **datetime** **import** date**from** **sqlalchemy** **import** (MetaData, Table, Column, Integer,

Date, select, literal, and\_, exists)

metadata = MetaData()

visitors = Table('visitors', metadata,

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

Column('date', Date, primary\_key=**True**),

Column('count', Integer),)

*# add 5 visitors for the product\_id == 1*product\_id = 1day = date.today()count = 5

update\_cte = (

visitors.update()

.where(and\_(visitors.c.product\_id == product\_id,

visitors.c.date == day))

.values(count=visitors.c.count + count)

.returning(literal(1))

.cte('update\_cte'))

upsert = visitors.insert().from\_select(

[visitors.c.product\_id, visitors.c.date, visitors.c.count],

select([literal(product\_id), literal(day), literal(count)])

.where(~exists(update\_cte.select())))

connection.execute(upsert)

**See also**

[orm.query.Query.cte()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.cte" \o "sqlalchemy.orm.query.Query.cte) - ORM version of [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).

**description**

*inherited from the* [description](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.description" \o "sqlalchemy.sql.expression.FromClause.description) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

a brief description of this FromClause.

Used primarily for error message formatting.

**execute**(*\*multiparams*, *\*\*params*)

*inherited from the* [execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable.execute" \o "sqlalchemy.sql.expression.Executable.execute) *method of* [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable)

Compile and execute this [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable).

**execution\_options**(*\*\*kw*)

*inherited from the* [execution\_options()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable.execution_options" \o "sqlalchemy.sql.expression.Executable.execution_options) *method of* [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable)

Set non-SQL options for the statement which take effect during execution.

为执行期间生效的语句设置非SQL选项。

Execution options can be set on a per-statement or per [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) basis. Additionally, the [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) and ORM [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) objects provide access to execution options which they in turn configure upon connections.

可以在每个语句或每个连接的基础上设置执行选项。 此外，引擎和ORM查询对象提供对执行选项的访问，它们在连接时依次配置。

The [execution\_options()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.CompoundSelect.execution_options" \o "sqlalchemy.sql.expression.CompoundSelect.execution_options) method is generative. A new instance of this statement is returned that contains the options:

execution\_options() 方法是生成的。 此语句的新实例将返回，其中包含以下选项：

statement = select([table.c.x, table.c.y])statement = statement.execution\_options(autocommit=**True**)

Note that only a subset of possible execution options can be applied to a statement - these include "autocommit" and "stream\_results", but not "isolation\_level" or "compiled\_cache". See [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) for a full list of possible options.

请注意，只有一个可能的执行选项的子集可以应用于一个语句 - 这些包括"autocommit"和"stream\_results"，而不是"isolation\_level"或"compiled\_cache"。 有关可能的选项的完整列表，请参阅Connection.execution\_options() 。

**See also**

[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)

[Query.execution\_options()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.execution_options" \o "sqlalchemy.orm.query.Query.execution_options)

**for\_update**

*inherited from the* [for\_update](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect.for_update" \o "sqlalchemy.sql.expression.GenerativeSelect.for_update) *attribute of* [GenerativeSelect](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect" \o "sqlalchemy.sql.expression.GenerativeSelect)

Provide legacy dialect support for the for\_update attribute.

**foreign\_keys**

*inherited from the* [foreign\_keys](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.foreign_keys" \o "sqlalchemy.sql.expression.FromClause.foreign_keys) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return the collection of ForeignKey objects which this FromClause references.

**group\_by**(*\*clauses*)

*inherited from the* [group\_by()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect.group_by" \o "sqlalchemy.sql.expression.GenerativeSelect.group_by) *method of* [GenerativeSelect](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect" \o "sqlalchemy.sql.expression.GenerativeSelect)

return a new selectable with the given list of GROUP BY criterion applied.

The criterion will be appended to any pre-existing GROUP BY criterion.

**join**(*right*, *onclause=None*, *isouter=False*, *full=False*)

*inherited from the* [join()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.join" \o "sqlalchemy.sql.expression.FromClause.join) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) from this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) to another [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

E.g.:

**from** **sqlalchemy** **import** join

j = user\_table.join(address\_table,

user\_table.c.id == address\_table.c.user\_id)stmt = select([user\_table]).select\_from(j)

would emit SQL along the lines of:

SELECT user.id, user.name FROM userJOIN address ON user.id = address.user\_id

|  |  |
| --- | --- |
| **Parameters:** | * ****right**** – the right side of the join; this is any [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) object such as a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object, and may also be a selectable-compatible object such as an ORM-mapped class.联合的右边; 这是任何FromClause对象，如Table对象，也可以是可选择兼容的对象，如ORM映射类。 * ****onclause**** – a SQL expression representing the ON clause of the join. If left at None, [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) will attempt to join the two tables based on a foreign key relationship.表示连接的ON子句的SQL表达式。 如果保持为None，则FromClause.join() 将根据外键关系尝试加入这两个表。 * ****isouter**** – if True, render a LEFT OUTER JOIN, instead of JOIN. * ****full –****if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN. Implies [FromClause.join.isouter](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.join.params.isouter" \o "sqlalchemy.sql.expression.FromClause.join).   *New in version 1.1.* |

**See also**

[join()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.join" \o "sqlalchemy.sql.expression.join) - standalone function

[Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) - the type of object produced

**label**(*name*)

*inherited from the* [label()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.SelectBase.label" \o "sqlalchemy.sql.expression.SelectBase.label) *method of* [SelectBase](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.SelectBase" \o "sqlalchemy.sql.expression.SelectBase)

return a 'scalar' representation of this selectable, embedded as a subquery with a label.返回此可选择的“标量”表示，作为具有标签的子查询嵌入。

**See also**

[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).

**lateral**(*name=None*)

*inherited from the* [lateral()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.lateral" \o "sqlalchemy.sql.expression.FromClause.lateral) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a LATERAL alias of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

返回此FromClause的LATERAL别名。

The return value is the [Lateral](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Lateral" \o "sqlalchemy.sql.expression.Lateral) construct also provided by the top-level [lateral()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.lateral" \o "sqlalchemy.sql.expression.lateral) function.

返回值是由顶级横向() 函数提供的横向结构。

*New in version 1.1.*

**See also**

[LATERAL correlation](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html" \l "lateral-selects) - overview of usage.

**limit**(*limit*)

*inherited from the* [limit()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect.limit" \o "sqlalchemy.sql.expression.GenerativeSelect.limit) *method of* [GenerativeSelect](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect" \o "sqlalchemy.sql.expression.GenerativeSelect)

return a new selectable with the given LIMIT criterion applied.

使用给定的LIMIT标准返回新的可选择。

This is a numerical value which usually renders as a LIMIT expression in the resulting select. Backends that don't support LIMIT will attempt to provide similar functionality.

这是一个数值，通常在结果选择中呈现为一个LIMIT表达式。 不支持LIMIT的后端将尝试提供类似的功能。

*Changed in version 1.0.0:*- [Select.limit()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.limit" \o "sqlalchemy.sql.expression.Select.limit) can now accept arbitrary SQL expressions as well as integer values.

|  |  |
| --- | --- |
| **Parameters:** | ****limit**** – an integer LIMIT parameter, or a SQL expression that provides an integer result. |

**offset**(*offset*)

*inherited from the* [offset()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect.offset" \o "sqlalchemy.sql.expression.GenerativeSelect.offset) *method of* [GenerativeSelect](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect" \o "sqlalchemy.sql.expression.GenerativeSelect)

return a new selectable with the given OFFSET criterion applied.

使用给定的OFFSET标准返回新的可选择。

This is a numeric value which usually renders as an OFFSET expression in the resulting select. Backends that don't support OFFSET will attempt to provide similar functionality.

这是一个数值，通常在结果选择中呈现为OFFSET表达式。 不支持OFFSET的后端将尝试提供类似的功能。

*Changed in version 1.0.0:*- [Select.offset()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.offset" \o "sqlalchemy.sql.expression.Select.offset) can now accept arbitrary SQL expressions as well as integer values.

|  |  |
| --- | --- |
| **Parameters:** | ****offset**** – an integer OFFSET parameter, or a SQL expression that provides an integer result. |

**order\_by**(*\*clauses*)

*inherited from the* [order\_by()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect.order_by" \o "sqlalchemy.sql.expression.GenerativeSelect.order_by) *method of* [GenerativeSelect](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect" \o "sqlalchemy.sql.expression.GenerativeSelect)

return a new selectable with the given list of ORDER BY criterion applied.

The criterion will be appended to any pre-existing ORDER BY criterion.

**outerjoin**(*right*, *onclause=None*, *full=False*)

*inherited from the* [outerjoin()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.outerjoin" \o "sqlalchemy.sql.expression.FromClause.outerjoin) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) from this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) to another [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause), with the "isouter" flag set to True.

E.g.:

**from** **sqlalchemy** **import** outerjoin

j = user\_table.outerjoin(address\_table,

user\_table.c.id == address\_table.c.user\_id)

The above is equivalent to:

j = user\_table.join(

address\_table,

user\_table.c.id == address\_table.c.user\_id,

isouter=**True**)

|  |  |
| --- | --- |
| **Parameters:** | * ****right**** – the right side of the join; this is any [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) object such as a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object, and may also be a selectable-compatible object such as an ORM-mapped class. * ****onclause**** – a SQL expression representing the ON clause of the join. If left at None, [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) will attempt to join the two tables based on a foreign key relationship. * ****full –****if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN.   *New in version 1.1.* |

**See also**

[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)

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

**params**(*\*optionaldict*, *\*\*kwargs*)

*inherited from the* [params()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.params" \o "sqlalchemy.sql.expression.ClauseElement.params) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Return a copy with [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) elements replaced.

Returns a copy of this ClauseElement with [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) elements replaced with values taken from the given dictionary:

**>>>** clause = column('x') + bindparam('foo')

**>>>** print clause.compile().params{'foo':None}

**>>>** print clause.params({'foo':7}).compile().params{'foo':7}

**primary\_key**

*inherited from the* [primary\_key](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.primary_key" \o "sqlalchemy.sql.expression.FromClause.primary_key) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return the collection of Column objects which comprise the primary key of this FromClause.

**replace\_selectable**(*old*, *alias*)

*inherited from the* [replace\_selectable()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.replace_selectable" \o "sqlalchemy.sql.expression.FromClause.replace_selectable) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

replace all occurrences of FromClause 'old' with the given Alias object, returning a copy of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

**scalar**(*\*multiparams*, *\*\*params*)

*inherited from the* [scalar()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable.scalar" \o "sqlalchemy.sql.expression.Executable.scalar) *method of* [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable)

Compile and execute this [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable), returning the result's scalar representation.

**select**(*whereclause=None*, *\*\*params*)

*inherited from the* [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.select" \o "sqlalchemy.sql.expression.FromClause.select) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

return a SELECT of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

**See also**

[select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) - general purpose method which allows for arbitrary column lists.

**tablesample**(*sampling*, *name=None*, *seed=None*)

*inherited from the* [tablesample()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.tablesample" \o "sqlalchemy.sql.expression.FromClause.tablesample) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a TABLESAMPLE alias of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

The return value is the [TableSample](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableSample" \o "sqlalchemy.sql.expression.TableSample) construct also provided by the top-level [tablesample()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.tablesample" \o "sqlalchemy.sql.expression.tablesample) function.

*New in version 1.1.*

**See also**

[tablesample()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.tablesample" \o "sqlalchemy.sql.expression.tablesample) - usage guidelines and parameters

**unique\_params**(*\*optionaldict*, *\*\*kwargs*)

*inherited from the* [unique\_params()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.unique_params" \o "sqlalchemy.sql.expression.ClauseElement.unique_params) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Return a copy with [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) elements replaced.

Same functionality as params(), except adds unique=True to affected bind parameters so that multiple statements can be used.

**with\_for\_update**(*nowait=False*, *read=False*, *of=None*, *skip\_locked=False*, *key\_share=False*)

*inherited from the* [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) *method of* [GenerativeSelect](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect" \o "sqlalchemy.sql.expression.GenerativeSelect)

Specify a FOR UPDATE clause for this [GenerativeSelect](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect" \o "sqlalchemy.sql.expression.GenerativeSelect).

E.g.:

stmt = select([table]).with\_for\_update(nowait=**True**)

On a database like PostgreSQL or Oracle, the above would render a statement like:

SELECT table.a, table.b FROM table FOR UPDATE NOWAIT

on other backends, the nowait option is ignored and instead would produce:

SELECT table.a, table.b FROM table FOR UPDATE

When called with no arguments, the statement will render with the suffix FOR UPDATE. Additional arguments can then be provided which allow for common database-specific variants.

|  |  |
| --- | --- |
| **Parameters:** | * ****nowait**** – boolean; will render FOR UPDATE NOWAIT on Oracle and PostgreSQL dialects. * ****read**** – boolean; will render LOCK IN SHARE MODE on MySQL, FOR SHARE on PostgreSQL. On PostgreSQL, when combined with nowait, will render FOR SHARE NOWAIT. * ****of**** – SQL expression or list of SQL expression elements (typically [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects or a compatible expression) which will render into a FOR UPDATEOF clause; supported by PostgreSQL and Oracle. May render as a table or as a column depending on backend. * ****skip\_locked –****boolean, will render FOR UPDATE SKIP LOCKED on Oracle and PostgreSQL dialects or FOR SHARE SKIP LOCKED if read=True is also specified.   *New in version 1.1.0.*   * ****key\_share –****boolean, will render FOR NO KEY UPDATE, or if combined with read=True will render FOR KEY SHARE, on the PostgreSQL dialect.   *New in version 1.1.0.* |

*class*sqlalchemy.sql.expression.**CTE**(*selectable*, *name=None*, *recursive=False*, *\_cte\_alias=None*, *\_restates=frozenset([])*, *\_suffixes=None*)

Bases: sqlalchemy.sql.expression.Generative, [sqlalchemy.sql.expression.HasSuffixes](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.HasSuffixes" \o "sqlalchemy.sql.expression.HasSuffixes), [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)

Represent a Common Table Expression.

The [CTE](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.CTE" \o "sqlalchemy.sql.expression.CTE) object is obtained using 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 from any selectable. See that method for complete examples.

*New in version 0.7.6.*

**c**

*inherited from the* [c](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.c" \o "sqlalchemy.sql.expression.FromClause.c) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

An alias for the [columns](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.CTE.columns" \o "sqlalchemy.sql.expression.CTE.columns) attribute.

**columns**

*inherited from the* [columns](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.columns" \o "sqlalchemy.sql.expression.FromClause.columns) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

A named-based collection of [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) objects maintained by this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

The [columns](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.CTE.columns" \o "sqlalchemy.sql.expression.CTE.columns), or [c](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.CTE.c" \o "sqlalchemy.sql.expression.CTE.c) collection, is the gateway to the construction of SQL expressions using table-bound or other selectable-bound columns:

select([mytable]).where(mytable.c.somecolumn == 5)

**compare**(*other*, *\*\*kw*)

*inherited from the* [compare()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.compare" \o "sqlalchemy.sql.expression.ClauseElement.compare) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Compare this ClauseElement to the given ClauseElement.

Subclasses should override the default behavior, which is a straight identity comparison.

\*\*kw are arguments consumed by subclass compare() methods and may be used to modify the criteria for comparison. (see [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement))

**compile**(*bind=None*, *dialect=None*, *\*\*kw*)

*inherited from the* [compile()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.compile" \o "sqlalchemy.sql.expression.ClauseElement.compile) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Compile this SQL expression.

The return value is a [Compiled](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Compiled" \o "sqlalchemy.engine.interfaces.Compiled) object. Calling str() or unicode() on the returned value will yield a string representation of the result. The[Compiled](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Compiled" \o "sqlalchemy.engine.interfaces.Compiled) object also can return a dictionary of bind parameter names and values using the params accessor.

|  |  |
| --- | --- |
| **Parameters:** | * ****bind**** – An Engine or Connection from which a Compiled will be acquired. This argument takes precedence over this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)'s bound engine, if any. * ****column\_keys**** – Used for INSERT and UPDATE statements, a list of column names which should be present in the VALUES clause of the compiled statement. If None, all columns from the target table object are rendered. * ****dialect**** – A Dialect instance from which a Compiled will be acquired. This argument takes precedence over the bind argument as well as this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)'s bound engine, if any. * ****inline**** – Used for INSERT statements, for a dialect which does not support inline retrieval of newly generated primary key columns, will force the expression used to create the new primary key value to be rendered inline within the INSERT statement's VALUES clause. This typically refers to Sequence execution but may also refer to any server-side default generation function associated with a primary key Column.用于INSERT语句，对于不支持新生成的主键列的内联检索的方言，将强制用于创建新主键值的表达式在INSERT语句的VALUES子句中内联呈现。 这通常是指Sequence执行，但也可以指与主键Column相关联的任何服务器端默认生成函数。 * ****compile\_kwargs –****optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the literal\_binds flag through:在所有"访问"方法中将附加参数的可选字典传递给编译器。 例如，这允许将任何自定义标志传递到自定义编译构造。 它也用于通过literal\_binds标志的情况：   **from** **sqlalchemy.sql** **import** table, column, select  t = table('t', column('x'))  s = select([t]).where(t.c.x == 5)  print s.compile(compile\_kwargs={"literal\_binds": **True**})  *New in version 0.9.0.* |

**See also**

[How do I render SQL expressions as strings, possibly with bound parameters inlined?](http://docs.sqlalchemy.org/en/rel_1_1/faq/sqlexpressions.html" \l "faq-sql-expression-string)

**correspond\_on\_equivalents**(*column*, *equivalents*)

*inherited from the* [correspond\_on\_equivalents()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.correspond_on_equivalents" \o "sqlalchemy.sql.expression.FromClause.correspond_on_equivalents) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return corresponding\_column for the given column, or if None search for a match in the given dictionary.

**corresponding\_column**(*column*, *require\_embedded=False*)

*inherited from the* [corresponding\_column()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.corresponding_column" \o "sqlalchemy.sql.expression.FromClause.corresponding_column) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Given a [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement), return the exported [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) object from this [Selectable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Selectable" \o "sqlalchemy.sql.expression.Selectable) which corresponds to that original [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) via a common ancestor column.

|  |  |
| --- | --- |
| **Parameters:** | * ****column**** – the target [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) to be matched * ****require\_embedded**** – only return corresponding columns for the given [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement), if the given [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) is actually present within a sub-element of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause). Normally the column will match if it merely shares a common ancestor with one of the exported columns of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause). |

**~~count~~**~~(~~*~~whereclause=None~~*~~,~~*~~\*\*params~~*~~)~~

*~~inherited from the~~*~~[count()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.count" \o "sqlalchemy.sql.expression.FromClause.count)~~*~~method of~~*~~[FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)~~

~~return a SELECT COUNT generated against this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).~~

*~~Deprecated since version 1.1:~~*~~FromClause.count() is deprecated. Counting rows requires that the correct column expression and accommodations for joins, DISTINCT, etc. must be made, otherwise results may not be what's expected.~~ Please use an appropriate func.count() expression directly.

The function generates COUNT against the first column in the primary key of the table, or against the first column in the table overall. Explicit use of func.count() should be preferred:

该函数针对表的主键中的第一列或整个表中的第一列生成COUNT。 显式使用func.count()应该是首选的：

row\_count = conn.scalar(

select([func.count('\*')]).select\_from(table))

**See also**

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

**foreign\_keys**

*inherited from the* [foreign\_keys](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.foreign_keys" \o "sqlalchemy.sql.expression.FromClause.foreign_keys) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return the collection of ForeignKey objects which this FromClause references.

返回此FromClause引用的ForeignKey对象的集合。

**join**(*right*, *onclause=None*, *isouter=False*, *full=False*)

*inherited from the* [join()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.join" \o "sqlalchemy.sql.expression.FromClause.join) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) from this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) to another [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

从这个[FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)返回一个[Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join)到另一个[FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)。

E.g.:

**from** **sqlalchemy** **import** join

j = user\_table.join(address\_table,

user\_table.c.id == address\_table.c.user\_id)stmt = select([user\_table]).select\_from(j)

would emit SQL along the lines of:

SELECT user.id, user.name FROM userJOIN address ON user.id = address.user\_id

|  |  |
| --- | --- |
| **Parameters:** | * ****right**** – the right side of the join; this is any [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) object such as a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object, and may also be a selectable-compatible object such as an ORM-mapped class.联合的右边; 这是任何[FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)对象，如[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)对象，也可以是可选择兼容的对象，如ORM映射类。 * ****onclause**** – a SQL expression representing the ON clause of the join. If left at None, [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) will attempt to join the two tables based on a foreign key relationship. * ****isouter**** – if True, render a LEFT OUTER JOIN, instead of JOIN. * ****full –****if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN. Implies [FromClause.join.isouter](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.join.params.isouter" \o "sqlalchemy.sql.expression.FromClause.join).   *New in version 1.1.* |

**See also**

[join()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.join" \o "sqlalchemy.sql.expression.join) - standalone function

[Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) - the type of object produced

**lateral**(*name=None*)

*inherited from the* [lateral()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.lateral" \o "sqlalchemy.sql.expression.FromClause.lateral) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a LATERAL alias of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

返回此[FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)的LATERAL别名。

The return value is the [Lateral](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Lateral" \o "sqlalchemy.sql.expression.Lateral) construct also provided by the top-level [lateral()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.lateral" \o "sqlalchemy.sql.expression.lateral) function.

返回值是由顶级[lateral()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.lateral" \o "sqlalchemy.sql.expression.lateral)函数提供的[Lateral](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Lateral" \o "sqlalchemy.sql.expression.Lateral)结构。

*New in version 1.1.*

**See also**

[LATERAL correlation](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html" \l "lateral-selects) - overview of usage.

**outerjoin**(*right*, *onclause=None*, *full=False*)

*inherited from the* [outerjoin()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.outerjoin" \o "sqlalchemy.sql.expression.FromClause.outerjoin) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) from this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) to another [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause), with the "isouter" flag set to True.

将从[FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)返回一个[Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join)到另一个[FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)，“isouter”标志设置为True。

E.g.:

**from** **sqlalchemy** **import** outerjoin

j = user\_table.outerjoin(address\_table,

user\_table.c.id == address\_table.c.user\_id)

The above is equivalent to:

以上相当于：

j = user\_table.join(

address\_table,

user\_table.c.id == address\_table.c.user\_id,

isouter=**True**)

|  |  |
| --- | --- |
| **Parameters:** | * ****right**** – the right side of the join; this is any [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) object such as a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object, and may also be a selectable-compatible object such as an ORM-mapped class.联合的右边; 这是任何FromClause对象，如Table对象，也可以是可选择兼容的对象，如ORM映射类。 * ****onclause**** – a SQL expression representing the ON clause of the join. If left at None, [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) will attempt to join the two tables based on a foreign key relationship.表示连接的ON子句的SQL表达式。 如果保持为None，则FromClause.join() 将根据外键关系尝试加入这两个表。 * ****full –****if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN.如果为True，则呈现FULL OUTER JOIN，而不是LEFT OUTER JOIN。   *New in version 1.1.* |

**See also**

[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)

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

**params**(*\*optionaldict*, *\*\*kwargs*)

*inherited from the* [params()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.params" \o "sqlalchemy.sql.expression.ClauseElement.params) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Return a copy with [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) elements replaced.

返回一个带有[bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam)元素的副本。

Returns a copy of this ClauseElement with [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) elements replaced with values taken from the given dictionary:

使用从指定字典取得的值替换[bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam)元素返回此ClauseElement的副本：

**>>>** clause = column('x') + bindparam('foo')

**>>>** print clause.compile().params{'foo':None}

**>>>** print clause.params({'foo':7}).compile().params{'foo':7}

**primary\_key**

*inherited from the* [primary\_key](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.primary_key" \o "sqlalchemy.sql.expression.FromClause.primary_key) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return the collection of Column objects which comprise the primary key of this FromClause.

返回构成此FromClause主键的Column对象的集合。

**replace\_selectable**(*old*, *alias*)

*inherited from the* [replace\_selectable()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.replace_selectable" \o "sqlalchemy.sql.expression.FromClause.replace_selectable) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

replace all occurrences of FromClause 'old' with the given Alias object, returning a copy of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

**select**(*whereclause=None*, *\*\*params*)

*inherited from the* [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.select" \o "sqlalchemy.sql.expression.FromClause.select) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

return a SELECT of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

**See also**

[select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) - general purpose method which allows for arbitrary column lists.

**suffix\_with**(*\*expr*, *\*\*kw*)

*inherited from the* [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) *method of* [HasSuffixes](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.HasSuffixes" \o "sqlalchemy.sql.expression.HasSuffixes)

Add one or more expressions following the statement as a whole.

This is used to support backend-specific suffix keywords on certain constructs.

E.g.:

stmt = select([col1, col2]).cte().suffix\_with(

"cycle empno set y\_cycle to 1 default 0", dialect="oracle")

Multiple suffixes can be specified by multiple calls to [suffix\_with()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.CTE.suffix_with" \o "sqlalchemy.sql.expression.CTE.suffix_with).

|  |  |
| --- | --- |
| **Parameters:** | * ****\*expr**** – textual or [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement) construct which will be rendered following the target clause. * ****\*\*kw**** – A single keyword 'dialect' is accepted. This is an optional string dialect name which will limit rendering of this suffix to only that dialect. |

**tablesample**(*sampling*, *name=None*, *seed=None*)

*inherited from the* [tablesample()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.tablesample" \o "sqlalchemy.sql.expression.FromClause.tablesample) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a TABLESAMPLE alias of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

The return value is the [TableSample](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableSample" \o "sqlalchemy.sql.expression.TableSample) construct also provided by the top-level [tablesample()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.tablesample" \o "sqlalchemy.sql.expression.tablesample) function.

*New in version 1.1.*

**See also**

[tablesample()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.tablesample" \o "sqlalchemy.sql.expression.tablesample) - usage guidelines and parameters

**unique\_params**(*\*optionaldict*, *\*\*kwargs*)

*inherited from the* [unique\_params()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.unique_params" \o "sqlalchemy.sql.expression.ClauseElement.unique_params) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Return a copy with [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) elements replaced.

Same functionality as params(), except adds unique=True to affected bind parameters so that multiple statements can be used.

*class*sqlalchemy.sql.expression.**Executable**

Bases: sqlalchemy.sql.expression.Generative

Mark a ClauseElement as supporting execution.

[Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable) is a superclass for all "statement" types of objects, including [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select), [delete()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.delete" \o "sqlalchemy.sql.expression.delete), [update()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.update" \o "sqlalchemy.sql.expression.update), [insert()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.insert" \o "sqlalchemy.sql.expression.insert), [text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text).

**bind**

Returns 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) to which this [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable) is bound, or None if none found.

This is a traversal which checks locally, then checks among the "from" clauses of associated objects until a bound engine or connection is found.

**execute**(*\*multiparams*, *\*\*params*)

Compile and execute this [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable).

**execution\_options**(*\*\*kw*)

Set non-SQL options for the statement which take effect during execution.

Execution options can be set on a per-statement or per [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) basis. Additionally, the [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) and ORM [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) objects provide access to execution options which they in turn configure upon connections.

The [execution\_options()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable.execution_options" \o "sqlalchemy.sql.expression.Executable.execution_options) method is generative. A new instance of this statement is returned that contains the options:

statement = select([table.c.x, table.c.y])statement = statement.execution\_options(autocommit=**True**)

Note that only a subset of possible execution options can be applied to a statement - these include "autocommit" and "stream\_results", but not "isolation\_level" or "compiled\_cache". See [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) for a full list of possible options.

**See also**

[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)

[Query.execution\_options()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.execution_options" \o "sqlalchemy.orm.query.Query.execution_options)

**scalar**(*\*multiparams*, *\*\*params*)

Compile and execute this [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable), returning the result's scalar representation.

*class*sqlalchemy.sql.expression.**FromClause**

Bases: [sqlalchemy.sql.expression.Selectable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Selectable" \o "sqlalchemy.sql.expression.Selectable)

Represent an element that can be used within the FROM clause of a SELECT statement.

The most common forms of [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) are the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) and the [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) constructs. Key features common to all [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) objects include:

* a [c](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.c" \o "sqlalchemy.sql.expression.FromClause.c) collection, which provides per-name access to a collection of [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) objects.
* a [primary\_key](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.primary_key" \o "sqlalchemy.sql.expression.FromClause.primary_key) attribute, which is a collection of all those [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) objects that indicate the primary\_key flag.
* Methods to generate various derivations of a "from" clause, including [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), [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), [FromClause.select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.select" \o "sqlalchemy.sql.expression.FromClause.select).

**alias**(*name=None*, *flat=False*)

return an alias of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

This is shorthand for calling:

**from** **sqlalchemy** **import** aliasa = alias(self, name=name)

See [alias()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.alias" \o "sqlalchemy.sql.expression.alias) for details.

**c**

An alias for the [columns](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.columns" \o "sqlalchemy.sql.expression.FromClause.columns) attribute.

**columns**

A named-based collection of [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) objects maintained by this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

The [columns](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.columns" \o "sqlalchemy.sql.expression.FromClause.columns), or [c](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.c" \o "sqlalchemy.sql.expression.FromClause.c) collection, is the gateway to the construction of SQL expressions using table-bound or other selectable-bound columns:

select([mytable]).where(mytable.c.somecolumn == 5)

**correspond\_on\_equivalents**(*column*, *equivalents*)

Return corresponding\_column for the given column, or if None search for a match in the given dictionary.

**corresponding\_column**(*column*, *require\_embedded=False*)

Given a [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement), return the exported [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) object from this [Selectable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Selectable" \o "sqlalchemy.sql.expression.Selectable) which corresponds to that original [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) via a common ancestor column.

|  |  |
| --- | --- |
| **Parameters:** | * ****column**** – the target [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) to be matched * ****require\_embedded**** – only return corresponding columns for the given [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement), if the given [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) is actually present within a sub-element of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause). Normally the column will match if it merely shares a common ancestor with one of the exported columns of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause). |

**count**(*whereclause=None*, *\*\*params*)

return a SELECT COUNT generated against this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

*Deprecated since version 1.1:*FromClause.count() is deprecated. Counting rows requires that the correct column expression and accommodations for joins, DISTINCT, etc. must be made, otherwise results may not be what's expected. Please use an appropriate func.count() expression directly.

The function generates COUNT against the first column in the primary key of the table, or against the first column in the table overall. Explicit use offunc.count() should be preferred:

row\_count = conn.scalar(

select([func.count('\*')]).select\_from(table))

**See also**

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

**description**

a brief description of this FromClause.

Used primarily for error message formatting.

**foreign\_keys**

Return the collection of ForeignKey objects which this FromClause references.

**is\_derived\_from**(*fromclause*)

Return True if this FromClause is 'derived' from the given FromClause.

An example would be an Alias of a Table is derived from that Table.

**join**(*right*, *onclause=None*, *isouter=False*, *full=False*)

Return a [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) from this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) to another [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

E.g.:

**from** **sqlalchemy** **import** join

j = user\_table.join(address\_table,

user\_table.c.id == address\_table.c.user\_id)stmt = select([user\_table]).select\_from(j)

would emit SQL along the lines of:

SELECT user.id, user.name FROM userJOIN address ON user.id = address.user\_id

|  |  |
| --- | --- |
| **Parameters:** | * ****right**** – the right side of the join; this is any [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) object such as a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object, and may also be a selectable-compatible object such as an ORM-mapped class. * ****onclause**** – a SQL expression representing the ON clause of the join. If left at None, [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) will attempt to join the two tables based on a foreign key relationship. * ****isouter**** – if True, render a LEFT OUTER JOIN, instead of JOIN. * ****full**** –   if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN. Implies [FromClause.join.isouter](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.join.params.isouter" \o "sqlalchemy.sql.expression.FromClause.join).  *New in version 1.1.* |

**See also**

[join()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.join" \o "sqlalchemy.sql.expression.join) - standalone function

[Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) - the type of object produced

**lateral**(*name=None*)

Return a LATERAL alias of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

The return value is the [Lateral](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Lateral" \o "sqlalchemy.sql.expression.Lateral) construct also provided by the top-level [lateral()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.lateral" \o "sqlalchemy.sql.expression.lateral) function.

*New in version 1.1.*

**See also**

[LATERAL correlation](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html" \l "lateral-selects) - overview of usage.

**outerjoin**(*right*, *onclause=None*, *full=False*)

Return a [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) from this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) to another [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause), with the "isouter" flag set to True.

E.g.:

**from** **sqlalchemy** **import** outerjoin

j = user\_table.outerjoin(address\_table,

user\_table.c.id == address\_table.c.user\_id)

The above is equivalent to:

j = user\_table.join(

address\_table,

user\_table.c.id == address\_table.c.user\_id,

isouter=**True**)

|  |  |
| --- | --- |
| **Parameters:** | * ****right**** – the right side of the join; this is any [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) object such as a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object, and may also be a selectable-compatible object such as an ORM-mapped class. * ****onclause**** – a SQL expression representing the ON clause of the join. If left at None, [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) will attempt to join the two tables based on a foreign key relationship. * ****full –****if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN.   *New in version 1.1.* |

**See also**

[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)

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

**primary\_key**

Return the collection of Column objects which comprise the primary key of this FromClause.

**replace\_selectable**(*old*, *alias*)

replace all occurrences of FromClause 'old' with the given Alias object, returning a copy of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

**schema***= None*

Define the 'schema' attribute for this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

This is typically None for most objects except that of [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table), where it is taken as the value of the [Table.schema](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.params.schema" \o "sqlalchemy.schema.Table) argument.

**select**(*whereclause=None*, *\*\*params*)

return a SELECT of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

**See also**

[select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) - general purpose method which allows for arbitrary column lists.

**tablesample**(*sampling*, *name=None*, *seed=None*)

Return a TABLESAMPLE alias of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

The return value is the [TableSample](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableSample" \o "sqlalchemy.sql.expression.TableSample) construct also provided by the top-level [tablesample()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.tablesample" \o "sqlalchemy.sql.expression.tablesample) function.

*New in version 1.1.*

**See also**

[tablesample()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.tablesample" \o "sqlalchemy.sql.expression.tablesample) - usage guidelines and parameters

*class*sqlalchemy.sql.expression.**GenerativeSelect**(*use\_labels=False*, *for\_update=False*, *limit=None*, *offset=None*, *order\_by=None*, *group\_by=None*, *bind=None*, *autocommit=None*)

Bases: [sqlalchemy.sql.expression.SelectBase](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.SelectBase" \o "sqlalchemy.sql.expression.SelectBase)

Base class for SELECT statements where additional elements can be added.

This serves as the base for [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select) and [CompoundSelect](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.CompoundSelect" \o "sqlalchemy.sql.expression.CompoundSelect) where elements such as ORDER BY, GROUP BY can be added and column rendering can be controlled. Compare to [TextAsFrom](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TextAsFrom" \o "sqlalchemy.sql.expression.TextAsFrom), which, while it subclasses [SelectBase](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.SelectBase" \o "sqlalchemy.sql.expression.SelectBase) and is also a SELECT construct, represents a fixed textual string which cannot be altered at this level, only wrapped as a subquery.

*New in version 0.9.0:*[GenerativeSelect](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect" \o "sqlalchemy.sql.expression.GenerativeSelect) was added to provide functionality specific to [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select) and [CompoundSelect](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.CompoundSelect" \o "sqlalchemy.sql.expression.CompoundSelect) while allowing [SelectBase](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.SelectBase" \o "sqlalchemy.sql.expression.SelectBase) to be used for other SELECT-like objects, e.g. [TextAsFrom](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TextAsFrom" \o "sqlalchemy.sql.expression.TextAsFrom).

**alias**(*name=None*, *flat=False*)

*inherited from the* [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 of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

return an alias of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

This is shorthand for calling:

**from** **sqlalchemy** **import** aliasa = alias(self, name=name)

See [alias()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.alias" \o "sqlalchemy.sql.expression.alias) for details.

**append\_group\_by**(*\*clauses*)

Append the given GROUP BY criterion applied to this selectable.

The criterion will be appended to any pre-existing GROUP BY criterion.

This is an ****in-place**** mutation method; the [group\_by()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect.group_by" \o "sqlalchemy.sql.expression.GenerativeSelect.group_by) method is preferred, as it provides standard [method chaining](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-method-chaining).

**append\_order\_by**(*\*clauses*)

Append the given ORDER BY criterion applied to this selectable.

The criterion will be appended to any pre-existing ORDER BY criterion.

This is an ****in-place**** mutation method; the [order\_by()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect.order_by" \o "sqlalchemy.sql.expression.GenerativeSelect.order_by) method is preferred, as it provides standard [method chaining](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-method-chaining).

**apply\_labels**()

return a new selectable with the 'use\_labels' flag set to True.

This will result in column expressions being generated using labels against their table name, such as "SELECT somecolumn AS tablename\_somecolumn". This allows selectables which contain multiple FROM clauses to produce a unique set of column names regardless of name conflicts among the individual FROM clauses.

**as\_scalar**()

*inherited from the* [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) *method of* [SelectBase](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.SelectBase" \o "sqlalchemy.sql.expression.SelectBase)

return a 'scalar' representation of this selectable, which can be used as a column expression.

Typically, a select statement which has only one column in its columns clause is eligible to be used as a scalar expression.

The returned object is an instance of [ScalarSelect](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.ScalarSelect" \o "sqlalchemy.sql.expression.ScalarSelect).

**~~autocommit~~**~~()~~

*~~inherited from the~~*~~[autocommit()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.SelectBase.autocommit" \o "sqlalchemy.sql.expression.SelectBase.autocommit)~~*~~method of~~*~~[SelectBase](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.SelectBase" \o "sqlalchemy.sql.expression.SelectBase)~~

~~return a new selectable with the 'autocommit' flag set to True.~~

*~~Deprecated since version 0.6:~~*~~autocommit() is deprecated.~~ Use [Executable.execution\_options()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable.execution_options" \o "sqlalchemy.sql.expression.Executable.execution_options) with the 'autocommit' flag.

**bind**

*inherited from the* [bind](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable.bind" \o "sqlalchemy.sql.expression.Executable.bind) *attribute of* [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable)

Returns 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) to which this [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable) is bound, or None if none found.

返回此可执行文件绑定到的引擎或连接，否则返回无。

This is a traversal which checks locally, then checks among the "from" clauses of associated objects until a bound engine or connection is found.

这是一个在本地进行检查的遍历，然后在相关对象的“从”子句之间进行检查，直到找到绑定的引擎或连接。

**c**

*inherited from the* [c](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.c" \o "sqlalchemy.sql.expression.FromClause.c) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

An alias for the [columns](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect.columns" \o "sqlalchemy.sql.expression.GenerativeSelect.columns) attribute.

**columns**

*inherited from the* [columns](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.columns" \o "sqlalchemy.sql.expression.FromClause.columns) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

A named-based collection of [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) objects maintained by this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

由此FromClause维护的基于命名的ColumnElement对象集合。

The [columns](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect.columns" \o "sqlalchemy.sql.expression.GenerativeSelect.columns), or [c](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect.c" \o "sqlalchemy.sql.expression.GenerativeSelect.c) collection, is the gateway to the construction of SQL expressions using table-bound or other selectable-bound columns:

列或c集合是使用表格绑定或其他可选择绑定列构建SQL表达式的网关：

select([mytable]).where(mytable.c.somecolumn == 5)

**compare**(*other*, *\*\*kw*)

*inherited from the* [compare()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.compare" \o "sqlalchemy.sql.expression.ClauseElement.compare) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Compare this ClauseElement to the given ClauseElement.

将此条款元素与给定的条款元素进行比较。

Subclasses should override the default behavior, which is a straight identity comparison.

子类应该覆盖默认行为，这是一个直接的身份比较。

\*\*kw are arguments consumed by subclass compare() methods and may be used to modify the criteria for comparison. (see [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement))

\*\* kw是由subclass compare() 方法消耗的参数，可用于修改比较条件。（见ColumnElement）

**compile**(*bind=None*, *dialect=None*, *\*\*kw*)

*inherited from the* [compile()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.compile" \o "sqlalchemy.sql.expression.ClauseElement.compile) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Compile this SQL expression.

编译此SQL表达式。

The return value is a [Compiled](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Compiled" \o "sqlalchemy.engine.interfaces.Compiled) object. Calling str() or unicode() on the returned value will yield a string representation of the result. The[Compiled](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Compiled" \o "sqlalchemy.engine.interfaces.Compiled) object also can return a dictionary of bind parameter names and values using the params accessor.

返回值是一个编译对象。 对返回的值调用str() 或unicode() 将产生结果的字符串表示形式。 Compiled对象还可以使用params访问器返回绑定参数名称和值的字典。

|  |  |
| --- | --- |
| **Parameters:** | * ****bind**** – An Engine or Connection from which a Compiled will be acquired. This argument takes precedence over this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)'s bound engine, if any.将从其获取编译的引擎或连接。 这个参数优先于这个ClauseElement的绑定引擎（如果有的话）。 * ****column\_keys**** – Used for INSERT and UPDATE statements, a list of column names which should be present in the VALUES clause of the compiled statement. If None, all columns from the target table object are rendered.用于INSERT和UPDATE语句，应在编译语句的VALUES子句中存在列名称列表。 如果为None，则会渲染目标表对象中的所有列 * ****dialect**** – A Dialect instance from which a Compiled will be acquired. This argument takes precedence over the bind argument as well as this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)'s bound engine, if any.将从其中获取编译的方言实例。 此参数优先于bind参数以及此ClauseElement的绑定引擎（如果有）。 * ****inline**** – Used for INSERT statements, for a dialect which does not support inline retrieval of newly generated primary key columns, will force the expression used to create the new primary key value to be rendered inline within the INSERT statement's VALUES clause. This typically refers to Sequence execution but may also refer to any server-side default generation function associated with a primary key Column.用于INSERT语句，对于不支持新生成的主键列的内联检索的方言，将强制用于创建新主键值的表达式在INSERT语句的VALUES子句中内联呈现。 这通常是指Sequence执行，但也可以指与主键Column相关联的任何服务器端默认生成函数。 * ****compile\_kwargs –****optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the literal\_binds flag through:在所有"访问"方法中将附加参数的可选字典传递给编译器。 例如，这允许将任何自定义标志传递到自定义编译构造。 它也用于通过literal\_binds标志的情况：   **from** **sqlalchemy.sql** **import** table, column, select  t = table('t', column('x'))  s = select([t]).where(t.c.x == 5)  print s.compile(compile\_kwargs={"literal\_binds": **True**})  *New in version 0.9.0.* |

**See also**

[How do I render SQL expressions as strings, possibly with bound parameters inlined?](http://docs.sqlalchemy.org/en/rel_1_1/faq/sqlexpressions.html" \l "faq-sql-expression-string)

**correspond\_on\_equivalents**(*column*, *equivalents*)

*inherited from the* [correspond\_on\_equivalents()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.correspond_on_equivalents" \o "sqlalchemy.sql.expression.FromClause.correspond_on_equivalents) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return corresponding\_column for the given column, or if None search for a match in the given dictionary.返回给定列的对应列，或者如果None在给定字典中搜索匹配。

**corresponding\_column**(*column*, *require\_embedded=False*)

*inherited from the* [corresponding\_column()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.corresponding_column" \o "sqlalchemy.sql.expression.FromClause.corresponding_column) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Given a [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement), return the exported [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) object from this [Selectable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Selectable" \o "sqlalchemy.sql.expression.Selectable) which corresponds to that original [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) via a common ancestor column.给定一个ColumnElement，通过一个共同的祖先列从该可选择返回导出的ColumnElement对象，该对象与该原始列相对应。

|  |  |
| --- | --- |
| **Parameters:** | * ****column**** – the target [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) to be matched * ****require\_embedded**** – only return corresponding columns for the given [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement), if the given [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) is actually present within a sub-element of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause). Normally the column will match if it merely shares a common ancestor with one of the exported columns of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause). |

**~~count~~**~~(~~*~~whereclause=None~~*~~,~~*~~\*\*params~~*~~)~~

*~~inherited from the~~*~~[count()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.count" \o "sqlalchemy.sql.expression.FromClause.count)~~*~~method of~~*~~[FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)~~

~~return a SELECT COUNT generated against this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).~~

*~~Deprecated since version 1.1:~~*~~FromClause.count() is deprecated. Counting rows requires that the correct column expression and accommodations for joins, DISTINCT, etc. must be made, otherwise results may not be what's expected. Please use an appropriate func.count() expression directly.~~

~~The function generates COUNT against the first column in the primary key of the table, or against the first column in the table overall. Explicit use offunc.count() should be preferred:~~

~~row\_count = conn.scalar(~~

~~select([func.count('\*')]).select\_from(table))~~

**~~See also~~**

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

**cte**(*name=None*, *recursive=False*)

*inherited from the* [cte()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.HasCTE.cte" \o "sqlalchemy.sql.expression.HasCTE.cte) *method of* [HasCTE](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.HasCTE" \o "sqlalchemy.sql.expression.HasCTE)

Return a new [CTE](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.CTE" \o "sqlalchemy.sql.expression.CTE), or Common Table Expression instance.

返回一个新的CTE或Common Table Expression实例。

Common table expressions are a SQL standard whereby SELECT statements can draw upon secondary statements specified along with the primary statement, using a clause called "WITH". Special semantics regarding UNION can also be employed to allow "recursive" queries, where a SELECT statement can draw upon the set of rows that have previously been selected.

公用表表达式是一种SQL标准，其中SELECT语句可以使用一个称为"WITH"的子句来使用主语句指定的辅助语句。 还可以使用有关UNION的特殊语义来允许"递归"查询，其中SELECT语句可以对先前已选择的行集合进行绘制。

CTEs can also be applied to DML constructs UPDATE, INSERT and DELETE on some databases, both as a source of CTE rows when combined with RETURNING, as well as a consumer of CTE rows.

CTE还可以应用于某些数据库上的DML构造UPDATE，INSERT和DELETE，作为与RETURNING组合的CTE行的源，以及CTE行的使用者。

SQLAlchemy detects [CTE](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.CTE" \o "sqlalchemy.sql.expression.CTE) objects, which are treated similarly to [Alias](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Alias" \o "sqlalchemy.sql.expression.Alias) objects, as special elements to be delivered to the FROM clause of the statement as well as to a WITH clause at the top of the statement.

SQLAlchemy检测到与Alias对象类似的CTE对象作为要传递到语句的FROM子句的特殊元素以及语句顶部的WITH子句。

*Changed in version 1.1:*Added support for UPDATE/INSERT/DELETE as CTE, CTEs added to UPDATE/INSERT/DELETE.

在版本1.1中更改：添加对UPDATE / INSERT / DELETE作为CTE的支持，CTE添加到UPDATE / INSERT / DELETE。

|  |  |
| --- | --- |
| **Parameters:** | * ****name**** – name given to the common table expression. Like \_FromClause.alias(), the name can be left as None in which case an anonymous symbol will be used at query compile time. * ****recursive**** – if True, will render WITH RECURSIVE. A recursive common table expression is intended to be used in conjunction with UNION ALL in order to derive rows from those already selected. |

The following examples include two from PostgreSQL's documentation at <http://www.postgresql.org/docs/current/static/queries-with.html>, as well as additional examples.

Example 1, non recursive:

**from** **sqlalchemy** **import** (Table, Column, String, Integer,

MetaData, select, func)

metadata = MetaData()

orders = Table('orders', metadata,

Column('region', String),

Column('amount', Integer),

Column('product', String),

Column('quantity', Integer))

regional\_sales = select([

orders.c.region,

func.sum(orders.c.amount).label('total\_sales')

]).group\_by(orders.c.region).cte("regional\_sales")

top\_regions = select([regional\_sales.c.region]).\

where(

regional\_sales.c.total\_sales >

select([

func.sum(regional\_sales.c.total\_sales)/10

])

).cte("top\_regions")

statement = select([

orders.c.region,

orders.c.product,

func.sum(orders.c.quantity).label("product\_units"),

func.sum(orders.c.amount).label("product\_sales")

]).where(orders.c.region.in\_(

select([top\_regions.c.region])

)).group\_by(orders.c.region, orders.c.product)

result = conn.execute(statement).fetchall()

Example 2, WITH RECURSIVE:

**from** **sqlalchemy** **import** (Table, Column, String, Integer,

MetaData, select, func)

metadata = MetaData()

parts = Table('parts', metadata,

Column('part', String),

Column('sub\_part', String),

Column('quantity', Integer),)

included\_parts = select([

parts.c.sub\_part,

parts.c.part,

parts.c.quantity]).\

where(parts.c.part=='our part').\

cte(recursive=**True**)

incl\_alias = included\_parts.alias()parts\_alias = parts.alias()included\_parts = included\_parts.union\_all(

select([

parts\_alias.c.sub\_part,

parts\_alias.c.part,

parts\_alias.c.quantity

]).

where(parts\_alias.c.part==incl\_alias.c.sub\_part))

statement = select([

included\_parts.c.sub\_part,

func.sum(included\_parts.c.quantity).

label('total\_quantity')

]).\

group\_by(included\_parts.c.sub\_part)

result = conn.execute(statement).fetchall()

Example 3, an upsert using UPDATE and INSERT with CTEs:

**from** **datetime** **import** date

**from** **sqlalchemy** **import** (MetaData, Table, Column, Integer,

Date, select, literal, and\_, exists)

metadata = MetaData()

visitors = Table('visitors', metadata,

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

Column('date', Date, primary\_key=**True**),

Column('count', Integer),)

*# add 5 visitors for the product\_id == 1*product\_id = 1day = date.today()count = 5

update\_cte = (

visitors.update()

.where(and\_(visitors.c.product\_id == product\_id,

visitors.c.date == day))

.values(count=visitors.c.count + count)

.returning(literal(1))

.cte('update\_cte'))

upsert = visitors.insert().from\_select(

[visitors.c.product\_id, visitors.c.date, visitors.c.count],

select([literal(product\_id), literal(day), literal(count)])

.where(~exists(update\_cte.select())))

connection.execute(upsert)

**See also**

[orm.query.Query.cte()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.cte" \o "sqlalchemy.orm.query.Query.cte) - ORM version of [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).

**description**

*inherited from the* [description](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.description" \o "sqlalchemy.sql.expression.FromClause.description) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

a brief description of this FromClause.

Used primarily for error message formatting.

**execute**(*\*multiparams*, *\*\*params*)

*inherited from the* [execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable.execute" \o "sqlalchemy.sql.expression.Executable.execute) *method of* [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable)

Compile and execute this [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable).

**execution\_options**(*\*\*kw*)

*inherited from the* [execution\_options()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable.execution_options" \o "sqlalchemy.sql.expression.Executable.execution_options) *method of* [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable)

Set non-SQL options for the statement which take effect during execution.

Execution options can be set on a per-statement or per [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) basis. Additionally, the [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) and ORM [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) objects provide access to execution options which they in turn configure upon connections.

The [execution\_options()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect.execution_options" \o "sqlalchemy.sql.expression.GenerativeSelect.execution_options) method is generative. A new instance of this statement is returned that contains the options:

statement = select([table.c.x, table.c.y])statement = statement.execution\_options(autocommit=**True**)

Note that only a subset of possible execution options can be applied to a statement - these include "autocommit" and "stream\_results", but not "isolation\_level" or "compiled\_cache". See [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) for a full list of possible options.

**See also**

[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)

[Query.execution\_options()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.execution_options" \o "sqlalchemy.orm.query.Query.execution_options)

**for\_update**

Provide legacy dialect support for the for\_update attribute.

**foreign\_keys**

*inherited from the* [foreign\_keys](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.foreign_keys" \o "sqlalchemy.sql.expression.FromClause.foreign_keys) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return the collection of ForeignKey objects which this FromClause references.

**get\_children**(*\*\*kwargs*)

*inherited from the* [get\_children()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.get_children" \o "sqlalchemy.sql.expression.ClauseElement.get_children) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Return immediate child elements of this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement).

返回此ClauseElement的即时子元素。

This is used for visit traversal.

这用于访问遍历。

\*\*kwargs may contain flags that change the collection that is returned, for example to return a subset of items in order to cut down on larger traversals, or to return child items from a different context (such as schema-level collections instead of clause-level).

\*\* kwargs可能包含改变所返回的集合的标志，例如返回一个子集，以减少较大的遍历，或从不同的上下文返回子项（如模式级集合而不是子句） -水平）。

**group\_by**(*\*clauses*)

return a new selectable with the given list of GROUP BY criterion applied.

The criterion will be appended to any pre-existing GROUP BY criterion.

**is\_derived\_from**(*fromclause*)

*inherited from the* [is\_derived\_from()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.is_derived_from" \o "sqlalchemy.sql.expression.FromClause.is_derived_from) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return True if this FromClause is 'derived' from the given FromClause.

An example would be an Alias of a Table is derived from that Table.

**join**(*right*, *onclause=None*, *isouter=False*, *full=False*)

*inherited from the* [join()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.join" \o "sqlalchemy.sql.expression.FromClause.join) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) from this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) to another [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

E.g.:

**from** **sqlalchemy** **import** join

j = user\_table.join(address\_table,

user\_table.c.id == address\_table.c.user\_id)stmt = select([user\_table]).select\_from(j)

would emit SQL along the lines of:

SELECT user.id, user.name FROM userJOIN address ON user.id = address.user\_id

|  |  |
| --- | --- |
| **Parameters:** | * ****right**** – the right side of the join; this is any [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) object such as a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object, and may also be a selectable-compatible object such as an ORM-mapped class. * ****onclause**** – a SQL expression representing the ON clause of the join. If left at None, [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) will attempt to join the two tables based on a foreign key relationship. * ****isouter**** – if True, render a LEFT OUTER JOIN, instead of JOIN. * ****full –****if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN. Implies [FromClause.join.isouter](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.join.params.isouter" \o "sqlalchemy.sql.expression.FromClause.join).   *New in version 1.1.* |

**See also**

[join()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.join" \o "sqlalchemy.sql.expression.join) - standalone function

[Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) - the type of object produced

**label**(*name*)

*inherited from the* [label()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.SelectBase.label" \o "sqlalchemy.sql.expression.SelectBase.label) *method of* [SelectBase](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.SelectBase" \o "sqlalchemy.sql.expression.SelectBase)

return a 'scalar' representation of this selectable, embedded as a subquery with a label.

**See also**

[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).

**lateral**(*name=None*)

*inherited from the* [lateral()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.lateral" \o "sqlalchemy.sql.expression.FromClause.lateral) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a LATERAL alias of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

The return value is the [Lateral](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Lateral" \o "sqlalchemy.sql.expression.Lateral) construct also provided by the top-level [lateral()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.lateral" \o "sqlalchemy.sql.expression.lateral) function.

*New in version 1.1.*

**See also**

[LATERAL correlation](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html" \l "lateral-selects) - overview of usage.

**limit**(*limit*)

return a new selectable with the given LIMIT criterion applied.

This is a numerical value which usually renders as a LIMIT expression in the resulting select. Backends that don't support LIMIT will attempt to provide similar functionality.

*Changed in version 1.0.0:*- [Select.limit()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.limit" \o "sqlalchemy.sql.expression.Select.limit) can now accept arbitrary SQL expressions as well as integer values.

|  |  |
| --- | --- |
| **Parameters:** | ****limit**** – an integer LIMIT parameter, or a SQL expression that provides an integer result. |

**offset**(*offset*)

return a new selectable with the given OFFSET criterion applied.

This is a numeric value which usually renders as an OFFSET expression in the resulting select. Backends that don't support OFFSET will attempt to provide similar functionality.

*Changed in version 1.0.0:*- [Select.offset()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.offset" \o "sqlalchemy.sql.expression.Select.offset) can now accept arbitrary SQL expressions as well as integer values.

|  |  |
| --- | --- |
| **Parameters:** | ****offset**** – an integer OFFSET parameter, or a SQL expression that provides an integer result. |

**order\_by**(*\*clauses*)

return a new selectable with the given list of ORDER BY criterion applied.

The criterion will be appended to any pre-existing ORDER BY criterion.

**outerjoin**(*right*, *onclause=None*, *full=False*)

*inherited from the* [outerjoin()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.outerjoin" \o "sqlalchemy.sql.expression.FromClause.outerjoin) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) from this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) to another [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause), with the "isouter" flag set to True.

E.g.:

**from** **sqlalchemy** **import** outerjoin

j = user\_table.outerjoin(address\_table,

user\_table.c.id == address\_table.c.user\_id)

The above is equivalent to:

j = user\_table.join(

address\_table,

user\_table.c.id == address\_table.c.user\_id,

isouter=**True**)

|  |  |
| --- | --- |
| **Parameters:** | * ****right**** – the right side of the join; this is any [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) object such as a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object, and may also be a selectable-compatible object such as an ORM-mapped class. * ****onclause**** – a SQL expression representing the ON clause of the join. If left at None, [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) will attempt to join the two tables based on a foreign key relationship. * ****full –****if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN.   *New in version 1.1.* |

**See also**

[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)

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

**params**(*\*optionaldict*, *\*\*kwargs*)

*inherited from the* [params()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.params" \o "sqlalchemy.sql.expression.ClauseElement.params) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Return a copy with [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) elements replaced.

Returns a copy of this ClauseElement with [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) elements replaced with values taken from the given dictionary:

**>>>** clause = column('x') + bindparam('foo')

**>>>** print clause.compile().params{'foo':None}

**>>>** print clause.params({'foo':7}).compile().params{'foo':7}

**primary\_key**

*inherited from the* [primary\_key](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.primary_key" \o "sqlalchemy.sql.expression.FromClause.primary_key) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return the collection of Column objects which comprise the primary key of this FromClause.

**replace\_selectable**(*old*, *alias*)

*inherited from the* [replace\_selectable()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.replace_selectable" \o "sqlalchemy.sql.expression.FromClause.replace_selectable) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

replace all occurrences of FromClause 'old' with the given Alias object, returning a copy of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

**scalar**(*\*multiparams*, *\*\*params*)

*inherited from the* [scalar()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable.scalar" \o "sqlalchemy.sql.expression.Executable.scalar) *method of* [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable)

Compile and execute this [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable), returning the result's scalar representation.

**select**(*whereclause=None*, *\*\*params*)

*inherited from the* [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.select" \o "sqlalchemy.sql.expression.FromClause.select) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

return a SELECT of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

**See also**

[select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) - general purpose method which allows for arbitrary column lists.

**self\_group**(*against=None*)

*inherited from the* [self\_group()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.self_group" \o "sqlalchemy.sql.expression.ClauseElement.self_group) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Apply a 'grouping' to this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement).

对此ClauseElement应用"分组"。

This method is overridden by subclasses to return a "grouping" construct, i.e. parenthesis. In particular it's used by "binary" expressions to provide a grouping around themselves when placed into a larger expression, as well as by [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) constructs when placed into the FROM clause of another [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select). (Note that subqueries should be normally created using the [Select.alias()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.alias" \o "sqlalchemy.sql.expression.Select.alias) method, as many platforms require nested SELECT statements to be named).

这个方法被子类覆盖，以返回一个"分组"结构，即括号。 特别地，它被"二进制"表达式用于在放入更大的表达式时以及通过将select() 结构放入另一个select() 的FROM子句时围绕它们进行分组。 （请注意，通常使用Select.alias() 方法创建子查询，因为许多平台需要嵌套的SELECT语句来命名）。

As expressions are composed together, the application of [self\_group()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect.self_group" \o "sqlalchemy.sql.expression.GenerativeSelect.self_group) is automatic - end-user code should never need to use this method directly. Note that SQLAlchemy's clause constructs take operator precedence into account - so parenthesis might not be needed, for example, in an expression like x OR (y AND z) - AND takes precedence over OR.

由于表达式组合在一起，self\_group() 的应用是自动的 - 最终用户代码不应该直接使用这种方法。 请注意，SQLAlchemy的子句构造考虑到运算符优先级，因此可能不需要括号，例如，在像x OR（y AND z）之类的表达式中，AND优先于OR。

The base [self\_group()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect.self_group" \o "sqlalchemy.sql.expression.GenerativeSelect.self_group) method of [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement) just returns self.

ClauseElement的基本self\_group() 方法只返回自身。

**tablesample**(*sampling*, *name=None*, *seed=None*)

*inherited from the* [tablesample()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.tablesample" \o "sqlalchemy.sql.expression.FromClause.tablesample) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a TABLESAMPLE alias of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

The return value is the [TableSample](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableSample" \o "sqlalchemy.sql.expression.TableSample) construct also provided by the top-level [tablesample()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.tablesample" \o "sqlalchemy.sql.expression.tablesample) function.

*New in version 1.1.*

**See also**

[tablesample()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.tablesample" \o "sqlalchemy.sql.expression.tablesample) - usage guidelines and parameters

**unique\_params**(*\*optionaldict*, *\*\*kwargs*)

*inherited from the* [unique\_params()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.unique_params" \o "sqlalchemy.sql.expression.ClauseElement.unique_params) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Return a copy with [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) elements replaced.

Same functionality as params(), except adds unique=True to affected bind parameters so that multiple statements can be used.

**with\_for\_update**(*nowait=False*, *read=False*, *of=None*, *skip\_locked=False*, *key\_share=False*)

Specify a FOR UPDATE clause for this [GenerativeSelect](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect" \o "sqlalchemy.sql.expression.GenerativeSelect).

E.g.:

stmt = select([table]).with\_for\_update(nowait=**True**)

On a database like PostgreSQL or Oracle, the above would render a statement like:

SELECT table.a, table.b FROM table FOR UPDATE NOWAIT

on other backends, the nowait option is ignored and instead would produce:

SELECT table.a, table.b FROM table FOR UPDATE

When called with no arguments, the statement will render with the suffix FOR UPDATE. Additional arguments can then be provided which allow for common database-specific variants.

当没有参数调用时，该语句将以后缀FOR UPDATE呈现。 然后可以提供其他参数，这些参数允许常见的数据库特定变体。

|  |  |
| --- | --- |
| **Parameters:** | * ****nowait**** – boolean; will render FOR UPDATE NOWAIT on Oracle and PostgreSQL dialects. * ****read**** – boolean; will render LOCK IN SHARE MODE on MySQL, FOR SHARE on PostgreSQL. On PostgreSQL, when combined with nowait, will render FOR SHARE NOWAIT. * ****of**** – SQL expression or list of SQL expression elements (typically [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects or a compatible expression) which will render into a FOR UPDATEOF clause; supported by PostgreSQL and Oracle. May render as a table or as a column depending on backend. * ****skip\_locked –****boolean, will render FOR UPDATE SKIP LOCKED on Oracle and PostgreSQL dialects or FOR SHARE SKIP LOCKED if read=True is also specified.   *New in version 1.1.0.*   * ****key\_share –****boolean, will render FOR NO KEY UPDATE, or if combined with read=True will render FOR KEY SHARE, on the PostgreSQL dialect.   *New in version 1.1.0.* |

*class*sqlalchemy.sql.expression.**HasCTE**

Mixin that declares a class to include CTE support.

*New in version 1.1.*

**cte**(*name=None*, *recursive=False*)

Return a new [CTE](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.CTE" \o "sqlalchemy.sql.expression.CTE), or Common Table Expression instance.

返回一个新的CTE或Common Table Expression实例。

Common table expressions are a SQL standard whereby SELECT statements can draw upon secondary statements specified along with the primary statement, using a clause called "WITH". Special semantics regarding UNION can also be employed to allow "recursive" queries, where a SELECT statement can draw upon the set of rows that have previously been selected.

公用表表达式是一种SQL标准，其中SELECT语句可以使用一个称为"WITH"的子句来使用主语句指定的辅助语句。 还可以使用有关UNION的特殊语义来允许"递归"查询，其中SELECT语句可以对先前已选择的行集合进行绘制。

CTEs can also be applied to DML constructs UPDATE, INSERT and DELETE on some databases, both as a source of CTE rows when combined with RETURNING, as well as a consumer of CTE rows.

CTE还可以应用于某些数据库上的DML构造UPDATE，INSERT和DELETE，作为与RETURNING组合的CTE行的源，以及CTE行的使用者。

SQLAlchemy detects [CTE](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.CTE" \o "sqlalchemy.sql.expression.CTE) objects, which are treated similarly to [Alias](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Alias" \o "sqlalchemy.sql.expression.Alias) objects, as special elements to be delivered to the FROM clause of the statement as well as to a WITH clause at the top of the statement.

SQLAlchemy检测到与Alias对象类似的CTE对象作为要传递到语句的FROM子句的特殊元素以及语句顶部的WITH子句。

*Changed in version 1.1:*Added support for UPDATE/INSERT/DELETE as CTE, CTEs added to UPDATE/INSERT/DELETE.

|  |  |
| --- | --- |
| **Parameters:** | * ****name**** – name given to the common table expression. Like \_FromClause.alias(), the name can be left as None in which case an anonymous symbol will be used at query compile time. * ****recursive**** – if True, will render WITH RECURSIVE. A recursive common table expression is intended to be used in conjunction with UNION ALL in order to derive rows from those already selected. |

The following examples include two from PostgreSQL's documentation at <http://www.postgresql.org/docs/current/static/queries-with.html>, as well as additional examples.

Example 1, non recursive:

**from** **sqlalchemy** **import** (Table, Column, String, Integer,

MetaData, select, func)

metadata = MetaData()

orders = Table('orders', metadata,

Column('region', String),

Column('amount', Integer),

Column('product', String),

Column('quantity', Integer))

regional\_sales = select([

orders.c.region,

func.sum(orders.c.amount).label('total\_sales')

]).group\_by(orders.c.region).cte("regional\_sales")

top\_regions = select([regional\_sales.c.region]).\

where(

regional\_sales.c.total\_sales >

select([

func.sum(regional\_sales.c.total\_sales)/10

])

).cte("top\_regions")

statement = select([

orders.c.region,

orders.c.product,

func.sum(orders.c.quantity).label("product\_units"),

func.sum(orders.c.amount).label("product\_sales")

]).where(orders.c.region.in\_(

select([top\_regions.c.region])

)).group\_by(orders.c.region, orders.c.product)

result = conn.execute(statement).fetchall()

Example 2, WITH RECURSIVE:

**from** **sqlalchemy** **import** (Table, Column, String, Integer,

MetaData, select, func)

metadata = MetaData()

parts = Table('parts', metadata,

Column('part', String),

Column('sub\_part', String),

Column('quantity', Integer),)

included\_parts = select([

parts.c.sub\_part,

parts.c.part,

parts.c.quantity]).\

where(parts.c.part=='our part').\

cte(recursive=**True**)

incl\_alias = included\_parts.alias()parts\_alias = parts.alias()included\_parts = included\_parts.union\_all(

select([

parts\_alias.c.sub\_part,

parts\_alias.c.part,

parts\_alias.c.quantity

]).

where(parts\_alias.c.part==incl\_alias.c.sub\_part))

statement = select([

included\_parts.c.sub\_part,

func.sum(included\_parts.c.quantity).

label('total\_quantity')

]).\

group\_by(included\_parts.c.sub\_part)

result = conn.execute(statement).fetchall()

Example 3, an upsert using UPDATE and INSERT with CTEs:

**from** **datetime** **import** date**from** **sqlalchemy** **import** (MetaData, Table, Column, Integer,

Date, select, literal, and\_, exists)

metadata = MetaData()

visitors = Table('visitors', metadata,

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

Column('date', Date, primary\_key=**True**),

Column('count', Integer),)

*# add 5 visitors for the product\_id == 1*product\_id = 1day = date.today()count = 5

update\_cte = (

visitors.update()

.where(and\_(visitors.c.product\_id == product\_id,

visitors.c.date == day))

.values(count=visitors.c.count + count)

.returning(literal(1))

.cte('update\_cte'))

upsert = visitors.insert().from\_select(

[visitors.c.product\_id, visitors.c.date, visitors.c.count],

select([literal(product\_id), literal(day), literal(count)])

.where(~exists(update\_cte.select())))

connection.execute(upsert)

**See also**

[orm.query.Query.cte()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.cte" \o "sqlalchemy.orm.query.Query.cte) - ORM version of [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).

*class*sqlalchemy.sql.expression.**HasPrefixes**

**prefix\_with**(*\*expr*, *\*\*kw*)

Add one or more expressions following the statement keyword, i.e. SELECT, INSERT, UPDATE, or DELETE. Generative.

This is used to support backend-specific prefix keywords such as those provided by MySQL.

E.g.:

stmt = table.insert().prefix\_with("LOW\_PRIORITY", dialect="mysql")

Multiple prefixes can be specified by multiple calls to [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).

|  |  |
| --- | --- |
| **Parameters:** | * ****\*expr**** – textual or [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement) construct which will be rendered following the INSERT, UPDATE, or DELETE keyword. * ****\*\*kw**** – A single keyword 'dialect' is accepted. This is an optional string dialect name which will limit rendering of this prefix to only that dialect. |

*class*sqlalchemy.sql.expression.**HasSuffixes**

**suffix\_with**(*\*expr*, *\*\*kw*)

Add one or more expressions following the statement as a whole.

This is used to support backend-specific suffix keywords on certain constructs.

E.g.:

stmt = select([col1, col2]).cte().suffix\_with(

"cycle empno set y\_cycle to 1 default 0", dialect="oracle")

Multiple suffixes can be specified by multiple calls to [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).

|  |  |
| --- | --- |
| **Parameters:** | * ****\*expr**** – textual or [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement) construct which will be rendered following the target clause. * ****\*\*kw**** – A single keyword 'dialect' is accepted. This is an optional string dialect name which will limit rendering of this suffix to only that dialect. |

*class*sqlalchemy.sql.expression.**Join**(*left*, *right*, *onclause=None*, *isouter=False*, *full=False*)

Bases: [sqlalchemy.sql.expression.FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

represent a JOIN construct between two [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) elements.

The public constructor function for [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) is the module-level [join()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.join" \o "sqlalchemy.sql.expression.join) function, as well as the [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) method of any [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) (e.g. such as [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)).

**See also**

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

[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)

**\_\_init\_\_**(*left*, *right*, *onclause=None*, *isouter=False*, *full=False*)

Construct a new [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join).

The usual entrypoint here is the [join()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.join" \o "sqlalchemy.sql.expression.join) function or the [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) method of any [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) object.

**alias**(*name=None*, *flat=False*)

return an alias of this [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join).

The default behavior here is to first produce a SELECT construct from this [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join), then to produce an [Alias](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Alias" \o "sqlalchemy.sql.expression.Alias) from that. So given a join of the form:

j = table\_a.join(table\_b, table\_a.c.id == table\_b.c.a\_id)

The JOIN by itself would look like:

table\_a JOIN table\_b ON table\_a.id = table\_b.a\_id

Whereas the alias of the above, j.alias(), would in a SELECT context look like:

(SELECT table\_a.id AS table\_a\_id, table\_b.id AS table\_b\_id,

table\_b.a\_id AS table\_b\_a\_id

FROM table\_a

JOIN table\_b ON table\_a.id = table\_b.a\_id) AS anon\_1

The equivalent long-hand form, given a [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) object j, is:

**from** **sqlalchemy** **import** select, alias

j = alias(

select([j.left, j.right]).\

select\_from(j).\

with\_labels(**True**).\

correlate(**False**),

name=name)

The selectable produced by [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) features the same columns as that of the two individual selectables presented under a single name - the individual columns are "auto-labeled", meaning the .c. collection of the resulting [Alias](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Alias" \o "sqlalchemy.sql.expression.Alias) represents the names of the individual columns using a<tablename>\_<columname> scheme:

j.c.table\_a\_idj.c.table\_b\_a\_id

[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) also features an alternate option for aliasing joins which produces no enclosing SELECT and does not normally apply labels to the column names. The flat=True option will call [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) against the left and right sides individually. Using this option, no new SELECT is produced; we instead, from a construct as below:

j = table\_a.join(table\_b, table\_a.c.id == table\_b.c.a\_id)j = j.alias(flat=**True**)

we get a result like this:

table\_a AS table\_a\_1 JOIN table\_b AS table\_b\_1 ONtable\_a\_1.id = table\_b\_1.a\_id

The flat=True argument is also propagated to the contained selectables, so that a composite join such as:

j = table\_a.join(

table\_b.join(table\_c,

table\_b.c.id == table\_c.c.b\_id),

table\_b.c.a\_id == table\_a.c.id

).alias(flat=**True**)

Will produce an expression like:

table\_a AS table\_a\_1 JOIN (

table\_b AS table\_b\_1 JOIN table\_c AS table\_c\_1

ON table\_b\_1.id = table\_c\_1.b\_id) ON table\_a\_1.id = table\_b\_1.a\_id

The standalone [alias()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.alias" \o "sqlalchemy.sql.expression.alias) function as well as the base [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 also support the flat=True argument as a no-op, so that the argument can be passed to the alias() method of any selectable.

*New in version 0.9.0:*Added the flat=True option to create "aliases" of joins without enclosing inside of a SELECT subquery.

|  |  |
| --- | --- |
| **Parameters:** | * ****name**** – name given to the alias. * ****flat –****if True, produce an alias of the left and right sides of this [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) and return the join of those two selectables. This produces join expression that does not include an enclosing SELECT.   *New in version 0.9.0.* |

**See also**

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

**c**

*inherited from the* [c](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.c" \o "sqlalchemy.sql.expression.FromClause.c) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

An alias for the [columns](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join.columns" \o "sqlalchemy.sql.expression.Join.columns) attribute.

**columns**

*inherited from the* [columns](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.columns" \o "sqlalchemy.sql.expression.FromClause.columns) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

A named-based collection of [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) objects maintained by this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

The [columns](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join.columns" \o "sqlalchemy.sql.expression.Join.columns), or [c](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join.c" \o "sqlalchemy.sql.expression.Join.c) collection, is the gateway to the construction of SQL expressions using table-bound or other selectable-bound columns:

select([mytable]).where(mytable.c.somecolumn == 5)

**compare**(*other*, *\*\*kw*)

*inherited from the* [compare()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.compare" \o "sqlalchemy.sql.expression.ClauseElement.compare) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Compare this ClauseElement to the given ClauseElement.

Subclasses should override the default behavior, which is a straight identity comparison.

\*\*kw are arguments consumed by subclass compare() methods and may be used to modify the criteria for comparison. (see [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement))

**compile**(*bind=None*, *dialect=None*, *\*\*kw*)

*inherited from the* [compile()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.compile" \o "sqlalchemy.sql.expression.ClauseElement.compile) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Compile this SQL expression.

The return value is a [Compiled](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Compiled" \o "sqlalchemy.engine.interfaces.Compiled) object. Calling str() or unicode() on the returned value will yield a string representation of the result. The[Compiled](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Compiled" \o "sqlalchemy.engine.interfaces.Compiled) object also can return a dictionary of bind parameter names and values using the params accessor.

|  |  |
| --- | --- |
| **Parameters:** | * ****bind**** – An Engine or Connection from which a Compiled will be acquired. This argument takes precedence over this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)'s bound engine, if any.将从其获取编译的引擎或连接。 这个参数优先于这个ClauseElement的绑定引擎（如果有的话）。 * ****column\_keys**** – Used for INSERT and UPDATE statements, a list of column names which should be present in the VALUES clause of the compiled statement. If None, all columns from the target table object are rendered.用于INSERT和UPDATE语句，应在编译语句的VALUES子句中存在列名称列表。 如果为None，则会渲染目标表对象中的所有列。 * ****dialect**** – A Dialect instance from which a Compiled will be acquired. This argument takes precedence over the bind argument as well as this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)'s bound engine, if any.将从其中获取编译的方言实例。 此参数优先于bind参数以及此ClauseElement的绑定引擎（如果有）。 * ****inline**** – Used for INSERT statements, for a dialect which does not support inline retrieval of newly generated primary key columns, will force the expression used to create the new primary key value to be rendered inline within the INSERT statement's VALUES clause. This typically refers to Sequence execution but may also refer to any server-side default generation function associated with a primary key Column.用于INSERT语句，对于不支持新生成的主键列的内联检索的方言，将强制用于创建新主键值的表达式在INSERT语句的VALUES子句中内联呈现。 这通常是指Sequence执行，但也可以指与主键Column相关联的任何服务器端默认生成函数。 * ****compile\_kwargs –****optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the literal\_binds flag through:在所有“访问”方法中将附加参数的可选字典传递给编译器。 例如，这允许将任何自定义标志传递到自定义编译构造。 它也用于通过literal\_binds标志的情况：   **from** **sqlalchemy.sql** **import** table, column, select  t = table('t', column('x'))  s = select([t]).where(t.c.x == 5)  print s.compile(compile\_kwargs={"literal\_binds": **True**})  *New in version 0.9.0.* |

**See also**

[How do I render SQL expressions as strings, possibly with bound parameters inlined?](http://docs.sqlalchemy.org/en/rel_1_1/faq/sqlexpressions.html" \l "faq-sql-expression-string)

**correspond\_on\_equivalents**(*column*, *equivalents*)

*inherited from the* [correspond\_on\_equivalents()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.correspond_on_equivalents" \o "sqlalchemy.sql.expression.FromClause.correspond_on_equivalents) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return corresponding\_column for the given column, or if None search for a match in the given dictionary.

返回给定列的对应列，或者如果None在给定字典中搜索匹配。

**corresponding\_column**(*column*, *require\_embedded=False*)

*inherited from the* [corresponding\_column()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.corresponding_column" \o "sqlalchemy.sql.expression.FromClause.corresponding_column) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Given a [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement), return the exported [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) object from this [Selectable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Selectable" \o "sqlalchemy.sql.expression.Selectable) which corresponds to that original [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) via a common ancestor column.

给定一个ColumnElement，通过一个共同的祖先列从该可选择返回导出的ColumnElement对象，该对象与该原始列相对应。

|  |  |
| --- | --- |
| **Parameters:** | * ****column**** – the target [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) to be matched * ****require\_embedded**** – only return corresponding columns for the given [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement), if the given [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) is actually present within a sub-element of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause). Normally the column will match if it merely shares a common ancestor with one of the exported columns of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).如果给定的ColumnElement实际存在于此FromClause的子元素中，则只返回给定ColumnElement的相应列。 通常，如果它仅与此FromClause的一个导出列共享一个共同的祖先，则列将匹配。 |

**~~count~~**~~(~~*~~whereclause=None~~*~~,~~*~~\*\*params~~*~~)~~

*~~inherited from the~~*~~[count()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.count" \o "sqlalchemy.sql.expression.FromClause.count)~~*~~method of~~*~~[FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)~~

~~return a SELECT COUNT generated against this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).~~

*~~Deprecated since version 1.1:~~*~~FromClause.count() is deprecated. Counting rows requires that the correct column expression and accommodations for joins, DISTINCT, etc. must be made, otherwise results may not be what's expected. Please use an appropriate func.count() expression directly.~~

~~The function generates COUNT against the first column in the primary key of the table, or against the first column in the table overall. Explicit use offunc.count() should be preferred:~~

~~row\_count = conn.scalar(~~

~~select([func.count('\*')]).select\_from(table))~~

**~~See also~~**

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

**foreign\_keys**

*inherited from the* [foreign\_keys](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.foreign_keys" \o "sqlalchemy.sql.expression.FromClause.foreign_keys) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return the collection of ForeignKey objects which this FromClause references.

**join**(*right*, *onclause=None*, *isouter=False*, *full=False*)

*inherited from the* [join()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.join" \o "sqlalchemy.sql.expression.FromClause.join) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) from this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) to another [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

E.g.:

**from** **sqlalchemy** **import** join

j = user\_table.join(address\_table,

user\_table.c.id == address\_table.c.user\_id)

stmt = select([user\_table]).select\_from(j)

would emit SQL along the lines of:

SELECT user.id, user.name FROM userJOIN address ON user.id = address.user\_id

|  |  |
| --- | --- |
| **Parameters:** | * ****right**** – the right side of the join; this is any [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) object such as a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object, and may also be a selectable-compatible object such as an ORM-mapped class.联合的右边; 这是任何FromClause对象，如Table对象，也可以是可选择兼容的对象，如ORM映射类。 * ****onclause**** – a SQL expression representing the ON clause of the join. If left at None, [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) will attempt to join the two tables based on a foreign key relationship.表示连接的ON子句的SQL表达式。 如果保持为None，则FromClause.join() 将根据外键关系尝试加入这两个表。 * ****isouter**** – if True, render a LEFT OUTER JOIN, instead of JOIN.如果为True，则呈现LEFT OUTER JOIN，而不是JOIN。 * ****full –****if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN. Implies [FromClause.join.isouter](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.join.params.isouter" \o "sqlalchemy.sql.expression.FromClause.join).如果为True，则呈现FULL OUTER JOIN，而不是LEFT OUTER JOIN。意味着FromClause.join.isouter。   *New in version 1.1.* |

**See also**

[join()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.join" \o "sqlalchemy.sql.expression.join) - standalone function

[Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) - the type of object produced

**lateral**(*name=None*)

*inherited from the* [lateral()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.lateral" \o "sqlalchemy.sql.expression.FromClause.lateral) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a LATERAL alias of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

The return value is the [Lateral](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Lateral" \o "sqlalchemy.sql.expression.Lateral) construct also provided by the top-level [lateral()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.lateral" \o "sqlalchemy.sql.expression.lateral) function.

*New in version 1.1.*

**See also**

[LATERAL correlation](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html" \l "lateral-selects) - overview of usage.

**outerjoin**(*right*, *onclause=None*, *full=False*)

*inherited from the* [outerjoin()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.outerjoin" \o "sqlalchemy.sql.expression.FromClause.outerjoin) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) from this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) to another [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause), with the "isouter" flag set to True.

E.g.:

**from** **sqlalchemy** **import** outerjoin

j = user\_table.outerjoin(address\_table,

user\_table.c.id == address\_table.c.user\_id)

The above is equivalent to:

j = user\_table.join(

address\_table,

user\_table.c.id == address\_table.c.user\_id,

isouter=**True**)

|  |  |
| --- | --- |
| **Parameters:** | * ****right**** – the right side of the join; this is any [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) object such as a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object, and may also be a selectable-compatible object such as an ORM-mapped class. * ****onclause**** – a SQL expression representing the ON clause of the join. If left at None, [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) will attempt to join the two tables based on a foreign key relationship. * ****full –****if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN.   *New in version 1.1.* |

**See also**

[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)

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

**params**(*\*optionaldict*, *\*\*kwargs*)

*inherited from the* [params()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.params" \o "sqlalchemy.sql.expression.ClauseElement.params) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Return a copy with [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) elements replaced.

Returns a copy of this ClauseElement with [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) elements replaced with values taken from the given dictionary:

**>>>** clause = column('x') + bindparam('foo')

**>>>** print clause.compile().params{'foo':None}

**>>>** print clause.params({'foo':7}).compile().params{'foo':7}

**primary\_key**

*inherited from the* [primary\_key](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.primary_key" \o "sqlalchemy.sql.expression.FromClause.primary_key) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return the collection of Column objects which comprise the primary key of this FromClause.

**replace\_selectable**(*old*, *alias*)

*inherited from the* [replace\_selectable()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.replace_selectable" \o "sqlalchemy.sql.expression.FromClause.replace_selectable) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

replace all occurrences of FromClause 'old' with the given Alias object, returning a copy of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

**select**(*whereclause=None*, *\*\*kwargs*)

Create a [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select) from this [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join).

The equivalent long-hand form, given a [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) object j, is:

**from** **sqlalchemy** **import** select

j = select([j.left, j.right], \*\*kw).\

where(whereclause).\

select\_from(j)

|  |  |
| --- | --- |
| **Parameters:** | * ****whereclause**** – the WHERE criterion that will be sent to the [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) function * ****\*\*kwargs**** – all other kwargs are sent to the underlying [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) function. |

**tablesample**(*sampling*, *name=None*, *seed=None*)

*inherited from the* [tablesample()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.tablesample" \o "sqlalchemy.sql.expression.FromClause.tablesample) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a TABLESAMPLE alias of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

The return value is the [TableSample](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableSample" \o "sqlalchemy.sql.expression.TableSample) construct also provided by the top-level [tablesample()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.tablesample" \o "sqlalchemy.sql.expression.tablesample) function.

*New in version 1.1.*

**See also**

[tablesample()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.tablesample" \o "sqlalchemy.sql.expression.tablesample) - usage guidelines and parameters

**unique\_params**(*\*optionaldict*, *\*\*kwargs*)

*inherited from the* [unique\_params()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.unique_params" \o "sqlalchemy.sql.expression.ClauseElement.unique_params) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Return a copy with [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) elements replaced.

Same functionality as params(), except adds unique=True to affected bind parameters so that multiple statements can be used.

*class*sqlalchemy.sql.expression.**Lateral**(*selectable*, *name=None*)

Bases: [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)

Represent a LATERAL subquery.

This object is constructed from the [lateral()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.lateral" \o "sqlalchemy.sql.expression.lateral) module level function as well as the [FromClause.lateral()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.lateral" \o "sqlalchemy.sql.expression.FromClause.lateral) method available on all [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)subclasses.

While LATERAL is part of the SQL standard, curently only more recent PostgreSQL versions provide support for this keyword.

*New in version 1.1.*

**See also**

[LATERAL correlation](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html" \l "lateral-selects) - overview of usage.

**alias**(*name=None*, *flat=False*)

*inherited from the* [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 of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

return an alias of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

This is shorthand for calling:

**from** **sqlalchemy** **import** alias

a = alias(self, name=name)

See [alias()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.alias" \o "sqlalchemy.sql.expression.alias) for details.

**c**

*inherited from the* [c](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.c" \o "sqlalchemy.sql.expression.FromClause.c) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

An alias for the [columns](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Lateral.columns" \o "sqlalchemy.sql.expression.Lateral.columns) attribute.

**columns**

*inherited from the* [columns](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.columns" \o "sqlalchemy.sql.expression.FromClause.columns) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

A named-based collection of [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) objects maintained by this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

The [columns](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Lateral.columns" \o "sqlalchemy.sql.expression.Lateral.columns), or [c](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Lateral.c" \o "sqlalchemy.sql.expression.Lateral.c) collection, is the gateway to the construction of SQL expressions using table-bound or other selectable-bound columns:

select([mytable]).where(mytable.c.somecolumn == 5)

**compare**(*other*, *\*\*kw*)

*inherited from the* [compare()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.compare" \o "sqlalchemy.sql.expression.ClauseElement.compare) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Compare this ClauseElement to the given ClauseElement.

Subclasses should override the default behavior, which is a straight identity comparison.

\*\*kw are arguments consumed by subclass compare() methods and may be used to modify the criteria for comparison. (see [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement))

**compile**(*bind=None*, *dialect=None*, *\*\*kw*)

*inherited from the* [compile()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.compile" \o "sqlalchemy.sql.expression.ClauseElement.compile) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Compile this SQL expression.

The return value is a [Compiled](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Compiled" \o "sqlalchemy.engine.interfaces.Compiled) object. Calling str() or unicode() on the returned value will yield a string representation of the result. The[Compiled](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Compiled" \o "sqlalchemy.engine.interfaces.Compiled) object also can return a dictionary of bind parameter names and values using the params accessor.

|  |  |
| --- | --- |
| **Parameters:** | * ****bind**** – An Engine or Connection from which a Compiled will be acquired. This argument takes precedence over this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)'s bound engine, if any. * ****column\_keys**** – Used for INSERT and UPDATE statements, a list of column names which should be present in the VALUES clause of the compiled statement. If None, all columns from the target table object are rendered. * ****dialect**** – A Dialect instance from which a Compiled will be acquired. This argument takes precedence over the bind argument as well as this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)'s bound engine, if any. * ****inline**** – Used for INSERT statements, for a dialect which does not support inline retrieval of newly generated primary key columns, will force the expression used to create the new primary key value to be rendered inline within the INSERT statement's VALUES clause. This typically refers to Sequence execution but may also refer to any server-side default generation function associated with a primary key Column. * ****compile\_kwargs –****optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the literal\_binds flag through:在所有“访问”方法中将附加参数的可选字典传递给编译器。 例如，这允许将任何自定义标志传递到自定义编译构造。 它也用于通过literal\_binds标志的情况：   **from** **sqlalchemy.sql** **import** table, column, select  t = table('t', column('x'))  s = select([t]).where(t.c.x == 5)  print s.compile(compile\_kwargs={"literal\_binds": **True**})  *New in version 0.9.0.* |

**See also**

[How do I render SQL expressions as strings, possibly with bound parameters inlined?](http://docs.sqlalchemy.org/en/rel_1_1/faq/sqlexpressions.html" \l "faq-sql-expression-string)

**correspond\_on\_equivalents**(*column*, *equivalents*)

*inherited from the* [correspond\_on\_equivalents()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.correspond_on_equivalents" \o "sqlalchemy.sql.expression.FromClause.correspond_on_equivalents) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return corresponding\_column for the given column, or if None search for a match in the given dictionary.

**corresponding\_column**(*column*, *require\_embedded=False*)

*inherited from the* [corresponding\_column()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.corresponding_column" \o "sqlalchemy.sql.expression.FromClause.corresponding_column) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Given a [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement), return the exported [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) object from this [Selectable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Selectable" \o "sqlalchemy.sql.expression.Selectable) which corresponds to that original [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) via a common ancestor column.

|  |  |
| --- | --- |
| **Parameters:** | * ****column**** – the target [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) to be matched * ****require\_embedded**** – only return corresponding columns for the given [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement), if the given [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) is actually present within a sub-element of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause). Normally the column will match if it merely shares a common ancestor with one of the exported columns of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause). |

**count**(*whereclause=None*, *\*\*params*)

*inherited from the* [count()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.count" \o "sqlalchemy.sql.expression.FromClause.count) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

return a SELECT COUNT generated against this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

*Deprecated since version 1.1:*FromClause.count() is deprecated. Counting rows requires that the correct column expression and accommodations for joins, DISTINCT, etc. must be made, otherwise results may not be what's expected. Please use an appropriate func.count() expression directly.

The function generates COUNT against the first column in the primary key of the table, or against the first column in the table overall. Explicit use offunc.count() should be preferred:

row\_count = conn.scalar(

select([func.count('\*')]).select\_from(table))

**See also**

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

**foreign\_keys**

*inherited from the* [foreign\_keys](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.foreign_keys" \o "sqlalchemy.sql.expression.FromClause.foreign_keys) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return the collection of ForeignKey objects which this FromClause references.

**join**(*right*, *onclause=None*, *isouter=False*, *full=False*)

*inherited from the* [join()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.join" \o "sqlalchemy.sql.expression.FromClause.join) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) from this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) to another [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

E.g.:

**from** **sqlalchemy** **import** join

j = user\_table.join(address\_table,

user\_table.c.id == address\_table.c.user\_id)stmt = select([user\_table]).select\_from(j)

would emit SQL along the lines of:

SELECT user.id, user.name FROM user

JOIN address ON user.id = address.user\_id

|  |  |
| --- | --- |
| **Parameters:** | * ****right**** – the right side of the join; this is any [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) object such as a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object, and may also be a selectable-compatible object such as an ORM-mapped class. * ****onclause**** – a SQL expression representing the ON clause of the join. If left at None, [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) will attempt to join the two tables based on a foreign key relationship. * ****isouter**** – if True, render a LEFT OUTER JOIN, instead of JOIN. * ****full –****if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN. Implies [FromClause.join.isouter](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.join.params.isouter" \o "sqlalchemy.sql.expression.FromClause.join).   *New in version 1.1.* |

**See also**

[join()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.join" \o "sqlalchemy.sql.expression.join) - standalone function

[Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) - the type of object produced

**lateral**(*name=None*)

*inherited from the* [lateral()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.lateral" \o "sqlalchemy.sql.expression.FromClause.lateral) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a LATERAL alias of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

The return value is the [Lateral](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Lateral" \o "sqlalchemy.sql.expression.Lateral) construct also provided by the top-level [lateral()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.lateral" \o "sqlalchemy.sql.expression.lateral) function.

*New in version 1.1.*

**See also**

[LATERAL correlation](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html" \l "lateral-selects) - overview of usage.

**outerjoin**(*right*, *onclause=None*, *full=False*)

*inherited from the* [outerjoin()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.outerjoin" \o "sqlalchemy.sql.expression.FromClause.outerjoin) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) from this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) to another [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause), with the "isouter" flag set to True.

E.g.:

**from** **sqlalchemy** **import** outerjoin

j = user\_table.outerjoin(address\_table,

user\_table.c.id == address\_table.c.user\_id)

The above is equivalent to:

j = user\_table.join(

address\_table,

user\_table.c.id == address\_table.c.user\_id,

isouter=**True**)

|  |  |
| --- | --- |
| **Parameters:** | * ****right**** – the right side of the join; this is any [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) object such as a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object, and may also be a selectable-compatible object such as an ORM-mapped class. * ****onclause**** – a SQL expression representing the ON clause of the join. If left at None, [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) will attempt to join the two tables based on a foreign key relationship. * ****full –****if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN.   *New in version 1.1.* |

**See also**

[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)

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

**params**(*\*optionaldict*, *\*\*kwargs*)

*inherited from the* [params()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.params" \o "sqlalchemy.sql.expression.ClauseElement.params) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Return a copy with [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) elements replaced.

Returns a copy of this ClauseElement with [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) elements replaced with values taken from the given dictionary:

**>>>** clause = column('x') + bindparam('foo')

**>>>** print clause.compile().params{'foo':None}

**>>>** print clause.params({'foo':7}).compile().params{'foo':7}

**primary\_key**

*inherited from the* [primary\_key](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.primary_key" \o "sqlalchemy.sql.expression.FromClause.primary_key) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return the collection of Column objects which comprise the primary key of this FromClause.

**replace\_selectable**(*old*, *alias*)

*inherited from the* [replace\_selectable()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.replace_selectable" \o "sqlalchemy.sql.expression.FromClause.replace_selectable) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

replace all occurrences of FromClause 'old' with the given Alias object, returning a copy of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

**select**(*whereclause=None*, *\*\*params*)

*inherited from the* [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.select" \o "sqlalchemy.sql.expression.FromClause.select) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

return a SELECT of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

**See also**

[select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) - general purpose method which allows for arbitrary column lists.

**tablesample**(*sampling*, *name=None*, *seed=None*)

*inherited from the* [tablesample()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.tablesample" \o "sqlalchemy.sql.expression.FromClause.tablesample) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a TABLESAMPLE alias of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

The return value is the [TableSample](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableSample" \o "sqlalchemy.sql.expression.TableSample) construct also provided by the top-level [tablesample()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.tablesample" \o "sqlalchemy.sql.expression.tablesample) function.

*New in version 1.1.*

**See also**

[tablesample()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.tablesample" \o "sqlalchemy.sql.expression.tablesample) - usage guidelines and parameters

**unique\_params**(*\*optionaldict*, *\*\*kwargs*)

*inherited from the* [unique\_params()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.unique_params" \o "sqlalchemy.sql.expression.ClauseElement.unique_params) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Return a copy with [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) elements replaced.

Same functionality as params(), except adds unique=True to affected bind parameters so that multiple statements can be used.

*class*sqlalchemy.sql.expression.**ScalarSelect**(*element*)

Bases: sqlalchemy.sql.expression.Generative, sqlalchemy.sql.expression.Grouping

**where**(*crit*)

Apply a WHERE clause to the SELECT statement referred to by this [ScalarSelect](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.ScalarSelect" \o "sqlalchemy.sql.expression.ScalarSelect).

*class*sqlalchemy.sql.expression.**Select**(*columns=None*, *whereclause=None*, *from\_obj=None*, *distinct=False*, *having=None*, *correlate=True*, *prefixes=None*, *suffixes=None*, *\*\*kwargs*)

Bases: [sqlalchemy.sql.expression.HasPrefixes](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.HasPrefixes" \o "sqlalchemy.sql.expression.HasPrefixes), [sqlalchemy.sql.expression.HasSuffixes](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.HasSuffixes" \o "sqlalchemy.sql.expression.HasSuffixes), [sqlalchemy.sql.expression.GenerativeSelect](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect" \o "sqlalchemy.sql.expression.GenerativeSelect)

Represents a SELECT statement.

**\_\_init\_\_**(*columns=None*, *whereclause=None*, *from\_obj=None*, *distinct=False*, *having=None*, *correlate=True*, *prefixes=None*, *suffixes=None*, *\*\*kwargs*)

Construct a new [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select) object.

This constructor is mirrored as a public API function; see [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) for a full usage and argument description.

**alias**(*name=None*, *flat=False*)

*inherited from the* [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 of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

return an alias of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

This is shorthand for calling:

**from** **sqlalchemy** **import** alias

a = alias(self, name=name)

See [alias()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.alias" \o "sqlalchemy.sql.expression.alias) for details.

**append\_column**(*column*)

append the given column expression to the columns clause of this select() construct.

E.g.:

my\_select.append\_column(some\_table.c.new\_column)

This is an ****in-place**** mutation method; the [column()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.column" \o "sqlalchemy.sql.expression.Select.column) method is preferred, as it provides standard [method chaining](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-method-chaining).

See the documentation for [Select.with\_only\_columns()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.with_only_columns" \o "sqlalchemy.sql.expression.Select.with_only_columns) for guidelines on adding /replacing the columns of a [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select) object.

**append\_correlation**(*fromclause*)

append the given correlation expression to this select() construct.

This is an ****in-place**** mutation method; the [correlate()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.correlate" \o "sqlalchemy.sql.expression.Select.correlate) method is preferred, as it provides standard [method chaining](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-method-chaining).

**append\_from**(*fromclause*)

append the given FromClause expression to this select() construct's FROM clause.

This is an ****in-place**** mutation method; the [select\_from()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.select_from" \o "sqlalchemy.sql.expression.Select.select_from) method is preferred, as it provides standard [method chaining](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-method-chaining).

**append\_group\_by**(*\*clauses*)

*inherited from the* [append\_group\_by()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect.append_group_by" \o "sqlalchemy.sql.expression.GenerativeSelect.append_group_by) *method of* [GenerativeSelect](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect" \o "sqlalchemy.sql.expression.GenerativeSelect)

Append the given GROUP BY criterion applied to this selectable.

The criterion will be appended to any pre-existing GROUP BY criterion.

This is an ****in-place**** mutation method; the [group\_by()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect.group_by" \o "sqlalchemy.sql.expression.GenerativeSelect.group_by) method is preferred, as it provides standard [method chaining](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-method-chaining).

**append\_having**(*having*)

append the given expression to this select() construct's HAVING criterion.

The expression will be joined to existing HAVING criterion via AND.

This is an ****in-place**** mutation method; the [having()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.having" \o "sqlalchemy.sql.expression.Select.having) method is preferred, as it provides standard [method chaining](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-method-chaining).

**append\_order\_by**(*\*clauses*)

*inherited from the* [append\_order\_by()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect.append_order_by" \o "sqlalchemy.sql.expression.GenerativeSelect.append_order_by) *method of* [GenerativeSelect](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect" \o "sqlalchemy.sql.expression.GenerativeSelect)

Append the given ORDER BY criterion applied to this selectable.

The criterion will be appended to any pre-existing ORDER BY criterion.

This is an ****in-place**** mutation method; the [order\_by()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect.order_by" \o "sqlalchemy.sql.expression.GenerativeSelect.order_by) method is preferred, as it provides standard [method chaining](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-method-chaining).

**append\_prefix**(*clause*)

append the given columns clause prefix expression to this select() construct.

This is an ****in-place**** mutation method; the [prefix\_with()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.prefix_with" \o "sqlalchemy.sql.expression.Select.prefix_with) method is preferred, as it provides standard [method chaining](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-method-chaining).

**append\_whereclause**(*whereclause*)

append the given expression to this select() construct's WHERE criterion.

将给定的表达式附加到这个select() 结构的WHERE标准。

The expression will be joined to existing WHERE criterion via AND.

该表达式将通过AND连接到现有的WHERE标准。

This is an ****in-place**** mutation method; the [where()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.where" \o "sqlalchemy.sql.expression.Select.where) method is preferred, as it provides standard [method chaining](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-method-chaining).

这是一种就地突变方法; where() 方法是首选的，因为它提供了标准的方法链接。

**apply\_labels**()

*inherited from the* [apply\_labels()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect.apply_labels" \o "sqlalchemy.sql.expression.GenerativeSelect.apply_labels) *method of* [GenerativeSelect](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect" \o "sqlalchemy.sql.expression.GenerativeSelect)

return a new selectable with the 'use\_labels' flag set to True.

返回一个新的可选项，将'use\_labels'标志设置为True。

This will result in column expressions being generated using labels against their table name, such as "SELECT somecolumn AS tablename\_somecolumn". This allows selectables which contain multiple FROM clauses to produce a unique set of column names regardless of name conflicts among the individual FROM clauses.

这将导致使用与其表名称相关的标签生成列表达式，例如“SELECT somecolumn AS tablename\_somecolumn”。 这允许包含多个FROM子句的可选项生成一组唯一的列名称，而不管各个FROM子句之间是否有冲突。

**as\_scalar**()

*inherited from the* [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) *method of* [SelectBase](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.SelectBase" \o "sqlalchemy.sql.expression.SelectBase)

return a 'scalar' representation of this selectable, which can be used as a column expression.

返回此可选择的“标量”表示，可用作列表达式。

Typically, a select statement which has only one column in its columns clause is eligible to be used as a scalar expression.

通常，在其column子句中只有一列的select语句有资格用作标量表达式。

The returned object is an instance of [ScalarSelect](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.ScalarSelect" \o "sqlalchemy.sql.expression.ScalarSelect).

返回的对象是ScalarSelect的一个实例。

**~~autocommit~~**~~()~~

*~~inherited from the~~*~~[autocommit()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.SelectBase.autocommit" \o "sqlalchemy.sql.expression.SelectBase.autocommit)~~*~~method of~~*~~[SelectBase](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.SelectBase" \o "sqlalchemy.sql.expression.SelectBase)~~

~~return a new selectable with the 'autocommit' flag set to True.~~

*~~Deprecated since version 0.6:~~*~~autocommit() is deprecated.~~ Use [Executable.execution\_options()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable.execution_options" \o "sqlalchemy.sql.expression.Executable.execution_options) with the 'autocommit' flag.

**c**

*inherited from the* [c](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.c" \o "sqlalchemy.sql.expression.FromClause.c) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

An alias for the [columns](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.columns" \o "sqlalchemy.sql.expression.Select.columns) attribute.

**column**(*column*)

return a new select() construct with the given column expression added to its columns clause.

E.g.:

my\_select = my\_select.column(table.c.new\_column)

See the documentation for [Select.with\_only\_columns()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.with_only_columns" \o "sqlalchemy.sql.expression.Select.with_only_columns) for guidelines on adding /replacing the columns of a [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select) object.

**columns**

*inherited from the* [columns](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.columns" \o "sqlalchemy.sql.expression.FromClause.columns) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

A named-based collection of [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) objects maintained by this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

The [columns](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.columns" \o "sqlalchemy.sql.expression.Select.columns), or [c](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.c" \o "sqlalchemy.sql.expression.Select.c) collection, is the gateway to the construction of SQL expressions using table-bound or other selectable-bound columns:

select([mytable]).where(mytable.c.somecolumn == 5)

**compare**(*other*, *\*\*kw*)

*inherited from the* [compare()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.compare" \o "sqlalchemy.sql.expression.ClauseElement.compare) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Compare this ClauseElement to the given ClauseElement.

Subclasses should override the default behavior, which is a straight identity comparison.

\*\*kw are arguments consumed by subclass compare() methods and may be used to modify the criteria for comparison. (see [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement))

**compile**(*bind=None*, *dialect=None*, *\*\*kw*)

*inherited from the* [compile()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.compile" \o "sqlalchemy.sql.expression.ClauseElement.compile) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Compile this SQL expression.

编译此SQL表达式。

The return value is a [Compiled](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Compiled" \o "sqlalchemy.engine.interfaces.Compiled) object. Calling str() or unicode() on the returned value will yield a string representation of the result. The[Compiled](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Compiled" \o "sqlalchemy.engine.interfaces.Compiled) object also can return a dictionary of bind parameter names and values using the params accessor.

返回值是一个编译对象。 对返回的值调用str() 或unicode() 将产生结果的字符串表示形式。 Compiled对象还可以使用params访问器返回绑定参数名称和值的字典。

|  |  |
| --- | --- |
| **Parameters:** | * ****bind**** – An Engine or Connection from which a Compiled will be acquired. This argument takes precedence over this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)'s bound engine, if any.将从其获取编译的引擎或连接。 这个参数优先于这个ClauseElement的绑定引擎（如果有的话）。 * ****column\_keys**** – Used for INSERT and UPDATE statements, a list of column names which should be present in the VALUES clause of the compiled statement. If None, all columns from the target table object are rendered.用于INSERT和UPDATE语句，应在编译语句的VALUES子句中存在列名称列表。 如果为None，则会渲染目标表对象中的所有列。 * ****dialect**** – A Dialect instance from which a Compiled will be acquired. This argument takes precedence over the bind argument as well as this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)'s bound engine, if any.将从其中获取编译的方言实例。 此参数优先于bind参数以及此ClauseElement的绑定引擎（如果有）。 * ****inline**** – Used for INSERT statements, for a dialect which does not support inline retrieval of newly generated primary key columns, will force the expression used to create the new primary key value to be rendered inline within the INSERT statement's VALUES clause. This typically refers to Sequence execution but may also refer to any server-side default generation function associated with a primary key Column.用于INSERT语句，对于不支持新生成的主键列的内联检索的方言，将强制用于创建新主键值的表达式在INSERT语句的VALUES子句中内联呈现。 这通常是指Sequence执行，但也可以指与主键Column相关联的任何服务器端默认生成函数。 * ****compile\_kwargs –****optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the literal\_binds flag through:在所有“访问”方法中将附加参数的可选字典传递给编译器。 例如，这允许将任何自定义标志传递到自定义编译构造。 它也用于通过literal\_binds标志的情况：   **from** **sqlalchemy.sql** **import** table, column, select  t = table('t', column('x'))  s = select([t]).where(t.c.x == 5)  print s.compile(compile\_kwargs={"literal\_binds": **True**})  *New in version 0.9.0.* |

**See also**

[How do I render SQL expressions as strings, possibly with bound parameters inlined?](http://docs.sqlalchemy.org/en/rel_1_1/faq/sqlexpressions.html" \l "faq-sql-expression-string)

**correlate**(*\*fromclauses*)

return a new [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select) which will correlate the given FROM clauses to that of an enclosing [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select).

返回一个新的Select，它将给定的FROM子句与封闭的Select关联。

Calling this method turns off the [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select) object's default behavior of "auto-correlation". Normally, FROM elements which appear in a [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select) that encloses this one via its [WHERE clause](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-where-clause), ORDER BY, HAVING or [columns clause](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-columns-clause) will be omitted from this [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select) object's [FROM clause](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-from-clause). Setting an explicit correlation collection using the [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) method provides a fixed list of FROM objects that can potentially take place in this process.

调用此方法将关闭“选择”对象的默认行为“自动相关”。通常，通过其WHERE子句包含在其中的Select中的FROM元素，将从该Select对象的FROM子句中省略ORDER BY，HAVING或columns子句。使用Select.correlate() 方法设置一个明确的关联集合提供了一个在此过程中可能发生的FROM对象的固定列表。

When [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) is used to apply specific FROM clauses for correlation, the FROM elements become candidates for correlation regardless of how deeply nested this [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select) object is, relative to an enclosing [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select) which refers to the same FROM object. This is in contrast to the behavior of "auto-correlation" which only correlates to an immediate enclosing [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select). Multi-level correlation ensures that the link between enclosed and enclosing [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select) is always via at least one WHERE/ORDER BY/HAVING/columns clause in order for correlation to take place.

当使用Select.correlate() 来应用特定的FROM子句进行关联时，FROM元素成为相关的候选者，无论这个Select对象是多么嵌套，相对于引用相同FROM对象的封闭Select。这与“自动相关”的行为相反，“自动相关”仅与紧邻的“选择”相关。多级关联确保封闭和封闭Select之间的链接始终通过至少一个WHERE / ORDER BY / HAVING / columns子句，以便进行相关。

If None is passed, the [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select) object will correlate none of its FROM entries, and all will render unconditionally in the local FROM clause.

如果没有传递，则Select对象将不关联它的任何FROM条目，并且所有的都将无条件地在本地FROM子句中呈现。

|  |  |
| --- | --- |
| **Parameters:** | ****\*fromclauses**** –a list of one or more [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) constructs, or other compatible constructs (i.e. ORM-mapped classes) to become part of the correlate collection.  *Changed in version 0.8.0:*ORM-mapped classes are accepted by [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). |

*Changed in version 0.8.0:*The [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) method no longer unconditionally removes entries from the FROM clause; instead, the candidate FROM entries must also be matched by a FROM entry located in an enclosing [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select), which ultimately encloses this one as present in the WHERE clause, ORDER BY clause, HAVING clause, or columns clause of an enclosing Select().

在版本0.8.0中更改：Select.correlate() 方法不再无条件地从FROM子句中删除条目; 相反，候选FROM条目也必须由位于封闭Select中的FROM条目匹配，最终将其包含在WHERE子句，ORDER BY子句，HAVING子句或附加的Select() 的列子句中。

*Changed in version 0.8.2:*explicit correlation takes place via any level of nesting of [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select) objects; in previous 0.8 versions, correlation would only occur relative to the immediate enclosing [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select) construct.

在版本0.8.2中更改：显式关联通过任意级别的嵌套选择对象进行; 在之前的0.8版本中，相关性只会相对于直接包围的Select构造发生。

**See also**

[Select.correlate\_except()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.correlate_except" \o "sqlalchemy.sql.expression.Select.correlate_except)

[Correlated Subqueries](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html" \l "correlated-subqueries)

**correlate\_except**(*\*fromclauses*)

return a new [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select) which will omit the given FROM clauses from the auto-correlation process.

返回一个新的Select，它将从自动相关过程中省略给定的FROM子句。

Calling [Select.correlate\_except()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.correlate_except" \o "sqlalchemy.sql.expression.Select.correlate_except) turns off the [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select) object's default behavior of "auto-correlation" for the given FROM elements. An element specified here will unconditionally appear in the FROM list, while all other FROM elements remain subject to normal auto-correlation behaviors.

调用Select.correlate\_except() 关闭给定FROM元素的Select对象的默认行为“auto-correlation”。 这里指定的元素将无条件地出现在FROM列表中，而所有其他FROM元素仍然保持正常的自相关行为。

*Changed in version 0.8.2:*The [Select.correlate\_except()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.correlate_except" \o "sqlalchemy.sql.expression.Select.correlate_except) method was improved to fully prevent FROM clauses specified here from being omitted from the immediate 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).

在版本0.8.2中更改：改进了Select.correlate\_except() 方法，以完全阻止此Select指定的FROM FROM子句中省略的FROM子句。

If None is passed, the [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select) object will correlate all of its FROM entries.

如果None传递，则Select对象将关联其所有FROM条目。

*Changed in version 0.8.2:*calling correlate\_except(None) will correctly auto-correlate all FROM clauses.

更改版本0.8.2：调用correlate\_except（None）将正确地自动关联所有FROM子句。

|  |  |
| --- | --- |
| **Parameters:** | ****\*fromclauses**** – a list of one or more [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) constructs, or other compatible constructs (i.e. ORM-mapped classes) to become part of the correlate-exception collection.一个或多个FromClause结构或其他兼容结构（即ORM映射类）的列表，以成为关联异常集合的一部分。 |

**See also**

[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)

[Correlated Subqueries](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html" \l "correlated-subqueries)

**correspond\_on\_equivalents**(*column*, *equivalents*)

*inherited from the* [correspond\_on\_equivalents()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.correspond_on_equivalents" \o "sqlalchemy.sql.expression.FromClause.correspond_on_equivalents) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return corresponding\_column for the given column, or if None search for a match in the given dictionary.

返回给定列的对应列，或者如果None在给定字典中搜索匹配。

**corresponding\_column**(*column*, *require\_embedded=False*)

*inherited from the* [corresponding\_column()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.corresponding_column" \o "sqlalchemy.sql.expression.FromClause.corresponding_column) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Given a [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement), return the exported [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) object from this [Selectable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Selectable" \o "sqlalchemy.sql.expression.Selectable) which corresponds to that original [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) via a common ancestor column.

给定一个ColumnElement，通过一个共同的祖先列从该可选择返回导出的ColumnElement对象，该对象与该原始列相对应。

|  |  |
| --- | --- |
| **Parameters:** | * ****column**** – the target [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) to be matched * ****require\_embedded**** – only return corresponding columns for the given [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement), if the given [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) is actually present within a sub-element of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause). Normally the column will match if it merely shares a common ancestor with one of the exported columns of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause). |

**~~count~~**~~(~~*~~whereclause=None~~*~~,~~*~~\*\*params~~*~~)~~

*~~inherited from the~~*~~[count()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.count" \o "sqlalchemy.sql.expression.FromClause.count)~~*~~method of~~*~~[FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)~~

~~return a SELECT COUNT generated against this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).~~

*~~Deprecated since version 1.1:~~*~~FromClause.count() is deprecated.~~ Counting rows requires that the correct column expression and accommodations for joins, DISTINCT, etc. must be made, otherwise results may not be what's expected. Please use an appropriate func.count() expression directly.

The function generates COUNT against the first column in the primary key of the table, or against the first column in the table overall. Explicit use offunc.count() should be preferred:

该函数针对表的主键中的第一列或整个表中的第一列生成COUNT。 显式使用func.count() 应该是首选的：

row\_count = conn.scalar(

select([func.count('\*')]).select\_from(table))

**See also**

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

**cte**(*name=None*, *recursive=False*)

*inherited from the* [cte()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.HasCTE.cte" \o "sqlalchemy.sql.expression.HasCTE.cte) *method of* [HasCTE](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.HasCTE" \o "sqlalchemy.sql.expression.HasCTE)

Return a new [CTE](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.CTE" \o "sqlalchemy.sql.expression.CTE), or Common Table Expression instance.

返回一个新的CTE或Common Table Expression实例。

Common table expressions are a SQL standard whereby SELECT statements can draw upon secondary statements specified along with the primary statement, using a clause called "WITH". Special semantics regarding UNION can also be employed to allow "recursive" queries, where a SELECT statement can draw upon the set of rows that have previously been selected.

公用表表达式是一种SQL标准，其中SELECT语句可以使用一个称为“WITH”的子句来使用主语句指定的辅助语句。 还可以使用有关UNION的特殊语义来允许“递归”查询，其中SELECT语句可以对先前已选择的行集合进行绘制。

CTEs can also be applied to DML constructs UPDATE, INSERT and DELETE on some databases, both as a source of CTE rows when combined with RETURNING, as well as a consumer of CTE rows.

CTE还可以应用于某些数据库上的DML构造UPDATE，INSERT和DELETE，作为与RETURNING组合的CTE行的源，以及CTE行的使用者。

SQLAlchemy detects [CTE](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.CTE" \o "sqlalchemy.sql.expression.CTE) objects, which are treated similarly to [Alias](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Alias" \o "sqlalchemy.sql.expression.Alias) objects, as special elements to be delivered to the FROM clause of the statement as well as to a WITH clause at the top of the statement.

SQLAlchemy检测到与Alias对象类似的CTE对象作为要传递到语句的FROM子句的特殊元素以及语句顶部的WITH子句。

*Changed in version 1.1:*Added support for UPDATE/INSERT/DELETE as CTE, CTEs added to UPDATE/INSERT/DELETE.

在版本1.1中更改：添加对UPDATE / INSERT / DELETE作为CTE的支持，CTE添加到UPDATE / INSERT / DELETE。

|  |  |
| --- | --- |
| **Parameters:** | * ****name**** – name given to the common table expression. Like \_FromClause.alias(), the name can be left as None in which case an anonymous symbol will be used at query compile time.给予公用表表达式的名称。 像 \_FromClause.alias()一样，名称可以保留为None，在这种情况下，在查询编译时将使用匿名符号。 * ****recursive**** – if True, will render WITH RECURSIVE. A recursive common table expression is intended to be used in conjunction with UNION ALL in order to derive rows from those already selected.如果为True，将显示为RECURSIVE。 递归公用表表达式旨在与UNION ALL结合使用，以便从已经选择的表中导出行。 |

The following examples include two from PostgreSQL's documentation at <http://www.postgresql.org/docs/current/static/queries-with.html>, as well as additional examples.

以下示例包括PostgreSQL在http://www.postgresql.org/docs/current/static/queries-with.html上的两个文档，以及其他示例。

Example 1, non recursive:

**from** **sqlalchemy** **import** (Table, Column, String, Integer,

MetaData, select, func)

metadata = MetaData()

orders = Table('orders', metadata,

Column('region', String),

Column('amount', Integer),

Column('product', String),

Column('quantity', Integer))

regional\_sales = select([

orders.c.region,

func.sum(orders.c.amount).label('total\_sales')

]).group\_by(orders.c.region).cte("regional\_sales")

top\_regions = select([regional\_sales.c.region]).\

where(

regional\_sales.c.total\_sales >

select([

func.sum(regional\_sales.c.total\_sales)/10

])

).cte("top\_regions")

statement = select([

orders.c.region,

orders.c.product,

func.sum(orders.c.quantity).label("product\_units"),

func.sum(orders.c.amount).label("product\_sales")

]).where(orders.c.region.in\_(

select([top\_regions.c.region])

)).group\_by(orders.c.region, orders.c.product)

result = conn.execute(statement).fetchall()

Example 2, WITH RECURSIVE:

**from** **sqlalchemy** **import** (Table, Column, String, Integer,

MetaData, select, func)

metadata = MetaData()

parts = Table('parts', metadata,

Column('part', String),

Column('sub\_part', String),

Column('quantity', Integer),)

included\_parts = select([

parts.c.sub\_part,

parts.c.part,

parts.c.quantity]).\

where(parts.c.part=='our part').\

cte(recursive=**True**)

incl\_alias = included\_parts.alias()parts\_alias = parts.alias()included\_parts = included\_parts.union\_all(

select([

parts\_alias.c.sub\_part,

parts\_alias.c.part,

parts\_alias.c.quantity

]).

where(parts\_alias.c.part==incl\_alias.c.sub\_part))

statement = select([

included\_parts.c.sub\_part,

func.sum(included\_parts.c.quantity).

label('total\_quantity')

]).\

group\_by(included\_parts.c.sub\_part)

result = conn.execute(statement).fetchall()

Example 3, an upsert using UPDATE and INSERT with CTEs:

示例3，使用UPDATE和INSERT与CTE进行升级：

**from** **datetime** **import** date

**from** **sqlalchemy** **import** (MetaData, Table, Column, Integer,

Date, select, literal, and\_, exists)

metadata = MetaData()

visitors = Table('visitors', metadata,

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

Column('date', Date, primary\_key=**True**),

Column('count', Integer),)

*# add 5 visitors for the product\_id == 1*product\_id = 1day = date.today()count = 5

update\_cte = (

visitors.update()

.where(and\_(visitors.c.product\_id == product\_id,

visitors.c.date == day))

.values(count=visitors.c.count + count)

.returning(literal(1))

.cte('update\_cte'))

upsert = visitors.insert().from\_select(

[visitors.c.product\_id, visitors.c.date, visitors.c.count],

select([literal(product\_id), literal(day), literal(count)])

.where(~exists(update\_cte.select())))

connection.execute(upsert)

**See also**

[orm.query.Query.cte()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.cte" \o "sqlalchemy.orm.query.Query.cte) - ORM version of [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).

**description**

*inherited from the* [description](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.description" \o "sqlalchemy.sql.expression.FromClause.description) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

a brief description of this FromClause.

这个FromClause的简要说明。

Used primarily for error message formatting.

主要用于错误消息格式化。

**distinct**(*\*expr*)

Return a new select() construct which will apply DISTINCT to its columns clause.

返回一个新的select()结构，它将对其子句应用DISTINCT。

|  |  |
| --- | --- |
| **Parameters:** | ****\*expr**** – optional column expressions. When present, the PostgreSQL dialect will render a DISTINCT ON (<expressions>>) construct.可选列表达式。 当存在时，PostgreSQL方言将呈现DISTINCT ON（<表达式>>）构造。 |

**except\_**(*other*, *\*\*kwargs*)

return a SQL EXCEPT of this select() construct against the given selectable.

根据给定的可选择返回此select（）构造的SQL EXCEPT。

**except\_all**(*other*, *\*\*kwargs*)

return a SQL EXCEPT ALL of this select() construct against the given selectable.

根据给定的可选择返回一个SQL EXCEPT ALL（）构造的全部。

**execute**(*\*multiparams*, *\*\*params*)

*inherited from the* [execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable.execute" \o "sqlalchemy.sql.expression.Executable.execute) *method of* [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable)

Compile and execute this [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable).

**execution\_options**(*\*\*kw*)

*inherited from the* [execution\_options()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable.execution_options" \o "sqlalchemy.sql.expression.Executable.execution_options) *method of* [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable)

Set non-SQL options for the statement which take effect during execution.

Execution options can be set on a per-statement or per [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) basis. Additionally, the [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) and ORM [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) objects provide access to execution options which they in turn configure upon connections.

The [execution\_options()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.execution_options" \o "sqlalchemy.sql.expression.Select.execution_options) method is generative. A new instance of this statement is returned that contains the options:

statement = select([table.c.x, table.c.y])statement = statement.execution\_options(autocommit=**True**)

Note that only a subset of possible execution options can be applied to a statement - these include "autocommit" and "stream\_results", but not "isolation\_level" or "compiled\_cache". See [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) for a full list of possible options.

**See also**

[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)

[Query.execution\_options()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.execution_options" \o "sqlalchemy.orm.query.Query.execution_options)

**for\_update**

*inherited from the* [for\_update](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect.for_update" \o "sqlalchemy.sql.expression.GenerativeSelect.for_update) *attribute of* [GenerativeSelect](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect" \o "sqlalchemy.sql.expression.GenerativeSelect)

Provide legacy dialect support for the for\_update attribute.

**foreign\_keys**

*inherited from the* [foreign\_keys](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.foreign_keys" \o "sqlalchemy.sql.expression.FromClause.foreign_keys) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return the collection of ForeignKey objects which this FromClause references.

**froms**

Return the displayed list of FromClause elements.

**get\_children**(*column\_collections=True*, *\*\*kwargs*)

return child elements as per the ClauseElement specification.

**group\_by**(*\*clauses*)

*inherited from the* [group\_by()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect.group_by" \o "sqlalchemy.sql.expression.GenerativeSelect.group_by) *method of* [GenerativeSelect](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect" \o "sqlalchemy.sql.expression.GenerativeSelect)

return a new selectable with the given list of GROUP BY criterion applied.

The criterion will be appended to any pre-existing GROUP BY criterion.

**having**(*having*)

return a new select() construct with the given expression added to its HAVING clause, joined to the existing clause via AND, if any.

**inner\_columns**

an iterator of all ColumnElement expressions which would be rendered into the columns clause of the resulting SELECT statement.

**intersect**(*other*, *\*\*kwargs*)

return a SQL INTERSECT of this select() construct against the given selectable.

**intersect\_all**(*other*, *\*\*kwargs*)

return a SQL INTERSECT ALL of this select() construct against the given selectable.

**join**(*right*, *onclause=None*, *isouter=False*, *full=False*)

*inherited from the* [join()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.join" \o "sqlalchemy.sql.expression.FromClause.join) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) from this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) to another [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

E.g.:

**from** **sqlalchemy** **import** join

j = user\_table.join(address\_table,

user\_table.c.id == address\_table.c.user\_id)stmt = select([user\_table]).select\_from(j)

would emit SQL along the lines of:

SELECT user.id, user.name FROM userJOIN address ON user.id = address.user\_id

|  |  |
| --- | --- |
| **Parameters:** | * ****right**** – the right side of the join; this is any [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) object such as a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object, and may also be a selectable-compatible object such as an ORM-mapped class. * ****onclause**** – a SQL expression representing the ON clause of the join. If left at None, [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) will attempt to join the two tables based on a foreign key relationship. * ****isouter**** – if True, render a LEFT OUTER JOIN, instead of JOIN. * ****full –****if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN. Implies [FromClause.join.isouter](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.join.params.isouter" \o "sqlalchemy.sql.expression.FromClause.join).   *New in version 1.1.* |

**See also**

[join()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.join" \o "sqlalchemy.sql.expression.join) - standalone function

[Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) - the type of object produced

**label**(*name*)

*inherited from the* [label()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.SelectBase.label" \o "sqlalchemy.sql.expression.SelectBase.label) *method of* [SelectBase](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.SelectBase" \o "sqlalchemy.sql.expression.SelectBase)

return a 'scalar' representation of this selectable, embedded as a subquery with a label.

**See also**

[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).

**lateral**(*name=None*)

*inherited from the* [lateral()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.lateral" \o "sqlalchemy.sql.expression.FromClause.lateral) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a LATERAL alias of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

The return value is the [Lateral](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Lateral" \o "sqlalchemy.sql.expression.Lateral) construct also provided by the top-level [lateral()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.lateral" \o "sqlalchemy.sql.expression.lateral) function.

*New in version 1.1.*

**See also**

[LATERAL correlation](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html" \l "lateral-selects) - overview of usage.

**limit**(*limit*)

*inherited from the* [limit()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect.limit" \o "sqlalchemy.sql.expression.GenerativeSelect.limit) *method of* [GenerativeSelect](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect" \o "sqlalchemy.sql.expression.GenerativeSelect)

return a new selectable with the given LIMIT criterion applied.

This is a numerical value which usually renders as a LIMIT expression in the resulting select. Backends that don't support LIMIT will attempt to provide similar functionality.

*Changed in version 1.0.0:*- [Select.limit()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.limit" \o "sqlalchemy.sql.expression.Select.limit) can now accept arbitrary SQL expressions as well as integer values.

|  |  |
| --- | --- |
| **Parameters:** | ****limit**** – an integer LIMIT parameter, or a SQL expression that provides an integer result. |

**locate\_all\_froms**(*\*args*, *\*\*kw*)

return a Set of all FromClause elements referenced by this Select.

This set is a superset of that returned by the froms property, which is specifically for those FromClause elements that would actually be rendered.

**offset**(*offset*)

*inherited from the* [offset()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect.offset" \o "sqlalchemy.sql.expression.GenerativeSelect.offset) *method of* [GenerativeSelect](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect" \o "sqlalchemy.sql.expression.GenerativeSelect)

return a new selectable with the given OFFSET criterion applied.

This is a numeric value which usually renders as an OFFSET expression in the resulting select. Backends that don't support OFFSET will attempt to provide similar functionality.

*Changed in version 1.0.0:*- [Select.offset()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.offset" \o "sqlalchemy.sql.expression.Select.offset) can now accept arbitrary SQL expressions as well as integer values.

|  |  |
| --- | --- |
| **Parameters:** | ****offset**** – an integer OFFSET parameter, or a SQL expression that provides an integer result. |

**order\_by**(*\*clauses*)

*inherited from the* [order\_by()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect.order_by" \o "sqlalchemy.sql.expression.GenerativeSelect.order_by) *method of* [GenerativeSelect](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect" \o "sqlalchemy.sql.expression.GenerativeSelect)

return a new selectable with the given list of ORDER BY criterion applied.

The criterion will be appended to any pre-existing ORDER BY criterion.

**outerjoin**(*right*, *onclause=None*, *full=False*)

*inherited from the* [outerjoin()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.outerjoin" \o "sqlalchemy.sql.expression.FromClause.outerjoin) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) from this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) to another [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause), with the "isouter" flag set to True.

E.g.:

**from** **sqlalchemy** **import** outerjoin

j = user\_table.outerjoin(address\_table,

user\_table.c.id == address\_table.c.user\_id)

The above is equivalent to:

j = user\_table.join(

address\_table,

user\_table.c.id == address\_table.c.user\_id,

isouter=**True**)

|  |  |
| --- | --- |
| **Parameters:** | * ****right**** – the right side of the join; this is any [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) object such as a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object, and may also be a selectable-compatible object such as an ORM-mapped class. * ****onclause**** – a SQL expression representing the ON clause of the join. If left at None, [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) will attempt to join the two tables based on a foreign key relationship. * ****full –****if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN.   *New in version 1.1.* |

**See also**

[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)

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

**params**(*\*optionaldict*, *\*\*kwargs*)

*inherited from the* [params()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.params" \o "sqlalchemy.sql.expression.ClauseElement.params) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Return a copy with [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) elements replaced.

Returns a copy of this ClauseElement with [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) elements replaced with values taken from the given dictionary:

**>>>** clause = column('x') + bindparam('foo')**>>>** print clause.compile().params{'foo':None}**>>>** print clause.params({'foo':7}).compile().params{'foo':7}

**prefix\_with**(*\*expr*, *\*\*kw*)

*inherited from the* [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) *method of* [HasPrefixes](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.HasPrefixes" \o "sqlalchemy.sql.expression.HasPrefixes)

Add one or more expressions following the statement keyword, i.e. SELECT, INSERT, UPDATE, or DELETE. Generative.

This is used to support backend-specific prefix keywords such as those provided by MySQL.

E.g.:

stmt = table.insert().prefix\_with("LOW\_PRIORITY", dialect="mysql")

Multiple prefixes can be specified by multiple calls to [prefix\_with()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.prefix_with" \o "sqlalchemy.sql.expression.Select.prefix_with).

|  |  |
| --- | --- |
| **Parameters:** | * ****\*expr**** – textual or [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement) construct which will be rendered following the INSERT, UPDATE, or DELETE keyword. * ****\*\*kw**** – A single keyword 'dialect' is accepted. This is an optional string dialect name which will limit rendering of this prefix to only that dialect. |

**primary\_key**

*inherited from the* [primary\_key](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.primary_key" \o "sqlalchemy.sql.expression.FromClause.primary_key) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return the collection of Column objects which comprise the primary key of this FromClause.

**reduce\_columns**(*only\_synonyms=True*)

Return a new :func`.select` construct with redundantly named, equivalently-valued columns removed from the columns clause.

"Redundant" here means two columns where one refers to the other either based on foreign key, or via a simple equality comparison in the WHERE clause of the statement. The primary purpose of this method is to automatically construct a select statement with all uniquely-named columns, without the need to use table-qualified labels as [apply\_labels()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.apply_labels" \o "sqlalchemy.sql.expression.Select.apply_labels) does.

When columns are omitted based on foreign key, the referred-to column is the one that's kept. When columns are omitted based on WHERE eqivalence, the first column in the columns clause is the one that's kept.

|  |  |
| --- | --- |
| **Parameters:** | ****only\_synonyms**** – when True, limit the removal of columns to those which have the same name as the equivalent. Otherwise, all columns that are equivalent to another are removed. |

*New in version 0.8.*

**replace\_selectable**(*old*, *alias*)

*inherited from the* [replace\_selectable()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.replace_selectable" \o "sqlalchemy.sql.expression.FromClause.replace_selectable) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

replace all occurrences of FromClause 'old' with the given Alias object, returning a copy of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

**scalar**(*\*multiparams*, *\*\*params*)

*inherited from the* [scalar()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable.scalar" \o "sqlalchemy.sql.expression.Executable.scalar) *method of* [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable)

Compile and execute this [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable), returning the result's scalar representation.

**select**(*whereclause=None*, *\*\*params*)

*inherited from the* [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.select" \o "sqlalchemy.sql.expression.FromClause.select) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

return a SELECT of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

**See also**

[select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) - general purpose method which allows for arbitrary column lists.

**select\_from**(*fromclause*)

return a new [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) construct with the given FROM expression merged into its list of FROM objects.

E.g.:

table1 = table('t1', column('a'))table2 = table('t2', column('b'))s = select([table1.c.a]).\

select\_from(

table1.join(table2, table1.c.a==table2.c.b)

)

The "from" list is a unique set on the identity of each element, so adding an already present [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) or other selectable will have no effect. Passing a [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join)that refers to an already present [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) or other selectable will have the effect of concealing the presence of that selectable as an individual element in the rendered FROM list, instead rendering it into a JOIN clause.

While the typical purpose of [Select.select\_from()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.select_from" \o "sqlalchemy.sql.expression.Select.select_from) is to replace the default, derived FROM clause with a join, it can also be called with individual table elements, multiple times if desired, in the case that the FROM clause cannot be fully derived from the columns clause:

select([func.count('\*')]).select\_from(table1)

**self\_group**(*against=None*)

return a 'grouping' construct as per the ClauseElement specification.

This produces an element that can be embedded in an expression. Note that this method is called automatically as needed when constructing expressions and should not require explicit use.

**suffix\_with**(*\*expr*, *\*\*kw*)

*inherited from the* [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) *method of* [HasSuffixes](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.HasSuffixes" \o "sqlalchemy.sql.expression.HasSuffixes)

Add one or more expressions following the statement as a whole.

This is used to support backend-specific suffix keywords on certain constructs.

E.g.:

stmt = select([col1, col2]).cte().suffix\_with(

"cycle empno set y\_cycle to 1 default 0", dialect="oracle")

Multiple suffixes can be specified by multiple calls to [suffix\_with()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.suffix_with" \o "sqlalchemy.sql.expression.Select.suffix_with).

|  |  |
| --- | --- |
| **Parameters:** | * ****\*expr**** – textual or [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement) construct which will be rendered following the target clause. * ****\*\*kw**** – A single keyword 'dialect' is accepted. This is an optional string dialect name which will limit rendering of this suffix to only that dialect. |

**tablesample**(*sampling*, *name=None*, *seed=None*)

*inherited from the* [tablesample()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.tablesample" \o "sqlalchemy.sql.expression.FromClause.tablesample) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a TABLESAMPLE alias of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

The return value is the [TableSample](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableSample" \o "sqlalchemy.sql.expression.TableSample) construct also provided by the top-level [tablesample()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.tablesample" \o "sqlalchemy.sql.expression.tablesample) function.

*New in version 1.1.*

**See also**

[tablesample()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.tablesample" \o "sqlalchemy.sql.expression.tablesample) - usage guidelines and parameters

**union**(*other*, *\*\*kwargs*)

return a SQL UNION of this select() construct against the given selectable.

**union\_all**(*other*, *\*\*kwargs*)

return a SQL UNION ALL of this select() construct against the given selectable.

**unique\_params**(*\*optionaldict*, *\*\*kwargs*)

*inherited from the* [unique\_params()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.unique_params" \o "sqlalchemy.sql.expression.ClauseElement.unique_params) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Return a copy with [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) elements replaced.

Same functionality as params(), except adds unique=True to affected bind parameters so that multiple statements can be used.

**where**(*whereclause*)

return a new select() construct with the given expression added to its WHERE clause, joined to the existing clause via AND, if any.

**with\_for\_update**(*nowait=False*, *read=False*, *of=None*, *skip\_locked=False*, *key\_share=False*)

*inherited from the* [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) *method of* [GenerativeSelect](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect" \o "sqlalchemy.sql.expression.GenerativeSelect)

Specify a FOR UPDATE clause for this [GenerativeSelect](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.GenerativeSelect" \o "sqlalchemy.sql.expression.GenerativeSelect).

E.g.:

stmt = select([table]).with\_for\_update(nowait=**True**)

On a database like PostgreSQL or Oracle, the above would render a statement like:

SELECT table.a, table.b FROM table FOR UPDATE NOWAIT

on other backends, the nowait option is ignored and instead would produce:

SELECT table.a, table.b FROM table FOR UPDATE

When called with no arguments, the statement will render with the suffix FOR UPDATE. Additional arguments can then be provided which allow for common database-specific variants.

|  |  |
| --- | --- |
| **Parameters:** | * ****nowait**** – boolean; will render FOR UPDATE NOWAIT on Oracle and PostgreSQL dialects. * ****read**** – boolean; will render LOCK IN SHARE MODE on MySQL, FOR SHARE on PostgreSQL. On PostgreSQL, when combined with nowait, will render FOR SHARE NOWAIT. * ****of**** – SQL expression or list of SQL expression elements (typically [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects or a compatible expression) which will render into a FOR UPDATEOF clause; supported by PostgreSQL and Oracle. May render as a table or as a column depending on backend. * ****skip\_locked**** –   boolean, will render FOR UPDATE SKIP LOCKED on Oracle and PostgreSQL dialects or FOR SHARE SKIP LOCKED if read=True is also specified.  *New in version 1.1.0.*   * ****key\_share**** –   boolean, will render FOR NO KEY UPDATE, or if combined with read=True will render FOR KEY SHARE, on the PostgreSQL dialect.  *New in version 1.1.0.* |

**with\_hint**(*selectable*, *text*, *dialect\_name='\*'*)

Add an indexing or other executional context hint for the given selectable to this [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select).

The text of the hint is rendered in the appropriate location for the database backend in use, relative to the given [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) passed as theselectable argument. The dialect implementation typically uses Python string substitution syntax with the token %(name)s to render the name of the table or alias. E.g. when using Oracle, the following:

select([mytable]).\

with\_hint(mytable, "index(*%(name)s* ix\_mytable)")

Would render SQL as:

select /\*+ index(mytable ix\_mytable) \*/ ... **from** **mytable**

The dialect\_name option will limit the rendering of a particular hint to a particular backend. Such as, to add hints for both Oracle and Sybase simultaneously:

select([mytable]).\

with\_hint(mytable, "index(*%(name)s* ix\_mytable)", 'oracle').\

with\_hint(mytable, "WITH INDEX ix\_mytable", 'sybase')

**See also**

[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)

**with\_only\_columns**(*columns*)

Return a new [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) construct with its columns clause replaced with the given columns.

This method is exactly equivalent to as if the original [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) had been called with the given columns clause. I.e. a statement:

s = select([table1.c.a, table1.c.b])s = s.with\_only\_columns([table1.c.b])

should be exactly equivalent to:

s = select([table1.c.b])

This means that FROM clauses which are only derived from the column list will be discarded if the new column list no longer contains that FROM:

**>>>** table1 = table('t1', column('a'), column('b'))**>>>** table2 = table('t2', column('a'), column('b'))**>>>** s1 = select([table1.c.a, table2.c.b])**>>>** print s1SELECT t1.a, t2.b FROM t1, t2**>>>** s2 = s1.with\_only\_columns([table2.c.b])**>>>** print s2SELECT t2.b FROM t1

The preferred way to maintain a specific FROM clause in the construct, assuming it won't be represented anywhere else (i.e. not in the WHERE clause, etc.) is to set it using [Select.select\_from()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.select_from" \o "sqlalchemy.sql.expression.Select.select_from):

**>>>** s1 = select([table1.c.a, table2.c.b]).\**...**  select\_from(table1.join(table2,**...**  table1.c.a==table2.c.a))**>>>** s2 = s1.with\_only\_columns([table2.c.b])**>>>** print s2SELECT t2.b FROM t1 JOIN t2 ON t1.a=t2.a

Care should also be taken to use the correct set of column objects passed to [Select.with\_only\_columns()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.with_only_columns" \o "sqlalchemy.sql.expression.Select.with_only_columns). Since the method is essentially equivalent to calling the [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) construct in the first place with the given columns, the columns passed to [Select.with\_only\_columns()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.with_only_columns" \o "sqlalchemy.sql.expression.Select.with_only_columns)should usually be a subset of those which were passed to the [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) construct, not those which are available from the .c collection of that [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select). That is:

s = select([table1.c.a, table1.c.b]).select\_from(table1)s = s.with\_only\_columns([table1.c.b])

and ****not****:

*# usually incorrect*s = s.with\_only\_columns([s.c.b])

The latter would produce the SQL:

SELECT bFROM (SELECT t1.a AS a, t1.b AS bFROM t1), t1

Since the [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) construct is essentially being asked to select both from table1 as well as itself.

**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.

Hints here are specific to the backend database and may include directives such as isolation levels, file directives, fetch directives, etc.

*New in version 1.0.0.*

**See also**

[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)

*class*sqlalchemy.sql.expression.**Selectable**

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

mark a class as being selectable

*class*sqlalchemy.sql.expression.**SelectBase**

Bases: [sqlalchemy.sql.expression.HasCTE](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.HasCTE" \o "sqlalchemy.sql.expression.HasCTE), [sqlalchemy.sql.expression.Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable), [sqlalchemy.sql.expression.FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Base class for SELECT statements.

This includes [Select](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select" \o "sqlalchemy.sql.expression.Select), [CompoundSelect](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.CompoundSelect" \o "sqlalchemy.sql.expression.CompoundSelect) and [TextAsFrom](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TextAsFrom" \o "sqlalchemy.sql.expression.TextAsFrom).

**\_\_init\_\_**

*inherited from the* \_\_init\_\_ *attribute of* object

x.\_\_init\_\_(…) initializes x; see help(type(x)) for signature

**alias**(*name=None*, *flat=False*)

*inherited from the* [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 of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

return an alias of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

This is shorthand for calling:

**from** **sqlalchemy** **import** aliasa = alias(self, name=name)

See [alias()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.alias" \o "sqlalchemy.sql.expression.alias) for details.

**as\_scalar**()

return a 'scalar' representation of this selectable, which can be used as a column expression.

Typically, a select statement which has only one column in its columns clause is eligible to be used as a scalar expression.

The returned object is an instance of [ScalarSelect](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.ScalarSelect" \o "sqlalchemy.sql.expression.ScalarSelect).

**autocommit**()

return a new selectable with the 'autocommit' flag set to True.

*Deprecated since version 0.6:*autocommit() is deprecated. Use [Executable.execution\_options()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable.execution_options" \o "sqlalchemy.sql.expression.Executable.execution_options) with the 'autocommit' flag.

**bind**

*inherited from the* [bind](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable.bind" \o "sqlalchemy.sql.expression.Executable.bind) *attribute of* [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable)

Returns 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) to which this [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable) is bound, or None if none found.

This is a traversal which checks locally, then checks among the "from" clauses of associated objects until a bound engine or connection is found.

**c**

*inherited from the* [c](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.c" \o "sqlalchemy.sql.expression.FromClause.c) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

An alias for the [columns](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.SelectBase.columns" \o "sqlalchemy.sql.expression.SelectBase.columns) attribute.

**columns**

*inherited from the* [columns](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.columns" \o "sqlalchemy.sql.expression.FromClause.columns) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

A named-based collection of [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) objects maintained by this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

The [columns](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.SelectBase.columns" \o "sqlalchemy.sql.expression.SelectBase.columns), or [c](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.SelectBase.c" \o "sqlalchemy.sql.expression.SelectBase.c) collection, is the gateway to the construction of SQL expressions using table-bound or other selectable-bound columns:

select([mytable]).where(mytable.c.somecolumn == 5)

**compare**(*other*, *\*\*kw*)

*inherited from the* [compare()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.compare" \o "sqlalchemy.sql.expression.ClauseElement.compare) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Compare this ClauseElement to the given ClauseElement.

Subclasses should override the default behavior, which is a straight identity comparison.

\*\*kw are arguments consumed by subclass compare() methods and may be used to modify the criteria for comparison. (see [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement))

**compile**(*bind=None*, *dialect=None*, *\*\*kw*)

*inherited from the* [compile()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.compile" \o "sqlalchemy.sql.expression.ClauseElement.compile) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Compile this SQL expression.

The return value is a [Compiled](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Compiled" \o "sqlalchemy.engine.interfaces.Compiled) object. Calling str() or unicode() on the returned value will yield a string representation of the result. The[Compiled](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Compiled" \o "sqlalchemy.engine.interfaces.Compiled) object also can return a dictionary of bind parameter names and values using the params accessor.

|  |  |
| --- | --- |
| **Parameters:** | * ****bind**** – An Engine or Connection from which a Compiled will be acquired. This argument takes precedence over this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)'s bound engine, if any. * ****column\_keys**** – Used for INSERT and UPDATE statements, a list of column names which should be present in the VALUES clause of the compiled statement. If None, all columns from the target table object are rendered. * ****dialect**** – A Dialect instance from which a Compiled will be acquired. This argument takes precedence over the bind argument as well as this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)'s bound engine, if any. * ****inline**** – Used for INSERT statements, for a dialect which does not support inline retrieval of newly generated primary key columns, will force the expression used to create the new primary key value to be rendered inline within the INSERT statement's VALUES clause. This typically refers to Sequence execution but may also refer to any server-side default generation function associated with a primary key Column. * ****compile\_kwargs**** –   optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the literal\_binds flag through:  **from** **sqlalchemy.sql** **import** table, column, select  t = table('t', column('x'))  s = select([t]).where(t.c.x == 5)  print s.compile(compile\_kwargs={"literal\_binds": **True**})  *New in version 0.9.0.* |

**See also**

[How do I render SQL expressions as strings, possibly with bound parameters inlined?](http://docs.sqlalchemy.org/en/rel_1_1/faq/sqlexpressions.html" \l "faq-sql-expression-string)

**correspond\_on\_equivalents**(*column*, *equivalents*)

*inherited from the* [correspond\_on\_equivalents()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.correspond_on_equivalents" \o "sqlalchemy.sql.expression.FromClause.correspond_on_equivalents) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return corresponding\_column for the given column, or if None search for a match in the given dictionary.

**corresponding\_column**(*column*, *require\_embedded=False*)

*inherited from the* [corresponding\_column()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.corresponding_column" \o "sqlalchemy.sql.expression.FromClause.corresponding_column) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Given a [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement), return the exported [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) object from this [Selectable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Selectable" \o "sqlalchemy.sql.expression.Selectable) which corresponds to that original [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) via a common ancestor column.

|  |  |
| --- | --- |
| **Parameters:** | * ****column**** – the target [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) to be matched * ****require\_embedded**** – only return corresponding columns for the given [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement), if the given [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) is actually present within a sub-element of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause). Normally the column will match if it merely shares a common ancestor with one of the exported columns of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause). |

**count**(*whereclause=None*, *\*\*params*)

*inherited from the* [count()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.count" \o "sqlalchemy.sql.expression.FromClause.count) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

return a SELECT COUNT generated against this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

*Deprecated since version 1.1:*FromClause.count() is deprecated. Counting rows requires that the correct column expression and accommodations for joins, DISTINCT, etc. must be made, otherwise results may not be what's expected. Please use an appropriate func.count() expression directly.

The function generates COUNT against the first column in the primary key of the table, or against the first column in the table overall. Explicit use offunc.count() should be preferred:

row\_count = conn.scalar(

select([func.count('\*')]).select\_from(table))

**See also**

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

**cte**(*name=None*, *recursive=False*)

*inherited from the* [cte()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.HasCTE.cte" \o "sqlalchemy.sql.expression.HasCTE.cte) *method of* [HasCTE](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.HasCTE" \o "sqlalchemy.sql.expression.HasCTE)

Return a new [CTE](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.CTE" \o "sqlalchemy.sql.expression.CTE), or Common Table Expression instance.

Common table expressions are a SQL standard whereby SELECT statements can draw upon secondary statements specified along with the primary statement, using a clause called "WITH". Special semantics regarding UNION can also be employed to allow "recursive" queries, where a SELECT statement can draw upon the set of rows that have previously been selected.

CTEs can also be applied to DML constructs UPDATE, INSERT and DELETE on some databases, both as a source of CTE rows when combined with RETURNING, as well as a consumer of CTE rows.

SQLAlchemy detects [CTE](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.CTE" \o "sqlalchemy.sql.expression.CTE) objects, which are treated similarly to [Alias](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Alias" \o "sqlalchemy.sql.expression.Alias) objects, as special elements to be delivered to the FROM clause of the statement as well as to a WITH clause at the top of the statement.

*Changed in version 1.1:*Added support for UPDATE/INSERT/DELETE as CTE, CTEs added to UPDATE/INSERT/DELETE.

|  |  |
| --- | --- |
| **Parameters:** | * ****name**** – name given to the common table expression. Like \_FromClause.alias(), the name can be left as None in which case an anonymous symbol will be used at query compile time. * ****recursive**** – if True, will render WITH RECURSIVE. A recursive common table expression is intended to be used in conjunction with UNION ALL in order to derive rows from those already selected. |

The following examples include two from PostgreSQL's documentation at <http://www.postgresql.org/docs/current/static/queries-with.html>, as well as additional examples.

Example 1, non recursive:

**from** **sqlalchemy** **import** (Table, Column, String, Integer,

MetaData, select, func)

metadata = MetaData()

orders = Table('orders', metadata,

Column('region', String),

Column('amount', Integer),

Column('product', String),

Column('quantity', Integer))

regional\_sales = select([

orders.c.region,

func.sum(orders.c.amount).label('total\_sales')

]).group\_by(orders.c.region).cte("regional\_sales")

top\_regions = select([regional\_sales.c.region]).\

where(

regional\_sales.c.total\_sales >

select([

func.sum(regional\_sales.c.total\_sales)/10

])

).cte("top\_regions")

statement = select([

orders.c.region,

orders.c.product,

func.sum(orders.c.quantity).label("product\_units"),

func.sum(orders.c.amount).label("product\_sales")

]).where(orders.c.region.in\_(

select([top\_regions.c.region])

)).group\_by(orders.c.region, orders.c.product)

result = conn.execute(statement).fetchall()

Example 2, WITH RECURSIVE:

**from** **sqlalchemy** **import** (Table, Column, String, Integer,

MetaData, select, func)

metadata = MetaData()

parts = Table('parts', metadata,

Column('part', String),

Column('sub\_part', String),

Column('quantity', Integer),)

included\_parts = select([

parts.c.sub\_part,

parts.c.part,

parts.c.quantity]).\

where(parts.c.part=='our part').\

cte(recursive=**True**)

incl\_alias = included\_parts.alias()parts\_alias = parts.alias()included\_parts = included\_parts.union\_all(

select([

parts\_alias.c.sub\_part,

parts\_alias.c.part,

parts\_alias.c.quantity

]).

where(parts\_alias.c.part==incl\_alias.c.sub\_part))

statement = select([

included\_parts.c.sub\_part,

func.sum(included\_parts.c.quantity).

label('total\_quantity')

]).\

group\_by(included\_parts.c.sub\_part)

result = conn.execute(statement).fetchall()

Example 3, an upsert using UPDATE and INSERT with CTEs:

**from** **datetime** **import** date

**from** **sqlalchemy** **import** (MetaData, Table, Column, Integer,

Date, select, literal, and\_, exists)

metadata = MetaData()

visitors = Table('visitors', metadata,

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

Column('date', Date, primary\_key=**True**),

Column('count', Integer),)

*# add 5 visitors for the product\_id == 1*product\_id = 1day = date.today()count = 5

update\_cte = (

visitors.update()

.where(and\_(visitors.c.product\_id == product\_id,

visitors.c.date == day))

.values(count=visitors.c.count + count)

.returning(literal(1))

.cte('update\_cte'))

upsert = visitors.insert().from\_select(

[visitors.c.product\_id, visitors.c.date, visitors.c.count],

select([literal(product\_id), literal(day), literal(count)])

.where(~exists(update\_cte.select())))

connection.execute(upsert)

**See also**

[orm.query.Query.cte()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.cte" \o "sqlalchemy.orm.query.Query.cte) - ORM version of [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).

**description**

*inherited from the* [description](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.description" \o "sqlalchemy.sql.expression.FromClause.description) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

a brief description of this FromClause.

Used primarily for error message formatting.

**execute**(*\*multiparams*, *\*\*params*)

*inherited from the* [execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable.execute" \o "sqlalchemy.sql.expression.Executable.execute) *method of* [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable)

Compile and execute this [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable).

**execution\_options**(*\*\*kw*)

*inherited from the* [execution\_options()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable.execution_options" \o "sqlalchemy.sql.expression.Executable.execution_options) *method of* [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable)

Set non-SQL options for the statement which take effect during execution.

Execution options can be set on a per-statement or per [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) basis. Additionally, the [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) and ORM [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) objects provide access to execution options which they in turn configure upon connections.

The [execution\_options()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.SelectBase.execution_options" \o "sqlalchemy.sql.expression.SelectBase.execution_options) method is generative. A new instance of this statement is returned that contains the options:

statement = select([table.c.x, table.c.y])statement = statement.execution\_options(autocommit=**True**)

Note that only a subset of possible execution options can be applied to a statement - these include "autocommit" and "stream\_results", but not "isolation\_level" or "compiled\_cache". See [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) for a full list of possible options.

**See also**

[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)

[Query.execution\_options()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.execution_options" \o "sqlalchemy.orm.query.Query.execution_options)

**foreign\_keys**

*inherited from the* [foreign\_keys](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.foreign_keys" \o "sqlalchemy.sql.expression.FromClause.foreign_keys) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return the collection of ForeignKey objects which this FromClause references.

**get\_children**(*\*\*kwargs*)

*inherited from the* [get\_children()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.get_children" \o "sqlalchemy.sql.expression.ClauseElement.get_children) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Return immediate child elements of this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement).

This is used for visit traversal.

\*\*kwargs may contain flags that change the collection that is returned, for example to return a subset of items in order to cut down on larger traversals, or to return child items from a different context (such as schema-level collections instead of clause-level).

**is\_derived\_from**(*fromclause*)

*inherited from the* [is\_derived\_from()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.is_derived_from" \o "sqlalchemy.sql.expression.FromClause.is_derived_from) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return True if this FromClause is 'derived' from the given FromClause.

An example would be an Alias of a Table is derived from that Table.

**join**(*right*, *onclause=None*, *isouter=False*, *full=False*)

*inherited from the* [join()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.join" \o "sqlalchemy.sql.expression.FromClause.join) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) from this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) to another [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

E.g.:

**from** **sqlalchemy** **import** join

j = user\_table.join(address\_table,

user\_table.c.id == address\_table.c.user\_id)stmt = select([user\_table]).select\_from(j)

would emit SQL along the lines of:

SELECT user.id, user.name FROM userJOIN address ON user.id = address.user\_id

|  |  |
| --- | --- |
| **Parameters:** | * ****right**** – the right side of the join; this is any [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) object such as a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object, and may also be a selectable-compatible object such as an ORM-mapped class. * ****onclause**** – a SQL expression representing the ON clause of the join. If left at None, [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) will attempt to join the two tables based on a foreign key relationship. * ****isouter**** – if True, render a LEFT OUTER JOIN, instead of JOIN. * ****full –****if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN. Implies [FromClause.join.isouter](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.join.params.isouter" \o "sqlalchemy.sql.expression.FromClause.join).   *New in version 1.1.* |

**See also**

[join()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.join" \o "sqlalchemy.sql.expression.join) - standalone function

[Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) - the type of object produced

**label**(*name*)

return a 'scalar' representation of this selectable, embedded as a subquery with a label.

**See also**

[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).

**lateral**(*name=None*)

*inherited from the* [lateral()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.lateral" \o "sqlalchemy.sql.expression.FromClause.lateral) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a LATERAL alias of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

The return value is the [Lateral](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Lateral" \o "sqlalchemy.sql.expression.Lateral) construct also provided by the top-level [lateral()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.lateral" \o "sqlalchemy.sql.expression.lateral) function.

*New in version 1.1.*

**See also**

[LATERAL correlation](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html" \l "lateral-selects) - overview of usage.

**outerjoin**(*right*, *onclause=None*, *full=False*)

*inherited from the* [outerjoin()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.outerjoin" \o "sqlalchemy.sql.expression.FromClause.outerjoin) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) from this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) to another [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause), with the "isouter" flag set to True.

E.g.:

**from** **sqlalchemy** **import** outerjoin

j = user\_table.outerjoin(address\_table,

user\_table.c.id == address\_table.c.user\_id)

The above is equivalent to:

j = user\_table.join(

address\_table,

user\_table.c.id == address\_table.c.user\_id,

isouter=**True**)

|  |  |
| --- | --- |
| **Parameters:** | * ****right**** – the right side of the join; this is any [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) object such as a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object, and may also be a selectable-compatible object such as an ORM-mapped class. * ****onclause**** – a SQL expression representing the ON clause of the join. If left at None, [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) will attempt to join the two tables based on a foreign key relationship. * ****full –****if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN.   *New in version 1.1.* |

**See also**

[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)

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

**params**(*\*optionaldict*, *\*\*kwargs*)

*inherited from the* [params()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.params" \o "sqlalchemy.sql.expression.ClauseElement.params) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Return a copy with [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) elements replaced.

Returns a copy of this ClauseElement with [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) elements replaced with values taken from the given dictionary:

**>>>** clause = column('x') + bindparam('foo')**>>>** print clause.compile().params{'foo':None}**>>>** print clause.params({'foo':7}).compile().params{'foo':7}

**primary\_key**

*inherited from the* [primary\_key](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.primary_key" \o "sqlalchemy.sql.expression.FromClause.primary_key) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return the collection of Column objects which comprise the primary key of this FromClause.

**replace\_selectable**(*old*, *alias*)

*inherited from the* [replace\_selectable()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.replace_selectable" \o "sqlalchemy.sql.expression.FromClause.replace_selectable) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

replace all occurrences of FromClause 'old' with the given Alias object, returning a copy of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

**scalar**(*\*multiparams*, *\*\*params*)

*inherited from the* [scalar()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable.scalar" \o "sqlalchemy.sql.expression.Executable.scalar) *method of* [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable)

Compile and execute this [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable), returning the result's scalar representation.

**select**(*whereclause=None*, *\*\*params*)

*inherited from the* [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.select" \o "sqlalchemy.sql.expression.FromClause.select) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

return a SELECT of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

**See also**

[select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) - general purpose method which allows for arbitrary column lists.

**self\_group**(*against=None*)

*inherited from the* [self\_group()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.self_group" \o "sqlalchemy.sql.expression.ClauseElement.self_group) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Apply a 'grouping' to this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement).

对此ClauseElement应用"分组"。

This method is overridden by subclasses to return a "grouping" construct, i.e. parenthesis. In particular it's used by "binary" expressions to provide a grouping around themselves when placed into a larger expression, as well as by [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) constructs when placed into the FROM clause of another [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select). (Note that subqueries should be normally created using the [Select.alias()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.alias" \o "sqlalchemy.sql.expression.Select.alias) method, as many platforms require nested SELECT statements to be named).

这个方法被子类覆盖，以返回一个"分组"结构，即括号。 特别地，它被"二进制"表达式用于在放入更大的表达式时以及通过将select() 结构放入另一个select() 的FROM子句时围绕它们进行分组。 （请注意，通常使用Select.alias() 方法创建子查询，因为许多平台需要嵌套的SELECT语句来命名）。

As expressions are composed together, the application of [self\_group()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.SelectBase.self_group" \o "sqlalchemy.sql.expression.SelectBase.self_group) is automatic - end-user code should never need to use this method directly. Note that SQLAlchemy's clause constructs take operator precedence into account - so parenthesis might not be needed, for example, in an expression like x OR (y AND z) - AND takes precedence over OR.

由于表达式组合在一起，self\_group() 的应用是自动的 - 最终用户代码不应该直接使用这种方法。 请注意，SQLAlchemy的子句构造考虑到运算符优先级，因此可能不需要括号，例如，在像x OR（y AND z）之类的表达式中，AND优先于OR。

The base [self\_group()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.SelectBase.self_group" \o "sqlalchemy.sql.expression.SelectBase.self_group) method of [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement) just returns self.

**tablesample**(*sampling*, *name=None*, *seed=None*)

*inherited from the* [tablesample()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.tablesample" \o "sqlalchemy.sql.expression.FromClause.tablesample) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a TABLESAMPLE alias of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

The return value is the [TableSample](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableSample" \o "sqlalchemy.sql.expression.TableSample) construct also provided by the top-level [tablesample()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.tablesample" \o "sqlalchemy.sql.expression.tablesample) function.

*New in version 1.1.*

**See also**

[tablesample()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.tablesample" \o "sqlalchemy.sql.expression.tablesample) - usage guidelines and parameters

**unique\_params**(*\*optionaldict*, *\*\*kwargs*)

*inherited from the* [unique\_params()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.unique_params" \o "sqlalchemy.sql.expression.ClauseElement.unique_params) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Return a copy with [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) elements replaced.

返回一个带有bindparam() 元素的副本。

Same functionality as params(), except adds unique=True to affected bind parameters so that multiple statements can be used.

与params() 具有相同的功能，除了对受影响的绑定参数添加unique = True，以便可以使用多个语句。

*class*sqlalchemy.sql.expression.**TableClause**(*name*, *\*columns*)

Bases: sqlalchemy.sql.expression.Immutable, [sqlalchemy.sql.expression.FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Represents a minimal "table" construct.

表示最小的“表”结构。

This is a lightweight table object that has only a name and a collection of columns, which are typically produced by the [expression.column()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.column" \o "sqlalchemy.sql.expression.column) function:

这是一个轻量级表对象，它只有一个名称和列的集合，这些列通常由expression.column() 函数生成：

**from** **sqlalchemy** **import** table, column

user = table("user",

column("id"),

column("name"),

column("description"),)

The [TableClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableClause" \o "sqlalchemy.sql.expression.TableClause) construct serves as the base for the more commonly used [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object, providing the usual set of [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) services including the .c. collection and statement generation methods.

TableClause结构作为更常用的Table对象的基础，提供了包括.c在内的通常的FromClause服务。 集合和语句生成方法。

It does ****not**** provide all the additional schema-level services of [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table), including constraints, references to other tables, or support for [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData)-level services. It's useful on its own as an ad-hoc construct used to generate quick SQL statements when a more fully fledged [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) is not on hand.

它不提供表的所有附加架构级服务，包括约束，对其他表的引用，或对MetaData级服务的支持。 当一个更完整的表不在手时，它本身就是一种用于生成快速SQL语句的ad-hoc结构。

**\_\_init\_\_**(*name*, *\*columns*)

Construct a new [TableClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableClause" \o "sqlalchemy.sql.expression.TableClause) object.

构造一个新的TableClause对象。

This constructor is mirrored as a public API function; see [table()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.table" \o "sqlalchemy.sql.expression.table) for a full usage and argument description.

该构造函数作为公共API函数进行镜像; 有关完整的用法和参数说明，请参阅table() 。

**alias**(*name=None*, *flat=False*)

*inherited from the* [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 of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

return an alias of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

This is shorthand for calling:

**from** **sqlalchemy** **import** aliasa = alias(self, name=name)

See [alias()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.alias" \o "sqlalchemy.sql.expression.alias) for details.

**c**

*inherited from the* [c](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.c" \o "sqlalchemy.sql.expression.FromClause.c) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

An alias for the [columns](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableClause.columns" \o "sqlalchemy.sql.expression.TableClause.columns) attribute.

**columns**

*inherited from the* [columns](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.columns" \o "sqlalchemy.sql.expression.FromClause.columns) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

A named-based collection of [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) objects maintained by this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

The [columns](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableClause.columns" \o "sqlalchemy.sql.expression.TableClause.columns), or [c](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableClause.c" \o "sqlalchemy.sql.expression.TableClause.c) collection, is the gateway to the construction of SQL expressions using table-bound or other selectable-bound columns:

select([mytable]).where(mytable.c.somecolumn == 5)

**compare**(*other*, *\*\*kw*)

*inherited from the* [compare()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.compare" \o "sqlalchemy.sql.expression.ClauseElement.compare) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Compare this ClauseElement to the given ClauseElement.

Subclasses should override the default behavior, which is a straight identity comparison.

\*\*kw are arguments consumed by subclass compare() methods and may be used to modify the criteria for comparison. (see [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement))

**compile**(*bind=None*, *dialect=None*, *\*\*kw*)

*inherited from the* [compile()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.compile" \o "sqlalchemy.sql.expression.ClauseElement.compile) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Compile this SQL expression.

The return value is a [Compiled](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Compiled" \o "sqlalchemy.engine.interfaces.Compiled) object. Calling str() or unicode() on the returned value will yield a string representation of the result. The[Compiled](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Compiled" \o "sqlalchemy.engine.interfaces.Compiled) object also can return a dictionary of bind parameter names and values using the params accessor.

|  |  |
| --- | --- |
| **Parameters:** | * ****bind**** – An Engine or Connection from which a Compiled will be acquired. This argument takes precedence over this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)'s bound engine, if any. * ****column\_keys**** – Used for INSERT and UPDATE statements, a list of column names which should be present in the VALUES clause of the compiled statement. If None, all columns from the target table object are rendered. * ****dialect**** – A Dialect instance from which a Compiled will be acquired. This argument takes precedence over the bind argument as well as this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)'s bound engine, if any. * ****inline**** – Used for INSERT statements, for a dialect which does not support inline retrieval of newly generated primary key columns, will force the expression used to create the new primary key value to be rendered inline within the INSERT statement's VALUES clause. This typically refers to Sequence execution but may also refer to any server-side default generation function associated with a primary key Column. * ****compile\_kwargs –****optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the literal\_binds flag through:在所有"访问"方法中将附加参数的可选字典传递给编译器。 例如，这允许将任何自定义标志传递到自定义编译构造。 它也用于通过literal\_binds标志的情况：   **from** **sqlalchemy.sql** **import** table, column, select  t = table('t', column('x'))  s = select([t]).where(t.c.x == 5)  print s.compile(compile\_kwargs={"literal\_binds": **True**})  *New in version 0.9.0.* |

**See also**

[How do I render SQL expressions as strings, possibly with bound parameters inlined?](http://docs.sqlalchemy.org/en/rel_1_1/faq/sqlexpressions.html" \l "faq-sql-expression-string)

**correspond\_on\_equivalents**(*column*, *equivalents*)

*inherited from the* [correspond\_on\_equivalents()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.correspond_on_equivalents" \o "sqlalchemy.sql.expression.FromClause.correspond_on_equivalents) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return corresponding\_column for the given column, or if None search for a match in the given dictionary.

**corresponding\_column**(*column*, *require\_embedded=False*)

*inherited from the* [corresponding\_column()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.corresponding_column" \o "sqlalchemy.sql.expression.FromClause.corresponding_column) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Given a [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement), return the exported [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) object from this [Selectable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Selectable" \o "sqlalchemy.sql.expression.Selectable) which corresponds to that original [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) via a common ancestor column.

|  |  |
| --- | --- |
| **Parameters:** | * ****column**** – the target [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) to be matched * ****require\_embedded**** – only return corresponding columns for the given [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement), if the given [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) is actually present within a sub-element of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause). Normally the column will match if it merely shares a common ancestor with one of the exported columns of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause). |

**count**(*whereclause=None*, *\*\*params*)

*inherited from the* [count()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.count" \o "sqlalchemy.sql.expression.FromClause.count) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

return a SELECT COUNT generated against this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

*Deprecated since version 1.1:*FromClause.count() is deprecated. Counting rows requires that the correct column expression and accommodations for joins, DISTINCT, etc. must be made, otherwise results may not be what's expected. Please use an appropriate func.count() expression directly.

The function generates COUNT against the first column in the primary key of the table, or against the first column in the table overall. Explicit use offunc.count() should be preferred:

row\_count = conn.scalar(

select([func.count('\*')]).select\_from(table))

**See also**

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

**delete**(*whereclause=None*, *\*\*kwargs*)

Generate a [delete()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.delete" \o "sqlalchemy.sql.expression.delete) construct against this [TableClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableClause" \o "sqlalchemy.sql.expression.TableClause).

E.g.:

table.delete().where(table.c.id==7)

See [delete()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.delete" \o "sqlalchemy.sql.expression.delete) for argument and usage information.

**foreign\_keys**

*inherited from the* [foreign\_keys](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.foreign_keys" \o "sqlalchemy.sql.expression.FromClause.foreign_keys) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return the collection of ForeignKey objects which this FromClause references.

**implicit\_returning***= False*

[TableClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableClause" \o "sqlalchemy.sql.expression.TableClause) doesn't support having a primary key or column -level defaults, so implicit returning doesn't apply.

**insert**(*values=None*, *inline=False*, *\*\*kwargs*)

Generate an [insert()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.insert" \o "sqlalchemy.sql.expression.insert) construct against this [TableClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableClause" \o "sqlalchemy.sql.expression.TableClause).

E.g.:

table.insert().values(name='foo')

See [insert()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.insert" \o "sqlalchemy.sql.expression.insert) for argument and usage information.

**is\_derived\_from**(*fromclause*)

*inherited from the* [is\_derived\_from()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.is_derived_from" \o "sqlalchemy.sql.expression.FromClause.is_derived_from) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return True if this FromClause is 'derived' from the given FromClause.

An example would be an Alias of a Table is derived from that Table.

**join**(*right*, *onclause=None*, *isouter=False*, *full=False*)

*inherited from the* [join()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.join" \o "sqlalchemy.sql.expression.FromClause.join) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) from this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) to another [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

E.g.:

**from** **sqlalchemy** **import** join

j = user\_table.join(address\_table,

user\_table.c.id == address\_table.c.user\_id)stmt = select([user\_table]).select\_from(j)

would emit SQL along the lines of:

SELECT user.id, user.name FROM userJOIN address ON user.id = address.user\_id

|  |  |
| --- | --- |
| **Parameters:** | * ****right**** – the right side of the join; this is any [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) object such as a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object, and may also be a selectable-compatible object such as an ORM-mapped class. * ****onclause**** – a SQL expression representing the ON clause of the join. If left at None, [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) will attempt to join the two tables based on a foreign key relationship. * ****isouter**** – if True, render a LEFT OUTER JOIN, instead of JOIN. * ****full**** –   if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN. Implies [FromClause.join.isouter](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.join.params.isouter" \o "sqlalchemy.sql.expression.FromClause.join).  *New in version 1.1.* |

**See also**

[join()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.join" \o "sqlalchemy.sql.expression.join) - standalone function

[Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) - the type of object produced

**lateral**(*name=None*)

*inherited from the* [lateral()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.lateral" \o "sqlalchemy.sql.expression.FromClause.lateral) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a LATERAL alias of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

The return value is the [Lateral](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Lateral" \o "sqlalchemy.sql.expression.Lateral) construct also provided by the top-level [lateral()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.lateral" \o "sqlalchemy.sql.expression.lateral) function.

*New in version 1.1.*

**See also**

[LATERAL correlation](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html" \l "lateral-selects) - overview of usage.

**outerjoin**(*right*, *onclause=None*, *full=False*)

*inherited from the* [outerjoin()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.outerjoin" \o "sqlalchemy.sql.expression.FromClause.outerjoin) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) from this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) to another [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause), with the "isouter" flag set to True.

E.g.:

**from** **sqlalchemy** **import** outerjoin

j = user\_table.outerjoin(address\_table,

user\_table.c.id == address\_table.c.user\_id)

The above is equivalent to:

j = user\_table.join(

address\_table,

user\_table.c.id == address\_table.c.user\_id,

isouter=**True**)

|  |  |
| --- | --- |
| **Parameters:** | * ****right**** – the right side of the join; this is any [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) object such as a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object, and may also be a selectable-compatible object such as an ORM-mapped class.联合的右边; 这是任何FromClause对象，如Table对象，也可以是可选择兼容的对象，如ORM映射类。 * ****onclause**** – a SQL expression representing the ON clause of the join. If left at None, [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) will attempt to join the two tables based on a foreign key relationship.表示连接的ON子句的SQL表达式。 如果保持为None，则FromClause.join() 将根据外键关系尝试加入这两个表。 * ****full –****if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN.如果为True，则呈现FULL OUTER JOIN，而不是LEFT OUTER JOIN。   *New in version 1.1.* |

**See also**

[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)

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

**primary\_key**

*inherited from the* [primary\_key](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.primary_key" \o "sqlalchemy.sql.expression.FromClause.primary_key) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return the collection of Column objects which comprise the primary key of this FromClause.

**replace\_selectable**(*old*, *alias*)

*inherited from the* [replace\_selectable()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.replace_selectable" \o "sqlalchemy.sql.expression.FromClause.replace_selectable) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

replace all occurrences of FromClause 'old' with the given Alias object, returning a copy of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

**select**(*whereclause=None*, *\*\*params*)

*inherited from the* [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.select" \o "sqlalchemy.sql.expression.FromClause.select) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

return a SELECT of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

**See also**

[select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) - general purpose method which allows for arbitrary column lists.

**self\_group**(*against=None*)

*inherited from the* [self\_group()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.self_group" \o "sqlalchemy.sql.expression.ClauseElement.self_group) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Apply a 'grouping' to this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement).

对此ClauseElement应用“分组”。

This method is overridden by subclasses to return a "grouping" construct, i.e. parenthesis. In particular it's used by "binary" expressions to provide a grouping around themselves when placed into a larger expression, as well as by [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) constructs when placed into the FROM clause of another [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select). (Note that subqueries should be normally created using the [Select.alias()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.alias" \o "sqlalchemy.sql.expression.Select.alias) method, as many platforms require nested SELECT statements to be named).

这个方法被子类覆盖，以返回一个“分组”结构，即括号。 特别地，它被“二进制”表达式用于在放入更大的表达式时以及通过将select() 结构放入另一个select() 的FROM子句时围绕它们进行分组。 （请注意，通常使用Select.alias() 方法创建子查询，因为许多平台需要嵌套的SELECT语句来命名）。

As expressions are composed together, the application of [self\_group()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableClause.self_group" \o "sqlalchemy.sql.expression.TableClause.self_group) is automatic - end-user code should never need to use this method directly. Note that SQLAlchemy's clause constructs take operator precedence into account - so parenthesis might not be needed, for example, in an expression like x OR (y AND z) - AND takes precedence over OR.

由于表达式组合在一起，self\_group() 的应用是自动的 - 最终用户代码不应该直接使用这种方法。 请注意，SQLAlchemy的子句构造考虑到运算符优先级，因此可能不需要括号，例如，在像x OR（y AND z）之类的表达式中，AND优先于OR。

The base [self\_group()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableClause.self_group" \o "sqlalchemy.sql.expression.TableClause.self_group) method of [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement) just returns self.

ClauseElement的基本self\_group() 方法只返回自身。

**tablesample**(*sampling*, *name=None*, *seed=None*)

*inherited from the* [tablesample()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.tablesample" \o "sqlalchemy.sql.expression.FromClause.tablesample) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a TABLESAMPLE alias of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

The return value is the [TableSample](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableSample" \o "sqlalchemy.sql.expression.TableSample) construct also provided by the top-level [tablesample()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.tablesample" \o "sqlalchemy.sql.expression.tablesample) function.

*New in version 1.1.*

**See also**

[tablesample()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.tablesample" \o "sqlalchemy.sql.expression.tablesample) - usage guidelines and parameters

**update**(*whereclause=None*, *values=None*, *inline=False*, *\*\*kwargs*)

Generate an [update()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.update" \o "sqlalchemy.sql.expression.update) construct against this [TableClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableClause" \o "sqlalchemy.sql.expression.TableClause).

E.g.:

table.update().where(table.c.id==7).values(name='foo')

See [update()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.update" \o "sqlalchemy.sql.expression.update) for argument and usage information.

*class*sqlalchemy.sql.expression.**TableSample**(*selectable*, *sampling*, *name=None*, *seed=None*)

Bases: [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)

Represent a TABLESAMPLE clause.

This object is constructed from the [tablesample()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.tablesample" \o "sqlalchemy.sql.expression.tablesample) module level function as well as the [FromClause.tablesample()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.tablesample" \o "sqlalchemy.sql.expression.FromClause.tablesample) method available on all [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) subclasses.

*New in version 1.1.*

**See also**

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

**alias**(*name=None*, *flat=False*)

*inherited from the* [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 of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

return an alias of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

This is shorthand for calling:

**from** **sqlalchemy** **import** aliasa = alias(self, name=name)

See [alias()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.alias" \o "sqlalchemy.sql.expression.alias) for details.

**c**

*inherited from the* [c](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.c" \o "sqlalchemy.sql.expression.FromClause.c) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

An alias for the [columns](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableSample.columns" \o "sqlalchemy.sql.expression.TableSample.columns) attribute.

**columns**

*inherited from the* [columns](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.columns" \o "sqlalchemy.sql.expression.FromClause.columns) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

A named-based collection of [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) objects maintained by this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

The [columns](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableSample.columns" \o "sqlalchemy.sql.expression.TableSample.columns), or [c](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableSample.c" \o "sqlalchemy.sql.expression.TableSample.c) collection, is the gateway to the construction of SQL expressions using table-bound or other selectable-bound columns:

select([mytable]).where(mytable.c.somecolumn == 5)

**compare**(*other*, *\*\*kw*)

*inherited from the* [compare()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.compare" \o "sqlalchemy.sql.expression.ClauseElement.compare) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Compare this ClauseElement to the given ClauseElement.

Subclasses should override the default behavior, which is a straight identity comparison.

\*\*kw are arguments consumed by subclass compare() methods and may be used to modify the criteria for comparison. (see [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement))

**compile**(*bind=None*, *dialect=None*, *\*\*kw*)

*inherited from the* [compile()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.compile" \o "sqlalchemy.sql.expression.ClauseElement.compile) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Compile this SQL expression.

The return value is a [Compiled](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Compiled" \o "sqlalchemy.engine.interfaces.Compiled) object. Calling str() or unicode() on the returned value will yield a string representation of the result. The[Compiled](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Compiled" \o "sqlalchemy.engine.interfaces.Compiled) object also can return a dictionary of bind parameter names and values using the params accessor.

|  |  |
| --- | --- |
| **Parameters:** | * ****bind**** – An Engine or Connection from which a Compiled will be acquired. This argument takes precedence over this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)'s bound engine, if any. * ****column\_keys**** – Used for INSERT and UPDATE statements, a list of column names which should be present in the VALUES clause of the compiled statement. If None, all columns from the target table object are rendered. * ****dialect**** – A Dialect instance from which a Compiled will be acquired. This argument takes precedence over the bind argument as well as this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)'s bound engine, if any. * ****inline**** – Used for INSERT statements, for a dialect which does not support inline retrieval of newly generated primary key columns, will force the expression used to create the new primary key value to be rendered inline within the INSERT statement's VALUES clause. This typically refers to Sequence execution but may also refer to any server-side default generation function associated with a primary key Column. * ****compile\_kwargs**** –   optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the literal\_binds flag through:  **from** **sqlalchemy.sql** **import** table, column, select  t = table('t', column('x'))  s = select([t]).where(t.c.x == 5)  print s.compile(compile\_kwargs={"literal\_binds": **True**})  *New in version 0.9.0.* |

**See also**

[How do I render SQL expressions as strings, possibly with bound parameters inlined?](http://docs.sqlalchemy.org/en/rel_1_1/faq/sqlexpressions.html" \l "faq-sql-expression-string)

**correspond\_on\_equivalents**(*column*, *equivalents*)

*inherited from the* [correspond\_on\_equivalents()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.correspond_on_equivalents" \o "sqlalchemy.sql.expression.FromClause.correspond_on_equivalents) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return corresponding\_column for the given column, or if None search for a match in the given dictionary.

**corresponding\_column**(*column*, *require\_embedded=False*)

*inherited from the* [corresponding\_column()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.corresponding_column" \o "sqlalchemy.sql.expression.FromClause.corresponding_column) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Given a [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement), return the exported [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) object from this [Selectable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Selectable" \o "sqlalchemy.sql.expression.Selectable) which corresponds to that original [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) via a common ancestor column.

|  |  |
| --- | --- |
| **Parameters:** | * ****column**** – the target [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) to be matched * ****require\_embedded**** – only return corresponding columns for the given [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement), if the given [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) is actually present within a sub-element of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause). Normally the column will match if it merely shares a common ancestor with one of the exported columns of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause). |

**~~count~~**~~(~~*~~whereclause=None~~*~~,~~*~~\*\*params~~*~~)~~

*~~inherited from the~~*~~[count()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.count" \o "sqlalchemy.sql.expression.FromClause.count)~~*~~method of~~*~~[FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)~~

~~return a SELECT COUNT generated against this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).~~

*~~Deprecated since version 1.1:~~*~~FromClause.count() is deprecated. Counting rows requires that the correct column expression and accommodations for joins, DISTINCT, etc. must be made, otherwise results may not be what's expected. Please use an appropriate func.count() expression directly.~~

~~The function generates COUNT against the first column in the primary key of the table, or against the first column in the table overall. Explicit use offunc.count() should be preferred:~~

~~row\_count = conn.scalar(~~

~~select([func.count('\*')]).select\_from(table))~~

**~~See also~~**

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

**foreign\_keys**

*inherited from the* [foreign\_keys](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.foreign_keys" \o "sqlalchemy.sql.expression.FromClause.foreign_keys) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return the collection of ForeignKey objects which this FromClause references.

**join**(*right*, *onclause=None*, *isouter=False*, *full=False*)

*inherited from the* [join()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.join" \o "sqlalchemy.sql.expression.FromClause.join) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) from this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) to another [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

E.g.:

**from** **sqlalchemy** **import** join

j = user\_table.join(address\_table,

user\_table.c.id == address\_table.c.user\_id)

stmt = select([user\_table]).select\_from(j)

would emit SQL along the lines of:

SELECT user.id, user.name FROM user

JOIN address ON user.id = address.user\_id

|  |  |
| --- | --- |
| **Parameters:** | * ****right**** – the right side of the join; this is any [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) object such as a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object, and may also be a selectable-compatible object such as an ORM-mapped class. * ****onclause**** – a SQL expression representing the ON clause of the join. If left at None, [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) will attempt to join the two tables based on a foreign key relationship. * ****isouter**** – if True, render a LEFT OUTER JOIN, instead of JOIN. * ****full –****if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN. Implies [FromClause.join.isouter](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.join.params.isouter" \o "sqlalchemy.sql.expression.FromClause.join).   *New in version 1.1.* |

**See also**

[join()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.join" \o "sqlalchemy.sql.expression.join) - standalone function

[Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) - the type of object produced

**lateral**(*name=None*)

*inherited from the* [lateral()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.lateral" \o "sqlalchemy.sql.expression.FromClause.lateral) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a LATERAL alias of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

The return value is the [Lateral](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Lateral" \o "sqlalchemy.sql.expression.Lateral) construct also provided by the top-level [lateral()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.lateral" \o "sqlalchemy.sql.expression.lateral) function.

*New in version 1.1.*

**See also**

[LATERAL correlation](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html" \l "lateral-selects) - overview of usage.

**outerjoin**(*right*, *onclause=None*, *full=False*)

*inherited from the* [outerjoin()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.outerjoin" \o "sqlalchemy.sql.expression.FromClause.outerjoin) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) from this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) to another [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause), with the "isouter" flag set to True.

E.g.:

**from** **sqlalchemy** **import** outerjoin

j = user\_table.outerjoin(address\_table,

user\_table.c.id == address\_table.c.user\_id)

The above is equivalent to:

j = user\_table.join(

address\_table,

user\_table.c.id == address\_table.c.user\_id,

isouter=**True**)

|  |  |
| --- | --- |
| **Parameters:** | * ****right**** – the right side of the join; this is any [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) object such as a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object, and may also be a selectable-compatible object such as an ORM-mapped class. * ****onclause**** – a SQL expression representing the ON clause of the join. If left at None, [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) will attempt to join the two tables based on a foreign key relationship. * ****full –****if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN.   *New in version 1.1.* |

**See also**

[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)

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

**params**(*\*optionaldict*, *\*\*kwargs*)

*inherited from the* [params()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.params" \o "sqlalchemy.sql.expression.ClauseElement.params) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Return a copy with [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) elements replaced.

Returns a copy of this ClauseElement with [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) elements replaced with values taken from the given dictionary:

**>>>** clause = column('x') + bindparam('foo')

**>>>** print clause.compile().params{'foo':None}

**>>>** print clause.params({'foo':7}).compile().params{'foo':7}

**primary\_key**

*inherited from the* [primary\_key](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.primary_key" \o "sqlalchemy.sql.expression.FromClause.primary_key) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return the collection of Column objects which comprise the primary key of this FromClause.

**replace\_selectable**(*old*, *alias*)

*inherited from the* [replace\_selectable()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.replace_selectable" \o "sqlalchemy.sql.expression.FromClause.replace_selectable) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

replace all occurrences of FromClause 'old' with the given Alias object, returning a copy of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

**select**(*whereclause=None*, *\*\*params*)

*inherited from the* [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.select" \o "sqlalchemy.sql.expression.FromClause.select) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

return a SELECT of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

**See also**

[select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) - general purpose method which allows for arbitrary column lists.

**tablesample**(*sampling*, *name=None*, *seed=None*)

*inherited from the* [tablesample()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.tablesample" \o "sqlalchemy.sql.expression.FromClause.tablesample) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a TABLESAMPLE alias of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

The return value is the [TableSample](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableSample" \o "sqlalchemy.sql.expression.TableSample) construct also provided by the top-level [tablesample()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.tablesample" \o "sqlalchemy.sql.expression.tablesample) function.

*New in version 1.1.*

**See also**

[tablesample()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.tablesample" \o "sqlalchemy.sql.expression.tablesample) - usage guidelines and parameters

**unique\_params**(*\*optionaldict*, *\*\*kwargs*)

*inherited from the* [unique\_params()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.unique_params" \o "sqlalchemy.sql.expression.ClauseElement.unique_params) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Return a copy with [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) elements replaced.

Same functionality as params(), except adds unique=True to affected bind parameters so that multiple statements can be used.

*class*sqlalchemy.sql.expression.**TextAsFrom**(*text*, *columns*, *positional=False*)

Bases: [sqlalchemy.sql.expression.SelectBase](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.SelectBase" \o "sqlalchemy.sql.expression.SelectBase)

Wrap a [TextClause](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.TextClause" \o "sqlalchemy.sql.expression.TextClause) construct within a [SelectBase](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.SelectBase" \o "sqlalchemy.sql.expression.SelectBase) interface.

This allows the [TextClause](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.TextClause" \o "sqlalchemy.sql.expression.TextClause) object to gain a .c collection and other FROM-like capabilities such as [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), [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), etc.

The [TextAsFrom](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TextAsFrom" \o "sqlalchemy.sql.expression.TextAsFrom) construct is produced via 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 - see that method for details.

*New in version 0.9.0.*

**See also**

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

[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)

**alias**(*name=None*, *flat=False*)

*inherited from the* [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 of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

return an alias of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

This is shorthand for calling:

**from** **sqlalchemy** **import** aliasa = alias(self, name=name)

See [alias()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.alias" \o "sqlalchemy.sql.expression.alias) for details.

**as\_scalar**()

*inherited from the* [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) *method of* [SelectBase](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.SelectBase" \o "sqlalchemy.sql.expression.SelectBase)

return a 'scalar' representation of this selectable, which can be used as a column expression.

返回此可选择的“标量”表示，可用作列表达式。

Typically, a select statement which has only one column in its columns clause is eligible to be used as a scalar expression.

通常，在其column子句中只有一列的select语句有资格用作标量表达式。

The returned object is an instance of [ScalarSelect](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.ScalarSelect" \o "sqlalchemy.sql.expression.ScalarSelect).

返回的对象是ScalarSelect的一个实例。

**~~autocommit~~**~~()~~

*~~inherited from the~~*~~[autocommit()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.SelectBase.autocommit" \o "sqlalchemy.sql.expression.SelectBase.autocommit)~~*~~method of~~*~~[SelectBase](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.SelectBase" \o "sqlalchemy.sql.expression.SelectBase)~~

~~return a new selectable with the 'autocommit' flag set to True.~~

*~~Deprecated since version 0.6:~~*~~autocommit() is deprecated. Use [Executable.execution\_options()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable.execution_options" \o "sqlalchemy.sql.expression.Executable.execution_options) with the 'autocommit' flag.~~

**bind**

*inherited from the* [bind](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable.bind" \o "sqlalchemy.sql.expression.Executable.bind) *attribute of* [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable)

Returns 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) to which this [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable) is bound, or None if none found.

This is a traversal which checks locally, then checks among the "from" clauses of associated objects until a bound engine or connection is found.

**c**

*inherited from the* [c](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.c" \o "sqlalchemy.sql.expression.FromClause.c) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

An alias for the [columns](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TextAsFrom.columns" \o "sqlalchemy.sql.expression.TextAsFrom.columns) attribute.

**columns**

*inherited from the* [columns](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.columns" \o "sqlalchemy.sql.expression.FromClause.columns) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

A named-based collection of [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) objects maintained by this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

The [columns](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TextAsFrom.columns" \o "sqlalchemy.sql.expression.TextAsFrom.columns), or [c](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TextAsFrom.c" \o "sqlalchemy.sql.expression.TextAsFrom.c) collection, is the gateway to the construction of SQL expressions using table-bound or other selectable-bound columns:

select([mytable]).where(mytable.c.somecolumn == 5)

**compare**(*other*, *\*\*kw*)

*inherited from the* [compare()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.compare" \o "sqlalchemy.sql.expression.ClauseElement.compare) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Compare this ClauseElement to the given ClauseElement.

Subclasses should override the default behavior, which is a straight identity comparison.

\*\*kw are arguments consumed by subclass compare() methods and may be used to modify the criteria for comparison. (see [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement))

**compile**(*bind=None*, *dialect=None*, *\*\*kw*)

*inherited from the* [compile()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.compile" \o "sqlalchemy.sql.expression.ClauseElement.compile) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Compile this SQL expression.

The return value is a [Compiled](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Compiled" \o "sqlalchemy.engine.interfaces.Compiled) object. Calling str() or unicode() on the returned value will yield a string representation of the result. The[Compiled](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Compiled" \o "sqlalchemy.engine.interfaces.Compiled) object also can return a dictionary of bind parameter names and values using the params accessor.

|  |  |
| --- | --- |
| **Parameters:** | * ****bind**** – An Engine or Connection from which a Compiled will be acquired. This argument takes precedence over this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)'s bound engine, if any. * ****column\_keys**** – Used for INSERT and UPDATE statements, a list of column names which should be present in the VALUES clause of the compiled statement. If None, all columns from the target table object are rendered. * ****dialect**** – A Dialect instance from which a Compiled will be acquired. This argument takes precedence over the bind argument as well as this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)'s bound engine, if any. * ****inline**** – Used for INSERT statements, for a dialect which does not support inline retrieval of newly generated primary key columns, will force the expression used to create the new primary key value to be rendered inline within the INSERT statement's VALUES clause. This typically refers to Sequence execution but may also refer to any server-side default generation function associated with a primary key Column. * ****compile\_kwargs –****optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the literal\_binds flag through:   **from** **sqlalchemy.sql** **import** table, column, select  t = table('t', column('x'))  s = select([t]).where(t.c.x == 5)  print s.compile(compile\_kwargs={"literal\_binds": **True**})  *New in version 0.9.0.* |

**See also**

[How do I render SQL expressions as strings, possibly with bound parameters inlined?](http://docs.sqlalchemy.org/en/rel_1_1/faq/sqlexpressions.html" \l "faq-sql-expression-string)

**correspond\_on\_equivalents**(*column*, *equivalents*)

*inherited from the* [correspond\_on\_equivalents()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.correspond_on_equivalents" \o "sqlalchemy.sql.expression.FromClause.correspond_on_equivalents) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return corresponding\_column for the given column, or if None search for a match in the given dictionary.

**corresponding\_column**(*column*, *require\_embedded=False*)

*inherited from the* [corresponding\_column()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.corresponding_column" \o "sqlalchemy.sql.expression.FromClause.corresponding_column) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Given a [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement), return the exported [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) object from this [Selectable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Selectable" \o "sqlalchemy.sql.expression.Selectable) which corresponds to that original [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) via a common ancestor column.

|  |  |
| --- | --- |
| **Parameters:** | * ****column**** – the target [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) to be matched * ****require\_embedded**** – only return corresponding columns for the given [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement), if the given [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) is actually present within a sub-element of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause). Normally the column will match if it merely shares a common ancestor with one of the exported columns of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause). |

**count**(*whereclause=None*, *\*\*params*)

*inherited from the* [count()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.count" \o "sqlalchemy.sql.expression.FromClause.count) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

return a SELECT COUNT generated against this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

*Deprecated since version 1.1:*FromClause.count() is deprecated. Counting rows requires that the correct column expression and accommodations for joins, DISTINCT, etc. must be made, otherwise results may not be what's expected. Please use an appropriate func.count() expression directly.

The function generates COUNT against the first column in the primary key of the table, or against the first column in the table overall. Explicit use offunc.count() should be preferred:

row\_count = conn.scalar(

select([func.count('\*')]).select\_from(table))

**See also**

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

**cte**(*name=None*, *recursive=False*)

*inherited from the* [cte()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.HasCTE.cte" \o "sqlalchemy.sql.expression.HasCTE.cte) *method of* [HasCTE](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.HasCTE" \o "sqlalchemy.sql.expression.HasCTE)

Return a new [CTE](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.CTE" \o "sqlalchemy.sql.expression.CTE), or Common Table Expression instance.

Common table expressions are a SQL standard whereby SELECT statements can draw upon secondary statements specified along with the primary statement, using a clause called "WITH". Special semantics regarding UNION can also be employed to allow "recursive" queries, where a SELECT statement can draw upon the set of rows that have previously been selected.

CTEs can also be applied to DML constructs UPDATE, INSERT and DELETE on some databases, both as a source of CTE rows when combined with RETURNING, as well as a consumer of CTE rows.

SQLAlchemy detects [CTE](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.CTE" \o "sqlalchemy.sql.expression.CTE) objects, which are treated similarly to [Alias](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Alias" \o "sqlalchemy.sql.expression.Alias) objects, as special elements to be delivered to the FROM clause of the statement as well as to a WITH clause at the top of the statement.

*Changed in version 1.1:*Added support for UPDATE/INSERT/DELETE as CTE, CTEs added to UPDATE/INSERT/DELETE.

|  |  |
| --- | --- |
| **Parameters:** | * ****name**** – name given to the common table expression. Like \_FromClause.alias(), the name can be left as None in which case an anonymous symbol will be used at query compile time. * ****recursive**** – if True, will render WITH RECURSIVE. A recursive common table expression is intended to be used in conjunction with UNION ALL in order to derive rows from those already selected. |

The following examples include two from PostgreSQL's documentation at <http://www.postgresql.org/docs/current/static/queries-with.html>, as well as additional examples.

Example 1, non recursive:

**from** **sqlalchemy** **import** (Table, Column, String, Integer,

MetaData, select, func)

metadata = MetaData()

orders = Table('orders', metadata,

Column('region', String),

Column('amount', Integer),

Column('product', String),

Column('quantity', Integer))

regional\_sales = select([

orders.c.region,

func.sum(orders.c.amount).label('total\_sales')

]).group\_by(orders.c.region).cte("regional\_sales")

top\_regions = select([regional\_sales.c.region]).\

where(

regional\_sales.c.total\_sales >

select([

func.sum(regional\_sales.c.total\_sales)/10

])

).cte("top\_regions")

statement = select([

orders.c.region,

orders.c.product,

func.sum(orders.c.quantity).label("product\_units"),

func.sum(orders.c.amount).label("product\_sales")

]).where(orders.c.region.in\_(

select([top\_regions.c.region])

)).group\_by(orders.c.region, orders.c.product)

result = conn.execute(statement).fetchall()

Example 2, WITH RECURSIVE:

**from** **sqlalchemy** **import** (Table, Column, String, Integer,

MetaData, select, func)

metadata = MetaData()

parts = Table('parts', metadata,

Column('part', String),

Column('sub\_part', String),

Column('quantity', Integer),)

included\_parts = select([

parts.c.sub\_part,

parts.c.part,

parts.c.quantity]).\

where(parts.c.part=='our part').\

cte(recursive=**True**)

incl\_alias = included\_parts.alias()parts\_alias = parts.alias()included\_parts = included\_parts.union\_all(

select([

parts\_alias.c.sub\_part,

parts\_alias.c.part,

parts\_alias.c.quantity

]).

where(parts\_alias.c.part==incl\_alias.c.sub\_part))

statement = select([

included\_parts.c.sub\_part,

func.sum(included\_parts.c.quantity).

label('total\_quantity')

]).\

group\_by(included\_parts.c.sub\_part)

result = conn.execute(statement).fetchall()

Example 3, an upsert using UPDATE and INSERT with CTEs:

**from** **datetime** **import** date**from** **sqlalchemy** **import** (MetaData, Table, Column, Integer,

Date, select, literal, and\_, exists)

metadata = MetaData()

visitors = Table('visitors', metadata,

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

Column('date', Date, primary\_key=**True**),

Column('count', Integer),)

*# add 5 visitors for the product\_id == 1*product\_id = 1day = date.today()count = 5

update\_cte = (

visitors.update()

.where(and\_(visitors.c.product\_id == product\_id,

visitors.c.date == day))

.values(count=visitors.c.count + count)

.returning(literal(1))

.cte('update\_cte'))

upsert = visitors.insert().from\_select(

[visitors.c.product\_id, visitors.c.date, visitors.c.count],

select([literal(product\_id), literal(day), literal(count)])

.where(~exists(update\_cte.select())))

connection.execute(upsert)

**See also**

[orm.query.Query.cte()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.cte" \o "sqlalchemy.orm.query.Query.cte) - ORM version of [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).

**description**

*inherited from the* [description](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.description" \o "sqlalchemy.sql.expression.FromClause.description) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

a brief description of this FromClause.

Used primarily for error message formatting.

**execute**(*\*multiparams*, *\*\*params*)

*inherited from the* [execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable.execute" \o "sqlalchemy.sql.expression.Executable.execute) *method of* [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable)

Compile and execute this [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable).

**execution\_options**(*\*\*kw*)

*inherited from the* [execution\_options()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable.execution_options" \o "sqlalchemy.sql.expression.Executable.execution_options) *method of* [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable)

Set non-SQL options for the statement which take effect during execution.

为执行期间生效的语句设置非SQL选项。

Execution options can be set on a per-statement or per [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) basis. Additionally, the [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) and ORM [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) objects provide access to execution options which they in turn configure upon connections.

可以在每个语句或每个[Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection)的基础上设置执行选项。 此外，[Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine)和ORM[Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query)对象提供对执行选项的访问，它们在连接时依次配置。

The [execution\_options()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TextAsFrom.execution_options" \o "sqlalchemy.sql.expression.TextAsFrom.execution_options) method is generative. A new instance of this statement is returned that contains the options:

[execution\_options()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TextAsFrom.execution_options" \o "sqlalchemy.sql.expression.TextAsFrom.execution_options)方法是生成的。 此语句的新实例将返回，其中包含以下选项：

statement = select([table.c.x, table.c.y])statement = statement.execution\_options(autocommit=**True**)

Note that only a subset of possible execution options can be applied to a statement - these include "autocommit" and "stream\_results", but not "isolation\_level" or "compiled\_cache". See [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) for a full list of possible options.

请注意，只有一个可能的执行选项的子集可以应用于一个语句 - 这些包括“autocommit”和“stream\_results”，而不是“isolation\_level”或“compiled\_cache”。 有关可能的选项的完整列表，请参阅[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)。

**See also**

[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)

[Query.execution\_options()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.execution_options" \o "sqlalchemy.orm.query.Query.execution_options)

**foreign\_keys**

*inherited from the* [foreign\_keys](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.foreign_keys" \o "sqlalchemy.sql.expression.FromClause.foreign_keys) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return the collection of ForeignKey objects which this FromClause references.

返回此FromClause引用的ForeignKey对象的集合。

**get\_children**(*\*\*kwargs*)

*inherited from the* [get\_children()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.get_children" \o "sqlalchemy.sql.expression.ClauseElement.get_children) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Return immediate child elements of this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement).

返回此[ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)的即时子元素。

This is used for visit traversal.

这用于访问遍历。

\*\*kwargs may contain flags that change the collection that is returned, for example to return a subset of items in order to cut down on larger traversals, or to return child items from a different context (such as schema-level collections instead of clause-level).

\*\* kwargs可能包含改变所返回的集合的标志，例如返回一个子集，以减少较大的遍历，或从不同的上下文返回子项（如模式级集合而不是语句级别）。

**is\_derived\_from**(*fromclause*)

*inherited from the* [is\_derived\_from()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.is_derived_from" \o "sqlalchemy.sql.expression.FromClause.is_derived_from) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return True if this FromClause is 'derived' from the given FromClause.

An example would be an Alias of a Table is derived from that Table.

如果FromClause是从给定的FromClause派生出来，返回True。

一个例子是从表中派生出一个表的别名。

**join**(*right*, *onclause=None*, *isouter=False*, *full=False*)

*inherited from the* [join()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.join" \o "sqlalchemy.sql.expression.FromClause.join) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) from this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) to another [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

E.g.:

**from** **sqlalchemy** **import** join

j = user\_table.join(address\_table,

user\_table.c.id == address\_table.c.user\_id)stmt = select([user\_table]).select\_from(j)

would emit SQL along the lines of:

SELECT user.id, user.name FROM userJOIN address ON user.id = address.user\_id

|  |  |
| --- | --- |
| **Parameters:** | * ****right**** – the right side of the join; this is any [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) object such as a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object, and may also be a selectable-compatible object such as an ORM-mapped class. * ****onclause**** – a SQL expression representing the ON clause of the join. If left at None, [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) will attempt to join the two tables based on a foreign key relationship. * ****isouter**** – if True, render a LEFT OUTER JOIN, instead of JOIN. * ****full –****if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN. Implies [FromClause.join.isouter](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.join.params.isouter" \o "sqlalchemy.sql.expression.FromClause.join).   *New in version 1.1.* |

**See also**

[join()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.join" \o "sqlalchemy.sql.expression.join) - standalone function

[Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) - the type of object produced

**label**(*name*)

*inherited from the* [label()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.SelectBase.label" \o "sqlalchemy.sql.expression.SelectBase.label) *method of* [SelectBase](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.SelectBase" \o "sqlalchemy.sql.expression.SelectBase)

return a 'scalar' representation of this selectable, embedded as a subquery with a label.

返回此可选择的“标量”表示，作为具有标签的子查询嵌入。

**See also**

[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).

**lateral**(*name=None*)

*inherited from the* [lateral()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.lateral" \o "sqlalchemy.sql.expression.FromClause.lateral) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a LATERAL alias of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

The return value is the [Lateral](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Lateral" \o "sqlalchemy.sql.expression.Lateral) construct also provided by the top-level [lateral()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.lateral" \o "sqlalchemy.sql.expression.lateral) function.

*New in version 1.1.*

**See also**

[LATERAL correlation](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html" \l "lateral-selects) - overview of usage.

**outerjoin**(*right*, *onclause=None*, *full=False*)

*inherited from the* [outerjoin()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.outerjoin" \o "sqlalchemy.sql.expression.FromClause.outerjoin) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) from this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) to another [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause), with the "isouter" flag set to True.

E.g.:

**from** **sqlalchemy** **import** outerjoin

j = user\_table.outerjoin(address\_table,

user\_table.c.id == address\_table.c.user\_id)

The above is equivalent to:

j = user\_table.join(

address\_table,

user\_table.c.id == address\_table.c.user\_id,

isouter=**True**)

|  |  |
| --- | --- |
| **Parameters:** | * ****right**** – the right side of the join; this is any [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) object such as a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object, and may also be a selectable-compatible object such as an ORM-mapped class. * ****onclause**** – a SQL expression representing the ON clause of the join. If left at None, [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) will attempt to join the two tables based on a foreign key relationship. * ****full**** –   if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN.  *New in version 1.1.* |

**See also**

[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)

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

**params**(*\*optionaldict*, *\*\*kwargs*)

*inherited from the* [params()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.params" \o "sqlalchemy.sql.expression.ClauseElement.params) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Return a copy with [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) elements replaced.

Returns a copy of this ClauseElement with [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) elements replaced with values taken from the given dictionary:

**>>>** clause = column('x') + bindparam('foo')**>>>** print clause.compile().params{'foo':None}**>>>** print clause.params({'foo':7}).compile().params{'foo':7}

**primary\_key**

*inherited from the* [primary\_key](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.primary_key" \o "sqlalchemy.sql.expression.FromClause.primary_key) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return the collection of Column objects which comprise the primary key of this FromClause.

**replace\_selectable**(*old*, *alias*)

*inherited from the* [replace\_selectable()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.replace_selectable" \o "sqlalchemy.sql.expression.FromClause.replace_selectable) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

replace all occurrences of FromClause 'old' with the given Alias object, returning a copy of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

**scalar**(*\*multiparams*, *\*\*params*)

*inherited from the* [scalar()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable.scalar" \o "sqlalchemy.sql.expression.Executable.scalar) *method of* [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable)

Compile and execute this [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable), returning the result's scalar representation.

**select**(*whereclause=None*, *\*\*params*)

*inherited from the* [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.select" \o "sqlalchemy.sql.expression.FromClause.select) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

return a SELECT of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

**See also**

[select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) - general purpose method which allows for arbitrary column lists.

**self\_group**(*against=None*)

*inherited from the* [self\_group()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.self_group" \o "sqlalchemy.sql.expression.ClauseElement.self_group) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Apply a 'grouping' to this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement).

对此ClauseElement应用"分组"。

This method is overridden by subclasses to return a "grouping" construct, i.e. parenthesis. In particular it's used by "binary" expressions to provide a grouping around themselves when placed into a larger expression, as well as by [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) constructs when placed into the FROM clause of another [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select). (Note that subqueries should be normally created using the [Select.alias()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.alias" \o "sqlalchemy.sql.expression.Select.alias) method, as many platforms require nested SELECT statements to be named).

这个方法被子类覆盖，以返回一个"分组"结构，即括号。 特别地，它被"二进制"表达式用于在放入更大的表达式时以及通过将select() 结构放入另一个select() 的FROM子句时围绕它们进行分组。 （请注意，通常使用Select.alias() 方法创建子查询，因为许多平台需要嵌套的SELECT语句来命名）。

As expressions are composed together, the application of [self\_group()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TextAsFrom.self_group" \o "sqlalchemy.sql.expression.TextAsFrom.self_group) is automatic - end-user code should never need to use this method directly. Note that SQLAlchemy's clause constructs take operator precedence into account - so parenthesis might not be needed, for example, in an expression like x OR (y AND z) - AND takes precedence over OR.

由于表达式组合在一起，self\_group() 的应用是自动的 - 最终用户代码不应该直接使用这种方法。 请注意，SQLAlchemy的子句构造考虑到运算符优先级，因此可能不需要括号，例如，在像x OR（y AND z）之类的表达式中，AND优先于OR。

The base [self\_group()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TextAsFrom.self_group" \o "sqlalchemy.sql.expression.TextAsFrom.self_group) method of [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement) just returns self.

**tablesample**(*sampling*, *name=None*, *seed=None*)

*inherited from the* [tablesample()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.tablesample" \o "sqlalchemy.sql.expression.FromClause.tablesample) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a TABLESAMPLE alias of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

The return value is the [TableSample](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableSample" \o "sqlalchemy.sql.expression.TableSample) construct also provided by the top-level [tablesample()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.tablesample" \o "sqlalchemy.sql.expression.tablesample) function.

*New in version 1.1.*

**See also**

[tablesample()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.tablesample" \o "sqlalchemy.sql.expression.tablesample) - usage guidelines and parameters

**unique\_params**(*\*optionaldict*, *\*\*kwargs*)

*inherited from the* [unique\_params()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.unique_params" \o "sqlalchemy.sql.expression.ClauseElement.unique_params) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Return a copy with [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) elements replaced.

Same functionality as params(), except adds unique=True to affected bind parameters so that multiple statements can be used.

## 2.3 Insert, Updates, Deletes

INSERT, UPDATE and DELETE statements build on a hierarchy starting with [UpdateBase](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase" \o "sqlalchemy.sql.expression.UpdateBase). The [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 build on the intermediary [ValuesBase](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase" \o "sqlalchemy.sql.expression.ValuesBase).

INSERT，UPDATE和DELETE语句构建在以UpdateBase开头的层次结构上。 插入和更新构造基于中间的ValueBase。

sqlalchemy.sql.expression.**delete**(*table*, *whereclause=None*, *bind=None*, *returning=None*, *prefixes=None*, *\*\*dialect\_kw*)

Construct [Delete](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Delete" \o "sqlalchemy.sql.expression.Delete) object.

Similar functionality is available via the [delete()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableClause.delete" \o "sqlalchemy.sql.expression.TableClause.delete) method on [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table).

|  |  |
| --- | --- |
| **Parameters:** | * ****table**** – The table to delete rows from. * ****whereclause**** – A [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement) describing the WHERE condition of the DELETE statement. Note that the [where()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Delete.where" \o "sqlalchemy.sql.expression.Delete.where) generative method may be used instead. |

**See also**

[Deletes](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html" \l "deletes) - SQL Expression Tutorial

sqlalchemy.sql.expression.**insert**(*table*, *values=None*, *inline=False*, *bind=None*, *prefixes=None*, *returning=None*, *return\_defaults=False*, *\*\*dialect\_kw*)

Construct an [Insert](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert" \o "sqlalchemy.sql.expression.Insert) object.

Similar functionality is available via the [insert()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableClause.insert" \o "sqlalchemy.sql.expression.TableClause.insert) method on [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table).

|  |  |
| --- | --- |
| **Parameters:** | * ****table**** – [TableClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableClause" \o "sqlalchemy.sql.expression.TableClause) which is the subject of the insert. * ****values**** – collection of values to be inserted; see [Insert.values()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert.values" \o "sqlalchemy.sql.expression.Insert.values) for a description of allowed formats here. Can be omitted entirely; a [Insert](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert" \o "sqlalchemy.sql.expression.Insert) construct will also dynamically render the VALUES clause at execution time based on the parameters passed to [Connection.execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.execute" \o "sqlalchemy.engine.Connection.execute). * ****inline**** – if True, no attempt will be made to retrieve the SQL-generated default values to be provided within the statement; in particular, this allows SQL expressions to be rendered 'inline' within the statement without the need to pre-execute them beforehand; for backends that support "returning", this turns off the "implicit returning" feature for the statement. |

If both values and compile-time bind parameters are present, the compile-time bind parameters override the information specified within values on a per-key basis.

如果存在值和编译时绑定参数，则编译时绑定参数将以每个键为基础覆盖值内指定的信息。

The keys within values can be either [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects or their string identifiers. Each key may reference one of:

值内的键可以是列对象或其字符串标识符。 每个键可以参考以下之一：

* a literal data value (i.e. string, number, etc.);
* a Column object;
* a SELECT statement.

If a SELECT statement is specified which references this INSERT statement's table, the statement will be correlated against the INSERT statement.

**See also**

[Insert Expressions](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html" \l "coretutorial-insert-expressions) - SQL Expression Tutorial

[Inserts, Updates and Deletes](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html" \l "inserts-and-updates) - SQL Expression Tutorial

sqlalchemy.sql.expression.**update**(*table*, *whereclause=None*, *values=None*, *inline=False*, *bind=None*, *prefixes=None*, *returning=None*, *return\_defaults=False*, *preserve\_parameter\_order=False*, *\*\*dialect\_kw*)

Construct an [Update](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Update" \o "sqlalchemy.sql.expression.Update) object.

E.g.:

**from** **sqlalchemy** **import** update

stmt = update(users).where(users.c.id==5).\

values(name='user #5')

Similar functionality is available via the [update()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableClause.update" \o "sqlalchemy.sql.expression.TableClause.update) method on [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table):

stmt = users.update().\

where(users.c.id==5).\

values(name='user #5')

|  |  |
| --- | --- |
| **Parameters:** | * ****table**** – A [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object representing the database table to be updated. * ****whereclause –****Optional SQL expression describing the WHERE condition of the UPDATE statement. Modern applications may prefer to use the generative [where()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Update.where" \o "sqlalchemy.sql.expression.Update.where) method to specify the WHERE clause.   The WHERE clause can refer to multiple tables. For databases which support this, an UPDATE FROM clause will be generated, or on MySQL, a multi-table update. The statement will fail on databases that don't have support for multi-table update statements. A SQL-standard method of referring to additional tables in the WHERE clause is to use a correlated subquery:  users.update().values(name='ed').where(  users.c.name==select([addresses.c.email\_address]).\  where(addresses.c.user\_id==users.c.id).\  as\_scalar()  )  *Changed in version 0.7.4:*The WHERE clause can refer to multiple tables.   * ****values –****Optional dictionary which specifies the SET conditions of the UPDATE. If left as None, the SET conditions are determined from those parameters passed to the statement during the execution and/or compilation of the statement. When compiled standalone without any parameters, the SET clause generates for all columns.   Modern applications may prefer to use the generative [Update.values()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Update.values" \o "sqlalchemy.sql.expression.Update.values) method to set the values of the UPDATE statement.   * ****inline**** – if True, SQL defaults present on [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects via the default keyword will be compiled 'inline' into the statement and not pre-executed. This means that their values will not be available in the dictionary returned from [ResultProxy.last\_updated\_params()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.last_updated_params" \o "sqlalchemy.engine.ResultProxy.last_updated_params). * ****preserve\_parameter\_order –****if True, the update statement is expected to receive parameters ****only**** via the [Update.values()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Update.values" \o "sqlalchemy.sql.expression.Update.values) method, and they must be passed as a Python list of 2-tuples. The rendered UPDATE statement will emit the SET clause for each referenced column maintaining this order.   *New in version 1.0.10.*  **See also**  [Parameter-Ordered Updates](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html" \l "updates-order-parameters) - full example of 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 |

If both values and compile-time bind parameters are present, the compile-time bind parameters override the information specified within values on a per-key basis.

如果存在值和编译时绑定参数，则编译时绑定参数将以每个键为基础覆盖值内指定的信息。

The keys within values can be either [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects or their string identifiers (specifically the "key" of the [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column), normally but not necessarily equivalent to its "name"). Normally, the [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects used here are expected to be part of the target [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) that is the table to be updated. However when using MySQL, a multiple-table UPDATE statement can refer to columns from any of the tables referred to in the WHERE clause.

值内的键可以是列对象或其字符串标识符（特别是列的"键"，通常但不一定等同于其"名称"）。 通常，此处使用的列对象预计将作为要更新的表的目标表的一部分。 但是，当使用MySQL时，多表UPDATE语句可以引用WHERE子句中引用的任何表中的列。

The values referred to in values are typically:

值中引用的值通常为：

* a literal data value (i.e. string, number, etc.)文字数据值（即字符串，数字等）
* a SQL expression, such as a related [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column), a scalar-returning [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) construct, etc.一个SQL表达式，例如相关的列，标量返回的select（）结构等。

When combining [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) constructs within the values clause of an [update()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.update" \o "sqlalchemy.sql.expression.update) construct, the subquery represented by the [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) should be *correlated*to the parent table, that is, providing criterion which links the table inside the subquery to the outer table being updated:

当在update() 结构的values子句中组合select() 结构时，由select() 表示的子查询应与父表相关联，即提供将子查询中的表链接到正在更新的外表的标准：

users.update().values(

name=select([addresses.c.email\_address]).\

where(addresses.c.user\_id==users.c.id).\

as\_scalar()

)

**See also**

[Inserts, Updates and Deletes](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html" \l "inserts-and-updates) - SQL Expression Language Tutorial

*class*sqlalchemy.sql.expression.**Delete**(*table*, *whereclause=None*, *bind=None*, *returning=None*, *prefixes=None*, *\*\*dialect\_kw*)

Bases: [sqlalchemy.sql.expression.UpdateBase](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase" \o "sqlalchemy.sql.expression.UpdateBase)

Represent a DELETE construct.

表示DELETE构造。

The [Delete](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Delete" \o "sqlalchemy.sql.expression.Delete) object is created using the [delete()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.delete" \o "sqlalchemy.sql.expression.delete) function.

Delete对象使用delete() 函数创建。

**\_\_init\_\_**(*table*, *whereclause=None*, *bind=None*, *returning=None*, *prefixes=None*, *\*\*dialect\_kw*)

Construct a new [Delete](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Delete" \o "sqlalchemy.sql.expression.Delete) object.

构造一个新的Delete对象。

This constructor is mirrored as a public API function; see [delete()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.delete" \o "sqlalchemy.sql.expression.delete) for a full usage and argument description.

该构造函数作为公共API函数进行镜像; 有关完整的用法和参数说明，请参阅delete() 。

**argument\_for**(*dialect\_name*, *argument\_name*, *default*)

*inherited from the* [argument\_for()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.argument_for" \o "sqlalchemy.sql.base.DialectKWArgs.argument_for) *method of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

Add a new kind of dialect-specific keyword argument for this class.

为此类添加一种新的方言特定的关键字参数。

E.g.:

例如：

Index.argument\_for("mydialect", "length", **None**)

some\_index = Index('a', 'b', mydialect\_length=5)

The [DialectKWArgs.argument\_for()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.argument_for" \o "sqlalchemy.sql.base.DialectKWArgs.argument_for) method is a per-argument way adding extra arguments to the[DefaultDialect.construct\_arguments](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.default.DefaultDialect.construct_arguments" \o "sqlalchemy.engine.default.DefaultDialect.construct_arguments) dictionary. This dictionary provides a list of argument names accepted by various schema-level constructs on behalf of a dialect.

DialectKWArgs.argument\_for() 方法是向DefaultDialect.construct\_arguments字典添加额外参数的每个参数的方式。 该字典提供了代表方言由各种模式级结构接受的参数名称列表。

New dialects should typically specify this dictionary all at once as a data member of the dialect class. The use case for ad-hoc addition of argument names is typically for end-user code that is also using a custom compilation scheme which consumes the additional arguments.

新的方言通常应该一律将该字典全部指定为方言类的数据成员。 用于临时添加参数名称的用例通常用于最终用户代码，该代码也使用消耗额外参数的自定义编译方案。

|  |  |
| --- | --- |
| **Parameters:** | * ****dialect\_name**** – name of a dialect. The dialect must be locatable, else a [NoSuchModuleError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.NoSuchModuleError" \o "sqlalchemy.exc.NoSuchModuleError) is raised. The dialect must also include an existing[DefaultDialect.construct\_arguments](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.default.DefaultDialect.construct_arguments" \o "sqlalchemy.engine.default.DefaultDialect.construct_arguments) collection, indicating that it participates in the keyword-argument validation and default system, else [ArgumentError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.ArgumentError" \o "sqlalchemy.exc.ArgumentError) is raised. If the dialect does not include this collection, then any keyword argument can be specified on behalf of this dialect already. All dialects packaged within SQLAlchemy include this collection, however for third party dialects, support may vary.一个方言的名字 方言必须是可定位的，否则会引发NoSuchModuleError。 方言还必须包含一个existingDefaultDialect.construct\_arguments集合，表示它参与了关键字参数验证和默认系统，否则引发了ArgumentError。 如果方言不包括此集合，则可以代表此方言指定任何关键字参数。 SQLAlchemy中包装的所有方言都包含此集合，但是对于第三方方言，支持可能会有所不同。 * ****argument\_name**** – name of the parameter.参数名称。 * ****default**** – default value of the parameter.参数的默认值。 |

*New in version 0.9.4.*

**bind**

*inherited from the* [bind](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase.bind" \o "sqlalchemy.sql.expression.UpdateBase.bind) *attribute of* [UpdateBase](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase" \o "sqlalchemy.sql.expression.UpdateBase)

Return a 'bind' linked to this [UpdateBase](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase" \o "sqlalchemy.sql.expression.UpdateBase) or a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) associated with it.

返回一个链接到此UpdateBase或与之关联的表的“绑定”。

**compare**(*other*, *\*\*kw*)

*inherited from the* [compare()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.compare" \o "sqlalchemy.sql.expression.ClauseElement.compare) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Compare this ClauseElement to the given ClauseElement.

将此条款元素与给定的条款元素进行比较。

Subclasses should override the default behavior, which is a straight identity comparison.

子类应该覆盖默认行为，这是一个直接的身份比较。

\*\*kw are arguments consumed by subclass compare() methods and may be used to modify the criteria for comparison. (see [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement))

\*\* kw是由subclass compare() 方法消耗的参数，可用于修改比较条件。（见ColumnElement）

**compile**(*bind=None*, *dialect=None*, *\*\*kw*)

*inherited from the* [compile()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.compile" \o "sqlalchemy.sql.expression.ClauseElement.compile) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Compile this SQL expression.

编译此SQL表达式。

The return value is a [Compiled](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Compiled" \o "sqlalchemy.engine.interfaces.Compiled) object. Calling str() or unicode() on the returned value will yield a string representation of the result. The[Compiled](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Compiled" \o "sqlalchemy.engine.interfaces.Compiled) object also can return a dictionary of bind parameter names and values using the params accessor.

返回值是一个编译对象。 对返回的值调用str() 或unicode() 将产生结果的字符串表示形式。 Compiled对象还可以使用params访问器返回绑定参数名称和值的字典。

|  |  |
| --- | --- |
| **Parameters:** | * ****bind**** – An Engine or Connection from which a Compiled will be acquired. This argument takes precedence over this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)'s bound engine, if any.将从其获取编译的引擎或连接。 这个参数优先于这个ClauseElement的绑定引擎（如果有的话）。 * ****column\_keys**** – Used for INSERT and UPDATE statements, a list of column names which should be present in the VALUES clause of the compiled statement. If None, all columns from the target table object are rendered.用于INSERT和UPDATE语句，应在编译的VALUES子句中存在列列表 * ****dialect**** – A Dialect instance from which a Compiled will be acquired. This argument takes precedence over the bind argument as well as this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)'s bound engine, if any.将从其中获取编译的方言实例。 此参数优先于bind参数以及此ClauseElement的绑定引擎（如果有）。 * ****inline**** – Used for INSERT statements, for a dialect which does not support inline retrieval of newly generated primary key columns, will force the expression used to create the new primary key value to be rendered inline within the INSERT statement's VALUES clause. This typically refers to Sequence execution but may also refer to any server-side default generation function associated with a primary key Column.用于INSERT语句，对于不支持新生成的主键列的内联检索的方言，将强制用于创建新主键值的表达式在INSERT语句的VALUES子句中内联呈现。 这通常是指Sequence执行，但也可以指与主键Column相关联的任何服务器端默认生成函数。 * ****compile\_kwargs –****optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the literal\_binds flag through:在所有“访问”方法中将附加参数的可选字典传递给编译器。 例如，这允许将任何自定义标志传递到自定义编译构造。 它也用于通过literal\_binds标志的情况：   **from** **sqlalchemy.sql** **import** table, column, select  t = table('t', column('x'))  s = select([t]).where(t.c.x == 5)  print s.compile(compile\_kwargs={"literal\_binds": **True**})  *New in version 0.9.0.* |

**See also**

[How do I render SQL expressions as strings, possibly with bound parameters inlined?](http://docs.sqlalchemy.org/en/rel_1_1/faq/sqlexpressions.html" \l "faq-sql-expression-string)

**cte**(*name=None*, *recursive=False*)

*inherited from the* [cte()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.HasCTE.cte" \o "sqlalchemy.sql.expression.HasCTE.cte) *method of* [HasCTE](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.HasCTE" \o "sqlalchemy.sql.expression.HasCTE)

Return a new [CTE](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.CTE" \o "sqlalchemy.sql.expression.CTE), or Common Table Expression instance.

返回一个新的CTE或Common Table Expression实例。

Common table expressions are a SQL standard whereby SELECT statements can draw upon secondary statements specified along with the primary statement, using a clause called "WITH". Special semantics regarding UNION can also be employed to allow "recursive" queries, where a SELECT statement can draw upon the set of rows that have previously been selected.

公用表表达式是一种SQL标准，其中SELECT语句可以使用一个称为"WITH"的子句来使用主语句指定的辅助语句。 还可以使用有关UNION的特殊语义来允许"递归"查询，其中SELECT语句可以对先前已选择的行集合进行绘制。

CTEs can also be applied to DML constructs UPDATE, INSERT and DELETE on some databases, both as a source of CTE rows when combined with RETURNING, as well as a consumer of CTE rows.

CTE还可以应用于某些数据库上的DML构造UPDATE，INSERT和DELETE，作为与RETURNING组合的CTE行的源，以及CTE行的使用者。

SQLAlchemy detects [CTE](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.CTE" \o "sqlalchemy.sql.expression.CTE) objects, which are treated similarly to [Alias](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Alias" \o "sqlalchemy.sql.expression.Alias) objects, as special elements to be delivered to the FROM clause of the statement as well as to a WITH clause at the top of the statement.

SQLAlchemy检测到与Alias对象类似的CTE对象作为要传递到语句的FROM子句的特殊元素以及语句顶部的WITH子句。

*Changed in version 1.1:*Added support for UPDATE/INSERT/DELETE as CTE, CTEs added to UPDATE/INSERT/DELETE.

在版本1.1中更改：添加对UPDATE / INSERT / DELETE作为CTE的支持，CTE添加到UPDATE / INSERT / DELETE。

|  |  |
| --- | --- |
| **Parameters:** | * ****name**** – name given to the common table expression. Like \_FromClause.alias(), the name can be left as None in which case an anonymous symbol will be used at query compile time. * ****recursive**** – if True, will render WITH RECURSIVE. A recursive common table expression is intended to be used in conjunction with UNION ALL in order to derive rows from those already selected. |

The following examples include two from PostgreSQL's documentation at <http://www.postgresql.org/docs/current/static/queries-with.html>, as well as additional examples.

Example 1, non recursive:

**from** **sqlalchemy** **import** (Table, Column, String, Integer,

MetaData, select, func)

metadata = MetaData()

orders = Table('orders', metadata,

Column('region', String),

Column('amount', Integer),

Column('product', String),

Column('quantity', Integer))

regional\_sales = select([

orders.c.region,

func.sum(orders.c.amount).label('total\_sales')

]).group\_by(orders.c.region).cte("regional\_sales")

top\_regions = select([regional\_sales.c.region]).\

where(

regional\_sales.c.total\_sales >

select([

func.sum(regional\_sales.c.total\_sales)/10

])

).cte("top\_regions")

statement = select([

orders.c.region,

orders.c.product,

func.sum(orders.c.quantity).label("product\_units"),

func.sum(orders.c.amount).label("product\_sales")

]).where(orders.c.region.in\_(

select([top\_regions.c.region])

)).group\_by(orders.c.region, orders.c.product)

result = conn.execute(statement).fetchall()

Example 2, WITH RECURSIVE:

**from** **sqlalchemy** **import** (Table, Column, String, Integer,

MetaData, select, func)

metadata = MetaData()

parts = Table('parts', metadata,

Column('part', String),

Column('sub\_part', String),

Column('quantity', Integer),)

included\_parts = select([

parts.c.sub\_part,

parts.c.part,

parts.c.quantity]).\

where(parts.c.part=='our part').\

cte(recursive=**True**)

incl\_alias = included\_parts.alias()parts\_alias = parts.alias()included\_parts = included\_parts.union\_all(

select([

parts\_alias.c.sub\_part,

parts\_alias.c.part,

parts\_alias.c.quantity

]).

where(parts\_alias.c.part==incl\_alias.c.sub\_part))

statement = select([

included\_parts.c.sub\_part,

func.sum(included\_parts.c.quantity).

label('total\_quantity')

]).\

group\_by(included\_parts.c.sub\_part)

result = conn.execute(statement).fetchall()

Example 3, an upsert using UPDATE and INSERT with CTEs:

**from** **datetime** **import** date

**from** **sqlalchemy** **import** (MetaData, Table, Column, Integer,

Date, select, literal, and\_, exists)

metadata = MetaData()

visitors = Table('visitors', metadata,

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

Column('date', Date, primary\_key=**True**),

Column('count', Integer),)

*# add 5 visitors for the product\_id == 1*

product\_id = 1

day = date.today()

count = 5

update\_cte = (

visitors.update()

.where(and\_(visitors.c.product\_id == product\_id,

visitors.c.date == day))

.values(count=visitors.c.count + count)

.returning(literal(1))

.cte('update\_cte'))

upsert = visitors.insert().from\_select(

[visitors.c.product\_id, visitors.c.date, visitors.c.count],

select([literal(product\_id), literal(day), literal(count)])

.where(~exists(update\_cte.select())))

connection.execute(upsert)

**See also**

[orm.query.Query.cte()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.cte" \o "sqlalchemy.orm.query.Query.cte) - ORM version of [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).

**dialect\_kwargs**

*inherited from the* [dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs) *attribute of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

A collection of keyword arguments specified as dialect-specific options to this construct.

指定为此构造的方言特定选项的关键字参数的集合。

The arguments are present here in their original <dialect>\_<kwarg> format. Only arguments that were actually passed are included; unlike the [DialectKWArgs.dialect\_options](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_options" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_options) collection, which contains all options known by this dialect including defaults.

参数以原始<dialect> \_ <kwarg>格式存在。 只包括实际通过的论据; 不同于DialectKWArgs.dialect\_options集合，其中包含此方言已知的所有选项，包括默认值。

The collection is also writable; keys are accepted of the form <dialect>\_<kwarg> where the value will be assembled into the list of options.

收藏也是可写的; 密钥被接受为<dialect> \_ <kwarg>的形式，其中值将被汇总到选项列表中。

*New in version 0.9.2.*

*Changed in version 0.9.4:*The [DialectKWArgs.dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs) collection is now writable.

**See also**

[DialectKWArgs.dialect\_options](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_options" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_options) - nested dictionary form

**dialect\_options**

*inherited from the* [dialect\_options](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_options" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_options) *attribute of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

A collection of keyword arguments specified as dialect-specific options to this construct.

This is a two-level nested registry, keyed to <dialect\_name> and <argument\_name>. For example, the postgresql\_where argument would be locatable as:

arg = my\_object.dialect\_options['postgresql']['where']

*New in version 0.9.2.*

**See also**

[DialectKWArgs.dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs) - flat dictionary form

**execute**(*\*multiparams*, *\*\*params*)

*inherited from the* [execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable.execute" \o "sqlalchemy.sql.expression.Executable.execute) *method of* [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable)

Compile and execute this [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable).

**execution\_options**(*\*\*kw*)

*inherited from the* [execution\_options()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable.execution_options" \o "sqlalchemy.sql.expression.Executable.execution_options) *method of* [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable)

Set non-SQL options for the statement which take effect during execution.

Execution options can be set on a per-statement or per [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) basis. Additionally, the [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) and ORM [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) objects provide access to execution options which they in turn configure upon connections.

The [execution\_options()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Delete.execution_options" \o "sqlalchemy.sql.expression.Delete.execution_options) method is generative. A new instance of this statement is returned that contains the options:

statement = select([table.c.x, table.c.y])statement = statement.execution\_options(autocommit=**True**)

Note that only a subset of possible execution options can be applied to a statement - these include "autocommit" and "stream\_results", but not "isolation\_level" or "compiled\_cache". See [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) for a full list of possible options.

**See also**

[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)

[Query.execution\_options()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.execution_options" \o "sqlalchemy.orm.query.Query.execution_options)

**kwargs**

*inherited from the* [kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.kwargs) *attribute of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

A synonym for [DialectKWArgs.dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs).

**params**(*\*arg*, *\*\*kw*)

*inherited from the* [params()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase.params" \o "sqlalchemy.sql.expression.UpdateBase.params) *method of* [UpdateBase](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase" \o "sqlalchemy.sql.expression.UpdateBase)

Set the parameters for the statement.

This method raises NotImplementedError on the base class, and is overridden by [ValuesBase](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase" \o "sqlalchemy.sql.expression.ValuesBase) to provide the SET/VALUES clause of UPDATE and INSERT.

**prefix\_with**(*\*expr*, *\*\*kw*)

*inherited from the* [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) *method of* [HasPrefixes](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.HasPrefixes" \o "sqlalchemy.sql.expression.HasPrefixes)

Add one or more expressions following the statement keyword, i.e. SELECT, INSERT, UPDATE, or DELETE. Generative.

This is used to support backend-specific prefix keywords such as those provided by MySQL.

E.g.:

stmt = table.insert().prefix\_with("LOW\_PRIORITY", dialect="mysql")

Multiple prefixes can be specified by multiple calls to [prefix\_with()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Delete.prefix_with" \o "sqlalchemy.sql.expression.Delete.prefix_with).

|  |  |
| --- | --- |
| **Parameters:** | * ****\*expr**** – textual or [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement) construct which will be rendered following the INSERT, UPDATE, or DELETE keyword. * ****\*\*kw**** – A single keyword 'dialect' is accepted. This is an optional string dialect name which will limit rendering of this prefix to only that dialect. |

**returning**(*\*cols*)

*inherited from the* [returning()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase.returning" \o "sqlalchemy.sql.expression.UpdateBase.returning) *method of* [UpdateBase](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase" \o "sqlalchemy.sql.expression.UpdateBase)

Add a [RETURNING](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-returning) or equivalent clause to this statement.

e.g.:

stmt = table.update().\

where(table.c.data == 'value').\

values(status='X').\

returning(table.c.server\_flag,

table.c.updated\_timestamp)

**for** server\_flag, updated\_timestamp **in** connection.execute(stmt):

print(server\_flag, updated\_timestamp)

The given collection of column expressions should be derived from the table that is the target of the INSERT, UPDATE, or DELETE. While [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects are typical, the elements can also be expressions:

stmt = table.insert().returning(

(table.c.first\_name + " " + table.c.last\_name).

label('fullname'))

Upon compilation, a RETURNING clause, or database equivalent, will be rendered within the statement. For INSERT and UPDATE, the values are the newly inserted/updated values. For DELETE, the values are those of the rows which were deleted.

编译后，RETURNING子句或数据库等价物将在语句中呈现。对于INSERT和UPDATE，值是新插入/更新的值。对于DELETE，值是被删除的行的值。

Upon execution, the values of the columns to be returned are made available via the result set and can be iterated using [ResultProxy.fetchone()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.fetchone" \o "sqlalchemy.engine.ResultProxy.fetchone)and similar. For DBAPIs which do not natively support returning values (i.e. cx\_oracle), SQLAlchemy will approximate this behavior at the result level so that a reasonable amount of behavioral neutrality is provided.

执行后，要返回的列的值通过结果集可用，并且可以使用ResultProxy.fetchone() 等进行迭代。对于不本地支持返回值（即cx\_oracle）的DBAPI，SQLAlchemy将在结果级别近似此行为，从而提供合理数量的行为中立性。

Note that not all databases/DBAPIs support RETURNING. For those backends with no support, an exception is raised upon compilation and/or execution. For those who do support it, the functionality across backends varies greatly, including restrictions on executemany() and other statements which return multiple rows. Please read the documentation notes for the database in use in order to determine the availability of RETURNING.

请注意，并非所有数据库/ DBAPI都支持RETURNING。对于不支持的后端，在编译和/或执行时会出现异常。对于那些支持它的人，后端的功能差异很大，包括对executemany() 和其他返回多行的语句的限制。请阅读正在使用的数据库的文档说明，以确定RETURNING的可用性。

**See also**

[ValuesBase.return\_defaults()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase.return_defaults" \o "sqlalchemy.sql.expression.ValuesBase.return_defaults) - an alternative method tailored towards efficient fetching of server-side defaults and triggers for single-row INSERTs or UPDATEs.

**scalar**(*\*multiparams*, *\*\*params*)

*inherited from the* [scalar()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable.scalar" \o "sqlalchemy.sql.expression.Executable.scalar) *method of* [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable)

Compile and execute this [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable), returning the result's scalar representation.

**self\_group**(*against=None*)

*inherited from the* [self\_group()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.self_group" \o "sqlalchemy.sql.expression.ClauseElement.self_group) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Apply a 'grouping' to this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement).

对此ClauseElement应用“分组”。

This method is overridden by subclasses to return a "grouping" construct, i.e. parenthesis. In particular it's used by "binary" expressions to provide a grouping around themselves when placed into a larger expression, as well as by [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) constructs when placed into the FROM clause of another [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select). (Note that subqueries should be normally created using the [Select.alias()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.alias" \o "sqlalchemy.sql.expression.Select.alias) method, as many platforms require nested SELECT statements to be named).

这个方法被子类覆盖，以返回一个“分组”结构，即括号。 特别地，它被“二进制”表达式用于在放入更大的表达式时以及通过将select() 结构放入另一个select() 的FROM子句时围绕它们进行分组。 （请注意，通常使用Select.alias() 方法创建子查询，因为许多平台需要嵌套的SELECT语句来命名）。

As expressions are composed together, the application of [self\_group()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Delete.self_group" \o "sqlalchemy.sql.expression.Delete.self_group) is automatic - end-user code should never need to use this method directly. Note that SQLAlchemy's clause constructs take operator precedence into account - so parenthesis might not be needed, for example, in an expression like x OR (y AND z) - AND takes precedence over OR.

由于表达式组合在一起，self\_group() 的应用是自动的 - 最终用户代码不应该直接使用这种方法。 请注意，SQLAlchemy的子句构造考虑到运算符优先级，因此可能不需要括号，例如，在像x OR（y AND z）之类的表达式中，AND优先于OR。

The base [self\_group()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Delete.self_group" \o "sqlalchemy.sql.expression.Delete.self_group) method of [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement) just returns self.

ClauseElement的基本self\_group() 方法只返回自身。

**unique\_params**(*\*optionaldict*, *\*\*kwargs*)

*inherited from the* [unique\_params()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.unique_params" \o "sqlalchemy.sql.expression.ClauseElement.unique_params) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Return a copy with [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) elements replaced.

返回一个带有bindparam() 元素的副本。

Same functionality as params(), except adds unique=True to affected bind parameters so that multiple statements can be used.

与params() 具有相同的功能，除了对受影响的绑定参数添加unique = True，以便可以使用多个语句。

**where**(*whereclause*)

Add the given WHERE clause to a newly returned delete construct.

将给定的WHERE子句添加到新返回的delete构造。

**with\_hint**(*text*, *selectable=None*, *dialect\_name='\*'*)

*inherited from the* [with\_hint()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase.with_hint" \o "sqlalchemy.sql.expression.UpdateBase.with_hint) *method of* [UpdateBase](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase" \o "sqlalchemy.sql.expression.UpdateBase)

Add a table hint for a single table to this INSERT/UPDATE/DELETE statement.

将单个表的表提示添加到此INSERT / UPDATE / DELETE语句。

**Note**

[UpdateBase.with\_hint()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase.with_hint" \o "sqlalchemy.sql.expression.UpdateBase.with_hint) currently applies only to Microsoft SQL Server. For MySQL INSERT/UPDATE/DELETE hints, use [UpdateBase.prefix\_with()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase.prefix_with" \o "sqlalchemy.sql.expression.UpdateBase.prefix_with).

提示的文本在相对于作为此语句主题的表的相应位置呈现在正在使用的数据库后端中，或者可选地作为可选参数传递给给定表的表。

The text of the hint is rendered in the appropriate location for the database backend in use, relative to the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) that is the subject of this statement, or optionally to that of the given [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) passed as the selectable argument.

UpdateBase.with\_hint() 目前仅适用于Microsoft SQL Server。 对于MySQL INSERT / UPDATE / DELETE提示，请使用UpdateBase.prefix\_with() 。

The dialect\_name option will limit the rendering of a particular hint to a particular backend. Such as, to add a hint that only takes effect for SQL Server:

dialect\_name选项将限制对特定后端的特定提示的呈现。 例如，添加仅对SQL Server生效的提示：

mytable.insert().with\_hint("WITH (PAGLOCK)", dialect\_name="mssql")

*New in version 0.7.6.*

|  |  |
| --- | --- |
| **Parameters:** | * ****text**** – Text of the hint. * ****selectable**** – optional [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) that specifies an element of the FROM clause within an UPDATE or DELETE to be the subject of the hint - applies only to certain backends. * ****dialect\_name**** – defaults to \*, if specified as the name of a particular dialect, will apply these hints only when that dialect is in use. |

*class*sqlalchemy.sql.expression.**Insert**(*table*, *values=None*, *inline=False*, *bind=None*, *prefixes=None*, *returning=None*, *return\_defaults=False*, *\*\*dialect\_kw*)

Bases: [sqlalchemy.sql.expression.ValuesBase](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase" \o "sqlalchemy.sql.expression.ValuesBase)

Represent an INSERT construct.

The [Insert](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert" \o "sqlalchemy.sql.expression.Insert) object is created using the [insert()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.insert" \o "sqlalchemy.sql.expression.insert) function.

**See also**

[Insert Expressions](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html" \l "coretutorial-insert-expressions)

**\_\_init\_\_**(*table*, *values=None*, *inline=False*, *bind=None*, *prefixes=None*, *returning=None*, *return\_defaults=False*, *\*\*dialect\_kw*)

Construct a new [Insert](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert" \o "sqlalchemy.sql.expression.Insert) object.

This constructor is mirrored as a public API function; see [insert()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.insert" \o "sqlalchemy.sql.expression.insert) for a full usage and argument description.

**argument\_for**(*dialect\_name*, *argument\_name*, *default*)

*inherited from the* [argument\_for()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.argument_for" \o "sqlalchemy.sql.base.DialectKWArgs.argument_for) *method of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

Add a new kind of dialect-specific keyword argument for this class.

E.g.:

Index.argument\_for("mydialect", "length", **None**)

some\_index = Index('a', 'b', mydialect\_length=5)

The [DialectKWArgs.argument\_for()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.argument_for" \o "sqlalchemy.sql.base.DialectKWArgs.argument_for) method is a per-argument way adding extra arguments to the[DefaultDialect.construct\_arguments](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.default.DefaultDialect.construct_arguments" \o "sqlalchemy.engine.default.DefaultDialect.construct_arguments) dictionary. This dictionary provides a list of argument names accepted by various schema-level constructs on behalf of a dialect.

DialectKWArgs.argument\_for() 方法是向DefaultDialect.construct\_arguments字典添加额外参数的每个参数的方式。 该字典提供了代表方言由各种模式级结构接受的参数名称列表。

New dialects should typically specify this dictionary all at once as a data member of the dialect class. The use case for ad-hoc addition of argument names is typically for end-user code that is also using a custom compilation scheme which consumes the additional arguments.

新的方言通常应该一律将该字典全部指定为方言类的数据成员。 用于临时添加参数名称的用例通常用于最终用户代码，该代码也使用消耗额外参数的自定义编译方案。

|  |  |
| --- | --- |
| **Parameters:** | * ****dialect\_name**** – name of a dialect. The dialect must be locatable, else a [NoSuchModuleError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.NoSuchModuleError" \o "sqlalchemy.exc.NoSuchModuleError) is raised. The dialect must also include an existing[DefaultDialect.construct\_arguments](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.default.DefaultDialect.construct_arguments" \o "sqlalchemy.engine.default.DefaultDialect.construct_arguments) collection, indicating that it participates in the keyword-argument validation and default system, else [ArgumentError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.ArgumentError" \o "sqlalchemy.exc.ArgumentError) is raised. If the dialect does not include this collection, then any keyword argument can be specified on behalf of this dialect already. All dialects packaged within SQLAlchemy include this collection, however for third party dialects, support may vary.一个方言的名字 方言必须是可定位的，否则会引发NoSuchModuleError。 方言还必须包含一个existingDefaultDialect.construct\_arguments集合，表示它参与了关键字参数验证和默认系统，否则引发了ArgumentError。 如果方言不包括此集合，则可以代表此方言指定任何关键字参数。 SQLAlchemy中包装的所有方言都包含此集合，但是对于第三方方言，支持可能会有所不同。 * ****argument\_name**** – name of the parameter. * ****default**** – default value of the parameter. |

*New in version 0.9.4.*

**bind**

*inherited from the* [bind](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase.bind" \o "sqlalchemy.sql.expression.UpdateBase.bind) *attribute of* [UpdateBase](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase" \o "sqlalchemy.sql.expression.UpdateBase)

Return a 'bind' linked to this [UpdateBase](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase" \o "sqlalchemy.sql.expression.UpdateBase) or a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) associated with it.

**compare**(*other*, *\*\*kw*)

*inherited from the* [compare()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.compare" \o "sqlalchemy.sql.expression.ClauseElement.compare) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Compare this ClauseElement to the given ClauseElement.

将此条款元素与给定的条款元素进行比较。

Subclasses should override the default behavior, which is a straight identity comparison.

子类应该覆盖默认行为，这是一个直接的身份比较。

\*\*kw are arguments consumed by subclass compare() methods and may be used to modify the criteria for comparison. (see [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement))

\*\* kw是由subclass compare() 方法消耗的参数，可用于修改比较条件。（见ColumnElement）

**compile**(*bind=None*, *dialect=None*, *\*\*kw*)

*inherited from the* [compile()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.compile" \o "sqlalchemy.sql.expression.ClauseElement.compile) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Compile this SQL expression.

编译此SQL表达式。

The return value is a [Compiled](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Compiled" \o "sqlalchemy.engine.interfaces.Compiled) object. Calling str() or unicode() on the returned value will yield a string representation of the result. The[Compiled](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Compiled" \o "sqlalchemy.engine.interfaces.Compiled) object also can return a dictionary of bind parameter names and values using the params accessor.

返回值是一个编译对象。 对返回的值调用str() 或unicode() 将产生结果的字符串表示形式。 Compiled对象还可以使用params访问器返回绑定参数名称和值的字典。

|  |  |
| --- | --- |
| **Parameters:** | * ****bind**** – An Engine or Connection from which a Compiled will be acquired. This argument takes precedence over this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)'s bound engine, if any. * ****column\_keys**** – Used for INSERT and UPDATE statements, a list of column names which should be present in the VALUES clause of the compiled statement. If None, all columns from the target table object are rendered. * ****dialect**** – A Dialect instance from which a Compiled will be acquired. This argument takes precedence over the bind argument as well as this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)'s bound engine, if any. * ****inline**** – Used for INSERT statements, for a dialect which does not support inline retrieval of newly generated primary key columns, will force the expression used to create the new primary key value to be rendered inline within the INSERT statement's VALUES clause. This typically refers to Sequence execution but may also refer to any server-side default generation function associated with a primary key Column. * ****compile\_kwargs –****optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the literal\_binds flag through:   **from** **sqlalchemy.sql** **import** table, column, select  t = table('t', column('x'))  s = select([t]).where(t.c.x == 5)  print s.compile(compile\_kwargs={"literal\_binds": **True**})  *New in version 0.9.0.* |

**See also**

[How do I render SQL expressions as strings, possibly with bound parameters inlined?](http://docs.sqlalchemy.org/en/rel_1_1/faq/sqlexpressions.html" \l "faq-sql-expression-string)

**cte**(*name=None*, *recursive=False*)

*inherited from the* [cte()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.HasCTE.cte" \o "sqlalchemy.sql.expression.HasCTE.cte) *method of* [HasCTE](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.HasCTE" \o "sqlalchemy.sql.expression.HasCTE)

Return a new [CTE](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.CTE" \o "sqlalchemy.sql.expression.CTE), or Common Table Expression instance.

返回一个新的CTE或Common Table Expression实例。

Common table expressions are a SQL standard whereby SELECT statements can draw upon secondary statements specified along with the primary statement, using a clause called "WITH". Special semantics regarding UNION can also be employed to allow "recursive" queries, where a SELECT statement can draw upon the set of rows that have previously been selected.

公用表表达式是一种SQL标准，其中SELECT语句可以使用一个称为“WITH”的子句来使用主语句指定的辅助语句。 还可以使用有关UNION的特殊语义来允许“递归”查询，其中SELECT语句可以对先前已选择的行集合进行绘制。

CTEs can also be applied to DML constructs UPDATE, INSERT and DELETE on some databases, both as a source of CTE rows when combined with RETURNING, as well as a consumer of CTE rows.

CTE还可以应用于某些数据库上的DML构造UPDATE，INSERT和DELETE，作为与RETURNING组合的CTE行的源，以及CTE行的使用者。

SQLAlchemy detects [CTE](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.CTE" \o "sqlalchemy.sql.expression.CTE) objects, which are treated similarly to [Alias](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Alias" \o "sqlalchemy.sql.expression.Alias) objects, as special elements to be delivered to the FROM clause of the statement as well as to a WITH clause at the top of the statement.

SQLAlchemy检测到与Alias对象类似的CTE对象作为要传递到语句的FROM子句的特殊元素以及语句顶部的WITH子句。

*Changed in version 1.1:*Added support for UPDATE/INSERT/DELETE as CTE, CTEs added to UPDATE/INSERT/DELETE.

在版本1.1中更改：添加对UPDATE / INSERT / DELETE作为CTE的支持，CTE添加到UPDATE / INSERT / DELETE。

|  |  |
| --- | --- |
| **Parameters:** | * ****name**** – name given to the common table expression. Like \_FromClause.alias(), the name can be left as None in which case an anonymous symbol will be used at query compile time.给予公用表表达式的名称。 像\_FromClause.alias() 一样，名称可以保留为None，在这种情况下，在查询编译时将使用匿名符号。 * ****recursive**** – if True, will render WITH RECURSIVE. A recursive common table expression is intended to be used in conjunction with UNION ALL in order to derive rows from those already selected.如果为True，将显示为RECURSIVE。 递归公用表表达式旨在与UNION ALL结合使用，以便从已经选择的表中导出行。 |

The following examples include two from PostgreSQL's documentation at <http://www.postgresql.org/docs/current/static/queries-with.html>, as well as additional examples.以下示例包括PostgreSQL在http://www.postgresql.org/docs/current/static/queries-with.html上的两个文档，以及其他示例。

Example 1, non recursive:

**from** **sqlalchemy** **import** (Table, Column, String, Integer,

MetaData, select, func)

metadata = MetaData()

orders = Table('orders', metadata,

Column('region', String),

Column('amount', Integer),

Column('product', String),

Column('quantity', Integer))

regional\_sales = select([

orders.c.region,

func.sum(orders.c.amount).label('total\_sales')

]).group\_by(orders.c.region).cte("regional\_sales")

top\_regions = select([regional\_sales.c.region]).\

where(

regional\_sales.c.total\_sales >

select([

func.sum(regional\_sales.c.total\_sales)/10

])

).cte("top\_regions")

statement = select([

orders.c.region,

orders.c.product,

func.sum(orders.c.quantity).label("product\_units"),

func.sum(orders.c.amount).label("product\_sales")

]).where(orders.c.region.in\_(

select([top\_regions.c.region])

)).group\_by(orders.c.region, orders.c.product)

result = conn.execute(statement).fetchall()

Example 2, WITH RECURSIVE:

**from** **sqlalchemy** **import** (Table, Column, String, Integer,

MetaData, select, func)

metadata = MetaData()

parts = Table('parts', metadata,

Column('part', String),

Column('sub\_part', String),

Column('quantity', Integer),)

included\_parts = select([

parts.c.sub\_part,

parts.c.part,

parts.c.quantity]).\

where(parts.c.part=='our part').\

cte(recursive=**True**)

incl\_alias = included\_parts.alias()parts\_alias = parts.alias()included\_parts = included\_parts.union\_all(

select([

parts\_alias.c.sub\_part,

parts\_alias.c.part,

parts\_alias.c.quantity

]).

where(parts\_alias.c.part==incl\_alias.c.sub\_part))

statement = select([

included\_parts.c.sub\_part,

func.sum(included\_parts.c.quantity).

label('total\_quantity')

]).\

group\_by(included\_parts.c.sub\_part)

result = conn.execute(statement).fetchall()

Example 3, an upsert using UPDATE and INSERT with CTEs:

**from** **datetime** **import** date**from** **sqlalchemy** **import** (MetaData, Table, Column, Integer,

Date, select, literal, and\_, exists)

metadata = MetaData()

visitors = Table('visitors', metadata,

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

Column('date', Date, primary\_key=**True**),

Column('count', Integer),)

*# add 5 visitors for the product\_id == 1*product\_id = 1day = date.today()count = 5

update\_cte = (

visitors.update()

.where(and\_(visitors.c.product\_id == product\_id,

visitors.c.date == day))

.values(count=visitors.c.count + count)

.returning(literal(1))

.cte('update\_cte'))

upsert = visitors.insert().from\_select(

[visitors.c.product\_id, visitors.c.date, visitors.c.count],

select([literal(product\_id), literal(day), literal(count)])

.where(~exists(update\_cte.select())))

connection.execute(upsert)

**See also**

[orm.query.Query.cte()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.cte" \o "sqlalchemy.orm.query.Query.cte) - ORM version of [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).

**dialect\_kwargs**

*inherited from the* [dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs) *attribute of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

A collection of keyword arguments specified as dialect-specific options to this construct.

指定为此构造的方言特定选项的关键字参数的集合。

The arguments are present here in their original <dialect>\_<kwarg> format. Only arguments that were actually passed are included; unlike the [DialectKWArgs.dialect\_options](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_options" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_options) collection, which contains all options known by this dialect including defaults.

参数以原始<dialect> \_ <kwarg>格式存在。 只包括实际通过的论据; 不同于DialectKWArgs.dialect\_options集合，其中包含此方言已知的所有选项，包括默认值。

The collection is also writable; keys are accepted of the form <dialect>\_<kwarg> where the value will be assembled into the list of options.

收藏也是可写的; 密钥被接受为<dialect> \_ <kwarg>的形式，其中值将被汇总到选项列表中。

*New in version 0.9.2.*

*Changed in version 0.9.4:*The [DialectKWArgs.dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs) collection is now writable.

**See also**

[DialectKWArgs.dialect\_options](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_options" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_options) - nested dictionary form

**dialect\_options**

*inherited from the* [dialect\_options](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_options" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_options) *attribute of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

A collection of keyword arguments specified as dialect-specific options to this construct.

指定为此构造的方言特定选项的关键字参数的集合。

This is a two-level nested registry, keyed to <dialect\_name> and <argument\_name>. For example, the postgresql\_where argument would be locatable as:

这是一个两级嵌套注册表，键入<dialect\_name>和<argument\_name>。 例如，postgresql\_where参数可以定位为：

arg = my\_object.dialect\_options['postgresql']['where']

*New in version 0.9.2.*

**See also**

[DialectKWArgs.dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs) - flat dictionary form

**execute**(*\*multiparams*, *\*\*params*)

*inherited from the* [execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable.execute" \o "sqlalchemy.sql.expression.Executable.execute) *method of* [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable)

Compile and execute this [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable).

编译并执行此可执行文件。

**execution\_options**(*\*\*kw*)

*inherited from the* [execution\_options()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable.execution_options" \o "sqlalchemy.sql.expression.Executable.execution_options) *method of* [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable)

Set non-SQL options for the statement which take effect during execution.

为执行期间生效的语句设置非SQL选项。

Execution options can be set on a per-statement or per [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) basis. Additionally, the [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) and ORM [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) objects provide access to execution options which they in turn configure upon connections.

可以在每个语句或每个连接的基础上设置执行选项。 此外，引擎和ORM查询对象提供对执行选项的访问，它们在连接时依次配置。

The [execution\_options()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert.execution_options" \o "sqlalchemy.sql.expression.Insert.execution_options) method is generative. A new instance of this statement is returned that contains the options:

execution\_options() 方法是生成的。 此语句的新实例将返回，其中包含以下选项：

statement = select([table.c.x, table.c.y])statement = statement.execution\_options(autocommit=**True**)

Note that only a subset of possible execution options can be applied to a statement - these include "autocommit" and "stream\_results", but not "isolation\_level" or "compiled\_cache". See [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) for a full list of possible options.

请注意，只有一个可能的执行选项的子集可以应用于一个语句 - 这些包括“autocommit”和“stream\_results”，而不是“isolation\_level”或“compiled\_cache”。 有关可能的选项的完整列表，请参阅Connection.execution\_options() 。

**See also**

[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)

[Query.execution\_options()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.execution_options" \o "sqlalchemy.orm.query.Query.execution_options)

**from\_select**(*names*, *select*, *include\_defaults=True*)

Return a new [Insert](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert" \o "sqlalchemy.sql.expression.Insert) construct which represents an INSERT...FROM SELECT statement.

e.g.:

sel = select([table1.c.a, table1.c.b]).where(table1.c.c > 5)ins = table2.insert().from\_select(['a', 'b'], sel)

|  |  |
| --- | --- |
| **Parameters:** | * ****names**** – a sequence of string column names or [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects representing the target columns.表示目标列的字符串列名称或列对象序列。 * ****select**** – a [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) construct, [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) or other construct which resolves into a [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause), such as an ORM [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) object, etc. The order of columns returned from this FROM clause should correspond to the order of columns sent as the names parameter; while this is not checked before passing along to the database, the database would normally raise an exception if these column lists don't correspond.一个select() 构造，FromClause或其他构造，它解析为一个FromClause，如ORM Query对象等。从这个FROM子句返回的列的顺序应该与作为names参数发送的列的顺序相对应。 而在传递到数据库之前，这并不被检查，如果这些列列表不对应，数据库通常会引发异常。 * ****include\_defaults –****if True, non-server default values and SQL expressions as specified on [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects (as documented in [Column Insert/Update Defaults](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html)) not otherwise specified in the list of names will be rendered into the INSERT and SELECT statements, so that these values are also included in the data to be inserted.如果为True，则在名称列表中没有另外指定的列对象（如列插入/更新默认值中所述）中指定的非服务器默认值和SQL表达式将被呈现为INSERT和SELECT语句，因此这些值也 包含在要插入的数据中   **Note**  A Python-side default that uses a Python callable function will only be invoked ****once**** for the whole statement, and ****not per row****.  使用Python可调用函数的Python端缺省值只能针对整个语句而不是每行调用一次。  *New in version 1.0.0:*- [Insert.from\_select()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert.from_select" \o "sqlalchemy.sql.expression.Insert.from_select) now renders Python-side and SQL expression column defaults into the SELECT statement for columns otherwise not included in the list of column names.  新版本1.0.0： - Insert.from\_select() 现在将Python边和SQL表达式列默认为SELECT列，否则不包括在列名列表中。 |

*Changed in version 1.0.0:*an INSERT that uses FROM SELECT implies that the [insert.inline](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html" \l "sqlalchemy.ext.mutable.MutableList.insert.params.inline" \o "sqlalchemy.ext.mutable.MutableList.insert) flag is set to True, indicating that the statement will not attempt to fetch the "last inserted primary key" or other defaults. The statement deals with an arbitrary number of rows, so the [ResultProxy.inserted\_primary\_key](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.inserted_primary_key" \o "sqlalchemy.engine.ResultProxy.inserted_primary_key)accessor does not apply.

在版本1.0.0中更改：使用FROM SELECT的INSERT意味着insert.inline标志设置为True，表示该语句不会尝试获取“最后插入的主键”或其他默认值。 该语句处理任意数量的行，因此ResultProxy.inserted\_primary\_keyaccessor不适用。

*New in version 0.8.3.*

**kwargs**

*inherited from the* [kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.kwargs) *attribute of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

A synonym for [DialectKWArgs.dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs).

**params**(*\*arg*, *\*\*kw*)

*inherited from the* [params()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase.params" \o "sqlalchemy.sql.expression.UpdateBase.params) *method of* [UpdateBase](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase" \o "sqlalchemy.sql.expression.UpdateBase)

Set the parameters for the statement.

设置语句的参数。

This method raises NotImplementedError on the base class, and is overridden by [ValuesBase](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase" \o "sqlalchemy.sql.expression.ValuesBase) to provide the SET/VALUES clause of UPDATE and INSERT.

此方法在基类上引发NotImplementedError，并由ValuesBase覆盖以提供UPDATE和INSERT的SET / VALUES子句。

**prefix\_with**(*\*expr*, *\*\*kw*)

*inherited from the* [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) *method of* [HasPrefixes](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.HasPrefixes" \o "sqlalchemy.sql.expression.HasPrefixes)

Add one or more expressions following the statement keyword, i.e. SELECT, INSERT, UPDATE, or DELETE. Generative.

在语句关键字之后添加一个或多个表达式，即SELECT，INSERT，UPDATE或DELETE。生成。

This is used to support backend-specific prefix keywords such as those provided by MySQL.

这用于支持后端特定的前缀关键字，如MySQL提供的那些。

E.g.:

stmt = table.insert().prefix\_with("LOW\_PRIORITY", dialect="mysql")

Multiple prefixes can be specified by multiple calls to [prefix\_with()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert.prefix_with" \o "sqlalchemy.sql.expression.Insert.prefix_with).

可以通过对prefix\_with() 的多次调用指定多个前缀。

|  |  |
| --- | --- |
| **Parameters:** | * ****\*expr**** – textual or [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement) construct which will be rendered following the INSERT, UPDATE, or DELETE keyword. * ****\*\*kw**** – A single keyword 'dialect' is accepted. This is an optional string dialect name which will limit rendering of this prefix to only that dialect. |

**return\_defaults**(*\*cols*)

*inherited from the* [return\_defaults()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase.return_defaults" \o "sqlalchemy.sql.expression.ValuesBase.return_defaults) *method of* [ValuesBase](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase" \o "sqlalchemy.sql.expression.ValuesBase)

Make use of a [RETURNING](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-returning) clause for the purpose of fetching server-side expressions and defaults.

为了获取服务器端表达式和默认值，使用RETURNING子句。

E.g.:

stmt = table.insert().values(data='newdata').return\_defaults()

result = connection.execute(stmt)

server\_created\_at = result.returned\_defaults['created\_at']

When used against a backend that supports RETURNING, all column values generated by SQL expression or server-side-default will be added to any existing RETURNING clause, provided that [UpdateBase.returning()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase.returning" \o "sqlalchemy.sql.expression.UpdateBase.returning) is not used simultaneously. The column values will then be available on the result using the [ResultProxy.returned\_defaults](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.returned_defaults" \o "sqlalchemy.engine.ResultProxy.returned_defaults) accessor as a dictionary, referring to values keyed to the [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) object as well as its .key.

当用于支持RETURNING的后端时，由SQL表达式或服务器端默认生成的所有列值都将添加到任何现有的RETURNING子句中，前提是不会同时使用UpdateBase.returning() 。 然后，列值将使用ResultProxy.returned\_defaults访问器作为字典在结果上可用，引用键对Column对象及其.key的值。

This method differs from [UpdateBase.returning()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase.returning" \o "sqlalchemy.sql.expression.UpdateBase.returning) in these ways:

此方法与UpdateBase.returning() 不同之处在于：

1. [ValuesBase.return\_defaults()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase.return_defaults" \o "sqlalchemy.sql.expression.ValuesBase.return_defaults) is only intended for use with an INSERT or an UPDATE statement that matches exactly one row. While the RETURNING construct in the general sense supports multiple rows for a multi-row UPDATE or DELETE statement, or for special cases of INSERT that return multiple rows (e.g. INSERT from SELECT, multi-valued VALUES clause), [ValuesBase.return\_defaults()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase.return_defaults" \o "sqlalchemy.sql.expression.ValuesBase.return_defaults) is intended only for an "ORM-style" single-row INSERT/UPDATE statement. The row returned by the statement is also consumed implicitly when[ValuesBase.return\_defaults()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase.return_defaults" \o "sqlalchemy.sql.expression.ValuesBase.return_defaults) is used. By contrast, [UpdateBase.returning()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase.returning" \o "sqlalchemy.sql.expression.UpdateBase.returning) leaves the RETURNING result-set intact with a collection of any number of rows.ValuesBase.return\_defaults() 仅用于与完全匹配一行的INSERT或UPDATE语句一起使用。 虽然RETURNING结构在一般意义上支持多行UPDATE或DELETE语句的多行，或对于返回多行的特殊情况（例如INSERT从SELECT，多值VALUES子句），ValuesBase.return\_defaults() 是 仅用于“ORM风格”单行INSERT / UPDATE语句。 当使用ValuesBase.return\_defaults() 时，语句返回的行也被隐含地消耗。 相比之下，UpdateBase.returning() 将使用任意数量的行的集合保留RETURNING结果集。
2. It is compatible with the existing logic to fetch auto-generated primary key values, also known as "implicit returning". Backends that support RETURNING will automatically make use of RETURNING in order to fetch the value of newly generated primary keys; while the[UpdateBase.returning()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase.returning" \o "sqlalchemy.sql.expression.UpdateBase.returning) method circumvents this behavior, [ValuesBase.return\_defaults()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase.return_defaults" \o "sqlalchemy.sql.expression.ValuesBase.return_defaults) leaves it intact.它与现有逻辑兼容，以获取自动生成的主键值，也称为“隐式返回”。 支持RETURNING的后端将自动使用RETURNING来获取新生成的主键的值; 而UpdateBase.returning() 方法规避了此行为，ValuesBase.return\_defaults() 将保持原样。
3. It can be called against any backend. Backends that don't support RETURNING will skip the usage of the feature, rather than raising an exception. The return value of [ResultProxy.returned\_defaults](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.returned_defaults" \o "sqlalchemy.engine.ResultProxy.returned_defaults) will be None可以调用任何后端。 不支持RETURNING的后端将跳过该功能的使用，而不是引发异常。 ResultProxy.returned\_defaults的返回值将为None

[ValuesBase.return\_defaults()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase.return_defaults" \o "sqlalchemy.sql.expression.ValuesBase.return_defaults) is used by the ORM to provide an efficient implementation for the eager\_defaults feature of [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper).

ValuesBase.return\_defaults() 由ORM用于为mapper() 的eager\_defaults功能提供有效的实现。

|  |  |
| --- | --- |
| **Parameters:** | ****cols**** – optional list of column key names or [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects. If omitted, all column expressions evaluated on the server are added to the returning list.列键名称或列对象的可选列表。 如果省略，则在服务器上计算的所有列表达式都将添加到返回列表中。 |

*New in version 0.9.0.*

**See also**

[UpdateBase.returning()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase.returning" \o "sqlalchemy.sql.expression.UpdateBase.returning)

[ResultProxy.returned\_defaults](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.returned_defaults" \o "sqlalchemy.engine.ResultProxy.returned_defaults)

**returning**(*\*cols*)

*inherited from the* [returning()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase.returning" \o "sqlalchemy.sql.expression.UpdateBase.returning) *method of* [UpdateBase](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase" \o "sqlalchemy.sql.expression.UpdateBase)

Add a [RETURNING](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-returning) or equivalent clause to this statement.

e.g.:

stmt = table.update().\

where(table.c.data == 'value').\

values(status='X').\

returning(table.c.server\_flag,

table.c.updated\_timestamp)

**for** server\_flag, updated\_timestamp **in** connection.execute(stmt):

print(server\_flag, updated\_timestamp)

The given collection of column expressions should be derived from the table that is the target of the INSERT, UPDATE, or DELETE. While [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects are typical, the elements can also be expressions:

列表达式的给定集合应该从作为INSERT，UPDATE或DELETE的目标的表导出。 虽然[Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column)对象是典型的，但元素也可以是表达式：

stmt = table.insert().returning(

(table.c.first\_name + " " + table.c.last\_name).

label('fullname'))

Upon compilation, a RETURNING clause, or database equivalent, will be rendered within the statement. For INSERT and UPDATE, the values are the newly inserted/updated values. For DELETE, the values are those of the rows which were deleted.

编译后，RETURNING子句或数据库等价物将在语句中呈现。对于INSERT和UPDATE，值是新插入/更新的值。对于DELETE，值是被删除的行的值。

Upon execution, the values of the columns to be returned are made available via the result set and can be iterated using [ResultProxy.fetchone()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.fetchone" \o "sqlalchemy.engine.ResultProxy.fetchone) and similar. For DBAPIs which do not natively support returning values (i.e. cx\_oracle), SQLAlchemy will approximate this behavior at the result level so that a reasonable amount of behavioral neutrality is provided.

执行后，要返回的列的值通过结果集可用，并且可以使用[ResultProxy.fetchone()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.fetchone" \o "sqlalchemy.engine.ResultProxy.fetchone) 等进行迭代。对于不本地支持返回值（即cx\_oracle）的DBAPI，SQLAlchemy将在结果级别近似此行为，从而提供合理数量的行为中立性。

Note that not all databases/DBAPIs support RETURNING. For those backends with no support, an exception is raised upon compilation and/or execution. For those who do support it, the functionality across backends varies greatly, including restrictions on executemany() and other statements which return multiple rows. Please read the documentation notes for the database in use in order to determine the availability of RETURNING.

请注意，并非所有数据库/ DBAPI都支持RETURNING。对于不支持的后端，在编译和/或执行时会出现异常。对于那些支持它的人，后端的功能差异很大，包括对executemany() 和其他返回多行的语句的限制。请阅读正在使用的数据库的文档说明，以确定RETURNING的可用性。

**See also**

[ValuesBase.return\_defaults()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase.return_defaults" \o "sqlalchemy.sql.expression.ValuesBase.return_defaults) - an alternative method tailored towards efficient fetching of server-side defaults and triggers for single-row INSERTs or UPDATEs.

[ValuesBase.return\_defaults()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase.return_defaults" \o "sqlalchemy.sql.expression.ValuesBase.return_defaults) - 一种替代方法，用于有效提取单行INSERT或UPDATE的服务器端默认值和触发器。

**scalar**(*\*multiparams*, *\*\*params*)

*inherited from the* [scalar()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable.scalar" \o "sqlalchemy.sql.expression.Executable.scalar) *method of* [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable)

Compile and execute this [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable), returning the result's scalar representation.

编译并执行此可执行文件，返回结果标量表示。

**self\_group**(*against=None*)

*inherited from the* [self\_group()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.self_group" \o "sqlalchemy.sql.expression.ClauseElement.self_group) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Apply a 'grouping' to this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement).

对此ClauseElement应用“分组”。

This method is overridden by subclasses to return a "grouping" construct, i.e. parenthesis. In particular it's used by "binary" expressions to provide a grouping around themselves when placed into a larger expression, as well as by [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) constructs when placed into the FROM clause of another [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select). (Note that subqueries should be normally created using the [Select.alias()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.alias" \o "sqlalchemy.sql.expression.Select.alias) method, as many platforms require nested SELECT statements to be named).

这个方法被子类覆盖，以返回一个“分组”结构，即括号。 特别地，它被“二进制”表达式用于在放入更大的表达式时以及通过将select() 结构放入另一个select() 的FROM子句时围绕它们进行分组。 （请注意，通常使用Select.alias() 方法创建子查询，因为许多平台需要嵌套的SELECT语句来命名）。

As expressions are composed together, the application of [self\_group()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert.self_group" \o "sqlalchemy.sql.expression.Insert.self_group) is automatic - end-user code should never need to use this method directly. Note that SQLAlchemy's clause constructs take operator precedence into account - so parenthesis might not be needed, for example, in an expression like x OR (y AND z) - AND takes precedence over OR.

由于表达式组合在一起，self\_group() 的应用是自动的 - 最终用户代码不应该直接使用这种方法。 请注意，SQLAlchemy的子句构造考虑到运算符优先级，因此可能不需要括号，例如，在像x OR（y AND z）之类的表达式中，AND优先于OR。

The base [self\_group()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert.self_group" \o "sqlalchemy.sql.expression.Insert.self_group) method of [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement) just returns self.

ClauseElement的基本self\_group() 方法只返回自身。

**unique\_params**(*\*optionaldict*, *\*\*kwargs*)

*inherited from the* [unique\_params()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.unique_params" \o "sqlalchemy.sql.expression.ClauseElement.unique_params) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Return a copy with [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) elements replaced.

返回一个带有bindparam() 元素的副本。

Same functionality as params(), except adds unique=True to affected bind parameters so that multiple statements can be used.

与params() 具有相同的功能，除了对受影响的绑定参数添加unique = True，以便可以使用多个语句。

**values**(*\*args*, *\*\*kwargs*)

*inherited from the* [values()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase.values" \o "sqlalchemy.sql.expression.ValuesBase.values) *method of* [ValuesBase](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase" \o "sqlalchemy.sql.expression.ValuesBase)

specify a fixed VALUES clause for an INSERT statement, or the SET clause for an UPDATE.

Note that the [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 support per-execution time formatting of the VALUES and/or SET clauses, based on the arguments passed to [Connection.execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.execute" \o "sqlalchemy.engine.Connection.execute). However, the [ValuesBase.values()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase.values" \o "sqlalchemy.sql.expression.ValuesBase.values) method can be used to "fix" a particular set of parameters into the statement.

Multiple calls to [ValuesBase.values()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase.values" \o "sqlalchemy.sql.expression.ValuesBase.values) will produce a new construct, each one with the parameter list modified to include the new parameters sent. In the typical case of a single dictionary of parameters, the newly passed keys will replace the same keys in the previous construct. In the case of a list-based "multiple values" construct, each new list of values is extended onto the existing list of values.

|  |  |
| --- | --- |
| **Parameters:** | * ****\*\*kwargs**** –   key value pairs representing the string key of a [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) mapped to the value to be rendered into the VALUES or SET clause:  users.insert().values(name="some name")  users.update().where(users.c.id==5).values(name="some name")   * ****\*args**** –   As an alternative to passing key/value parameters, a dictionary, tuple, or list of dictionaries or tuples can be passed as a single positional argument in order to form the VALUES or SET clause of the statement. The forms that are accepted vary based on whether this is an [Insert](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert" \o "sqlalchemy.sql.expression.Insert) or an [Update](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Update" \o "sqlalchemy.sql.expression.Update)construct.  For either an [Insert](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert" \o "sqlalchemy.sql.expression.Insert) or [Update](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Update" \o "sqlalchemy.sql.expression.Update) construct, a single dictionary can be passed, which works the same as that of the kwargs form:  users.insert().values({"name": "some name"})  users.update().values({"name": "some new name"})  Also for either form but more typically for the [Insert](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert" \o "sqlalchemy.sql.expression.Insert) construct, a tuple that contains an entry for every column in the table is also accepted:  users.insert().values((5, "some name"))  The [Insert](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert" \o "sqlalchemy.sql.expression.Insert) construct also supports being passed a list of dictionaries or full-table-tuples, which on the server will render the less common SQL syntax of "multiple values" - this syntax is supported on backends such as SQLite, PostgreSQL, MySQL, but not necessarily others:  users.insert().values([  {"name": "some name"},  {"name": "some other name"},  {"name": "yet another name"},  ])  The above form would render a multiple VALUES statement similar to:  INSERT INTO users (name) VALUES  (:name\_1),  (:name\_2),  (:name\_3)  It is essential to note that ****passing multiple values is NOT the same as using traditional executemany() form****. The above syntax is a ****special**** syntax not typically used. To emit an INSERT statement against multiple rows, the normal method is to pass a multiple values list to the [Connection.execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.execute" \o "sqlalchemy.engine.Connection.execute) method, which is supported by all database backends and is generally more efficient for a very large number of parameters.  **See also**  [Executing Multiple Statements](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html" \l "execute-multiple) - an introduction to the traditional Core method of multiple parameter set invocation for INSERTs and other statements.  *Changed in version 1.0.0:*an INSERT that uses a multiple-VALUES clause, even a list of length one, implies that the [Insert.inline](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert.params.inline" \o "sqlalchemy.sql.expression.Insert) flag is set to True, indicating that the statement will not attempt to fetch the "last inserted primary key" or other defaults. The statement deals with an arbitrary number of rows, so the [ResultProxy.inserted\_primary\_key](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.inserted_primary_key" \o "sqlalchemy.engine.ResultProxy.inserted_primary_key) accessor does not apply.  *Changed in version 1.0.0:*A multiple-VALUES INSERT now supports columns with Python side default values and callables in the same way as that of an "executemany" style of invocation; the callable is invoked for each row. See [Python-side defaults invoked for each row invidually when using a multivalued insert](http://docs.sqlalchemy.org/en/rel_1_1/changelog/migration_10.html" \l "bug-3288) for other details.  The [Update](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Update" \o "sqlalchemy.sql.expression.Update) construct supports a special form which is a list of 2-tuples, which when provided must be passed in conjunction with 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) parameter. This form causes the UPDATE statement to render the SET clauses using the order of parameters given to [Update.values()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Update.values" \o "sqlalchemy.sql.expression.Update.values), rather than the ordering of columns given in the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table).  *New in version 1.0.10:*- added support for parameter-ordered UPDATE statements via 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.  **See also**  [Parameter-Ordered Updates](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html" \l "updates-order-parameters) - full example of 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 |

**See also**

[Inserts, Updates and Deletes](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html" \l "inserts-and-updates) - SQL Expression Language Tutorial

[insert()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.insert" \o "sqlalchemy.sql.expression.insert) - produce an INSERT statement

[update()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.update" \o "sqlalchemy.sql.expression.update) - produce an UPDATE statement

**with\_hint**(*text*, *selectable=None*, *dialect\_name='\*'*)

*inherited from the* [with\_hint()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase.with_hint" \o "sqlalchemy.sql.expression.UpdateBase.with_hint) *method of* [UpdateBase](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase" \o "sqlalchemy.sql.expression.UpdateBase)

Add a table hint for a single table to this INSERT/UPDATE/DELETE statement.

**Note**

[UpdateBase.with\_hint()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase.with_hint" \o "sqlalchemy.sql.expression.UpdateBase.with_hint) currently applies only to Microsoft SQL Server. For MySQL INSERT/UPDATE/DELETE hints, use [UpdateBase.prefix\_with()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase.prefix_with" \o "sqlalchemy.sql.expression.UpdateBase.prefix_with).

The text of the hint is rendered in the appropriate location for the database backend in use, relative to the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) that is the subject of this statement, or optionally to that of the given [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) passed as the selectable argument.

The dialect\_name option will limit the rendering of a particular hint to a particular backend. Such as, to add a hint that only takes effect for SQL Server:

mytable.insert().with\_hint("WITH (PAGLOCK)", dialect\_name="mssql")

*New in version 0.7.6.*

|  |  |
| --- | --- |
| **Parameters:** | * ****text**** – Text of the hint. * ****selectable**** – optional [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) that specifies an element of the FROM clause within an UPDATE or DELETE to be the subject of the hint - applies only to certain backends. * ****dialect\_name**** – defaults to \*, if specified as the name of a particular dialect, will apply these hints only when that dialect is in use. |

*class*sqlalchemy.sql.expression.**Update**(*table*, *whereclause=None*, *values=None*, *inline=False*, *bind=None*, *prefixes=None*, *returning=None*, *return\_defaults=False*, *preserve\_parameter\_order=False*, *\*\*dialect\_kw*)

Bases: [sqlalchemy.sql.expression.ValuesBase](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase" \o "sqlalchemy.sql.expression.ValuesBase)

Represent an Update construct.

The [Update](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Update" \o "sqlalchemy.sql.expression.Update) object is created using the [update()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.update" \o "sqlalchemy.sql.expression.update) function.

**\_\_init\_\_**(*table*, *whereclause=None*, *values=None*, *inline=False*, *bind=None*, *prefixes=None*, *returning=None*, *return\_defaults=False*, *preserve\_parameter\_order=False*, *\*\*dialect\_kw*)

Construct a new [Update](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Update" \o "sqlalchemy.sql.expression.Update) object.

This constructor is mirrored as a public API function; see [update()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.update" \o "sqlalchemy.sql.expression.update) for a full usage and argument description.

**argument\_for**(*dialect\_name*, *argument\_name*, *default*)

*inherited from the* [argument\_for()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.argument_for" \o "sqlalchemy.sql.base.DialectKWArgs.argument_for) *method of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

Add a new kind of dialect-specific keyword argument for this class.

E.g.:

Index.argument\_for("mydialect", "length", **None**)

some\_index = Index('a', 'b', mydialect\_length=5)

The [DialectKWArgs.argument\_for()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.argument_for" \o "sqlalchemy.sql.base.DialectKWArgs.argument_for) method is a per-argument way adding extra arguments to the[DefaultDialect.construct\_arguments](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.default.DefaultDialect.construct_arguments" \o "sqlalchemy.engine.default.DefaultDialect.construct_arguments) dictionary. This dictionary provides a list of argument names accepted by various schema-level constructs on behalf of a dialect.

[DialectKWArgs.argument\_for()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.argument_for" \o "sqlalchemy.sql.base.DialectKWArgs.argument_for)方法是一种按参数方式向[DefaultDialect.construct\_arguments](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.default.DefaultDialect.construct_arguments" \o "sqlalchemy.engine.default.DefaultDialect.construct_arguments)字典添加额外参数的方法。 这个字典提供了一个由各种模式级别的结构代表一个方言接受的参数名称列表。

New dialects should typically specify this dictionary all at once as a data member of the dialect class. The use case for ad-hoc addition of argument names is typically for end-user code that is also using a custom compilation scheme which consumes the additional arguments.

新的方言通常应该将这个字典一次性地指定为方言类的数据成员。 用于临时添加参数名称的用例通常用于最终用户代码，该代码也使用了使用额外参数的自定义编译方案。

|  |  |
| --- | --- |
| **Parameters:** | * ****dialect\_name**** – name of a dialect. The dialect must be locatable, else a [NoSuchModuleError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.NoSuchModuleError" \o "sqlalchemy.exc.NoSuchModuleError) is raised. The dialect must also include an existing[DefaultDialect.construct\_arguments](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.default.DefaultDialect.construct_arguments" \o "sqlalchemy.engine.default.DefaultDialect.construct_arguments) collection, indicating that it participates in the keyword-argument validation and default system, else [ArgumentError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.ArgumentError" \o "sqlalchemy.exc.ArgumentError) is raised. If the dialect does not include this collection, then any keyword argument can be specified on behalf of this dialect already. All dialects packaged within SQLAlchemy include this collection, however for third party dialects, support may vary.一个方言的名字。 方言必须是可定位的，否则会引发NoSuchModuleError。 该方言还必须包含一个existingDefaultDialect.construct\_arguments集合，指示它参与关键字参数验证和默认系统，否则引发ArgumentError。 如果方言不包括这个集合，那么任何关键字参数都可以代表这个方言指定。 所有包含在SQLAlchemy中的方言都包括这个集合，但是对于第三方方言，支持可能会有所不同。 * ****argument\_name**** – name of the parameter. * ****default**** – default value of the parameter. |

*New in version 0.9.4.*

**bind**

*inherited from the* [bind](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase.bind" \o "sqlalchemy.sql.expression.UpdateBase.bind) *attribute of* [UpdateBase](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase" \o "sqlalchemy.sql.expression.UpdateBase)

Return a 'bind' linked to this [UpdateBase](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase" \o "sqlalchemy.sql.expression.UpdateBase) or a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) associated with it.

**compare**(*other*, *\*\*kw*)

*inherited from the* [compare()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.compare" \o "sqlalchemy.sql.expression.ClauseElement.compare) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Compare this ClauseElement to the given ClauseElement.

Subclasses should override the default behavior, which is a straight identity comparison.

\*\*kw are arguments consumed by subclass compare() methods and may be used to modify the criteria for comparison. (see [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement))

**compile**(*bind=None*, *dialect=None*, *\*\*kw*)

*inherited from the* [compile()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.compile" \o "sqlalchemy.sql.expression.ClauseElement.compile) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Compile this SQL expression.

The return value is a [Compiled](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Compiled" \o "sqlalchemy.engine.interfaces.Compiled) object. Calling str() or unicode() on the returned value will yield a string representation of the result. The[Compiled](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Compiled" \o "sqlalchemy.engine.interfaces.Compiled) object also can return a dictionary of bind parameter names and values using the params accessor.

|  |  |
| --- | --- |
| **Parameters:** | * ****bind**** – An Engine or Connection from which a Compiled will be acquired. This argument takes precedence over this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)'s bound engine, if any. * ****column\_keys**** – Used for INSERT and UPDATE statements, a list of column names which should be present in the VALUES clause of the compiled statement. If None, all columns from the target table object are rendered. * ****dialect**** – A Dialect instance from which a Compiled will be acquired. This argument takes precedence over the bind argument as well as this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)'s bound engine, if any. * ****inline**** – Used for INSERT statements, for a dialect which does not support inline retrieval of newly generated primary key columns, will force the expression used to create the new primary key value to be rendered inline within the INSERT statement's VALUES clause. This typically refers to Sequence execution but may also refer to any server-side default generation function associated with a primary key Column. * ****compile\_kwargs –****optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the literal\_binds flag through:   **from** **sqlalchemy.sql** **import** table, column, select  t = table('t', column('x'))  s = select([t]).where(t.c.x == 5)  print s.compile(compile\_kwargs={"literal\_binds": **True**})  *New in version 0.9.0.* |

**See also**

[How do I render SQL expressions as strings, possibly with bound parameters inlined?](http://docs.sqlalchemy.org/en/rel_1_1/faq/sqlexpressions.html" \l "faq-sql-expression-string)

**cte**(*name=None*, *recursive=False*)

*inherited from the* [cte()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.HasCTE.cte" \o "sqlalchemy.sql.expression.HasCTE.cte) *method of* [HasCTE](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.HasCTE" \o "sqlalchemy.sql.expression.HasCTE)

Return a new [CTE](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.CTE" \o "sqlalchemy.sql.expression.CTE), or Common Table Expression instance.

Common table expressions are a SQL standard whereby SELECT statements can draw upon secondary statements specified along with the primary statement, using a clause called "WITH". Special semantics regarding UNION can also be employed to allow "recursive" queries, where a SELECT statement can draw upon the set of rows that have previously been selected.

CTEs can also be applied to DML constructs UPDATE, INSERT and DELETE on some databases, both as a source of CTE rows when combined with RETURNING, as well as a consumer of CTE rows.

SQLAlchemy detects [CTE](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.CTE" \o "sqlalchemy.sql.expression.CTE) objects, which are treated similarly to [Alias](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Alias" \o "sqlalchemy.sql.expression.Alias) objects, as special elements to be delivered to the FROM clause of the statement as well as to a WITH clause at the top of the statement.

*Changed in version 1.1:*Added support for UPDATE/INSERT/DELETE as CTE, CTEs added to UPDATE/INSERT/DELETE.

|  |  |
| --- | --- |
| **Parameters:** | * ****name**** – name given to the common table expression. Like \_FromClause.alias(), the name can be left as None in which case an anonymous symbol will be used at query compile time. * ****recursive**** – if True, will render WITH RECURSIVE. A recursive common table expression is intended to be used in conjunction with UNION ALL in order to derive rows from those already selected. |

The following examples include two from PostgreSQL's documentation at <http://www.postgresql.org/docs/current/static/queries-with.html>, as well as additional examples.

Example 1, non recursive:

**from** **sqlalchemy** **import** (Table, Column, String, Integer,

MetaData, select, func)

metadata = MetaData()

orders = Table('orders', metadata,

Column('region', String),

Column('amount', Integer),

Column('product', String),

Column('quantity', Integer))

regional\_sales = select([

orders.c.region,

func.sum(orders.c.amount).label('total\_sales')

]).group\_by(orders.c.region).cte("regional\_sales")

top\_regions = select([regional\_sales.c.region]).\

where(

regional\_sales.c.total\_sales >

select([

func.sum(regional\_sales.c.total\_sales)/10

])

).cte("top\_regions")

statement = select([

orders.c.region,

orders.c.product,

func.sum(orders.c.quantity).label("product\_units"),

func.sum(orders.c.amount).label("product\_sales")

]).where(orders.c.region.in\_(

select([top\_regions.c.region])

)).group\_by(orders.c.region, orders.c.product)

result = conn.execute(statement).fetchall()

Example 2, WITH RECURSIVE:

**from** **sqlalchemy** **import** (Table, Column, String, Integer,

MetaData, select, func)

metadata = MetaData()

parts = Table('parts', metadata,

Column('part', String),

Column('sub\_part', String),

Column('quantity', Integer),)

included\_parts = select([

parts.c.sub\_part,

parts.c.part,

parts.c.quantity]).\

where(parts.c.part=='our part').\

cte(recursive=**True**)

incl\_alias = included\_parts.alias()parts\_alias = parts.alias()included\_parts = included\_parts.union\_all(

select([

parts\_alias.c.sub\_part,

parts\_alias.c.part,

parts\_alias.c.quantity

]).

where(parts\_alias.c.part==incl\_alias.c.sub\_part))

statement = select([

included\_parts.c.sub\_part,

func.sum(included\_parts.c.quantity).

label('total\_quantity')

]).\

group\_by(included\_parts.c.sub\_part)

result = conn.execute(statement).fetchall()

Example 3, an upsert using UPDATE and INSERT with CTEs:

**from** **datetime** **import** date**from** **sqlalchemy** **import** (MetaData, Table, Column, Integer,

Date, select, literal, and\_, exists)

metadata = MetaData()

visitors = Table('visitors', metadata,

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

Column('date', Date, primary\_key=**True**),

Column('count', Integer),)

*# add 5 visitors for the product\_id == 1*product\_id = 1day = date.today()count = 5

update\_cte = (

visitors.update()

.where(and\_(visitors.c.product\_id == product\_id,

visitors.c.date == day))

.values(count=visitors.c.count + count)

.returning(literal(1))

.cte('update\_cte'))

upsert = visitors.insert().from\_select(

[visitors.c.product\_id, visitors.c.date, visitors.c.count],

select([literal(product\_id), literal(day), literal(count)])

.where(~exists(update\_cte.select())))

connection.execute(upsert)

**See also**

[orm.query.Query.cte()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.cte" \o "sqlalchemy.orm.query.Query.cte) - ORM version of [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).

**dialect\_kwargs**

*inherited from the* [dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs) *attribute of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

A collection of keyword arguments specified as dialect-specific options to this construct.

The arguments are present here in their original <dialect>\_<kwarg> format. Only arguments that were actually passed are included; unlike the [DialectKWArgs.dialect\_options](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_options" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_options) collection, which contains all options known by this dialect including defaults.

The collection is also writable; keys are accepted of the form <dialect>\_<kwarg> where the value will be assembled into the list of options.

*New in version 0.9.2.*

*Changed in version 0.9.4:*The [DialectKWArgs.dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs) collection is now writable.

**See also**

[DialectKWArgs.dialect\_options](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_options" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_options) - nested dictionary form

**dialect\_options**

*inherited from the* [dialect\_options](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_options" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_options) *attribute of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

A collection of keyword arguments specified as dialect-specific options to this construct.

This is a two-level nested registry, keyed to <dialect\_name> and <argument\_name>. For example, the postgresql\_where argument would be locatable as:

arg = my\_object.dialect\_options['postgresql']['where']

*New in version 0.9.2.*

**See also**

[DialectKWArgs.dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs) - flat dictionary form

**execute**(*\*multiparams*, *\*\*params*)

*inherited from the* [execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable.execute" \o "sqlalchemy.sql.expression.Executable.execute) *method of* [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable)

Compile and execute this [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable).

**execution\_options**(*\*\*kw*)

*inherited from the* [execution\_options()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable.execution_options" \o "sqlalchemy.sql.expression.Executable.execution_options) *method of* [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable)

Set non-SQL options for the statement which take effect during execution.

Execution options can be set on a per-statement or per [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) basis. Additionally, the [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) and ORM [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) objects provide access to execution options which they in turn configure upon connections.

The [execution\_options()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Update.execution_options" \o "sqlalchemy.sql.expression.Update.execution_options) method is generative. A new instance of this statement is returned that contains the options:

statement = select([table.c.x, table.c.y])statement = statement.execution\_options(autocommit=**True**)

Note that only a subset of possible execution options can be applied to a statement - these include "autocommit" and "stream\_results", but not "isolation\_level" or "compiled\_cache". See [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) for a full list of possible options.

**See also**

[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)

[Query.execution\_options()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.execution_options" \o "sqlalchemy.orm.query.Query.execution_options)

**kwargs**

*inherited from the* [kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.kwargs) *attribute of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

A synonym for [DialectKWArgs.dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs).

**params**(*\*arg*, *\*\*kw*)

*inherited from the* [params()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase.params" \o "sqlalchemy.sql.expression.UpdateBase.params) *method of* [UpdateBase](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase" \o "sqlalchemy.sql.expression.UpdateBase)

Set the parameters for the statement.

This method raises NotImplementedError on the base class, and is overridden by [ValuesBase](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase" \o "sqlalchemy.sql.expression.ValuesBase) to provide the SET/VALUES clause of UPDATE and INSERT.

**prefix\_with**(*\*expr*, *\*\*kw*)

*inherited from the* [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) *method of* [HasPrefixes](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.HasPrefixes" \o "sqlalchemy.sql.expression.HasPrefixes)

Add one or more expressions following the statement keyword, i.e. SELECT, INSERT, UPDATE, or DELETE. Generative.

This is used to support backend-specific prefix keywords such as those provided by MySQL.

E.g.:

stmt = table.insert().prefix\_with("LOW\_PRIORITY", dialect="mysql")

Multiple prefixes can be specified by multiple calls to [prefix\_with()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Update.prefix_with" \o "sqlalchemy.sql.expression.Update.prefix_with).

|  |  |
| --- | --- |
| **Parameters:** | * ****\*expr**** – textual or [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement) construct which will be rendered following the INSERT, UPDATE, or DELETE keyword. * ****\*\*kw**** – A single keyword 'dialect' is accepted. This is an optional string dialect name which will limit rendering of this prefix to only that dialect. |

**return\_defaults**(*\*cols*)

*inherited from the* [return\_defaults()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase.return_defaults" \o "sqlalchemy.sql.expression.ValuesBase.return_defaults) *method of* [ValuesBase](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase" \o "sqlalchemy.sql.expression.ValuesBase)

Make use of a [RETURNING](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-returning) clause for the purpose of fetching server-side expressions and defaults.

E.g.:

stmt = table.insert().values(data='newdata').return\_defaults()

result = connection.execute(stmt)

server\_created\_at = result.returned\_defaults['created\_at']

When used against a backend that supports RETURNING, all column values generated by SQL expression or server-side-default will be added to any existing RETURNING clause, provided that [UpdateBase.returning()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase.returning" \o "sqlalchemy.sql.expression.UpdateBase.returning) is not used simultaneously. The column values will then be available on the result using the [ResultProxy.returned\_defaults](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.returned_defaults" \o "sqlalchemy.engine.ResultProxy.returned_defaults) accessor as a dictionary, referring to values keyed to the [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) object as well as its .key.

当用于支持RETURNING的后端时，由SQL表达式或服务器端默认生成的所有列值都将添加到任何现有的RETURNING子句中，前提是不会同时使用UpdateBase.returning() 。 然后，列值将使用ResultProxy.returned\_defaults访问器作为字典在结果上可用，引用键对Column对象及其.key的值。

This method differs from [UpdateBase.returning()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase.returning" \o "sqlalchemy.sql.expression.UpdateBase.returning) in these ways:

此方法与UpdateBase.returning() 不同之处在于：

1. [ValuesBase.return\_defaults()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase.return_defaults" \o "sqlalchemy.sql.expression.ValuesBase.return_defaults) is only intended for use with an INSERT or an UPDATE statement that matches exactly one row. While the RETURNING construct in the general sense supports multiple rows for a multi-row UPDATE or DELETE statement, or for special cases of INSERT that return multiple rows (e.g. INSERT from SELECT, multi-valued VALUES clause), [ValuesBase.return\_defaults()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase.return_defaults" \o "sqlalchemy.sql.expression.ValuesBase.return_defaults) is intended only for an "ORM-style" single-row INSERT/UPDATE statement. The row returned by the statement is also consumed implicitly when[ValuesBase.return\_defaults()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase.return_defaults" \o "sqlalchemy.sql.expression.ValuesBase.return_defaults) is used. By contrast, [UpdateBase.returning()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase.returning" \o "sqlalchemy.sql.expression.UpdateBase.returning) leaves the RETURNING result-set intact with a collection of any number of rows.
2. It is compatible with the existing logic to fetch auto-generated primary key values, also known as "implicit returning". Backends that support RETURNING will automatically make use of RETURNING in order to fetch the value of newly generated primary keys; while the[UpdateBase.returning()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase.returning" \o "sqlalchemy.sql.expression.UpdateBase.returning) method circumvents this behavior, [ValuesBase.return\_defaults()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase.return_defaults" \o "sqlalchemy.sql.expression.ValuesBase.return_defaults) leaves it intact.
3. It can be called against any backend. Backends that don't support RETURNING will skip the usage of the feature, rather than raising an exception. The return value of [ResultProxy.returned\_defaults](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.returned_defaults" \o "sqlalchemy.engine.ResultProxy.returned_defaults) will be None

[ValuesBase.return\_defaults()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase.return_defaults" \o "sqlalchemy.sql.expression.ValuesBase.return_defaults) is used by the ORM to provide an efficient implementation for the eager\_defaults feature of [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper).

|  |  |
| --- | --- |
| **Parameters:** | ****cols**** – optional list of column key names or [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects. If omitted, all column expressions evaluated on the server are added to the returning list. |

*New in version 0.9.0.*

**See also**

[UpdateBase.returning()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase.returning" \o "sqlalchemy.sql.expression.UpdateBase.returning)

[ResultProxy.returned\_defaults](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.returned_defaults" \o "sqlalchemy.engine.ResultProxy.returned_defaults)

**returning**(*\*cols*)

*inherited from the* [returning()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase.returning" \o "sqlalchemy.sql.expression.UpdateBase.returning) *method of* [UpdateBase](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase" \o "sqlalchemy.sql.expression.UpdateBase)

Add a [RETURNING](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-returning) or equivalent clause to this statement.

e.g.:

stmt = table.update().\

where(table.c.data == 'value').\

values(status='X').\

returning(table.c.server\_flag,

table.c.updated\_timestamp)

**for** server\_flag, updated\_timestamp **in** connection.execute(stmt):

print(server\_flag, updated\_timestamp)

The given collection of column expressions should be derived from the table that is the target of the INSERT, UPDATE, or DELETE. While [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects are typical, the elements can also be expressions:

stmt = table.insert().returning(

(table.c.first\_name + " " + table.c.last\_name).

label('fullname'))

Upon compilation, a RETURNING clause, or database equivalent, will be rendered within the statement. For INSERT and UPDATE, the values are the newly inserted/updated values. For DELETE, the values are those of the rows which were deleted.

Upon execution, the values of the columns to be returned are made available via the result set and can be iterated using [ResultProxy.fetchone()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.fetchone" \o "sqlalchemy.engine.ResultProxy.fetchone)and similar. For DBAPIs which do not natively support returning values (i.e. cx\_oracle), SQLAlchemy will approximate this behavior at the result level so that a reasonable amount of behavioral neutrality is provided.

Note that not all databases/DBAPIs support RETURNING. For those backends with no support, an exception is raised upon compilation and/or execution. For those who do support it, the functionality across backends varies greatly, including restrictions on executemany() and other statements which return multiple rows. Please read the documentation notes for the database in use in order to determine the availability of RETURNING.

**See also**

[ValuesBase.return\_defaults()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase.return_defaults" \o "sqlalchemy.sql.expression.ValuesBase.return_defaults) - an alternative method tailored towards efficient fetching of server-side defaults and triggers for single-row INSERTs or UPDATEs.

**scalar**(*\*multiparams*, *\*\*params*)

*inherited from the* [scalar()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable.scalar" \o "sqlalchemy.sql.expression.Executable.scalar) *method of* [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable)

Compile and execute this [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable), returning the result's scalar representation.

**self\_group**(*against=None*)

*inherited from the* [self\_group()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.self_group" \o "sqlalchemy.sql.expression.ClauseElement.self_group) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Apply a 'grouping' to this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement).

对这个ClauseElement应用一个“分组”。

This method is overridden by subclasses to return a "grouping" construct, i.e. parenthesis. In particular it's used by "binary" expressions to provide a grouping around themselves when placed into a larger expression, as well as by [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) constructs when placed into the FROM clause of another [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select). (Note that subqueries should be normally created using the [Select.alias()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.alias" \o "sqlalchemy.sql.expression.Select.alias) method, as many platforms require nested SELECT statements to be named).

该方法被子类覆盖以返回“分组”结构，即括号。 特别是“二进制”表达式在被放置到一个更大的表达式时提供了一个分组，当被放置到另一个select（）的FROM子句中时，通过select（）构造。 （请注意，通常应使用Select.alias（）方法创建子查询，因为许多平台需要命名嵌套的SELECT语句）。

As expressions are composed together, the application of [self\_group()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Update.self_group" \o "sqlalchemy.sql.expression.Update.self_group) is automatic - end-user code should never need to use this method directly. Note that SQLAlchemy's clause constructs take operator precedence into account - so parenthesis might not be needed, for example, in an expression like x OR (y AND z) - AND takes precedence over OR.

由于表达式组合在一起，[self\_group()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Update.self_group" \o "sqlalchemy.sql.expression.Update.self_group)的应用程序是自动的 - 最终用户代码不需要直接使用此方法。 请注意，SQLAlchemy的子句构造将运算符优先级考虑在内 - 因此可能不需要括号，例如在x OR (y AND z)等表达式中，AND优先于OR。

The base [self\_group()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Update.self_group" \o "sqlalchemy.sql.expression.Update.self_group) method of [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement) just returns self.

ClauseElement的[self\_group()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Update.self_group" \o "sqlalchemy.sql.expression.Update.self_group)方法只返回self。

**unique\_params**(*\*optionaldict*, *\*\*kwargs*)

*inherited from the* [unique\_params()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.unique_params" \o "sqlalchemy.sql.expression.ClauseElement.unique_params) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Return a copy with [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) elements replaced.

Same functionality as params(), except adds unique=True to affected bind parameters so that multiple statements can be used.

**values**(*\*args*, *\*\*kwargs*)

*inherited from the* [values()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase.values" \o "sqlalchemy.sql.expression.ValuesBase.values) *method of* [ValuesBase](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase" \o "sqlalchemy.sql.expression.ValuesBase)

specify a fixed VALUES clause for an INSERT statement, or the SET clause for an UPDATE.

Note that the [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 support per-execution time formatting of the VALUES and/or SET clauses, based on the arguments passed to [Connection.execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.execute" \o "sqlalchemy.engine.Connection.execute). However, the [ValuesBase.values()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase.values" \o "sqlalchemy.sql.expression.ValuesBase.values) method can be used to "fix" a particular set of parameters into the statement.

Multiple calls to [ValuesBase.values()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase.values" \o "sqlalchemy.sql.expression.ValuesBase.values) will produce a new construct, each one with the parameter list modified to include the new parameters sent. In the typical case of a single dictionary of parameters, the newly passed keys will replace the same keys in the previous construct. In the case of a list-based "multiple values" construct, each new list of values is extended onto the existing list of values.

|  |  |
| --- | --- |
| **Parameters:** | * ****\*\*kwargs –****key value pairs representing the string key of a [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) mapped to the value to be rendered into the VALUES or SET clause:   users.insert().values(name="some name")  users.update().where(users.c.id==5).values(name="some name")   * ****\*args**** –   As an alternative to passing key/value parameters, a dictionary, tuple, or list of dictionaries or tuples can be passed as a single positional argument in order to form the VALUES or SET clause of the statement. The forms that are accepted vary based on whether this is an [Insert](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert" \o "sqlalchemy.sql.expression.Insert) or an [Update](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Update" \o "sqlalchemy.sql.expression.Update)construct.  For either an [Insert](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert" \o "sqlalchemy.sql.expression.Insert) or [Update](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Update" \o "sqlalchemy.sql.expression.Update) construct, a single dictionary can be passed, which works the same as that of the kwargs form:  users.insert().values({"name": "some name"})  users.update().values({"name": "some new name"})  Also for either form but more typically for the [Insert](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert" \o "sqlalchemy.sql.expression.Insert) construct, a tuple that contains an entry for every column in the table is also accepted:  users.insert().values((5, "some name"))  The [Insert](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert" \o "sqlalchemy.sql.expression.Insert) construct also supports being passed a list of dictionaries or full-table-tuples, which on the server will render the less common SQL syntax of "multiple values" - this syntax is supported on backends such as SQLite, PostgreSQL, MySQL, but not necessarily others:  users.insert().values([  {"name": "some name"},  {"name": "some other name"},  {"name": "yet another name"},  ])  The above form would render a multiple VALUES statement similar to:  INSERT INTO users (name) VALUES  (:name\_1),  (:name\_2),  (:name\_3)  It is essential to note that ****passing multiple values is NOT the same as using traditional executemany() form****. The above syntax is a ****special**** syntax not typically used. To emit an INSERT statement against multiple rows, the normal method is to pass a multiple values list to the [Connection.execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.execute" \o "sqlalchemy.engine.Connection.execute) method, which is supported by all database backends and is generally more efficient for a very large number of parameters.  **See also**  [Executing Multiple Statements](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html" \l "execute-multiple) - an introduction to the traditional Core method of multiple parameter set invocation for INSERTs and other statements.  *Changed in version 1.0.0:*an INSERT that uses a multiple-VALUES clause, even a list of length one, implies that the [Insert.inline](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert.params.inline" \o "sqlalchemy.sql.expression.Insert) flag is set to True, indicating that the statement will not attempt to fetch the "last inserted primary key" or other defaults. The statement deals with an arbitrary number of rows, so the [ResultProxy.inserted\_primary\_key](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.inserted_primary_key" \o "sqlalchemy.engine.ResultProxy.inserted_primary_key) accessor does not apply.  *Changed in version 1.0.0:*A multiple-VALUES INSERT now supports columns with Python side default values and callables in the same way as that of an "executemany" style of invocation; the callable is invoked for each row. See [Python-side defaults invoked for each row invidually when using a multivalued insert](http://docs.sqlalchemy.org/en/rel_1_1/changelog/migration_10.html" \l "bug-3288) for other details.  The [Update](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Update" \o "sqlalchemy.sql.expression.Update) construct supports a special form which is a list of 2-tuples, which when provided must be passed in conjunction with 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) parameter. This form causes the UPDATE statement to render the SET clauses using the order of parameters given to [Update.values()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Update.values" \o "sqlalchemy.sql.expression.Update.values), rather than the ordering of columns given in the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table).  *New in version 1.0.10:*- added support for parameter-ordered UPDATE statements via 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.  **See also**  [Parameter-Ordered Updates](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html" \l "updates-order-parameters) - full example of 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 |

**See also**

[Inserts, Updates and Deletes](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html" \l "inserts-and-updates) - SQL Expression Language Tutorial

[insert()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.insert" \o "sqlalchemy.sql.expression.insert) - produce an INSERT statement

[update()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.update" \o "sqlalchemy.sql.expression.update) - produce an UPDATE statement

**where**(*whereclause*)

return a new update() construct with the given expression added to its WHERE clause, joined to the existing clause via AND, if any.

**with\_hint**(*text*, *selectable=None*, *dialect\_name='\*'*)

*inherited from the* [with\_hint()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase.with_hint" \o "sqlalchemy.sql.expression.UpdateBase.with_hint) *method of* [UpdateBase](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase" \o "sqlalchemy.sql.expression.UpdateBase)

Add a table hint for a single table to this INSERT/UPDATE/DELETE statement.

**Note**

[UpdateBase.with\_hint()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase.with_hint" \o "sqlalchemy.sql.expression.UpdateBase.with_hint) currently applies only to Microsoft SQL Server. For MySQL INSERT/UPDATE/DELETE hints, use [UpdateBase.prefix\_with()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase.prefix_with" \o "sqlalchemy.sql.expression.UpdateBase.prefix_with).

The text of the hint is rendered in the appropriate location for the database backend in use, relative to the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) that is the subject of this statement, or optionally to that of the given [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) passed as the selectable argument.

The dialect\_name option will limit the rendering of a particular hint to a particular backend. Such as, to add a hint that only takes effect for SQL Server:

mytable.insert().with\_hint("WITH (PAGLOCK)", dialect\_name="mssql")

*New in version 0.7.6.*

|  |  |
| --- | --- |
| **Parameters:** | * ****text**** – Text of the hint. * ****selectable**** – optional [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) that specifies an element of the FROM clause within an UPDATE or DELETE to be the subject of the hint - applies only to certain backends. * ****dialect\_name**** – defaults to \*, if specified as the name of a particular dialect, will apply these hints only when that dialect is in use. |

*class*sqlalchemy.sql.expression.**UpdateBase**

Bases: [sqlalchemy.sql.expression.HasCTE](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.HasCTE" \o "sqlalchemy.sql.expression.HasCTE), [sqlalchemy.sql.base.DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs), [sqlalchemy.sql.expression.HasPrefixes](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.HasPrefixes" \o "sqlalchemy.sql.expression.HasPrefixes), [sqlalchemy.sql.expression.Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable), [sqlalchemy.sql.expression.ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Form the base for INSERT, UPDATE, and DELETE statements.

**\_\_init\_\_**

*inherited from the* \_\_init\_\_ *attribute of* object

x.\_\_init\_\_(…) initializes x; see help(type(x)) for signature

**argument\_for**(*dialect\_name*, *argument\_name*, *default*)

*inherited from the* [argument\_for()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.argument_for" \o "sqlalchemy.sql.base.DialectKWArgs.argument_for) *method of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

Add a new kind of dialect-specific keyword argument for this class.

E.g.:

Index.argument\_for("mydialect", "length", **None**)

some\_index = Index('a', 'b', mydialect\_length=5)

The [DialectKWArgs.argument\_for()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.argument_for" \o "sqlalchemy.sql.base.DialectKWArgs.argument_for) method is a per-argument way adding extra arguments to the[DefaultDialect.construct\_arguments](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.default.DefaultDialect.construct_arguments" \o "sqlalchemy.engine.default.DefaultDialect.construct_arguments) dictionary. This dictionary provides a list of argument names accepted by various schema-level constructs on behalf of a dialect.

New dialects should typically specify this dictionary all at once as a data member of the dialect class. The use case for ad-hoc addition of argument names is typically for end-user code that is also using a custom compilation scheme which consumes the additional arguments.

|  |  |
| --- | --- |
| **Parameters:** | * ****dialect\_name**** – name of a dialect. The dialect must be locatable, else a [NoSuchModuleError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.NoSuchModuleError" \o "sqlalchemy.exc.NoSuchModuleError) is raised. The dialect must also include an existing[DefaultDialect.construct\_arguments](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.default.DefaultDialect.construct_arguments" \o "sqlalchemy.engine.default.DefaultDialect.construct_arguments) collection, indicating that it participates in the keyword-argument validation and default system, else [ArgumentError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.ArgumentError" \o "sqlalchemy.exc.ArgumentError) is raised. If the dialect does not include this collection, then any keyword argument can be specified on behalf of this dialect already. All dialects packaged within SQLAlchemy include this collection, however for third party dialects, support may vary. * ****argument\_name**** – name of the parameter. * ****default**** – default value of the parameter. |

*New in version 0.9.4.*

**bind**

Return a 'bind' linked to this [UpdateBase](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase" \o "sqlalchemy.sql.expression.UpdateBase) or a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) associated with it.

**compare**(*other*, *\*\*kw*)

*inherited from the* [compare()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.compare" \o "sqlalchemy.sql.expression.ClauseElement.compare) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Compare this ClauseElement to the given ClauseElement.

Subclasses should override the default behavior, which is a straight identity comparison.

\*\*kw are arguments consumed by subclass compare() methods and may be used to modify the criteria for comparison. (see [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement))

**compile**(*bind=None*, *dialect=None*, *\*\*kw*)

*inherited from the* [compile()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.compile" \o "sqlalchemy.sql.expression.ClauseElement.compile) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Compile this SQL expression.

编译此SQL表达式。

The return value is a [Compiled](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Compiled" \o "sqlalchemy.engine.interfaces.Compiled) object. Calling str() or unicode() on the returned value will yield a string representation of the result. The[Compiled](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Compiled" \o "sqlalchemy.engine.interfaces.Compiled) object also can return a dictionary of bind parameter names and values using the params accessor.

返回值是一个编译对象。 对返回的值调用str() 或unicode() 将产生结果的字符串表示形式。 Compiled对象还可以使用params访问器返回绑定参数名称和值的字典。

|  |  |
| --- | --- |
| **Parameters:** | * ****bind**** – An Engine or Connection from which a Compiled will be acquired. This argument takes precedence over this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)'s bound engine, if any. * ****column\_keys**** – Used for INSERT and UPDATE statements, a list of column names which should be present in the VALUES clause of the compiled statement. If None, all columns from the target table object are rendered.用于INSERT和UPDATE语句，应在编译语句的VALUES子句中存在列名称列表。 如果为None，则会渲染目标表对象中的所有列。 * ****dialect**** – A Dialect instance from which a Compiled will be acquired. This argument takes precedence over the bind argument as well as this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)'s bound engine, if any.将从其中获取编译的方言实例。 此参数优先于bind参数以及此ClauseElement的绑定引擎（如果有）。 * ****inline**** – Used for INSERT statements, for a dialect which does not support inline retrieval of newly generated primary key columns, will force the expression used to create the new primary key value to be rendered inline within the INSERT statement's VALUES clause. This typically refers to Sequence execution but may also refer to any server-side default generation function associated with a primary key Column.用于INSERT语句，对于不支持新生成的主键列的内联检索的方言，将强制用于创建新主键值的表达式在INSERT语句的VALUES子句中内联呈现。 这通常是指Sequence执行，但也可以指与主键Column相关联的任何服务器端默认生成函数。 * ****compile\_kwargs –****optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the literal\_binds flag through:在所有“访问”方法中将附加参数的可选字典传递给编译器。 例如，这允许将任何自定义标志传递到自定义编译构造。 它也用于通过literal\_binds标志的情况：   **from** **sqlalchemy.sql** **import** table, column, select  t = table('t', column('x'))  s = select([t]).where(t.c.x == 5)  print s.compile(compile\_kwargs={"literal\_binds": **True**})  *New in version 0.9.0.* |

**See also**

[How do I render SQL expressions as strings, possibly with bound parameters inlined?](http://docs.sqlalchemy.org/en/rel_1_1/faq/sqlexpressions.html" \l "faq-sql-expression-string)

**cte**(*name=None*, *recursive=False*)

*inherited from the* [cte()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.HasCTE.cte" \o "sqlalchemy.sql.expression.HasCTE.cte) *method of* [HasCTE](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.HasCTE" \o "sqlalchemy.sql.expression.HasCTE)

Return a new [CTE](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.CTE" \o "sqlalchemy.sql.expression.CTE), or Common Table Expression instance.

返回一个新的CTE或Common Table Expression实例。

Common table expressions are a SQL standard whereby SELECT statements can draw upon secondary statements specified along with the primary statement, using a clause called "WITH". Special semantics regarding UNION can also be employed to allow "recursive" queries, where a SELECT statement can draw upon the set of rows that have previously been selected.

公用表表达式是一种SQL标准，其中SELECT语句可以使用一个称为“WITH”的子句来使用主语句指定的辅助语句。 还可以使用有关UNION的特殊语义来允许“递归”查询，其中SELECT语句可以对先前已选择的行集合进行绘制。

CTEs can also be applied to DML constructs UPDATE, INSERT and DELETE on some databases, both as a source of CTE rows when combined with RETURNING, as well as a consumer of CTE rows.

CTE还可以应用于某些数据库上的DML构造UPDATE，INSERT和DELETE，作为与RETURNING组合的CTE行的源，以及CTE行的使用者。

SQLAlchemy detects [CTE](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.CTE" \o "sqlalchemy.sql.expression.CTE) objects, which are treated similarly to [Alias](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Alias" \o "sqlalchemy.sql.expression.Alias) objects, as special elements to be delivered to the FROM clause of the statement as well as to a WITH clause at the top of the statement.

SQLAlchemy检测到与Alias对象类似的CTE对象作为要传递到语句的FROM子句的特殊元素以及语句顶部的WITH子句。

*Changed in version 1.1:*Added support for UPDATE/INSERT/DELETE as CTE, CTEs added to UPDATE/INSERT/DELETE.

|  |  |
| --- | --- |
| **Parameters:** | * ****name**** – name given to the common table expression. Like \_FromClause.alias(), the name can be left as None in which case an anonymous symbol will be used at query compile time. * ****recursive**** – if True, will render WITH RECURSIVE. A recursive common table expression is intended to be used in conjunction with UNION ALL in order to derive rows from those already selected. |

The following examples include two from PostgreSQL's documentation at <http://www.postgresql.org/docs/current/static/queries-with.html>, as well as additional examples.

以下示例包括PostgreSQL在http://www.postgresql.org/docs/current/static/queries-with.html上的两个文档，以及其他示例。

Example 1, non recursive:

**from** **sqlalchemy** **import** (Table, Column, String, Integer,

MetaData, select, func)

metadata = MetaData()

orders = Table('orders', metadata,

Column('region', String),

Column('amount', Integer),

Column('product', String),

Column('quantity', Integer))

regional\_sales = select([

orders.c.region,

func.sum(orders.c.amount).label('total\_sales')

]).group\_by(orders.c.region).cte("regional\_sales")

top\_regions = select([regional\_sales.c.region]).\

where(

regional\_sales.c.total\_sales >

select([

func.sum(regional\_sales.c.total\_sales)/10

])

).cte("top\_regions")

statement = select([

orders.c.region,

orders.c.product,

func.sum(orders.c.quantity).label("product\_units"),

func.sum(orders.c.amount).label("product\_sales")

]).where(orders.c.region.in\_(

select([top\_regions.c.region])

)).group\_by(orders.c.region, orders.c.product)

result = conn.execute(statement).fetchall()

Example 2, WITH RECURSIVE:

**from** **sqlalchemy** **import** (Table, Column, String, Integer,

MetaData, select, func)

metadata = MetaData()

parts = Table('parts', metadata,

Column('part', String),

Column('sub\_part', String),

Column('quantity', Integer),)

included\_parts = select([

parts.c.sub\_part,

parts.c.part,

parts.c.quantity]).\

where(parts.c.part=='our part').\

cte(recursive=**True**)

incl\_alias = included\_parts.alias()parts\_alias = parts.alias()included\_parts = included\_parts.union\_all(

select([

parts\_alias.c.sub\_part,

parts\_alias.c.part,

parts\_alias.c.quantity

]).

where(parts\_alias.c.part==incl\_alias.c.sub\_part))

statement = select([

included\_parts.c.sub\_part,

func.sum(included\_parts.c.quantity).

label('total\_quantity')

]).\

group\_by(included\_parts.c.sub\_part)

result = conn.execute(statement).fetchall()

Example 3, an upsert using UPDATE and INSERT with CTEs:

**from** **datetime** **import** date

**from** **sqlalchemy** **import** (MetaData, Table, Column, Integer,

Date, select, literal, and\_, exists)

metadata = MetaData()

visitors = Table('visitors', metadata,

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

Column('date', Date, primary\_key=**True**),

Column('count', Integer),)

*# add 5 visitors for the product\_id == 1*product\_id = 1day = date.today()count = 5

update\_cte = (

visitors.update()

.where(and\_(visitors.c.product\_id == product\_id,

visitors.c.date == day))

.values(count=visitors.c.count + count)

.returning(literal(1))

.cte('update\_cte'))

upsert = visitors.insert().from\_select(

[visitors.c.product\_id, visitors.c.date, visitors.c.count],

select([literal(product\_id), literal(day), literal(count)])

.where(~exists(update\_cte.select())))

connection.execute(upsert)

**See also**

[orm.query.Query.cte()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.cte" \o "sqlalchemy.orm.query.Query.cte) - ORM version of [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).

**dialect\_kwargs**

*inherited from the* [dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs) *attribute of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

A collection of keyword arguments specified as dialect-specific options to this construct.

指定为此构造的方言特定选项的关键字参数的集合。

The arguments are present here in their original <dialect>\_<kwarg> format. Only arguments that were actually passed are included; unlike the [DialectKWArgs.dialect\_options](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_options" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_options) collection, which contains all options known by this dialect including defaults.

参数以原始<dialect> \_ <kwarg>格式存在。 只包括实际通过的论据; 不同于DialectKWArgs.dialect\_options集合，其中包含此方言已知的所有选项，包括默认值。

The collection is also writable; keys are accepted of the form <dialect>\_<kwarg> where the value will be assembled into the list of options.

收藏也是可写的; 密钥被接受为<dialect> \_ <kwarg>的形式，其中值将被汇总到选项列表中。

*New in version 0.9.2.*

*Changed in version 0.9.4:*The [DialectKWArgs.dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs) collection is now writable.

**See also**

[DialectKWArgs.dialect\_options](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_options" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_options) - nested dictionary form

**dialect\_options**

*inherited from the* [dialect\_options](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_options" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_options) *attribute of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

A collection of keyword arguments specified as dialect-specific options to this construct.

指定为此构造的方言特定选项的关键字参数的集合。

This is a two-level nested registry, keyed to <dialect\_name> and <argument\_name>. For example, the postgresql\_where argument would be locatable as:

arg = my\_object.dialect\_options['postgresql']['where']

*New in version 0.9.2.*

**See also**

[DialectKWArgs.dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs) - flat dictionary form

**execute**(*\*multiparams*, *\*\*params*)

*inherited from the* [execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable.execute" \o "sqlalchemy.sql.expression.Executable.execute) *method of* [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable)

Compile and execute this [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable).

**execution\_options**(*\*\*kw*)

*inherited from the* [execution\_options()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable.execution_options" \o "sqlalchemy.sql.expression.Executable.execution_options) *method of* [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable)

Set non-SQL options for the statement which take effect during execution.

为执行期间生效的语句设置非SQL选项。

Execution options can be set on a per-statement or per [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) basis. Additionally, the [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) and ORM [Query](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query" \o "sqlalchemy.orm.query.Query) objects provide access to execution options which they in turn configure upon connections.

可以在每个语句或每个连接的基础上设置执行选项。 此外，引擎和ORM查询对象提供对执行选项的访问，它们在连接时依次配置。

The [execution\_options()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase.execution_options" \o "sqlalchemy.sql.expression.UpdateBase.execution_options) method is generative. A new instance of this statement is returned that contains the options:

execution\_options() 方法是生成的。 此语句的新实例将返回，其中包含以下选项：

statement = select([table.c.x, table.c.y])statement = statement.execution\_options(autocommit=**True**)

Note that only a subset of possible execution options can be applied to a statement - these include "autocommit" and "stream\_results", but not "isolation\_level" or "compiled\_cache". See [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) for a full list of possible options.

请注意，只有一个可能的执行选项的子集可以应用于一个语句 - 这些包括“autocommit”和“stream\_results”，而不是“isolation\_level”或“compiled\_cache”。 有关可能的选项的完整列表，请参阅Connection.execution\_options() 。

**See also**

[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)

[Query.execution\_options()](http://docs.sqlalchemy.org/en/rel_1_1/orm/query.html" \l "sqlalchemy.orm.query.Query.execution_options" \o "sqlalchemy.orm.query.Query.execution_options)

**get\_children**(*\*\*kwargs*)

*inherited from the* [get\_children()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.get_children" \o "sqlalchemy.sql.expression.ClauseElement.get_children) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Return immediate child elements of this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement).

返回此ClauseElement的即时子元素。

This is used for visit traversal.

这用于访问遍历。

\*\*kwargs may contain flags that change the collection that is returned, for example to return a subset of items in order to cut down on larger traversals, or to return child items from a different context (such as schema-level collections instead of clause-level).

\*\* kwargs可能包含改变所返回的集合的标志，例如返回一个子集，以减少较大的遍历，或从不同的上下文返回子项（如模式级集合而不是子句） -水平）。

**kwargs**

*inherited from the* [kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.kwargs) *attribute of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

A synonym for [DialectKWArgs.dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs).

**params**(*\*arg*, *\*\*kw*)

Set the parameters for the statement.

设置语句的参数。

This method raises NotImplementedError on the base class, and is overridden by [ValuesBase](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase" \o "sqlalchemy.sql.expression.ValuesBase) to provide the SET/VALUES clause of UPDATE and INSERT.

此方法在基类上引发NotImplementedError，并由ValuesBase覆盖以提供UPDATE和INSERT的SET / VALUES子句。

**prefix\_with**(*\*expr*, *\*\*kw*)

*inherited from the* [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) *method of* [HasPrefixes](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.HasPrefixes" \o "sqlalchemy.sql.expression.HasPrefixes)

Add one or more expressions following the statement keyword, i.e. SELECT, INSERT, UPDATE, or DELETE. Generative.

在语句关键字之后添加一个或多个表达式，即SELECT，INSERT，UPDATE或DELETE。生成。

This is used to support backend-specific prefix keywords such as those provided by MySQL.

这用于支持后端特定的前缀关键字，如MySQL提供的那些。

E.g.:

stmt = table.insert().prefix\_with("LOW\_PRIORITY", dialect="mysql")

Multiple prefixes can be specified by multiple calls to [prefix\_with()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase.prefix_with" \o "sqlalchemy.sql.expression.UpdateBase.prefix_with).

可以通过对prefix\_with() 的多次调用指定多个前缀。

|  |  |
| --- | --- |
| **Parameters:** | * ****\*expr**** – textual or [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement) construct which will be rendered following the INSERT, UPDATE, or DELETE keyword.文本或ClauseElement构造，将在INSERT，UPDATE或DELETE关键字之后呈现。 * ****\*\*kw**** – A single keyword 'dialect' is accepted. This is an optional string dialect name which will limit rendering of this prefix to only that dialect.单一关键字“方言”被接受。 这是一个可选的字符串方言名称，它将限制将该前缀的渲染仅限于该方言。 |

**returning**(*\*cols*)

Add a [RETURNING](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-returning) or equivalent clause to this statement.

e.g.:

stmt = table.update().\

where(table.c.data == 'value').\

values(status='X').\

returning(table.c.server\_flag,

table.c.updated\_timestamp)

**for** server\_flag, updated\_timestamp **in** connection.execute(stmt):

print(server\_flag, updated\_timestamp)

The given collection of column expressions should be derived from the table that is the target of the INSERT, UPDATE, or DELETE. While [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects are typical, the elements can also be expressions:

列表达式的给定集合应该从作为INSERT，UPDATE或DELETE的目标的表导出。 虽然Column对象是典型的，但元素也可以是表达式：

stmt = table.insert().returning(

(table.c.first\_name + " " + table.c.last\_name).

label('fullname'))

Upon compilation, a RETURNING clause, or database equivalent, will be rendered within the statement. For INSERT and UPDATE, the values are the newly inserted/updated values. For DELETE, the values are those of the rows which were deleted.

编译后，RETURNING子句或数据库等价物将在语句中呈现。对于INSERT和UPDATE，值是新插入/更新的值。对于DELETE，值是被删除的行的值。

Upon execution, the values of the columns to be returned are made available via the result set and can be iterated using [ResultProxy.fetchone()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.fetchone" \o "sqlalchemy.engine.ResultProxy.fetchone)and similar. For DBAPIs which do not natively support returning values (i.e. cx\_oracle), SQLAlchemy will approximate this behavior at the result level so that a reasonable amount of behavioral neutrality is provided.

执行后，要返回的列的值通过结果集可用，并且可以使用ResultProxy.fetchone() 等进行迭代。对于不本地支持返回值（即cx\_oracle）的DBAPI，SQLAlchemy将在结果级别近似此行为，从而提供合理数量的行为中立性。

Note that not all databases/DBAPIs support RETURNING. For those backends with no support, an exception is raised upon compilation and/or execution. For those who do support it, the functionality across backends varies greatly, including restrictions on executemany() and other statements which return multiple rows. Please read the documentation notes for the database in use in order to determine the availability of RETURNING.

请注意，并非所有数据库/ DBAPI都支持RETURNING。对于不支持的后端，在编译和/或执行时会出现异常。对于那些支持它的人，后端的功能差异很大，包括对executemany() 和其他返回多行的语句的限制。请阅读正在使用的数据库的文档说明，以确定RETURNING的可用性。

**See also**

[ValuesBase.return\_defaults()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase.return_defaults" \o "sqlalchemy.sql.expression.ValuesBase.return_defaults) - an alternative method tailored towards efficient fetching of server-side defaults and triggers for single-row INSERTs or UPDATEs.

**scalar**(*\*multiparams*, *\*\*params*)

*inherited from the* [scalar()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable.scalar" \o "sqlalchemy.sql.expression.Executable.scalar) *method of* [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable)

Compile and execute this [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable), returning the result's scalar representation.

**self\_group**(*against=None*)

*inherited from the* [self\_group()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.self_group" \o "sqlalchemy.sql.expression.ClauseElement.self_group) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Apply a 'grouping' to this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement).

对此ClauseElement应用“分组”。

This method is overridden by subclasses to return a "grouping" construct, i.e. parenthesis. In particular it's used by "binary" expressions to provide a grouping around themselves when placed into a larger expression, as well as by [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) constructs when placed into the FROM clause of another [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select). (Note that subqueries should be normally created using the [Select.alias()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.alias" \o "sqlalchemy.sql.expression.Select.alias) method, as many platforms require nested SELECT statements to be named).

这个方法被子类覆盖，以返回一个“分组”结构，即括号。 特别地，它被“二进制”表达式用于在放入更大的表达式时以及通过将select() 结构放入另一个select() 的FROM子句时围绕它们进行分组。 （请注意，通常使用Select.alias() 方法创建子查询，因为许多平台需要嵌套的SELECT语句来命名）。

As expressions are composed together, the application of [self\_group()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase.self_group" \o "sqlalchemy.sql.expression.UpdateBase.self_group) is automatic - end-user code should never need to use this method directly. Note that SQLAlchemy's clause constructs take operator precedence into account - so parenthesis might not be needed, for example, in an expression like x OR (y AND z) - AND takes precedence over OR.

由于表达式组合在一起，self\_group() 的应用是自动的 - 最终用户代码不应该直接使用这种方法。 请注意，SQLAlchemy的子句构造考虑到运算符优先级，因此可能不需要括号，例如，在像x OR（y AND z）之类的表达式中，AND优先于OR。

The base [self\_group()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase.self_group" \o "sqlalchemy.sql.expression.UpdateBase.self_group) method of [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement) just returns self.

ClauseElement的基本self\_group() 方法只返回自身。

**unique\_params**(*\*optionaldict*, *\*\*kwargs*)

*inherited from the* [unique\_params()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.unique_params" \o "sqlalchemy.sql.expression.ClauseElement.unique_params) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Return a copy with [bindparam()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.bindparam" \o "sqlalchemy.sql.expression.bindparam) elements replaced.

用bindparam（）元素替换返回一个副本。

Same functionality as params(), except adds unique=True to affected bind parameters so that multiple statements can be used.

除了为受影响的绑定参数添加unique = True之外，其他功能与params（）相同，因此可以使用多个语句。

**with\_hint**(*text*, *selectable=None*, *dialect\_name='\*'*)

Add a table hint for a single table to this INSERT/UPDATE/DELETE statement.

为这个INSERT / UPDATE / DELETE语句添加一个表提示。

**Note**注意

[UpdateBase.with\_hint()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase.with_hint" \o "sqlalchemy.sql.expression.UpdateBase.with_hint) currently applies only to Microsoft SQL Server. For MySQL INSERT/UPDATE/DELETE hints, use [UpdateBase.prefix\_with()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase.prefix_with" \o "sqlalchemy.sql.expression.UpdateBase.prefix_with).

UpdateBase.with\_hint（）当前仅适用于Microsoft SQL Server。 对于MySQL INSERT / UPDATE / DELETE提示，请使用UpdateBase.prefix\_with（）。

The text of the hint is rendered in the appropriate location for the database backend in use, relative to the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) that is the subject of this statement, or optionally to that of the given [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) passed as the selectable argument.

相对于作为此语句主题的表，相对于作为可选参数传递的给定表格，提示的文本呈现在正在使用的数据库后端的适当位置。

The dialect\_name option will limit the rendering of a particular hint to a particular backend. Such as, to add a hint that only takes effect for SQL Server:

dialect\_name选项将限制特定提示到特定后端的呈现。 比如，添加一个只对SQL Server有效的提示：

mytable.insert().with\_hint("WITH (PAGLOCK)", dialect\_name="mssql")

*New in version 0.7.6.*

|  |  |
| --- | --- |
| **Parameters:** | * ****text**** – Text of the hint.提示的文本。 * ****selectable**** – optional [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) that specifies an element of the FROM clause within an UPDATE or DELETE to be the subject of the hint - applies only to certain backends.指定UPDATE或DELETE中的FROM子句的元素作为提示的主题的可选表格 - 仅适用于特定的后端。 * ****dialect\_name**** – defaults to \*, if specified as the name of a particular dialect, will apply these hints only when that dialect is in use.如果指定为特定方言的名称，则默认为\*，只有当该方言正在使用时才会应用这些提示。 |

*class*sqlalchemy.sql.expression.**ValuesBase**(*table*, *values*, *prefixes*)

Bases: [sqlalchemy.sql.expression.UpdateBase](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase" \o "sqlalchemy.sql.expression.UpdateBase)

Supplies support for [ValuesBase.values()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase.values" \o "sqlalchemy.sql.expression.ValuesBase.values) to INSERT and UPDATE constructs.

提供对ValuesBase.values（）到INSERT和UPDATE结构的支持。

**return\_defaults**(*\*cols*)

Make use of a [RETURNING](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-returning) clause for the purpose of fetching server-side expressions and defaults.

使用RETURNING子句来获取服务器端表达式和默认值。

E.g.:

stmt = table.insert().values(data='newdata').return\_defaults()

result = connection.execute(stmt)

server\_created\_at = result.returned\_defaults['created\_at']

When used against a backend that supports RETURNING, all column values generated by SQL expression or server-side-default will be added to any existing RETURNING clause, provided that [UpdateBase.returning()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase.returning" \o "sqlalchemy.sql.expression.UpdateBase.returning) is not used simultaneously. The column values will then be available on the result using the [ResultProxy.returned\_defaults](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.returned_defaults" \o "sqlalchemy.engine.ResultProxy.returned_defaults) accessor as a dictionary, referring to values keyed to the [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) object as well as its .key.

当用于支持RETURNING的后端时，由SQL表达式或server-side-default生成的所有列值将被添加到任何现有的RETURNING子句中，前提是不同时使用UpdateBase.returning（）。 然后，使用ResultProxy.returned\_defaults访问器作为字典，可以在结果上使用列值，引用键列到Column对象的值以及.key。

This method differs from [UpdateBase.returning()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase.returning" \o "sqlalchemy.sql.expression.UpdateBase.returning) in these ways:

这个方法与UpdateBase.returning（）在这些方面有所不同：

1. [ValuesBase.return\_defaults()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase.return_defaults" \o "sqlalchemy.sql.expression.ValuesBase.return_defaults) is only intended for use with an INSERT or an UPDATE statement that matches exactly one row. While the RETURNING construct in the general sense supports multiple rows for a multi-row UPDATE or DELETE statement, or for special cases of INSERT that return multiple rows (e.g. INSERT from SELECT, multi-valued VALUES clause), [ValuesBase.return\_defaults()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase.return_defaults" \o "sqlalchemy.sql.expression.ValuesBase.return_defaults) is intended only for an "ORM-style" single-row INSERT/UPDATE statement. The row returned by the statement is also consumed implicitly when[ValuesBase.return\_defaults()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase.return_defaults" \o "sqlalchemy.sql.expression.ValuesBase.return_defaults) is used. By contrast, [UpdateBase.returning()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase.returning" \o "sqlalchemy.sql.expression.UpdateBase.returning) leaves the RETURNING result-set intact with a collection of any number of rows.ValuesBase.return\_defaults（）仅用于与一行匹配的INSERT或UPDATE语句。 虽然一般意义上的RETURNING构造支持多行UPDATE或DELETE语句，或者对于返回多行的INSERT的特殊情况（例如INSERT SELECT，多值VALUES子句），ValuesBase.return\_defaults（）是 仅用于“ORM风格”的单行INSERT / UPDATE语句。 语句返回的行也会在使用ValueBase.return\_defaults（）时隐式消耗。 相比之下，UpdateBase.returning（）将RETURNING结果集保留为任意数量行的集合。
2. It is compatible with the existing logic to fetch auto-generated primary key values, also known as "implicit returning". Backends that support RETURNING will automatically make use of RETURNING in order to fetch the value of newly generated primary keys; while the[UpdateBase.returning()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase.returning" \o "sqlalchemy.sql.expression.UpdateBase.returning) method circumvents this behavior, [ValuesBase.return\_defaults()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase.return_defaults" \o "sqlalchemy.sql.expression.ValuesBase.return_defaults) leaves it intact.与现有的逻辑兼容，以获取自动生成的主键值，也称为“隐式返回”。 支持RETURNING的后端将自动使用RETURNING来获取新生成的主键的值; 而UpdateBase.returning（）方法规避了这种行为，而ValuesBase.return\_defaults（）则保持不变。
3. It can be called against any backend. Backends that don't support RETURNING will skip the usage of the feature, rather than raising an exception. The return value of [ResultProxy.returned\_defaults](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.returned_defaults" \o "sqlalchemy.engine.ResultProxy.returned_defaults) will be None它可以被称为反对任何后端。 不支持RETURNING的后端将跳过该功能的使用，而不是引发异常。 ResultProxy.returned\_defaults的返回值将是None

[ValuesBase.return\_defaults()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase.return_defaults" \o "sqlalchemy.sql.expression.ValuesBase.return_defaults) is used by the ORM to provide an efficient implementation for the eager\_defaults feature of [mapper()](http://docs.sqlalchemy.org/en/rel_1_1/orm/mapping_api.html" \l "sqlalchemy.orm.mapper" \o "sqlalchemy.orm.mapper).

ORM使用ValuesBase.return\_defaults（）为mapper（）的eager\_defaults特性提供了一个高效的实现。

|  |  |
| --- | --- |
| **Parameters:** | ****cols**** – optional list of column key names or [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects. If omitted, all column expressions evaluated on the server are added to the returning list.列键名称或列对象的可选列表。 如果省略，则服务器上评估的所有列表达式都将添加到返回列表中。 |

*New in version 0.9.0.*

**See also**

[UpdateBase.returning()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase.returning" \o "sqlalchemy.sql.expression.UpdateBase.returning)

[ResultProxy.returned\_defaults](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.returned_defaults" \o "sqlalchemy.engine.ResultProxy.returned_defaults)

**values**(*\*args*, *\*\*kwargs*)

specify a fixed VALUES clause for an INSERT statement, or the SET clause for an UPDATE.

为INSERT语句或UPDATE的SET子句指定一个固定的VALUES子句。

Note that the [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 support per-execution time formatting of the VALUES and/or SET clauses, based on the arguments passed to [Connection.execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.execute" \o "sqlalchemy.engine.Connection.execute). However, the [ValuesBase.values()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase.values" \o "sqlalchemy.sql.expression.ValuesBase.values) method can be used to "fix" a particular set of parameters into the statement.

请注意，插入和更新构造基于传递给Connection.execute() 的参数支持VALUES和/或SET子句的每个执行时间格式。 但是，ValuesBase.values() 方法可用于将一组特定的参数“修复”到语句中。

Multiple calls to [ValuesBase.values()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase.values" \o "sqlalchemy.sql.expression.ValuesBase.values) will produce a new construct, each one with the parameter list modified to include the new parameters sent. In the typical case of a single dictionary of parameters, the newly passed keys will replace the same keys in the previous construct. In the case of a list-based "multiple values" construct, each new list of values is extended onto the existing list of values.

对ValuesBase.values() 的多次调用将产生一个新构造，每个构造都修改了参数列表，以包括发送的新参数。 在典型的单个参数字典的情况下，新传递的键将替换先前结构中相同的键。 在基于列表的“多值”结构的情况下，每个新的值列表将扩展到现有的值列表。

|  |  |
| --- | --- |
| **Parameters:** | * ****\*\*kwargs**** –   key value pairs representing the string key of a [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) mapped to the value to be rendered into the VALUES or SET clause:  users.insert().values(name="some name")  users.update().where(users.c.id==5).values(name="some name")   * –   As an alternative to passing key/value parameters, a dictionary, tuple, or list of dictionaries or tuples can be passed as a single positional argument in order to form the VALUES or SET clause of the statement. The forms that are accepted vary based on whether this is an [Insert](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert" \o "sqlalchemy.sql.expression.Insert) or an [Update](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Update" \o "sqlalchemy.sql.expression.Update)construct.  作为传递键/值参数的替代方法，字典，元组或字典或元组列表可以作为单个位置参数传递，以形成语句的VALUES或SET子句。 接受的表单根据这是插入还是更新构造而有所不同。  For either an [Insert](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert" \o "sqlalchemy.sql.expression.Insert) or [Update](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Update" \o "sqlalchemy.sql.expression.Update) construct, a single dictionary can be passed, which works the same as that of the kwargs form:  对于“插入”或“更新”构造，可以传递单个字典，其工作方式与kwargs形式相同：  users.insert().values({"name": "some name"})  users.update().values({"name": "some new name"})  Also for either form but more typically for the [Insert](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert" \o "sqlalchemy.sql.expression.Insert) construct, a tuple that contains an entry for every column in the table is also accepted:同样对于任何一种形式，但更常用于“插入”构造，也可以接受包含表中每列的条目的元组：  users.insert().values((5, "some name"))  The [Insert](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert" \o "sqlalchemy.sql.expression.Insert) construct also supports being passed a list of dictionaries or full-table-tuples, which on the server will render the less common SQL syntax of "multiple values" - this syntax is supported on backends such as SQLite, PostgreSQL, MySQL, but not necessarily others:  “插入”构造还支持传递一个字典列表或全表格元组，其中服务器将呈现“多个值”的较不常见的SQL语法 - 后缀如SQLite，PostgreSQL，MySQL等支持此语法，但是 不一定是其他的：  users.insert().values([  {"name": "some name"},  {"name": "some other name"},  {"name": "yet another name"},  ])  The above form would render a multiple VALUES statement similar to:上述形式将呈现类似于以下的多个VALUES语句：  INSERT INTO users (name) VALUES  (:name\_1),  (:name\_2),  (:name\_3)  It is essential to note that ****passing multiple values is NOT the same as using traditional executemany() form****. The above syntax is a ****special**** syntax not typically used. To emit an INSERT statement against multiple rows, the normal method is to pass a multiple values list to the [Connection.execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.execute" \o "sqlalchemy.engine.Connection.execute) method, which is supported by all database backends and is generally more efficient for a very large number of parameters.  注意传递多个值与使用传统的executemany（）形式是不一样的。 上面的语法是通常不使用的特殊语法。 要针对多行发出INSERT语句，常规方法是将多值列表传递给Connection.execute（）方法，该方法由所有数据库后端支持，并且对于大量参数通常更高效。  **See also**  [Executing Multiple Statements](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html" \l "execute-multiple) - an introduction to the traditional Core method of multiple parameter set invocation for INSERTs and other statements.  *Changed in version 1.0.0:*an INSERT that uses a multiple-VALUES clause, even a list of length one, implies that the [Insert.inline](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert.params.inline" \o "sqlalchemy.sql.expression.Insert) flag is set to True, indicating that the statement will not attempt to fetch the "last inserted primary key" or other defaults. The statement deals with an arbitrary number of rows, so the [ResultProxy.inserted\_primary\_key](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.inserted_primary_key" \o "sqlalchemy.engine.ResultProxy.inserted_primary_key) accessor does not apply.  执行多个语句 - INSERT和其他语句的多参数集调用的传统Core方法的介绍。  在版本1.0.0中进行了更改：使用多个VALUES子句（即使是长度为1的列表）的INSERT意味着将Insert.inline标志设置为True，表示该语句不会尝试读取“上次插入的主要 键“或其他默认值。 该语句处理任意数量的行，所以ResultProxy.inserted\_primary\_key访问器不适用。  *Changed in version 1.0.0:*A multiple-VALUES INSERT now supports columns with Python side default values and callables in the same way as that of an "executemany" style of invocation; the callable is invoked for each row. See [Python-side defaults invoked for each row invidually when using a multivalued insert](http://docs.sqlalchemy.org/en/rel_1_1/changelog/migration_10.html" \l "bug-3288) for other details.  在版本1.0.0中进行了更改：多值插入现在支持具有Python侧默认值和可调用列的列，方式与“executemany”样式的调用相同; 可调用的是每行调用的。 当为其他细节使用多值插入时，请参阅为每行调用的Python方面的默认值。  The [Update](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Update" \o "sqlalchemy.sql.expression.Update) construct supports a special form which is a list of 2-tuples, which when provided must be passed in conjunction with 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) parameter. This form causes the UPDATE statement to render the SET clauses using the order of parameters given to [Update.values()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Update.values" \o "sqlalchemy.sql.expression.Update.values), rather than the ordering of columns given in the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table).  Update构造支持一个特殊的形式，它是一个2元组列表，当提供的时候必须和preserve\_parameter\_order参数一起传递。 这种形式导致UPDATE语句使用给定给Update.values（）的参数的顺序来呈现SET子句，而不是表中给出的列的顺序。  *New in version 1.0.10:*- added support for parameter-ordered UPDATE statements via 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.  **See also**  [Parameter-Ordered Updates](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html" \l "updates-order-parameters) - full example of 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 |

**See also**

[Inserts, Updates and Deletes](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html" \l "inserts-and-updates) - SQL Expression Language Tutorial

[insert()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.insert" \o "sqlalchemy.sql.expression.insert) - produce an INSERT statement

[update()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.update" \o "sqlalchemy.sql.expression.update) - produce an UPDATE statement

## 2.4 SQL and Generic Functions

SQL functions which are known to SQLAlchemy with regards to database-specific rendering, return types and argument behavior. Generic functions are invoked like all SQL functions, using the [func](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.func" \o "sqlalchemy.sql.expression.func) attribute:

关于数据库特定渲染，返回类型和参数行为，SQLAlchemy已知的SQL函数。 泛型函数像所有SQL函数一样被调用，使用func属性：

select([func.count()]).select\_from(sometable)

Note that any name not known to [func](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.func" \o "sqlalchemy.sql.expression.func) generates the function name as is - there is no restriction on what SQL functions can be called, known or unknown to SQLAlchemy, built-in or user defined. The section here only describes those functions where SQLAlchemy already knows what argument and return types are in use.

请注意，func不知道的任何名称都会生成函数名称 - 对SQLAlchemy，内置或用户定义的SQL函数可以调用的SQL函数没有任何限制。 这里的部分仅描述那些SQLAlchemy已经知道使用了什么参数和返回类型的函数。

SQL function API, factories, and built-in functions.

*class*sqlalchemy.sql.functions.**AnsiFunction**(*\*\*kwargs*)

Bases: [sqlalchemy.sql.functions.GenericFunction](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.GenericFunction" \o "sqlalchemy.sql.functions.GenericFunction)

**identifier***= 'AnsiFunction'*

**name***= 'AnsiFunction'*

*class*sqlalchemy.sql.functions.**Function**(*name*, *\*clauses*, *\*\*kw*)

Bases: [sqlalchemy.sql.functions.FunctionElement](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.FunctionElement" \o "sqlalchemy.sql.functions.FunctionElement)

Describe a named SQL function.

See the superclass [FunctionElement](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.FunctionElement" \o "sqlalchemy.sql.functions.FunctionElement) for a description of public methods.

**See also**

[func](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.func" \o "sqlalchemy.sql.expression.func) - namespace which produces registered or ad-hoc [Function](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.Function" \o "sqlalchemy.sql.functions.Function) instances.

[GenericFunction](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.GenericFunction" \o "sqlalchemy.sql.functions.GenericFunction) - allows creation of registered function types.

**\_\_init\_\_**(*name*, *\*clauses*, *\*\*kw*)

Construct a [Function](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.Function" \o "sqlalchemy.sql.functions.Function).

The [func](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.func" \o "sqlalchemy.sql.expression.func) construct is normally used to construct new [Function](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.Function" \o "sqlalchemy.sql.functions.Function) instances.

*class*sqlalchemy.sql.functions.**FunctionElement**(*\*clauses*, *\*\*kwargs*)

Bases: [sqlalchemy.sql.expression.Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable), [sqlalchemy.sql.expression.ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement), [sqlalchemy.sql.expression.FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Base for SQL function-oriented constructs.

**See also**

[Function](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.Function" \o "sqlalchemy.sql.functions.Function) - named SQL function.

[func](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.func" \o "sqlalchemy.sql.expression.func) - namespace which produces registered or ad-hoc [Function](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.Function" \o "sqlalchemy.sql.functions.Function) instances.

[GenericFunction](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.GenericFunction" \o "sqlalchemy.sql.functions.GenericFunction) - allows creation of registered function types.

**\_\_init\_\_**(*\*clauses*, *\*\*kwargs*)

Construct a [FunctionElement](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.FunctionElement" \o "sqlalchemy.sql.functions.FunctionElement).

**alias**(*name=None*, *flat=False*)

Produce a [Alias](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Alias" \o "sqlalchemy.sql.expression.Alias) construct against this [FunctionElement](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.FunctionElement" \o "sqlalchemy.sql.functions.FunctionElement).

This construct wraps the function in a named alias which is suitable for the FROM clause, in the style accepted for example by PostgreSQL.

e.g.:

**from** **sqlalchemy.sql** **import** column

stmt = select([column('data\_view')]).\

select\_from(SomeTable).\

select\_from(func.unnest(SomeTable.data).alias('data\_view'))

Would produce:

**SELECT** data\_view

**FROM** sometable, **unnest**(sometable.**data**) **AS** data\_view

*New in version 0.9.8:*The [FunctionElement.alias()](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.FunctionElement.alias" \o "sqlalchemy.sql.functions.FunctionElement.alias) method is now supported. Previously, this method's behavior was undefined and did not behave consistently across versions.

**clauses**

Return the underlying [ClauseList](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseList" \o "sqlalchemy.sql.expression.ClauseList) which contains the arguments for this [FunctionElement](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.FunctionElement" \o "sqlalchemy.sql.functions.FunctionElement).

**columns**

The set of columns exported by this [FunctionElement](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.FunctionElement" \o "sqlalchemy.sql.functions.FunctionElement).

Function objects currently have no result column names built in; this method returns a single-element column collection with an anonymously named column.

An interim approach to providing named columns for a function as a FROM clause is to build a [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) with the desired columns:

**from** **sqlalchemy.sql** **import** column

stmt = select([column('x'), column('y')]). select\_from(func.myfunction())

**execute**()

Execute this [FunctionElement](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.FunctionElement" \o "sqlalchemy.sql.functions.FunctionElement) against an embedded 'bind'.

This first calls [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.FunctionElement.select" \o "sqlalchemy.sql.functions.FunctionElement.select) to produce a SELECT construct.

Note that [FunctionElement](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.FunctionElement" \o "sqlalchemy.sql.functions.FunctionElement) can be passed to the [Connectable.execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connectable.execute" \o "sqlalchemy.engine.Connectable.execute) method of [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) or [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine).

**filter**(*\*criterion*)

Produce a FILTER clause against this function.

Used against aggregate and window functions, for database backends that support the "FILTER" clause.

The expression:

func.count(1).filter(**True**)

is shorthand for:

**from** **sqlalchemy** **import** funcfilter

funcfilter(func.count(1), **True**)

*New in version 1.0.0.*

**See also**

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

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

**get\_children**(*\*\*kwargs*)

**over**(*partition\_by=None*, *order\_by=None*, *rows=None*, *range\_=None*)

Produce an OVER clause against this function.

Used against aggregate or so-called "window" functions, for database backends that support window functions.

The expression:

func.row\_number().over(order\_by='x')

is shorthand for:

**from** **sqlalchemy** **import** over

over(func.row\_number(), order\_by='x')

See [over()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.over" \o "sqlalchemy.sql.expression.over) for a full description.

*New in version 0.7.*

**packagenames***= ()*

**scalar**()

Execute this [FunctionElement](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.FunctionElement" \o "sqlalchemy.sql.functions.FunctionElement) against an embedded 'bind' and return a scalar value.

This first calls [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.FunctionElement.select" \o "sqlalchemy.sql.functions.FunctionElement.select) to produce a SELECT construct.

Note that [FunctionElement](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.FunctionElement" \o "sqlalchemy.sql.functions.FunctionElement) can be passed to the [Connectable.scalar()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connectable.scalar" \o "sqlalchemy.engine.Connectable.scalar) method of [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) or [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine).

**select**()

Produce a [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) construct against this [FunctionElement](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.FunctionElement" \o "sqlalchemy.sql.functions.FunctionElement).

This is shorthand for:

s = select([function\_element])

**self\_group**(*against=None*)

**within\_group**(*\*order\_by*)

Produce a WITHIN GROUP (ORDER BY expr) clause against this function.

针对此函数生成WITHIN GROUP（ORDER BY expr）子句。

Used against so-called "ordered set aggregate" and "hypothetical set aggregate" functions, including [percentile\_cont](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.percentile_cont" \o "sqlalchemy.sql.functions.percentile_cont), [rank](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.rank" \o "sqlalchemy.sql.functions.rank), [dense\_rank](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.dense_rank" \o "sqlalchemy.sql.functions.dense_rank), etc.

用于所谓的“有序集合”和“假设集合”功能，包括percentile\_cont，rank，dense\_rank等。

See [within\_group()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.within_group" \o "sqlalchemy.sql.expression.within_group) for a full description.

有关完整描述，请参阅within\_group() 。

*New in version 1.1.*

**within\_group\_type**(*within\_group*)

For types that define their return type as based on the criteria within a WITHIN GROUP (ORDER BY) expression, called by the [WithinGroup](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.WithinGroup" \o "sqlalchemy.sql.expression.WithinGroup) construct.

对于根据内部GROUP（ORDER BY）表达式中由WithinGroup构造调用的条件定义其返回类型的类型。

Returns None by default, in which case the function's normal .type is used.

默认情况下返回None，在这种情况下，使用函数的normal .type。

*class*sqlalchemy.sql.functions.**GenericFunction**(*\*args*, *\*\*kwargs*)

Bases: [sqlalchemy.sql.functions.Function](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.Function" \o "sqlalchemy.sql.functions.Function)

Define a 'generic' function.

定义“通用”函数。

A generic function is a pre-established [Function](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.Function" \o "sqlalchemy.sql.functions.Function) class that is instantiated automatically when called by name from the [func](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.func" \o "sqlalchemy.sql.expression.func) attribute. Note that calling any name from [func](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.func" \o "sqlalchemy.sql.expression.func) has the effect that a new [Function](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.Function" \o "sqlalchemy.sql.functions.Function) instance is created automatically, given that name. The primary use case for defining a [GenericFunction](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.GenericFunction" \o "sqlalchemy.sql.functions.GenericFunction) class is so that a function of a particular name may be given a fixed return type. It can also include custom argument parsing schemes as well as additional methods.

通用函数是一个预先建立的Function类，它通过名称从func属性调用时自动实例化。 请注意，从func调用任何名称会产生一个新的Function实例自动创建，给定该名称。 定义GenericFunction类的主要用例是使特定名称的函数可以被赋予固定的返回类型。 它还可以包括自定义参数解析方案以及其他方法。

Subclasses of [GenericFunction](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.GenericFunction" \o "sqlalchemy.sql.functions.GenericFunction) are automatically registered under the name of the class. For example, a user-defined function as\_utc() would be available immediately:

GenericFunction的子类自动注册在类的名称下。 例如，用户定义的函数as\_utc() 将立即可用：

**from** **sqlalchemy.sql.functions** **import** GenericFunction

**from** **sqlalchemy.types** **import** DateTime

**class** **as\_utc**(GenericFunction):

type = DateTime

print select([func.as\_utc()])

User-defined generic functions can be organized into packages by specifying the "package" attribute when defining [GenericFunction](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.GenericFunction" \o "sqlalchemy.sql.functions.GenericFunction). Third party libraries containing many functions may want to use this in order to avoid name conflicts with other systems. For example, if our as\_utc() function were part of a package "time":

用户定义的泛型函数可以通过在定义GenericFunction时指定“package”属性来组织成包。 包含许多功能的第三方库可能希望使用它来避免与其他系统的名称冲突。 例如，如果我们的as\_utc() 函数是包“time”的一部分：

**class** **as\_utc**(GenericFunction):

type = DateTime

package = "time"

The above function would be available from [func](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.func" \o "sqlalchemy.sql.expression.func) using the package name time:

以上功能将从func中使用包名称时间可用：

print select([func.time.as\_utc()])

A final option is to allow the function to be accessed from one name in [func](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.func" \o "sqlalchemy.sql.expression.func) but to render as a different name. The identifier attribute will override the name used to access the function as loaded from [func](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.func" \o "sqlalchemy.sql.expression.func), but will retain the usage of name as the rendered name:

最后一个选择是允许从func中的一个名称访问该函数，但是以不同的名称呈现。 标识符属性将覆盖用于访问从func加载的函数的名称，但将保留名称作为呈现名称的用法：

**class** **GeoBuffer**(GenericFunction):

type = Geometry

package = "geo"

name = "ST\_Buffer"

identifier = "buffer"

The above function will render as follows:

上述功能如下：

**>>>** print func.geo.buffer()ST\_Buffer()

*New in version 0.8:*[GenericFunction](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.GenericFunction" \o "sqlalchemy.sql.functions.GenericFunction) now supports automatic registration of new functions as well as package and custom naming support.

新版本0.8：GenericFunction现在支持自动注册新功能以及包和自定义命名支持。

*Changed in version 0.8:*The attribute name type is used to specify the function's return type at the class level. Previously, the name \_\_return\_type\_\_ was used. This name is still recognized for backwards-compatibility.

在版本0.8中更改：属性名称类型用于在类级别指定函数的返回类型。 以前，使用了\_\_return\_type\_\_的名字。 此名称仍然被识别为向后兼容性。

**coerce\_arguments***= True*

**identifier***= 'GenericFunction'*

**name***= 'GenericFunction'*

*class*sqlalchemy.sql.functions.**OrderedSetAgg**(*\*args*, *\*\*kwargs*)

Bases: [sqlalchemy.sql.functions.GenericFunction](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.GenericFunction" \o "sqlalchemy.sql.functions.GenericFunction)

Define a function where the return type is based on the sort expression type as defined by the expression passed to the[FunctionElement.within\_group()](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.FunctionElement.within_group" \o "sqlalchemy.sql.functions.FunctionElement.within_group) method.

**array\_for\_multi\_clause***= False*

**identifier***= 'OrderedSetAgg'*

**name***= 'OrderedSetAgg'*

**within\_group\_type**(*within\_group*)

*class*sqlalchemy.sql.functions.**ReturnTypeFromArgs**(*\*args*, *\*\*kwargs*)

Bases: [sqlalchemy.sql.functions.GenericFunction](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.GenericFunction" \o "sqlalchemy.sql.functions.GenericFunction)

Define a function whose return type is the same as its arguments.

**identifier***= 'ReturnTypeFromArgs'*

**name***= 'ReturnTypeFromArgs'*

*class*sqlalchemy.sql.functions.**array\_agg**(*\*args*, *\*\*kwargs*)

Bases: [sqlalchemy.sql.functions.GenericFunction](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.GenericFunction" \o "sqlalchemy.sql.functions.GenericFunction)

support for the ARRAY\_AGG function.

The func.array\_agg(expr) construct returns an expression of type [types.ARRAY](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.ARRAY" \o "sqlalchemy.types.ARRAY).

e.g.:

stmt = select([func.array\_agg(table.c.values)[2:5]])

*New in version 1.1.*

**See also**

[postgresql.array\_agg()](http://docs.sqlalchemy.org/en/rel_1_1/dialects/postgresql.html" \l "sqlalchemy.dialects.postgresql.array_agg" \o "sqlalchemy.dialects.postgresql.array_agg) - PostgreSQL-specific version that returns [postgresql.ARRAY](http://docs.sqlalchemy.org/en/rel_1_1/dialects/postgresql.html" \l "sqlalchemy.dialects.postgresql.ARRAY" \o "sqlalchemy.dialects.postgresql.ARRAY), which has PG-specific operators added.

**identifier***= 'array\_agg'*

**name***= 'array\_agg'*

**type**

alias of ARRAY

*class*sqlalchemy.sql.functions.**char\_length**(*arg*, *\*\*kwargs*)

Bases: [sqlalchemy.sql.functions.GenericFunction](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.GenericFunction" \o "sqlalchemy.sql.functions.GenericFunction)

**identifier***= 'char\_length'*

**name***= 'char\_length'*

**type**

alias of Integer

*class*sqlalchemy.sql.functions.**coalesce**(*\*args*, *\*\*kwargs*)

Bases: [sqlalchemy.sql.functions.ReturnTypeFromArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.ReturnTypeFromArgs" \o "sqlalchemy.sql.functions.ReturnTypeFromArgs)

**identifier***= 'coalesce'*

**name***= 'coalesce'*

*class*sqlalchemy.sql.functions.**concat**(*\*args*, *\*\*kwargs*)

Bases: [sqlalchemy.sql.functions.GenericFunction](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.GenericFunction" \o "sqlalchemy.sql.functions.GenericFunction)

**identifier***= 'concat'*

**name***= 'concat'*

**type**

alias of String

*class*sqlalchemy.sql.functions.**count**(*expression=None*, *\*\*kwargs*)

Bases: [sqlalchemy.sql.functions.GenericFunction](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.GenericFunction" \o "sqlalchemy.sql.functions.GenericFunction)

The ANSI COUNT aggregate function. With no arguments, emits COUNT \*.

**identifier***= 'count'*

**name***= 'count'*

**type**

alias of Integer

*class*sqlalchemy.sql.functions.**cume\_dist**(*\*args*, *\*\*kwargs*)

Bases: [sqlalchemy.sql.functions.GenericFunction](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.GenericFunction" \o "sqlalchemy.sql.functions.GenericFunction)

Implement the cume\_dist hypothetical-set aggregate function.

This function must be used with the [FunctionElement.within\_group()](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.FunctionElement.within_group" \o "sqlalchemy.sql.functions.FunctionElement.within_group) modifier to supply a sort expression to operate upon.

The return type of this function is [Numeric](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Numeric" \o "sqlalchemy.types.Numeric).

实现cume\_dist假设集合函数。

必须与FunctionElement.within\_group() 修饰符一起使用此函数来提供要进行操作的排序表达式。

此函数的返回类型为Numeric。

*New in version 1.1.*

**identifier***= 'cume\_dist'*

**name***= 'cume\_dist'*

**type***= Numeric()*

*class*sqlalchemy.sql.functions.**current\_date**(*\*\*kwargs*)

Bases: [sqlalchemy.sql.functions.AnsiFunction](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.AnsiFunction" \o "sqlalchemy.sql.functions.AnsiFunction)

**identifier***= 'current\_date'*

**name***= 'current\_date'*

**type**

alias of Date

*class*sqlalchemy.sql.functions.**current\_time**(*\*\*kwargs*)

Bases: [sqlalchemy.sql.functions.AnsiFunction](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.AnsiFunction" \o "sqlalchemy.sql.functions.AnsiFunction)

**identifier***= 'current\_time'*

**name***= 'current\_time'*

**type**

alias of Time

*class*sqlalchemy.sql.functions.**current\_timestamp**(*\*\*kwargs*)

Bases: [sqlalchemy.sql.functions.AnsiFunction](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.AnsiFunction" \o "sqlalchemy.sql.functions.AnsiFunction)

**identifier***= 'current\_timestamp'*

**name***= 'current\_timestamp'*

**type**

alias of DateTime

*class*sqlalchemy.sql.functions.**current\_user**(*\*\*kwargs*)

Bases: [sqlalchemy.sql.functions.AnsiFunction](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.AnsiFunction" \o "sqlalchemy.sql.functions.AnsiFunction)

**identifier***= 'current\_user'*

**name***= 'current\_user'*

**type**

alias of String

*class*sqlalchemy.sql.functions.**dense\_rank**(*\*args*, *\*\*kwargs*)

Bases: [sqlalchemy.sql.functions.GenericFunction](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.GenericFunction" \o "sqlalchemy.sql.functions.GenericFunction)

Implement the dense\_rank hypothetical-set aggregate function.

This function must be used with the [FunctionElement.within\_group()](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.FunctionElement.within_group" \o "sqlalchemy.sql.functions.FunctionElement.within_group) modifier to supply a sort expression to operate upon.

The return type of this function is [Integer](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Integer" \o "sqlalchemy.types.Integer).

实现dense\_rank假设集合函数。

必须与FunctionElement.within\_group() 修饰符一起使用此函数来提供要进行操作的排序表达式。

此函数的返回类型为Integer。

*New in version 1.1.*

**identifier***= 'dense\_rank'*

**name***= 'dense\_rank'*

**type***= Integer()*

*class*sqlalchemy.sql.functions.**localtime**(*\*\*kwargs*)

Bases: [sqlalchemy.sql.functions.AnsiFunction](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.AnsiFunction" \o "sqlalchemy.sql.functions.AnsiFunction)

**identifier***= 'localtime'*

**name***= 'localtime'*

**type**

alias of DateTime

*class*sqlalchemy.sql.functions.**localtimestamp**(*\*\*kwargs*)

Bases: [sqlalchemy.sql.functions.AnsiFunction](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.AnsiFunction" \o "sqlalchemy.sql.functions.AnsiFunction)

**identifier***= 'localtimestamp'*

**name***= 'localtimestamp'*

**type**

alias of DateTime

*class*sqlalchemy.sql.functions.**max**(*\*args*, *\*\*kwargs*)

Bases: [sqlalchemy.sql.functions.ReturnTypeFromArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.ReturnTypeFromArgs" \o "sqlalchemy.sql.functions.ReturnTypeFromArgs)

**identifier***= 'max'*

**name***= 'max'*

*class*sqlalchemy.sql.functions.**min**(*\*args*, *\*\*kwargs*)

Bases: [sqlalchemy.sql.functions.ReturnTypeFromArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.ReturnTypeFromArgs" \o "sqlalchemy.sql.functions.ReturnTypeFromArgs)

**identifier***= 'min'*

**name***= 'min'*

*class*sqlalchemy.sql.functions.**mode**(*\*args*, *\*\*kwargs*)

Bases: [sqlalchemy.sql.functions.OrderedSetAgg](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.OrderedSetAgg" \o "sqlalchemy.sql.functions.OrderedSetAgg)

implement the mode ordered-set aggregate function.

This function must be used with the [FunctionElement.within\_group()](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.FunctionElement.within_group" \o "sqlalchemy.sql.functions.FunctionElement.within_group) modifier to supply a sort expression to operate upon.

The return type of this function is the same as the sort expression.

*New in version 1.1.*

**identifier***= 'mode'*

**name***= 'mode'*

*class*sqlalchemy.sql.functions.**next\_value**(*seq*, *\*\*kw*)

Bases: [sqlalchemy.sql.functions.GenericFunction](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.GenericFunction" \o "sqlalchemy.sql.functions.GenericFunction)

Represent the 'next value', given a [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) as its single argument.

Compiles into the appropriate function on each backend, or will raise NotImplementedError if used on a backend that does not provide support for sequences.

**identifier***= 'next\_value'*

**name***= 'next\_value'*

**type***= Integer()*

*class*sqlalchemy.sql.functions.**now**(*\*args*, *\*\*kwargs*)

Bases: [sqlalchemy.sql.functions.GenericFunction](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.GenericFunction" \o "sqlalchemy.sql.functions.GenericFunction)

**identifier***= 'now'*

**name***= 'now'*

**type**

alias of DateTime

*class*sqlalchemy.sql.functions.**percent\_rank**(*\*args*, *\*\*kwargs*)

Bases: [sqlalchemy.sql.functions.GenericFunction](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.GenericFunction" \o "sqlalchemy.sql.functions.GenericFunction)

Implement the percent\_rank hypothetical-set aggregate function.

This function must be used with the [FunctionElement.within\_group()](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.FunctionElement.within_group" \o "sqlalchemy.sql.functions.FunctionElement.within_group) modifier to supply a sort expression to operate upon.

The return type of this function is [Numeric](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Numeric" \o "sqlalchemy.types.Numeric).

*New in version 1.1.*

**identifier***= 'percent\_rank'*

**name***= 'percent\_rank'*

**type***= Numeric()*

*class*sqlalchemy.sql.functions.**percentile\_cont**(*\*args*, *\*\*kwargs*)

Bases: [sqlalchemy.sql.functions.OrderedSetAgg](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.OrderedSetAgg" \o "sqlalchemy.sql.functions.OrderedSetAgg)

implement the percentile\_cont ordered-set aggregate function.

This function must be used with the [FunctionElement.within\_group()](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.FunctionElement.within_group" \o "sqlalchemy.sql.functions.FunctionElement.within_group) modifier to supply a sort expression to operate upon.

The return type of this function is the same as the sort expression, or if the arguments are an array, an [types.ARRAY](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.ARRAY" \o "sqlalchemy.types.ARRAY) of the sort expression's type.

*New in version 1.1.*

**array\_for\_multi\_clause***= True*

**identifier***= 'percentile\_cont'*

**name***= 'percentile\_cont'*

*class*sqlalchemy.sql.functions.**percentile\_disc**(*\*args*, *\*\*kwargs*)

Bases: [sqlalchemy.sql.functions.OrderedSetAgg](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.OrderedSetAgg" \o "sqlalchemy.sql.functions.OrderedSetAgg)

implement the percentile\_disc ordered-set aggregate function.

This function must be used with the [FunctionElement.within\_group()](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.FunctionElement.within_group" \o "sqlalchemy.sql.functions.FunctionElement.within_group) modifier to supply a sort expression to operate upon.

The return type of this function is the same as the sort expression, or if the arguments are an array, an [types.ARRAY](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.ARRAY" \o "sqlalchemy.types.ARRAY) of the sort expression's type.

*New in version 1.1.*

**array\_for\_multi\_clause***= True*

**identifier***= 'percentile\_disc'*

**name***= 'percentile\_disc'*

*class*sqlalchemy.sql.functions.**random**(*\*args*, *\*\*kwargs*)

Bases: [sqlalchemy.sql.functions.GenericFunction](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.GenericFunction" \o "sqlalchemy.sql.functions.GenericFunction)

**identifier***= 'random'*

**name***= 'random'*

*class*sqlalchemy.sql.functions.**rank**(*\*args*, *\*\*kwargs*)

Bases: [sqlalchemy.sql.functions.GenericFunction](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.GenericFunction" \o "sqlalchemy.sql.functions.GenericFunction)

Implement the rank hypothetical-set aggregate function.

This function must be used with the [FunctionElement.within\_group()](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.FunctionElement.within_group" \o "sqlalchemy.sql.functions.FunctionElement.within_group) modifier to supply a sort expression to operate upon.

The return type of this function is [Integer](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Integer" \o "sqlalchemy.types.Integer).

*New in version 1.1.*

**identifier***= 'rank'*

**name***= 'rank'*

**type***= Integer()*

sqlalchemy.sql.functions.**register\_function**(*identifier*, *fn*, *package='\_default'*)

Associate a callable with a particular func. name.

This is normally called by \_GenericMeta, but is also available by itself so that a non-Function construct can be associated with the [func](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.func" \o "sqlalchemy.sql.expression.func) accessor (i.e. CAST, EXTRACT).

*class*sqlalchemy.sql.functions.**session\_user**(*\*\*kwargs*)

Bases: [sqlalchemy.sql.functions.AnsiFunction](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.AnsiFunction" \o "sqlalchemy.sql.functions.AnsiFunction)

**identifier***= 'session\_user'*

**name***= 'session\_user'*

**type**

alias of String

*class*sqlalchemy.sql.functions.**sum**(*\*args*, *\*\*kwargs*)

Bases: [sqlalchemy.sql.functions.ReturnTypeFromArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.ReturnTypeFromArgs" \o "sqlalchemy.sql.functions.ReturnTypeFromArgs)

**identifier***= 'sum'*

**name***= 'sum'*

*class*sqlalchemy.sql.functions.**sysdate**(*\*\*kwargs*)

Bases: [sqlalchemy.sql.functions.AnsiFunction](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.AnsiFunction" \o "sqlalchemy.sql.functions.AnsiFunction)

**identifier***= 'sysdate'*

**name***= 'sysdate'*

**type**

alias of DateTime

*class*sqlalchemy.sql.functions.**user**(*\*\*kwargs*)

Bases: [sqlalchemy.sql.functions.AnsiFunction](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.AnsiFunction" \o "sqlalchemy.sql.functions.AnsiFunction)

**identifier***= 'user'*

**name***= 'user'*

**type**

alias of String

## 2.6 Custom SQL Constructs and Compilation Extension

Provides an API for creation of custom ClauseElements and compilers.

2.6.1 Synopsis

Usage involves the creation of one or more [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement) subclasses and one or more callables defining its compilation:

**from** **sqlalchemy.ext.compiler** **import** compiles

**from** **sqlalchemy.sql.expression** **import** ColumnClause

**class** **MyColumn**(ColumnClause):

**pass**

**@compiles**(MyColumn)

**def** compile\_mycolumn(element, compiler, \*\*kw):

**return** "[*%s*]" % element.name

Above, MyColumn extends [ColumnClause](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnClause" \o "sqlalchemy.sql.expression.ColumnClause), the base expression element for named column objects. The compiles decorator registers itself with the MyColumnclass so that it is invoked when the object is compiled to a string:

**from** **sqlalchemy** **import** select

s = select([MyColumn('x'), MyColumn('y')])print str(s)

Produces:

SELECT [x], [y]

### 2.6.2 Dialect-specific compilation rules

Compilers can also be made dialect-specific. The appropriate compiler will be invoked for the dialect in use:

**from** **sqlalchemy.schema** **import** DDLElement

**class** **AlterColumn**(DDLElement):

**def** \_\_init\_\_(self, column, cmd):

self.column = column

self.cmd = cmd

**@compiles**(AlterColumn)

**def** visit\_alter\_column(element, compiler, \*\*kw):

**return** "ALTER COLUMN *%s* ..." % element.column.name

**@compiles**(AlterColumn, 'postgresql')

**def** visit\_alter\_column(element, compiler, \*\*kw):

**return** "ALTER TABLE *%s* ALTER COLUMN *%s* ..." % (element.table.name,

element.column.name)

The second visit\_alter\_table will be invoked when any postgresql dialect is used.

2.6.3 Compiling sub-elements of a custom expression construct

The compiler argument is the [Compiled](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Compiled" \o "sqlalchemy.engine.interfaces.Compiled) object in use. This object can be inspected for any information about the in-progress compilation, including compiler.dialect, compiler.statement etc. The [SQLCompiler](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.sql.compiler.SQLCompiler" \o "sqlalchemy.sql.compiler.SQLCompiler) and [DDLCompiler](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.sql.compiler.DDLCompiler" \o "sqlalchemy.sql.compiler.DDLCompiler) both include a process() method which can be used for compilation of embedded attributes:

**from** **sqlalchemy.sql.expression** **import** Executable, ClauseElement

**class** **InsertFromSelect**(Executable, ClauseElement):

**def** \_\_init\_\_(self, table, select):

self.table = table

self.select = select

**@compiles**(InsertFromSelect)

**def** visit\_insert\_from\_select(element, compiler, \*\*kw):

**return** "INSERT INTO *%s* (*%s*)" % (

compiler.process(element.table, asfrom=**True**),

compiler.process(element.select)

)

insert = InsertFromSelect(t1, select([t1]).where(t1.c.x>5))

print insert

Produces:

"INSERT INTO mytable (SELECT mytable.x, mytable.y, mytable.z

FROM mytable WHERE mytable.x > :x\_1)"

**Note**

The above InsertFromSelect construct is only an example, this actual functionality is already available using the [Insert.from\_select()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert.from_select" \o "sqlalchemy.sql.expression.Insert.from_select) method.

**Note**

The above InsertFromSelect construct probably wants to have "autocommit" enabled. See [Enabling Autocommit on a Construct](http://docs.sqlalchemy.org/en/rel_1_1/core/compiler.html" \l "enabling-compiled-autocommit) for this step.

### Cross Compiling between SQL and DDL compilers

SQL and DDL constructs are each compiled using different base compilers - SQLCompiler and DDLCompiler. A common need is to access the compilation rules of SQL expressions from within a DDL expression. The DDLCompiler includes an accessor sql\_compiler for this reason, such as below where we generate a CHECK constraint that embeds a SQL expression:

**@compiles**(MyConstraint)

**def** compile\_my\_constraint(constraint, ddlcompiler, \*\*kw):

**return** "CONSTRAINT *%s* CHECK (*%s*)" % (

constraint.name,

ddlcompiler.sql\_compiler.process(

constraint.expression, literal\_binds=**True**)

)

Above, we add an additional flag to the process step as called by SQLCompiler.process(), which is the literal\_binds flag. This indicates that any SQL expression which refers to a [BindParameter](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.BindParameter" \o "sqlalchemy.sql.expression.BindParameter) object or other "literal" object such as those which refer to strings or integers should be rendered ****in-place****, rather than being referred to as a bound parameter; when emitting DDL, bound parameters are typically not supported.

在上面，我们为流程步骤添加了一个额外的标志，由SQLCompiler.process() 调用，它是literal\_binds标志。 这表示引用BindParameter对象或其他“literal”对象（如引用字符串或整数的对象）的任何SQL表达式都应该原位呈现，而不是被称为绑定参数; 当发射DDL时，通常不支持绑定的参数。

2.6.4 Enabling Autocommit on a Construct

Recall from the section [Understanding Autocommit](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "autocommit) that the [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine), when asked to execute a construct in the absence of a user-defined transaction, detects if the given construct represents DML or DDL, that is, a data modification or data definition statement, which requires (or may require, in the case of DDL) that the transaction generated by the DBAPI be committed (recall that DBAPI always has a transaction going on regardless of what SQLAlchemy does). Checking for this is actually accomplished by checking for the "autocommit" execution option on the construct. When building a construct like an INSERT derivation, a new DDL type, or perhaps a stored procedure that alters data, the "autocommit" option needs to be set in order for the statement to function with "connectionless" execution (as described in [Connectionless Execution, Implicit Execution](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "dbengine-implicit)).

从“了解自动提交”一节回想，引擎在被要求在没有用户定义事务的情况下执行构造时会检测给定的结构是否表示DML或DDL，即数据修改或数据定义语句， 或者在DDL的情况下可能需要提交DBAPI生成的事务（请注意，无论SQLAlchemy如何，DBAPI始终都有事务进行）。 通过检查构造上的“自动提交”执行选项，实际上可以检查这一点。 当构建类似于INSERT派生的构造，新的DDL类型或者可能改变数据的存储过程时，需要设置“autocommit”选项，以便语句以“无连接”的方式执行（如“无连接执行” ，隐式执行）。

Currently a quick way to do this is to subclass [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable), then add the "autocommit" flag to the \_execution\_options dictionary (note this is a "frozen" dictionary which supplies a generative union() method):

目前一个快速的方法是将Executable子类化，然后将“autocommit”标志添加到\_execution\_options字典（注意这是一个提供一个生成的union() 方法）的“冻结”字典：

**from** **sqlalchemy.sql.expression** **import** Executable, ClauseElement

**class** **MyInsertThing**(Executable, ClauseElement):

\_execution\_options = \

Executable.\_execution\_options.union({'autocommit': **True**})

More succinctly, if the construct is truly similar to an INSERT, UPDATE, or DELETE, [UpdateBase](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.UpdateBase" \o "sqlalchemy.sql.expression.UpdateBase) can be used, which already is a subclass of [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable), [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement) and includes the autocommit flag:

更简洁，如果构造与INSERT，UPDATE或DELETE真正相似，则可以使用UpdateBase，它已经是Executable，ClauseElement的子类，并且包含autocommit标志：

**from** **sqlalchemy.sql.expression** **import** UpdateBase

**class** **MyInsertThing**(UpdateBase):

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

...

DDL elements that subclass [DDLElement](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.DDLElement" \o "sqlalchemy.schema.DDLElement) already have the "autocommit" flag turned on.

2.6.5 Changing the default compilation of existing constructs

The compiler extension applies just as well to the existing constructs. When overriding the compilation of a built in SQL construct, the @compiles decorator is invoked upon the appropriate class (be sure to use the class, i.e. Insert or Select, instead of the creation function such as insert() or select()).

编译器扩展也适用于现有的构造。 当覆盖内置SQL构造的编译时，@compiles装饰器将在适当的类上被调用（请确保使用类，即插入或选择，而不是像insert() 或select() 这样的创建函数）。

Within the new compilation function, to get at the "original" compilation routine, use the appropriate visit\_XXX method - this because compiler.process() will call upon the overriding routine and cause an endless loop. Such as, to add "prefix" to all insert statements:

在新的编译功能中，要获得"原始"编译例程，请使用适当的visit\_XXX方法 - 这是因为compile.process() 将调用重写例程并导致无限循环。 例如，为所有insert语句添加"prefix"：

**from** **sqlalchemy.sql.expression** **import** Insert

**@compiles**(Insert)

**def** prefix\_inserts(insert, compiler, \*\*kw):

**return** compiler.visit\_insert(insert.prefix\_with("some prefix"), \*\*kw)

The above compiler will prefix all INSERT statements with "some prefix" when compiled.

上述编译器将在编译时将所有INSERT语句前缀为“some prefix”。

2.6.6 Changing Compilation of Types

compiler works for types, too, such as below where we implement the MS-SQL specific 'max' keyword for String/VARCHAR:

编译器也适用于类型，例如下面我们为String / VARCHAR实现MS-SQL特定的'max'关键字：

**@compiles**(String, 'mssql')

**@compiles**(VARCHAR, 'mssql')

**def** compile\_varchar(element, compiler, \*\*kw):

**if** element.length == 'max':

**return** "VARCHAR('max')"

**else**:

**return** compiler.visit\_VARCHAR(element, \*\*kw)

foo = Table('foo', metadata,

Column('data', VARCHAR('max')))

### 2.6.7 Subclassing Guidelines

A big part of using the compiler extension is subclassing SQLAlchemy expression constructs. To make this easier, the expression and schema packages feature a set of "bases" intended for common tasks. A synopsis is as follows:

使用编译器扩展的很大一部分是对SQLAlchemy表达式构造进行子类化。 为了使这更容易，表达式和模式包具有一组用于常见任务的"基础"。 摘要如下：

[ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement) - This is the root expression class. Any SQL expression can be derived from this base, and is probably the best choice for longer constructs such as specialized INSERT statements.

ClauseElement - 这是根表达式类。 任何SQL表达式都可以从此基础派生，并且可能是更长的构造（如专用INSERT语句）的最佳选择。

[ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) - The root of all "column-like" elements. Anything that you'd place in the "columns" clause of a SELECT statement (as well as order by and group by) can derive from this - the object will automatically have Python "comparison" behavior.

ColumnElement - 所有"列类"元素的根。 任何你放在SELECT语句（以及order by和group by）的"columns"子句中的东西都可以从这里得到 - 对象将自动具有Python"比较"的行为。

[ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) classes want to have a type member which is expression's return type. This can be established at the instance level in the constructor, or at the class level if its generally constant:

ColumnElement类希望有一个类型成员，它是表达式的返回类型。 这可以在构造函数的实例级别建立，或者在类级别建立，如果它通常是常量的：

**class** **timestamp**(ColumnElement):

type = TIMESTAMP()

[FunctionElement](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.FunctionElement" \o "sqlalchemy.sql.functions.FunctionElement) - This is a hybrid of a ColumnElement and a "from clause" like object, and represents a SQL function or stored procedure type of call. Since most databases support statements along the line of "SELECT FROM <some function>" FunctionElement adds in the ability to be used in the FROM clause of a select() construct:

FunctionElement - 这是一个ColumnElement和一个"from子句"的对象的混合，它表示一个SQL函数或存储过程类型的调用。 由于大多数数据库支持"SELECT FROM <some function"行的功能，所以FunctionElement增加了在select() 结构的FROM子句中使用的能力：

**from** **sqlalchemy.sql.expression** **import** FunctionElement

**class** **coalesce**(FunctionElement):

name = 'coalesce'

**@compiles**(coalesce)

**def** compile(element, compiler, \*\*kw):

**return** "coalesce(*%s*)" % compiler.process(element.clauses)

**@compiles**(coalesce, 'oracle')

**def** compile(element, compiler, \*\*kw):

**if** len(element.clauses) > 2:

**raise** TypeError("coalesce only supports two arguments on Oracle")

**return** "nvl(*%s*)" % compiler.process(element.clauses)

[DDLElement](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.DDLElement" \o "sqlalchemy.schema.DDLElement) - The root of all DDL expressions, like CREATE TABLE, ALTER TABLE, etc. Compilation of DDLElement subclasses is issued by a DDLCompilerinstead of a SQLCompiler. DDLElement also features Table and MetaData event hooks via the execute\_at() method, allowing the construct to be invoked during CREATE TABLE and DROP TABLE sequences.

[Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable) - This is a mixin which should be used with any expression class that represents a "standalone" SQL statement that can be passed directly to an execute() method. It is already implicit within DDLElement and FunctionElement.

2.6.8 Further Examples

### "UTC timestamp" function

A function that works like "CURRENT\_TIMESTAMP" except applies the appropriate conversions so that the time is in UTC time. Timestamps are best stored in relational databases as UTC, without time zones. UTC so that your database doesn't think time has gone backwards in the hour when daylight savings ends, without timezones because timezones are like character encodings - they're best applied only at the endpoints of an application (i.e. convert to UTC upon user input, re-apply desired timezone upon display).

一个类似"CURRENT\_TIMESTAMP"的功能除了应用适当的转换，以便时间在UTC时间。 时间戳最好以UTC形式存储在关系数据库中，无时区。 UTC，以便您的数据库不会在夏令时结束的时候没有时间倒退，没有时区，因为时区就像字符编码 - 它们最好仅应用于应用程序的端点（即在用户输入时转换为UTC ，显示时重新应用所需的时区）。

For PostgreSQL and Microsoft SQL Server:

对于PostgreSQL和Microsoft SQL Server：

**from** **sqlalchemy.sql** **import** expression

**from** **sqlalchemy.ext.compiler** **import** compiles

**from** **sqlalchemy.types** **import** DateTime

**class** **utcnow**(expression.FunctionElement):

type = DateTime()

**@compiles**(utcnow, 'postgresql')

**def** pg\_utcnow(element, compiler, \*\*kw):

**return** "TIMEZONE('utc', CURRENT\_TIMESTAMP)"

**@compiles**(utcnow, 'mssql')

**def** ms\_utcnow(element, compiler, \*\*kw):

**return** "GETUTCDATE()"

Example usage:

**from** **sqlalchemy** **import** (

Table, Column, Integer, String, DateTime, MetaData

)metadata = MetaData()event = Table("event", metadata,

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

Column("description", String(50), nullable=**False**),

Column("timestamp", DateTime, server\_default=utcnow()))

### "GREATEST" function

The "GREATEST" function is given any number of arguments and returns the one that is of the highest value - its equivalent to Python's max function. A SQL standard version versus a CASE based version which only accommodates two arguments:

给出了"GREATEST"函数的任意数量的参数，并返回一个值最高的值，这相当于Python的最大函数。 SQL标准版本与基于CASE的版本，仅适用于两个参数：

**from** **sqlalchemy.sql** **import** expression

**from** **sqlalchemy.ext.compiler** **import** compiles

**from** **sqlalchemy.types** **import** Numeric

**class** **greatest**(expression.FunctionElement):

type = Numeric()

name = 'greatest'

**@compiles**(greatest)

**def** default\_greatest(element, compiler, \*\*kw):

**return** compiler.visit\_function(element)

**@compiles**(greatest, 'sqlite')

**@compiles**(greatest, 'mssql')

**@compiles**(greatest, 'oracle')

**def** case\_greatest(element, compiler, \*\*kw):

arg1, arg2 = list(element.clauses)

**return** "CASE WHEN *%s* > *%s* THEN *%s* ELSE *%s* END" % (

compiler.process(arg1),

compiler.process(arg2),

compiler.process(arg1),

compiler.process(arg2),

)

Example usage:

Session.query(Account).\

filter(

greatest(

Account.checking\_balance,

Account.savings\_balance) > 10000

)

### "false" expression

Render a "false" constant expression, rendering as "0" on platforms that don't have a "false" constant:

渲染"false"常量表达式，在没有"false"常量的平台上呈现为"0"

**from** **sqlalchemy.sql** **import** expression

**from** **sqlalchemy.ext.compiler** **import** compiles

**class** **sql\_false**(expression.ColumnElement):

**pass**

**@compiles**(sql\_false)

**def** default\_false(element, compiler, \*\*kw):

**return** "false"

**@compiles**(sql\_false, 'mssql')

**@compiles**(sql\_false, 'mysql')

**@compiles**(sql\_false, 'oracle')

**def** int\_false(element, compiler, \*\*kw):

**return** "0"

Example usage:

**from** **sqlalchemy** **import** select, union\_all

exp = union\_all(

select([users.c.name, sql\_false().label("enrolled")]),

select([customers.c.name, customers.c.enrolled]))

sqlalchemy.ext.compiler.**compiles**(*class\_*, *\*specs*)

Register a function as a compiler for a given [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement) type.

sqlalchemy.ext.compiler.**deregister**(*class\_*)

Remove all custom compilers associated with a given [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement) type.

## 2.7 Expression Serializer Extension

Serializer/Deserializer objects for usage with SQLAlchemy query structures, allowing "contextual" deserialization.

使用SQLAlchemy查询结构的Serializer / Deserializer对象，允许"上下文"反序列化。

Any SQLAlchemy query structure, either based on sqlalchemy.sql.\* or sqlalchemy.orm.\* can be used. The mappers, Tables, Columns, Session etc. which are referenced by the structure are not persisted in serialized form, but are instead re-associated with the query structure when it is deserialized.

可以使用任何基于sqlalchemy.sql。\*或sqlalchemy.orm。\*的SQLAlchemy查询结构。 由结构引用的映射器，表，列，会话等不会以序列化形式持久化，而是在反序列化时重新与查询结构相关联。

Usage is nearly the same as that of the standard Python pickle module:

用法与Python pickle模块的使用几乎相同：

**from** **sqlalchemy.ext.serializer** **import** loads, dumps

metadata = MetaData(bind=some\_engine)Session = scoped\_session(sessionmaker())

*# ... define mappers*

query = Session.query(MyClass).

filter(MyClass.somedata=='foo').order\_by(MyClass.sortkey)

*# pickle the query*

serialized = dumps(query)

*# unpickle. Pass in metadata + scoped\_session*

query2 = loads(serialized, metadata, Session)

print query2.all()

Similar restrictions as when using raw pickle apply; mapped classes must be themselves be pickleable, meaning they are importable from a module-level namespace.

使用原料泡菜时使用类似的限制; 映射类必须本身是可挑选的，这意味着它们可以从模块级命名空间导入。

The serializer module is only appropriate for query structures. It is not needed for:

串行器模块仅适用于查询结构。 不需要：

* instances of user-defined classes. These contain no references to engines, sessions or expression constructs in the typical case and can be serialized directly.用户定义类的情况。 它们在典型情况下不包含引擎，会话或表达式构造的引用，并且可以直接序列化。
* Table metadata that is to be loaded entirely from the serialized structure (i.e. is not already declared in the application). Regular pickle.loads()/dumps() can be used to fully dump any MetaData object, typically one which was reflected from an existing database at some previous point in time. The serializer module is specifically for the opposite case, where the Table metadata is already present in memory.要从序列化结构中完全加载的表元数据（即尚未在应用程序中声明）。 常规pickle.loads() / dumps() 可用于完全转储任何MetaData对象，通常是在某个先前时间点从现有数据库反映的对象。 串行器模块专门用于相反的情况，其中表元数据已经存在于存储器中。

sqlalchemy.ext.serializer.**Serializer**(*\*args*, *\*\*kw*)

sqlalchemy.ext.serializer.**Deserializer**(*file*, *metadata=None*, *scoped\_session=None*, *engine=None*)

sqlalchemy.ext.serializer.**dumps**(*obj*, *protocol=0*)

sqlalchemy.ext.serializer.**loads**(*data*, *metadata=None*, *scoped\_session=None*, *engine=None*)

# Chapter 3 Schema Definition Language

This section references SQLAlchemy ****schema metadata****, a comprehensive system of describing and inspecting database schemas.

本部分引用SQLAlchemy架构元数据，这是一个描述和检查数据库模式的综合系统。

The core of SQLAlchemy's query and object mapping operations are supported by *database metadata*, which is comprised of Python objects that describe tables and other schema-level objects. These objects are at the core of three major types of operations - issuing CREATE and DROP statements (known as *DDL*), constructing SQL queries, and expressing information about structures that already exist within the database.

SQLAlchemy的查询和对象映射操作的核心由数据库元数据支持，该元数据由描述表和其他模式级对象的Python对象组成。这些对象是三种主要操作类型的核心 - 发布CREATE和DROP语句（称为DDL），构造SQL查询以及表达有关数据库中已存在的结构的信息。

Database metadata can be expressed by explicitly naming the various components and their properties, using constructs such as [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table), [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column), [ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey) and[Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence), all of which are imported from the sqlalchemy.schema package. It can also be generated by SQLAlchemy using a process called *reflection*, which means you start with a single object such as [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table), assign it a name, and then instruct SQLAlchemy to load all the additional information related to that name from a particular engine source.

数据库元数据可以通过使用诸如[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)，[Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column)，[ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey)和[Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence)之类的构造来明确命名各种组件及其属性来表示，所有这些构造都是从sqlalchemy.schema包导入的。它也可以由SQLAlchemy使用称为反射的过程生成，这意味着您从单个对象（如[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)）开始，为其分配名称，然后指示SQLAlchemy从特定引擎源加载与该名称相关的所有其他信息。

A key feature of SQLAlchemy's database metadata constructs is that they are designed to be used in a *declarative* style which closely resembles that of real DDL. They are therefore most intuitive to those who have some background in creating real schema generation scripts.

SQLAlchemy的数据库元数据结构的一个关键特性是它们被设计为以非常类似于真实DDL的声明样式使用。因此，对于在创建真正的模式生成脚本中有一些背景的人来说，这是最直观的。

### 3.1 Describing Databases with MetaData

This section discusses the fundamental [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table), [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) and [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) objects.

本节讨论基本的[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)，[Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column)和[MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData)对象。

A collection of metadata entities is stored in an object aptly named [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData):

元数据实体的集合存储在一个名为[MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData)的对象中：

**from** **sqlalchemy** **import** \*

metadata = MetaData()

[MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) is a container object that keeps together many different features of a database (or multiple databases) being described.

[MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData)是一个容器对象，用于保存正在描述的数据库（或多个数据库）的许多不同功能。

To represent a table, use the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) class. Its two primary arguments are the table name, then the [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) object which it will be associated with. The remaining positional arguments are mostly [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects describing each column:

要表示表，请使用[MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData)类。 它的[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)个主要参数是表名称，然后是与它相关联的[MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData)对象。 剩下的位置参数主要是描述每一列的[Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column)对象：

user = Table('user', metadata,

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

Column('user\_name', String(16), nullable=**False**),

Column('email\_address', String(60)),

Column('password', String(20), nullable=**False**))

Above, a table called user is described, which contains four columns. The primary key of the table consists of the user\_id column. Multiple columns may be assigned the primary\_key=True flag which denotes a multi-column primary key, known as a *composite* primary key.

以上，描述了一个名为user的表，其中包含四列。 该表的主键由user\_id列组成。 可以为多个列分配primary\_key=True标志，该标志表示多列主键，称为复合主键。

Note also that each column describes its datatype using objects corresponding to genericized types, such as [Integer](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Integer" \o "sqlalchemy.types.Integer) and [String](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.String" \o "sqlalchemy.types.String). SQLAlchemy features dozens of types of varying levels of specificity as well as the ability to create custom types. Documentation on the type system can be found at [Column and Data Types](http://docs.sqlalchemy.org/en/rel_1_1/core/types.html).

还要注意，每一列都使用与泛型类型相对应的对象来描述其数据类型，例如[Integer](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Integer" \o "sqlalchemy.types.Integer)和[String](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.String" \o "sqlalchemy.types.String)。 SQLAlchemy具有几十种类型的不同级别的特异性以及创建自定义类型的能力。 类型系统上的文档可以在列和数据类型中找到。

3.1.1 Accessing Tables and Columns

The [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) object contains all of the schema constructs we've associated with it. It supports a few methods of accessing these table objects, such as the sorted\_tables accessor which returns a list of each [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object in order of foreign key dependency (that is, each table is preceded by all tables which it references):

[MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData)对象包含我们与之关联的所有模式结构。 它支持访问这些表对象的几种方法，例如sorted\_tables存取器，它按照外键依赖性的顺序返回每个[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)对象的列表（也就是说，每个表前面都是引用的所有表）：

**>>> for** t **in** metadata.sorted\_tables:

**...**  print(t.name)

user

user\_preference

invoice

invoice\_item

In most cases, individual [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects have been explicitly declared, and these objects are typically accessed directly as module-level variables in an application. Once a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) has been defined, it has a full set of accessors which allow inspection of its properties. Given the following [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) definition:

在大多数情况下，已经显式声明了各个[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)对象，这些对象通常直接作为应用程序中的模块级变量访问。 一旦定义了[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)，它有一整套访问器，可以检查其属性。 给定以下[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)定义：

employees = Table('employees', metadata,

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

Column('employee\_name', String(60), nullable=**False**),

Column('employee\_dept', Integer, ForeignKey("departments.department\_id")))

Note the [ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey) object used in this table - this construct defines a reference to a remote table, and is fully described in [Defining Foreign Keys](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "metadata-foreignkeys). Methods of accessing information about this table include:

注意此表中使用的[ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey)对象 - 此构造定义了对远程表的引用，并在定义外键中进行了详细描述。 访问有关此表的信息的方法包括：

*# access the column "EMPLOYEE\_ID":*

employees.columns.employee\_id

*# or just*

employees.c.employee\_id

*# via string*

employees.c['employee\_id']

*# iterate through all columns*

**for** c **in** employees.c:

print(c)

*# get the table's primary key columns*

**for** primary\_key **in** employees.primary\_key:

print(primary\_key)

*# get the table's foreign key objects:*

**for** fkey **in** employees.foreign\_keys:

print(fkey)

*# access the table's MetaData:*

employees.metadata

*# access the table's bound Engine or Connection, if its MetaData is bound:*

employees.bind

*# access a column's name, type, nullable, primary key, foreign key*

employees.c.employee\_id.name

employees.c.employee\_id.type

employees.c.employee\_id.nullable

employees.c.employee\_id.primary\_key

employees.c.employee\_dept.foreign\_keys

*# get the "key" of a column, which defaults to its name, but can# be any user-defined string:*

employees.c.employee\_name.key

*# access a column's table:*

employees.c.employee\_id.table **is** employees

*# get the table related by a foreign key*

list(employees.c.employee\_dept.foreign\_keys)[0].column.table

### 3.1.2 Creating and Dropping Database Tables

Once you've defined some [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects, assuming you're working with a brand new database one thing you might want to do is issue CREATE statements for those tables and their related constructs (as an aside, it's also quite possible that you *don't* want to do this, if you already have some preferred methodology such as tools included with your database or an existing scripting system - if that's the case, feel free to skip this section - SQLAlchemy has no requirement that it be used to create your tables).

一旦你定义了一些[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)对象，假设你正在使用一个全新的数据库，你可能想做的一件事就是为这些表和它们的相关结构发出CREATE语句（除此之外， 如果您已经有了一些首选的方法，例如您的数据库或现有的脚本系统所包含的工具，如果是这样，请随时跳过本节 - SQLAlchemy不需要用于创建表）。

The usual way to issue CREATE is to use [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) on the [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) object. This method will issue queries that first check for the existence of each individual table, and if not found will issue the CREATE statements:

发布CREATE的通常方法是在[MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData)对象上使用[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)。 此方法将发出查询，首先检查每个单独的表的存在，如果没有找到将发出CREATE语句：

engine = create\_engine('sqlite:///:memory:')

metadata = MetaData()

user = Table('user', metadata,

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

Column('user\_name', String(16), nullable=False),

Column('email\_address', String(60), key='email'),

Column('password', String(20), nullable=False))

user\_prefs = Table('user\_prefs', metadata,

Column('pref\_id', Integer, primary\_key=True),

Column('user\_id', Integer, ForeignKey("user.user\_id"), nullable=False),

Column('pref\_name', String(40), nullable=False),

Column('pref\_value', String(100)))

metadata.create\_all(engine)

PRAGMA table\_info(user){}

CREATE TABLE user(

user\_id INTEGER NOT NULL PRIMARY KEY,

user\_name VARCHAR(16) NOT NULL,

email\_address VARCHAR(60),

password VARCHAR(20) NOT NULL

)

PRAGMA table\_info(user\_prefs){}

CREATE TABLE user\_prefs(

pref\_id INTEGER NOT NULL PRIMARY KEY,

user\_id INTEGER NOT NULL REFERENCES user(user\_id),

pref\_name VARCHAR(40) NOT NULL,

pref\_value VARCHAR(100)

)

[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) creates foreign key constraints between tables usually inline with the table definition itself, and for this reason it also generates the tables in order of their dependency. There are options to change this behavior such that ALTER TABLE is used instead.

[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)创建通常与表定义本身内联的表之间的外键约束，因此，它还会依依依次生成表。 有更改此行为的选项，以改为使用ALTER TABLE。

Dropping all tables is similarly achieved using the [drop\_all()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData.drop_all" \o "sqlalchemy.schema.MetaData.drop_all) method. This method does the exact opposite of [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) - the presence of each table is checked first, and tables are dropped in reverse order of dependency.

使用[drop\_all()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData.drop_all" \o "sqlalchemy.schema.MetaData.drop_all)方法同样实现删除所有表。 该方法与[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)完全相反，首先检查每个表的存在，并按相反的顺序删除表。

Creating and dropping individual tables can be done via the create() and drop() methods of [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table). These methods by default issue the CREATE or DROP regardless of the table being present:

创建和删除单个表可以通过[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)的create()和drop()方法完成。 默认情况下，这些方法会发出CREATE或DROP，而不管该表是否存在：

engine = create\_engine('sqlite:///:memory:')

meta = MetaData()

employees = Table('employees', meta,

Column('employee\_id', Integer, primary\_key=True),

Column('employee\_name', String(60), nullable=False, key='name'),

Column('employee\_dept', Integer, ForeignKey("departments.department\_id")))

employees.create(engine)

CREATE TABLE employees(

employee\_id SERIAL NOT NULL PRIMARY KEY,

employee\_name VARCHAR(60) NOT NULL,

employee\_dept INTEGER REFERENCES departments(department\_id)

)

{}

drop() method:

employees.drop(engine)

DROP TABLE employees

{}

To enable the "check first for the table existing" logic, add the checkfirst=True argument to create() or drop():

employees.create(engine, checkfirst=**True**)

employees.drop(engine, checkfirst=**False**)

### 3.1.3 Altering Schemas through Migrations

While SQLAlchemy directly supports emitting CREATE and DROP statements for schema constructs, the ability to alter those constructs, usually via the ALTER statement as well as other database-specific constructs, is outside of the scope of SQLAlchemy itself. While it's easy enough to emit ALTER statements and similar by hand, such as by passing a string to [Connection.execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.execute" \o "sqlalchemy.engine.Connection.execute) or by using the [DDL](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.DDL" \o "sqlalchemy.schema.DDL) construct, it's a common practice to automate the maintenance of database schemas in relation to application code using schema migration tools.

虽然SQLAlchemy直接支持为模式结构发布CREATE和DROP语句，但通常通过ALTER语句以及其他特定于数据库的结构来更改这些结构的能力不在SQLAlchemy本身的范围之内。 尽管通过将字符串传递给[Connection.execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.execute" \o "sqlalchemy.engine.Connection.execute)或通过使用[DDL](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.DDL" \o "sqlalchemy.schema.DDL)构造来发送ALTER语句和类似操作非常简单，但是通过使用模式自动执行与应用程序代码相关的数据库模式的维护是一个常见的做法 迁移工具。

There are two major migration tools available for SQLAlchemy:

SQLAlchemy有两个主要的迁移工具：

* [Alembic](http://alembic.zzzcomputing.com/) - Written by the author of SQLAlchemy, Alembic features a highly customizable environment and a minimalistic usage pattern, supporting such features as transactional DDL, automatic generation of "candidate" migrations, an "offline" mode which generates SQL scripts, and support for branch resolution.
* Alembic - 由SQLAlchemy的作者撰写，Alembic具有高度可定制的环境和简约的使用模式，支持事务DDL，自动生成"候选"迁移功能，生成SQL脚本的"离线"模式，并支持 分支决议。
* [SQLAlchemy-Migrate](https://github.com/openstack/sqlalchemy-migrate) - The original migration tool for SQLAlchemy, SQLAlchemy-Migrate is widely used and continues under active development. SQLAlchemy-Migrate includes features such as SQL script generation, ORM class generation, ORM model comparison, and extensive support for SQLite migrations.
* SQLAlchemy，SQLAlchemy-Migrate的原生迁移工具被广泛使用并继续积极发展。 SQLAlchemy-Migrate包括SQL脚本生成，ORM类生成，ORM模型比较以及对SQLite迁移的广泛支持等功能。

3.1.4 Specifying the Schema Name

Some databases support the concept of multiple schemas. A [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) can reference this by specifying the schema keyword argument:

一些数据库支持多种模式的概念。 表可以通过指定schema关键字参数来引用它：

financial\_info = Table('financial\_info', meta,

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

Column('value', String(100), nullable=**False**),

schema='remote\_banks')

Within the [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) collection, this table will be identified by the combination of financial\_info and remote\_banks. If another table called financial\_info is referenced without the remote\_banks schema, it will refer to a different [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table). [ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey) objects can specify references to columns in this table using the form remote\_banks.financial\_info.id.

在[MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData)集合中，该表将由financial\_info和remote\_banks的组合来标识。 如果没有remote\_banks模式引用另一个名为financial\_info的表，那么它将引用另一个不同的[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)。 [ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey)对象可以使用表单remote\_banks.financial\_info.id指定此表中对列的引用。

The schema argument should be used for any name qualifiers required, including Oracle's "owner" attribute and similar. It also can accommodate a dotted name for longer schemes:

schema参数应该用于所需的任何名称限定符，包括Oracle的"owner"属性和类似的。 它也可以容纳一个dotted name的更长的模式：

schema="dbo.scott"

### 3.1.5 Backend-Specific Options

[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) supports database-specific options. For example, MySQL has different table backend types, including "MyISAM" and "InnoDB". This can be expressed with [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) using mysql\_engine:

表支持特定于数据库的选项。 例如，MySQL具有不同的表后端类型，包括“MyISAM”和“InnoDB”。 这可以用表使用mysql\_engine表示：

addresses = Table('engine\_email\_addresses', meta,

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

Column('remote\_user\_id', Integer, ForeignKey(users.c.user\_id)),

Column('email\_address', String(20)),

mysql\_engine='InnoDB')

Other backends may support table-level options as well - these would be described in the individual documentation sections for each dialect.

其他后端还可以支持表级选项，这些将在每个方言的各个文档部分中进行描述。

3.1.6 Column, Table, MetaData API

sqlalchemy.schema.**BLANK\_SCHEMA**

Symbol indicating that a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) or [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) should have 'None' for its schema, even if the parent [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) has specified a schema.

表示表或序列对其模式应具有“无”，即使父元MetaData指定了模式。

**See also**

[MetaData.schema](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData.params.schema" \o "sqlalchemy.schema.MetaData)

[Table.schema](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.params.schema" \o "sqlalchemy.schema.Table)

[Sequence.schema](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence.params.schema" \o "sqlalchemy.schema.Sequence)

*New in version 1.0.14.*

*class*sqlalchemy.schema.**Column**(*\*args*, *\*\*kwargs*)

Bases: [sqlalchemy.schema.SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem), [sqlalchemy.sql.expression.ColumnClause](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnClause" \o "sqlalchemy.sql.expression.ColumnClause)

Represents a column in a database table.

基地：sqlalchemy.schema.SchemaItem，sqlalchemy.sql.expression.ColumnClause

表示数据库表中的列。

**\_\_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.

实现==运算符。

In a column context, produces the clause a = b. If the target is None, produces a IS NULL.

在列上下文中，生成子句a = b。 如果目标为无，则产生IS NULL。

**\_\_init\_\_**(*\*args*, *\*\*kwargs*)

Construct a new Column object.

|  |  |
| --- | --- |
| **Parameters:** | * ****name –****The name of this column as represented in the database. This argument may be the first positional argument, or specified via keyword.数据库中表示的此列的名称。 此参数可能是第一个位置参数，或通过关键字指定。   Names which contain no upper case characters will be treated as case insensitive names, and will not be quoted unless they are a reserved word. Names with any number of upper case characters will be quoted and sent exactly. Note that this behavior applies even for databases which standardize upper case names as case insensitive such as Oracle.  不包含大写字母的名称将被视为不区分大小写的名称，除非是保留字，否则不会引用。 具有任何大写字母数字的名称将被引用并准确发送。 请注意，这种行为甚至适用于将大写字母名称标识为不区分大小写的数据库，如Oracle。  The name field may be omitted at construction time and applied later, at any time before the Column is associated with a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table). This is to support convenient usage within the [declarative](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/api.html" \l "module-sqlalchemy.ext.declarative" \o "sqlalchemy.ext.declarative) extension.  名称字段在施工时可能会被省略，稍后应用在列与表关联之前的任何时间。 这是为了支持声明式扩展中的方便使用。   * ****type\_ –****The column's type, indicated using an instance which subclasses [TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine). If no arguments are required for the type, the class of the type can be sent as well, e.g.:列的类型，使用TypeEngine子类的实例指示。 如果类型不需要参数，也可以发送类型的类，例如：   *# use a type with arguments*  Column('data', String(50))  *# use no arguments*  Column('level', Integer)  The type argument may be the second positional argument or specified by keyword.  type参数可以是第二个位置参数或由关键字指定。  If the type is None or is omitted, it will first default to the special type [NullType](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.NullType" \o "sqlalchemy.types.NullType). If and when this [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) is made to refer to another column using[ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey) and/or [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint), the type of the remote-referenced column will be copied to this column as well, at the moment that the foreign key is resolved against that remote [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) object.  如果类型为None或省略，则首先将其默认为特殊类型NullType。 如果此列被用来引用另一列使用ForeignKey和/或ForeignKeyConstraint，那么远程引用的列的类型也将被复制到该列，而外键是针对该远程Column对象进行解析的。  *Changed in version 0.9.0:*Support for propagation of type to a [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) from its [ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey) object has been improved and should be more reliable and timely.  版本0.9.0更改：支持从ForeignKey对象向列传播类型已得到改进，应该更加可靠和及时。   * ****\*args**** – Additional positional arguments include various [SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem) derived constructs which will be applied as options to the column. These include instances of [Constraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.Constraint" \o "sqlalchemy.schema.Constraint), [ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey), [ColumnDefault](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.ColumnDefault" \o "sqlalchemy.schema.ColumnDefault), and [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence). In some cases an equivalent keyword argument is available such as server\_default, default and unique.其他位置参数包括将作为列的选项应用的各种SchemaItem派生结构。 这些包括约束，ForeignKey，ColumnDefault和Sequence的实例。 在某些情况下，可以使用等效的关键字参数，如server\_default，default和unique。 * ****autoincrement –****Set up "auto increment" semantics for an integer primary key column. The default value is the string "auto" which indicates that a single-column primary key that is of an INTEGER type with no stated client-side or python-side defaults should receive auto increment semantics automatically; all other varieties of primary key columns will not. This includes that [DDL](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-ddl) such as PostgreSQL SERIAL or MySQL AUTO\_INCREMENT will be emitted for this column during a table create, as well as that the column is assumed to generate new integer primary key values when an INSERT statement invokes which will be retrieved by the dialect.为整数主键列设置"自动递增"语义。 默认值是字符串"auto"，表示没有指定客户端或python侧默认值为INTEGER类型的单列主键应自动接收自动增量语义; 所有其他品种的主键列都不会。 这包括在表创建期间将为此列发出诸如PostgreSQL SERIAL或MySQL AUTO\_INCREMENT之类的[DDL](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-ddl)，以及当INSERT语句调用将由方言检索时，该列被假定为生成新的整数主键值。   The flag may be set to True to indicate that a column which is part of a composite (e.g. multi-column) primary key should have autoincrement semantics, though note that only one column within a primary key may have this setting. It can also be set to True to indicate autoincrement semantics on a column that has a client-side or server-side default configured, however note that not all dialects can accommodate all styles of default as an "autoincrement". It can also be set to False on a single-column primary key that has a datatype of INTEGER in order to disable auto increment semantics for that column  标志可能设置为True，表示作为复合（例如多列）主键的一部分的列应具有自动增量语义，但请注意，主键中只有一列可能具有此设置。 也可以将其设置为True，以便在已配置客户端或服务器端的列上指示自动增量语义，但请注意，并非所有方言都可以将所有样式的默认值都适用为"自动增量"。 也可以在数据类型为INTEGER的单列主键上设置为False，以禁用该列的自动增量语义  *Changed in version 1.1:*The autoincrement flag now defaults to "auto" which indicates autoincrement semantics by default for single-column integer primary keys only; for composite (multi-column) primary keys, autoincrement is never implicitly enabled; as always, autoincrement=True will allow for at most one of those columns to be an "autoincrement" column. autoincrement=True may also be set on a [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) that has an explicit client-side or server-side default, subject to limitations of the backend database and dialect.  版本1.1中更改：自动增量标志现在默认为"auto"，默认情况下仅显示单列整数主键的autoincrement语义; 对于复合（多列）主键，自动增量不会隐式启用; 一如以往，autoincrement = True将允许这些列中的最多一个作为"自动增量"列。 autoincrement = True也可以在具有明确的客户端或服务器端默认值的列上设置，但受到后端数据库和方言的限制。  The setting *only* has an effect for columns which are:  该设置仅对以下列的影响：   * + Integer derived (i.e. INT, SMALLINT, BIGINT).   + Part of the primary key   + Not referring to another column via [ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey), unless the value is specified as 'ignore\_fk':   *# turn on autoincrement for this column despite# the ForeignKey()*  Column('id', ForeignKey('other.id'),  primary\_key=**True**, autoincrement='ignore\_fk')  It is typically not desirable to have "autoincrement" enabled on a column that refers to another via foreign key, as such a column is required to refer to a value that originates from elsewhere.  在通过外键引用另一个列的列上启用“自动增量”通常是不希望的，因为这个列需要引用来自别处的值。  The setting has these two effects on columns that meet the above criteria:  该设置对满足上述标准的列有两个影响：   * + DDL issued for the column will include database-specific keywords intended to signify this column as an "autoincrement" column, such as AUTO INCREMENT on MySQL, SERIAL on PostgreSQL, and IDENTITY on MS-SQL. It does not issue AUTOINCREMENT for SQLite since this is a special SQLite flag that is not required for autoincrementing behavior.该列的DDL将包括旨在将此列表示为“自动增量”列的数据库特定关键字，例如MySQL上的AUTO INCREMENT，PostgreSQL上的SERIAL和MS-SQL上的IDENTITY。 它不会对SQLite发出AUTOINCREMENT，因为这是一个特殊的SQLite标志，对于自动增量行为不是必需的。   **See also**  [SQLite Auto Incrementing Behavior](http://docs.sqlalchemy.org/en/rel_1_1/dialects/sqlite.html" \l "sqlite-autoincrement)   * + The column will be considered to be available using an "autoincrement" method specific to the backend database, such as calling upon cursor.lastrowid, using RETURNING in an INSERT statement to get at a sequence-generated value, or using special functions such as "SELECT scope\_identity()". These methods are highly specific to the DBAPIs and databases in use and vary greatly, so care should be taken when associating autoincrement=True with a custom default generation function.该列将被认为是可用的，使用特定于后端数据库的“自动增量”方法，例如调用cursor.lastrowid，在INSERT语句中使用RETURNING获取序列生成的值，或使用特殊功能，例如“ SELECT scope\_identity() “。 这些方法对于正在使用的DBAPI和数据库具有高度的特殊性，并且变化很大，因此在将autoincrement = True与自定义默认生成函数相关联时应小心。 * ****default –****A scalar, Python callable, or [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) expression representing the *default value* for this column, which will be invoked upon insert if this column is otherwise not specified in the VALUES clause of the insert. This is a shortcut to using [ColumnDefault](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.ColumnDefault" \o "sqlalchemy.schema.ColumnDefault) as a positional argument; see that class for full detail on the structure of the argument.代表此列的默认值的标量，Python可调用或ColumnElement表达式，如果此列在插入的VALUES子句中未指定，则将在插入时调用该列。 这是使用ColumnDefault作为位置参数的快捷方式; 看到这个类的参数的结构的全部细节。   Contrast this argument to [Column.server\_default](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column.params.server_default" \o "sqlalchemy.schema.Column) which creates a default generator on the database side.将此参数与Column.server\_default对比，在数据库端创建一个默认生成器。  **See also**  [Column Insert/Update Defaults](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html)   * ****doc –**** optional String that can be used by the ORM or similar to document attributes. This attribute does not render SQL comments (a future attribute 'comment' will achieve that).可选字符串可以由ORM使用或类似于文档属性。 此属性不会呈现SQL注释（未来的属性“注释”将会实现）。 * ****key**** – An optional string identifier which will identify this Column object on the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table). When a key is provided, this is the only identifier referencing the Column within the application, including ORM attribute mapping; the name field is used only when rendering SQL.一个可选的字符串标识符，用于标识表上的Column对象。 当提供密钥时，这是唯一标识应用程序中列的标识符，包括ORM属性映射; 该名称字段仅在呈现SQL时使用。 * ****index**** – When True, indicates that the column is indexed. This is a shortcut for using a [Index](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.Index" \o "sqlalchemy.schema.Index) construct on the table. To specify indexes with explicit names or indexes that contain multiple columns, use the [Index](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.Index" \o "sqlalchemy.schema.Index) construct instead.当为True时，表示列已编入索引。 这是在表上使用Index构造的快捷方式。 要使用包含多个列的显式名称或索引来指定索引，请改用Index构造。 * ****info**** – Optional data dictionary which will be populated into the [SchemaItem.info](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem.info" \o "sqlalchemy.schema.SchemaItem.info) attribute of this object.可选的数据字典将被填充到该对象的SchemaItem.info属性中。 * ****nullable**** – When set to False, will cause the "NOT NULL" phrase to be added when generating DDL for the column. When True, will normally generate nothing (in SQL this defaults to "NULL"), except in some very specific backend-specific edge cases where "NULL" may render explicitly. Defaults to True unless [primary\_key](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column.params.primary_key" \o "sqlalchemy.schema.Column) is also True, in which case it defaults to False. This parameter is only used when issuing CREATE TABLE statements.当设置为False时，会在为列生成DDL时添加“NOT NULL”短语。 当为True时，通常不会生成任何内容（在SQL中，默认为“NULL”），除非在某些特定的后端特定边缘情况下，“NULL”可能会显式呈现。 默认为True，除非primary\_key也为True，在这种情况下，它默认为False。 此参数仅在发出CREATE TABLE语句时使用。 * ****onupdate –****A scalar, Python callable, or [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement) representing a default value to be applied to the column within UPDATE statements, which wil be invoked upon update if this column is not present in the SET clause of the update. This is a shortcut to using [ColumnDefault](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.ColumnDefault" \o "sqlalchemy.schema.ColumnDefault) as a positional argument with for\_update=True.标量，Python可调用或ClauseElement表示要应用于UPDATE语句中的列的默认值，如果该列不在更新的SET子句中，则更新后将被调用。 这是使用ColumnDefault作为for\_update = True的位置参数的快捷方式。   **See also**  [Column Insert/Update Defaults](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "metadata-defaults) - complete discussion of onupdate   * ****primary\_key**** – If True, marks this column as a primary key column. Multiple columns can have this flag set to specify composite primary keys. As an alternative, the primary key of a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) can be specified via an explicit [PrimaryKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.PrimaryKeyConstraint" \o "sqlalchemy.schema.PrimaryKeyConstraint) object.如果为True，则将此列标记为主键列。 多个列可以设置此标志来指定复合主键。 作为替代，可以通过显式的PrimaryKeyConstraint对象来指定表的主键。 * ****erver\_default**** –A [FetchedValue](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.FetchedValue" \o "sqlalchemy.schema.FetchedValue) instance, str, Unicode or [text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text) construct representing the DDL DEFAULT value for the column.FetchedValue实例，str，Unicode或text() 构造，表示列的DDL DEFAULT值。   String types will be emitted as-is, surrounded by single quotes:字符串类型将按原样排列，由单引号包围：  Column('x', Text, server\_default="val")  x TEXT DEFAULT 'val'  A [text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text) expression will be rendered as-is, without quotes:  text() 表达式将按原样呈现，不带引号：  Column('y', DateTime, server\_default=text('NOW()'))  y DATETIME DEFAULT NOW()  Strings and text() will be converted into a [DefaultClause](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.DefaultClause" \o "sqlalchemy.schema.DefaultClause) object upon initialization.  初始化时，字符串和文本() 将被转换为DefaultClause对象。  Use [FetchedValue](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.FetchedValue" \o "sqlalchemy.schema.FetchedValue) to indicate that an already-existing column will generate a default value on the database side which will be available to SQLAlchemy for post-fetch after inserts. This construct does not specify any DDL and the implementation is left to the database, such as via a trigger.  使用FetchedValue表示已经存在的列将在数据库端生成一个默认值，该值将在SQLAlchemy中插入后进行后置提取。 此构造没有指定任何DDL，并且实现将留给数据库，例如通过触发器。  **See also**  [Server Side Defaults](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "server-defaults) - complete discussion of server side defaults   * ****server\_onupdate –****A [FetchedValue](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.FetchedValue" \o "sqlalchemy.schema.FetchedValue) instance representing a database-side default generation function, such as a trigger. This indicates to SQLAlchemy that a newly generated value will be available after updates. This construct does not actually implement any kind of generation function within the database, which instead must be specified separately.表示数据库端默认生成函数（例如触发器）的FetchedValue实例。 这表明SQLAlchemy在更新后新生成的值将可用。 该构造实际上并不实现数据库中的任何类型的生成函数，而是必须单独指定。   **See also**  [Triggered Columns](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "triggered-columns)   * ****quote**** – Force quoting of this column's name on or off, corresponding to True or False. When left at its default of None, the column identifier will be quoted according to whether the name is case sensitive (identifiers with at least one upper case character are treated as case sensitive), or if it's a reserved word. This flag is only needed to force quoting of a reserved word which is not known by the SQLAlchemy dialect.强制引用此列的名称，打开或关闭，对应于True或False。 当默认为None时，列标识符将根据该名称是否区分大小写引用（至少有一个大写字符的标识符被视为区分大小写），或者如果它是保留字。 该标志只需要强制引用SQLAlchemy方言不知道的保留字。 * ****unique**** – When True, indicates that this column contains a unique constraint, or if index is True as well, indicates that the [Index](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.Index" \o "sqlalchemy.schema.Index) should be created with the unique flag. To specify multiple columns in the constraint/index or to specify an explicit name, use the [UniqueConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.UniqueConstraint" \o "sqlalchemy.schema.UniqueConstraint) or[Index](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.Index" \o "sqlalchemy.schema.Index) constructs explicitly.当为True时，表示此列包含唯一的约束，或者如果index也为True，则表示应使用唯一标记创建Index。 要在约束/索引中指定多个列或指定显式名称，请明确使用UniqueConstraint或InDex构造。 * ****system –****When True, indicates this is a "system" column, that is a column which is automatically made available by the database, and should not be included in the columns list for a CREATE TABLE statement.当为True时，表示这是一个“系统”列，它是由数据库自动使用的列，不应包含在CREATE TABLE语句的列列表中。   For more elaborate scenarios where columns should be conditionally rendered differently on different backends, consider custom compilation rules for [CreateColumn](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.CreateColumn" \o "sqlalchemy.schema.CreateColumn).  对于需要在不同后端有条件地对列进行不同渲染的更精细的场景，请考虑CreateColumn的自定义编译规则。  *New in version 0.8.3:*Added the system=True parameter to [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column). |

**\_\_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.

**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.*

**anon\_label**

*inherited from the* [anon\_label](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement.anon_label" \o "sqlalchemy.sql.expression.ColumnElement.anon_label) *attribute of* [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement)

provides a constant 'anonymous label' for this ColumnElement.

为此ColumnElement提供了一个不变的“匿名标签”。

This is a label() expression which will be named at compile time. The same label() is returned each time anon\_label is called so that expressions can reference anon\_label multiple times, producing the same label name at compile time.

这是一个在编译时被命名的label() 表达式。 每次调用anon\_label时都会返回相同的label() ，以便表达式可以多次引用anon\_label，在编译时生成相同的标签名称。

the compiler uses this function automatically at compile time for expressions that are known to be 'unnamed' like binary expressions and function calls.

编译器在编译时自动使用这个函数，这个表达式被称为“未命名”，就像二进制表达式和函数调用一样。

**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.

**cast**(*type\_*)

*inherited from the* [cast()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement.cast" \o "sqlalchemy.sql.expression.ColumnElement.cast) *method of* [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement)

Produce a type cast, i.e. CAST(<expression> AS <type>).

This is a shortcut to the [cast()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.cast" \o "sqlalchemy.sql.expression.cast) function.

*New in version 1.0.7.*

**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.

**compare**(*other*, *use\_proxies=False*, *equivalents=None*, *\*\*kw*)

*inherited from the* [compare()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement.compare" \o "sqlalchemy.sql.expression.ColumnElement.compare) *method of* [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement)

Compare this ColumnElement to another.

Special arguments understood:

|  |  |
| --- | --- |
| **Parameters:** | * ****use\_proxies**** – when True, consider two columns that share a common base column as equivalent (i.e. shares\_lineage()) * ****equivalents**** – a dictionary of columns as keys mapped to sets of columns. If the given "other" column is present in this dictionary, if any of the columns in the corresponding set() pass the comparison test, the result is True. This is used to expand the comparison to other columns that may be known to be equivalent to this one via foreign key or other criterion. |

**compile**(*bind=None*, *dialect=None*, *\*\*kw*)

*inherited from the* [compile()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.compile" \o "sqlalchemy.sql.expression.ClauseElement.compile) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Compile this SQL expression.

The return value is a [Compiled](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Compiled" \o "sqlalchemy.engine.interfaces.Compiled) object. Calling str() or unicode() on the returned value will yield a string representation of the result. The[Compiled](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Compiled" \o "sqlalchemy.engine.interfaces.Compiled) object also can return a dictionary of bind parameter names and values using the params accessor.

|  |  |
| --- | --- |
| **Parameters:** | * ****bind**** – An Engine or Connection from which a Compiled will be acquired. This argument takes precedence over this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)'s bound engine, if any. * ****column\_keys**** – Used for INSERT and UPDATE statements, a list of column names which should be present in the VALUES clause of the compiled statement. If None, all columns from the target table object are rendered. * ****dialect**** – A Dialect instance from which a Compiled will be acquired. This argument takes precedence over the bind argument as well as this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)'s bound engine, if any. * ****inline**** – Used for INSERT statements, for a dialect which does not support inline retrieval of newly generated primary key columns, will force the expression used to create the new primary key value to be rendered inline within the INSERT statement's VALUES clause. This typically refers to Sequence execution but may also refer to any server-side default generation function associated with a primary key Column. * ****compile\_kwargs**** –   optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the literal\_binds flag through:  **from** **sqlalchemy.sql** **import** table, column, select  t = table('t', column('x'))  s = select([t]).where(t.c.x == 5)  print s.compile(compile\_kwargs={"literal\_binds": **True**})  *New in version 0.9.0.* |

**See also**

[How do I render SQL expressions as strings, possibly with bound parameters inlined?](http://docs.sqlalchemy.org/en/rel_1_1/faq/sqlexpressions.html" \l "faq-sql-expression-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>%'

**copy**(*\*\*kw*)

Create a copy of this Column, unitialized.

This is used in Table.tometadata.

**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>'

**expression**

*inherited from the* [expression](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement.expression" \o "sqlalchemy.sql.expression.ColumnElement.expression) *attribute of* [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement)

Return a column expression.

Part of the inspection interface; returns self.

**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**

*inherited from the* [info](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem.info" \o "sqlalchemy.schema.SchemaItem.info) *attribute of* [SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem)

Info dictionary associated with the object, allowing user-defined data to be associated with this [SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem).

The dictionary is automatically generated when first accessed. It can also be specified in the constructor of some objects, such as [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) and [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column).

**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.*

**label**(*name*)

*inherited from 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) *method of* [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement)

Produce a column label, i.e. <columnname> AS <name>.

This is a shortcut to the [label()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.label" \o "sqlalchemy.sql.expression.label) function.

if 'name' is None, an anonymous label name will be generated.

**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. * ****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)

**quote**

*inherited from the* [quote](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem.quote" \o "sqlalchemy.schema.SchemaItem.quote) *attribute of* [SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem)

Return the value of the quote flag passed to this schema object, for those schema items which have a name field.

*Deprecated since version 0.9:*Use <obj>.name.quote

**references**(*column*)

Return True if this Column references the given column via foreign key.

**shares\_lineage**(*othercolumn*)

*inherited from the* [shares\_lineage()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement.shares_lineage" \o "sqlalchemy.sql.expression.ColumnElement.shares_lineage) *method of* [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement)

Return True if the given [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) has a common ancestor to this [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement).

**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.schema.**MetaData**(*bind=None*, *reflect=False*, *schema=None*, *quote\_schema=None*, *naming\_convention=immutabledict({'ix': 'ix\_%(column\_0\_label)s'})*, *info=None*)

Bases: [sqlalchemy.schema.SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem)

A collection of [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects and their associated schema constructs.

Table对象及其关联模式结构的集合。

Holds a collection of [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects as well as an optional binding to 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). If bound, the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects in the collection and their columns may participate in implicit SQL execution.

持有Table对象的集合以及对引擎或连接的可选绑定。 如果绑定，则集合中的Table对象及其列可能会参与隐式SQL执行。

The [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects themselves are stored in the [MetaData.tables](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData.tables" \o "sqlalchemy.schema.MetaData.tables) dictionary.

Table对象本身存储在MetaData.tables字典中。

[MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) is a thread-safe object for read operations. Construction of new tables within a single [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) object, either explicitly or via reflection, may not be completely thread-safe.

MetaData是读操作的线程安全对象。 在单个MetaData对象中，通过显式或通过反射来构建新表可能不是完全线程安全的。

**See also**

[Describing Databases with MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "metadata-describing) - Introduction to database metadata

**\_\_init\_\_**(*bind=None*, *reflect=False*, *schema=None*, *quote\_schema=None*, *naming\_convention=immutabledict({'ix': 'ix\_%(column\_0\_label)s'})*, *info=None*)

Create a new MetaData object.

|  |  |
| --- | --- |
| **Parameters:** | * ****bind**** – An Engine or Connection to bind to. May also be a string or URL instance, these are passed to create\_engine() and this MetaData will be bound to the resulting engine. * ****~~reflect –~~****~~Optional, automatically load all tables from the bound database. Defaults to False. bind is required when this option is set.~~   *~~Deprecated since version 0.8:~~*Please use 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.   * ****schema –****The default schema to use for the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table), [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence), and potentially other objects associated with this [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData). Defaults to None.用于与此MetaData相关联的表，序列和潜在其他对象的默认模式。 默认为无。   When this value is set, any [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) or [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) which specifies None for the schema parameter will instead have this schema name defined. To build a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) or [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) that still has None for the schema even when this parameter is present, use the [BLANK\_SCHEMA](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.sqlalchemy.schema.BLANK_SCHEMA" \o "sqlalchemy.schema.sqlalchemy.schema.BLANK_SCHEMA) symbol.  当设置此值时，将为schema参数指定None的任何表或序列将定义此模式名称。 要构建表或序列，即使存在此参数，该模式仍然具有“否”，请使用BLANK\_SCHEMA符号。  **Note**  As refered above, the [MetaData.schema](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData.params.schema" \o "sqlalchemy.schema.MetaData) parameter only refers to the ****default value**** that will be applied to the [Table.schema](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.params.schema" \o "sqlalchemy.schema.Table) parameter of an incoming [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)object. It does not refer to how the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) is catalogued within the [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData), which remains consistent vs. a [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) collection that does not define this parameter. The [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 still be keyed based on its schema-qualified name, e.g.my\_metadata.tables["some\_schema.my\_table"].  如上所述，MetaData.schema参数仅指将应用于传入Table对象的Table.schema参数的默认值。 它不是指MetaData中如何对表进行编目，它与MetaData集合保持一致，但不定义此参数。 MetaData中的表仍将根据其模式限定名称（如my\_metadata.tables [“some\_schema.my\_table”]）键入。  The current behavior of the [ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey) object is to circumvent this restriction, where it can locate a table given the table name alone, where the schema will be assumed to be present from this value as specified on the owning [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) collection. However, this implies that a table qualified with BLANK\_SCHEMA cannot currently be referred to by string name from [ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey). Other parts of SQLAlchemy such as Declarative may not have similar behaviors built in, however may do so in a future release, along with a consistent method of referring to a table in BLANK\_SCHEMA.  ForeignKey对象的当前行为是绕过这个限制，在那里它可以定位一个给定表名的表，其中假定模式从这个值出现，该属性在所属的MetaData集合中指定。 但是，这意味着使用BLANK\_SCHEMA限定的表格目前无法通过ForeignKey的字符串名称引用。 SQLAlchemy的其他部分（如Declarative）可能不具有内置的类似行为，但在将来的版本中可能会这样做，以及引用BLANK\_SCHEMA中的表的一致方法。  **See also**  [Table.schema](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.params.schema" \o "sqlalchemy.schema.Table)  [Sequence.schema](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence.params.schema" \o "sqlalchemy.schema.Sequence)   * ****quote\_schema**** – Sets the quote\_schema flag for those [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table), [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence), and other objects which make usage of the local schema name.为那些使用本地架构名称的表，序列和其他对象设置quote\_schema标志。 * ****info –****Optional data dictionary which will be populated into the [SchemaItem.info](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem.info" \o "sqlalchemy.schema.SchemaItem.info) attribute of this object.可选的数据字典将被填充到该对象的SchemaItem.info属性中。   *New in version 1.0.0.*   * ****naming\_convention –****a dictionary referring to values which will establish default naming conventions for [Constraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.Constraint" \o "sqlalchemy.schema.Constraint) and [Index](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.Index" \o "sqlalchemy.schema.Index) objects, for those objects which are not given a name explicitly.一个引用将为Constraint和Index对象建立默认命名约定的值的字典，用于那些没有明确赋予名称的对象。   The keys of this dictionary may be:   * + a constraint or Index class, e.g. the [UniqueConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.UniqueConstraint" \o "sqlalchemy.schema.UniqueConstraint), [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint) class, the [Index](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.Index" \o "sqlalchemy.schema.Index) class   + a string mnemonic for one of the known constraint classes; "fk", "pk", "ix", "ck", "uq" for foreign key, primary key, index, check, and unique constraint, respectively.   + the string name of a user-defined "token" that can be used to define new naming tokens.   The values associated with each "constraint class" or "constraint mnemonic" key are string naming templates, such as "uq\_%(table\_name)s\_%(column\_0\_name)s", which describe how the name should be composed. The values associated with user-defined "token" keys should be callables of the form fn(constraint, table), which accepts the constraint/index object and [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) as arguments, returning a string result.  与每个“约束类”或“约束助记符”密钥相关联的值是字符串命名模板，例如“uq \_％（table\_name）s \_％（column\_0\_name）s”，它们描述如何组成名称。 与用户定义的“令牌”密钥相关联的值应为fn（constraint，table）形式的可调用函数，它接受constraint / index对象和Table作为参数，返回字符串结果。  The built-in names are as follows, some of which may only be available for certain types of constraint:  内置的名称如下，其中一些可能仅适用于某些类型的约束：   * + %(table\_name)s - the name of the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object associated with the constraint.   + %(referred\_table\_name)s - the name of the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object associated with the referencing target of a[ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint).   + %(column\_0\_name)s - the name of the [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) at index position "0" within the constraint.   + %(column\_0\_label)s - the label of the [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) at index position "0", e.g. Column.label   + %(column\_0\_key)s - the key of the [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) at index position "0", e.g. Column.key   + %(referred\_column\_0\_name)s - the name of a [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) at index position "0" referenced by a [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint).   + %(constraint\_name)s - a special key that refers to the existing name given to the constraint. When this key is present, the [Constraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.Constraint" \o "sqlalchemy.schema.Constraint) object's existing name will be replaced with one that is composed from template string that uses this token. When this token is present, it is required that the [Constraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.Constraint" \o "sqlalchemy.schema.Constraint) is given an explicit name ahead of time.   + user-defined: any additional token may be implemented by passing it along with a fn(constraint, table) callable to the naming\_convention dictionary.   *New in version 0.9.2.*  **See also**  [Configuring Constraint Naming Conventions](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "constraint-naming-conventions) - for detailed usage examples. |

**append\_ddl\_listener**(*event\_name*, *listener*)

Append a DDL event listener to this MetaData.

*Deprecated since version 0.7:*See [DDLEvents](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.DDLEvents" \o "sqlalchemy.events.DDLEvents).

**bind**

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) to which this [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) is bound.

Typically, a [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) is assigned to this attribute so that "implicit execution" may be used, or alternatively as a means of providing engine binding information to an ORM [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) object:

engine = create\_engine("someurl://")metadata.bind = engine

**See also**

[Connectionless Execution, Implicit Execution](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "dbengine-implicit) - background on "bound metadata"

**clear**()

Clear all Table objects from this MetaData.

**create\_all**(*bind=None*, *tables=None*, *checkfirst=True*)

Create all tables stored in this metadata.

Conditional by default, will not attempt to recreate tables already present in the target database.

|  |  |
| --- | --- |
| **Parameters:** | * ****bind**** – A [Connectable](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connectable" \o "sqlalchemy.engine.Connectable) used to access the database; if None, uses the existing bind on this MetaData, if any. * ****tables**** – Optional list of Table objects, which is a subset of the total tables in the MetaData (others are ignored). * ****checkfirst**** – Defaults to True, don't issue CREATEs for tables already present in the target database. |

**drop\_all**(*bind=None*, *tables=None*, *checkfirst=True*)

Drop all tables stored in this metadata.

Conditional by default, will not attempt to drop tables not present in the target database.

|  |  |
| --- | --- |
| **Parameters:** | * ****bind**** – A [Connectable](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connectable" \o "sqlalchemy.engine.Connectable) used to access the database; if None, uses the existing bind on this MetaData, if any. * ****tables**** – Optional list of Table objects, which is a subset of the total tables in the MetaData (others are ignored). * ****checkfirst**** – Defaults to True, only issue DROPs for tables confirmed to be present in the target database. |

**is\_bound**()

True if this MetaData is bound to an Engine or Connection.

**reflect**(*bind=None*, *schema=None*, *views=False*, *only=None*, *extend\_existing=False*, *autoload\_replace=True*, *\*\*dialect\_kwargs*)

Load all available table definitions from the database.

Automatically creates Table entries in this MetaData for any table available in the database but not yet present in the MetaData. May be called multiple times to pick up tables recently added to the database, however no special action is taken if a table in this MetaData no longer exists in the database.

|  |  |
| --- | --- |
| **Parameters:** | * ****bind**** – A [Connectable](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connectable" \o "sqlalchemy.engine.Connectable) used to access the database; if None, uses the existing bind on this MetaData, if any. * ****schema**** – Optional, query and reflect tables from an alterate schema. If None, the schema associated with this [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) is used, if any. * ****views –**** If True, also reflect views. * ****only –****Optional. Load only a sub-set of available named tables. May be specified as a sequence of names or a callable.   If a sequence of names is provided, only those tables will be reflected. An error is raised if a table is requested but not available. Named tables already present in this MetaData are ignored.  If a callable is provided, it will be used as a boolean predicate to filter the list of potential table names. The callable is called with a table name and this MetaData instance as positional arguments and should return a true value for any table to reflect.   * ****extend\_existing –****Passed along to each [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) as [Table.extend\_existing](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.params.extend_existing" \o "sqlalchemy.schema.Table).   *New in version 0.9.1.*   * ****autoload\_replace –****Passed along to each [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) as [Table.autoload\_replace](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.params.autoload_replace" \o "sqlalchemy.schema.Table).   *New in version 0.9.1.*   * ****\*\*dialect\_kwargs –****Additional keyword arguments not mentioned above are dialect specific, and passed in the form <dialectname>\_<argname>. See the documentation regarding an individual dialect at [Dialects](http://docs.sqlalchemy.org/en/rel_1_1/dialects/index.html) for detail on documented arguments.   *New in version 0.9.2:*- Added [MetaData.reflect.\*\*dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData.reflect.params.**dialect_kwargs" \o "sqlalchemy.schema.MetaData.reflect) to support dialect-level reflection options for all [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects reflected. |

**remove**(*table*)

Remove the given Table object from this MetaData.

**sorted\_tables**

Returns a list of [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects sorted in order of foreign key dependency.

The sorting will place [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects that have dependencies first, before the dependencies themselves, representing the order in which they can be created. To get the order in which the tables would be dropped, use the reversed() Python built-in.

**Warning**

The [sorted\_tables](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData.sorted_tables" \o "sqlalchemy.schema.MetaData.sorted_tables) accessor cannot by itself accommodate automatic resolution of dependency cycles between tables, which are usually caused by mutually dependent foreign key constraints. To resolve these cycles, either the [ForeignKeyConstraint.use\_alter](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint.params.use_alter" \o "sqlalchemy.schema.ForeignKeyConstraint) parameter may be appled to those constraints, or use the [schema.sort\_tables\_and\_constraints()](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.sort_tables_and_constraints" \o "sqlalchemy.schema.sort_tables_and_constraints) function which will break out foreign key constraints involved in cycles separately.

**See also**

[schema.sort\_tables()](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.sort_tables" \o "sqlalchemy.schema.sort_tables)

[schema.sort\_tables\_and\_constraints()](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.sort_tables_and_constraints" \o "sqlalchemy.schema.sort_tables_and_constraints)

[MetaData.tables](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData.tables" \o "sqlalchemy.schema.MetaData.tables)

[Inspector.get\_table\_names()](http://docs.sqlalchemy.org/en/rel_1_1/core/reflection.html" \l "sqlalchemy.engine.reflection.Inspector.get_table_names" \o "sqlalchemy.engine.reflection.Inspector.get_table_names)

[Inspector.get\_sorted\_table\_and\_fkc\_names()](http://docs.sqlalchemy.org/en/rel_1_1/core/reflection.html" \l "sqlalchemy.engine.reflection.Inspector.get_sorted_table_and_fkc_names" \o "sqlalchemy.engine.reflection.Inspector.get_sorted_table_and_fkc_names)

**tables***= None*

A dictionary of [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects keyed to their name or "table key".

The exact key is that determined by the [Table.key](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.key" \o "sqlalchemy.schema.Table.key) attribute; for a table with no Table.schema attribute, this is the same as Table.name. For a table with a schema, it is typically of the form schemaname.tablename.

**See also**

[MetaData.sorted\_tables](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData.sorted_tables" \o "sqlalchemy.schema.MetaData.sorted_tables)

*class*sqlalchemy.schema.**SchemaItem**

Bases: sqlalchemy.sql.expression.SchemaEventTarget, sqlalchemy.sql.visitors.Visitable

Base class for items that define a database schema.

**get\_children**(*\*\*kwargs*)

used to allow SchemaVisitor access

**info**

Info dictionary associated with the object, allowing user-defined data to be associated with this [SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem).

The dictionary is automatically generated when first accessed. It can also be specified in the constructor of some objects, such as [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) and [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column).

**quote**

Return the value of the quote flag passed to this schema object, for those schema items which have a name field.

*Deprecated since version 0.9:*Use <obj>.name.quote

*class*sqlalchemy.schema.**Table**(*\*args*, *\*\*kw*)

Bases: [sqlalchemy.sql.base.DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs), [sqlalchemy.schema.SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem), [sqlalchemy.sql.expression.TableClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableClause" \o "sqlalchemy.sql.expression.TableClause)

Represent a table in a database.

e.g.:

mytable = Table("mytable", metadata,

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

Column('value', String(50))

)

The [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object constructs a unique instance of itself based on its name and optional schema name within the given [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) object. Calling the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)constructor with the same name and same [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) argument a second time will return the *same* [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object - in this way the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) constructor acts as a registry function.

**See also**

[Describing Databases with MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "metadata-describing) - Introduction to database metadata

Constructor arguments are as follows:

|  |  |
| --- | --- |
| **Parameters:** | * ****name**** –   The name of this table as represented in the database.  The table name, along with the value of the schema parameter, forms a key which uniquely identifies this [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) within the owning [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) collection. Additional calls to [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) with the same name, metadata, and schema name will return the same [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object.  Names which contain no upper case characters will be treated as case insensitive names, and will not be quoted unless they are a reserved word or contain special characters. A name with any number of upper case characters is considered to be case sensitive, and will be sent as quoted.  To enable unconditional quoting for the table name, specify the flag quote=True to the constructor, or use the [quoted\_name](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.elements.quoted_name" \o "sqlalchemy.sql.elements.quoted_name) construct to specify the name.   * ****metadata**** – a [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) object which will contain this table. The metadata is used as a point of association of this table with other tables which are referenced via foreign key. It also may be used to associate this table with a particular [Connectable](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connectable" \o "sqlalchemy.engine.Connectable). * ****\*args**** – Additional positional arguments are used primarily to add the list of [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects contained within this table. Similar to the style of a CREATE TABLE statement, other [SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem) constructs may be added here, including [PrimaryKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.PrimaryKeyConstraint" \o "sqlalchemy.schema.PrimaryKeyConstraint), and [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint). * ****autoload**** –   Defaults to False, unless [Table.autoload\_with](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.params.autoload_with" \o "sqlalchemy.schema.Table) is set in which case it defaults to True; [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects for this table should be reflected from the database, possibly augmenting or replacing existing [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects that were explicitly specified.  *Changed in version 1.0.0:*setting the [Table.autoload\_with](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.params.autoload_with" \o "sqlalchemy.schema.Table) parameter implies that [Table.autoload](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.params.autoload" \o "sqlalchemy.schema.Table) will default to True.  **See also**  [Reflecting Database Objects](http://docs.sqlalchemy.org/en/rel_1_1/core/reflection.html)   * ****autoload\_replace**** –   Defaults to True; when using [Table.autoload](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.params.autoload" \o "sqlalchemy.schema.Table) in conjunction with [Table.extend\_existing](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.params.extend_existing" \o "sqlalchemy.schema.Table), indicates 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 already-existing [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object should be replaced with columns of the same name retrieved from the autoload process. When False, columns already present under existing names will be omitted from the reflection process.  Note that this setting does not impact [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects specified programmatically within the call to [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) that also is autoloading; those [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects will always replace existing columns of the same name when [Table.extend\_existing](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.params.extend_existing" \o "sqlalchemy.schema.Table) is True.  *New in version 0.7.5.*  **See also**  [Table.autoload](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.params.autoload" \o "sqlalchemy.schema.Table)  [Table.extend\_existing](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.params.extend_existing" \o "sqlalchemy.schema.Table)   * ****autoload\_with**** –   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) object with which this [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object will be reflected; when set to a non-None value, it implies that [Table.autoload](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.params.autoload" \o "sqlalchemy.schema.Table) is True. If left unset, but [Table.autoload](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.params.autoload" \o "sqlalchemy.schema.Table) is explicitly set to True, an autoload operation will attempt to proceed by locating 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)bound to the underlying [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) object.  **See also**  [Table.autoload](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.params.autoload" \o "sqlalchemy.schema.Table)   * ****extend\_existing**** –   When True, indicates that if this [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) is already present in the given [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData), apply further arguments within the constructor to the existing [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table).  If [Table.extend\_existing](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.params.extend_existing" \o "sqlalchemy.schema.Table) or [Table.keep\_existing](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.params.keep_existing" \o "sqlalchemy.schema.Table) are not set, and the given name of the new [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) refers to a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) that is already present in the target [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) collection, and this [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) specifies additional columns or other constructs or flags that modify the table's state, an error is raised. The purpose of these two mutually-exclusive flags is to specify what action should be taken when a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) is specified that matches an existing [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table), yet specifies additional constructs.  [Table.extend\_existing](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.params.extend_existing" \o "sqlalchemy.schema.Table) will also work in conjunction with [Table.autoload](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.params.autoload" \o "sqlalchemy.schema.Table) to run a new reflection operation against the database, even if a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)of the same name is already present in the target [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData); newly reflected [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects and other options will be added into the state of the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table), potentially overwriting existing columns and options of the same name.  *Changed in version 0.7.4:*[Table.extend\_existing](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.params.extend_existing" \o "sqlalchemy.schema.Table) will invoke a new reflection operation when combined with [Table.autoload](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.params.autoload" \o "sqlalchemy.schema.Table) set to True.  As is always the case with [Table.autoload](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.params.autoload" \o "sqlalchemy.schema.Table), [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects can be specified in the same [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) constructor, which will take precedence. Below, the existing table mytable will be augmented with [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects both reflected from the database, as well as the given [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) named "y":  Table("mytable", metadata,  Column('y', Integer),  extend\_existing=**True**,  autoload=**True**,  autoload\_with=engine  )  **See also**  [Table.autoload](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.params.autoload" \o "sqlalchemy.schema.Table)  [Table.autoload\_replace](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.params.autoload_replace" \o "sqlalchemy.schema.Table)  [Table.keep\_existing](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.params.keep_existing" \o "sqlalchemy.schema.Table)   * ****implicit\_returning**** – True by default - indicates that RETURNING can be used by default to fetch newly inserted primary key values, for backends which support this. Note that create\_engine() also provides an implicit\_returning flag. * ****include\_columns**** – A list of strings indicating a subset of columns to be loaded via the autoload operation; table columns who aren't present in this list will not be represented on the resulting Table object. Defaults to None which indicates all columns should be reflected. * ****info**** – Optional data dictionary which will be populated into the [SchemaItem.info](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem.info" \o "sqlalchemy.schema.SchemaItem.info) attribute of this object. * ****keep\_existing –****When True, indicates that if this Table is already present in the given [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData), ignore further arguments within the constructor to the existing [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table), and return the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object as originally created. This is to allow a function that wishes to define a new [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) on first call, but on subsequent calls will return the same [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table), without any of the declarations (particularly constraints) being applied a second time.   If [Table.extend\_existing](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.params.extend_existing" \o "sqlalchemy.schema.Table) or [Table.keep\_existing](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.params.keep_existing" \o "sqlalchemy.schema.Table) are not set, and the given name of the new [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) refers to a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) that is already present in the target [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) collection, and this [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) specifies additional columns or other constructs or flags that modify the table's state, an error is raised. The purpose of these two mutually-exclusive flags is to specify what action should be taken when a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) is specified that matches an existing [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table), yet specifies additional constructs.  **See also**  [Table.extend\_existing](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.params.extend_existing" \o "sqlalchemy.schema.Table)   * ****listeners –****A list of tuples of the form (<eventname>, <fn>) which will be passed to [event.listen()](http://docs.sqlalchemy.org/en/rel_1_1/core/event.html" \l "sqlalchemy.event.listen" \o "sqlalchemy.event.listen) upon construction. This alternate hook to [event.listen()](http://docs.sqlalchemy.org/en/rel_1_1/core/event.html" \l "sqlalchemy.event.listen" \o "sqlalchemy.event.listen) allows the establishment of a listener function specific to this [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) before the "autoload" process begins. Particularly useful for 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:   **def** listen\_for\_reflect(table, column\_info):  "handle the column reflection event"  *# ...*  t = Table(  'sometable',  autoload=**True**,  listeners=[  ('column\_reflect', listen\_for\_reflect)  ])   * ****mustexist**** – When True, indicates that this Table must already be present in the given [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) collection, else an exception is raised. * ****prefixes**** – A list of strings to insert after CREATE in the CREATE TABLE statement. They will be separated by spaces. * ****quote**** – Force quoting of this table's name on or off, corresponding to True or False. When left at its default of None, the column identifier will be quoted according to whether the name is case sensitive (identifiers with at least one upper case character are treated as case sensitive), or if it's a reserved word. This flag is only needed to force quoting of a reserved word which is not known by the SQLAlchemy dialect. * ****quote\_schema**** – same as 'quote' but applies to the schema identifier. * ****schema –****The schema name for this table, which is required if the table resides in a schema other than the default selected schema for the engine's database connection. Defaults to None.   If the owning [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) of this [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) specifies its own [MetaData.schema](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData.params.schema" \o "sqlalchemy.schema.MetaData) parameter, then that schema name will be applied to this [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) if the schema parameter here is set to None. To set a blank schema name on a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) that would otherwise use the schema set on the owning [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData), specify the special symbol [BLANK\_SCHEMA](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.sqlalchemy.schema.BLANK_SCHEMA" \o "sqlalchemy.schema.sqlalchemy.schema.BLANK_SCHEMA).  *New in version 1.0.14:*Added the [BLANK\_SCHEMA](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.sqlalchemy.schema.BLANK_SCHEMA" \o "sqlalchemy.schema.sqlalchemy.schema.BLANK_SCHEMA) symbol to allow a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) to have a blank schema name even when the parent [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) specifies [MetaData.schema](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData.params.schema" \o "sqlalchemy.schema.MetaData).  The quoting rules for the schema name are the same as those for the name parameter, in that quoting is applied for reserved words or case-sensitive names; to enable unconditional quoting for the schema name, specify the flag quote\_schema=True to the constructor, or use the [quoted\_name](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.elements.quoted_name" \o "sqlalchemy.sql.elements.quoted_name) construct to specify the name.   * ****useexisting**** – Deprecated. Use [Table.extend\_existing](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.params.extend_existing" \o "sqlalchemy.schema.Table). * ****\*\*kw**** – Additional keyword arguments not mentioned above are dialect specific, and passed in the form <dialectname>\_<argname>. See the documentation regarding an individual dialect at [Dialects](http://docs.sqlalchemy.org/en/rel_1_1/dialects/index.html) for detail on documented arguments. |

**\_\_init\_\_**(*\*args*, *\*\*kw*)

Constructor for [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table).

This method is a no-op. See the top-level documentation for [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) for constructor arguments.

**add\_is\_dependent\_on**(*table*)

Add a 'dependency' for this Table.

This is another Table object which must be created first before this one can, or dropped after this one.

Usually, dependencies between tables are determined via ForeignKey objects. However, for other situations that create dependencies outside of foreign keys (rules, inheriting), this method can manually establish such a link.

**alias**(*name=None*, *flat=False*)

*inherited from the* [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 of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

return an alias of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

This is shorthand for calling:

**from** **sqlalchemy** **import** alias

a = alias(self, name=name)

See [alias()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.alias" \o "sqlalchemy.sql.expression.alias) for details.

**append\_column**(*column*)

Append a [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) to this [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table).

The "key" of the newly added [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column), i.e. the value of its .key attribute, will then be available in the .c collection of this [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table), and the column definition will be included in any CREATE TABLE, SELECT, UPDATE, etc. statements generated from this [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) construct.

Note that this does ****not**** change the definition of the table as it exists within any underlying database, assuming that table has already been created in the database. Relational databases support the addition of columns to existing tables using the SQL ALTER command, which would need to be emitted for an already-existing table that doesn't contain the newly added column.

**append\_constraint**(*constraint*)

Append a [Constraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.Constraint" \o "sqlalchemy.schema.Constraint) to this [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table).

This has the effect of the constraint being included in any future CREATE TABLE statement, assuming specific DDL creation events have not been associated with the given [Constraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.Constraint" \o "sqlalchemy.schema.Constraint) object.

Note that this does ****not**** produce the constraint within the relational database automatically, for a table that already exists in the database. To add a constraint to an existing relational database table, the SQL ALTER command must be used. SQLAlchemy also provides the [AddConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.AddConstraint" \o "sqlalchemy.schema.AddConstraint) construct which can produce this SQL when invoked as an executable clause.

**append\_ddl\_listener**(*event\_name*, *listener*)

Append a DDL event listener to this Table.

*Deprecated since version 0.7:*See [DDLEvents](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.DDLEvents" \o "sqlalchemy.events.DDLEvents).

**argument\_for**(*dialect\_name*, *argument\_name*, *default*)

*inherited from the* [argument\_for()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.argument_for" \o "sqlalchemy.sql.base.DialectKWArgs.argument_for) *method of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

Add a new kind of dialect-specific keyword argument for this class.

E.g.:

Index.argument\_for("mydialect", "length", **None**)

some\_index = Index('a', 'b', mydialect\_length=5)

The [DialectKWArgs.argument\_for()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.argument_for" \o "sqlalchemy.sql.base.DialectKWArgs.argument_for) method is a per-argument way adding extra arguments to the[DefaultDialect.construct\_arguments](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.default.DefaultDialect.construct_arguments" \o "sqlalchemy.engine.default.DefaultDialect.construct_arguments) dictionary. This dictionary provides a list of argument names accepted by various schema-level constructs on behalf of a dialect.

New dialects should typically specify this dictionary all at once as a data member of the dialect class. The use case for ad-hoc addition of argument names is typically for end-user code that is also using a custom compilation scheme which consumes the additional arguments.

|  |  |
| --- | --- |
| **Parameters:** | * ****dialect\_name**** – name of a dialect. The dialect must be locatable, else a [NoSuchModuleError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.NoSuchModuleError" \o "sqlalchemy.exc.NoSuchModuleError) is raised. The dialect must also include an existing[DefaultDialect.construct\_arguments](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.default.DefaultDialect.construct_arguments" \o "sqlalchemy.engine.default.DefaultDialect.construct_arguments) collection, indicating that it participates in the keyword-argument validation and default system, else [ArgumentError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.ArgumentError" \o "sqlalchemy.exc.ArgumentError) is raised. If the dialect does not include this collection, then any keyword argument can be specified on behalf of this dialect already. All dialects packaged within SQLAlchemy include this collection, however for third party dialects, support may vary. * ****argument\_name**** – name of the parameter. * ****default**** – default value of the parameter. |

*New in version 0.9.4.*

**bind**

Return the connectable associated with this Table.

**c**

*inherited from the* [c](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.c" \o "sqlalchemy.sql.expression.FromClause.c) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

An alias for the [columns](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.columns" \o "sqlalchemy.schema.Table.columns) attribute.

**columns**

*inherited from the* [columns](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.columns" \o "sqlalchemy.sql.expression.FromClause.columns) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

A named-based collection of [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) objects maintained by this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

The [columns](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.columns" \o "sqlalchemy.schema.Table.columns), or [c](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.c" \o "sqlalchemy.schema.Table.c) collection, is the gateway to the construction of SQL expressions using table-bound or other selectable-bound columns:

select([mytable]).where(mytable.c.somecolumn == 5)

**compare**(*other*, *\*\*kw*)

*inherited from the* [compare()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.compare" \o "sqlalchemy.sql.expression.ClauseElement.compare) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Compare this ClauseElement to the given ClauseElement.

Subclasses should override the default behavior, which is a straight identity comparison.

\*\*kw are arguments consumed by subclass compare() methods and may be used to modify the criteria for comparison. (see [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement))

**compile**(*bind=None*, *dialect=None*, *\*\*kw*)

*inherited from the* [compile()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.compile" \o "sqlalchemy.sql.expression.ClauseElement.compile) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Compile this SQL expression.

The return value is a [Compiled](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Compiled" \o "sqlalchemy.engine.interfaces.Compiled) object. Calling str() or unicode() on the returned value will yield a string representation of the result. The[Compiled](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Compiled" \o "sqlalchemy.engine.interfaces.Compiled) object also can return a dictionary of bind parameter names and values using the params accessor.

|  |  |
| --- | --- |
| **Parameters:** | * ****bind**** – An Engine or Connection from which a Compiled will be acquired. This argument takes precedence over this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)'s bound engine, if any. * ****column\_keys**** – Used for INSERT and UPDATE statements, a list of column names which should be present in the VALUES clause of the compiled statement. If None, all columns from the target table object are rendered. * ****dialect**** – A Dialect instance from which a Compiled will be acquired. This argument takes precedence over the bind argument as well as this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)'s bound engine, if any. * ****inline**** – Used for INSERT statements, for a dialect which does not support inline retrieval of newly generated primary key columns, will force the expression used to create the new primary key value to be rendered inline within the INSERT statement's VALUES clause. This typically refers to Sequence execution but may also refer to any server-side default generation function associated with a primary key Column. * ****compile\_kwargs –****optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the literal\_binds flag through:   **from** **sqlalchemy.sql** **import** table, column, select  t = table('t', column('x'))  s = select([t]).where(t.c.x == 5)  print s.compile(compile\_kwargs={"literal\_binds": **True**})  *New in version 0.9.0.* |

**See also**

[How do I render SQL expressions as strings, possibly with bound parameters inlined?](http://docs.sqlalchemy.org/en/rel_1_1/faq/sqlexpressions.html" \l "faq-sql-expression-string)

**correspond\_on\_equivalents**(*column*, *equivalents*)

*inherited from the* [correspond\_on\_equivalents()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.correspond_on_equivalents" \o "sqlalchemy.sql.expression.FromClause.correspond_on_equivalents) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return corresponding\_column for the given column, or if None search for a match in the given dictionary.

**corresponding\_column**(*column*, *require\_embedded=False*)

*inherited from the* [corresponding\_column()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.corresponding_column" \o "sqlalchemy.sql.expression.FromClause.corresponding_column) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Given a [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement), return the exported [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) object from this [Selectable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Selectable" \o "sqlalchemy.sql.expression.Selectable) which corresponds to that original [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) via a common ancestor column.

|  |  |
| --- | --- |
| **Parameters:** | * ****column**** – the target [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) to be matched * ****require\_embedded**** – only return corresponding columns for the given [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement), if the given [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) is actually present within a sub-element of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause). Normally the column will match if it merely shares a common ancestor with one of the exported columns of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause). |

**count**(*whereclause=None*, *\*\*params*)

*inherited from the* [count()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.count" \o "sqlalchemy.sql.expression.FromClause.count) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

return a SELECT COUNT generated against this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

*Deprecated since version 1.1:*FromClause.count() is deprecated. Counting rows requires that the correct column expression and accommodations for joins, DISTINCT, etc. must be made, otherwise results may not be what's expected. Please use an appropriate func.count() expression directly.

The function generates COUNT against the first column in the primary key of the table, or against the first column in the table overall. Explicit use offunc.count() should be preferred:

row\_count = conn.scalar(

select([func.count('\*')]).select\_from(table))

**See also**

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

**create**(*bind=None*, *checkfirst=False*)

Issue a CREATE statement for this [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table), using the given [Connectable](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connectable" \o "sqlalchemy.engine.Connectable) for connectivity.

**See also**

[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).

**delete**(*whereclause=None*, *\*\*kwargs*)

*inherited from the* [delete()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableClause.delete" \o "sqlalchemy.sql.expression.TableClause.delete) *method of* [TableClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableClause" \o "sqlalchemy.sql.expression.TableClause)

Generate a [delete()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.delete" \o "sqlalchemy.sql.expression.delete) construct against this [TableClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableClause" \o "sqlalchemy.sql.expression.TableClause).

E.g.:

table.delete().where(table.c.id==7)

See [delete()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.delete" \o "sqlalchemy.sql.expression.delete) for argument and usage information.

**dialect\_kwargs**

*inherited from the* [dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs) *attribute of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

A collection of keyword arguments specified as dialect-specific options to this construct.

The arguments are present here in their original <dialect>\_<kwarg> format. Only arguments that were actually passed are included; unlike the [DialectKWArgs.dialect\_options](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_options" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_options) collection, which contains all options known by this dialect including defaults.

The collection is also writable; keys are accepted of the form <dialect>\_<kwarg> where the value will be assembled into the list of options.

*New in version 0.9.2.*

*Changed in version 0.9.4:*The [DialectKWArgs.dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs) collection is now writable.

**See also**

[DialectKWArgs.dialect\_options](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_options" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_options) - nested dictionary form

**dialect\_options**

*inherited from the* [dialect\_options](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_options" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_options) *attribute of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

A collection of keyword arguments specified as dialect-specific options to this construct.

This is a two-level nested registry, keyed to <dialect\_name> and <argument\_name>. For example, the postgresql\_where argument would be locatable as:

arg = my\_object.dialect\_options['postgresql']['where']

*New in version 0.9.2.*

**See also**

[DialectKWArgs.dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs) - flat dictionary form

**drop**(*bind=None*, *checkfirst=False*)

Issue a DROP statement for this [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table), using the given [Connectable](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connectable" \o "sqlalchemy.engine.Connectable) for connectivity.

**See also**

[MetaData.drop\_all()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData.drop_all" \o "sqlalchemy.schema.MetaData.drop_all).

**exists**(*bind=None*)

Return True if this table exists.

**foreign\_key\_constraints**

[ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint) objects referred to by this [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table).

This list is produced from the collection of [ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey) objects currently associated.

*New in version 1.0.0.*

**foreign\_keys**

*inherited from the* [foreign\_keys](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.foreign_keys" \o "sqlalchemy.sql.expression.FromClause.foreign_keys) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return the collection of ForeignKey objects which this FromClause references.

**info**

*inherited from the* [info](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem.info" \o "sqlalchemy.schema.SchemaItem.info) *attribute of* [SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem)

Info dictionary associated with the object, allowing user-defined data to be associated with this [SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem).

The dictionary is automatically generated when first accessed. It can also be specified in the constructor of some objects, such as [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) and [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column).

**insert**(*values=None*, *inline=False*, *\*\*kwargs*)

*inherited from the* [insert()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableClause.insert" \o "sqlalchemy.sql.expression.TableClause.insert) *method of* [TableClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableClause" \o "sqlalchemy.sql.expression.TableClause)

Generate an [insert()](http://docs.sqlalchemy.org/en/rel_1_1/dialects/postgresql.html" \l "sqlalchemy.dialects.postgresql.dml.insert" \o "sqlalchemy.dialects.postgresql.dml.insert) construct against this [TableClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableClause" \o "sqlalchemy.sql.expression.TableClause).

E.g.:

table.insert().values(name='foo')

See [insert()](http://docs.sqlalchemy.org/en/rel_1_1/dialects/postgresql.html" \l "sqlalchemy.dialects.postgresql.dml.insert" \o "sqlalchemy.dialects.postgresql.dml.insert) for argument and usage information.

**is\_derived\_from**(*fromclause*)

*inherited from the* [is\_derived\_from()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.is_derived_from" \o "sqlalchemy.sql.expression.FromClause.is_derived_from) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return True if this FromClause is 'derived' from the given FromClause.

An example would be an Alias of a Table is derived from that Table.

**join**(*right*, *onclause=None*, *isouter=False*, *full=False*)

*inherited from the* [join()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.join" \o "sqlalchemy.sql.expression.FromClause.join) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) from this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) to another FromClause.

E.g.:

**from** **sqlalchemy** **import** join

j = user\_table.join(address\_table,

user\_table.c.id == address\_table.c.user\_id)stmt = select([user\_table]).select\_from(j)

would emit SQL along the lines of:

SELECT user.id, user.name FROM userJOIN address ON user.id = address.user\_id

|  |  |
| --- | --- |
| **Parameters:** | * ****right**** – the right side of the join; this is any [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) object such as a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object, and may also be a selectable-compatible object such as an ORM-mapped class. * ****onclause**** – a SQL expression representing the ON clause of the join. If left at None, [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) will attempt to join the two tables based on a foreign key relationship. * ****isouter**** – if True, render a LEFT OUTER JOIN, instead of JOIN. * ****full –****if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN. Implies [FromClause.join.isouter](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.join.params.isouter" \o "sqlalchemy.sql.expression.FromClause.join)   *New in version 1.1.* |

**See also**

[join()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.join" \o "sqlalchemy.sql.expression.join) - standalone function

[Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) - the type of object produced

**key**

Return the 'key' for this [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table).

This value is used as the dictionary key within the [MetaData.tables](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData.tables" \o "sqlalchemy.schema.MetaData.tables) collection. It is typically the same as that of Table.name for a table with noTable.schema set; otherwise it is typically of the form schemaname.tablename.

**kwargs**

*inherited from the* [kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.kwargs) *attribute of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

A synonym for [DialectKWArgs.dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs).

**lateral**(*name=None*)

*inherited from the* [lateral()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.lateral" \o "sqlalchemy.sql.expression.FromClause.lateral) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a LATERAL alias of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

The return value is the [Lateral](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Lateral" \o "sqlalchemy.sql.expression.Lateral) construct also provided by the top-level [lateral()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.lateral" \o "sqlalchemy.sql.expression.lateral) function.

*New in version 1.1.*

**See also**

[LATERAL correlation](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html" \l "lateral-selects) - overview of usage.

**outerjoin**(*right*, *onclause=None*, *full=False*)

*inherited from the* [outerjoin()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.outerjoin" \o "sqlalchemy.sql.expression.FromClause.outerjoin) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a [Join](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Join" \o "sqlalchemy.sql.expression.Join) from this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) to another FromClause, with the "isouter" flag set to True.

E.g.:

**from** **sqlalchemy** **import** outerjoin

j = user\_table.outerjoin(address\_table,

user\_table.c.id == address\_table.c.user\_id)

The above is equivalent to:

j = user\_table.join(

address\_table,

user\_table.c.id == address\_table.c.user\_id,

isouter=**True**)

|  |  |
| --- | --- |
| **Parameters:** | * ****right**** – the right side of the join; this is any [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause) object such as a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object, and may also be a selectable-compatible object such as an ORM-mapped class. * ****onclause**** – a SQL expression representing the ON clause of the join. If left at None, [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) will attempt to join the two tables based on a foreign key relationship. * ****full –****if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN.   *New in version 1.1.* |

**See also**

[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)

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

**primary\_key**

*inherited from the* [primary\_key](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.primary_key" \o "sqlalchemy.sql.expression.FromClause.primary_key) *attribute of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return the collection of Column objects which comprise the primary key of this FromClause.

**quote**

*inherited from the* [quote](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem.quote" \o "sqlalchemy.schema.SchemaItem.quote) *attribute of* [SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem)

Return the value of the quote flag passed to this schema object, for those schema items which have a name field.

*Deprecated since version 0.9:*Use <obj>.name.quote

**quote\_schema**

Return the value of the quote\_schema flag passed to this [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table).

*Deprecated since version 0.9:*Use table.schema.quote

**replace\_selectable**(*old*, *alias*)

*inherited from the* [replace\_selectable()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.replace_selectable" \o "sqlalchemy.sql.expression.FromClause.replace_selectable) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

replace all occurrences of FromClause 'old' with the given Alias object, returning a copy of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

**select**(*whereclause=None*, *\*\*params*)

*inherited from the* [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.select" \o "sqlalchemy.sql.expression.FromClause.select) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

return a SELECT of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

**See also**

[select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) - general purpose method which allows for arbitrary column lists.

**self\_group**(*against=None*)

*inherited from the* [self\_group()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement.self_group" \o "sqlalchemy.sql.expression.ClauseElement.self_group) *method of* [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement)

Apply a 'grouping' to this [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement).

This method is overridden by subclasses to return a "grouping" construct, i.e. parenthesis. In particular it's used by "binary" expressions to provide a grouping around themselves when placed into a larger expression, as well as by [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) constructs when placed into the FROM clause of another [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select). (Note that subqueries should be normally created using the [Select.alias()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Select.alias" \o "sqlalchemy.sql.expression.Select.alias) method, as many platforms require nested SELECT statements to be named).

As expressions are composed together, the application of [self\_group()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.self_group" \o "sqlalchemy.schema.Table.self_group) is automatic - end-user code should never need to use this method directly. Note that SQLAlchemy's clause constructs take operator precedence into account - so parenthesis might not be needed, for example, in an expression like x OR (y AND z) - AND takes precedence over OR.

The base [self\_group()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.self_group" \o "sqlalchemy.schema.Table.self_group) method of [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement) just returns self.

**tablesample**(*sampling*, *name=None*, *seed=None*)

*inherited from the* [tablesample()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause.tablesample" \o "sqlalchemy.sql.expression.FromClause.tablesample) *method of* [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause)

Return a TABLESAMPLE alias of this [FromClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.FromClause" \o "sqlalchemy.sql.expression.FromClause).

The return value is the [TableSample](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableSample" \o "sqlalchemy.sql.expression.TableSample) construct also provided by the top-level [tablesample()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.tablesample" \o "sqlalchemy.sql.expression.tablesample) function.

*New in version 1.1.*

**See also**

[tablesample()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.tablesample" \o "sqlalchemy.sql.expression.tablesample) - usage guidelines and parameters

**tometadata**(*metadata*, *schema=symbol('retain\_schema')*, *referred\_schema\_fn=None*, *name=None*)

Return a copy of this [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) associated with a different [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData).

E.g.:

m1 = MetaData()

user = Table('user', m1, Column('id', Integer, priamry\_key=**True**))

m2 = MetaData()user\_copy = user.tometadata(m2)

|  |  |
| --- | --- |
| **Parameters:** | * ****metadata**** – Target [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) object, into which the new [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object will be created. * ****schema –****optional string name indicating the target schema. Defaults to the special symbol RETAIN\_SCHEMA which indicates that no change to the schema name should be made in the new [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table). If set to a string name, the new [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) will have this new name as the .schema. If set to None, the schema will be set to that of the schema set on the target [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData), which is typically None as well, unless set explicitly:   m2 = MetaData(schema='newschema')  *# user\_copy\_one will have "newschema" as the schema name*user\_copy\_one = user.tometadata(m2, schema=**None**)  m3 = MetaData() *# schema defaults to None*  *# user\_copy\_two will have None as the schema name*user\_copy\_two = user.tometadata(m3, schema=**None**)   * ****referred\_schema\_fn**** –   optional callable which can be supplied in order to provide for the schema name that should be assigned to the referenced table of a [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint). The callable accepts this parent [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table), the target schema that we are changing to, the [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint)object, and the existing "target schema" of that constraint. The function should return the string schema name that should be applied. E.g.:  **def** referred\_schema\_fn(table, to\_schema,  constraint, referred\_schema):  **if** referred\_schema == 'base\_tables':  **return** referred\_schema  **else**:  **return** to\_schema  new\_table = table.tometadata(m2, schema="alt\_schema",  referred\_schema\_fn=referred\_schema\_fn)  *New in version 0.9.2.*   * ****name**** –   optional string name indicating the target table name. If not specified or None, the table name is retained. This allows a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) to be copied to the same [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) target with a new name.  *New in version 1.0.0.* |

**update**(*whereclause=None*, *values=None*, *inline=False*, *\*\*kwargs*)

*inherited from the* [update()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableClause.update" \o "sqlalchemy.sql.expression.TableClause.update) *method of* [TableClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableClause" \o "sqlalchemy.sql.expression.TableClause)

Generate an [update()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.update" \o "sqlalchemy.sql.expression.update) construct against this [TableClause](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.TableClause" \o "sqlalchemy.sql.expression.TableClause).

E.g.:

table.update().where(table.c.id==7).values(name='foo')

See [update()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.update" \o "sqlalchemy.sql.expression.update) for argument and usage information.

*class*sqlalchemy.schema.**ThreadLocalMetaData**

Bases: [sqlalchemy.schema.MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData)

A MetaData variant that presents a different bind in every thread.

Makes the bind property of the MetaData a thread-local value, allowing this collection of tables to be bound to different Engine implementations or connections in each thread.

The ThreadLocalMetaData starts off bound to None in each thread. Binds must be made explicitly by assigning to the bind property or using connect(). You can also re-bind dynamically multiple times per thread, just like a regular MetaData.

**\_\_init\_\_**()

Construct a ThreadLocalMetaData.

**bind**

The bound Engine or Connection for this thread.

This property may be assigned an Engine or Connection, or assigned a string or URL to automatically create a basic Engine for this bind with create\_engine().

**dispose**()

Dispose all bound engines, in all thread contexts.

**is\_bound**()

True if there is a bind for this thread.

## 3.2 Reflecting Database Objects

A [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object can be instructed to load information about itself from the corresponding database schema object already existing within the database. This process is called *reflection*. In the most simple case you need only specify the table name, a [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) object, and the autoload=True flag. If the [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) is not persistently bound, also add the autoload\_with argument:

可以指示Table对象从数据库中已有的相应数据库模式对象加载有关自己的信息。 这个过程被称为反射。 在最简单的情况下，您只需要指定表名称，MetaData对象和autoload = True标志。 如果元数据不是永久绑定的，还要添加autoload\_with参数：

**>>>** messages = Table('messages', meta, autoload=**True**, autoload\_with=engine)

**>>>** [c.name **for** c **in** messages.columns]['message\_id', 'message\_name', 'date']

The above operation will use the given engine to query the database for information about the messages table, and will then generate [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column), [ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey), and other objects corresponding to this information as though the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object were hand-constructed in Python.

When tables are reflected, if a given table references another one via foreign key, a second [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object is created within the [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) object representing the connection. Below, assume the table shopping\_cart\_items references a table named shopping\_carts. Reflecting the shopping\_cart\_items table has the effect such that the shopping\_carts table will also be loaded:

上面的操作将使用给定的引擎来查询数据库中关于消息表的信息，然后生成与这个信息相对应的Column，ForeignKey和其他对象，就好像Table对象是用Python手工构造的一样。

当表被反映时，如果给定的表通过外键引用另一个表，则在表示连接的MetaData对象内创建第二个表对象。 在下面，假设表shopping\_cart\_items引用了一个名为shopping\_carts的表。 反映shopping\_cart\_items表具有这样的效果，即shopping\_carts表也将被加载：

**>>>** shopping\_cart\_items = Table('shopping\_cart\_items', meta, autoload=**True**, autoload\_with=engine)

**>>>** 'shopping\_carts' **in** meta.tables:True

The [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) has an interesting "singleton-like" behavior such that if you requested both tables individually, [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) will ensure that exactly one [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object is created for each distinct table name. The [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) constructor actually returns to you the already-existing [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object if one already exists with the given name. Such as below, we can access the already generated shopping\_carts table just by naming it:

MetaData有一个有趣的“类似单身”的行为，如果您单独请求两个表，MetaData将确保为每个不同的表名创建一个Table对象。 如果已经存在具有给定名称的表，则表构造函数实际返回已经存在的Table对象。 如下所示，我们可以通过命名来访问已经生成的shopping\_carts表：

shopping\_carts = Table('shopping\_carts', meta)

Of course, it's a good idea to use autoload=True with the above table regardless. This is so that the table's attributes will be loaded if they have not been already. The autoload operation only occurs for the table if it hasn't already been loaded; once loaded, new calls to [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) with the same name will not re-issue any reflection queries.

当然，无论如何，使用autoload = True和上面的表格是一个好主意。 这样就可以加载表的属性，如果还没有。 如果尚未加载表，则仅对该表进行自动加载操作; 一旦加载，对同名的表的新调用将不会重新发出任何反射查询。

3.2.1 Overriding Reflected Columns

Individual columns can be overridden with explicit values when reflecting tables; this is handy for specifying custom datatypes, constraints such as primary keys that may not be configured within the database, etc.:

反射表时，可以用显式值覆盖各列; 这有助于指定自定义数据类型，诸如可能在数据库中未配置的主键等约束。

**>>>** mytable = Table('mytable', meta,

**...** Column('id', Integer, primary\_key=**True**), *# override reflected 'id' to have primary key*

**...** Column('mydata', Unicode(50)), *# override reflected 'mydata' to be Unicode*

**...** autoload=**True**)

### 3.2.2 Reflecting Views

The reflection system can also reflect views. Basic usage is the same as that of a table:

my\_view = Table("some\_view", metadata, autoload=**True**)

Above, my\_view is a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object with [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects representing the names and types of each column within the view "some\_view".

Usually, it's desired to have at least a primary key constraint when reflecting a view, if not foreign keys as well. View reflection doesn't extrapolate these constraints.

Use the "override" technique for this, specifying explicitly those columns which are part of the primary key or have foreign key constraints:

my\_view = Table("some\_view", metadata,

Column("view\_id", Integer, primary\_key=**True**),

Column("related\_thing", Integer, ForeignKey("othertable.thing\_id")),

autoload=**True**)

### 3.2.3 Reflecting All Tables at Once

The [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) object can also get a listing of tables and reflect the full set. This is achieved by using the [reflect()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData.reflect" \o "sqlalchemy.schema.MetaData.reflect) method. After calling it, all located tables are present within the [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) object's dictionary of tables:

meta = MetaData()meta.reflect(bind=someengine)users\_table = meta.tables['users']addresses\_table = meta.tables['addresses']

metadata.reflect() also provides a handy way to clear or delete all the rows in a database:

meta = MetaData()meta.reflect(bind=someengine)**for** table **in** reversed(meta.sorted\_tables):

someengine.execute(table.delete())

### 3.2.4 Fine Grained Reflection with Inspector

A low level interface which provides a backend-agnostic system of loading lists of schema, table, column, and constraint descriptions from a given database is also available. This is known as the "Inspector":

还提供了一个低级接口，它提供了一个从给定数据库加载模式，表，列和约束描述列表的后端无关系统。 这被称为“检查员”：

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

**from** **sqlalchemy.engine** **import** reflection

engine = create\_engine('...')

insp = reflection.Inspector.from\_engine(engine)

print(insp.get\_table\_names())

*class*sqlalchemy.engine.reflection.**Inspector**(*bind*)

Performs database schema inspection.

执行数据库模式检查。

The Inspector acts as a proxy to the reflection methods of the [Dialect](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Dialect" \o "sqlalchemy.engine.interfaces.Dialect), providing a consistent interface as well as caching support for previously fetched metadata.

检查员充当方言的反射方法的代理，为先前提取的元数据提供一致的界面以及缓存支持。

A [Inspector](http://docs.sqlalchemy.org/en/rel_1_1/core/reflection.html" \l "sqlalchemy.engine.reflection.Inspector" \o "sqlalchemy.engine.reflection.Inspector) object is usually created via the [inspect()](http://docs.sqlalchemy.org/en/rel_1_1/core/inspection.html" \l "sqlalchemy.inspection.inspect" \o "sqlalchemy.inspection.inspect) function:

Inspector对象通常通过inspect() 函数创建：

**from** **sqlalchemy** **import** inspect, create\_engine

engine = create\_engine('...')

insp = inspect(engine)

The inspection method above is equivalent to using the [Inspector.from\_engine()](http://docs.sqlalchemy.org/en/rel_1_1/core/reflection.html" \l "sqlalchemy.engine.reflection.Inspector.from_engine" \o "sqlalchemy.engine.reflection.Inspector.from_engine) method, i.e.:

engine = create\_engine('...')insp = Inspector.from\_engine(engine)

Where above, the [Dialect](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Dialect" \o "sqlalchemy.engine.interfaces.Dialect) may opt to return an [Inspector](http://docs.sqlalchemy.org/en/rel_1_1/core/reflection.html" \l "sqlalchemy.engine.reflection.Inspector" \o "sqlalchemy.engine.reflection.Inspector) subclass that provides additional methods specific to the dialect's target database.

**\_\_init\_\_**(*bind*)

Initialize a new [Inspector](http://docs.sqlalchemy.org/en/rel_1_1/core/reflection.html" \l "sqlalchemy.engine.reflection.Inspector" \o "sqlalchemy.engine.reflection.Inspector).

|  |  |
| --- | --- |
| **Parameters:** | ****bind**** – a [Connectable](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connectable" \o "sqlalchemy.engine.Connectable), which is typically an instance of [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). |

For a dialect-specific instance of [Inspector](http://docs.sqlalchemy.org/en/rel_1_1/core/reflection.html" \l "sqlalchemy.engine.reflection.Inspector" \o "sqlalchemy.engine.reflection.Inspector), see [Inspector.from\_engine()](http://docs.sqlalchemy.org/en/rel_1_1/core/reflection.html" \l "sqlalchemy.engine.reflection.Inspector.from_engine" \o "sqlalchemy.engine.reflection.Inspector.from_engine)

**default\_schema\_name**

Return the default schema name presented by the dialect for the current engine's database user.

返回当前引擎的数据库用户的方言显示的默认模式名称。

E.g. this is typically public for PostgreSQL and dbo for SQL Server.

例如。 这通常是PostgreSQL公开的，对于SQL Server是dbo。

*classmethod***from\_engine**(*bind*)

Construct a new dialect-specific Inspector object from the given engine or connection.

从给定的引擎或连接构建一个新的方言特定的Inspector对象。

|  |  |
| --- | --- |
| **Parameters:** | ****bind**** – a [Connectable](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connectable" \o "sqlalchemy.engine.Connectable), which is typically an instance of [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). |

This method differs from direct a direct constructor call of [Inspector](http://docs.sqlalchemy.org/en/rel_1_1/core/reflection.html" \l "sqlalchemy.engine.reflection.Inspector" \o "sqlalchemy.engine.reflection.Inspector) in that the [Dialect](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Dialect" \o "sqlalchemy.engine.interfaces.Dialect) is given a chance to provide a dialect-specific [Inspector](http://docs.sqlalchemy.org/en/rel_1_1/core/reflection.html" \l "sqlalchemy.engine.reflection.Inspector" \o "sqlalchemy.engine.reflection.Inspector)instance, which may provide additional methods.

该方法与Inspector的直接构造函数调用不同之处在于，方言有机会提供方言特定的检查器，可提供其他方法。

See the example at [Inspector](http://docs.sqlalchemy.org/en/rel_1_1/core/reflection.html" \l "sqlalchemy.engine.reflection.Inspector" \o "sqlalchemy.engine.reflection.Inspector).

**get\_check\_constraints**(*table\_name*, *schema=None*, *\*\*kw*)

Return information about check constraints in table\_name.

返回有关table\_name中检查约束的信息。

Given a string table\_name and an optional string schema, return check constraint information as a list of dicts with these keys:

给定一个字符串table\_name和一个可选的字符串模式，返回检查约束信息作为具有这些键的dicts列表：

name

the check constraint's name

sqltext

the check constraint's SQL expression

|  |  |
| --- | --- |
| **Parameters:** | * ****table\_name**** – string name of the table. For special quoting, use [quoted\_name](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.elements.quoted_name" \o "sqlalchemy.sql.elements.quoted_name). * ****schema**** – string schema name; if omitted, uses the default schema of the database connection. For special quoting, use [quoted\_name](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.elements.quoted_name" \o "sqlalchemy.sql.elements.quoted_name). |

*New in version 1.1.0.*

**get\_columns**(*table\_name*, *schema=None*, *\*\*kw*)

Return information about columns in table\_name.

Given a string table\_name and an optional string schema, return column information as a list of dicts with these keys:

* name - the column's name
* type - the type of this column; an instance of [TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine)
* nullable - boolean flag if the column is NULL or NOT NULL
* default - the column's server default value - this is returned as a string SQL expression.
* attrs - dict containing optional column attributes

|  |  |
| --- | --- |
| **Parameters:** | * ****table\_name**** – string name of the table. For special quoting, use [quoted\_name](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.elements.quoted_name" \o "sqlalchemy.sql.elements.quoted_name). * ****schema**** – string schema name; if omitted, uses the default schema of the database connection. For special quoting, use [quoted\_name](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.elements.quoted_name" \o "sqlalchemy.sql.elements.quoted_name). |
| **Returns:** | list of dictionaries, each representing the definition of a database column. |

**get\_foreign\_keys**(*table\_name*, *schema=None*, *\*\*kw*)

Return information about foreign\_keys in table\_name.

Given a string table\_name, and an optional string schema, return foreign key information as a list of dicts with these keys:

constrained\_columns

a list of column names that make up the foreign key

referred\_schema

the name of the referred schema

referred\_table

the name of the referred table

referred\_columns

a list of column names in the referred table that correspond to constrained\_columns

name

optional name of the foreign key constraint.

|  |  |
| --- | --- |
| **Parameters:** | * ****table\_name**** – string name of the table. For special quoting, use [quoted\_name](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.elements.quoted_name" \o "sqlalchemy.sql.elements.quoted_name). * ****schema**** – string schema name; if omitted, uses the default schema of the database connection. For special quoting, use [quoted\_name](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.elements.quoted_name" \o "sqlalchemy.sql.elements.quoted_name). |

**get\_indexes**(*table\_name*, *schema=None*, *\*\*kw*)

Return information about indexes in table\_name.

Given a string table\_name and an optional string schema, return index information as a list of dicts with these keys:

name

the index's name

column\_names

list of column names in order

unique

boolean

dialect\_options

dict of dialect-specific index options. May not be present for all dialects.

*New in version 1.0.0.*

|  |  |
| --- | --- |
| **Parameters:** | * ****table\_name**** – string name of the table. For special quoting, use [quoted\_name](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.elements.quoted_name" \o "sqlalchemy.sql.elements.quoted_name). * ****schema**** – string schema name; if omitted, uses the default schema of the database connection. For special quoting, use [quoted\_name](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.elements.quoted_name" \o "sqlalchemy.sql.elements.quoted_name). |

**get\_pk\_constraint**(*table\_name*, *schema=None*, *\*\*kw*)

Return information about primary key constraint on table\_name.

Given a string table\_name, and an optional string schema, return primary key information as a dictionary with these keys:

constrained\_columns

a list of column names that make up the primary key

name

optional name of the primary key constraint.

|  |  |
| --- | --- |
| **Parameters:** | * ****table\_name**** – string name of the table. For special quoting, use [quoted\_name](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.elements.quoted_name" \o "sqlalchemy.sql.elements.quoted_name). * ****schema**** – string schema name; if omitted, uses the default schema of the database connection. For special quoting, use [quoted\_name](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.elements.quoted_name" \o "sqlalchemy.sql.elements.quoted_name). |

**get\_primary\_keys**(*table\_name*, *schema=None*, *\*\*kw*)

Return information about primary keys in table\_name.

*Deprecated since version 0.7:*Call to deprecated method get\_primary\_keys. Use get\_pk\_constraint instead.

Given a string table\_name, and an optional string schema, return primary key information as a list of column names.

**get\_schema\_names**()

Return all schema names.

**get\_sorted\_table\_and\_fkc\_names**(*schema=None*)

Return dependency-sorted table and foreign key constraint names in referred to within a particular schema.

This will yield 2-tuples of (tablename, [(tname, fkname), (tname, fkname), ...]) consisting of table names in CREATE order grouped with the foreign key constraint names that are not detected as belonging to a cycle. The final element will be (None, [(tname, fkname), (tname,fkname), ..]) which will consist of remaining foreign key constraint names that would require a separate CREATE step after-the-fact, based on dependencies between tables.

*New in version 1.0.-.*

**See also**

[Inspector.get\_table\_names()](http://docs.sqlalchemy.org/en/rel_1_1/core/reflection.html" \l "sqlalchemy.engine.reflection.Inspector.get_table_names" \o "sqlalchemy.engine.reflection.Inspector.get_table_names)

[sort\_tables\_and\_constraints()](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.sort_tables_and_constraints" \o "sqlalchemy.schema.sort_tables_and_constraints) - similar method which works

with an already-given [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData).

**get\_table\_names**(*schema=None*, *order\_by=None*)

Return all table names in referred to within a particular schema.

The names are expected to be real tables only, not views. Views are instead returned using the [Inspector.get\_view\_names()](http://docs.sqlalchemy.org/en/rel_1_1/core/reflection.html" \l "sqlalchemy.engine.reflection.Inspector.get_view_names" \o "sqlalchemy.engine.reflection.Inspector.get_view_names) method.

|  |  |
| --- | --- |
| **Parameters:** | * ****schema**** – Schema name. If schema is left at None, the database's default schema is used, else the named schema is searched. If the database does not support named schemas, behavior is undefined if schema is not passed as None. For special quoting, use [quoted\_name](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.elements.quoted_name" \o "sqlalchemy.sql.elements.quoted_name). * ****order\_by**** –   Optional, may be the string "foreign\_key" to sort the result on foreign key dependencies. Does not automatically resolve cycles, and will raise [CircularDependencyError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.CircularDependencyError" \o "sqlalchemy.exc.CircularDependencyError) if cycles exist.  *Deprecated since version 1.0.0:*- see [Inspector.get\_sorted\_table\_and\_fkc\_names()](http://docs.sqlalchemy.org/en/rel_1_1/core/reflection.html" \l "sqlalchemy.engine.reflection.Inspector.get_sorted_table_and_fkc_names" \o "sqlalchemy.engine.reflection.Inspector.get_sorted_table_and_fkc_names) for a version of this which resolves foreign key cycles between tables automatically.  *Changed in version 0.8:*the "foreign\_key" sorting sorts tables in order of dependee to dependent; that is, in creation order, rather than in drop order. This is to maintain consistency with similar features such as [MetaData.sorted\_tables](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData.sorted_tables" \o "sqlalchemy.schema.MetaData.sorted_tables) and util.sort\_tables(). |

**See also**

[Inspector.get\_sorted\_table\_and\_fkc\_names()](http://docs.sqlalchemy.org/en/rel_1_1/core/reflection.html" \l "sqlalchemy.engine.reflection.Inspector.get_sorted_table_and_fkc_names" \o "sqlalchemy.engine.reflection.Inspector.get_sorted_table_and_fkc_names)

[MetaData.sorted\_tables](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData.sorted_tables" \o "sqlalchemy.schema.MetaData.sorted_tables)

**get\_table\_options**(*table\_name*, *schema=None*, *\*\*kw*)

Return a dictionary of options specified when the table of the given name was created.

This currently includes some options that apply to MySQL tables.

|  |  |
| --- | --- |
| **Parameters:** | * ****table\_name**** – string name of the table. For special quoting, use [quoted\_name](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.elements.quoted_name" \o "sqlalchemy.sql.elements.quoted_name). * ****schema**** – string schema name; if omitted, uses the default schema of the database connection. For special quoting, use [quoted\_name](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.elements.quoted_name" \o "sqlalchemy.sql.elements.quoted_name). |

**get\_temp\_table\_names**()

return a list of temporary table names for the current bind.

This method is unsupported by most dialects; currently only SQLite implements it.

*New in version 1.0.0.*

**get\_temp\_view\_names**()

return a list of temporary view names for the current bind.

This method is unsupported by most dialects; currently only SQLite implements it.

*New in version 1.0.0.*

**get\_unique\_constraints**(*table\_name*, *schema=None*, *\*\*kw*)

Return information about unique constraints in table\_name.

Given a string table\_name and an optional string schema, return unique constraint information as a list of dicts with these keys:

name

the unique constraint's name

column\_names

list of column names in order

|  |  |
| --- | --- |
| **Parameters:** | * ****table\_name**** – string name of the table. For special quoting, use [quoted\_name](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.elements.quoted_name" \o "sqlalchemy.sql.elements.quoted_name). * ****schema**** – string schema name; if omitted, uses the default schema of the database connection. For special quoting, use [quoted\_name](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.elements.quoted_name" \o "sqlalchemy.sql.elements.quoted_name). |

*New in version 0.8.4.*

**get\_view\_definition**(*view\_name*, *schema=None*)

Return definition for view\_name.

|  |  |
| --- | --- |
| **Parameters:** | ****schema**** – Optional, retrieve names from a non-default schema. For special quoting, use [quoted\_name](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.elements.quoted_name" \o "sqlalchemy.sql.elements.quoted_name). |

**get\_view\_names**(*schema=None*)

Return all view names in schema.

|  |  |
| --- | --- |
| **Parameters:** | ****schema**** – Optional, retrieve names from a non-default schema. For special quoting, use [quoted\_name](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.elements.quoted_name" \o "sqlalchemy.sql.elements.quoted_name). |

**reflecttable**(*table*, *include\_columns*, *exclude\_columns=()*, *\_extend\_on=None*)

Given a Table object, load its internal constructs based on introspection.

给定一个Table对象，加载其内部构造基于内省。

This is the underlying method used by most dialects to produce table reflection. Direct usage is like:

这是大多数方言用于产生表反射的基础方法。 直接使用就像：

**from** **sqlalchemy** **import** create\_engine, MetaData, Table

**from** **sqlalchemy.engine** **import** reflection

engine = create\_engine('...')

meta = MetaData()

user\_table = Table('user', meta)

insp = Inspector.from\_engine(engine)

insp.reflecttable(user\_table, **None**)

|  |  |
| --- | --- |
| **Parameters:** | * ****table**** – a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) instance. * ****include\_columns**** – a list of string column names to include in the reflection process. If None, all columns are reflected. |

### 3.2.5 Limitations of Reflection

It's important to note that the reflection process recreates [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) metadata using only information which is represented in the relational database. This process by definition cannot restore aspects of a schema that aren't actually stored in the database. State which is not available from reflection includes but is not limited to:

重要的是要注意，反射过程仅使用在关系数据库中表示的信息重新创建[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)元数据。 定义的这个过程无法恢复实际上不存储在数据库中的模式的各个方面。 不能反思的状态包括但不限于：

* Client side defaults, either Python functions or SQL expressions defined using the default keyword of [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) (note this is separate from server\_default, which specifically is what's available via reflection).
* Column information, e.g. data that might have been placed into the [Column.info](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column.info" \o "sqlalchemy.schema.Column.info) dictionary
* The value of the .quote setting for [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) or [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)
* The association of a particular [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) with a given [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column)

The relational database also in many cases reports on table metadata in a different format than what was specified in SQLAlchemy. The [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects returned from reflection cannot be always relied upon to produce the identical DDL as the original Python-defined [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects. Areas where this occurs includes server defaults, column-associated sequences and various idosyncrasies regarding constraints and datatypes. Server side defaults may be returned with cast directives (typically PostgreSQL will include a ::<type> cast) or different quoting patterns than originally specified.

关系数据库在很多情况下也会以与SQLAlchemy中指定的格式不同的格式报告表元数据。 从反射返回的[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)对象不能始终依赖于生成与原始Python定义的[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)对象相同的DDL。 发生这种情况的领域包括服务器默认值，列相关序列以及关于约束和数据类型的各种idosyncr。 可以使用转换指令返回服务器端默认值（通常PostgreSQL将包含一个::<type>cast）或与原始指定的不同的引用模式。

Another category of limitation includes schema structures for which reflection is only partially or not yet defined. Recent improvements to reflection allow things like views, indexes and foreign key options to be reflected. As of this writing, structures like CHECK constraints, table comments, and triggers are not reflected.

另一类限制包括仅部分或尚未定义反射的模式结构。 反思的最近改进允许反映视图，索引和外键选项等内容。 在撰写本文时，结构如CHECK约束，表格注释和触发器都没有反映出来。

# **3.3 Column Insert/Update Defaults**

SQLAlchemy provides a very rich featureset regarding column level events which take place during INSERT and UPDATE statements. Options include:

SQLAlchemy为INSERT和UPDATE语句中的列级别事件提供了非常丰富的功能集。 选项包括：

* Scalar values used as defaults during INSERT and UPDATE operations
* Python functions which execute upon INSERT and UPDATE operations
* SQL expressions which are embedded in INSERT statements (or in some cases execute beforehand)
* SQL expressions which are embedded in UPDATE statements
* Server side default values used during INSERT
* Markers for server-side triggers used during UPDATE

The general rule for all insert/update defaults is that they only take effect if no value for a particular column is passed as an execute() parameter; otherwise, the given value is used.

所有插入/更新默认值的一般规则是它们只有在特定列的值作为execute() 参数传递时才会生效; 否则，使用给定的值。

3.3.1 Scalar Defaults

The simplest kind of default is a scalar value used as the default value of a column:

Table("mytable", meta,

Column("somecolumn", Integer, default=12))

Above, the value "12" will be bound as the column value during an INSERT if no other value is supplied.

如果没有提供其他值，则在“INSERT”期间，值“12”将被绑定为列值。

A scalar value may also be associated with an UPDATE statement, though this is not very common (as UPDATE statements are usually looking for dynamic defaults):

标量值也可能与UPDATE语句相关联，尽管这不是很常见（因为UPDATE语句通常会查找动态默认值）：

Table("mytable", meta,

Column("somecolumn", Integer, onupdate=25))

### 3.3.2 Python-Executed Functions

The [Column.default](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column.params.default" \o "sqlalchemy.schema.Column) and [Column.onupdate](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column.params.onupdate" \o "sqlalchemy.schema.Column) keyword arguments also accept Python functions. These functions are invoked at the time of insert or update if no other value for that column is supplied, and the value returned is used for the column's value. Below illustrates a crude "sequence" that assigns an incrementing counter to a primary key column:

Column.default和Column.onupdate关键字参数也接受Python函数。 如果没有提供该列的其他值，则在插入或更新时调用这些函数，并且返回的值用于列的值。 下面说明了一个粗略的“序列”，它将一个增量计数器分配给主键列：

*# a function which counts upwards*i = 0

**def** mydefault():

**global** i

i += 1

**return** i

t = Table("mytable", meta,

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

It should be noted that for real "incrementing sequence" behavior, the built-in capabilities of the database should normally be used, which may include sequence objects or other autoincrementing capabilities. For primary key columns, SQLAlchemy will in most cases use these capabilities automatically. See the API documentation for [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) including the [Column.autoincrement](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column.params.autoincrement" \o "sqlalchemy.schema.Column) flag, as well as the section on [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) later in this chapter for background on standard primary key generation techniques.

应该注意的是，对于实际的“递增序列”行为，通常应该使用数据库的内置功能，其可以包括序列对象或其他自动增量功能。 对于主键列，SQLAlchemy在大多数情况下将自动使用这些功能。 请参阅列的API文档，其中包括Column.autoincrement标志，以及本章后面部分关于标准主键生成技术背景的部分。

To illustrate onupdate, we assign the Python datetime function now to the [Column.onupdate](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column.params.onupdate" \o "sqlalchemy.schema.Column) attribute:

为了说明onupdate，我们现在将Python datetime函数分配给Column.onupdate属性：

**import** **datetime**

t = Table("mytable", meta,

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

*# define 'last\_updated' to be populated with datetime.now()*

Column('last\_updated', DateTime, onupdate=datetime.datetime.now),)

When an update statement executes and no value is passed for last\_updated, the datetime.datetime.now() Python function is executed and its return value used as the value for last\_updated. Notice that we provide now as the function itself without calling it (i.e. there are no parenthesis following) - SQLAlchemy will execute the function at the time the statement executes.

当执行update语句并且没有为last\_updated传递值时，将执行datetime.datetime.now() Python函数，并将其返回值用作last\_updated的值。 请注意，我们现在提供的函数本身没有调用它（即没有括号下面） - SQLAlchemy将在执行语句时执行该函数。

### Context-Sensitive Default Functions

The Python functions used by [Column.default](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column.params.default" \o "sqlalchemy.schema.Column) and [Column.onupdate](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column.params.onupdate" \o "sqlalchemy.schema.Column) may also make use of the current statement's context in order to determine a value. The context of a statement is an internal SQLAlchemy object which contains all information about the statement being executed, including its source expression, the parameters associated with it and the cursor. The typical use case for this context with regards to default generation is to have access to the other values being inserted or updated on the row. To access the context, provide a function that accepts a single context argument:

Column.default和Column.onupdate使用的Python函数也可以使用当前语句的上下文来确定一个值。 语句的上下文是一个内部SQLAlchemy对象，其中包含有关正在执行的语句的所有信息，包括其源表达式，与其关联的参数和游标。 关于默认生成的上下文的典型用例是访问在该行上插入或更新的其他值。 要访问上下文，请提供一个接受单个上下文参数的函数：

**def** mydefault(context):

**return** context.current\_parameters['counter'] + 12

t = Table('mytable', meta,

Column('counter', Integer),

Column('counter\_plus\_twelve', Integer, default=mydefault, onupdate=mydefault))

Above we illustrate a default function which will execute for all INSERT and UPDATE statements where a value for counter\_plus\_twelve was otherwise not provided, and the value will be that of whatever value is present in the execution for the counter column, plus the number 12.

上面我们说明了一个默认的函数，它将为所有的INSERT和UPDATE语句执行，其中没有提供counter\_plus\_twelve的值，并且该值将是计数器列的执行中加上数字12的值。

While the context object passed to the default function has many attributes, the current\_parameters member is a special member provided only during the execution of a default function for the purposes of deriving defaults from its existing values. For a single statement that is executing many sets of bind parameters, the user-defined function is called for each set of parameters, and current\_parameters will be provided with each individual parameter set for each execution.

传递给默认函数的上下文对象具有许多属性，但current\_parameters成员是仅在执行默认函数时提供的特殊成员，以从其现有值中导出默认值。 对于执行许多绑定参数集合的单个语句，将为每组参数调用用户定义的函数，并为每个执行提供每个参数集的current\_parameters。

3.3.3 SQL Expressions

The "default" and "onupdate" keywords may also be passed SQL expressions, including select statements or direct function calls:

“默认”和“onupdate”关键字也可以传递SQL表达式，包括select语句或直接函数调用：

t = Table("mytable", meta,

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

*# define 'create\_date' to default to now()*

Column('create\_date', DateTime, default=func.now()),

*# define 'key' to pull its default from the 'keyvalues' table*

Column('key', String(20), default=keyvalues.select(keyvalues.c.type='type1', limit=1)),

*# define 'last\_modified' to use the current\_timestamp SQL function on update*

Column('last\_modified', DateTime, onupdate=func.utc\_timestamp())

)

Above, the create\_date column will be populated with the result of the now() SQL function (which, depending on backend, compiles into NOW() or CURRENT\_TIMESTAMP in most cases) during an INSERT statement, and the key column with the result of a SELECT subquery from another table. Thelast\_modified column will be populated with the value of UTC\_TIMESTAMP(), a function specific to MySQL, when an UPDATE statement is emitted for this table.

以上，在INSERT语句中，将使用now() SQL函数（其在多数情况下根据后端编译为NOW() 或CURRENT\_TIMESTAMP）的结果填充create\_date列，结果为a 从另一个表中选择子查询。 当为此表发出UPDATE语句时，Thelast\_modified列将填充UTC\_TIMESTAMP() 的值，这是MySQL特有的函数。

Note that when using func functions, unlike when using Python datetime functions we *do* call the function, i.e. with parenthesis "()" - this is because what we want in this case is the return value of the function, which is the SQL expression construct that will be rendered into the INSERT or UPDATE statement.

请注意，当使用func函数时，与使用Python datetime函数时不同，我们调用函数，即括号“() ” - 这是因为在这种情况下我们想要的是函数的返回值，即SQL表达式构造 这将被呈现为INSERT或UPDATE语句。

The above SQL functions are usually executed "inline" with the INSERT or UPDATE statement being executed, meaning, a single statement is executed which embeds the given expressions or subqueries within the VALUES or SET clause of the statement. Although in some cases, the function is "pre-executed" in a SELECT statement of its own beforehand. This happens when all of the following is true:

上述SQL函数通常在执行INSERT或UPDATE语句时执行“内联”，这意味着，执行一个单独的语句，它将给定的表达式或子查询嵌入在语句的VALUES或SET子句中。 虽然在某些情况下，该函数是事先在其自己的SELECT语句中“预执行”的。 当发生以下所有情况时都会发生这种情况：

* the column is a primary key column
* the database dialect does not support a usable cursor.lastrowid accessor (or equivalent); this currently includes PostgreSQL, Oracle, and Firebird, as well as some MySQL dialects.
* the dialect does not support the "RETURNING" clause or similar, or the implicit\_returning flag is set to False for the dialect. Dialects which support RETURNING currently include PostgreSQL, Oracle, Firebird, and MS-SQL.
* the statement is a single execution, i.e. only supplies one set of parameters and doesn't use "executemany" behavior
* the inline=True flag is not set on the [Insert()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert" \o "sqlalchemy.sql.expression.Insert) or [Update()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Update" \o "sqlalchemy.sql.expression.Update) construct, and the statement has not defined an explicit returning() clause.

Whether or not the default generation clause "pre-executes" is not something that normally needs to be considered, unless it is being addressed for performance reasons.

默认生成条款“预执行”是否不是通常需要考虑的事情，除非出于性能原因而被解决。

When the statement is executed with a single set of parameters (that is, it is not an "executemany" style execution), the returned [ResultProxy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy" \o "sqlalchemy.engine.ResultProxy) will contain a collection accessible via [ResultProxy.postfetch\_cols()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.postfetch_cols" \o "sqlalchemy.engine.ResultProxy.postfetch_cols) which contains a list of all [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects which had an inline-executed default. Similarly, all parameters which were bound to the statement, including all Python and SQL expressions which were pre-executed, are present in the [ResultProxy.last\_inserted\_params()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.last_inserted_params" \o "sqlalchemy.engine.ResultProxy.last_inserted_params) or [ResultProxy.last\_updated\_params()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.last_updated_params" \o "sqlalchemy.engine.ResultProxy.last_updated_params) collections on [ResultProxy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy" \o "sqlalchemy.engine.ResultProxy). The [ResultProxy.inserted\_primary\_key](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.inserted_primary_key" \o "sqlalchemy.engine.ResultProxy.inserted_primary_key) collection contains a list of primary key values for the row inserted (a list so that single-column and composite-column primary keys are represented in the same format).

当使用单个参数集执行语句（即不是执行“执行方式”）时，返回的ResultProxy将包含可通过ResultProxy.postfetch\_cols() 访问的集合，该集合包含所有列对象的列表， 一个内联执行的默认值。 类似地，绑定到语句的所有参数，包括预先执行的所有Python和SQL表达式都存在于ResultProxy上的ResultProxy.last\_inserted\_params() 或ResultProxy.last\_updated\_params() 集合中。 ResultProxy.inserted\_primary\_key集合包含插入行的主键值列表（列表，以使单列和复合列主键以相同的格式表示）。

3.3.4 Server Side Defaults

A variant on the SQL expression default is the [Column.server\_default](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column.params.server_default" \o "sqlalchemy.schema.Column), which gets placed in the CREATE TABLE statement during a [Table.create()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.create" \o "sqlalchemy.schema.Table.create)operation:

SQL表达式默认的变体是Column.server\_default，它在Table.create() 操作期间被放置在CREATE TABLE语句中：

t = Table('test', meta,

Column('abc', String(20), server\_default='abc'),

Column('created\_at', DateTime, server\_default=text("sysdate")))

A create call for the above table will produce:

CREATE TABLE test (

abc varchar(20) default 'abc',

created\_at datetime default sysdate)

The behavior of [Column.server\_default](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column.params.server_default" \o "sqlalchemy.schema.Column) is similar to that of a regular SQL default; if it's placed on a primary key column for a database which doesn't have a way to "postfetch" the ID, and the statement is not "inlined", the SQL expression is pre-executed; otherwise, SQLAlchemy lets the default fire off on the database side normally.

Column.server\_default的行为类似于常规SQL默认值的行为; 如果它放置在数据库的主键列上，该数据库没有方法“后续”该ID，并且该语句不是“内联”，那么SQL表达式是预先执行的; 否则，SQLAlchemy允许在数据库端正常关闭默认值。

3.3.5 Triggered Columns

Columns with values set by a database trigger or other external process may be called out using [FetchedValue](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.FetchedValue" \o "sqlalchemy.schema.FetchedValue) as a marker:

可以使用FetchedValue作为标记来调用具有数据库触发器或其他外部进程设置的值的列：

t = Table('test', meta,

Column('abc', String(20), server\_default=FetchedValue()),

Column('def', String(20), server\_onupdate=FetchedValue()))

These markers do not emit a "default" clause when the table is created, however they do set the same internal flags as a static server\_default clause, providing hints to higher-level tools that a "post-fetch" of these rows should be performed after an insert or update.

当创建表时，这些标记不会发出“默认”子句，但是它们设置与静态server\_default子句相同的内部标志，为更高级别的工具提供提示，以便执行这些行的“后置” 插入或更新后。

**Note**

It's generally not appropriate to use [FetchedValue](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.FetchedValue" \o "sqlalchemy.schema.FetchedValue) in conjunction with a primary key column, particularly when using the ORM or any other scenario where the [ResultProxy.inserted\_primary\_key](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.inserted_primary_key" \o "sqlalchemy.engine.ResultProxy.inserted_primary_key) attribute is required. This is becaue the "post-fetch" operation requires that the primary key value already be available, so that the row can be selected on its primary key.

将FetchedValue与主键列结合使用通常是不合适的，特别是在使用ORM或需要ResultProxy.inserted\_primary\_key属性的任何其他场景时。 这是因为“后抓取”操作要求主键值已经可用，以便可以在其主键上选择该行。

For a server-generated primary key value, all databases provide special accessors or other techniques in order to acquire the "last inserted primary key" column of a table. These mechanisms aren't affected by the presence of [FetchedValue](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.FetchedValue" \o "sqlalchemy.schema.FetchedValue). For special situations where triggers are used to generate primary key values, and the database in use does not support the RETURNING clause, it may be necessary to forego the usage of the trigger and instead apply the SQL expression or function as a "pre execute" expression:

对于服务器生成的主键值，所有数据库都提供特殊的访问器或其他技术，以获取表的“最后插入的主键”列。 这些机制不受FetchedValue的存在的影响。 对于使用触发器来生成主键值的特殊情况，正在使用的数据库不支持RETURNING子句，可能需要放弃触发器的使用，而将SQL表达式或函数应用于“预执行” 表达：

t = Table('test', meta,

Column('abc', MyType, default=func.generate\_new\_value(), primary\_key=**True**))

Where above, when [Table.insert()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.insert" \o "sqlalchemy.schema.Table.insert) is used, the func.generate\_new\_value() expression will be pre-executed in the context of a scalar SELECT statement, and the new value will be applied to the subsequent INSERT, while at the same time being made available to the [ResultProxy.inserted\_primary\_key](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.inserted_primary_key" \o "sqlalchemy.engine.ResultProxy.inserted_primary_key) attribute.

在上面的位置，当使用Table.insert() 时，func.generate\_new\_value() 表达式将在标量SELECT语句的上下文中预先执行，并且新值将被应用于后续的INSERT，同时 被提供给ResultProxy.inserted\_primary\_key属性。

3.3.6 Defining Sequences

SQLAlchemy represents database sequences using the [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) object, which is considered to be a special case of "column default". It only has an effect on databases which have explicit support for sequences, which currently includes PostgreSQL, Oracle, and Firebird. The [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) object is otherwise ignored.

SQLAlchemy表示使用Sequence对象的数据库序列，它被认为是“column default”的特殊情况。 它只对具有明确支持序列的数据库产生影响，目前它们包括PostgreSQL，Oracle和Firebird。 否则将忽略Sequence对象。

The [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) may be placed on any column as a "default" generator to be used during INSERT operations, and can also be configured to fire off during UPDATE operations if desired. It is most commonly used in conjunction with a single integer primary key column:

序列可以放置在任何列作为在INSERT操作期间使用的“默认”生成器，并且还可以被配置为在UPDATE操作期间按需要触发。 它最常用于单个整数主键列：

table = Table("cartitems", meta,

Column(

"cart\_id",

Integer,

Sequence('cart\_id\_seq', metadata=meta), primary\_key=**True**),

Column("description", String(40)),

Column("createdate", DateTime()))

Where above, the table "cartitems" is associated with a sequence named "cart\_id\_seq". When INSERT statements take place for "cartitems", and no value is passed for the "cart\_id" column, the "cart\_id\_seq" sequence will be used to generate a value. Typically, the sequence function is embedded in the INSERT statement, which is combined with RETURNING so that the newly generated value can be returned to the Python code:

在上方，“cartitems”表与名为“cart\_id\_seq”的序列相关联。 当“cartitems”发生INSERT语句时，“cart\_id”列不会传递任何值，“cart\_id\_seq”序列将用于生成一个值。 通常，序列函数嵌入到INSERT语句中，该语句与RETURNING相结合，以便将新生成的值返回给Python代码：

INSERT INTO cartitems (cart\_id, description, createdate)VALUES (next\_val(cart\_id\_seq), 'some description', '2015-10-15 12:00:15')RETURNING cart\_id

When the [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) is associated with a [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) as its ****Python-side**** default generator, the [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) will also be subject to "CREATE SEQUENCE" and "DROP SEQUENCE" DDL when similar DDL is emitted for the owning [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table). This is a limited scope convenience feature that does not accommodate for inheritance of other aspects of the [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData), such as the default schema. Therefore, it is best practice that for a [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) which is local to a certain [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) / [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table), that it be explicitly associated with the [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) using the [Sequence.metadata](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence.params.metadata" \o "sqlalchemy.schema.Sequence) parameter. See the section [Associating a Sequence with the MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sequence-metadata) for more background on this.

当序列与一个列作为其Python端默认生成器相关联时，当发布相似的DDL时，序列也将受到“CREATE SEQUENCE”和“DROP SEQUENCE”DDL的约束。 这是一个有限范围的便利功能，不适用于MetaData的其他方面的继承，例如默认模式。 因此，最好的做法是对于特定列/表的本地的序列，它使用Sequence.metadata参数与MetaData显式关联。 有关更多背景信息，请参阅将序列与MetaData关联的部分。

### Associating a Sequence on a SERIAL column

PostgreSQL's SERIAL datatype is an auto-incrementing type that implies the implicit creation of a PostgreSQL sequence when CREATE TABLE is emitted. If a [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column)specifies an explicit [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) object which also specifies a true value for the [Sequence.optional](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence.params.optional" \o "sqlalchemy.schema.Sequence) boolean flag, the [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) will not take effect under PostgreSQL, and the SERIAL datatype will proceed normally. Instead, the [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) will only take effect when used against other sequence-supporting databases, currently Oracle and Firebird.

PostgreSQL的SERIAL数据类型是一种自动递增类型，意味着在发布CREATE TABLE时隐式创建PostgreSQL序列。 如果一个Columnspecified一个显式Sequence对象，该对象也为Sequence.optional布尔标志指定了一个真正的值，则Sequence不会在PostgreSQL下生效，SERIAL数据类型将正常进行。 相反，序列仅在与其他支持序列的数据库（目前为Oracle和Firebird）一起使用时生效。

### Executing a Sequence Standalone

A SEQUENCE is a first class schema object in SQL and can be used to generate values independently in the database. If you have a [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) object, it can be invoked with its "next value" instruction by passing it directly to a SQL execution method:

SEQUENCE是SQL中的第一类模式对象，可用于在数据库中独立生成值。 如果您有一个Sequence对象，则可以通过将其“下一个值”指令直接传递给SQL执行方法来调用它。

**with** my\_engine.connect() **as** conn:

seq = Sequence('some\_sequence')

nextid = conn.execute(seq)

In order to embed the "next value" function of a [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) inside of a SQL statement like a SELECT or INSERT, use the [Sequence.next\_value()](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence.next_value" \o "sqlalchemy.schema.Sequence.next_value) method, which will render at statement compilation time a SQL function that is appropriate for the target backend:

为了在SQL语句（如SELECT或INSERT）中嵌入Sequence的“next value”函数，请使用Sequence.next\_value() 方法，它将在语句编译时渲染适合于目标后端的SQL函数：

**>>>** my\_seq = Sequence('some\_sequence')

**>>>** stmt = select([my\_seq.next\_value()])

**>>>** print stmt.compile(dialect=postgresql.dialect())SELECT nextval('some\_sequence') AS next\_value\_1

### Associating a Sequence with the MetaData

For many years, the SQLAlchemy documentation referred to the example of associating a [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) with a table as follows:

多年来，SQLAlchemy文档参考了将Sequence与表相关联的示例如下：

table = Table("cartitems", meta,

Column("cart\_id", Integer, Sequence('cart\_id\_seq'),

primary\_key=**True**),

Column("description", String(40)),

Column("createdate", DateTime()))

While the above is a prominent idiomatic pattern, it is recommended that the [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) in most cases be explicitly associated with the [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData), using the [Sequence.metadata](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence.params.metadata" \o "sqlalchemy.schema.Sequence) parameter:

虽然上述是一个突出的惯用模式，但建议使用Sequence.metadata参数，大多数情况下，Sequence与MetaData明确关联：

table = Table("cartitems", meta,

Column(

"cart\_id",

Integer,

Sequence('cart\_id\_seq', metadata=meta), primary\_key=**True**),

Column("description", String(40)),

Column("createdate", DateTime()))

The [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) object is a first class schema construct that can exist independently of any table in a database, and can also be shared among tables. Therefore SQLAlchemy does not implicitly modify the [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) when it is associated with a [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) object as either the Python-side or server-side default generator. While the CREATE SEQUENCE / DROP SEQUENCE DDL is emitted for a [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) defined as a Python side generator at the same time the table itself is subject to CREATE or DROP, this is a convenience feature that does not imply that the [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) is fully associated with the [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) object.

Sequence对象是可以独立于数据库中的任何表存在的第一类模式构造，也可以在表之间共享。 因此，当与Column对象关联时，SQLAlchemy不会将Sequence与.NET端或服务器端的默认生成器相关联。 虽然CREATE SEQUENCE / DROP SEQUENCE DDL被发布为定义为Python边生成器的序列，但表本身也受到CREATE或DROP的约束，这是一个方便的功能，并不意味着Sequence与元数据对象。

Explicitly associating the [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) with [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) allows for the following behaviors:

显式地将序列与MetaData相关联允许以下行为：

* The [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) will inherit the [MetaData.schema](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData.params.schema" \o "sqlalchemy.schema.MetaData) parameter specified to the target [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData), which affects the production of CREATE / DROP DDL, if any.序列将继承指定给目标MetaData的MetaData.schema参数，影响CREATE / DROP DDL的生成（如果有）。
* The [Sequence.create()](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence.create" \o "sqlalchemy.schema.Sequence.create) and [Sequence.drop()](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence.drop" \o "sqlalchemy.schema.Sequence.drop) methods automatically use the engine bound to the [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) object, if any.Sequence.create() 和Sequence.drop() 方法自动使用绑定到MetaData对象的引擎（如果有的话）。
* 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) and [MetaData.drop\_all()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData.drop_all" \o "sqlalchemy.schema.MetaData.drop_all) methods will emit CREATE / DROP for this [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence), even if the [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) is not associated with any [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) / [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) that's a member of this [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData).MetaData.create\_all() 和MetaData.drop\_all() 方法将为此序列发出CREATE / DROP，即使该序列与此MetaData的任何表/列不相关联。

Since the vast majority of cases that deal with [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) expect that [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) to be fully "owned" by the assocated [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) and that options like default schema are propagated, setting the [Sequence.metadata](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence.params.metadata" \o "sqlalchemy.schema.Sequence) parameter should be considered a best practice.

由于处理Sequence的绝大多数情况都希望Sequence被完全“关联”到表中，并且默认模式等选项也被传播，所以设置Sequence.metadata参数应该被认为是最佳做法。

### Associating a Sequence as the Server Side Default

The preceding sections illustrate how to associate a [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) with a [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) as the ****Python side default generator****:

前面的部分说明了如何将一个序列与一个列相关联作为Python侧的默认生成器：

Column(

"cart\_id", Integer, Sequence('cart\_id\_seq', metadata=meta),

primary\_key=**True**)

In the above case, the [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) will automatically be subject to CREATE SEQUENCE / DROP SEQUENCE DDL when the related [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) is subject to CREATE / DROP. However, the sequence will ****not**** be present as the server-side default for the column when CREATE TABLE is emitted.

在上述情况下，当相关表格受到CREATE / DROP的影响时，序列将自动受到CREATE SEQUENCE / DROP SEQUENCE DDL的约束。 但是，当发出CREATE TABLE时，该序列将不会作为列的服务器端默认存在。

If we want the sequence to be used as a server-side default, meaning it takes place even if we emit INSERT commands to the table from the SQL command line, we can use the [Column.server\_default](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column.params.server_default" \o "sqlalchemy.schema.Column) parameter in conjunction with the value-generation function of the sequence, available from the [Sequence.next\_value()](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence.next_value" \o "sqlalchemy.schema.Sequence.next_value)method. Below we illustrate the same [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) being associated with the [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) both as the Python-side default generator as well as the server-side default generator:

如果我们希望将序列用作服务器端默认，即使我们从SQL命令行向表发出INSERT命令，也可以使用Column.server\_default参数与值生成函数 的序列，可从Sequence.next\_value() 方法获得。 下面我们将将与Column相关联的序列与Python侧默认生成器以及服务器端默认生成器进行说明：

cart\_id\_seq = Sequence('cart\_id\_seq', metadata=meta)table = Table("cartitems", meta,

Column(

"cart\_id", Integer, cart\_id\_seq,

server\_default=cart\_id\_seq.next\_value(), primary\_key=**True**),

Column("description", String(40)),

Column("createdate", DateTime()))

or with the ORM:

**class** **CartItem**(Base):

\_\_tablename\_\_ = 'cartitems'

cart\_id\_seq = Sequence('cart\_id\_seq', metadata=Base.metadata)

cart\_id = Column(

Integer, cart\_id\_seq,

server\_default=cart\_id\_seq.next\_value(), primary\_key=**True**)

description = Column(String(40))

createdate = Column(DateTime)

When the "CREATE TABLE" statement is emitted, on PostgreSQL it would be emitted as:

CREATE TABLE cartitems (

cart\_id INTEGER DEFAULT nextval('cart\_id\_seq') NOT NULL,

description VARCHAR(40),

createdate TIMESTAMP WITHOUT TIME ZONE,

PRIMARY KEY (cart\_id))

Placement of the [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) in both the Python-side and server-side default generation contexts ensures that the "primary key fetch" logic works in all cases. Typically, sequence-enabled databases also support RETURNING for INSERT statements, which is used automatically by SQLAlchemy when emitting this statement. However if RETURNING is not used for a particular insert, then SQLAlchemy would prefer to "pre-execute" the sequence outside of the INSERT statement itself, which only works if the sequence is included as the Python-side default generator function.

序列在Python侧和服务器端默认生成上下文中的放置可确保“主键提取”逻辑在所有情况下都可以工作。 通常，启用顺序的数据库还支持对INSERT语句进行RETURNING，由SQLAlchemy在发出此语句时自动使用。 但是，如果RETURNING不用于特定的插入，那么SQLAlchemy更喜欢“预执行”INSERT语句本身之外的序列，只有当该序列作为Python侧的默认生成函数包含时，它才起作用。

The example also associates the [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) with the enclosing [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) directly, which again ensures that the [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) is fully associated with the parameters of the [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) collection including the default schema, if any.

该示例还将Sequence与包含的MetaData直接相关联，这再次确保Sequence与MetaData集合的参数完全关联，包括默认模式（如果有）。

**See also**

[Sequences/SERIAL](http://docs.sqlalchemy.org/en/rel_1_1/dialects/postgresql.html" \l "postgresql-sequences) - in the PostgreSQL dialect documentation

[RETURNING Support](http://docs.sqlalchemy.org/en/rel_1_1/dialects/oracle.html" \l "oracle-returning) - in the Oracle dialect documentation

3.3.7 Default Objects API

*class*sqlalchemy.schema.**ColumnDefault**(*arg*, *\*\*kwargs*)

Bases: [sqlalchemy.schema.DefaultGenerator](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.DefaultGenerator" \o "sqlalchemy.schema.DefaultGenerator)

A plain default value on a column.

[ColumnDefault](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.ColumnDefault" \o "sqlalchemy.schema.ColumnDefault) can be passed positionally as well.

列上的普通默认值。

This could correspond to a constant, a callable function, or a SQL clause.

这可以对应于常量，可调用函数或SQL子句。

[ColumnDefault](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.ColumnDefault" \o "sqlalchemy.schema.ColumnDefault) is generated automatically whenever the default, onupdate arguments of [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) are used. A

每当使用默认的onupdate参数时，ColumnDefault会自动生成。 ColumnDefault也可以在位置上传递。

For example, the following:

Column('foo', Integer, default=50)

Is equivalent to:

Column('foo', Integer, ColumnDefault(50))

*class*sqlalchemy.schema.**DefaultClause**(*arg*, *for\_update=False*, *\_reflected=False*)

Bases: [sqlalchemy.schema.FetchedValue](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.FetchedValue" \o "sqlalchemy.schema.FetchedValue)

A DDL-specified DEFAULT column value.

一个DDL指定的DEFAULT列值。

[DefaultClause](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.DefaultClause" \o "sqlalchemy.schema.DefaultClause) is a [FetchedValue](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.FetchedValue" \o "sqlalchemy.schema.FetchedValue) that also generates a "DEFAULT" clause when "CREATE TABLE" is emitted.

DefaultClause是一个FetchedValue，当发出“CREATE TABLE”时也会生成一个“DEFAULT”子句。

[DefaultClause](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.DefaultClause" \o "sqlalchemy.schema.DefaultClause) is generated automatically whenever the server\_default, server\_onupdate arguments of [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) are used. A [DefaultClause](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.DefaultClause" \o "sqlalchemy.schema.DefaultClause)can be passed positionally as well.

每当使用列的server\_default，server\_onupdate参数时，将自动生成DefaultClause。 一个DefaultClausecan也可以通过位置。

For example, the following:

Column('foo', Integer, server\_default="50")

Is equivalent to:

Column('foo', Integer, DefaultClause("50"))

*class*sqlalchemy.schema.**DefaultGenerator**(*for\_update=False*)

Bases: sqlalchemy.schema.\_NotAColumnExpr, [sqlalchemy.schema.SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem)

Base class for column *default* values.

*class*sqlalchemy.schema.**FetchedValue**(*for\_update=False*)

Bases: sqlalchemy.schema.\_NotAColumnExpr, sqlalchemy.sql.expression.SchemaEventTarget

A marker for a transparent database-side default.

透明数据库端默认的标记。

Use [FetchedValue](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.FetchedValue" \o "sqlalchemy.schema.FetchedValue) when the database is configured to provide some automatic default for a column.

当数据库配置为列提供一些自动默认值时，请使用FetchedValue。

E.g.:

Column('foo', Integer, FetchedValue())

Would indicate that some trigger or default generator will create a new value for the foo column during an INSERT.

将指示一些触发器或默认生成器将在INSERT期间为foo列创建一个新值。

**See also**

[Triggered Columns](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "triggered-columns)

*class*sqlalchemy.schema.**PassiveDefault**(*\*arg*, *\*\*kw*)

Bases: [sqlalchemy.schema.DefaultClause](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.DefaultClause" \o "sqlalchemy.schema.DefaultClause)

A DDL-specified DEFAULT column value.

*Deprecated since version 0.6:*[PassiveDefault](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.PassiveDefault" \o "sqlalchemy.schema.PassiveDefault) is deprecated. Use [DefaultClause](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.DefaultClause" \o "sqlalchemy.schema.DefaultClause).

*class*sqlalchemy.schema.**Sequence**(*name*, *start=None*, *increment=None*, *minvalue=None*, *maxvalue=None*, *nominvalue=None*, *nomaxvalue=None*, *cycle=None*, *schema=None*, *cache=None*, *order=None*, *optional=False*, *quote=None*, *metadata=None*, *quote\_schema=None*, *for\_update=False*)

Bases: [sqlalchemy.schema.DefaultGenerator](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.DefaultGenerator" \o "sqlalchemy.schema.DefaultGenerator)

Represents a named database sequence.

表示一个命名数据库序列。

The [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) object represents the name and configurational parameters of a database sequence. It also represents a construct that can be "executed" by a SQLAlchemy [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), rendering the appropriate "next value" function for the target database and returning a result.

Sequence对象表示数据库序列的名称和配置参数。 它还表示可以由SQLAlchemy Engine或Connection“执行”的结构，为目标数据库提供适当的“下一个值”函数并返回结果。

The [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) is typically associated with a primary key column:

序列通常与主键列相关联：

some\_table = Table(

'some\_table', metadata,

Column('id', Integer, Sequence('some\_table\_seq'),

primary\_key=**True**))

When CREATE TABLE is emitted for the above [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table), if the target platform supports sequences, a CREATE SEQUENCE statement will be emitted as well. For platforms that don't support sequences, the [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) construct is ignored.

当为上表发出CREATE TABLE时，如果目标平台支持序列，则也会发出CREATE SEQUENCE语句。 对于不支持序列的平台，将忽略Sequence构造。

**See also**

[CreateSequence](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.CreateSequence" \o "sqlalchemy.schema.CreateSequence)

[DropSequence](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.DropSequence" \o "sqlalchemy.schema.DropSequence)

**\_\_init\_\_**(*name*, *start=None*, *increment=None*, *minvalue=None*, *maxvalue=None*, *nominvalue=None*, *nomaxvalue=None*, *cycle=None*, *schema=None*, *cache=None*, *order=None*, *optional=False*, *quote=None*, *metadata=None*, *quote\_schema=None*, *for\_update=False*)

Construct a [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) object.

|  |  |
| --- | --- |
| **Parameters:** | * ****name**** – The name of the sequence. * ****start**** – the starting index of the sequence. This value is used when the CREATE SEQUENCE command is emitted to the database as the value of the "START WITH" clause. If None, the clause is omitted, which on most platforms indicates a starting value of 1. * ****increment**** – the increment value of the sequence. This value is used when the CREATE SEQUENCE command is emitted to the database as the value of the "INCREMENT BY" clause. If None, the clause is omitted, which on most platforms indicates an increment of 1. * ****minvalue –****the minimum value of the sequence. This value is used when the CREATE SEQUENCE command is emitted to the database as the value of the "MINVALUE" clause. If None, the clause is omitted, which on most platforms indicates a minvalue of 1 and -2^63-1 for ascending and descending sequences, respectively.序列的最小值。 当将CREATE SEQUENCE命令作为“MINVALUE”子句的值发送到数据库时，将使用此值。 如果没有，则省略该子句，大多数平台上分别表示升序和降序的小值为1和-2 ^ 63-1。   *New in version 1.0.7.*   * ****maxvalue –****the maximum value of the sequence. This value is used when the CREATE SEQUENCE command is emitted to the database as the value of the "MAXVALUE" clause. If None, the clause is omitted, which on most platforms indicates a maxvalue of 2^63-1 and -1 for ascending and descending sequences, respectively.序列的最大值。 当将CREATE SEQUENCE命令作为“MAXVALUE”子句的值发送到数据库时，将使用此值。 如果None，则省略该子句，大多数平台上分别表示升序和降序的最大值为2 ^ 63-1，-1。   *New in version 1.0.7.*   * ****nominvalue –****no minimum value of the sequence. This value is used when the CREATE SEQUENCE command is emitted to the database as the value of the "NO MINVALUE" clause. If None, the clause is omitted, which on most platforms indicates a minvalue of 1 and -2^63-1 for ascending and descending sequences, respectively.没有序列的最小值。 当将CREATE SEQUENCE命令作为“NO MINVALUE”子句的值发送到数据库时，将使用此值。 如果没有，则省略该子句，大多数平台上分别表示升序和降序的小值为1和-2 ^ 63-1。   *New in version 1.0.7.*   * ****nomaxvalue –****no maximum value of the sequence. This value is used when the CREATE SEQUENCE command is emitted to the database as the value of the "NO MAXVALUE" clause. If None, the clause is omitted, which on most platforms indicates a maxvalue of 2^63-1 and -1 for ascending and descending sequences, respectively.没有序列的最大值。 当将CREATE SEQUENCE命令作为“NO MAXVALUE”子句的值发送到数据库时，将使用此值。 如果None，则省略该子句，大多数平台上分别表示升序和降序的最大值为2 ^ 63-1，-1。   *New in version 1.0.7.*   * ****cycle –****allows the sequence to wrap around when the maxvalue or minvalue has been reached by an ascending or descending sequence respectively. This value is used when the CREATE SEQUENCE command is emitted to the database as the "CYCLE" clause. If the limit is reached, the next number generated will be the minvalue or maxvalue, respectively. If cycle=False (the default) any calls to nextval after the sequence has reached its maximum value will return an error.当分别以升序或降序达到最大值或最小值时，允许序列循环。 当将CREATE SEQUENCE命令作为“CYCLE”子句发送到数据库时，将使用此值。 如果达到极限，则生成的下一个数字将分别为最小值或最大值。 如果cycle = False（默认值）在序列达到其最大值后对nextval的任何调用都将返回错误。   *New in version 1.0.7.*   * ****schema**** – Optional schema name for the sequence, if located in a schema other than the default. The rules for selecting the schema name when a [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) is also present are the same as that of [Table.schema](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.params.schema" \o "sqlalchemy.schema.Table).序列的可选模式名称，如果位于不同于默认模式的模式中。 当MetaData也存在时，选择模式名称的规则与Table.schema相同。 * ****cache –****optional integer value; number of future values in the sequence which are calculated in advance. Renders the CACHE keyword understood by Oracle and PostgreSQL.可选整数值; 预先计算的序列中未来值的数量。 呈现Oracle和PostgreSQL理解的CACHE关键字。   *New in version 1.1.12.*   * ****order –****optional boolean value; if true, renders the ORDER keyword, understood by Oracle, indicating the sequence is definitively ordered. May be necessary to provide deterministic ordering using Oracle RAC.可选布尔值; 如果为true，则呈现由Oracle了解的ORDER关键字，表明序列已被明确排序。 可能需要使用Oracle RAC提供确定性的排序。   *New in version 1.1.12.*   * ****optional**** – boolean value, when True, indicates that this [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) object only needs to be explicitly generated on backends that don't provide another way to generate primary key identifiers. Currently, it essentially means, "don't create this sequence on the PostgreSQL backend, where the SERIAL keyword creates a sequence for us automatically".布尔值，当为True时，表示此Sequence对象仅需要在不提供另一种方式生成主键标识符的后端上显式生成。 目前，它基本上意味着“不要在PostgreSQL后端创建此序列，其中SERIAL关键字自动为我们创建一个序列”。 * ****quote**** – boolean value, when True or False, explicitly forces quoting of the schema name on or off. When left at its default of None, normal quoting rules based on casing and reserved words take place.布尔值，当为True或False时，显式地强制引用模式名称打开或关闭。 当默认为无，基于套管和保留字的正常引用规则发生。 * ****quote\_schema**** – set the quoting preferences for the schema name.设置模式名称的引用首选项。 * ****metadata –****optional [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) object which this [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) will be associated with. A [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) that is associated with a [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) gains the following capabilities:此序列将与之相关联的可选MetaData对象。 与MetaData相关联的序列获得以下功能：   + The [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) will inherit the [MetaData.schema](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData.params.schema" \o "sqlalchemy.schema.MetaData) parameter specified to the target [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData), which affects the production of CREATE / DROP DDL, if any.   + The [Sequence.create()](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence.create" \o "sqlalchemy.schema.Sequence.create) and [Sequence.drop()](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence.drop" \o "sqlalchemy.schema.Sequence.drop) methods automatically use the engine bound to the [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) object, if any.   + 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) and [MetaData.drop\_all()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData.drop_all" \o "sqlalchemy.schema.MetaData.drop_all) methods will emit CREATE / DROP for this [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence), even if the [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence)is not associated with any [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) / [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) that's a member of this [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData).   The above behaviors can only occur if the [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) is explicitly associated with the [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) via this parameter.  **See also**  [Associating a Sequence with the MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sequence-metadata) - full discussion of the [Sequence.metadata](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence.params.metadata" \o "sqlalchemy.schema.Sequence) parameter.   * ****for\_update**** – Indicates this [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence), when associated with a [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column), should be invoked for UPDATE statements on that column's table, rather than for INSERT statements, when no value is otherwise present for that column in the statement.当该列与列关联时，表示此序列应该在该列的表上调用UPDATE语句，而不是在该语句中该列不存在值的情况下用于INSERT语句。 |

**create**(*bind=None*, *checkfirst=True*)

Creates this sequence in the database.

**drop**(*bind=None*, *checkfirst=True*)

Drops this sequence from the database.

**next\_value**()

Return a [next\_value](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.next_value" \o "sqlalchemy.sql.functions.next_value) function element which will render the appropriate increment function for this [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) within any SQL expression.

## 3.4 Defining Constraints and Indexes

This section will discuss SQL [constraints](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-constraints) and indexes. In SQLAlchemy the key classes include [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint) and [Index](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.Index" \o "sqlalchemy.schema.Index).

3.4.1 Defining Foreign Keys

A *foreign key* in SQL is a table-level construct that constrains one or more columns in that table to only allow values that are present in a different set of columns, typically but not always located on a different table. We call the columns which are constrained the *foreign key* columns and the columns which they are constrained towards the *referenced* columns. The referenced columns almost always define the primary key for their owning table, though there are exceptions to this. The foreign key is the "joint" that connects together pairs of rows which have a relationship with each other, and SQLAlchemy assigns very deep importance to this concept in virtually every area of its operation.

In SQLAlchemy as well as in DDL, foreign key constraints can be defined as additional attributes within the table clause, or for single-column foreign keys they may optionally be specified within the definition of a single column. The single column foreign key is more common, and at the column level is specified by constructing a [ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey) object as an argument to a [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) object:

user\_preference = Table('user\_preference', metadata,

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

Column('user\_id', Integer, ForeignKey("user.user\_id"), nullable=**False**),

Column('pref\_name', String(40), nullable=**False**),

Column('pref\_value', String(100)))

Above, we define a new table user\_preference for which each row must contain a value in the user\_id column that also exists in the user table's user\_idcolumn.

The argument to [ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey) is most commonly a string of the form *<tablename>.<columnname>*, or for a table in a remote schema or "owner" of the form *<schemaname>.<tablename>.<columnname>*. It may also be an actual [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) object, which as we'll see later is accessed from an existing [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object via its ccollection:

ForeignKey(user.c.user\_id)

The advantage to using a string is that the in-python linkage between user and user\_preference is resolved only when first needed, so that table objects can be easily spread across multiple modules and defined in any order.

Foreign keys may also be defined at the table level, using the [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint) object. This object can describe a single- or multi-column foreign key. A multi-column foreign key is known as a *composite* foreign key, and almost always references a table that has a composite primary key. Below we define a table invoice which has a composite primary key:

invoice = Table('invoice', metadata,

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

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

Column('description', String(60), nullable=**False**))

And then a table invoice\_item with a composite foreign key referencing invoice:

invoice\_item = Table('invoice\_item', metadata,

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

Column('item\_name', String(60), nullable=**False**),

Column('invoice\_id', Integer, nullable=**False**),

Column('ref\_num', Integer, nullable=**False**),

ForeignKeyConstraint(['invoice\_id', 'ref\_num'], ['invoice.invoice\_id', 'invoice.ref\_num']))

It's important to note that the [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint) is the only way to define a composite foreign key. While we could also have placed individual [ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey)objects on both the invoice\_item.invoice\_id and invoice\_item.ref\_num columns, SQLAlchemy would not be aware that these two values should be paired together - it would be two individual foreign key constraints instead of a single composite foreign key referencing two columns.

### Creating/Dropping Foreign Key Constraints via ALTER

The behavior we've seen in tutorials and elsewhere involving foreign keys with DDL illustrates that the constraints are typically rendered "inline" within the CREATE TABLE statement, such as:

**CREATE** **TABLE** addresses (

id INTEGER **NOT** **NULL**,

user\_id INTEGER,

email\_address VARCHAR **NOT** **NULL**,

**PRIMARY** **KEY** (id),

**CONSTRAINT** user\_id\_fk **FOREIGN** **KEY**(user\_id) **REFERENCES** users (id))

The CONSTRAINT .. FOREIGN KEY directive is used to create the constraint in an "inline" fashion within the CREATE TABLE definition.

CONSTRAINT .. FOREIGN KEY指令用于在CREATE TABLE定义中以“内联”方式创建约束。 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) and [MetaData.drop\_all()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData.drop_all" \o "sqlalchemy.schema.MetaData.drop_all) methods do this by default, using a topological sort of all the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects involved such that tables are created and dropped in order of their foreign key dependency (this sort is also available via the [MetaData.sorted\_tables](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData.sorted_tables" \o "sqlalchemy.schema.MetaData.sorted_tables) accessor).

TheMetaData.create\_all() 和MetaData.drop\_all() 方法默认情况下使用所涉及的所有表对象的拓扑排序，以便按照其外键依赖性的顺序创建和删除表（此类也可通过MetaData.sorted\_tables访问）。

This approach can't work when two or more foreign key constraints are involved in a "dependency cycle", where a set of tables are mutually dependent on each other, assuming the backend enforces foreign keys (always the case except on SQLite, MySQL/MyISAM). The methods will therefore break out constraints in such a cycle into separate ALTER statements, on all backends other than SQLite which does not support most forms of ALTER. Given a schema like:

当两个或多个外键约束涉及“依赖关系周期”时，这种方法无法工作，其中一组表彼此相互依赖，假设后端强制执行外键（除了SQLite，MySQL /MyISAM数据）。 因此，这些方法会将这种循环中的约束分解为单独的ALTER语句，除了不支持大多数形式的ALTER的SQLite之外的所有后端。 给出一个模式，如：

node = Table(

'node', metadata,

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

Column(

'primary\_element', Integer,

ForeignKey('element.element\_id')

))

element = Table(

'element', metadata,

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

Column('parent\_node\_id', Integer),

ForeignKeyConstraint(

['parent\_node\_id'], ['node.node\_id'],

name='fk\_element\_parent\_node\_id'

))

When we call upon [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) on a backend such as the PostgreSQL backend, the cycle between these two tables is resolved and the constraints are created separately:

当我们在后端（如PostgreSQL后端）上调用MetaData.create\_all() 时，会解决这两个表之间的循环，并分别创建约束：

**>>> with** engine.connect() **as** conn:

**...**  metadata.create\_all(conn, checkfirst=False)

CREATE TABLE element (

element\_id SERIAL NOT NULL,

parent\_node\_id INTEGER,

PRIMARY KEY (element\_id)

)

CREATE TABLE node (

node\_id SERIAL NOT NULL,

primary\_element INTEGER,

PRIMARY KEY (node\_id)

)

ALTER TABLE element ADD CONSTRAINT fk\_element\_parent\_node\_id

FOREIGN KEY(parent\_node\_id) REFERENCES node (node\_id)

ALTER TABLE node ADD FOREIGN KEY(primary\_element)

REFERENCES element (element\_id)

In order to emit DROP for these tables, the same logic applies, however note here that in SQL, to emit DROP CONSTRAINT requires that the constraint has a name. In the case of the 'node' table above, we haven't named this constraint; the system will therefore attempt to emit DROP for only those constraints that are named:

为了为这些表发出DROP，相同的逻辑适用，但是请注意，在SQL中，要发出DROP CONSTRAINT要求约束具有一个名称。 在上面的'node'表的情况下，我们没有命名这个约束; 因此，系统将尝试仅针对命名的约束发出DROP：

**>>> with** engine.connect() **as** conn:

**...**  metadata.drop\_all(conn, checkfirst=False)

ALTER TABLE element DROP CONSTRAINT fk\_element\_parent\_node\_id

DROP TABLE node

DROP TABLE element

In the case where the cycle cannot be resolved, such as if we hadn't applied a name to either constraint here, we will receive the following error:

在循环无法解决的情况下，例如如果我们在这里没有应用任何约束的名称，我们将收到以下错误：

sqlalchemy.exc.CircularDependencyError: Can't sort tables for DROP;an unresolvable foreign key dependency exists between tables:element, node. Please ensure that the ForeignKey **and** ForeignKeyConstraintobjects involved **in** the cycle have names so that they can be droppedusing DROP CONSTRAINT.

This error only applies to the DROP case as we can emit "ADD CONSTRAINT" in the CREATE case without a name; the database typically assigns one automatically.

此错误仅适用于DROP案例，因为我们可以在没有名称的情况下在CREATE案例中发出“ADD CONSTRAINT”; 数据库通常会自动分配一个。

The [ForeignKeyConstraint.use\_alter](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint.params.use_alter" \o "sqlalchemy.schema.ForeignKeyConstraint) and [ForeignKey.use\_alter](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey.params.use_alter" \o "sqlalchemy.schema.ForeignKey) keyword arguments can be used to manually resolve dependency cycles. We can add this flag only to the 'element' table as follows:

ForeignKeyConstraint.use\_alter和ForeignKey.use\_alter关键字参数可用于手动解析依赖关系周期。 我们可以将此标志仅添加到“元素”表中，如下所示：

element = Table(

'element', metadata,

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

Column('parent\_node\_id', Integer),

ForeignKeyConstraint(

['parent\_node\_id'], ['node.node\_id'],

use\_alter=**True**, name='fk\_element\_parent\_node\_id'

))

in our CREATE DDL we will see the ALTER statement only for this constraint, and not the other one:

**>>> with** engine.connect() **as** conn:

**...**  metadata.create\_all(conn, checkfirst=False)

CREATE TABLE element (

element\_id SERIAL NOT NULL,

parent\_node\_id INTEGER,

PRIMARY KEY (element\_id)

)

CREATE TABLE node (

node\_id SERIAL NOT NULL,

primary\_element INTEGER,

PRIMARY KEY (node\_id),

FOREIGN KEY(primary\_element) REFERENCES element (element\_id)

)

ALTER TABLE element ADD CONSTRAINT fk\_element\_parent\_node\_id

FOREIGN KEY(parent\_node\_id) REFERENCES node (node\_id)

[ForeignKeyConstraint.use\_alter](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint.params.use_alter" \o "sqlalchemy.schema.ForeignKeyConstraint) and [ForeignKey.use\_alter](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey.params.use_alter" \o "sqlalchemy.schema.ForeignKey), when used in conjunction with a drop operation, will require that the constraint is named, else an error like the following is generated:

ForeignKeyConstraint.use\_alter和ForeignKey.use\_alter与drop操作结合使用时，将要求约束被命名，否则会生成如下的错误：

sqlalchemy.exc.CompileError:

Can't emit DROP CONSTRAINT for constraint

ForeignKeyConstraint(...);

it has no name

*Changed in version 1.0.0:*- The DDL system invoked by [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) and [MetaData.drop\_all()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData.drop_all" \o "sqlalchemy.schema.MetaData.drop_all) will now automatically resolve mutually depdendent foreign keys between tables declared by [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint) and [ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey) objects, without the need to explicitly set the [ForeignKeyConstraint.use\_alter](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint.params.use_alter" \o "sqlalchemy.schema.ForeignKeyConstraint) flag.

在版本1.0.0中更改： - MetaData.create\_all() 和MetaData.drop\_all() 调用的DDL系统现在将自动解决ForeignKeyConstraint和ForeignKey对象声明的表之间的相互间断的外键，而不需要显式设置ForeignKeyConstraint。use\_alter标志。

*Changed in version 1.0.0:*- The [ForeignKeyConstraint.use\_alter](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint.params.use_alter" \o "sqlalchemy.schema.ForeignKeyConstraint) flag can be used with an un-named constraint; only the DROP operation will emit a specific error when actually called upon.

在版本1.0.0中更改： - ForeignKeyConstraint.use\_alter标志可以与未命名约束一起使用; 只有DROP操作会在实际调用时发出特定的错误。

**See also**

[Configuring Constraint Naming Conventions](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "constraint-naming-conventions)

[sort\_tables\_and\_constraints()](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.sort_tables_and_constraints" \o "sqlalchemy.schema.sort_tables_and_constraints)

### ON UPDATE and ON DELETE

Most databases support *cascading* of foreign key values, that is the when a parent row is updated the new value is placed in child rows, or when the parent row is deleted all corresponding child rows are set to null or deleted. In data definition language these are specified using phrases like "ON UPDATE CASCADE", "ON DELETE CASCADE", and "ON DELETE SET NULL", corresponding to foreign key constraints. The phrase after "ON UPDATE" or "ON DELETE" may also other allow other phrases that are specific to the database in use. The [ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey) and [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint) objects support the generation of this clause via the onupdate and ondelete keyword arguments. The value is any string which will be output after the appropriate "ON UPDATE" or "ON DELETE" phrase:

大多数数据库支持外键值的级联，即更新父行时，新值将放置在子行中，或者当删除父行时，所有相应的子行都将设置为空或删除。 在数据定义语言中，这些使用“ON UPDATE CASCADE”，“ON DELETE CASCADE”和“ON DELETE SET NULL”这样的短语指定，对应于外键约束。 “ON UPDATE”或“ON DELETE”之后的短语还可以允许在使用中特定于数据库的其他短语。 ForeignKey和ForeignKeyConstraint对象支持通过onupdate和ondelete关键字参数生成此子句。 该值是在适当的“ON UPDATE”或“ON DELETE”短语后输出的任何字符串：

child = Table('child', meta,

Column('id', Integer,

ForeignKey('parent.id', onupdate="CASCADE", ondelete="CASCADE"),

primary\_key=**True**

))

composite = Table('composite', meta,

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

Column('rev\_id', Integer),

Column('note\_id', Integer),

ForeignKeyConstraint(

['rev\_id', 'note\_id'],

['revisions.id', 'revisions.note\_id'],

onupdate="CASCADE", ondelete="SET NULL"

))

Note that these clauses are not supported on SQLite, and require InnoDB tables when used with MySQL. They may also not be supported on other databases.

请注意，SQLite不支持这些子句，与MySQL一起使用时需要使用InnoDB表。 也可能不支持其他数据库。

3.4.2 UNIQUE Constraint

Unique constraints can be created anonymously on a single column using the unique keyword on [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column). Explicitly named unique constraints and/or those with multiple columns are created via the [UniqueConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.UniqueConstraint" \o "sqlalchemy.schema.UniqueConstraint) table-level construct.

可以使用Column上的唯一关键字在单个列上匿名创建唯一约束。 通过UniqueConstraint表级结构创建明确命名的唯一约束和/或具有多个列的约束。

**from** **sqlalchemy** **import** UniqueConstraint

meta = MetaData()mytable = Table('mytable', meta,

*# per-column anonymous unique constraint*

Column('col1', Integer, unique=True),

Column('col2', Integer),

Column('col3', Integer),

*# explicit/composite unique constraint. 'name' is optional.*

UniqueConstraint('col2', 'col3', name='uix\_1')

)

### 3.4.3 CHECK Constraint

Check constraints can be named or unnamed and can be created at the Column or Table level, using the [CheckConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.CheckConstraint" \o "sqlalchemy.schema.CheckConstraint) construct. The text of the check constraint is passed directly through to the database, so there is limited "database independent" behavior. Column level check constraints generally should only refer to the column to which they are placed, while table level constraints can refer to any columns in the table.

检查约束可以命名或未命名，并且可以使用CheckConstraint构造在Column或Table级别创建。 检查约束的文本直接传递到数据库，因此有限的“数据库独立”行为。 列级检查约束通常仅应参考它们所在的列，而表级约束可以引用表中的任何列。

检查约束可以命名或未命名，并且可以使用CheckConstraint构造在Column或Table级别创建。 检查约束的文本直接传递到数据库，因此有限的“数据库独立”行为。 列级检查约束通常仅应参考它们所在的列，而表级约束可以引用表中的任何列。

Note that some databases do not actively support check constraints such as MySQL.

请注意，某些数据库不会主动支持诸如MySQL之类的检查约束。

**from** **sqlalchemy** **import** CheckConstraint

meta = MetaData()mytable = Table('mytable', meta,

*# per-column CHECK constraint*

Column('col1', Integer, CheckConstraint('col1>5')),

Column('col2', Integer),

Column('col3', Integer),

*# table level CHECK constraint. 'name' is optional.*

CheckConstraint('col2 > col3 + 5', name='check1')

)

mytable.create(engine)

CREATE TABLE mytable (

col1 INTEGER CHECK (col1>5),

col2 INTEGER,

col3 INTEGER,

CONSTRAINT check1 CHECK (col2 > col3 + 5)

)

### 3.4.4 PRIMARY KEY Constraint

The primary key constraint of any [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object is implicitly present, based on the [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects that are marked with the [Column.primary\_key](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column.params.primary_key" \o "sqlalchemy.schema.Column) flag. The [PrimaryKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.PrimaryKeyConstraint" \o "sqlalchemy.schema.PrimaryKeyConstraint) object provides explicit access to this constraint, which includes the option of being configured directly:

基于使用Column.primary\_key标记的Column对象，任何Table对象的主键约束是隐含存在的。 PrimaryKeyConstraint对象提供对此约束的显式访问，其中包括直接配置的选项：

**from** **sqlalchemy** **import** PrimaryKeyConstraint

my\_table = Table('mytable', metadata,

Column('id', Integer),

Column('version\_id', Integer),

Column('data', String(50)),

PrimaryKeyConstraint('id', 'version\_id', name='mytable\_pk')

)

**See also**

[PrimaryKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.PrimaryKeyConstraint" \o "sqlalchemy.schema.PrimaryKeyConstraint) - detailed API documentation.

3.4.5 Setting up Constraints when using the Declarative ORM Extension

The [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) is the SQLAlchemy Core construct that allows one to define table metadata, which among other things can be used by the SQLAlchemy ORM as a target to map a class. The [Declarative](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/index.html) extension allows the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object to be created automatically, given the contents of the table primarily as a mapping of [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects.

该表是SQLAlchemy Core结构，允许用户定义表元数据，其中可以由SQLAlchemy ORM用作映射类的目标。 声明式扩展允许自动创建Table对象，给定表的内容主要作为Column对象的映射。

To apply table-level constraint objects such as [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint) to a table defined using Declarative, use the \_\_table\_args\_\_ attribute, described at [Table Configuration](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/declarative/table_config.html" \l "declarative-table-args).

要将表级约束对象（如ForeignKeyConstraint）应用于使用Declarative定义的表，请使用表配置中描述的\_\_table\_args\_\_属性。

### 3.4.6 Configuring Constraint Naming Conventions

Relational databases typically assign explicit names to all constraints and indexes. In the common case that a table is created using CREATE TABLE where constraints such as CHECK, UNIQUE, and PRIMARY KEY constraints are produced inline with the table definition, the database usually has a system in place in which names are automatically assigned to these constraints, if a name is not otherwise specified. When an existing database table is altered in a database using a command such as ALTER TABLE, this command typically needs to specify explicit names for new constraints as well as be able to specify the name of an existing constraint that is to be dropped or modified.

关系数据库通常为所有约束和索引分配显式名称。在常见的情况下，使用CREATE TABLE创建表，其中诸如CHECK，UNIQUE和PRIMARY KEY约束之类的约束与表定义一起生成，数据库通常具有一个系统，其中名称被自动分配给这些约束，如果没有另外指定名称。当使用诸如ALTER TABLE的命令在数据库中更改现有数据库表时，此命令通常需要为新约束指定显式名称，并且能够指定要删除或修改的现有约束的名称。

Constraints can be named explicitly using the [Constraint.name](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.Constraint.params.name" \o "sqlalchemy.schema.Constraint) parameter, and for indexes the [Index.name](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.Index.params.name" \o "sqlalchemy.schema.Index) parameter. However, in the case of constraints this parameter is optional. There are also the use cases of using the [Column.unique](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column.params.unique" \o "sqlalchemy.schema.Column) and [Column.index](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column.params.index" \o "sqlalchemy.schema.Column) parameters which create [UniqueConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.UniqueConstraint" \o "sqlalchemy.schema.UniqueConstraint) and [Index](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.Index" \o "sqlalchemy.schema.Index)objects without an explicit name being specified.

约束可以使用Constraint.name参数和索引Index.name参数显式命名。但是，在约束的情况下，此参数是可选的。还有使用Column.unique和Column.index参数的用例，它们创建UniqueConstraint和Indexobject，而不指定明确的名称。

The use case of alteration of existing tables and constraints can be handled by schema migration tools such as [Alembic](http://alembic.zzzcomputing.com/). However, neither Alembic nor SQLAlchemy currently create names for constraint objects where the name is otherwise unspecified, leading to the case where being able to alter existing constraints means that one must reverse-engineer the naming system used by the relational database to auto-assign names, or that care must be taken to ensure that all constraints are named.

现有表和约束的更改的用例可以通过模式迁移工具（如Alembic）来处理。然而，Alembic和SQLAlchemy当前都没有为其他未指定名称的约束对象创建名称，导致可以更改现有约束的情况意味着必须对关系数据库使用的命名系统进行逆向工程以自动分配名称，或者必须小心确保所有约束都被命名。

In contrast to having to assign explicit names to all [Constraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.Constraint" \o "sqlalchemy.schema.Constraint) and [Index](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.Index" \o "sqlalchemy.schema.Index) objects, automated naming schemes can be constructed using events. This approach has the advantage that constraints will get a consistent naming scheme without the need for explicit name parameters throughout the code, and also that the convention takes place just as well for those constraints and indexes produced by the [Column.unique](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column.params.unique" \o "sqlalchemy.schema.Column) and [Column.index](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column.params.index" \o "sqlalchemy.schema.Column) parameters. As of SQLAlchemy 0.9.2 this event-based approach is included, and can be configured using the argument [MetaData.naming\_convention](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData.params.naming_convention" \o "sqlalchemy.schema.MetaData).

与必须为所有约束和索引对象分配显式名称相反，自动命名方案可以使用事件构建。这种方法的优点在于，约束将获得一致的命名方案，而不需要在整个代码中使用明确的名称参数，并且对于由Column.unique和Column.index参数生成的约束和索引，约定也是一样。 。从SQLAlchemy 0.9.2开始，包括了基于事件的方法，并且可以使用参数MetaData.naming\_convention进行配置。

[MetaData.naming\_convention](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData.params.naming_convention" \o "sqlalchemy.schema.MetaData) refers to a dictionary which accepts the [Index](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.Index" \o "sqlalchemy.schema.Index) class or individual [Constraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.Constraint" \o "sqlalchemy.schema.Constraint) classes as keys, and Python string templates as values. It also accepts a series of string-codes as alternative keys, "fk", "pk", "ix", "ck", "uq" for foreign key, primary key, index, check, and unique constraint, respectively. The string templates in this dictionary are used whenever a constraint or index is associated with this [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) object that does not have an existing name given (including one exception case where an existing name can be further embellished).

An example naming convention that suits basic cases is as follows:

MetaData.naming\_convention是指一个字典，它接受Index类或单个Constraint类作为键，Python字符串模板作为值。它还分别接受一系列字符串代码作为替代键，分别为外键，主键，索引，检查和唯一约束的"fk", "pk", "ix", "ck", "uq"。只要约束或索引与此MetaData对象相关联但不包含现有名称（包括一个可以进一步修饰现有名称的异常情况），则会使用此字典中的字符串模板。

适用于基本情况的示例命名约定如下：

convention = {

"ix": 'ix\_*%(column\_0\_label)s*',

"uq": "uq\_*%(table\_name)s*\_*%(column\_0\_name)s*",

"ck": "ck\_*%(table\_name)s*\_*%(constraint\_name)s*",

"fk": "fk\_*%(table\_name)s*\_*%(column\_0\_name)s*\_*%(referred\_table\_name)s*",

"pk": "pk\_*%(table\_name)s*"}

metadata = MetaData(naming\_convention=convention)

The above convention will establish names for all constraints within the target [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) collection. For example, we can observe the name produced when we create an unnamed [UniqueConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.UniqueConstraint" \o "sqlalchemy.schema.UniqueConstraint):

上述约定将为目标MetaData集合中的所有约束建立名称。 例如，我们可以观察到当我们创建一个未命名的UniqueConstraint时产生的名称：

**>>>** user\_table = Table('user', metadata,

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

**...**  Column('name', String(30), nullable=**False**),

**...**  UniqueConstraint('name')

**...** )

**>>>** list(user\_table.constraints)[1].name

'uq\_user\_name'

This same feature takes effect even if we just use the [Column.unique](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column.params.unique" \o "sqlalchemy.schema.Column) flag:

即使我们使用Column.unique标志，这个相同的功能也会生效：

**>>>** user\_table = Table('user', metadata,

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

**...**  Column('name', String(30), nullable=**False**, unique=**True**)

**...**  )

**>>>** list(user\_table.constraints)[1].name

'uq\_user\_name'

A key advantage to the naming convention approach is that the names are established at Python construction time, rather than at DDL emit time. The effect this has when using Alembic's --autogenerate feature is that the naming convention will be explicit when a new migration script is generated:

命名约定方法的一个关键优点是，这些名称是在Python构建时建立的，而不是在DDL发布时间。 使用Alembic的新特性时的效果是，当生成新的迁移脚本时，命名约定将是显式的：

**def** upgrade():

op.create\_unique\_constraint("uq\_user\_name", "user", ["name"])

The above "uq\_user\_name" string was copied from the [UniqueConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.UniqueConstraint" \o "sqlalchemy.schema.UniqueConstraint) object that --autogenerate located in our metadata.

上述“uq\_user\_name”字符串是从我们的元数据中的--autogenerate的UniqueConstraint对象复制的。

The default value for [MetaData.naming\_convention](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData.params.naming_convention" \o "sqlalchemy.schema.MetaData) handles the long-standing SQLAlchemy behavior of assigning a name to a [Index](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.Index" \o "sqlalchemy.schema.Index) object that is created using the [Column.index](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column.params.index" \o "sqlalchemy.schema.Column) parameter:

MetaData.naming\_convention的默认值处理了使用Column.index参数创建的Index对象分配名称的长期SQLAlchemy行为：

**>>> from** **sqlalchemy.sql.schema** **import** DEFAULT\_NAMING\_CONVENTION

**>>>** DEFAULT\_NAMING\_CONVENTIONimmutabledict({'ix': 'ix\_%(column\_0\_label)s'})

The tokens available include %(table\_name)s, %(referred\_table\_name)s, %(column\_0\_name)s, %(column\_0\_label)s, %(column\_0\_key)s, %(referred\_column\_0\_name)s, and %(constraint\_name)s; the documentation for [MetaData.naming\_convention](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData.params.naming_convention" \o "sqlalchemy.schema.MetaData) describes each individually. New tokens can also be added, by specifying an additional token and a callable within the naming\_convention dictionary. For example, if we wanted to name our foreign key constraints using a GUID scheme, we could do that as follows:

可用的令牌包括％（table\_name）s，％（refer\_table\_name）s，％（column\_0\_name）s，％（column\_0\_label）s，％（column\_0\_key）s，％（refer\_column\_0\_name）s和％（constraint\_name） MetaData.naming\_convention的文档单独描述。 还可以通过在naming\_convention字典中指定附加令牌和可调用来添加新令牌。 例如，如果我们想使用GUID方案命名我们的外键约束，我们可以这样做：

**import** **uuid**

**def** fk\_guid(constraint, table):

str\_tokens = [

table.name,

] + [

element.parent.name **for** element **in** constraint.elements

] + [

element.target\_fullname **for** element **in** constraint.elements

]

guid = uuid.uuid5(uuid.NAMESPACE\_OID, "\_".join(str\_tokens).encode('ascii'))

**return** str(guid)

convention = {

"fk\_guid": fk\_guid,

"ix": 'ix\_*%(column\_0\_label)s*',

"fk": "fk\_*%(fk\_guid)s*",}

Above, when we create a new [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint), we will get a name as follows:

以上，当我们创建一个新的ForeignKeyConstraint时，我们将得到如下的名称：

**>>>** metadata = MetaData(naming\_convention=convention)

**>>>** user\_table = Table('user', metadata,

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

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

**...**  Column('data', String(30))**...**  )

**>>>** address\_table = Table('address', metadata,

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

**...**  Column('user\_id', Integer),

**...**  Column('user\_version\_id', Integer)

**...**  )

**>>>** fk = ForeignKeyConstraint(['user\_id', 'user\_version\_id'],**...**  ['user.id', 'user.version'])

**>>>** address\_table.append\_constraint(fk)

**>>>** fk.namefk\_0cd51ab5-8d70-56e8-a83c-86661737766d

**See also**

[MetaData.naming\_convention](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData.params.naming_convention" \o "sqlalchemy.schema.MetaData) - for additional usage details as well as a listing of all available naming components.

MetaData.naming\_convention - 有关其他使用细节以及所有可用命名组件的列表。

[The Importance of Naming Constraints](http://alembic.zzzcomputing.com/en/latest/naming.html) - in the Alembic documentation.

*New in version 0.9.2:*Added the [MetaData.naming\_convention](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData.params.naming_convention" \o "sqlalchemy.schema.MetaData) argument.

### Naming CHECK Constraints

The [CheckConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.CheckConstraint" \o "sqlalchemy.schema.CheckConstraint) object is configured against an arbitrary SQL expression, which can have any number of columns present, and additionally is often configured using a raw SQL string. Therefore a common convention to use with [CheckConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.CheckConstraint" \o "sqlalchemy.schema.CheckConstraint) is one where we expect the object to have a name already, and we then enhance it with other convention elements. A typical convention is "ck\_%(table\_name)s\_%(constraint\_name)s":

CheckConstraint对象被配置为一个任意的SQL表达式，它可以有任何数量的列存在，另外通常使用原始SQL字符串进行配置。 因此，与CheckConstraint一起使用的常见约定是我们期望对象具有名称的常规约定，然后我们使用其他约定元素来增强它。 一个典型的约定是“ck \_％（table\_name）s \_％（constraint\_name）s”：

metadata = MetaData(

naming\_convention={"ck": "ck\_*%(table\_name)s*\_*%(constraint\_name)s*"})

Table('foo', metadata,

Column('value', Integer),

CheckConstraint('value > 5', name='value\_gt\_5'))

The above table will produce the name ck\_foo\_value\_gt\_5:

上表将产生名称ck\_foo\_value\_gt\_5：

CREATE TABLE foo (

value INTEGER,

CONSTRAINT ck\_foo\_value\_gt\_5 CHECK (value > 5))

[CheckConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.CheckConstraint" \o "sqlalchemy.schema.CheckConstraint) also supports the %(columns\_0\_name)s token; we can make use of this by ensuring we use a [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) or [sql.expression.column()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.column" \o "sqlalchemy.sql.expression.column)element within the constraint's expression, either by declaring the constraint separate from the table:

CheckConstraint还支持％（columns\_0\_name）的令牌; 我们可以通过确保我们在约束的表达式中使用Column或sql.expression.column() 元素，通过声明约束与表格分离：

metadata = MetaData(

naming\_convention={"ck": "ck\_*%(table\_name)s*\_*%(column\_0\_name)s*"})

foo = Table('foo', metadata,

Column('value', Integer))

CheckConstraint(foo.c.value > 5)

or by using a [sql.expression.column()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.column" \o "sqlalchemy.sql.expression.column) inline:

**from** **sqlalchemy** **import** column

metadata = MetaData(

naming\_convention={"ck": "ck\_*%(table\_name)s*\_*%(column\_0\_name)s*"})

foo = Table('foo', metadata,

Column('value', Integer),

CheckConstraint(column('value') > 5))

Both will produce the name ck\_foo\_value:

两者都会产生名称ck\_foo\_value：

CREATE TABLE foo (

value INTEGER,

CONSTRAINT ck\_foo\_value CHECK (value > 5))

The determination of the name of "column zero" is performed by scanning the given expression for column objects. If the expression has more than one column present, the scan does use a deterministic search, however the structure of the expression will determine which column is noted as "column zero".

通过扫描列对象的给定表达式来执行“列零”名称的确定。 如果表达式有多个列存在，则扫描确实使用确定性搜索，但是表达式的结构将确定哪个列被标记为“列零”。

*New in version 1.0.0:*The [CheckConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.CheckConstraint" \o "sqlalchemy.schema.CheckConstraint) object now supports the column\_0\_name naming convention token.

新版本1.0.0：CheckConstraint对象现在支持column\_0\_name命名约定令牌。

### Configuring Naming for Boolean, Enum, and other schema types

The [SchemaType](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.SchemaType" \o "sqlalchemy.types.SchemaType) class refers to type objects such as [Boolean](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Boolean" \o "sqlalchemy.types.Boolean) and [Enum](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Enum" \o "sqlalchemy.types.Enum) which generate a CHECK constraint accompanying the type. The name for the constraint here is most directly set up by sending the "name" parameter, e.g. [Boolean.name](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Boolean.params.name" \o "sqlalchemy.types.Boolean):

Table('foo', metadata,

Column('flag', Boolean(name='ck\_foo\_flag')))

The naming convention feature may be combined with these types as well, normally by using a convention which includes %(constraint\_name)s and then applying a name to the type:

metadata = MetaData(

naming\_convention={"ck": "ck\_*%(table\_name)s*\_*%(constraint\_name)s*"})

Table('foo', metadata,

Column('flag', Boolean(name='flag\_bool')))

The above table will produce the constraint name ck\_foo\_flag\_bool:

CREATE TABLE foo (

flag BOOL,

CONSTRAINT ck\_foo\_flag\_bool CHECK (flag IN (0, 1)))

The [SchemaType](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.SchemaType" \o "sqlalchemy.types.SchemaType) classes use special internal symbols so that the naming convention is only determined at DDL compile time. On PostgreSQL, there's a native BOOLEAN type, so the CHECK constraint of [Boolean](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Boolean" \o "sqlalchemy.types.Boolean) is not needed; we are safe to set up a [Boolean](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Boolean" \o "sqlalchemy.types.Boolean) type without a name, even though a naming convention is in place for check constraints. This convention will only be consulted for the CHECK constraint if we run against a database without a native BOOLEAN type like SQLite or MySQL.

The CHECK constraint may also make use of the column\_0\_name token, which works nicely with [SchemaType](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.SchemaType" \o "sqlalchemy.types.SchemaType) since these constraints have only one column:

metadata = MetaData(

naming\_convention={"ck": "ck\_*%(table\_name)s*\_*%(column\_0\_name)s*"})

Table('foo', metadata,

Column('flag', Boolean()))

The above schema will produce:

CREATE TABLE foo (

flag BOOL,

CONSTRAINT ck\_foo\_flag CHECK (flag IN (0, 1)))

*Changed in version 1.0:*Constraint naming conventions that don't include %(constraint\_name)s again work with [SchemaType](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.SchemaType" \o "sqlalchemy.types.SchemaType) constraints.

3.4.7 Constraints API

*class*sqlalchemy.schema.**Constraint**(*name=None*, *deferrable=None*, *initially=None*, *\_create\_rule=None*, *info=None*, *\_type\_bound=False*, *\*\*dialect\_kw*)

Bases: [sqlalchemy.sql.base.DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs), [sqlalchemy.schema.SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem)

A table-level SQL constraint.

**\_\_init\_\_**(*name=None*, *deferrable=None*, *initially=None*, *\_create\_rule=None*, *info=None*, *\_type\_bound=False*, *\*\*dialect\_kw*)

Create a SQL constraint.

|  |  |
| --- | --- |
| **Parameters:** | * ****name**** – Optional, the in-database name of this Constraint. * ****deferrable**** – Optional bool. If set, emit DEFERRABLE or NOT DEFERRABLE when issuing DDL for this constraint. * ****initially**** – Optional string. If set, emit INITIALLY <value> when issuing DDL for this constraint. * ****info –****Optional data dictionary which will be populated into the [SchemaItem.info](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem.info" \o "sqlalchemy.schema.SchemaItem.info) attribute of this object.   *New in version 1.0.0.*   * ****\_create\_rule**** –   a callable which is passed the DDLCompiler object during compilation. Returns True or False to signal inline generation of this Constraint.  The AddConstraint and DropConstraint DDL constructs provide DDLElement's more comprehensive "conditional DDL" approach that is passed a database connection when DDL is being issued. \_create\_rule is instead called during any CREATE TABLE compilation, where there may not be any transaction/connection in progress. However, it allows conditional compilation of the constraint even for backends which do not support addition of constraints through ALTER TABLE, which currently includes SQLite.  \_create\_rule is used by some types to create constraints. Currently, its call signature is subject to change at any time.   * ****\*\*dialect\_kw**** – Additional keyword arguments are dialect specific, and passed in the form <dialectname>\_<argname>. See the documentation regarding an individual dialect at [Dialects](http://docs.sqlalchemy.org/en/rel_1_1/dialects/index.html) for detail on documented arguments. |

*class*sqlalchemy.schema.**ColumnCollectionMixin**(*\*columns*, *\*\*kw*)

**columns***= None*

A [ColumnCollection](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnCollection" \o "sqlalchemy.sql.expression.ColumnCollection) of [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects.

This collection represents the columns which are referred to by this object.

*class*sqlalchemy.schema.**ColumnCollectionConstraint**(*\*columns*, *\*\*kw*)

Bases: [sqlalchemy.schema.ColumnCollectionMixin](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ColumnCollectionMixin" \o "sqlalchemy.schema.ColumnCollectionMixin), [sqlalchemy.schema.Constraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.Constraint" \o "sqlalchemy.schema.Constraint)

A constraint that proxies a ColumnCollection.

**\_\_init\_\_**(*\*columns*, *\*\*kw*)

|  |  |
| --- | --- |
| **Parameters:** | * ****\*columns**** – A sequence of column names or Column objects. * ****name**** – Optional, the in-database name of this constraint. * ****deferrable**** – Optional bool. If set, emit DEFERRABLE or NOT DEFERRABLE when issuing DDL for this constraint. * ****initially**** – Optional string. If set, emit INITIALLY <value> when issuing DDL for this constraint. * ****\*\*kw**** – other keyword arguments including dialect-specific arguments are propagated to the [Constraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.Constraint" \o "sqlalchemy.schema.Constraint) superclass. |

**argument\_for**(*dialect\_name*, *argument\_name*, *default*)

*inherited from the* [argument\_for()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.argument_for" \o "sqlalchemy.sql.base.DialectKWArgs.argument_for) *method of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

Add a new kind of dialect-specific keyword argument for this class.

E.g.:

Index.argument\_for("mydialect", "length", **None**)

some\_index = Index('a', 'b', mydialect\_length=5)

The [DialectKWArgs.argument\_for()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.argument_for" \o "sqlalchemy.sql.base.DialectKWArgs.argument_for) method is a per-argument way adding extra arguments to the[DefaultDialect.construct\_arguments](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.default.DefaultDialect.construct_arguments" \o "sqlalchemy.engine.default.DefaultDialect.construct_arguments) dictionary. This dictionary provides a list of argument names accepted by various schema-level constructs on behalf of a dialect.

New dialects should typically specify this dictionary all at once as a data member of the dialect class. The use case for ad-hoc addition of argument names is typically for end-user code that is also using a custom compilation scheme which consumes the additional arguments.

|  |  |
| --- | --- |
| **Parameters:** | * ****dialect\_name**** – name of a dialect. The dialect must be locatable, else a [NoSuchModuleError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.NoSuchModuleError" \o "sqlalchemy.exc.NoSuchModuleError) is raised. The dialect must also include an existing[DefaultDialect.construct\_arguments](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.default.DefaultDialect.construct_arguments" \o "sqlalchemy.engine.default.DefaultDialect.construct_arguments) collection, indicating that it participates in the keyword-argument validation and default system, else [ArgumentError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.ArgumentError" \o "sqlalchemy.exc.ArgumentError) is raised. If the dialect does not include this collection, then any keyword argument can be specified on behalf of this dialect already. All dialects packaged within SQLAlchemy include this collection, however for third party dialects, support may vary. * ****argument\_name**** – name of the parameter. * ****default**** – default value of the parameter. |

*New in version 0.9.4.*

**columns***= None*

A [ColumnCollection](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnCollection" \o "sqlalchemy.sql.expression.ColumnCollection) representing the set of columns for this constraint.

**contains\_column**(*col*)

Return True if this constraint contains the given column.

Note that this object also contains an attribute .columns which is a [ColumnCollection](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnCollection" \o "sqlalchemy.sql.expression.ColumnCollection) of [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects.

**dialect\_kwargs**

*inherited from the* [dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs) *attribute of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

A collection of keyword arguments specified as dialect-specific options to this construct.

The arguments are present here in their original <dialect>\_<kwarg> format. Only arguments that were actually passed are included; unlike the [DialectKWArgs.dialect\_options](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_options" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_options) collection, which contains all options known by this dialect including defaults.

The collection is also writable; keys are accepted of the form <dialect>\_<kwarg> where the value will be assembled into the list of options.

*New in version 0.9.2.*

*Changed in version 0.9.4:*The [DialectKWArgs.dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs) collection is now writable.

**See also**

[DialectKWArgs.dialect\_options](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_options" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_options) - nested dictionary form

**dialect\_options**

*inherited from the* [dialect\_options](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_options" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_options) *attribute of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

A collection of keyword arguments specified as dialect-specific options to this construct.

This is a two-level nested registry, keyed to <dialect\_name> and <argument\_name>. For example, the postgresql\_where argument would be locatable as:

arg = my\_object.dialect\_options['postgresql']['where']

*New in version 0.9.2.*

**See also**

[DialectKWArgs.dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs) - flat dictionary form

**get\_children**(*\*\*kwargs*)

*inherited from the* [get\_children()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem.get_children" \o "sqlalchemy.schema.SchemaItem.get_children) *method of* [SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem)

used to allow SchemaVisitor access

**info**

*inherited from the* [info](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem.info" \o "sqlalchemy.schema.SchemaItem.info) *attribute of* [SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem)

Info dictionary associated with the object, allowing user-defined data to be associated with this [SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem).

The dictionary is automatically generated when first accessed. It can also be specified in the constructor of some objects, such as [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) and [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column).

**kwargs**

*inherited from the* [kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.kwargs) *attribute of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

A synonym for [DialectKWArgs.dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs).

**quote**

*inherited from the* [quote](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem.quote" \o "sqlalchemy.schema.SchemaItem.quote) *attribute of* [SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem)

Return the value of the quote flag passed to this schema object, for those schema items which have a name field.

*Deprecated since version 0.9:*Use <obj>.name.quote

*class*sqlalchemy.schema.**CheckConstraint**(*sqltext*, *name=None*, *deferrable=None*, *initially=None*, *table=None*, *info=None*, *\_create\_rule=None*, *\_autoattach=True*, *\_type\_bound=False*)

Bases: [sqlalchemy.schema.ColumnCollectionConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ColumnCollectionConstraint" \o "sqlalchemy.schema.ColumnCollectionConstraint)

A table- or column-level CHECK constraint.

Can be included in the definition of a Table or Column.

**\_\_init\_\_**(*sqltext*, *name=None*, *deferrable=None*, *initially=None*, *table=None*, *info=None*, *\_create\_rule=None*, *\_autoattach=True*, *\_type\_bound=False*)

Construct a CHECK constraint.

|  |  |
| --- | --- |
| **Parameters:** | * ****sqltext**** –   A string containing the constraint definition, which will be used verbatim, or a SQL expression construct. If given as a string, the object is converted to a [Text](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Text" \o "sqlalchemy.types.Text) object. If the textual string includes a colon character, escape this using a backslash:  CheckConstraint(r"foo ~ E'a(?\:b|c)d")   * ****name**** – Optional, the in-database name of the constraint. * ****deferrable**** – Optional bool. If set, emit DEFERRABLE or NOT DEFERRABLE when issuing DDL for this constraint. * ****initially**** – Optional string. If set, emit INITIALLY <value> when issuing DDL for this constraint. * ****info**** –   Optional data dictionary which will be populated into the [SchemaItem.info](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem.info" \o "sqlalchemy.schema.SchemaItem.info) attribute of this object.  *New in version 1.0.0.* |

**argument\_for**(*dialect\_name*, *argument\_name*, *default*)

*inherited from the* [argument\_for()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.argument_for" \o "sqlalchemy.sql.base.DialectKWArgs.argument_for) *method of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

Add a new kind of dialect-specific keyword argument for this class.

E.g.:

Index.argument\_for("mydialect", "length", **None**)

some\_index = Index('a', 'b', mydialect\_length=5)

The [DialectKWArgs.argument\_for()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.argument_for" \o "sqlalchemy.sql.base.DialectKWArgs.argument_for) method is a per-argument way adding extra arguments to the[DefaultDialect.construct\_arguments](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.default.DefaultDialect.construct_arguments" \o "sqlalchemy.engine.default.DefaultDialect.construct_arguments) dictionary. This dictionary provides a list of argument names accepted by various schema-level constructs on behalf of a dialect.

New dialects should typically specify this dictionary all at once as a data member of the dialect class. The use case for ad-hoc addition of argument names is typically for end-user code that is also using a custom compilation scheme which consumes the additional arguments.

|  |  |
| --- | --- |
| **Parameters:** | * ****dialect\_name**** – name of a dialect. The dialect must be locatable, else a [NoSuchModuleError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.NoSuchModuleError" \o "sqlalchemy.exc.NoSuchModuleError) is raised. The dialect must also include an existing[DefaultDialect.construct\_arguments](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.default.DefaultDialect.construct_arguments" \o "sqlalchemy.engine.default.DefaultDialect.construct_arguments) collection, indicating that it participates in the keyword-argument validation and default system, else [ArgumentError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.ArgumentError" \o "sqlalchemy.exc.ArgumentError) is raised. If the dialect does not include this collection, then any keyword argument can be specified on behalf of this dialect already. All dialects packaged within SQLAlchemy include this collection, however for third party dialects, support may vary. * ****argument\_name**** – name of the parameter. * ****default**** – default value of the parameter. |

*New in version 0.9.4.*

**contains\_column**(*col*)

*inherited from the* [contains\_column()](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ColumnCollectionConstraint.contains_column" \o "sqlalchemy.schema.ColumnCollectionConstraint.contains_column) *method of* [ColumnCollectionConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ColumnCollectionConstraint" \o "sqlalchemy.schema.ColumnCollectionConstraint)

Return True if this constraint contains the given column.

Note that this object also contains an attribute .columns which is a [ColumnCollection](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnCollection" \o "sqlalchemy.sql.expression.ColumnCollection) of [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects.

**dialect\_kwargs**

*inherited from the* [dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs) *attribute of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

A collection of keyword arguments specified as dialect-specific options to this construct.

The arguments are present here in their original <dialect>\_<kwarg> format. Only arguments that were actually passed are included; unlike the [DialectKWArgs.dialect\_options](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_options" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_options) collection, which contains all options known by this dialect including defaults.

The collection is also writable; keys are accepted of the form <dialect>\_<kwarg> where the value will be assembled into the list of options.

*New in version 0.9.2.*

*Changed in version 0.9.4:*The [DialectKWArgs.dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs) collection is now writable.

**See also**

[DialectKWArgs.dialect\_options](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_options" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_options) - nested dictionary form

**dialect\_options**

*inherited from the* [dialect\_options](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_options" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_options) *attribute of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

A collection of keyword arguments specified as dialect-specific options to this construct.

This is a two-level nested registry, keyed to <dialect\_name> and <argument\_name>. For example, the postgresql\_where argument would be locatable as:

arg = my\_object.dialect\_options['postgresql']['where']

*New in version 0.9.2.*

**See also**

[DialectKWArgs.dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs) - flat dictionary form

**get\_children**(*\*\*kwargs*)

*inherited from the* [get\_children()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem.get_children" \o "sqlalchemy.schema.SchemaItem.get_children) *method of* [SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem)

used to allow SchemaVisitor access

**info**

*inherited from the* [info](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem.info" \o "sqlalchemy.schema.SchemaItem.info) *attribute of* [SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem)

Info dictionary associated with the object, allowing user-defined data to be associated with this [SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem).

The dictionary is automatically generated when first accessed. It can also be specified in the constructor of some objects, such as [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) and [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column).

**kwargs**

*inherited from the* [kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.kwargs) *attribute of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

A synonym for [DialectKWArgs.dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs).

**quote**

*inherited from the* [quote](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem.quote" \o "sqlalchemy.schema.SchemaItem.quote) *attribute of* [SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem)

Return the value of the quote flag passed to this schema object, for those schema items which have a name field.

*Deprecated since version 0.9:*Use <obj>.name.quote

*class*sqlalchemy.schema.**ForeignKey**(*column*, *\_constraint=None*, *use\_alter=False*, *name=None*, *onupdate=None*, *ondelete=None*, *deferrable=None*, *initially=None*, *link\_to\_name=False*, *match=None*, *info=None*, *\*\*dialect\_kw*)

Bases: [sqlalchemy.sql.base.DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs), [sqlalchemy.schema.SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem)

Defines a dependency between two columns.

ForeignKey is specified as an argument to a [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) object, e.g.:

t = Table("remote\_table", metadata,

Column("remote\_id", ForeignKey("main\_table.id")))

Note that ForeignKey is only a marker object that defines a dependency between two columns. The actual constraint is in all cases represented by the [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint) object. This object will be generated automatically when a ForeignKey is associated with a [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) which in turn is associated with a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table). Conversely, when [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint) is applied to a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table), ForeignKey markers are automatically generated to be present on each associated [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column), which are also associated with the constraint object.

Note that you cannot define a "composite" foreign key constraint, that is a constraint between a grouping of multiple parent/child columns, using ForeignKeyobjects. To define this grouping, the [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint) object must be used, and applied to the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table). The associated ForeignKey objects are created automatically.

The ForeignKey objects associated with an individual [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) object are available in the foreign\_keys collection of that column.

Further examples of foreign key configuration are in [Defining Foreign Keys](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "metadata-foreignkeys).

**\_\_init\_\_**(*column*, *\_constraint=None*, *use\_alter=False*, *name=None*, *onupdate=None*, *ondelete=None*, *deferrable=None*, *initially=None*, *link\_to\_name=False*, *match=None*, *info=None*, *\*\*dialect\_kw*)

Construct a column-level FOREIGN KEY.

The [ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey) object when constructed generates a [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint) which is associated with the parent [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object's collection of constraints.

|  |  |
| --- | --- |
| **Parameters:** | * ****column –****A single target column for the key relationship. A [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) object or a column name as a string: tablename.columnkey or schema.tablename.columnkey. columnkey is the key which has been assigned to the column (defaults to the column name itself), unless link\_to\_name is True in which case the rendered name of the column is used.   *New in version 0.7.4:*Note that if the schema name is not included, and the underlying [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) has a "schema", that value will be used.   * ****name**** – Optional string. An in-database name for the key if constraint is not provided. * ****onupdate**** – Optional string. If set, emit ON UPDATE <value> when issuing DDL for this constraint. Typical values include CASCADE, DELETE and RESTRICT. * ****ondelete**** – Optional string. If set, emit ON DELETE <value> when issuing DDL for this constraint. Typical values include CASCADE, DELETE and RESTRICT. * ****deferrable**** – Optional bool. If set, emit DEFERRABLE or NOT DEFERRABLE when issuing DDL for this constraint. * ****initially**** – Optional string. If set, emit INITIALLY <value> when issuing DDL for this constraint. * ****link\_to\_name**** – if True, the string name given in column is the rendered name of the referenced column, not its locally assigned key. * ****use\_alter**** –   passed to the underlying [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint) to indicate the constraint should be generated/dropped externally from the CREATE TABLE/ DROP TABLE statement. See [ForeignKeyConstraint.use\_alter](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint.params.use_alter" \o "sqlalchemy.schema.ForeignKeyConstraint) for further description.  **See also**  [ForeignKeyConstraint.use\_alter](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint.params.use_alter" \o "sqlalchemy.schema.ForeignKeyConstraint)  [Creating/Dropping Foreign Key Constraints via ALTER](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "use-alter)   * ****match**** – Optional string. If set, emit MATCH <value> when issuing DDL for this constraint. Typical values include SIMPLE, PARTIAL and FULL. * ****info**** –   Optional data dictionary which will be populated into the [SchemaItem.info](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem.info" \o "sqlalchemy.schema.SchemaItem.info) attribute of this object.  *New in version 1.0.0.*   * ****\*\*dialect\_kw**** –   Additional keyword arguments are dialect specific, and passed in the form <dialectname>\_<argname>. The arguments are ultimately handled by a corresponding [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint). See the documentation regarding an individual dialect at [Dialects](http://docs.sqlalchemy.org/en/rel_1_1/dialects/index.html) for detail on documented arguments.  *New in version 0.9.2.* |

**argument\_for**(*dialect\_name*, *argument\_name*, *default*)

*inherited from the* [argument\_for()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.argument_for" \o "sqlalchemy.sql.base.DialectKWArgs.argument_for) *method of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

Add a new kind of dialect-specific keyword argument for this class.

E.g.:

Index.argument\_for("mydialect", "length", **None**)

some\_index = Index('a', 'b', mydialect\_length=5)

The [DialectKWArgs.argument\_for()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.argument_for" \o "sqlalchemy.sql.base.DialectKWArgs.argument_for) method is a per-argument way adding extra arguments to the[DefaultDialect.construct\_arguments](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.default.DefaultDialect.construct_arguments" \o "sqlalchemy.engine.default.DefaultDialect.construct_arguments) dictionary. This dictionary provides a list of argument names accepted by various schema-level constructs on behalf of a dialect.

New dialects should typically specify this dictionary all at once as a data member of the dialect class. The use case for ad-hoc addition of argument names is typically for end-user code that is also using a custom compilation scheme which consumes the additional arguments.

|  |  |
| --- | --- |
| **Parameters:** | * ****dialect\_name**** – name of a dialect. The dialect must be locatable, else a [NoSuchModuleError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.NoSuchModuleError" \o "sqlalchemy.exc.NoSuchModuleError) is raised. The dialect must also include an existing[DefaultDialect.construct\_arguments](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.default.DefaultDialect.construct_arguments" \o "sqlalchemy.engine.default.DefaultDialect.construct_arguments) collection, indicating that it participates in the keyword-argument validation and default system, else [ArgumentError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.ArgumentError" \o "sqlalchemy.exc.ArgumentError) is raised. If the dialect does not include this collection, then any keyword argument can be specified on behalf of this dialect already. All dialects packaged within SQLAlchemy include this collection, however for third party dialects, support may vary. * ****argument\_name**** – name of the parameter. * ****default**** – default value of the parameter. |

*New in version 0.9.4.*

**column**

Return the target [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) referenced by this [ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey).

If no target column has been established, an exception is raised.

*Changed in version 0.9.0:*Foreign key target column resolution now occurs as soon as both the ForeignKey object and the remote Column to which it refers are both associated with the same MetaData object.

**copy**(*schema=None*)

Produce a copy of this [ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey) object.

The new [ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey) will not be bound to any [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column).

This method is usually used by the internal copy procedures of [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column), [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table), and [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData).

|  |  |
| --- | --- |
| **Parameters:** | ****schema**** – The returned [ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey) will reference the original table and column name, qualified by the given string schema name. |

**dialect\_kwargs**

*inherited from the* [dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs) *attribute of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

A collection of keyword arguments specified as dialect-specific options to this construct.

The arguments are present here in their original <dialect>\_<kwarg> format. Only arguments that were actually passed are included; unlike the [DialectKWArgs.dialect\_options](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_options" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_options) collection, which contains all options known by this dialect including defaults.

The collection is also writable; keys are accepted of the form <dialect>\_<kwarg> where the value will be assembled into the list of options.

*New in version 0.9.2.*

*Changed in version 0.9.4:*The [DialectKWArgs.dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs) collection is now writable.

**See also**

[DialectKWArgs.dialect\_options](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_options" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_options) - nested dictionary form

**dialect\_options**

*inherited from the* [dialect\_options](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_options" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_options) *attribute of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

A collection of keyword arguments specified as dialect-specific options to this construct.

This is a two-level nested registry, keyed to <dialect\_name> and <argument\_name>. For example, the postgresql\_where argument would be locatable as:

arg = my\_object.dialect\_options['postgresql']['where']

*New in version 0.9.2.*

**See also**

[DialectKWArgs.dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs) - flat dictionary form

**get\_children**(*\*\*kwargs*)

*inherited from the* [get\_children()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem.get_children" \o "sqlalchemy.schema.SchemaItem.get_children) *method of* [SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem)

used to allow SchemaVisitor access

**get\_referent**(*table*)

Return the [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) in the given [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) referenced by this [ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey).

Returns None if this [ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey) does not reference the given [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table).

**info**

*inherited from the* [info](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem.info" \o "sqlalchemy.schema.SchemaItem.info) *attribute of* [SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem)

Info dictionary associated with the object, allowing user-defined data to be associated with this [SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem).

The dictionary is automatically generated when first accessed. It can also be specified in the constructor of some objects, such as [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) and [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column).

**kwargs**

*inherited from the* [kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.kwargs) *attribute of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

A synonym for [DialectKWArgs.dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs).

**quote**

*inherited from the* [quote](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem.quote" \o "sqlalchemy.schema.SchemaItem.quote) *attribute of* [SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem)

Return the value of the quote flag passed to this schema object, for those schema items which have a name field.

*Deprecated since version 0.9:*Use <obj>.name.quote

**references**(*table*)

Return True if the given [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) is referenced by this [ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey).

**target\_fullname**

Return a string based 'column specification' for this [ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey).

This is usually the equivalent of the string-based "tablename.colname" argument first passed to the object's constructor.

*class*sqlalchemy.schema.**ForeignKeyConstraint**(*columns*, *refcolumns*, *name=None*, *onupdate=None*, *ondelete=None*, *deferrable=None*, *initially=None*, *use\_alter=False*, *link\_to\_name=False*, *match=None*, *table=None*, *info=None*, *\*\*dialect\_kw*)

Bases: [sqlalchemy.schema.ColumnCollectionConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ColumnCollectionConstraint" \o "sqlalchemy.schema.ColumnCollectionConstraint)

A table-level FOREIGN KEY constraint.

Defines a single column or composite FOREIGN KEY … REFERENCES constraint. For a no-frills, single column foreign key, adding a [ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey) to the definition of a [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) is a shorthand equivalent for an unnamed, single column [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint).

Examples of foreign key configuration are in [Defining Foreign Keys](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "metadata-foreignkeys).

**\_\_init\_\_**(*columns*, *refcolumns*, *name=None*, *onupdate=None*, *ondelete=None*, *deferrable=None*, *initially=None*, *use\_alter=False*, *link\_to\_name=False*, *match=None*, *table=None*, *info=None*, *\*\*dialect\_kw*)

Construct a composite-capable FOREIGN KEY.

|  |  |
| --- | --- |
| **Parameters:** | * ****columns**** – A sequence of local column names. The named columns must be defined and present in the parent Table. The names should match the keygiven to each column (defaults to the name) unless link\_to\_name is True. * ****refcolumns**** – A sequence of foreign column names or Column objects. The columns must all be located within the same Table. * ****name**** – Optional, the in-database name of the key. * ****onupdate**** – Optional string. If set, emit ON UPDATE <value> when issuing DDL for this constraint. Typical values include CASCADE, DELETE and RESTRICT. * ****ondelete**** – Optional string. If set, emit ON DELETE <value> when issuing DDL for this constraint. Typical values include CASCADE, DELETE and RESTRICT. * ****deferrable**** – Optional bool. If set, emit DEFERRABLE or NOT DEFERRABLE when issuing DDL for this constraint. * ****initially**** – Optional string. If set, emit INITIALLY <value> when issuing DDL for this constraint. * ****link\_to\_name**** – if True, the string name given in column is the rendered name of the referenced column, not its locally assigned key. * ****use\_alter**** –   If True, do not emit the DDL for this constraint as part of the CREATE TABLE definition. Instead, generate it via an ALTER TABLE statement issued after the full collection of tables have been created, and drop it via an ALTER TABLE statement before the full collection of tables are dropped.  The use of [ForeignKeyConstraint.use\_alter](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint.params.use_alter" \o "sqlalchemy.schema.ForeignKeyConstraint) is particularly geared towards the case where two or more tables are established within a mutually-dependent foreign key constraint relationship; however, 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) and [MetaData.drop\_all()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData.drop_all" \o "sqlalchemy.schema.MetaData.drop_all) methods will perform this resolution automatically, so the flag is normally not needed.  *Changed in version 1.0.0:*Automatic resolution of foreign key cycles has been added, removing the need to use the [ForeignKeyConstraint.use\_alter](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint.params.use_alter" \o "sqlalchemy.schema.ForeignKeyConstraint) in typical use cases.  **See also**  [Creating/Dropping Foreign Key Constraints via ALTER](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "use-alter)   * ****match**** – Optional string. If set, emit MATCH <value> when issuing DDL for this constraint. Typical values include SIMPLE, PARTIAL and FULL. * ****info**** –   Optional data dictionary which will be populated into the [SchemaItem.info](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem.info" \o "sqlalchemy.schema.SchemaItem.info) attribute of this object.  *New in version 1.0.0.*   * ****\*\*dialect\_kw**** –   Additional keyword arguments are dialect specific, and passed in the form <dialectname>\_<argname>. See the documentation regarding an individual dialect at [Dialects](http://docs.sqlalchemy.org/en/rel_1_1/dialects/index.html) for detail on documented arguments.  *New in version 0.9.2.* |

**argument\_for**(*dialect\_name*, *argument\_name*, *default*)

*inherited from the* [argument\_for()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.argument_for" \o "sqlalchemy.sql.base.DialectKWArgs.argument_for) *method of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

Add a new kind of dialect-specific keyword argument for this class.

E.g.:

Index.argument\_for("mydialect", "length", **None**)

some\_index = Index('a', 'b', mydialect\_length=5)

The [DialectKWArgs.argument\_for()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.argument_for" \o "sqlalchemy.sql.base.DialectKWArgs.argument_for) method is a per-argument way adding extra arguments to the[DefaultDialect.construct\_arguments](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.default.DefaultDialect.construct_arguments" \o "sqlalchemy.engine.default.DefaultDialect.construct_arguments) dictionary. This dictionary provides a list of argument names accepted by various schema-level constructs on behalf of a dialect.

New dialects should typically specify this dictionary all at once as a data member of the dialect class. The use case for ad-hoc addition of argument names is typically for end-user code that is also using a custom compilation scheme which consumes the additional arguments.

|  |  |
| --- | --- |
| **Parameters:** | * ****dialect\_name**** – name of a dialect. The dialect must be locatable, else a [NoSuchModuleError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.NoSuchModuleError" \o "sqlalchemy.exc.NoSuchModuleError) is raised. The dialect must also include an existing[DefaultDialect.construct\_arguments](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.default.DefaultDialect.construct_arguments" \o "sqlalchemy.engine.default.DefaultDialect.construct_arguments) collection, indicating that it participates in the keyword-argument validation and default system, else [ArgumentError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.ArgumentError" \o "sqlalchemy.exc.ArgumentError) is raised. If the dialect does not include this collection, then any keyword argument can be specified on behalf of this dialect already. All dialects packaged within SQLAlchemy include this collection, however for third party dialects, support may vary. * ****argument\_name**** – name of the parameter. * ****default**** – default value of the parameter. |

*New in version 0.9.4.*

**column\_keys**

Return a list of string keys representing the local columns in this [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint).

This list is either the original string arguments sent to the constructor of the [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint), or if the constraint has been initialized with [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects, is the string .key of each element.

*New in version 1.0.0.*

**columns***= None*

A [ColumnCollection](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnCollection" \o "sqlalchemy.sql.expression.ColumnCollection) representing the set of columns for this constraint.

**contains\_column**(*col*)

*inherited from the* [contains\_column()](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ColumnCollectionConstraint.contains_column" \o "sqlalchemy.schema.ColumnCollectionConstraint.contains_column) *method of* [ColumnCollectionConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ColumnCollectionConstraint" \o "sqlalchemy.schema.ColumnCollectionConstraint)

Return True if this constraint contains the given column.

Note that this object also contains an attribute .columns which is a [ColumnCollection](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnCollection" \o "sqlalchemy.sql.expression.ColumnCollection) of [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects.

**dialect\_kwargs**

*inherited from the* [dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs) *attribute of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

A collection of keyword arguments specified as dialect-specific options to this construct.

The arguments are present here in their original <dialect>\_<kwarg> format. Only arguments that were actually passed are included; unlike the [DialectKWArgs.dialect\_options](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_options" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_options) collection, which contains all options known by this dialect including defaults.

The collection is also writable; keys are accepted of the form <dialect>\_<kwarg> where the value will be assembled into the list of options.

*New in version 0.9.2.*

*Changed in version 0.9.4:*The [DialectKWArgs.dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs) collection is now writable.

**See also**

[DialectKWArgs.dialect\_options](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_options" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_options) - nested dictionary form

**dialect\_options**

*inherited from the* [dialect\_options](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_options" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_options) *attribute of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

A collection of keyword arguments specified as dialect-specific options to this construct.

This is a two-level nested registry, keyed to <dialect\_name> and <argument\_name>. For example, the postgresql\_where argument would be locatable as:

arg = my\_object.dialect\_options['postgresql']['where']

*New in version 0.9.2.*

**See also**

[DialectKWArgs.dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs) - flat dictionary form

**elements***= None*

A sequence of [ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey) objects.

Each [ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey) represents a single referring column/referred column pair.

This collection is intended to be read-only.

**get\_children**(*\*\*kwargs*)

*inherited from the* [get\_children()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem.get_children" \o "sqlalchemy.schema.SchemaItem.get_children) *method of* [SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem)

used to allow SchemaVisitor access

**info**

*inherited from the* [info](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem.info" \o "sqlalchemy.schema.SchemaItem.info) *attribute of* [SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem)

Info dictionary associated with the object, allowing user-defined data to be associated with this [SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem).

The dictionary is automatically generated when first accessed. It can also be specified in the constructor of some objects, such as [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) and [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column).

**kwargs**

*inherited from the* [kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.kwargs) *attribute of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

A synonym for [DialectKWArgs.dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs).

**quote**

*inherited from the* [quote](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem.quote" \o "sqlalchemy.schema.SchemaItem.quote) *attribute of* [SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem)

Return the value of the quote flag passed to this schema object, for those schema items which have a name field.

*Deprecated since version 0.9:*Use <obj>.name.quote

**referred\_table**

The [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object to which this [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint) references.

This is a dynamically calculated attribute which may not be available if the constraint and/or parent table is not yet associated with a metadata collection that contains the referred table.

*New in version 1.0.0.*

*class*sqlalchemy.schema.**PrimaryKeyConstraint**(*\*columns*, *\*\*kw*)

Bases: [sqlalchemy.schema.ColumnCollectionConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ColumnCollectionConstraint" \o "sqlalchemy.schema.ColumnCollectionConstraint)

A table-level PRIMARY KEY constraint.

The [PrimaryKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.PrimaryKeyConstraint" \o "sqlalchemy.schema.PrimaryKeyConstraint) object is present automatically on any [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object; it is assigned a set of [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects corresponding to those marked with the [Column.primary\_key](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column.params.primary_key" \o "sqlalchemy.schema.Column) flag:

**>>>** my\_table = Table('mytable', metadata,**...**  Column('id', Integer, primary\_key=**True**),**...**  Column('version\_id', Integer, primary\_key=**True**),**...**  Column('data', String(50))**...**  )**>>>** my\_table.primary\_keyPrimaryKeyConstraint( Column('id', Integer(), table=<mytable>, primary\_key=True, nullable=False), Column('version\_id', Integer(), table=<mytable>, primary\_key=True, nullable=False))

The primary key of a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) can also be specified by using a [PrimaryKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.PrimaryKeyConstraint" \o "sqlalchemy.schema.PrimaryKeyConstraint) object explicitly; in this mode of usage, the "name" of the constraint can also be specified, as well as other options which may be recognized by dialects:

my\_table = Table('mytable', metadata,

Column('id', Integer),

Column('version\_id', Integer),

Column('data', String(50)),

PrimaryKeyConstraint('id', 'version\_id',

name='mytable\_pk')

)

The two styles of column-specification should generally not be mixed. An warning is emitted if the columns present in the [PrimaryKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.PrimaryKeyConstraint" \o "sqlalchemy.schema.PrimaryKeyConstraint) don't match the columns that were marked as primary\_key=True, if both are present; in this case, the columns are taken strictly from the[PrimaryKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.PrimaryKeyConstraint" \o "sqlalchemy.schema.PrimaryKeyConstraint) declaration, and those columns otherwise marked as primary\_key=True are ignored. This behavior is intended to be backwards compatible with previous behavior.

*Changed in version 0.9.2:*Using a mixture of columns within a [PrimaryKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.PrimaryKeyConstraint" \o "sqlalchemy.schema.PrimaryKeyConstraint) in addition to columns marked as primary\_key=True now emits a warning if the lists don't match. The ultimate behavior of ignoring those columns marked with the flag only is currently maintained for backwards compatibility; this warning may raise an exception in a future release.

For the use case where specific options are to be specified on the [PrimaryKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.PrimaryKeyConstraint" \o "sqlalchemy.schema.PrimaryKeyConstraint), but the usual style of using primary\_key=True flags is still desirable, an empty [PrimaryKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.PrimaryKeyConstraint" \o "sqlalchemy.schema.PrimaryKeyConstraint) may be specified, which will take on the primary key column collection from the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) based on the flags:

my\_table = Table('mytable', metadata,

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

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

Column('data', String(50)),

PrimaryKeyConstraint(name='mytable\_pk',

mssql\_clustered=**True**)

)

*New in version 0.9.2:*an empty [PrimaryKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.PrimaryKeyConstraint" \o "sqlalchemy.schema.PrimaryKeyConstraint) may now be specified for the purposes of establishing keyword arguments with the constraint, independently of the specification of "primary key" columns within the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) itself; columns marked as primary\_key=True will be gathered into the empty constraint's column collection.

**argument\_for**(*dialect\_name*, *argument\_name*, *default*)

*inherited from the* [argument\_for()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.argument_for" \o "sqlalchemy.sql.base.DialectKWArgs.argument_for) *method of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

Add a new kind of dialect-specific keyword argument for this class.

E.g.:

Index.argument\_for("mydialect", "length", **None**)

some\_index = Index('a', 'b', mydialect\_length=5)

The [DialectKWArgs.argument\_for()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.argument_for" \o "sqlalchemy.sql.base.DialectKWArgs.argument_for) method is a per-argument way adding extra arguments to the[DefaultDialect.construct\_arguments](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.default.DefaultDialect.construct_arguments" \o "sqlalchemy.engine.default.DefaultDialect.construct_arguments) dictionary. This dictionary provides a list of argument names accepted by various schema-level constructs on behalf of a dialect.

New dialects should typically specify this dictionary all at once as a data member of the dialect class. The use case for ad-hoc addition of argument names is typically for end-user code that is also using a custom compilation scheme which consumes the additional arguments.

|  |  |
| --- | --- |
| **Parameters:** | * ****dialect\_name**** – name of a dialect. The dialect must be locatable, else a [NoSuchModuleError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.NoSuchModuleError" \o "sqlalchemy.exc.NoSuchModuleError) is raised. The dialect must also include an existing[DefaultDialect.construct\_arguments](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.default.DefaultDialect.construct_arguments" \o "sqlalchemy.engine.default.DefaultDialect.construct_arguments) collection, indicating that it participates in the keyword-argument validation and default system, else [ArgumentError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.ArgumentError" \o "sqlalchemy.exc.ArgumentError) is raised. If the dialect does not include this collection, then any keyword argument can be specified on behalf of this dialect already. All dialects packaged within SQLAlchemy include this collection, however for third party dialects, support may vary. * ****argument\_name**** – name of the parameter. * ****default**** – default value of the parameter. |

*New in version 0.9.4.*

**contains\_column**(*col*)

*inherited from the* [contains\_column()](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ColumnCollectionConstraint.contains_column" \o "sqlalchemy.schema.ColumnCollectionConstraint.contains_column) *method of* [ColumnCollectionConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ColumnCollectionConstraint" \o "sqlalchemy.schema.ColumnCollectionConstraint)

Return True if this constraint contains the given column.

Note that this object also contains an attribute .columns which is a [ColumnCollection](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnCollection" \o "sqlalchemy.sql.expression.ColumnCollection) of [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects.

**dialect\_kwargs**

*inherited from the* [dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs) *attribute of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

A collection of keyword arguments specified as dialect-specific options to this construct.

The arguments are present here in their original <dialect>\_<kwarg> format. Only arguments that were actually passed are included; unlike the [DialectKWArgs.dialect\_options](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_options" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_options) collection, which contains all options known by this dialect including defaults.

The collection is also writable; keys are accepted of the form <dialect>\_<kwarg> where the value will be assembled into the list of options.

*New in version 0.9.2.*

*Changed in version 0.9.4:*The [DialectKWArgs.dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs) collection is now writable.

**See also**

[DialectKWArgs.dialect\_options](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_options" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_options) - nested dictionary form

**dialect\_options**

*inherited from the* [dialect\_options](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_options" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_options) *attribute of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

A collection of keyword arguments specified as dialect-specific options to this construct.

This is a two-level nested registry, keyed to <dialect\_name> and <argument\_name>. For example, the postgresql\_where argument would be locatable as:

arg = my\_object.dialect\_options['postgresql']['where']

*New in version 0.9.2.*

**See also**

[DialectKWArgs.dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs) - flat dictionary form

**get\_children**(*\*\*kwargs*)

*inherited from the* [get\_children()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem.get_children" \o "sqlalchemy.schema.SchemaItem.get_children) *method of* [SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem)

used to allow SchemaVisitor access

**info**

*inherited from the* [info](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem.info" \o "sqlalchemy.schema.SchemaItem.info) *attribute of* [SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem)

Info dictionary associated with the object, allowing user-defined data to be associated with this [SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem).

The dictionary is automatically generated when first accessed. It can also be specified in the constructor of some objects, such as [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) and [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column).

**kwargs**

*inherited from the* [kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.kwargs) *attribute of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

A synonym for [DialectKWArgs.dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs).

**quote**

*inherited from the* [quote](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem.quote" \o "sqlalchemy.schema.SchemaItem.quote) *attribute of* [SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem)

Return the value of the quote flag passed to this schema object, for those schema items which have a name field.

*Deprecated since version 0.9:*Use <obj>.name.quote

*class*sqlalchemy.schema.**UniqueConstraint**(*\*columns*, *\*\*kw*)

Bases: [sqlalchemy.schema.ColumnCollectionConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ColumnCollectionConstraint" \o "sqlalchemy.schema.ColumnCollectionConstraint)

A table-level UNIQUE constraint.

Defines a single column or composite UNIQUE constraint. For a no-frills, single column constraint, adding unique=True to the Column definition is a shorthand equivalent for an unnamed, single column UniqueConstraint.

**\_\_init\_\_**(*\*columns*, *\*\*kw*)

*inherited from the* [\_\_init\_\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ColumnCollectionConstraint.__init__" \o "sqlalchemy.schema.ColumnCollectionConstraint.__init__) *method of* [ColumnCollectionConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ColumnCollectionConstraint" \o "sqlalchemy.schema.ColumnCollectionConstraint)

|  |  |
| --- | --- |
| **Parameters:** | * ****\*columns**** – A sequence of column names or Column objects. * ****name**** – Optional, the in-database name of this constraint. * ****deferrable**** – Optional bool. If set, emit DEFERRABLE or NOT DEFERRABLE when issuing DDL for this constraint. * ****initially**** – Optional string. If set, emit INITIALLY <value> when issuing DDL for this constraint. * ****\*\*kw**** – other keyword arguments including dialect-specific arguments are propagated to the [Constraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.Constraint" \o "sqlalchemy.schema.Constraint) superclass. |

**argument\_for**(*dialect\_name*, *argument\_name*, *default*)

*inherited from the* [argument\_for()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.argument_for" \o "sqlalchemy.sql.base.DialectKWArgs.argument_for) *method of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

Add a new kind of dialect-specific keyword argument for this class.

E.g.:

Index.argument\_for("mydialect", "length", **None**)

some\_index = Index('a', 'b', mydialect\_length=5)

The [DialectKWArgs.argument\_for()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.argument_for" \o "sqlalchemy.sql.base.DialectKWArgs.argument_for) method is a per-argument way adding extra arguments to the[DefaultDialect.construct\_arguments](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.default.DefaultDialect.construct_arguments" \o "sqlalchemy.engine.default.DefaultDialect.construct_arguments) dictionary. This dictionary provides a list of argument names accepted by various schema-level constructs on behalf of a dialect.

New dialects should typically specify this dictionary all at once as a data member of the dialect class. The use case for ad-hoc addition of argument names is typically for end-user code that is also using a custom compilation scheme which consumes the additional arguments.

|  |  |
| --- | --- |
| **Parameters:** | * ****dialect\_name**** – name of a dialect. The dialect must be locatable, else a [NoSuchModuleError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.NoSuchModuleError" \o "sqlalchemy.exc.NoSuchModuleError) is raised. The dialect must also include an existing[DefaultDialect.construct\_arguments](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.default.DefaultDialect.construct_arguments" \o "sqlalchemy.engine.default.DefaultDialect.construct_arguments) collection, indicating that it participates in the keyword-argument validation and default system, else [ArgumentError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.ArgumentError" \o "sqlalchemy.exc.ArgumentError) is raised. If the dialect does not include this collection, then any keyword argument can be specified on behalf of this dialect already. All dialects packaged within SQLAlchemy include this collection, however for third party dialects, support may vary. * ****argument\_name**** – name of the parameter. * ****default**** – default value of the parameter. |

*New in version 0.9.4.*

**contains\_column**(*col*)

*inherited from the* [contains\_column()](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ColumnCollectionConstraint.contains_column" \o "sqlalchemy.schema.ColumnCollectionConstraint.contains_column) *method of* [ColumnCollectionConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ColumnCollectionConstraint" \o "sqlalchemy.schema.ColumnCollectionConstraint)

Return True if this constraint contains the given column.

Note that this object also contains an attribute .columns which is a [ColumnCollection](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnCollection" \o "sqlalchemy.sql.expression.ColumnCollection) of [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects.

**dialect\_kwargs**

*inherited from the* [dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs) *attribute of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

A collection of keyword arguments specified as dialect-specific options to this construct.

The arguments are present here in their original <dialect>\_<kwarg> format. Only arguments that were actually passed are included; unlike the [DialectKWArgs.dialect\_options](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_options" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_options) collection, which contains all options known by this dialect including defaults.

The collection is also writable; keys are accepted of the form <dialect>\_<kwarg> where the value will be assembled into the list of options.

*New in version 0.9.2.*

*Changed in version 0.9.4:*The [DialectKWArgs.dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs) collection is now writable.

**See also**

[DialectKWArgs.dialect\_options](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_options" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_options) - nested dictionary form

**dialect\_options**

*inherited from the* [dialect\_options](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_options" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_options) *attribute of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

A collection of keyword arguments specified as dialect-specific options to this construct.

This is a two-level nested registry, keyed to <dialect\_name> and <argument\_name>. For example, the postgresql\_where argument would be locatable as:

arg = my\_object.dialect\_options['postgresql']['where']

*New in version 0.9.2.*

**See also**

[DialectKWArgs.dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs) - flat dictionary form

**get\_children**(*\*\*kwargs*)

*inherited from the* [get\_children()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem.get_children" \o "sqlalchemy.schema.SchemaItem.get_children) *method of* [SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem)

used to allow SchemaVisitor access

**info**

*inherited from the* [info](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem.info" \o "sqlalchemy.schema.SchemaItem.info) *attribute of* [SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem)

Info dictionary associated with the object, allowing user-defined data to be associated with this [SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem).

The dictionary is automatically generated when first accessed. It can also be specified in the constructor of some objects, such as [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) and [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column).

**kwargs**

*inherited from the* [kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.kwargs) *attribute of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

A synonym for [DialectKWArgs.dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs).

**quote**

*inherited from the* [quote](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem.quote" \o "sqlalchemy.schema.SchemaItem.quote) *attribute of* [SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem)

Return the value of the quote flag passed to this schema object, for those schema items which have a name field.

*Deprecated since version 0.9:*Use <obj>.name.quote

sqlalchemy.schema.**conv**(*cls*, *value*, *quote=None*)

Mark a string indicating that a name has already been converted by a naming convention.

标记一个字符串，表示名称已经被命名约定转换了。

This is a string subclass that indicates a name that should not be subject to any further naming conventions.

这是一个字符串子类，它指示不应受到任何进一步命名约定的名称。

E.g. when we create a [Constraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.Constraint" \o "sqlalchemy.schema.Constraint) using a naming convention as follows:

例如。 当我们使用命名约定创建约束时，如下所示：

m = MetaData(naming\_convention={

"ck": "ck\_*%(table\_name)s*\_*%(constraint\_name)s*"})t = Table('t', m, Column('x', Integer),

CheckConstraint('x > 5', name='x5'))

The name of the above constraint will be rendered as "ck\_t\_x5". That is, the existing name x5 is used in the naming convention as the constraint\_nametoken.

上述约束的名称将被渲染为“ck\_t\_x5”。 也就是说，命名约定中使用现有名称x5作为constraint\_nametoken。

In some situations, such as in migration scripts, we may be rendering the above [CheckConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.CheckConstraint" \o "sqlalchemy.schema.CheckConstraint) with a name that's already been converted. In order to make sure the name isn't double-modified, the new name is applied using the [schema.conv()](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.conv" \o "sqlalchemy.schema.conv) marker. We can use this explicitly as follows:

在某些情况下，例如在迁移脚本中，我们可能正在使用已经转换的名称来呈现上面的CheckConstraint。 为了确保名称不被重复修改，使用schema.conv（）标记来应用新名称。 我们可以如下明确地使用它：

m = MetaData(naming\_convention={

"ck": "ck\_*%(table\_name)s*\_*%(constraint\_name)s*"})t = Table('t', m, Column('x', Integer),

CheckConstraint('x > 5', name=conv('ck\_t\_x5')))

Where above, the [schema.conv()](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.conv" \o "sqlalchemy.schema.conv) marker indicates that the constraint name here is final, and the name will render as "ck\_t\_x5" and not"ck\_t\_ck\_t\_x5"

在上面的地方，schema.conv()标记表明这里的约束名称是final的，名字将会呈现为“ck\_t\_x5”而不是“ck\_t\_ck\_t\_x5”

*New in version 0.9.4.*

**See also**

[Configuring Constraint Naming Conventions](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "constraint-naming-conventions)

3.4.8 Indexes

Indexes can be created anonymously (using an auto-generated name ix\_<column label>) for a single column using the inline index keyword on [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column), which also modifies the usage of unique to apply the uniqueness to the index itself, instead of adding a separate UNIQUE constraint. For indexes with specific names or which encompass more than one column, use the [Index](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.Index" \o "sqlalchemy.schema.Index) construct, which requires a name.

可以使用Column上的内联索引关键字匿名创建索引（使用自动生成的名称ix\_ <column label>），该列也修改将唯一性应用于索引本身的唯一性的使用，而不是添加 单独的UNIQUE约束。 对于具有特定名称或包含多个列的索引，请使用需要名称的Index构造。

Below we illustrate a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) with several [Index](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.Index" \o "sqlalchemy.schema.Index) objects associated. The DDL for "CREATE INDEX" is issued right after the create statements for the table:

下面我们来说明一个与几个Index对象关联的表。 “CREATE INDEX”的DDL正好在表的create语句之后发布：

meta = MetaData()mytable = Table('mytable', meta,

*# an indexed column, with index "ix\_mytable\_col1"*

Column('col1', Integer, index=True),

*# a uniquely indexed column with index "ix\_mytable\_col2"*

Column('col2', Integer, index=True, unique=True),

Column('col3', Integer),

Column('col4', Integer),

Column('col5', Integer),

Column('col6', Integer),

)

*# place an index on col3, col4*

Index('idx\_col34', mytable.c.col3, mytable.c.col4)

*# place a unique index on col5, col6*

Index('myindex', mytable.c.col5, mytable.c.col6, unique=True)

[SQL](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html)mytable.create(engine)

Note in the example above, the [Index](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.Index" \o "sqlalchemy.schema.Index) construct is created externally to the table which it corresponds, using [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects directly. [Index](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.Index" \o "sqlalchemy.schema.Index) also supports "inline" definition inside the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table), using string names to identify columns:

注意在上面的例子中，Index结构是直接在Column对象的外部创建的。 索引还支持表内的“内联”定义，使用字符串名称来标识列：

meta = MetaData()mytable = Table('mytable', meta,

Column('col1', Integer),

Column('col2', Integer),

Column('col3', Integer),

Column('col4', Integer),

*# place an index on col1, col2*

Index('idx\_col12', 'col1', 'col2'),

*# place a unique index on col3, col4*

Index('idx\_col34', 'col3', 'col4', unique=**True**))

*New in version 0.7:*Support of "inline" definition inside the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) for [Index](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.Index" \o "sqlalchemy.schema.Index).

The [Index](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.Index" \o "sqlalchemy.schema.Index) object also supports its own create() method:

i = Index('someindex', mytable.c.col5)[SQL](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html)i.create(engine)

### Functional Indexes

[Index](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.Index" \o "sqlalchemy.schema.Index) supports SQL and function expressions, as supported by the target backend. To create an index against a column using a descending value, the [ColumnElement.desc()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement.desc" \o "sqlalchemy.sql.expression.ColumnElement.desc) modifier may be used:

**from** **sqlalchemy** **import** Index

Index('someindex', mytable.c.somecol.desc())

Or with a backend that supports functional indexes such as PostgreSQL, a "case insensitive" index can be created using the lower() function:

**from** **sqlalchemy** **import** func, Index

Index('someindex', func.lower(mytable.c.somecol))

*New in version 0.8:*[Index](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.Index" \o "sqlalchemy.schema.Index) supports SQL expressions and functions as well as plain columns.

3.4.9 Index API

*class*sqlalchemy.schema.**Index**(*name*, *\*expressions*, *\*\*kw*)

Bases: [sqlalchemy.sql.base.DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs), [sqlalchemy.schema.ColumnCollectionMixin](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ColumnCollectionMixin" \o "sqlalchemy.schema.ColumnCollectionMixin), [sqlalchemy.schema.SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem)

A table-level INDEX.

Defines a composite (one or more column) INDEX.

E.g.:

sometable = Table("sometable", metadata,

Column("name", String(50)),

Column("address", String(100))

)

Index("some\_index", sometable.c.name)

For a no-frills, single column index, adding [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) also supports index=True:

sometable = Table("sometable", metadata,

Column("name", String(50), index=**True**)

)

For a composite index, multiple columns can be specified:

Index("some\_index", sometable.c.name, sometable.c.address)

Functional indexes are supported as well, typically by using the [func](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.func" \o "sqlalchemy.sql.expression.func) construct in conjunction with table-bound [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) objects:

Index("some\_index", func.lower(sometable.c.name))

*New in version 0.8:*support for functional and expression-based indexes.

An [Index](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.Index" \o "sqlalchemy.schema.Index) can also be manually associated with a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table), either through inline declaration or using [Table.append\_constraint()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.append_constraint" \o "sqlalchemy.schema.Table.append_constraint). When this approach is used, the names of the indexed columns can be specified as strings:

Table("sometable", metadata,

Column("name", String(50)),

Column("address", String(100)),

Index("some\_index", "name", "address")

)

To support functional or expression-based indexes in this form, the [text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text) construct may be used:

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

Table("sometable", metadata,

Column("name", String(50)),

Column("address", String(100)),

Index("some\_index", text("lower(name)"))

)

*New in version 0.9.5:*the [text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text) construct may be used to specify [Index](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.Index" \o "sqlalchemy.schema.Index) expressions, provided the [Index](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.Index" \o "sqlalchemy.schema.Index) is explicitly associated with the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table).

**See also**

[Indexes](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "schema-indexes) - General information on [Index](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.Index" \o "sqlalchemy.schema.Index).

[PostgreSQL-Specific Index Options](http://docs.sqlalchemy.org/en/rel_1_1/dialects/postgresql.html" \l "postgresql-indexes) - PostgreSQL-specific options available for the [Index](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.Index" \o "sqlalchemy.schema.Index) construct.

[MySQL Specific Index Options](http://docs.sqlalchemy.org/en/rel_1_1/dialects/mysql.html" \l "mysql-indexes) - MySQL-specific options available for the [Index](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.Index" \o "sqlalchemy.schema.Index) construct.

[Clustered Index Support](http://docs.sqlalchemy.org/en/rel_1_1/dialects/mssql.html" \l "mssql-indexes) - MSSQL-specific options available for the [Index](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.Index" \o "sqlalchemy.schema.Index) construct.

**\_\_init\_\_**(*name*, *\*expressions*, *\*\*kw*)

Construct an index object.

|  |  |
| --- | --- |
| **Parameters:** | * ****name**** – The name of the index * ****\*expressions**** – Column expressions to include in the index. The expressions are normally instances of [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column), but may also be arbitrary SQL expressions which ultimately refer to a [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column). * ****unique=False**** – Keyword only argument; if True, create a unique index. * ****quote=None**** – Keyword only argument; whether to apply quoting to the name of the index. Works in the same manner as that of [Column.quote](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column.params.quote" \o "sqlalchemy.schema.Column). * ****info=None –****Optional data dictionary which will be populated into the [SchemaItem.info](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem.info" \o "sqlalchemy.schema.SchemaItem.info) attribute of this object.   *New in version 1.0.0.*   * ****\*\*kw**** – Additional keyword arguments not mentioned above are dialect specific, and passed in the form <dialectname>\_<argname>. See the documentation regarding an individual dialect at [Dialects](http://docs.sqlalchemy.org/en/rel_1_1/dialects/index.html) for detail on documented arguments. |

**argument\_for**(*dialect\_name*, *argument\_name*, *default*)

*inherited from the* [argument\_for()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.argument_for" \o "sqlalchemy.sql.base.DialectKWArgs.argument_for) *method of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

Add a new kind of dialect-specific keyword argument for this class.

E.g.:

Index.argument\_for("mydialect", "length", **None**)

some\_index = Index('a', 'b', mydialect\_length=5)

The [DialectKWArgs.argument\_for()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.argument_for" \o "sqlalchemy.sql.base.DialectKWArgs.argument_for) method is a per-argument way adding extra arguments to the[DefaultDialect.construct\_arguments](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.default.DefaultDialect.construct_arguments" \o "sqlalchemy.engine.default.DefaultDialect.construct_arguments) dictionary. This dictionary provides a list of argument names accepted by various schema-level constructs on behalf of a dialect.

New dialects should typically specify this dictionary all at once as a data member of the dialect class. The use case for ad-hoc addition of argument names is typically for end-user code that is also using a custom compilation scheme which consumes the additional arguments.

|  |  |
| --- | --- |
| **Parameters:** | * ****dialect\_name**** – name of a dialect. The dialect must be locatable, else a [NoSuchModuleError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.NoSuchModuleError" \o "sqlalchemy.exc.NoSuchModuleError) is raised. The dialect must also include an existing[DefaultDialect.construct\_arguments](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.default.DefaultDialect.construct_arguments" \o "sqlalchemy.engine.default.DefaultDialect.construct_arguments) collection, indicating that it participates in the keyword-argument validation and default system, else [ArgumentError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.ArgumentError" \o "sqlalchemy.exc.ArgumentError) is raised. If the dialect does not include this collection, then any keyword argument can be specified on behalf of this dialect already. All dialects packaged within SQLAlchemy include this collection, however for third party dialects, support may vary. * ****argument\_name**** – name of the parameter. * ****default**** – default value of the parameter. |

*New in version 0.9.4.*

**bind**

Return the connectable associated with this Index.

**create**(*bind=None*)

Issue a CREATE statement for this [Index](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.Index" \o "sqlalchemy.schema.Index), using the given [Connectable](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connectable" \o "sqlalchemy.engine.Connectable) for connectivity.

**See also**

[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).

**dialect\_kwargs**

*inherited from the* [dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs) *attribute of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

A collection of keyword arguments specified as dialect-specific options to this construct.

The arguments are present here in their original <dialect>\_<kwarg> format. Only arguments that were actually passed are included; unlike the [DialectKWArgs.dialect\_options](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_options" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_options) collection, which contains all options known by this dialect including defaults.

The collection is also writable; keys are accepted of the form <dialect>\_<kwarg> where the value will be assembled into the list of options.

*New in version 0.9.2.*

*Changed in version 0.9.4:*The [DialectKWArgs.dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs) collection is now writable.

**See also**

[DialectKWArgs.dialect\_options](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_options" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_options) - nested dictionary form

**dialect\_options**

*inherited from the* [dialect\_options](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_options" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_options) *attribute of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

A collection of keyword arguments specified as dialect-specific options to this construct.

This is a two-level nested registry, keyed to <dialect\_name> and <argument\_name>. For example, the postgresql\_where argument would be locatable as:

arg = my\_object.dialect\_options['postgresql']['where']

*New in version 0.9.2.*

**See also**

[DialectKWArgs.dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs) - flat dictionary form

**drop**(*bind=None*)

Issue a DROP statement for this [Index](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.Index" \o "sqlalchemy.schema.Index), using the given [Connectable](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connectable" \o "sqlalchemy.engine.Connectable) for connectivity.

**See also**

[MetaData.drop\_all()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData.drop_all" \o "sqlalchemy.schema.MetaData.drop_all).

**get\_children**(*\*\*kwargs*)

*inherited from the* [get\_children()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem.get_children" \o "sqlalchemy.schema.SchemaItem.get_children) *method of* [SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem)

used to allow SchemaVisitor access

**info**

*inherited from the* [info](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem.info" \o "sqlalchemy.schema.SchemaItem.info) *attribute of* [SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem)

Info dictionary associated with the object, allowing user-defined data to be associated with this [SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem).

The dictionary is automatically generated when first accessed. It can also be specified in the constructor of some objects, such as [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) and [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column).

**kwargs**

*inherited from the* [kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.kwargs) *attribute of* [DialectKWArgs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs" \o "sqlalchemy.sql.base.DialectKWArgs)

A synonym for [DialectKWArgs.dialect\_kwargs](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs" \o "sqlalchemy.sql.base.DialectKWArgs.dialect_kwargs).

**quote**

*inherited from the* [quote](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem.quote" \o "sqlalchemy.schema.SchemaItem.quote) *attribute of* [SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem)

Return the value of the quote flag passed to this schema object, for those schema items which have a name field.

*Deprecated since version 0.9:*Use <obj>.name.quote

## 3.5 Customizing DDL

In the preceding sections we've discussed a variety of schema constructs including [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table), [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint), [CheckConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.CheckConstraint" \o "sqlalchemy.schema.CheckConstraint), and [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence). Throughout, we've relied upon the create() and [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) methods of [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) and [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) in order to issue data definition language (DDL) for all constructs. When issued, a pre-determined order of operations is invoked, and DDL to create each table is created unconditionally including all constraints and other objects associated with it. For more complex scenarios where database-specific DDL is required, SQLAlchemy offers two techniques which can be used to add any DDL based on any condition, either accompanying the standard generation of tables or by itself.

3.5.1 Custom DDL

Custom DDL phrases are most easily achieved using the [DDL](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.DDL" \o "sqlalchemy.schema.DDL) construct. This construct works like all the other DDL elements except it accepts a string which is the text to be emitted:

event.listen(

metadata,

"after\_create",

DDL("ALTER TABLE users ADD CONSTRAINT "

"cst\_user\_name\_length "

" CHECK (length(user\_name) >= 8)"))

A more comprehensive method of creating libraries of DDL constructs is to use custom compilation - see [Custom SQL Constructs and Compilation Extension](http://docs.sqlalchemy.org/en/rel_1_1/core/compiler.html) for details.

3.5.2 Controlling DDL Sequences

The [DDL](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.DDL" \o "sqlalchemy.schema.DDL) construct introduced previously also has the ability to be invoked conditionally based on inspection of the database. This feature is available using the [DDLElement.execute\_if()](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.DDLElement.execute_if" \o "sqlalchemy.schema.DDLElement.execute_if) method. For example, if we wanted to create a trigger but only on the PostgreSQL backend, we could invoke this as:

mytable = Table(

'mytable', metadata,

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

Column('data', String(50)))

trigger = DDL(

"CREATE TRIGGER dt\_ins BEFORE INSERT ON mytable "

"FOR EACH ROW BEGIN SET NEW.data='ins'; END")

event.listen(

mytable,

'after\_create',

trigger.execute\_if(dialect='postgresql'))

The [DDLElement.execute\_if.dialect](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.DDLElement.execute_if.params.dialect" \o "sqlalchemy.schema.DDLElement.execute_if) keyword also accepts a tuple of string dialect names:

event.listen(

mytable,

"after\_create",

trigger.execute\_if(dialect=('postgresql', 'mysql')))event.listen(

mytable,

"before\_drop",

trigger.execute\_if(dialect=('postgresql', 'mysql')))

The [DDLElement.execute\_if()](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.DDLElement.execute_if" \o "sqlalchemy.schema.DDLElement.execute_if) method can also work against a callable function that will receive the database connection in use. In the example below, we use this to conditionally create a CHECK constraint, first looking within the PostgreSQL catalogs to see if it exists:

**def** should\_create(ddl, target, connection, \*\*kw):

row = connection.execute(

"select conname from pg\_constraint where conname='*%s*'" %

ddl.element.name).scalar()

**return** **not** bool(row)

**def** should\_drop(ddl, target, connection, \*\*kw):

**return** **not** should\_create(ddl, target, connection, \*\*kw)

event.listen(

users,

"after\_create",

DDL(

"ALTER TABLE users ADD CONSTRAINT "

"cst\_user\_name\_length CHECK (length(user\_name) >= 8)"

).execute\_if(callable\_=should\_create))event.listen(

users,

"before\_drop",

DDL(

"ALTER TABLE users DROP CONSTRAINT cst\_user\_name\_length"

).execute\_if(callable\_=should\_drop))

users.create(engine)

CREATE TABLE users (

user\_id SERIAL NOT NULL,

user\_name VARCHAR(40) NOT NULL,

PRIMARY KEY (user\_id)

)

select conname from pg\_constraint where conname='cst\_user\_name\_length'

ALTER TABLE users ADD CONSTRAINT cst\_user\_name\_length CHECK (length(user\_name) >= 8)

users.drop(engine)

select conname from pg\_constraint where conname='cst\_user\_name\_length'

ALTER TABLE users DROP CONSTRAINT cst\_user\_name\_length

DROP TABLE users

### 3.5.3 Using the built-in DDLElement Classes

The sqlalchemy.schema package contains SQL expression constructs that provide DDL expressions. For example, to produce a CREATE TABLE statement:

**from** **sqlalchemy.schema** **import** CreateTable

engine.execute(CreateTable(mytable))

CREATE TABLE mytable (

col1 INTEGER,

col2 INTEGER,

col3 INTEGER,

col4 INTEGER,

col5 INTEGER,

col6 INTEGER

)

Above, the [CreateTable](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.CreateTable" \o "sqlalchemy.schema.CreateTable) construct works like any other expression construct (such as select(), table.insert(), etc.). All of SQLAlchemy's DDL oriented constructs are subclasses of the [DDLElement](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.DDLElement" \o "sqlalchemy.schema.DDLElement) base class; this is the base of all the objects corresponding to CREATE and DROP as well as ALTER, not only in SQLAlchemy but in Alembic Migrations as well. A full reference of available constructs is in [DDL Expression Constructs API](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "schema-api-ddl).

User-defined DDL constructs may also be created as subclasses of [DDLElement](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.DDLElement" \o "sqlalchemy.schema.DDLElement) itself. The documentation in [Custom SQL Constructs and Compilation Extension](http://docs.sqlalchemy.org/en/rel_1_1/core/compiler.html) has several examples of this.

The event-driven DDL system described in the previous section [Controlling DDL Sequences](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "schema-ddl-sequences) is available with other [DDLElement](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.DDLElement" \o "sqlalchemy.schema.DDLElement) objects as well. However, when dealing with the built-in constructs such as [CreateIndex](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.CreateIndex" \o "sqlalchemy.schema.CreateIndex), [CreateSequence](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.CreateSequence" \o "sqlalchemy.schema.CreateSequence), etc, the event system is of ****limited**** use, as methods like [Table.create()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.create" \o "sqlalchemy.schema.Table.create) and[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) will invoke these constructs unconditionally. In a future SQLAlchemy release, the DDL event system including conditional execution will taken into account for built-in constructs that currently invoke in all cases.

We can illustrate an event-driven example with the [AddConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.AddConstraint" \o "sqlalchemy.schema.AddConstraint) and [DropConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.DropConstraint" \o "sqlalchemy.schema.DropConstraint) constructs, as the event-driven system will work for CHECK and UNIQUE constraints, using these as we did in our previous example of [DDLElement.execute\_if()](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.DDLElement.execute_if" \o "sqlalchemy.schema.DDLElement.execute_if):

**def** should\_create(ddl, target, connection, \*\*kw):

row = connection.execute(

"select conname from pg\_constraint where conname='*%s*'" %

ddl.element.name).scalar()

**return** **not** bool(row)

**def** should\_drop(ddl, target, connection, \*\*kw):

**return** **not** should\_create(ddl, target, connection, \*\*kw)

event.listen(

users,

"after\_create",

AddConstraint(constraint).execute\_if(callable\_=should\_create))event.listen(

users,

"before\_drop",

DropConstraint(constraint).execute\_if(callable\_=should\_drop))

users.create(engine)

CREATE TABLE users (

user\_id SERIAL NOT NULL,

user\_name VARCHAR(40) NOT NULL,

PRIMARY KEY (user\_id)

)

select conname from pg\_constraint where conname='cst\_user\_name\_length'

ALTER TABLE users ADD CONSTRAINT cst\_user\_name\_length CHECK (length(user\_name) >= 8)

users.drop(engine)

select conname from pg\_constraint where conname='cst\_user\_name\_length'

ALTER TABLE users DROP CONSTRAINT cst\_user\_name\_length

DROP TABLE users

While the above example is against the built-in [AddConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.AddConstraint" \o "sqlalchemy.schema.AddConstraint) and [DropConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.DropConstraint" \o "sqlalchemy.schema.DropConstraint) objects, the main usefulness of DDL events for now remains focused on the use of the [DDL](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.DDL" \o "sqlalchemy.schema.DDL) construct itself, as well as with user-defined subclasses of [DDLElement](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.DDLElement" \o "sqlalchemy.schema.DDLElement) that aren't already part of 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), [Table.create()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.create" \o "sqlalchemy.schema.Table.create), and corresponding "drop" processes.

3.5.4 DDL Expression Constructs API

sqlalchemy.schema.**sort\_tables**(*tables*, *skip\_fn=None*, *extra\_dependencies=None*)

sort a collection of [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects based on dependency.

This is a dependency-ordered sort which will emit [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects such that they will follow their dependent [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects. Tables are dependent on another based on the presence of [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint) objects as well as explicit dependencies added by [Table.add\_is\_dependent\_on()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.add_is_dependent_on" \o "sqlalchemy.schema.Table.add_is_dependent_on).

**Warning**

The [sort\_tables()](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.sort_tables" \o "sqlalchemy.schema.sort_tables) function cannot by itself accommodate automatic resolution of dependency cycles between tables, which are usually caused by mutually dependent foreign key constraints. To resolve these cycles, either the [ForeignKeyConstraint.use\_alter](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint.params.use_alter" \o "sqlalchemy.schema.ForeignKeyConstraint) parameter may be appled to those constraints, or use thesql.sort\_tables\_and\_constraints() function which will break out foreign key constraints involved in cycles separately.

|  |  |
| --- | --- |
| **Parameters:** | * ****tables**** – a sequence of [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects. * ****skip\_fn**** – optional callable which will be passed a [ForeignKey](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey) object; if it returns True, this constraint will not be considered as a dependency. Note this is****different**** from the same parameter in [sort\_tables\_and\_constraints()](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.sort_tables_and_constraints" \o "sqlalchemy.schema.sort_tables_and_constraints), which is instead passed the owning [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint) object. * ****extra\_dependencies**** – a sequence of 2-tuples of tables which will also be considered as dependent on each other. |

**See also**

[sort\_tables\_and\_constraints()](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.sort_tables_and_constraints" \o "sqlalchemy.schema.sort_tables_and_constraints)

MetaData.sorted\_tables() - uses this function to sort

sqlalchemy.schema.**sort\_tables\_and\_constraints**(*tables*, *filter\_fn=None*, *extra\_dependencies=None*)

sort a collection of [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) / [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint) objects.

This is a dependency-ordered sort which will emit tuples of (Table, [ForeignKeyConstraint, ...]) such that each [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) follows its dependent [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects. Remaining [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint) objects that are separate due to dependency rules not satisfied by the sort are emitted afterwards as (None, [ForeignKeyConstraint ...]).

Tables are dependent on another based on the presence of [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint) objects, explicit dependencies added by [Table.add\_is\_dependent\_on()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.add_is_dependent_on" \o "sqlalchemy.schema.Table.add_is_dependent_on), as well as dependencies stated here using the [skip\_fn](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.sort_tables_and_constraints.params.skip_fn" \o "sqlalchemy.schema.sort_tables_and_constraints) and/or [extra\_dependencies](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.sort_tables_and_constraints.params.extra_dependencies" \o "sqlalchemy.schema.sort_tables_and_constraints) parameters.

|  |  |
| --- | --- |
| **Parameters:** | * ****tables**** – a sequence of [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects. * ****filter\_fn**** – optional callable which will be passed a [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint) object, and returns a value based on whether this constraint should definitely be included or excluded as an inline constraint, or neither. If it returns False, the constraint will definitely be included as a dependency that cannot be subject to ALTER; if True, it will ****only**** be included as an ALTER result at the end. Returning None means the constraint is included in the table-based result unless it is detected as part of a dependency cycle. * ****extra\_dependencies**** – a sequence of 2-tuples of tables which will also be considered as dependent on each other. |

*New in version 1.0.0.*

**See also**

[sort\_tables()](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.sort_tables" \o "sqlalchemy.schema.sort_tables)

*class*sqlalchemy.schema.**DDLElement**

Bases: [sqlalchemy.sql.expression.Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable), sqlalchemy.schema.\_DDLCompiles

Base class for DDL expression constructs.

This class is the base for the general purpose [DDL](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.DDL" \o "sqlalchemy.schema.DDL) class, as well as the various create/drop clause constructs such as [CreateTable](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.CreateTable" \o "sqlalchemy.schema.CreateTable), [DropTable](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.DropTable" \o "sqlalchemy.schema.DropTable), [AddConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.AddConstraint" \o "sqlalchemy.schema.AddConstraint), etc.

[DDLElement](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.DDLElement" \o "sqlalchemy.schema.DDLElement) integrates closely with SQLAlchemy events, introduced in [Events](http://docs.sqlalchemy.org/en/rel_1_1/core/event.html). An instance of one is itself an event receiving callable:

event.listen(

users,

'after\_create',

AddConstraint(constraint).execute\_if(dialect='postgresql'))

**See also**

[DDL](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.DDL" \o "sqlalchemy.schema.DDL)

[DDLEvents](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.DDLEvents" \o "sqlalchemy.events.DDLEvents)

[Events](http://docs.sqlalchemy.org/en/rel_1_1/core/event.html)

[Controlling DDL Sequences](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "schema-ddl-sequences)

**\_\_call\_\_**(*target*, *bind*, *\*\*kw*)

Execute the DDL as a ddl\_listener.

**against**(*target*)

Return a copy of this DDL against a specific schema item.

**bind**

**callable\_***= None*

**dialect***= None*

**execute**(*bind=None*, *target=None*)

Execute this DDL immediately.

Executes the DDL statement in isolation using the supplied [Connectable](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connectable" \o "sqlalchemy.engine.Connectable) or [Connectable](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connectable" \o "sqlalchemy.engine.Connectable) assigned to the .bind property, if not supplied. If the DDL has a conditional on criteria, it will be invoked with None as the event.

|  |  |
| --- | --- |
| **Parameters:** | * ****bind**** – Optional, an Engine or Connection. If not supplied, a valid [Connectable](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connectable" \o "sqlalchemy.engine.Connectable) must be present in the .bind property. * ****target**** – Optional, defaults to None. The target SchemaItem for the execute call. Will be passed to the on callable if any, and may also provide string expansion data for the statement. See execute\_at for more information. |

**execute\_at**(*event\_name*, *target*)

Link execution of this DDL to the DDL lifecycle of a SchemaItem.

*Deprecated since version 0.7:*See [DDLEvents](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.DDLEvents" \o "sqlalchemy.events.DDLEvents), as well as [DDLElement.execute\_if()](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.DDLElement.execute_if" \o "sqlalchemy.schema.DDLElement.execute_if).

Links this DDLElement to a Table or MetaData instance, executing it when that schema item is created or dropped. The DDL statement will be executed using the same Connection and transactional context as the Table create/drop itself. The .bind property of this statement is ignored.

|  |  |
| --- | --- |
| **Parameters:** | * ****event**** – One of the events defined in the schema item's .ddl\_events; e.g. 'before-create', 'after-create', 'before-drop' or 'after-drop' * ****target**** – The Table or MetaData instance for which this DDLElement will be associated with. |

A DDLElement instance can be linked to any number of schema items.

execute\_at builds on the append\_ddl\_listener interface of [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) and [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects.

Caveat: Creating or dropping a Table in isolation will also trigger any DDL set to execute\_at that Table's MetaData. This may change in a future release.

**execute\_if**(*dialect=None*, *callable\_=None*, *state=None*)

Return a callable that will execute this DDLElement conditionally.

Used to provide a wrapper for event listening:

event.listen(

metadata,

'before\_create',

DDL("my\_ddl").execute\_if(dialect='postgresql')

)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Parameters:** | * ****dialect**** –   May be a string, tuple or a callable predicate. If a string, it will be compared to the name of the executing database dialect:  DDL('something').execute\_if(dialect='postgresql')  If a tuple, specifies multiple dialect names:  DDL('something').execute\_if(dialect=('postgresql', 'mysql'))   * ****callable\_**** –   A callable, which will be invoked with four positional arguments as well as optional keyword arguments:   |  |  | | --- | --- | | **ddl:** | This DDL element. | | **target:** | The [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) or [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) object which is the target of this event. May be None if the DDL is executed explicitly. | | **bind:** | The [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) being used for DDL execution | | **tables:** | Optional keyword argument - a list of Table objects which are to be created/ dropped within a MetaData.create\_all() or drop\_all() method call. | | **state:** | Optional keyword argument - will be the state argument passed to this function. | | **checkfirst:** | Keyword argument, will be True if the 'checkfirst' flag was set during the call to create(), create\_all(), drop(), drop\_all(). |   If the callable returns a true value, the DDL statement will be executed.   * ****state**** – any value which will be passed to the callable\_ as the state keyword argument. |

**See also**

[DDLEvents](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.DDLEvents" \o "sqlalchemy.events.DDLEvents)

[Events](http://docs.sqlalchemy.org/en/rel_1_1/core/event.html)

**on***= None*

**target***= None*

*class*sqlalchemy.schema.**DDL**(*statement*, *on=None*, *context=None*, *bind=None*)

Bases: [sqlalchemy.schema.DDLElement](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.DDLElement" \o "sqlalchemy.schema.DDLElement)

A literal DDL statement.

Specifies literal SQL DDL to be executed by the database. DDL objects function as DDL event listeners, and can be subscribed to those events listed in [DDLEvents](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.DDLEvents" \o "sqlalchemy.events.DDLEvents), using either [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) or [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) objects as targets. Basic templating support allows a single DDL instance to handle repetitive tasks for multiple tables.

Examples:

**from** **sqlalchemy** **import** event, DDL

tbl = Table('users', metadata, Column('uid', Integer))event.listen(tbl, 'before\_create', DDL('DROP TRIGGER users\_trigger'))

spow = DDL('ALTER TABLE *%(table)s* SET secretpowers TRUE')event.listen(tbl, 'after\_create', spow.execute\_if(dialect='somedb'))

drop\_spow = DDL('ALTER TABLE users SET secretpowers FALSE')connection.execute(drop\_spow)

When operating on Table events, the following statement string substitions are available:

%(table)s - the Table name, **with** any required quoting applied%(schema)s - the schema name, **with** any required quoting applied%(fullname)s - the Table name including schema, quoted **if** needed

The DDL's "context", if any, will be combined with the standard substitutions noted above. Keys present in the context will override the standard substitutions.

**\_\_init\_\_**(*statement*, *on=None*, *context=None*, *bind=None*)

Create a DDL statement.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Parameters:** | * ****statement**** –   A string or unicode string to be executed. Statements will be processed with Python's string formatting operator. See the context argument and the execute\_at method.  A literal '%' in a statement must be escaped as '%%'.  SQL bind parameters are not available in DDL statements.   * ****on**** –   *Deprecated since version 0.7:*See [DDLElement.execute\_if()](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.DDLElement.execute_if" \o "sqlalchemy.schema.DDLElement.execute_if).  Optional filtering criteria. May be a string, tuple or a callable predicate. If a string, it will be compared to the name of the executing database dialect:  DDL('something', on='postgresql')  If a tuple, specifies multiple dialect names:  DDL('something', on=('postgresql', 'mysql'))  If a callable, it will be invoked with four positional arguments as well as optional keyword arguments:   |  |  | | --- | --- | | **ddl:** | This DDL element. | | **event:** | The name of the event that has triggered this DDL, such as 'after-create' Will be None if the DDL is executed explicitly. | | **target:** | The Table or MetaData object which is the target of this event. May be None if the DDL is executed explicitly. | | **connection:** | The Connection being used for DDL execution | | **tables:** | Optional keyword argument - a list of Table objects which are to be created/ dropped within a MetaData.create\_all() or drop\_all() method call. |   If the callable returns a true value, the DDL statement will be executed.   * ****context**** – Optional dictionary, defaults to None. These values will be available for use in string substitutions on the DDL statement. * ****bind**** – Optional. A [Connectable](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connectable" \o "sqlalchemy.engine.Connectable), used by default when execute() is invoked without a bind argument. |

**See also**

[DDLEvents](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.DDLEvents" \o "sqlalchemy.events.DDLEvents)

[Events](http://docs.sqlalchemy.org/en/rel_1_1/core/event.html)

*class*sqlalchemy.schema.**\_CreateDropBase**(*element*, *on=None*, *bind=None*)

Bases: [sqlalchemy.schema.DDLElement](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.DDLElement" \o "sqlalchemy.schema.DDLElement)

Base class for DDL constructs that represent CREATE and DROP or equivalents.

The common theme of \_CreateDropBase is a single element attribute which refers to the element to be created or dropped.

*class*sqlalchemy.schema.**CreateTable**(*element*, *on=None*, *bind=None*, *include\_foreign\_key\_constraints=None*)

Bases: [sqlalchemy.schema.\_CreateDropBase](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema._CreateDropBase" \o "sqlalchemy.schema._CreateDropBase)

Represent a CREATE TABLE statement.

**\_\_init\_\_**(*element*, *on=None*, *bind=None*, *include\_foreign\_key\_constraints=None*)

Create a [CreateTable](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.CreateTable" \o "sqlalchemy.schema.CreateTable) construct.

|  |  |
| --- | --- |
| **Parameters:** | * ****element**** – a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) that's the subject of the CREATE * ****on**** – See the description for 'on' in [DDL](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.DDL" \o "sqlalchemy.schema.DDL). * ****bind**** – See the description for 'bind' in [DDL](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.DDL" \o "sqlalchemy.schema.DDL). * ****include\_foreign\_key\_constraints**** –   optional sequence of [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint) objects that will be included inline within the CREATE construct; if omitted, all foreign key constraints that do not specify use\_alter=True are included.  *New in version 1.0.0.* |

*class*sqlalchemy.schema.**DropTable**(*element*, *on=None*, *bind=None*)

Bases: [sqlalchemy.schema.\_CreateDropBase](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema._CreateDropBase" \o "sqlalchemy.schema._CreateDropBase)

Represent a DROP TABLE statement.

*class*sqlalchemy.schema.**CreateColumn**(*element*)

Bases: sqlalchemy.schema.\_DDLCompiles

Represent a [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) as rendered in a CREATE TABLE statement, via the [CreateTable](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.CreateTable" \o "sqlalchemy.schema.CreateTable) construct.

This is provided to support custom column DDL within the generation of CREATE TABLE statements, by using the compiler extension documented in [Custom SQL Constructs and Compilation Extension](http://docs.sqlalchemy.org/en/rel_1_1/core/compiler.html) to extend [CreateColumn](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.CreateColumn" \o "sqlalchemy.schema.CreateColumn).

Typical integration is to examine the incoming [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) object, and to redirect compilation if a particular flag or condition is found:

**from** **sqlalchemy** **import** schema**from** **sqlalchemy.ext.compiler** **import** compiles

**@compiles**(schema.CreateColumn)**def** compile(element, compiler, \*\*kw):

column = element.element

**if** "special" **not** **in** column.info:

**return** compiler.visit\_create\_column(element, \*\*kw)

text = "*%s* SPECIAL DIRECTIVE *%s*" % (

column.name,

compiler.type\_compiler.process(column.type)

)

default = compiler.get\_column\_default\_string(column)

**if** default **is** **not** **None**:

text += " DEFAULT " + default

**if** **not** column.nullable:

text += " NOT NULL"

**if** column.constraints:

text += " ".join(

compiler.process(const)

**for** const **in** column.constraints)

**return** text

The above construct can be applied to a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) as follows:

**from** **sqlalchemy** **import** Table, Metadata, Column, Integer, String**from** **sqlalchemy** **import** schema

metadata = MetaData()

table = Table('mytable', MetaData(),

Column('x', Integer, info={"special":**True**}, primary\_key=**True**),

Column('y', String(50)),

Column('z', String(20), info={"special":**True**})

)

metadata.create\_all(conn)

Above, the directives we've added to the [Column.info](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column.info" \o "sqlalchemy.schema.Column.info) collection will be detected by our custom compilation scheme:

CREATE TABLE mytable (

x SPECIAL DIRECTIVE INTEGER NOT NULL,

y VARCHAR(50),

z SPECIAL DIRECTIVE VARCHAR(20),

PRIMARY KEY (x))

The [CreateColumn](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.CreateColumn" \o "sqlalchemy.schema.CreateColumn) construct can also be used to skip certain columns when producing a CREATE TABLE. This is accomplished by creating a compilation rule that conditionally returns None. This is essentially how to produce the same effect as using the system=True argument on [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column), which marks a column as an implicitly-present "system" column.

For example, suppose we wish to produce a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) which skips rendering of the PostgreSQL xmin column against the PostgreSQL backend, but on other backends does render it, in anticipation of a triggered rule. A conditional compilation rule could skip this name only on PostgreSQL:

**from** **sqlalchemy.schema** **import** CreateColumn

**@compiles**(CreateColumn, "postgresql")**def** skip\_xmin(element, compiler, \*\*kw):

**if** element.element.name == 'xmin':

**return** **None**

**else**:

**return** compiler.visit\_create\_column(element, \*\*kw)

my\_table = Table('mytable', metadata,

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

Column('xmin', Integer)

)

Above, a [CreateTable](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.CreateTable" \o "sqlalchemy.schema.CreateTable) construct will generate a CREATE TABLE which only includes the id column in the string; the xmin column will be omitted, but only against the PostgreSQL backend.

*New in version 0.8.3:*The [CreateColumn](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.CreateColumn" \o "sqlalchemy.schema.CreateColumn) construct supports skipping of columns by returning None from a custom compilation rule.

*New in version 0.8:*The [CreateColumn](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.CreateColumn" \o "sqlalchemy.schema.CreateColumn) construct was added to support custom column creation styles.

*class*sqlalchemy.schema.**CreateSequence**(*element*, *on=None*, *bind=None*)

Bases: [sqlalchemy.schema.\_CreateDropBase](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema._CreateDropBase" \o "sqlalchemy.schema._CreateDropBase)

Represent a CREATE SEQUENCE statement.

*class*sqlalchemy.schema.**DropSequence**(*element*, *on=None*, *bind=None*)

Bases: [sqlalchemy.schema.\_CreateDropBase](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema._CreateDropBase" \o "sqlalchemy.schema._CreateDropBase)

Represent a DROP SEQUENCE statement.

*class*sqlalchemy.schema.**CreateIndex**(*element*, *on=None*, *bind=None*)

Bases: [sqlalchemy.schema.\_CreateDropBase](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema._CreateDropBase" \o "sqlalchemy.schema._CreateDropBase)

Represent a CREATE INDEX statement.

*class*sqlalchemy.schema.**DropIndex**(*element*, *on=None*, *bind=None*)

Bases: [sqlalchemy.schema.\_CreateDropBase](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema._CreateDropBase" \o "sqlalchemy.schema._CreateDropBase)

Represent a DROP INDEX statement.

*class*sqlalchemy.schema.**AddConstraint**(*element*, *\*args*, *\*\*kw*)

Bases: [sqlalchemy.schema.\_CreateDropBase](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema._CreateDropBase" \o "sqlalchemy.schema._CreateDropBase)

Represent an ALTER TABLE ADD CONSTRAINT statement.

*class*sqlalchemy.schema.**DropConstraint**(*element*, *cascade=False*, *\*\*kw*)

Bases: [sqlalchemy.schema.\_CreateDropBase](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema._CreateDropBase" \o "sqlalchemy.schema._CreateDropBase)

Represent an ALTER TABLE DROP CONSTRAINT statement.

*class*sqlalchemy.schema.**CreateSchema**(*name*, *quote=None*, *\*\*kw*)

Bases: [sqlalchemy.schema.\_CreateDropBase](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema._CreateDropBase" \o "sqlalchemy.schema._CreateDropBase)

Represent a CREATE SCHEMA statement.

*New in version 0.7.4.*

The argument here is the string name of the schema.

**\_\_init\_\_**(*name*, *quote=None*, *\*\*kw*)

Create a new [CreateSchema](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.CreateSchema" \o "sqlalchemy.schema.CreateSchema) construct.

*class*sqlalchemy.schema.**DropSchema**(*name*, *quote=None*, *cascade=False*, *\*\*kw*)

Bases: [sqlalchemy.schema.\_CreateDropBase](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema._CreateDropBase" \o "sqlalchemy.schema._CreateDropBase)

Represent a DROP SCHEMA statement.

The argument here is the string name of the schema.

*New in version 0.7.4.*

**\_\_init\_\_**(*name*, *quote=None*, *cascade=False*, *\*\*kw*)

Create a new [DropSchema](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.DropSchema" \o "sqlalchemy.schema.DropSchema) construct.

# Chapter 4 Column and Data Types

# **4.1 Column and Data Types**

SQLAlchemy provides abstractions for most common database data types, and a mechanism for specifying your own custom data types.

SQLAlchemy提供了大多数常见数据库数据类型的抽象，以及用于指定自己的自定义数据类型的机制。

The methods and attributes of type objects are rarely used directly. Type objects are supplied to [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) definitions and can be supplied as type hints to functions for occasions where the database driver returns an incorrect type.

类型对象的方法和属性很少直接使用。 类型对象被提供给表定义，并且可以以数据库驱动程序返回不正确类型的场合作为类型提示。

**>>>** users = Table('users', metadata,

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

**...**  Column('login', String(32))

**...**  )

SQLAlchemy will use the Integer and String(32) type information when issuing a CREATE TABLE statement and will use it again when reading back rows SELECTed from the database. Functions that accept a type (such as [Column()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column)) will typically accept a type class or instance; Integer is equivalent to Integer()with no construction arguments in this case.

SQLAlchemy将在发出CREATE TABLE语句时使用Integer和String（32）类型的信息，并在读取从数据库中选择的行时再次使用它。 接受类型的函数（如Column() ）通常会接受类型类或实例; 在这种情况下，整数相当于没有构造参数的Integer() 。

4.1.1 Generic Types

Generic types specify a column that can read, write and store a particular type of Python data. SQLAlchemy will choose the best database column type available on the target database when issuing a CREATE TABLE statement. For complete control over which column type is emitted in CREATE TABLE, such as VARCHAR see [SQL Standard and Multiple Vendor Types](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "types-sqlstandard) and the other sections of this chapter.

通用类型指定可以读取，写入和存储特定类型的Python数据的列。 SQLAlchemy将在发出CREATE TABLE语句时选择目标数据库上可用的最佳数据库列类型。 要完整控制在CREATE TABLE中发出的列类型，例如VARCHAR，请参阅SQL标准和多个供应商类型以及本章的其他部分。

*class*sqlalchemy.types.**BigInteger**

Bases: [sqlalchemy.types.Integer](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Integer" \o "sqlalchemy.types.Integer)

A type for bigger int integers.

Typically generates a BIGINT in DDL, and otherwise acts like a normal [Integer](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Integer" \o "sqlalchemy.types.Integer) on the Python side.

*class*sqlalchemy.types.**Boolean**(*create\_constraint=True*, *name=None*, *\_create\_events=True*)

Bases: [sqlalchemy.types.TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine), [sqlalchemy.types.SchemaType](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.SchemaType" \o "sqlalchemy.types.SchemaType)

A bool datatype.

Boolean typically uses BOOLEAN or SMALLINT on the DDL side, and on the Python side deals in True or False.

**\_\_init\_\_**(*create\_constraint=True*, *name=None*, *\_create\_events=True*)

Construct a Boolean.

|  |  |
| --- | --- |
| **Parameters:** | ****create\_constraint**** – defaults to True. If the boolean is generated as an int/smallint, also create a CHECK constraint on the table that ensures 1 or 0 as a value.  ****name**** – if a CHECK constraint is generated, specify the name of the constraint. |

*class*sqlalchemy.types.**Date**

Bases: sqlalchemy.types.\_DateAffinity, [sqlalchemy.types.TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine)

A type for datetime.date() objects.

*class*sqlalchemy.types.**DateTime**(*timezone=False*)

Bases: sqlalchemy.types.\_DateAffinity, [sqlalchemy.types.TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine)

A type for datetime.datetime() objects.

Date and time types return objects from the Python datetime module. Most DBAPIs have built in support for the datetime module, with the noted exception of SQLite. In the case of SQLite, date and time types are stored as strings which are then converted back to datetime objects when rows are returned.

For the time representation within the datetime type, some backends include additional options, such as timezone support and fractional seconds support. For fractional seconds, use the dialect-specific datatype, such as [mysql.TIME](http://docs.sqlalchemy.org/en/rel_1_1/dialects/mysql.html" \l "sqlalchemy.dialects.mysql.TIME" \o "sqlalchemy.dialects.mysql.TIME). For timezone support, use at least the [TIMESTAMP](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.TIMESTAMP" \o "sqlalchemy.types.TIMESTAMP) datatype, if not the dialect-specific datatype object.

**\_\_init\_\_**(*timezone=False*)

Construct a new [DateTime](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.DateTime" \o "sqlalchemy.types.DateTime).

|  |  |
| --- | --- |
| **Parameters:** | ****timezone**** – boolean. Indicates that the datetime type should enable timezone support, if available on the ****base date/time-holding type only****. It is recommended to make use of the [TIMESTAMP](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.TIMESTAMP" \o "sqlalchemy.types.TIMESTAMP) datatype directly when using this flag, as some databases include separate generic date/time-holding types distinct from the timezone-capable TIMESTAMP datatype, such as Oracle. |

*class*sqlalchemy.types.**Enum**(*\*enums*, *\*\*kw*)

Bases: [sqlalchemy.types.String](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.String" \o "sqlalchemy.types.String), [sqlalchemy.types.SchemaType](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.SchemaType" \o "sqlalchemy.types.SchemaType)

Generic Enum Type.

The [Enum](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Enum" \o "sqlalchemy.types.Enum) type provides a set of possible string values which the column is constrained towards.

The [Enum](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Enum" \o "sqlalchemy.types.Enum) type will make use of the backend's native "ENUM" type if one is available; otherwise, it uses a VARCHAR datatype and produces a CHECK constraint. Use of the backend-native enum type can be disabled using the [Enum.native\_enum](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Enum.params.native_enum" \o "sqlalchemy.types.Enum) flag, and the production of the CHECK constraint is configurable using the [Enum.create\_constraint](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Enum.params.create_constraint" \o "sqlalchemy.types.Enum) flag.

The [Enum](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Enum" \o "sqlalchemy.types.Enum) type also provides in-Python validation of string values during both read and write operations. When reading a value from the database in a result set, the string value is always checked against the list of possible values and a LookupError is raised if no match is found. When passing a value to the database as a plain string within a SQL statement, if the [Enum.validate\_strings](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Enum.params.validate_strings" \o "sqlalchemy.types.Enum) parameter is set to True, a LookupError is raised for any string value that's not located in the given list of possible values; note that this impacts usage of LIKE expressions with enumerated values (an unusual use case).

*Changed in version 1.1:*the [Enum](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Enum" \o "sqlalchemy.types.Enum) type now provides in-Python validation of input values as well as on data being returned by the database.

The source of enumerated values may be a list of string values, or alternatively a PEP-435-compliant enumerated class. For the purposes of the [Enum](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Enum" \o "sqlalchemy.types.Enum) datatype, this class need only provide a \_\_members\_\_ method.

When using an enumerated class, the enumerated objects are used both for input and output, rather than strings as is the case with a plain-string enumerated type:

**import** **enum**

**class** **MyEnum**(enum.Enum):

one = 1

two = 2

three = 3

t = Table(

'data', MetaData(),

Column('value', Enum(MyEnum)))

connection.execute(t.insert(), {"value": MyEnum.two})**assert** connection.scalar(t.select()) **is** MyEnum.two

Above, the string names of each element, e.g. "one", "two", "three", are persisted to the database; the values of the Python Enum, here indicated as integers, are ****not**** used; the value of each enum can therefore be any kind of Python object whether or not it is persistable.

*New in version 1.1:*- support for PEP-435-style enumerated classes.

**See also**

[ENUM](http://docs.sqlalchemy.org/en/rel_1_1/dialects/postgresql.html" \l "sqlalchemy.dialects.postgresql.ENUM" \o "sqlalchemy.dialects.postgresql.ENUM) - PostgreSQL-specific type, which has additional functionality.

**\_\_init\_\_**(*\*enums*, *\*\*kw*)

Construct an enum.

Keyword arguments which don't apply to a specific backend are ignored by that backend.

|  |  |
| --- | --- |
| **Parameters:** | * ****\*enums –****either exactly one PEP-435 compliant enumerated type or one or more string or unicode enumeration labels. If unicode labels are present, the convert\_unicode flag is auto-enabled.   *New in version 1.1:*a PEP-435 style enumerated class may be passed.   * ****convert\_unicode**** – Enable unicode-aware bind parameter and result-set processing for this Enum's data. This is set automatically based on the presence of unicode label strings. * ****create\_constraint –****   defaults to True. When creating a non-native enumerated type, also build a CHECK constraint on the database against the valid values.  *New in version 1.1:*- added [Enum.create\_constraint](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Enum.params.create_constraint" \o "sqlalchemy.types.Enum) which provides the option to disable the production of the CHECK constraint for a non-native enumerated type.   * ****metadata**** – Associate this type directly with a MetaData object. For types that exist on the target database as an independent schema construct (PostgreSQL), this type will be created and dropped within create\_all() and drop\_all() operations. If the type is not associated with any MetaData object, it will associate itself with each Table in which it is used, and will be created when any of those individual tables are created, after a check is performed for its existence. The type is only dropped when drop\_all() is called for that Table object's metadata, however. * ****name**** – The name of this type. This is required for PostgreSQL and any future supported database which requires an explicitly named type, or an explicitly named constraint in order to generate the type and/or a table that uses it. If a PEP-435 enumerated class was used, its name (converted to lower case) is used by default. * ****native\_enum**** – Use the database's native ENUM type when available. Defaults to True. When False, uses VARCHAR + check constraint for all backends. * ****schema –****Schema name of this type. For types that exist on the target database as an independent schema construct (PostgreSQL), this parameter specifies the named schema in which the type is present.   **Note**  The schema of the [Enum](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Enum" \o "sqlalchemy.types.Enum) type does not by default make use of the schema established on the owning [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table). If this behavior is desired, set the inherit\_schema flag to True.   * ****quote**** – Set explicit quoting preferences for the type's name. * ****inherit\_schema**** – When True, the "schema" from the owning [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) will be copied to the "schema" attribute of this [Enum](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Enum" \o "sqlalchemy.types.Enum), replacing whatever value was passed for the schema attribute. This also takes effect when using the [Table.tometadata()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.tometadata" \o "sqlalchemy.schema.Table.tometadata) operation. * ****validate\_strings –****when True, string values that are being passed to the database in a SQL statement will be checked for validity against the list of enumerated values. Unrecognized values will result in a LookupError being raised.   *New in version 1.1.0b2.* |

**create**(*bind=None*, *checkfirst=False*)

*inherited from the* [create()](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.SchemaType.create" \o "sqlalchemy.types.SchemaType.create) *method of* [SchemaType](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.SchemaType" \o "sqlalchemy.types.SchemaType)

Issue CREATE ddl for this type, if applicable.

**drop**(*bind=None*, *checkfirst=False*)

*inherited from the* [drop()](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.SchemaType.drop" \o "sqlalchemy.types.SchemaType.drop) *method of* [SchemaType](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.SchemaType" \o "sqlalchemy.types.SchemaType)

Issue DROP ddl for this type, if applicable.

*class*sqlalchemy.types.**Float**(*precision=None*, *asdecimal=False*, *decimal\_return\_scale=None*, *\*\*kwargs*)

Bases: [sqlalchemy.types.Numeric](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Numeric" \o "sqlalchemy.types.Numeric)

Type representing floating point types, such as FLOAT or REAL.

This type returns Python float objects by default, unless the [Float.asdecimal](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Float.params.asdecimal" \o "sqlalchemy.types.Float) flag is set to True, in which case they are coerced to decimal.Decimalobjects.

**Note**

The [Float](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Float" \o "sqlalchemy.types.Float) type is designed to receive data from a database type that is explicitly known to be a floating point type (e.g. FLOAT, REAL, others) and not a decimal type (e.g. DECIMAL, NUMERIC, others). If the database column on the server is in fact a Numeric type, such as DECIMAL or NUMERIC, use the [Numeric](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Numeric" \o "sqlalchemy.types.Numeric) type or a subclass, otherwise numeric coercion between float/Decimal may or may not function as expected.

**\_\_init\_\_**(*precision=None*, *asdecimal=False*, *decimal\_return\_scale=None*, *\*\*kwargs*)

Construct a Float.

|  |  |
| --- | --- |
| **Parameters:** | * ****precision**** – the numeric precision for use in DDL CREATE TABLE. * ****asdecimal**** – the same flag as that of [Numeric](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Numeric" \o "sqlalchemy.types.Numeric), but defaults to False. Note that setting this flag to True results in floating point conversion. * ****decimal\_return\_scale –****Default scale to use when converting from floats to Python decimals. Floating point values will typically be much longer due to decimal inaccuracy, and most floating point database types don't have a notion of "scale", so by default the float type looks for the first ten decimal places when converting. Specfiying this value will override that length. Note that the MySQL float types, which do include "scale", will use "scale" as the default for decimal\_return\_scale, if not otherwise specified.   *New in version 0.9.0.*   * ****\*\*kwargs**** – deprecated. Additional arguments here are ignored by the default [Float](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Float" \o "sqlalchemy.types.Float) type. For database specific floats that support additional arguments, see that dialect's documentation for details, such as [sqlalchemy.dialects.mysql.FLOAT](http://docs.sqlalchemy.org/en/rel_1_1/dialects/mysql.html" \l "sqlalchemy.dialects.mysql.FLOAT" \o "sqlalchemy.dialects.mysql.FLOAT). |

*class*sqlalchemy.types.**Integer**

Bases: sqlalchemy.types.\_DateAffinity, [sqlalchemy.types.TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine)

A type for int integers.

*class*sqlalchemy.types.**Interval**(*native=True*, *second\_precision=None*, *day\_precision=None*)

Bases: sqlalchemy.types.\_DateAffinity, [sqlalchemy.types.TypeDecorator](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator" \o "sqlalchemy.types.TypeDecorator)

A type for datetime.timedelta() objects.

The Interval type deals with datetime.timedelta objects. In PostgreSQL, the native INTERVAL type is used; for others, the value is stored as a date which is relative to the "epoch" (Jan. 1, 1970).

Note that the Interval type does not currently provide date arithmetic operations on platforms which do not support interval types natively. Such operations usually require transformation of both sides of the expression (such as, conversion of both sides into integer epoch values first) which currently is a manual procedure (such as via [func](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.func" \o "sqlalchemy.sql.expression.func)).

**\_\_init\_\_**(*native=True*, *second\_precision=None*, *day\_precision=None*)

Construct an Interval object.

|  |  |
| --- | --- |
| **Parameters:** | * ****native**** – when True, use the actual INTERVAL type provided by the database, if supported (currently PostgreSQL, Oracle). Otherwise, represent the interval data as an epoch value regardless. * ****second\_precision**** – For native interval types which support a "fractional seconds precision" parameter, i.e. Oracle and PostgreSQL * ****day\_precision**** – for native interval types which support a "day precision" parameter, i.e. Oracle. |

**coerce\_compared\_value**(*op*, *value*)

See [TypeEngine.coerce\_compared\_value()](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.coerce_compared_value" \o "sqlalchemy.types.TypeEngine.coerce_compared_value) for a description.

**impl**

alias of [DateTime](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.DateTime" \o "sqlalchemy.types.DateTime)

*class*sqlalchemy.types.**LargeBinary**(*length=None*)

Bases: sqlalchemy.types.\_Binary

A type for large binary byte data.

The [LargeBinary](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.LargeBinary" \o "sqlalchemy.types.LargeBinary) type corresponds to a large and/or unlengthed binary type for the target platform, such as BLOB on MySQL and BYTEA for PostgreSQL. It also handles the necessary conversions for the DBAPI.

**\_\_init\_\_**(*length=None*)

Construct a LargeBinary type.

|  |  |
| --- | --- |
| **Parameters:** | ****length**** – optional, a length for the column for use in DDL statements, for those binary types that accept a length, such as the MySQL BLOB type. |

*class*sqlalchemy.types.**MatchType**(*create\_constraint=True*, *name=None*, *\_create\_events=True*)

Bases: [sqlalchemy.types.Boolean](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Boolean" \o "sqlalchemy.types.Boolean)

Refers to the return type of the MATCH operator.

As the [ColumnOperators.match()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.match" \o "sqlalchemy.sql.operators.ColumnOperators.match) is probably the most open-ended operator in generic SQLAlchemy Core, we can't assume the return type at SQL evaluation time, as MySQL returns a floating point, not a boolean, and other backends might do something different. So this type acts as a placeholder, currently subclassing [Boolean](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Boolean" \o "sqlalchemy.types.Boolean). The type allows dialects to inject result-processing functionality if needed, and on MySQL will return floating-point values.

*New in version 1.0.0.*

*class*sqlalchemy.types.**Numeric**(*precision=None*, *scale=None*, *decimal\_return\_scale=None*, *asdecimal=True*)

Bases: sqlalchemy.types.\_DateAffinity, [sqlalchemy.types.TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine)

A type for fixed precision numbers, such as NUMERIC or DECIMAL.

This type returns Python decimal.Decimal objects by default, unless the [Numeric.asdecimal](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Numeric.params.asdecimal" \o "sqlalchemy.types.Numeric) flag is set to False, in which case they are coerced to Python float objects.

**Note**

The [Numeric](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Numeric" \o "sqlalchemy.types.Numeric) type is designed to receive data from a database type that is explicitly known to be a decimal type (e.g. DECIMAL, NUMERIC, others) and not a floating point type (e.g. FLOAT, REAL, others). If the database column on the server is in fact a floating-point type type, such as FLOAT or REAL, use the [Float](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Float" \o "sqlalchemy.types.Float) type or a subclass, otherwise numeric coercion between float/Decimal may or may not function as expected.

**Note**

The Python decimal.Decimal class is generally slow performing; cPython 3.3 has now switched to use the [cdecimal](http://pypi.python.org/pypi/cdecimal/) library natively. For older Python versions, the cdecimal library can be patched into any application where it will replace the decimal library fully, however this needs to be applied globally and before any other modules have been imported, as follows:

**import** **sys**

**import** **cdecimal**

sys.modules["decimal"] = cdecimal

Note that the cdecimal and decimal libraries are ****not compatible with each other****, so patching cdecimal at the global level is the only way it can be used effectively with various DBAPIs that hardcode to import the decimal library.

**\_\_init\_\_**(*precision=None*, *scale=None*, *decimal\_return\_scale=None*, *asdecimal=True*)

Construct a Numeric.

|  |  |
| --- | --- |
| **Parameters:** | * ****precision**** – the numeric precision for use in DDL CREATE TABLE. * ****scale**** – the numeric scale for use in DDL CREATE TABLE. * ****asdecimal**** – default True. Return whether or not values should be sent as Python Decimal objects, or as floats. Different DBAPIs send one or the other based on datatypes - the Numeric type will ensure that return values are one or the other across DBAPIs consistently. * ****decimal\_return\_scale –****Default scale to use when converting from floats to Python decimals. Floating point values will typically be much longer due to decimal inaccuracy, and most floating point database types don't have a notion of "scale", so by default the float type looks for the first ten decimal places when converting. Specfiying this value will override that length. Types which do include an explicit ".scale" value, such as the base [Numeric](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Numeric" \o "sqlalchemy.types.Numeric) as well as the MySQL float types, will use the value of ".scale" as the default for decimal\_return\_scale, if not otherwise specified.   *New in version 0.9.0.* |

When using the Numeric type, care should be taken to ensure that the asdecimal setting is apppropriate for the DBAPI in use - when Numeric applies a conversion from Decimal->float or float-> Decimal, this conversion incurs an additional performance overhead for all result columns received.

DBAPIs that return Decimal natively (e.g. psycopg2) will have better accuracy and higher performance with a setting of True, as the native translation to Decimal reduces the amount of floating- point issues at play, and the Numeric type itself doesn't need to apply any further conversions. However, another DBAPI which returns floats natively *will* incur an additional conversion overhead, and is still subject to floating point data loss - in which case asdecimal=False will at least remove the extra conversion overhead.

*class*sqlalchemy.types.**PickleType**(*protocol=2*, *pickler=None*, *comparator=None*)

Bases: [sqlalchemy.types.TypeDecorator](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator" \o "sqlalchemy.types.TypeDecorator)

Holds Python objects, which are serialized using pickle.

PickleType builds upon the Binary type to apply Python's pickle.dumps() to incoming objects, and pickle.loads() on the way out, allowing any pickleable Python object to be stored as a serialized binary field.

To allow ORM change events to propagate for elements associated with [PickleType](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.PickleType" \o "sqlalchemy.types.PickleType), see [Mutation Tracking](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html).

**\_\_init\_\_**(*protocol=2*, *pickler=None*, *comparator=None*)

Construct a PickleType.

|  |  |
| --- | --- |
| **Parameters:** | * ****protocol**** – defaults to pickle.HIGHEST\_PROTOCOL. * ****pickler**** – defaults to cPickle.pickle or pickle.pickle if cPickle is not available. May be any object with pickle-compatible dumps` and ``loadsmethods. * ****comparator**** – a 2-arg callable predicate used to compare values of this type. If left as None, the Python "equals" operator is used to compare values. |

**impl**

alias of [LargeBinary](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.LargeBinary" \o "sqlalchemy.types.LargeBinary)

*class*sqlalchemy.types.**SchemaType**(*name=None*, *schema=None*, *metadata=None*, *inherit\_schema=False*, *quote=None*, *\_create\_events=True*)

Bases: sqlalchemy.sql.expression.SchemaEventTarget

Mark a type as possibly requiring schema-level DDL for usage.

将某个类型标记为可能需要使用架构级别的DDL。

Supports types that must be explicitly created/dropped (i.e. PG ENUM type) as well as types that are complimented by table or schema level constraints, triggers, and other rules.

支持必须显式创建/删除的类型（即PG ENUM类型）以及由表或模式级别约束，触发器和其他规则所赞扬的类型。

[SchemaType](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.SchemaType" \o "sqlalchemy.types.SchemaType) classes can also be targets for the [DDLEvents.before\_parent\_attach()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.DDLEvents.before_parent_attach" \o "sqlalchemy.events.DDLEvents.before_parent_attach) and [DDLEvents.after\_parent\_attach()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.DDLEvents.after_parent_attach" \o "sqlalchemy.events.DDLEvents.after_parent_attach) events, where the events fire off surrounding the association of the type object with a parent [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column).

SchemaType类也可以是DDLEvents.before\_parent\_attach（）和DDLEvents.after\_parent\_attach（）事件的目标，其中事件触发类型对象与父列的关联。

**See also**

[Enum](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Enum" \o "sqlalchemy.types.Enum)

[Boolean](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Boolean" \o "sqlalchemy.types.Boolean)

**adapt**(*impltype*, *\*\*kw*)

**bind**

**copy**(*\*\*kw*)

**create**(*bind=None*, *checkfirst=False*)

Issue CREATE ddl for this type, if applicable.

**drop**(*bind=None*, *checkfirst=False*)

Issue DROP ddl for this type, if applicable.

*class*sqlalchemy.types.**SmallInteger**

Bases: [sqlalchemy.types.Integer](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Integer" \o "sqlalchemy.types.Integer)

A type for smaller int integers.

Typically generates a SMALLINT in DDL, and otherwise acts like a normal [Integer](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Integer" \o "sqlalchemy.types.Integer) on the Python side.

*class*sqlalchemy.types.**String**(*length=None*, *collation=None*, *convert\_unicode=False*, *unicode\_error=None*, *\_warn\_on\_bytestring=False*)

Bases: [sqlalchemy.types.Concatenable](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.Concatenable" \o "sqlalchemy.types.Concatenable), [sqlalchemy.types.TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine)

The base for all string and character types.

所有字符串和字符类型的基础。

In SQL, corresponds to VARCHAR. Can also take Python unicode objects and encode to the database's encoding in bind params (and the reverse for result sets.)

在SQL中，对应于VARCHAR。 也可以采用Python unicode对象，并在绑定参数中对数据库的编码进行编码（结果集反过来）。

The length field is usually required when the String type is used within a CREATE TABLE statement, as VARCHAR requires a length on most databases.

当在CREATE TABLE语句中使用字符串类型时，通常需要长度字段，因为VARCHAR在大多数数据库上需要长度。

**\_\_init\_\_**(*length=None*, *collation=None*, *convert\_unicode=False*, *unicode\_error=None*, *\_warn\_on\_bytestring=False*)

Create a string-holding type.

|  |  |
| --- | --- |
| **Parameters:** | ****length**** – optional, a length for the column for use in DDL and CAST expressions. May be safely omitted if no CREATE TABLE will be issued. Certain databases may require a length for use in DDL, and will raise an exception when the CREATE TABLE DDL is issued if a VARCHAR with no length is included. Whether the value is interpreted as bytes or characters is database specific.可选，用于DDL和CAST表达式的列的长度。 如果不发出CREATE TABLE，可以安全省略。 某些数据库可能需要在DDL中使用的长度，并且如果包含无长度的VARCHAR，则在发出CREATE TABLE DDL时引发异常。 值是否被解释为字节或字符是数据库特定的。  ****collation**** –Optional, a column-level collation for use in DDL and CAST expressions. Renders using the COLLATE keyword supported by SQLite, MySQL, and PostgreSQL. E.g.:可选的是用于DDL和CAST表达式的列级排序规则。 渲染使用SQLite，MySQL和PostgreSQL支持的COLLATE关键字。 例如。：  **>>> from** **sqlalchemy** **import** cast, select, String  **>>>** print select([cast('some string', String(collation='utf8'))])SELECT CAST(:param\_1 AS VARCHAR COLLATE utf8) AS anon\_1  *New in version 0.8:*Added support for COLLATE to all string types.  ****convert\_unicode –****When set to True, the [String](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.String" \o "sqlalchemy.types.String) type will assume that input is to be passed as Python unicode objects, and results returned as Python unicodeobjects. If the DBAPI in use does not support Python unicode (which is fewer and fewer these days), SQLAlchemy will encode/decode the value, using the value of the encoding parameter passed to [create\_engine()](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "sqlalchemy.create_engine" \o "sqlalchemy.create_engine) as the encoding.当设置为True时，String类型将假定输入将作为Python unicode对象传递，结果返回为Python unicodeobjects。 如果正在使用的DBAPI不支持Python unicode（这些日子越来越少），SQLAlchemy将使用传递给create\_engine() 作为编码的encoding参数的值对该值进行编码/解码。  When using a DBAPI that natively supports Python unicode objects, this flag generally does not need to be set. For columns that are explicitly intended to store non-ASCII data, the [Unicode](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Unicode" \o "sqlalchemy.types.Unicode) or [UnicodeText](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.UnicodeText" \o "sqlalchemy.types.UnicodeText) types should be used regardless, which feature the same behavior of convert\_unicodebut also indicate an underlying column type that directly supports unicode, such as NVARCHAR.当使用本机支持Python unicode对象的DBAPI时，通常不需要设置该标志。 对于明确打算存储非ASCII数据的列，应使用Unicode或UnicodeText类型，而不管其特性与convert\_unicode相同，还表示直接支持unicode的底层列类型，例如NVARCHAR。  For the extremely rare case that Python unicode is to be encoded/decoded by SQLAlchemy on a backend that does natively support Python unicode, the value force can be passed here which will cause SQLAlchemy's encode/decode services to be used unconditionally.对于非常罕见的情况，Python unicode将通过SQLAlchemy在本地支持Python unicode的后端进行编码/解码，因此可以传递值强制，这将导致SQLAlchemy的无条件使用的编码/解码服务。  ****unicode\_error**** – Optional, a method to use to handle Unicode conversion errors. Behaves like the errors keyword argument to the standard library's string.decode() functions. This flag requires that convert\_unicode is set to force - otherwise, SQLAlchemy is not guaranteed to handle the task of unicode conversion. Note that this flag adds significant performance overhead to row-fetching operations for backends that already return unicode objects natively (which most DBAPIs do). This flag should only be used as a last resort for reading strings from a column with varied or corrupted encodings.可选，用于处理Unicode转换错误的方法。 像标准库的string.decode() 函数的errors关键字参数一样。 该标志要求convert\_unicode设置为强制 - 否则，SQLAlchemy不能保证处理unicode转换的任务。 请注意，此标志为已经返回unicode对象（大多数DBAPI执行）的后端的行提取操作增加了显着的性能开销。 该标志只能用作从列中读取字符串的最后手段，该列具有不同或损坏的编码。 |

*class*sqlalchemy.types.**Text**(*length=None*, *collation=None*, *convert\_unicode=False*, *unicode\_error=None*, *\_warn\_on\_bytestring=False*)

Bases: [sqlalchemy.types.String](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.String" \o "sqlalchemy.types.String)

A variably sized string type.

一个可变大小的字符串类型。

In SQL, usually corresponds to CLOB or TEXT. Can also take Python unicode objects and encode to the database's encoding in bind params (and the reverse for result sets.) In general, TEXT objects do not have a length; while some databases will accept a length argument here, it will be rejected by others.

在SQL中，通常对应于CLOB或TEXT。 也可以使用Python unicode对象，并将其编码为绑定参数中的数据库编码（反之亦然）。一般而言，TEXT对象没有长度; 而一些数据库将在这里接受一个长度的参数，它将被其他人拒绝。

*class*sqlalchemy.types.**Time**(*timezone=False*)

Bases: sqlalchemy.types.\_DateAffinity, [sqlalchemy.types.TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine)

A type for datetime.time() objects.

*class*sqlalchemy.types.**Unicode**(*length=None*, *\*\*kwargs*)

Bases: [sqlalchemy.types.String](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.String" \o "sqlalchemy.types.String)

A variable length Unicode string type.

可变长度的Unicode字符串类型。

The [Unicode](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Unicode" \o "sqlalchemy.types.Unicode) type is a [String](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.String" \o "sqlalchemy.types.String) subclass that assumes input and output as Python unicode data, and in that regard is equivalent to the usage of theconvert\_unicode flag with the [String](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.String" \o "sqlalchemy.types.String) type. However, unlike plain [String](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.String" \o "sqlalchemy.types.String), it also implies an underlying column type that is explicitly supporting of non-ASCII data, such as NVARCHAR on Oracle and SQL Server. This can impact the output of CREATE TABLE statements and CAST functions at the dialect level, and can also affect the handling of bound parameters in some specific DBAPI scenarios.

Unicode类型是一个String子类，它将输入和输出视为Python unicode数据，在这方面等同于使用String类型的convert\_unicode标志。 但是，与plain String不同，它还意味着明确支持非ASCII数据的基础列类型，例如Oracle和SQL Server上的NVARCHAR。 这可能会影响方言级别的CREATE TABLE语句和CAST函数的输出，并且还可能影响某些特定DBAPI方案中绑定参数的处理。

The encoding used by the [Unicode](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Unicode" \o "sqlalchemy.types.Unicode) type is usually determined by the DBAPI itself; most modern DBAPIs feature support for Python unicode objects as bound values and result set values, and the encoding should be configured as detailed in the notes for the target DBAPI in the [Dialects](http://docs.sqlalchemy.org/en/rel_1_1/dialects/index.html) section.

Unicode类型使用的编码通常由DBAPI本身决定; 大多数现代DBAPI都支持Python unicode对象作为绑定值和结果集值，并且编码应该在方言部分的目标DBAPI的注释中进行配置。

For those DBAPIs which do not support, or are not configured to accommodate Python unicode objects directly, SQLAlchemy does the encoding and decoding outside of the DBAPI. The encoding in this scenario is determined by the encoding flag passed to [create\_engine()](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "sqlalchemy.create_engine" \o "sqlalchemy.create_engine).

对于那些不支持或未配置为直接适应Python unicode对象的DBAPI，SQLAlchemy在DBAPI外进行编码和解码。 在这种情况下的编码由传递给create\_engine() 的编码标志确定。

When using the [Unicode](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Unicode" \o "sqlalchemy.types.Unicode) type, it is only appropriate to pass Python unicode objects, and not plain str. If a plain str is passed under Python 2, a warning is emitted. If you notice your application emitting these warnings but you're not sure of the source of them, the Python warnings filter, documented at<http://docs.python.org/library/warnings.html>, can be used to turn these warnings into exceptions which will illustrate a stack trace:

当使用Unicode类型时，只适用于传递Python unicode对象，而不是plain str。 如果在Python 2下传递一个简单的str，则会发出警告。 如果您注意到您的应用程序发出这些警告，但是您不确定它们的来源，则可以使用Python警告过滤器（文档：http：//docs.python.org/library/warnings.html）将这些警告转换为 将说明堆栈跟踪的异常：

**import** **warnings**

warnings.simplefilter('error')

For an application that wishes to pass plain bytestrings and Python unicode objects to the Unicode type equally, the bytestrings must first be decoded into unicode. The recipe at [Coercing Encoded Strings to Unicode](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "coerce-to-unicode) illustrates how this is done.

对于希望将普通的测试和Python unicode对象平等地传递给Unicode类型的应用程序，必须首先将字符串解码为unicode。 加密编码字符串到Unicode的配方说明了如何完成。

See also:

[UnicodeText](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.UnicodeText" \o "sqlalchemy.types.UnicodeText) - unlengthed textual counterpart to [Unicode](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Unicode" \o "sqlalchemy.types.Unicode).

**\_\_init\_\_**(*length=None*, *\*\*kwargs*)

Create a [Unicode](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Unicode" \o "sqlalchemy.types.Unicode) object.

创建一个Unicode对象。

Parameters are the same as that of [String](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.String" \o "sqlalchemy.types.String), with the exception that convert\_unicode defaults to True.

参数与String相同，但convert\_unicode默认为True。

*class*sqlalchemy.types.**UnicodeText**(*length=None*, *\*\*kwargs*)

Bases: [sqlalchemy.types.Text](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Text" \o "sqlalchemy.types.Text)

An unbounded-length Unicode string type.

无限长Unicode字符串类型。

See [Unicode](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Unicode" \o "sqlalchemy.types.Unicode) for details on the unicode behavior of this object.

有关此对象的unicode行为的详细信息，请参阅Unicode。

Like [Unicode](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Unicode" \o "sqlalchemy.types.Unicode), usage the [UnicodeText](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.UnicodeText" \o "sqlalchemy.types.UnicodeText) type implies a unicode-capable type being used on the backend, such as NCLOB, NTEXT.

像Unicode一样，使用Unicode文本类型意味着在后端使用unicode功能的类型，如CLOB，NTEXT。

**\_\_init\_\_**(*length=None*, *\*\*kwargs*)

Create a Unicode-converting Text type.

创建Unicode转换文本类型。

Parameters are the same as that of [Text](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Text" \o "sqlalchemy.types.Text), with the exception that convert\_unicode defaults to True.

参数与文本相同，但convert\_unicode默认为True。

4.1.2 SQL Standard and Multiple Vendor Types

This category of types refers to types that are either part of the SQL standard, or are potentially found within a subset of database backends. Unlike the "generic" types, the SQL standard/multi-vendor types have ****no**** guarantee of working on all backends, and will only work on those backends that explicitly support them by name. That is, the type will always emit its exact name in DDL with CREATE TABLE is issued.

这种类型的类型是指作为SQL标准的一部分的类型，或者可能在数据库后端的子集中找到。 与“通用”类型不同，SQL标准/多供应商类型不保证在所有后端上工作，并且只能在名称上显式支持它们的后端上工作。 也就是说，类型将始终使用CREATE TABLE发出DDL中的确切名称。

*class*sqlalchemy.types.**ARRAY**(*item\_type*, *as\_tuple=False*, *dimensions=None*, *zero\_indexes=False*)

Bases: [sqlalchemy.types.Indexable](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.Indexable" \o "sqlalchemy.types.Indexable), [sqlalchemy.types.Concatenable](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.Concatenable" \o "sqlalchemy.types.Concatenable), [sqlalchemy.types.TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine)

Represent a SQL Array type.

**Note**

This type serves as the basis for all ARRAY operations. However, currently ****only the PostgreSQL backend has support for SQL arrays in SQLAlchemy****. It is recommended to use the [postgresql.ARRAY](http://docs.sqlalchemy.org/en/rel_1_1/dialects/postgresql.html" \l "sqlalchemy.dialects.postgresql.ARRAY" \o "sqlalchemy.dialects.postgresql.ARRAY) type directly when using ARRAY types with PostgreSQL, as it provides additional operators specific to that backend.

这种类型作为所有ARRAY操作的基础。 但是，目前只有PostgreSQL后端在SQLAlchemy中支持SQL数组。 建议在PostgreSQL使用ARRAY类型时直接使用postgresql.ARRAY类型，因为它提供了特定于后端的其他操作符。

[types.ARRAY](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.ARRAY" \o "sqlalchemy.types.ARRAY) is part of the Core in support of various SQL standard functions such as [array\_agg](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.array_agg" \o "sqlalchemy.sql.functions.array_agg) which explicitly involve arrays; however, with the exception of the PostgreSQL backend and possibly some third-party dialects, no other SQLAlchemy built-in dialect has support for this type.

types.ARRAY是Core的一部分，支持各种SQL标准功能，如array\_agg，其明确涉及数组; 然而，除PostgreSQL后端和可能的一些第三方方言外，没有其他SQLAlchemy内置方言支持此类型。

An [types.ARRAY](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.ARRAY" \o "sqlalchemy.types.ARRAY) type is constructed given the "type" of element:

type.ARRAY类型是由元素的“type”构成的：

mytable = Table("mytable", metadata,

Column("data", ARRAY(Integer))

)

The above type represents an N-dimensional array, meaning a supporting backend such as PostgreSQL will interpret values with any number of dimensions automatically. To produce an INSERT construct that passes in a 1-dimensional array of integers:

上述类型表示N维数组，意味着PostgreSQL等支持后端将自动解释具有任意数量维度的值。 要产生一个传递一维整数数组的INSERT结构：

connection.execute(

mytable.insert(),

data=[1,2,3])

The [types.ARRAY](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.ARRAY" \o "sqlalchemy.types.ARRAY) type can be constructed given a fixed number of dimensions:

可以使用固定数量的维度来构造types.ARRAY类型：

mytable = Table("mytable", metadata,

Column("data", ARRAY(Integer, dimensions=2))

)

Sending a number of dimensions is optional, but recommended if the datatype is to represent arrays of more than one dimension. This number is used:

发送多个维度是可选的，但如果数据类型表示多个维度的数组，则建议使用。 使用这个数字：

When emitting the type declaration itself to the database, e.g. INTEGER[][]

当将类型声明本身发布到数据库时，例如， 整数[][]

When translating Python values to database values, and vice versa, e.g. an ARRAY of [Unicode](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Unicode" \o "sqlalchemy.types.Unicode) objects uses this number to efficiently access the string values inside of array structures without resorting to per-row type inspection

将Python值转换为数据库值时，反之亦然。 Unicode对象的ARRAY使用此数字可以有效地访问数组结构中的字符串值，而无需依赖于每行类型检查

When used with the Python getitem accessor, the number of dimensions serves to define the kind of type that the [] operator should return, e.g. for an ARRAY of INTEGER with two dimensions:

当与Python getitem访问器一起使用时，维数用于定义[]运算符应该返回的类型的类型，例如 对于具有二维的INTEGER阵列：

**>>>** expr = table.c.column[5] *# returns ARRAY(Integer, dimensions=1)*

**>>>** expr = expr[6] *# returns Integer*

For 1-dimensional arrays, an [types.ARRAY](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.ARRAY" \o "sqlalchemy.types.ARRAY) instance with no dimension parameter will generally assume single-dimensional behaviors.

对于1维数组，没有维度参数的types.ARRAY实例通常将采用单维行为。

SQL expressions of type [types.ARRAY](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.ARRAY" \o "sqlalchemy.types.ARRAY) have support for "index" and "slice" behavior. The Python [] operator works normally here, given integer indexes or slices. Arrays default to 1-based indexing. The operator produces binary expression constructs which will produce the appropriate SQL, both for SELECT statements:

类型类型的SQL表达式.ARRAY支持“索引”和“切片”行为。 Python []运算符在这里正常工作，给定整数索引或片。 数组默认为基于1的索引。 运算符生成二进制表达式构造，它将为SELECT语句生成适当的SQL：

select([mytable.c.data[5], mytable.c.data[2:7]])

as well as UPDATE statements when the [Update.values()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Update.values" \o "sqlalchemy.sql.expression.Update.values) method is used:

以及使用Update.values() 方法时的UPDATE语句：

mytable.update().values({

mytable.c.data[5]: 7,

mytable.c.data[2:7]: [1, 2, 3]})

The [types.ARRAY](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.ARRAY" \o "sqlalchemy.types.ARRAY) type also provides for the operators [types.ARRAY.Comparator.any()](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.ARRAY.Comparator.any" \o "sqlalchemy.types.ARRAY.Comparator.any) and [types.ARRAY.Comparator.all()](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.ARRAY.Comparator.all" \o "sqlalchemy.types.ARRAY.Comparator.all). The PostgreSQL-specific version of [types.ARRAY](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.ARRAY" \o "sqlalchemy.types.ARRAY) also provides additional operators.

types.ARRAY类型还为运算符类型.ARRAY.Comparator.any() 和types.ARRAY.Comparator.all() 提供。 PostgreSQL特定版本的类型.ARRAY还提供了额外的运算符。

*New in version 1.1.0.*

**See also**

[postgresql.ARRAY](http://docs.sqlalchemy.org/en/rel_1_1/dialects/postgresql.html" \l "sqlalchemy.dialects.postgresql.ARRAY" \o "sqlalchemy.dialects.postgresql.ARRAY)

*class***Comparator**(*expr*)

Bases: sqlalchemy.types.Comparator, sqlalchemy.types.Comparator

Define comparison operations for [types.ARRAY](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.ARRAY" \o "sqlalchemy.types.ARRAY).

定义types.ARRAY的比较操作。

More operators are available on the dialect-specific form of this type. See [postgresql.ARRAY.Comparator](http://docs.sqlalchemy.org/en/rel_1_1/dialects/postgresql.html" \l "sqlalchemy.dialects.postgresql.ARRAY.Comparator" \o "sqlalchemy.dialects.postgresql.ARRAY.Comparator).

更多的运营商可以使用这种类型的方言特定形式。见postgresql.ARRAY.Comparator。

**all**(*other*, *operator=None*)

Return other operator ALL (array) clause.

Argument places are switched, because ALL requires array expression to be on the right hand-side.

参数位置被切换，因为ALL需要阵列表达式位于右侧。

E.g.:

**from** **sqlalchemy.sql** **import** operators

conn.execute(

select([table.c.data]).where(

table.c.data.all(7, operator=operators.lt)

))

|  |  |
| --- | --- |
| **Parameters:** | * ****other**** – expression to be compared * ****operator**** – an operator object from the sqlalchemy.sql.operators package, defaults to operators.eq(). |

**See also**

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

[types.ARRAY.Comparator.any()](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.ARRAY.Comparator.any" \o "sqlalchemy.types.ARRAY.Comparator.any)

**any**(*other*, *operator=None*)

Return other operator ANY (array) clause.

返回其他操作者ANY（阵列）子句。

Argument places are switched, because ANY requires array expression to be on the right hand-side.

参数位置被切换，因为ANY需要阵列表达式位于右侧。

E.g.:

**from** **sqlalchemy.sql** **import** operators

conn.execute(

select([table.c.data]).where(

table.c.data.any(7, operator=operators.lt)

))

|  |  |
| --- | --- |
| **Parameters:** | * ****other**** – expression to be compared * ****operator**** – an operator object from the sqlalchemy.sql.operators package, defaults to operators.eq(). |

**See also**

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

[types.ARRAY.Comparator.all()](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.ARRAY.Comparator.all" \o "sqlalchemy.types.ARRAY.Comparator.all)

**\_\_init\_\_**(*item\_type*, *as\_tuple=False*, *dimensions=None*, *zero\_indexes=False*)

Construct an [types.ARRAY](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.ARRAY" \o "sqlalchemy.types.ARRAY).

E.g.:

Column('myarray', ARRAY(Integer))

Arguments are:

|  |  |
| --- | --- |
| **Parameters:** | * ****item\_type**** – The data type of items of this array. Note that dimensionality is irrelevant here, so multi-dimensional arrays like INTEGER[][], are constructed as ARRAY(Integer), not as ARRAY(ARRAY(Integer)) or such.该数组的数据类型。 请注意，维度在这里是无关紧要的，因此像INTEGER [] []这样的多维数组被构造为ARRAY（整数），而不是ARRAY（ARRAY（整数））。 * ****as\_tuple=False**** – Specify whether return results should be converted to tuples from lists. This parameter is not generally needed as a Python list corresponds well to a SQL array.指定返回结果是否应该从列表转换为元组。 通常不需要这个参数，因为Python列表很好地对应于SQL数组。 * ****dimensions**** – if non-None, the ARRAY will assume a fixed number of dimensions. This impacts how the array is declared on the database, how it goes about interpreting Python and result values, as well as how expression behavior in conjunction with the "getitem" operator works. See the description at[types.ARRAY](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.ARRAY" \o "sqlalchemy.types.ARRAY) for additional detail.如果不是，则ARRAY将承担固定数量的维度。 这会影响数组中声明数组的方式，如何解释Python和结果值，以及如何与“getitem”操作符一起使用表达式行为。 有关详细信息，请参阅attypes.ARRAY的说明。 * ****zero\_indexes=False**** – when True, index values will be converted between Python zero-based and SQL one-based indexes, e.g. a value of one will be added to all index values before passing to the database.当为True时，索引值将在基于Python的基于零和基于SQL的索引之间进行转换，例如。 在传递到数据库之前，将为所有索引值添加一个值。 |

**comparator\_factory**

alias of [Comparator](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.ARRAY.Comparator" \o "sqlalchemy.types.ARRAY.Comparator)

**zero\_indexes***= False*

if True, Python zero-based indexes should be interpreted as one-based on the SQL expression side.

*class*sqlalchemy.types.**BIGINT**

Bases: [sqlalchemy.types.BigInteger](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.BigInteger" \o "sqlalchemy.types.BigInteger)

The SQL BIGINT type.

*class*sqlalchemy.types.**BINARY**(*length=None*)

Bases: sqlalchemy.types.\_Binary

The SQL BINARY type.

*class*sqlalchemy.types.**BLOB**(*length=None*)

Bases: [sqlalchemy.types.LargeBinary](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.LargeBinary" \o "sqlalchemy.types.LargeBinary)

The SQL BLOB type.

*class*sqlalchemy.types.**BOOLEAN**(*create\_constraint=True*, *name=None*, *\_create\_events=True*)

Bases: [sqlalchemy.types.Boolean](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Boolean" \o "sqlalchemy.types.Boolean)

The SQL BOOLEAN type.

*class*sqlalchemy.types.**CHAR**(*length=None*, *collation=None*, *convert\_unicode=False*, *unicode\_error=None*, *\_warn\_on\_bytestring=False*)

Bases: [sqlalchemy.types.String](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.String" \o "sqlalchemy.types.String)

The SQL CHAR type.

*class*sqlalchemy.types.**CLOB**(*length=None*, *collation=None*, *convert\_unicode=False*, *unicode\_error=None*, *\_warn\_on\_bytestring=False*)

Bases: [sqlalchemy.types.Text](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Text" \o "sqlalchemy.types.Text)

The CLOB type.

This type is found in Oracle and Informix.

*class*sqlalchemy.types.**DATE**

Bases: [sqlalchemy.types.Date](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Date" \o "sqlalchemy.types.Date)

The SQL DATE type.

*class*sqlalchemy.types.**DATETIME**(*timezone=False*)

Bases: [sqlalchemy.types.DateTime](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.DateTime" \o "sqlalchemy.types.DateTime)

The SQL DATETIME type.

*class*sqlalchemy.types.**DECIMAL**(*precision=None*, *scale=None*, *decimal\_return\_scale=None*, *asdecimal=True*)

Bases: [sqlalchemy.types.Numeric](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Numeric" \o "sqlalchemy.types.Numeric)

The SQL DECIMAL type.

*class*sqlalchemy.types.**FLOAT**(*precision=None*, *asdecimal=False*, *decimal\_return\_scale=None*, *\*\*kwargs*)

Bases: [sqlalchemy.types.Float](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Float" \o "sqlalchemy.types.Float)

The SQL FLOAT type.

sqlalchemy.types.**INT**

alias of [INTEGER](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.INTEGER" \o "sqlalchemy.types.INTEGER)

*class*sqlalchemy.types.**JSON**(*none\_as\_null=False*)

Bases: [sqlalchemy.types.Indexable](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.Indexable" \o "sqlalchemy.types.Indexable), [sqlalchemy.types.TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine)

Represent a SQL JSON type.

**Note**

[types.JSON](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.JSON" \o "sqlalchemy.types.JSON) is provided as a facade for vendor-specific JSON types. Since it supports JSON SQL operations, it only works on backends that have an actual JSON type, currently PostgreSQL as well as certain versions of MySQL.

types.JSON作为供应商特定的JSON类型的外观提供。 由于它支持JSON SQL操作，所以它仅适用于具有实际JSON类型的后端，目前为PostgreSQL以及某些版本的MySQL。

[types.JSON](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.JSON" \o "sqlalchemy.types.JSON) is part of the Core in support of the growing popularity of native JSON datatypes.

types.JSON是Core的一部分，用于支持本机JSON数据类型日益普及。

The [types.JSON](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.JSON" \o "sqlalchemy.types.JSON) type stores arbitrary JSON format data, e.g.:

类型.JSON类型存储任意的JSON格式数据，例如：

data\_table = Table('data\_table', metadata,

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

Column('data', JSON))

**with** engine.connect() **as** conn:

conn.execute(

data\_table.insert(),

data = {"key1": "value1", "key2": "value2"}

)

The base [types.JSON](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.JSON" \o "sqlalchemy.types.JSON) provides these two operations:

Keyed index operations:

data\_table.c.data['some key']

Integer index operations:

data\_table.c.data[3]

Path index operations:

data\_table.c.data[('key\_1', 'key\_2', 5, ..., 'key\_n')]

Additional operations are available from the dialect-specific versions of [types.JSON](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.JSON" \o "sqlalchemy.types.JSON), such as [postgresql.JSON](http://docs.sqlalchemy.org/en/rel_1_1/dialects/postgresql.html" \l "sqlalchemy.dialects.postgresql.JSON" \o "sqlalchemy.dialects.postgresql.JSON) and [postgresql.JSONB](http://docs.sqlalchemy.org/en/rel_1_1/dialects/postgresql.html" \l "sqlalchemy.dialects.postgresql.JSONB" \o "sqlalchemy.dialects.postgresql.JSONB), each of which offer more operators than just the basic type.

可以从方言特定版本的类型JSON获得附加操作，例如postgresql.JSON和postgresql.JSONB，每个操作符都提供比仅基本类型更多的操作符。

Index operations return an expression object whose type defaults to [JSON](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.JSON" \o "sqlalchemy.types.JSON) by default, so that further JSON-oriented instructions may be called upon the result type. Note that there are backend-specific idiosyncracies here, including that the Postgresql database does not generally compare a "json" to a "json" structure without type casts. These idiosyncracies can be accommodated in a backend-neutral way by by making explicit use of the [cast()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.cast" \o "sqlalchemy.sql.expression.cast) and [type\_coerce()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.type_coerce" \o "sqlalchemy.sql.expression.type_coerce)constructs. Comparison of specific index elements of a [JSON](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.JSON" \o "sqlalchemy.types.JSON) object to other objects work best if the ****left hand side is CAST to a string**** and the ****right hand side is rendered as a json string****; a future SQLAlchemy feature such as a generic "astext" modifier may simplify this at some point:

默认情况下，索引操作返回一个类型默认为JSON的表达式对象，因此可能会在结果类型上调用进一步的面向JSON的指令。 请注意，这里有后端特定的特性，包括Postgresql数据库通常不会将“json”与“json”结构进行比较，而不使用类型转换。 通过明确使用cast() 和type\_coerce() 结构，这些特性可以通过后端中立的方式来适应。 将JSON对象与其他对象的特定索引元素进行比较，如果左侧是CAST到一个字符串，右侧被渲染为一个json字符串; 未来的SQLAlchemy功能（如通用“astext”）修饰符可能会在某些时候简化：

****Compare an element of a JSON structure to a string****:

**from** **sqlalchemy** **import** cast, type\_coerce**from** **sqlalchemy** **import** String, JSON

cast(

data\_table.c.data['some\_key'], String) == '"some\_value"'

cast(

data\_table.c.data['some\_key'], String) == type\_coerce("some\_value", JSON)

****Compare an element of a JSON structure to an integer****:

将JSON结构的元素与整数进行比较：

**from** **sqlalchemy** **import** cast, type\_coerce

**from** **sqlalchemy** **import** String, JSON

cast(data\_table.c.data['some\_key'], String) == '55'

cast(

data\_table.c.data['some\_key'], String) == type\_coerce(55, JSON)

****Compare an element of a JSON structure to some other JSON structure**** - note that Python dictionaries are typically not ordered so care should be taken here to assert that the JSON structures are identical:

将JSON结构的元素与其他JSON结构进行比较 - 请注意，Python字典通常不会被排序，因此应该谨慎地断言JSON结构是相同的：

**from** **sqlalchemy** **import** cast, type\_coerce

**from** **sqlalchemy** **import** String, JSON

**import** **json**

cast(

data\_table.c.data['some\_key'], String) == json.dumps({"foo": "bar"})

cast(

data\_table.c.data['some\_key'], String) == type\_coerce({"foo": "bar"}, JSON)

The [JSON](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.JSON" \o "sqlalchemy.types.JSON) type, when used with the SQLAlchemy ORM, does not detect in-place mutations to the structure. In order to detect these, 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 must be used. This extension will allow "in-place" changes to the datastructure to produce events which will be detected by the unit of work. See the example at [HSTORE](http://docs.sqlalchemy.org/en/rel_1_1/dialects/postgresql.html" \l "sqlalchemy.dialects.postgresql.HSTORE" \o "sqlalchemy.dialects.postgresql.HSTORE) for a simple example involving a dictionary.

当与SQLAlchemy ORM一起使用时，JSON类型不会检测到结构的就地突变。 为了检测这些，必须使用thesalalchemy.ext.mutable扩展名。 此扩展将允许数据结构的“就地”更改以产生将由工作单元检测到的事件。 请参阅HSTORE中的示例，以获得涉及字典的简单示例。

When working with NULL values, the [JSON](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.JSON" \o "sqlalchemy.types.JSON) type recommends the use of two specific constants in order to differentiate between a column that evaluates to SQL NULL, e.g. no value, vs. the JSON-encoded string of "null". To insert or select against a value that is SQL NULL, use the constant [null()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.null" \o "sqlalchemy.sql.expression.null):

当使用NULL值时，JSON类型建议使用两个特定的常量，以区分评估为SQL NULL的列，例如。 没有值，而JSON编码的字符串为“null”。 要插入或选择一个值为SQL NULL，请使用常量null() ：

**from** **sqlalchemy** **import** null

conn.execute(table.insert(), json\_value=null())

To insert or select against a value that is JSON "null", use the constant [JSON.NULL](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.JSON.NULL" \o "sqlalchemy.types.JSON.NULL):

要插入或选择一个值为JSON“null”，请使用常量JSON.NULL：

conn.execute(table.insert(), json\_value=JSON.NULL)

The [JSON](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.JSON" \o "sqlalchemy.types.JSON) type supports a flag [JSON.none\_as\_null](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.JSON.params.none_as_null" \o "sqlalchemy.types.JSON) which when set to True will result in the Python constant None evaluating to the value of SQL NULL, and when set to False results in the Python constant None evaluating to the value of JSON "null". The Python value None may be used in conjunction with either[JSON.NULL](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.JSON.NULL" \o "sqlalchemy.types.JSON.NULL) and [null()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.null" \o "sqlalchemy.sql.expression.null) in order to indicate NULL values, but care must be taken as to the value of the [JSON.none\_as\_null](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.JSON.params.none_as_null" \o "sqlalchemy.types.JSON) in these cases.

JSON类型支持一个标志JSON.none\_as\_null，当设置为True将导致Python常量无评估为SQL NULL的值，并且当设置为False时，会导致Python常量无法评估JSON“null”的值。 为了指示NULL值，Python值None可以与JSON.NULL和null() 结合使用，但在这些情况下，必须注意JSON.none\_as\_null的值。

**See also**

[postgresql.JSON](http://docs.sqlalchemy.org/en/rel_1_1/dialects/postgresql.html" \l "sqlalchemy.dialects.postgresql.JSON" \o "sqlalchemy.dialects.postgresql.JSON)

[postgresql.JSONB](http://docs.sqlalchemy.org/en/rel_1_1/dialects/postgresql.html" \l "sqlalchemy.dialects.postgresql.JSONB" \o "sqlalchemy.dialects.postgresql.JSONB)

[mysql.JSON](http://docs.sqlalchemy.org/en/rel_1_1/dialects/mysql.html" \l "sqlalchemy.dialects.mysql.JSON" \o "sqlalchemy.dialects.mysql.JSON)

*New in version 1.1.*

*class***Comparator**(*expr*)

Bases: sqlalchemy.types.Comparator, sqlalchemy.types.Comparator

Define comparison operations for [types.JSON](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.JSON" \o "sqlalchemy.types.JSON).

*class***JSONElementType**

Bases: [sqlalchemy.types.TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine)

common function for index / path elements in a JSON expression.

*class***JSONIndexType**

Bases: sqlalchemy.types.JSONElementType

Placeholder for the datatype of a JSON index value.

This allows execution-time processing of JSON index values for special syntaxes.

*class***JSONPathType**

Bases: sqlalchemy.types.JSONElementType

Placeholder type for JSON path operations.

JSON路径操作的占位符类型。

This allows execution-time processing of a path-based index value into a specific SQL syntax.

这允许将基于路径的索引值的执行时处理为特定的SQL语法。

**NULL***= symbol('JSON\_NULL')*

Describe the json value of NULL.

描述json值为NULL。

This value is used to force the JSON value of "null" to be used as the value. A value of Python None will be recognized either as SQL NULL or JSON "null", based on the setting of the [JSON.none\_as\_null](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.JSON.params.none_as_null" \o "sqlalchemy.types.JSON) flag; the [JSON.NULL](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.JSON.NULL" \o "sqlalchemy.types.JSON.NULL) constant can be used to always resolve to JSON "null" regardless of this setting. This is in contrast to the sql.null() construct, which always resolves to SQL NULL. E.g.:

该值用于强制使用“null”的JSON值作为值。 基于JSON.none\_as\_null标志的设置，Python None的值将被识别为SQL NULL或JSON“null”。 无论此设置如何，JSON.NULL常量都可以用于始终解析为JSON“null”。 这与sql.null() 结构形成对比，它总是解析为SQL NULL。 例如。：

**from** **sqlalchemy** **import** null

**from** **sqlalchemy.dialects.postgresql** **import** JSON

obj1 = MyObject(json\_value=null()) *# will \*always\* insert SQL NULL*obj2 = MyObject(json\_value=JSON.NULL) *# will \*always\* insert JSON string "null"*

session.add\_all([obj1, obj2])session.commit()

In order to set JSON NULL as a default value for a column, the most transparent method is to use [text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text):

为了将JSON NULL设置为列的默认值，最透明的方法是使用text() ：

Table(

'my\_table', metadata,

Column('json\_data', JSON, default=text("'null'")))

While it is possible to use [JSON.NULL](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.JSON.NULL" \o "sqlalchemy.types.JSON.NULL) in this context, the [JSON.NULL](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.JSON.NULL" \o "sqlalchemy.types.JSON.NULL) value will be returned as the value of the column, which in the context of the ORM or other repurposing of the default value, may not be desirable. Using a SQL expression means the value will be re-fetched from the database within the context of retrieving generated defaults.

虽然可以在此上下文中使用JSON.NULL，但JSON.NULL值将作为列的值返回，该值在ORM的上下文中或默认值的其他重用方式可能不受欢迎。 使用SQL表达式意味着将在检索生成的默认值的上下文中从数据库重新获取该值。

**\_\_init\_\_**(*none\_as\_null=False*)

Construct a [types.JSON](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.JSON" \o "sqlalchemy.types.JSON) type.

|  |  |
| --- | --- |
| **Parameters:** | ****none\_as\_null=False**** –  if True, persist the value None as a SQL NULL value, not the JSON encoding of null. Note that when this flag is False, the [null()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.null" \o "sqlalchemy.sql.expression.null) construct can still be used to persist a NULL value:  如果为True，则将值None作为SQL NULL值，而不是JSON的编码。 请注意，当该标志为False时，仍然可以使用null() 结构来保留NULL值：  **from** **sqlalchemy** **import** nullconn.execute(table.insert(), data=null())  **Note**  [JSON.none\_as\_null](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.JSON.params.none_as_null" \o "sqlalchemy.types.JSON) does ****not**** apply to the values passed to [Column.default](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column.params.default" \o "sqlalchemy.schema.Column) and [Column.server\_default](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column.params.server_default" \o "sqlalchemy.schema.Column); a value of None passed for these parameters means "no default present".  JSON.none\_as\_null不适用于传递给Column.default和Column.server\_default的值; 这些参数的值为None，表示“no default present”。  **See also**  [types.JSON.NULL](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.JSON.NULL" \o "sqlalchemy.types.JSON.NULL) |

**comparator\_factory**

alias of [Comparator](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.JSON.Comparator" \o "sqlalchemy.types.JSON.Comparator)

*class*sqlalchemy.types.**INTEGER**

Bases: [sqlalchemy.types.Integer](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Integer" \o "sqlalchemy.types.Integer)

The SQL INT or INTEGER type.

*class*sqlalchemy.types.**NCHAR**(*length=None*, *\*\*kwargs*)

Bases: [sqlalchemy.types.Unicode](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Unicode" \o "sqlalchemy.types.Unicode)

The SQL NCHAR type.

*class*sqlalchemy.types.**NVARCHAR**(*length=None*, *\*\*kwargs*)

Bases: [sqlalchemy.types.Unicode](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Unicode" \o "sqlalchemy.types.Unicode)

The SQL NVARCHAR type.

*class*sqlalchemy.types.**NUMERIC**(*precision=None*, *scale=None*, *decimal\_return\_scale=None*, *asdecimal=True*)

Bases: [sqlalchemy.types.Numeric](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Numeric" \o "sqlalchemy.types.Numeric)

The SQL NUMERIC type.

*class*sqlalchemy.types.**REAL**(*precision=None*, *asdecimal=False*, *decimal\_return\_scale=None*, *\*\*kwargs*)

Bases: [sqlalchemy.types.Float](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Float" \o "sqlalchemy.types.Float)

The SQL REAL type.

*class*sqlalchemy.types.**SMALLINT**

Bases: [sqlalchemy.types.SmallInteger](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.SmallInteger" \o "sqlalchemy.types.SmallInteger)

The SQL SMALLINT type.

*class*sqlalchemy.types.**TEXT**(*length=None*, *collation=None*, *convert\_unicode=False*, *unicode\_error=None*, *\_warn\_on\_bytestring=False*)

Bases: [sqlalchemy.types.Text](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Text" \o "sqlalchemy.types.Text)

The SQL TEXT type.

*class*sqlalchemy.types.**TIME**(*timezone=False*)

Bases: [sqlalchemy.types.Time](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Time" \o "sqlalchemy.types.Time)

The SQL TIME type.

*class*sqlalchemy.types.**TIMESTAMP**(*timezone=False*)

Bases: [sqlalchemy.types.DateTime](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.DateTime" \o "sqlalchemy.types.DateTime)

The SQL TIMESTAMP type.

[TIMESTAMP](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.TIMESTAMP" \o "sqlalchemy.types.TIMESTAMP) datatypes have support for timezone storage on some backends, such as PostgreSQL and Oracle. Use the timezone argument in order to enable "TIMESTAMP WITH TIMEZONE" for these backends.

TIMESTAMP数据类型支持某些后端的时区存储，如PostgreSQL和Oracle。 使用timezone参数为这些后端启用“TIMESTAMP WITH TIMEZONE”。

**\_\_init\_\_**(*timezone=False*)

Construct a new [TIMESTAMP](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.TIMESTAMP" \o "sqlalchemy.types.TIMESTAMP).

|  |  |
| --- | --- |
| **Parameters:** | ****timezone**** – boolean. Indicates that the TIMESTAMP type should enable timezone support, if available on the target database. On a per-dialect basis is similar to "TIMESTAMP WITH TIMEZONE". If the target database does not support timezones, this flag is ignored.  布尔值。 表示TIMESTAMP类型应启用时区支持（如果在目标数据库上可用）。 在每个方言的基础上，类似于“TIMESTAMP WITH TIMEZONE”。 如果目标数据库不支持时区，该标志将被忽略。 |

*class*sqlalchemy.types.**VARBINARY**(*length=None*)

Bases: sqlalchemy.types.\_Binary

The SQL VARBINARY type.

*class*sqlalchemy.types.**VARCHAR**(*length=None*, *collation=None*, *convert\_unicode=False*, *unicode\_error=None*, *\_warn\_on\_bytestring=False*)

Bases: [sqlalchemy.types.String](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.String" \o "sqlalchemy.types.String)

The SQL VARCHAR type.

4.1.3 Vendor-Specific Types

Database-specific types are also available for import from each database's dialect module. See the [Dialects](http://docs.sqlalchemy.org/en/rel_1_1/dialects/index.html) reference for the database you're interested in.

数据库特定类型也可从每个数据库的方言模块导入。 请参阅您感兴趣的数据库的方言参考。

For example, MySQL has a BIGINT type and PostgreSQL has an INET type. To use these, import them from the module explicitly:

例如，MySQL具有BIGINT类型，PostgreSQL具有INET类型。 要使用这些，请从模块中导入它们：

**from** **sqlalchemy.dialects** **import** mysql

table = Table('foo', metadata,

Column('id', mysql.BIGINT),

Column('enumerates', mysql.ENUM('a', 'b', 'c')))

Or some PostgreSQL types:

**from** **sqlalchemy.dialects** **import** postgresql

table = Table('foo', metadata,

Column('ipaddress', postgresql.INET),

Column('elements', postgresql.ARRAY(String)))

Each dialect provides the full set of typenames supported by that backend within its \_\_all\_\_ collection, so that a simple import \* or similar will import all supported types as implemented for that backend:

每个方言提供其\_\_all\_\_集合中该后端支持的完整的类型名称，以便简单的导入\*或类似的方式将导入为该后端实现的所有支持的类型：

**from** **sqlalchemy.dialects.postgresql** **import** \*

t = Table('mytable', metadata,

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

Column('name', VARCHAR(300)),

Column('inetaddr', INET))

Where above, the INTEGER and VARCHAR types are ultimately from sqlalchemy.types, and INET is specific to the PostgreSQL dialect.

在上面，INTEGER和VARCHAR类型最终来自sqlalchemy.types，INET特定于PostgreSQL方言。

Some dialect level types have the same name as the SQL standard type, but also provide additional arguments. For example, MySQL implements the full range of character and string types including additional arguments such as collation and charset:

某些方言级别类型与SQL标准类型具有相同的名称，但也提供其他参数。 例如，MySQL实现了完整的字符和字符串类型，包括其他参数，如排序规则和字符集：

**from** **sqlalchemy.dialects.mysql** **import** VARCHAR, TEXT

table = Table('foo', meta,

Column('col1', VARCHAR(200, collation='binary')),

Column('col2', TEXT(charset='latin1')))

## Custom Types

A variety of methods exist to redefine the behavior of existing types as well as to provide new ones.

存在各种方法来重新定义现有类型的行为以及提供新的类型。

4.2.1 Overriding Type Compilation

A frequent need is to force the "string" version of a type, that is the one rendered in a CREATE TABLE statement or other SQL function like CAST, to be changed. For example, an application may want to force the rendering of BINARY for all platforms except for one, in which is wants BLOB to be rendered. Usage of an existing generic type, in this case [LargeBinary](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.LargeBinary" \o "sqlalchemy.types.LargeBinary), is preferred for most use cases. But to control types more accurately, a compilation directive that is per-dialect can be associated with any type:

频繁的需要是强制类型的"字符串"版本，即在CREATE TABLE语句或其他SQL函数（如CAST）中呈现的"字符串"版本进行更改。 例如，应用程序可能希望强制对除所有平台之外的所有平台的BINARY进行渲染，其中需要渲染BLOB。 现有的泛型类型的使用（在这种情况下为大型Binary），对大多数用例而言都是首选。 但是要更精确地控制类型，每个方言的编译指令可以与任何类型相关联：

**from** **sqlalchemy.ext.compiler** **import** compiles

**from** **sqlalchemy.types** **import** BINARY

**@compiles**(BINARY, "sqlite")**def** compile\_binary\_sqlite(type\_, compiler, \*\*kw):

**return** "BLOB"

The above code allows the usage of [types.BINARY](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.BINARY" \o "sqlalchemy.types.BINARY), which will produce the string BINARY against all backends except SQLite, in which case it will produce BLOB.

上述代码允许使用types.BINARY，它将针对除SQLite之外的所有后端产生字符串BINARY，在这种情况下，它将生成BLOB。

See the section [Changing Compilation of Types](http://docs.sqlalchemy.org/en/rel_1_1/core/compiler.html" \l "type-compilation-extension), a subsection of [Custom SQL Constructs and Compilation Extension](http://docs.sqlalchemy.org/en/rel_1_1/core/compiler.html), for additional examples.

有关其他示例，请参阅"更改编辑类型"一节，"自定义SQL构造和编译扩展"一节。

4.2.2 Augmenting Existing Types

The [TypeDecorator](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator" \o "sqlalchemy.types.TypeDecorator) allows the creation of custom types which add bind-parameter and result-processing behavior to an existing type object. It is used when additional in-Python marshaling of data to and from the database is required.

TypeDecorator允许创建自定义类型，它们将绑定参数和结果处理行为添加到现有类型的对象。 当需要在数据库之外附加的Python数据传输时使用它。

**Note**

注意

The bind- and result-processing of [TypeDecorator](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator" \o "sqlalchemy.types.TypeDecorator) is *in addition* to the processing already performed by the hosted type, which is customized by SQLAlchemy on a per-DBAPI basis to perform processing specific to that DBAPI. To change the DBAPI-level processing for an existing type, see the section [Replacing the Bind/Result Processing of Existing Types](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "replacing-processors).

TypeDecorator的绑定和结果处理是由托管类型执行的处理之外的补充，该对象类型由SQLAlchemy根据每个DBAPI进行定制，以执行特定于该DBAPI的处理。 要更改现有类型的DBAPI级别处理，请参阅"替换现有类型的绑定/结果处理"一节。

*class*sqlalchemy.types.**TypeDecorator**(*\*args*, *\*\*kwargs*)

Bases: sqlalchemy.sql.expression.SchemaEventTarget, [sqlalchemy.types.TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine)

Allows the creation of types which add additional functionality to an existing type.

允许创建为现有类型添加附加功能的类型。

This method is preferred to direct subclassing of SQLAlchemy's built-in types as it ensures that all required functionality of the underlying type is kept in place.

这种方法优先于SQLAlchemy的内置类型的直接子类化，因为它可以确保底层类型的所有必需功能保持原位。

Typical usage:

典型用法：

**import** **sqlalchemy.types** **as** **types**

**class** **MyType**(types.TypeDecorator):

*'''Prefixes Unicode values with "PREFIX:" on the way in and strips it off on the way out. '''*

impl = types.Unicode

**def** process\_bind\_param(self, value, dialect):

**return** "PREFIX:" + value

**def** process\_result\_value(self, value, dialect):

**return** value[7:]

**def** copy(self, \*\*kw):

**return** MyType(self.impl.length)

The class-level "impl" attribute is required, and can reference any TypeEngine class. Alternatively, the load\_dialect\_impl() method can be used to provide different type classes based on the dialect given; in this case, the "impl" variable can reference TypeEngine as a placeholder.

类级别"impl"属性是必需的，可以引用任何TypeEngine类。 或者，可以使用load\_dialect\_impl() 方法根据给出的方言来提供不同类型的类; 在这种情况下，"impl"变量可以引用TypeEngine作为占位符。

Types that receive a Python type that isn't similar to the ultimate type used may want to define the [TypeDecorator.coerce\_compared\_value()](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator.coerce_compared_value" \o "sqlalchemy.types.TypeDecorator.coerce_compared_value) method. This is used to give the expression system a hint when coercing Python objects into bind parameters within expressions. Consider this expression:

接收与所使用的最终类型不相似的Python类型的类型可能需要定义TypeDecorator.coerce\_compared\_value() 方法。 当将Python对象强制为表达式中的绑定参数时，这用于给表达式系统提供一个提示。 考虑这个表达式：

mytable.c.somecol + datetime.date(2009, 5, 15)

Above, if "somecol" is an Integer variant, it makes sense that we're doing date arithmetic, where above is usually interpreted by databases as adding a number of days to the given date. The expression system does the right thing by not attempting to coerce the "date()" value into an integer-oriented bind parameter.

以上，如果"somecol"是一个整数变体，那么我们正在进行日期算术是有道理的，上面的数据通常由数据库解释为在给定日期添加了几天。 表达式系统通过不尝试将"date() "值强制转换成面向整数的绑定参数来做正确的事情。

However, in the case of TypeDecorator, we are usually changing an incoming Python type to something new - TypeDecorator by default will "coerce" the non-typed side to be the same type as itself. Such as below, we define an "epoch" type that stores a date value as an integer:

然而，在TypeDecorator的情况下，我们通常会将传入的Python类型更改为新的类型 - 默认情况下，TypeDecorator将"强制"非类型方与本身相同。 如下所示，我们定义一个"日期"类型，将日期值存储为整数：

**class** **MyEpochType**(types.TypeDecorator):

impl = types.Integer

epoch = datetime.date(1970, 1, 1)

**def** process\_bind\_param(self, value, dialect):

**return** (value - self.epoch).days

**def** process\_result\_value(self, value, dialect):

**return** self.epoch + timedelta(days=value)

Our expression of somecol + date with the above type will coerce the "date" on the right side to also be treated as MyEpochType.

我们用上述类型表示somecol + date将强制右侧的"date"也被视为MyEpochType。

This behavior can be overridden via the [coerce\_compared\_value()](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator.coerce_compared_value" \o "sqlalchemy.types.TypeDecorator.coerce_compared_value) method, which returns a type that should be used for the value of the expression. Below we set it such that an integer value will be treated as an Integer, and any other value is assumed to be a date and will be treated as a MyEpochType:

可以通过coerce\_compared\_value() 方法覆盖此行为，该方法返回应该用于表达式的值的类型。 下面我们设置一个整数值将被视为一个整数，任何其他值被假定为一个日期，并将被视为一个MyEpochType：

**def** coerce\_compared\_value(self, op, value):

**if** isinstance(value, int):

**return** Integer()

**else**:

**return** self

**Warning**

Note that the ****behavior of coerce\_compared\_value is not inherited by default from that of the base type****. If the [TypeDecorator](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator" \o "sqlalchemy.types.TypeDecorator) is augmenting a type that requires special logic for certain types of operators, this method ****must**** be overridden. A key example is when decorating the [postgresql.JSON](http://docs.sqlalchemy.org/en/rel_1_1/dialects/postgresql.html" \l "sqlalchemy.dialects.postgresql.JSON" \o "sqlalchemy.dialects.postgresql.JSON) and [postgresql.JSONB](http://docs.sqlalchemy.org/en/rel_1_1/dialects/postgresql.html" \l "sqlalchemy.dialects.postgresql.JSONB" \o "sqlalchemy.dialects.postgresql.JSONB) types; the default rules of [TypeEngine.coerce\_compared\_value()](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.coerce_compared_value" \o "sqlalchemy.types.TypeEngine.coerce_compared_value) should be used in order to deal with operators like index operations:

请注意，默认情况下，coerce\_compared\_value的行为不会从基本类型继承。 如果TypeDecorator正在扩充一种对特定类型的运算符需要特殊逻辑的类型，则必须覆盖该方法。 一个关键的例子是装饰postgresql.JSON和postgresql.JSONB类型时; 应该使用TypeEngine.coerce\_compared\_value() 的默认规则来处理像索引操作这样的运算符：

**class** **MyJsonType**(TypeDecorator):

impl = postgresql.JSON

**def** coerce\_compared\_value(self, op, value):

**return** self.impl.coerce\_compared\_value(op, value)

Without the above step, index operations such as mycol['foo'] will cause the index value 'foo' to be JSON encoded.

**\_\_init\_\_**(*\*args*, *\*\*kwargs*)

Construct a [TypeDecorator](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator" \o "sqlalchemy.types.TypeDecorator).

Arguments sent here are passed to the constructor of the class assigned to the impl class level attribute, assuming the impl is a callable, and the resulting object is assigned to the self.impl instance attribute (thus overriding the class attribute of the same name).

这里发送的参数传递给分配给impl类级别属性的类的构造函数，假设impl是可调用的，并将生成的对象分配给self.impl实例属性（从而覆盖同名的类属性）。

If the class level impl is not a callable (the unusual case), it will be assigned to the same instance attribute 'as-is', ignoring those arguments passed to the constructor.

如果类级别impl不是可调用（异常情况），那么它将被分配给同样的实例属性"as-is"，忽略传递给构造函数的参数

Subclasses can override this to customize the generation of self.impl entirely.

子类可以覆盖这个来完全自定义self.impl的生成。

**adapt**(*cls*, *\*\*kw*)

*inherited from the* [adapt()](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.adapt" \o "sqlalchemy.types.TypeEngine.adapt) *method of* [TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine)

Produce an "adapted" form of this type, given an "impl" class to work with.

This method is used internally to associate generic types with "implementation" types that are specific to a particular dialect.

产生一种"适应"的这种类型的形式，给予一个"impl"类与之配合。

内部使用此方法将通用类型与特定于特定方言的"实现"类型相关联。

**bind\_expression**(*bindvalue*)

*inherited from the* [bind\_expression()](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.bind_expression" \o "sqlalchemy.types.TypeEngine.bind_expression) *method of* [TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine)

"Given a bind value (i.e. a [BindParameter](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.BindParameter" \o "sqlalchemy.sql.expression.BindParameter) instance), return a SQL expression in its place.

"给定绑定值（即BindParameter实例），返回一个SQL表达式。

This is typically a SQL function that wraps the existing bound parameter within the statement. It is used for special data types that require literals being wrapped in some special database function in order to coerce an application-level value into a database-specific format. It is the SQL analogue of the[TypeEngine.bind\_processor()](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.bind_processor" \o "sqlalchemy.types.TypeEngine.bind_processor) method.

这通常是一个SQL函数，它将现有的绑定参数包含在语句中。 它用于需要将文字包装在某些特殊数据库函数中的特殊数据类型，以将应用程序级值强制为特定于数据库的格式。 它是TypeEngine.bind\_processor() 方法的SQL模拟。

The method is evaluated at statement compile time, as opposed to statement construction time.

该方法在语句编译时进行评估，而不是语句构建时间。

Note that this method, when implemented, should always return the exact same structure, without any conditional logic, as it may be used in an executemany() call against an arbitrary number of bound parameter sets.

请注意，该方法在实现时应始终返回完全相同的结构，而不需要任何条件逻辑，因为它可以用于针对任意数量的绑定参数集的executemany() 调用。

See also:

[Applying SQL-level Bind/Result Processing](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "types-sql-value-processing)

也可以看看：

应用SQL级绑定/结果处理

**bind\_processor**(*dialect*)

Provide a bound value processing function for the given [Dialect](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Dialect" \o "sqlalchemy.engine.interfaces.Dialect).

为给定的方言提供绑定值处理功能。

This is the method that fulfills the [TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine) contract for bound value conversion. [TypeDecorator](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator" \o "sqlalchemy.types.TypeDecorator) will wrap a user-defined implementation of[process\_bind\_param()](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator.process_bind_param" \o "sqlalchemy.types.TypeDecorator.process_bind_param) here.

这是满足绑定值转换的TypeEngine合同的方法。 TypeDecorator将在这里包装一个用户定义的processprocess\_bind\_param() 实现。

User-defined code can override this method directly, though its likely best to use [process\_bind\_param()](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator.process_bind_param" \o "sqlalchemy.types.TypeDecorator.process_bind_param) so that the processing provided by self.impl is maintained.

用户定义的代码可以直接覆盖此方法，尽管它最好使用process\_bind\_param() ，以便维护由self.impl提供的处理。

|  |  |
| --- | --- |
| **Parameters:** | ****dialect**** – Dialect instance in use. |

This method is the reverse counterpart to the [result\_processor()](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator.result_processor" \o "sqlalchemy.types.TypeDecorator.result_processor) method of this class.

此方法与此类的result\_processor() 方法相反。

**coerce\_compared\_value**(*op*, *value*)

Suggest a type for a 'coerced' Python value in an expression.

在表达式中建议一个“强制”Python值的类型。

By default, returns self. This method is called by the expression system when an object using this type is on the left or right side of an expression against a plain Python object which does not yet have a SQLAlchemy type assigned:

默认情况下，返回自身。 当使用此类型的对象位于表达式的左侧或右侧时，表达式系统会调用此方法，该对象对于尚未分配SQLAlchemy类型的普通Python对象：

expr = table.c.somecolumn + 35

Where above, if somecolumn uses this type, this method will be called with the value operator.add and 35. The return value is whatever SQLAlchemy type should be used for 35 for this particular operation.

在上面的位置，如果somecolumn使用此类型，则将使用值为operator.add和35调用此方法。返回值是用于此特定操作的任何SQLAlchemy类型应用于35。

**coerce\_to\_is\_types***= (<type 'NoneType'>,)*

Specify those Python types which should be coerced at the expression level to "IS <constant>" when compared using == (and same for IS NOT in conjunction with !=.

指定这些Python类型，当与==比较时，应该在表达式级别强制为“IS <constant>”（并且与IS不一致！=）。

对于大多数SQLAlchemy类型，这包括NoneType以及bool。

For most SQLAlchemy types, this includes NoneType, as well as bool.

[TypeDecorator](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator" \o "sqlalchemy.types.TypeDecorator) modifies this list to only include NoneType, as typedecorator implementations that deal with boolean types are common.

TypeDecorator将此列表修改为仅包含NoneType，因为处理布尔类型的typedecorator实现是常见的。

Custom [TypeDecorator](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator" \o "sqlalchemy.types.TypeDecorator) classes can override this attribute to return an empty tuple, in which case no values will be coerced to constants.

Custom TypeDecorator类可以覆盖此属性以返回一个空的元组，在这种情况下，不会将任何值强制为常量。

*New in version 0.8.2:*Added [TypeDecorator.coerce\_to\_is\_types](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator.coerce_to_is_types" \o "sqlalchemy.types.TypeDecorator.coerce_to_is_types) to allow for easier control of \_\_eq\_\_() \_\_ne\_\_() operations.

新版本0.8.2：添加了TypeDecorator.coerce\_to\_is\_types以便更容易地控制\_\_eq \_\_() \_\_ne \_\_() 操作。

**column\_expression**(*colexpr*)

*inherited from the* [column\_expression()](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.column_expression" \o "sqlalchemy.types.TypeEngine.column_expression) *method of* [TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine)

Given a SELECT column expression, return a wrapping SQL expression.

给定一个SELECT列表达式，返回一个包装SQL表达式。

This is typically a SQL function that wraps a column expression as rendered in the columns clause of a SELECT statement. It is used for special data types that require columns to be wrapped in some special database function in order to coerce the value before being sent back to the application. It is the SQL analogue of the [TypeEngine.result\_processor()](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.result_processor" \o "sqlalchemy.types.TypeEngine.result_processor) method.

这通常是一个SQL函数，它将列表达式包含在SELECT语句的columns子句中。 它用于特殊数据类型，需要将列包装在某些特殊数据库函数中，以便在发送回应用程序之前强制该值。 它是TypeEngine.result\_processor() 方法的SQL模拟。

The method is evaluated at statement compile time, as opposed to statement construction time.

该方法在语句编译时进行评估，而不是语句构建时间。

See also:

[Applying SQL-level Bind/Result Processing](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "types-sql-value-processing)

**compare\_against\_backend**(*dialect*, *conn\_type*)

*inherited from the* [compare\_against\_backend()](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.compare_against_backend" \o "sqlalchemy.types.TypeEngine.compare_against_backend) *method of* [TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine)

Compare this type against the given backend type.

将此类型与给定的后端类型进行比较。

This function is currently not implemented for SQLAlchemy types, and for all built in types will return None. However, it can be implemented by a user-defined type where it can be consumed by schema comparison tools such as Alembic autogenerate.

此功能目前尚未实现为SQLAlchemy类型，而对于所有内置类型将返回无。 但是，它可以通过用户定义的类型来实现，它可以通过模式比较工具（如Alembic autogenerate）来消耗。

A future release of SQLAlchemy will potentially impement this method for builtin types as well.

SQLAlchemy的未来版本也将对内置类型的这种方法产生影响。

The function should return True if this type is equivalent to the given type; the type is typically reflected from the database so should be database specific. The dialect in use is also passed. It can also return False to assert that the type is not equivalent.

如果此类型等于给定类型，该函数应返回True; 该类型通常从数据库反映出来，因此应该是数据库特定的。 正在使用的方言也通过。 它也可以返回False来声明该类型不等同。

|  |  |
| --- | --- |
| **Parameters:** | * ****dialect**** – a [Dialect](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Dialect" \o "sqlalchemy.engine.interfaces.Dialect) that is involved in the comparison. * ****conn\_type**** – the type object reflected from the backend. |

*New in version 1.0.3.*

**compare\_values**(*x*, *y*)

Given two values, compare them for equality.

By default this calls upon [TypeEngine.compare\_values()](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.compare_values" \o "sqlalchemy.types.TypeEngine.compare_values) of the underlying "impl", which in turn usually uses the Python equals operator ==.

This function is used by the ORM to compare an original-loaded value with an intercepted "changed" value, to determine if a net change has occurred.

**compile**(*dialect=None*)

*inherited from the* [compile()](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.compile" \o "sqlalchemy.types.TypeEngine.compile) *method of* [TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine)

Produce a string-compiled form of this [TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine).

When called with no arguments, uses a "default" dialect to produce a string result.

|  |  |
| --- | --- |
| **Parameters:** | ****dialect**** – a [Dialect](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Dialect" \o "sqlalchemy.engine.interfaces.Dialect) instance. |

**copy**(*\*\*kw*)

Produce a copy of this [TypeDecorator](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator" \o "sqlalchemy.types.TypeDecorator) instance.

This is a shallow copy and is provided to fulfill part of the [TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine) contract. It usually does not need to be overridden unless the user-defined [TypeDecorator](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator" \o "sqlalchemy.types.TypeDecorator) has local state that should be deep-copied.

**dialect\_impl**(*dialect*)

*inherited from the* [dialect\_impl()](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.dialect_impl" \o "sqlalchemy.types.TypeEngine.dialect_impl) *method of* [TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine)

Return a dialect-specific implementation for this [TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine).

**evaluates\_none**()

*inherited from the* [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 of* [TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine)

Return a copy of this type which has the [should\_evaluate\_none](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.should_evaluate_none" \o "sqlalchemy.types.TypeEngine.should_evaluate_none) flag set to True.

E.g.:

Table(

'some\_table', metadata,

Column(

String(50).evaluates\_none(),

nullable=**True**,

server\_default='no value'))

The ORM uses this flag to indicate that a positive value of None is passed to the column in an INSERT statement, rather than omitting the column from the INSERT statement which has the effect of firing off column-level defaults. It also allows for types which have special behavior associated with the Python None value to indicate that the value doesn't necessarily translate into SQL NULL; a prime example of this is a JSON type which may wish to persist the JSON value 'null'.

In all cases, the actual NULL SQL value can be always be persisted in any column by 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 in an INSERT statement or associated with an ORM-mapped attribute.

**Note**

The "evaulates none" flag does ****not**** apply to a value of None passed to [Column.default](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column.params.default" \o "sqlalchemy.schema.Column) or [Column.server\_default](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column.params.server_default" \o "sqlalchemy.schema.Column); in these cases, None still means "no default".

*New in version 1.1.*

**See also**

[Forcing NULL on a column with a default](http://docs.sqlalchemy.org/en/rel_1_1/orm/persistence_techniques.html" \l "session-forcing-null) - in the ORM documentation

[postgresql.JSON.none\_as\_null](http://docs.sqlalchemy.org/en/rel_1_1/dialects/postgresql.html" \l "sqlalchemy.dialects.postgresql.JSON.params.none_as_null" \o "sqlalchemy.dialects.postgresql.JSON) - PostgreSQL JSON interaction with this flag.

[TypeEngine.should\_evaluate\_none](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.should_evaluate_none" \o "sqlalchemy.types.TypeEngine.should_evaluate_none) - class-level flag

**get\_dbapi\_type**(*dbapi*)

Return the DBAPI type object represented by this [TypeDecorator](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator" \o "sqlalchemy.types.TypeDecorator).

By default this calls upon [TypeEngine.get\_dbapi\_type()](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.get_dbapi_type" \o "sqlalchemy.types.TypeEngine.get_dbapi_type) of the underlying "impl".

**literal\_processor**(*dialect*)

Provide a literal processing function for the given [Dialect](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Dialect" \o "sqlalchemy.engine.interfaces.Dialect).

Subclasses here will typically override [TypeDecorator.process\_literal\_param()](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator.process_literal_param" \o "sqlalchemy.types.TypeDecorator.process_literal_param) instead of this method directly.

By default, this method makes use of [TypeDecorator.process\_bind\_param()](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator.process_bind_param" \o "sqlalchemy.types.TypeDecorator.process_bind_param) if that method is implemented, where [TypeDecorator.process\_literal\_param()](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator.process_literal_param" \o "sqlalchemy.types.TypeDecorator.process_literal_param) is not. The rationale here is that [TypeDecorator](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator" \o "sqlalchemy.types.TypeDecorator) typically deals with Python conversions of data that are above the layer of database presentation. With the value converted by [TypeDecorator.process\_bind\_param()](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator.process_bind_param" \o "sqlalchemy.types.TypeDecorator.process_bind_param), the underlying type will then handle whether it needs to be presented to the DBAPI as a bound parameter or to the database as an inline SQL value.

*New in version 0.9.0.*

**load\_dialect\_impl**(*dialect*)

Return a [TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine) object corresponding to a dialect.

This is an end-user override hook that can be used to provide differing types depending on the given dialect. It is used by the [TypeDecorator](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator" \o "sqlalchemy.types.TypeDecorator)implementation of [type\_engine()](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator.type_engine" \o "sqlalchemy.types.TypeDecorator.type_engine) to help determine what type should ultimately be returned for a given [TypeDecorator](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator" \o "sqlalchemy.types.TypeDecorator).

By default returns self.impl.

**process\_bind\_param**(*value*, *dialect*)

Receive a bound parameter value to be converted.

接收要转换的绑定参数值。

Subclasses override this method to return the value that should be passed along to the underlying [TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine) object, and from there to the DBAPI execute() method.

子类覆盖此方法以将要传递的值返回到底层的TypeEngine对象，并从那里返回到DBAPI execute() 方法。

The operation could be anything desired to perform custom behavior, such as transforming or serializing data. This could also be used as a hook for validating logic.

操作可能是任何需要执行自定义行为的任何操作，例如转换或序列化数据。 这也可以用作验证逻辑的钩子。

This operation should be designed with the reverse operation in mind, which would be the process\_result\_value method of this class.

这个操作应该设计为反向操作，这将是此类的process\_result\_value方法。

|  |  |
| --- | --- |
| **Parameters:** | * ****value**** – Data to operate upon, of any type expected by this method in the subclass. Can be None. * ****dialect**** – the [Dialect](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Dialect" \o "sqlalchemy.engine.interfaces.Dialect) in use. |

**process\_literal\_param**(*value*, *dialect*)

Receive a literal parameter value to be rendered inline within a statement.

接收一个文字参数值，以便在一个语句内呈现。

This method is used when the compiler renders a literal value without using binds, typically within DDL such as in the "server default" of a column or an expression within a CHECK constraint.

当编译器在不使用绑定的情况下呈现文字值时使用此方法，通常在DDL中，例如列的“服务器默认”或CHECK约束中的表达式。

The returned string will be rendered into the output string.

返回的字符串将被渲染为输出字符串。

*New in version 0.9.0.*

**process\_result\_value**(*value*, *dialect*)

Receive a result-row column value to be converted.

接收要转换的结果行列值。

Subclasses should implement this method to operate on data fetched from the database.

子类应实现此方法来对从数据库获取的数据进行操作。

Subclasses override this method to return the value that should be passed back to the application, given a value that is already processed by the underlying [TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine) object, originally from the DBAPI cursor method fetchone() or similar.

给定一个已经由底层的TypeEngine对象处理的值（最初来自DBAPI游标方法fetchone() 或类似的值），Subclasses将覆盖此方法返回应该传回应用程序的值。

The operation could be anything desired to perform custom behavior, such as transforming or serializing data. This could also be used as a hook for validating logic.

操作可能是任何需要执行自定义行为的任何操作，例如转换或序列化数据。 这也可以用作验证逻辑的钩子。

|  |  |
| --- | --- |
| **Parameters:** | * ****value**** – Data to operate upon, of any type expected by this method in the subclass. Can be None. * ****dialect**** – the [Dialect](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Dialect" \o "sqlalchemy.engine.interfaces.Dialect) in use. |

This operation should be designed to be reversible by the "process\_bind\_param" method of this class.

该操作应该被设计为可以通过此类的“process\_bind\_param”方法来逆转。

**python\_type**

*inherited from the* [python\_type](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.python_type" \o "sqlalchemy.types.TypeEngine.python_type) *attribute of* [TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine)

Return the Python type object expected to be returned by instances of this type, if known.

返回预期由此类型的实例返回的Python类型对象（如果已知）。

Basically, for those types which enforce a return type, or are known across the board to do such for all common DBAPIs (like int for example), will return that type.

基本上，对于那些强制执行返回类型的类型，或者是全面知道为所有常见的DBAPI（例如int）这样做的类型，将返回该类型。

If a return type is not defined, raises NotImplementedError.

如果未定义返回类型，则引发NotImplementedError。

Note that any type also accommodates NULL in SQL which means you can also get back None from any type in practice.

请注意，任何类型的SQL也适用于NULL，这意味着您也可以从实践中获取任何类型的无。

**result\_processor**(*dialect*, *coltype*)

Provide a result value processing function for the given [Dialect](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Dialect" \o "sqlalchemy.engine.interfaces.Dialect).

为给定的方言提供结果值处理功能。

This is the method that fulfills the [TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine) contract for result value conversion. [TypeDecorator](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator" \o "sqlalchemy.types.TypeDecorator) will wrap a user-defined implementation of[process\_result\_value()](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator.process_result_value" \o "sqlalchemy.types.TypeDecorator.process_result_value) here.

这是实现结果值转换的TypeEngine合同的方法。 TypeDecorator将在此包装一个用户定义的实现的process\_result\_value() 。

User-defined code can override this method directly, though its likely best to use [process\_result\_value()](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator.process_result_value" \o "sqlalchemy.types.TypeDecorator.process_result_value) so that the processing provided by self.impl is maintained.

用户定义的代码可以直接覆盖此方法，尽管它最好使用process\_result\_value() ，以便维护由self.impl提供的处理。

|  |  |
| --- | --- |
| **Parameters:** | * ****dialect**** – Dialect instance in use. * ****coltype**** – A SQLAlchemy data type |

This method is the reverse counterpart to the [bind\_processor()](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator.bind_processor" \o "sqlalchemy.types.TypeDecorator.bind_processor) method of this class.

此方法与此类的bind\_processor() 方法相反。

**type\_engine**(*dialect*)

Return a dialect-specific [TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine) instance for this [TypeDecorator](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator" \o "sqlalchemy.types.TypeDecorator).

返回此TypeDecorator的特定于方言的TypeEngine实例。

In most cases this returns a dialect-adapted form of the [TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine) type represented by self.impl. Makes usage of [dialect\_impl()](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator.dialect_impl" \o "sqlalchemy.types.TypeDecorator.dialect_impl) but also traverses into wrapped [TypeDecorator](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator" \o "sqlalchemy.types.TypeDecorator) instances. Behavior can be customized here by overriding [load\_dialect\_impl()](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator.load_dialect_impl" \o "sqlalchemy.types.TypeDecorator.load_dialect_impl).

在大多数情况下，这将返回由self.impl表示的TypeEngine类型的方言适配形式。 使用dialect\_impl() ，也可以遍历到包装的TypeDecorator实例。 可以通过覆盖load\_dialect\_impl() 来定制行为。

**with\_variant**(*type\_*, *dialect\_name*)

*inherited from the* [with\_variant()](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.with_variant" \o "sqlalchemy.types.TypeEngine.with_variant) *method of* [TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine)

Produce a new type object that will utilize the given type when applied to the dialect of the given name.

生成一个新的类型对象，当应用于给定名称的方言时，将使用给定的类型。

e.g.:

**from** **sqlalchemy.types** **import** String

**from** **sqlalchemy.dialects** **import** mysql

s = String()

s = s.with\_variant(mysql.VARCHAR(collation='foo'), 'mysql')

The construction of [TypeEngine.with\_variant()](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.with_variant" \o "sqlalchemy.types.TypeEngine.with_variant) is always from the "fallback" type to that which is dialect specific. The returned type is an instance of [Variant](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.Variant" \o "sqlalchemy.types.Variant), which itself provides a Variant.with\_variant() that can be called repeatedly.

TypeEngine.with\_variant() 的构造总是从“回退”类型到具体的方言。 返回的类型是Variant的一个实例，它本身提供了可以重复调用的Variant.with\_variant() 。

|  |  |
| --- | --- |
| **Parameters:** | * ****type\_**** – a [TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine) that will be selected as a variant from the originating type, when a dialect of the given name is in use. * ****dialect\_name**** – base name of the dialect which uses this type. (i.e. 'postgresql', 'mysql', etc.) |

*New in version 0.7.2.*

4.2.3 TypeDecorator Recipes

A few key [TypeDecorator](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator" \o "sqlalchemy.types.TypeDecorator) recipes follow.

### **4.2.3.1 Coercing Encoded Strings to Unicode**

A common source of confusion regarding the [Unicode](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Unicode" \o "sqlalchemy.types.Unicode) type is that it is intended to deal *only* with Python unicode objects on the Python side, meaning values passed to it as bind parameters must be of the form u'some string' if using Python 2 and not 3. The encoding/decoding functions it performs are only to suit what the DBAPI in use requires, and are primarily a private implementation detail.

关于Unicode类型的一个常见的混淆来源是，它仅仅是在Python端处理Python unicode对象，这意味着作为绑定参数传递给它的值必须是u'some string的形式，如果使用Python 2而不是 3.它执行的编码/解码功能只适用于所需的DBAPI，主要是私有的实现细节。

The use case of a type that can safely receive Python bytestrings, that is strings that contain non-ASCII characters and are not u'' objects in Python 2, can be achieved using a [TypeDecorator](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator" \o "sqlalchemy.types.TypeDecorator) which coerces as needed:

可以安全地接收Python bytestrings的类型的用例，即包含非ASCII字符的字符串，而不是Python 2中的u“对象，可以使用根据需要强制的TypeDecorator来实现：

**from** **sqlalchemy.types** **import** TypeDecorator, Unicode

**class** **CoerceUTF8**(TypeDecorator):

*"""Safely coerce Python bytestrings to Unicode before passing off to the database."""*

impl = Unicode

**def** process\_bind\_param(self, value, dialect):

**if** isinstance(value, str):

value = value.decode('utf-8')

**return** value

### **4.2.3.2 Rounding Numerics**

Some database connectors like those of SQL Server choke if a Decimal is passed with too many decimal places. Here's a recipe that rounds them down:

某些数据库连接器，像SQL Server那样的数据库连接器如果十进制数传递了太多的小数位，就会阻塞。 这是一个让他们倒下的食谱：

**from** **sqlalchemy.types** **import** TypeDecorator, Numeric

**from** **decimal** **import** Decimal

**class** **SafeNumeric**(TypeDecorator):

*"""Adds quantization to Numeric."""*

impl = Numeric

**def** \_\_init\_\_(self, \*arg, \*\*kw):

TypeDecorator.\_\_init\_\_(self, \*arg, \*\*kw)

self.quantize\_int = - self.impl.scale

self.quantize = Decimal(10) \*\* self.quantize\_int

**def** process\_bind\_param(self, value, dialect):

**if** isinstance(value, Decimal) **and** \

value.as\_tuple()[2] < self.quantize\_int:

value = value.quantize(self.quantize)

**return** value

### **4.2.3.3 Backend-agnostic GUID Type**

Receives and returns Python uuid() objects. Uses the PG UUID type when using PostgreSQL, CHAR(32) on other backends, storing them in stringified hex format. Can be modified to store binary in CHAR(16) if desired:

接收并返回Python uuid() 对象。 在其他后端使用PostgreSQL，CHAR（32）时使用PG UUID类型，将其存储为十六进制格式。 如果需要，可以修改为在CHAR（16）中存储二进制：

**from** **sqlalchemy.types** **import** TypeDecorator, CHAR

**from** **sqlalchemy.dialects.postgresql** **import** UUID

**import** **uuid**

**class** **GUID**(TypeDecorator):

*"""Platform-independent GUID type.*

*Uses PostgreSQL's UUID type, otherwise uses CHAR(32), storing as stringified hex values.*

*"""*

impl = CHAR

**def** load\_dialect\_impl(self, dialect):

**if** dialect.name == 'postgresql':

**return** dialect.type\_descriptor(UUID())

**else**:

**return** dialect.type\_descriptor(CHAR(32))

**def** process\_bind\_param(self, value, dialect):

**if** value **is** **None**:

**return** value

**elif** dialect.name == 'postgresql':

**return** str(value)

**else**:

**if** **not** isinstance(value, uuid.UUID):

**return** "*%.32x*" % uuid.UUID(value).int

**else**:

*# hexstring*

**return** "*%.32x*" % value.int

**def** process\_result\_value(self, value, dialect):

**if** value **is** **None**:

**return** value

**else**:

**return** uuid.UUID(value)

### **4.2.3.4 Marshal JSON Strings**

This type uses simplejson to marshal Python data structures to/from JSON. Can be modified to use Python's builtin json encoder:

这种类型使用simplejson将JSON数据结构传播到/从JSON传播。 可以修改为使用Python的内置json编码器：

**from** **sqlalchemy.types** **import** TypeDecorator, VARCHAR

**import** **json**

**class** **JSONEncodedDict**(TypeDecorator):

*"""Represents an immutable structure as a json-encoded string.*

*Usage::*

*JSONEncodedDict(255)*

*"""*

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

#### Adding Mutability

The ORM by default will not detect "mutability" on such a type as above - meaning, in-place changes to values will not be detected and will not be flushed. Without further steps, you instead would need to replace the existing value with a new one on each parent object to detect changes:

obj.json\_value["key"] = "value" *# will \*not\* be detected by the ORM*

obj.json\_value = {"key": "value"} *# \*will\* be detected by the ORM*

The above limitation may be fine, as many applications may not require that the values are ever mutated once created. For those which do have this requirement, support for mutability is best applied using the sqlalchemy.ext.mutable extension. For a dictionary-oriented JSON structure, we can apply this as:

json\_type = MutableDict.as\_mutable(JSONEncodedDict)

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

*# ...*

json\_data = Column(json\_type)

**See also**

[Mutation Tracking](http://docs.sqlalchemy.org/en/rel_1_1/orm/extensions/mutable.html)

#### Dealing with Comparison Operations

The default behavior of [TypeDecorator](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator" \o "sqlalchemy.types.TypeDecorator) is to coerce the "right hand side" of any expression into the same type. For a type like JSON, this means that any operator used must make sense in terms of JSON. For some cases, users may wish for the type to behave like JSON in some circumstances, and as plain text in others. One example is if one wanted to handle the LIKE operator for the JSON type. LIKE makes no sense against a JSON structure, but it does make sense against the underlying textual representation. To get at this with a type like JSONEncodedDict, we need to ****coerce**** the column to a textual form using [cast()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.cast" \o "sqlalchemy.sql.expression.cast) or [type\_coerce()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.type_coerce" \o "sqlalchemy.sql.expression.type_coerce) before attempting to use this operator:

TypeDecorator的默认行为是将任何表达式的“右侧”强制为同一类型。 对于像JSON这样的类型，这意味着任何使用的操作符都必须符合JSON。 在某些情况下，用户可能希望类型在某些情况下表现为JSON，而在其他情况下可能是纯文本。 一个例子是如果想要处理JSON类型的LIKE运算符。 LIKE对JSON结构没有任何意义，但它对底层的文本表示是有意义的。 为了得到类似JSONEncodedDict的类型，我们需要在尝试使用此运算符之前使用cast() 或type\_coerce() 将列强制为文本形式：

**from** **sqlalchemy** **import** type\_coerce, String

stmt = select([my\_table]).where(

type\_coerce(my\_table.c.json\_data, String).like('*%f*oo%'))

[TypeDecorator](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator" \o "sqlalchemy.types.TypeDecorator) provides a built-in system for working up type translations like these based on operators. If we wanted to frequently use the LIKE operator with our JSON object interpreted as a string, we can build it into the type by overriding the [TypeDecorator.coerce\_compared\_value()](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator.coerce_compared_value" \o "sqlalchemy.types.TypeDecorator.coerce_compared_value) method:

TypeDecorator提供了一个内置的系统，用于处理基于操作符的类似翻译。 如果我们想经常使用LIKE操作符将JSON对象解释为字符串，我们可以通过覆盖TypeDecorator.coerce\_compared\_value() 方法将其构建为类型：

**from** **sqlalchemy.sql** **import** operators

**from** **sqlalchemy** **import** String

**class** **JSONEncodedDict**(TypeDecorator):

impl = VARCHAR

**def** coerce\_compared\_value(self, op, value):

**if** op **in** (operators.like\_op, operators.notlike\_op):

**return** String()

**else**:

**return** self

**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

Above is just one approach to handling an operator like "LIKE". Other applications may wish to raise NotImplementedError for operators that have no meaning with a JSON object such as "LIKE", rather than automatically coercing to text.

以上只是处理像“LIKE”这样的操作符的一种方法。 其他应用程序可能希望为具有JSON对象（如“LIKE”）无意义的运算符引发NotImplementedError，而不是自动强制转换为文本。

4.2.4 Replacing the Bind/Result Processing of Existing Types

Most augmentation of type behavior at the bind/result level is achieved using [TypeDecorator](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator" \o "sqlalchemy.types.TypeDecorator). For the rare scenario where the specific processing applied by SQLAlchemy at the DBAPI level needs to be replaced, the SQLAlchemy type can be subclassed directly, and the bind\_processor() or result\_processor()methods can be overridden. Doing so requires that the adapt() method also be overridden. This method is the mechanism by which SQLAlchemy produces DBAPI-specific type behavior during statement execution. Overriding it allows a copy of the custom type to be used in lieu of a DBAPI-specific type. Below we subclass the [types.TIME](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.TIME" \o "sqlalchemy.types.TIME) type to have custom result processing behavior. The process() function will receive value from the DBAPI cursor directly:

绑定/结果级别的类型行为的大多数增加是使用TypeDecorator实现的。 对于SQLAlchemy在DBAPI级别应用的特定处理需要替换的罕见情况，SQLAlchemy类型可以直接进行子类化，bind\_processor() 或result\_processor() 方法可以被覆盖。 这样做需要使用adapt() 方法也可以被覆盖。 此方法是SQLAlchemy在语句执行期间生成特定于DBAPI的类型行为的机制。 覆盖它允许使用自定义类型的副本来代替DBAPI特定类型。 下面我们将types.TIME类型子类化为具有自定义结果处理行为。 process() 函数将直接从DBAPI游标接收值：

**class** **MySpecialTime**(TIME):

**def** \_\_init\_\_(self, special\_argument):

super(MySpecialTime, self).\_\_init\_\_()

self.special\_argument = special\_argument

**def** result\_processor(self, dialect, coltype):

**import** **datetime**

time = datetime.time

**def** process(value):

**if** value **is** **not** **None**:

microseconds = value.microseconds

seconds = value.seconds

minutes = seconds / 60

**return** time(

minutes / 60,

minutes % 60,

seconds - minutes \* 60,

microseconds)

**else**:

**return** **None**

**return** process

**def** adapt(self, impltype):

**return** MySpecialTime(self.special\_argument)

### 4.2.5 Applying SQL-level Bind/Result Processing

As seen in the sections [Augmenting Existing Types](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "types-typedecorator) and [Replacing the Bind/Result Processing of Existing Types](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "replacing-processors), SQLAlchemy allows Python functions to be invoked both when parameters are sent to a statement, as well as when result rows are loaded from the database, to apply transformations to the values as they are sent to or from the database. It is also possible to define SQL-level transformations as well. The rationale here is when only the relational database contains a particular series of functions that are necessary to coerce incoming and outgoing data between an application and persistence format. Examples include using database-defined encryption/decryption functions, as well as stored procedures that handle geographic data. The PostGIS extension to PostgreSQL includes an extensive array of SQL functions that are necessary for coercing data into particular formats.

从现有类型的增强现有类型和替换现有类型的绑定/结果处理的部分可以看出，SQLAlchemy允许在将参数发送到语句时以及从数据库加载结果行时调用Python函数来应用转换 到它们发送到数据库或从数据库发送的值。 也可以定义SQL级别转换。 这里的理由是只有关系数据库包含强制应用程序和持久性格式之间的传入和传出数据所必需的特定系列功能。 示例包括使用数据库定义的加密/解密功能以及处理地理数据的存储过程。 PostGIS的Postgre扩展包括了将数据强制转换为特定格式所需的大量SQL函数。

Any [TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine), [UserDefinedType](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.UserDefinedType" \o "sqlalchemy.types.UserDefinedType) or [TypeDecorator](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator" \o "sqlalchemy.types.TypeDecorator) subclass can include implementations of [TypeEngine.bind\_expression()](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.bind_expression" \o "sqlalchemy.types.TypeEngine.bind_expression) and/or [TypeEngine.column\_expression()](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.column_expression" \o "sqlalchemy.types.TypeEngine.column_expression), which when defined to return a non-None value should return a [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) expression to be injected into the SQL statement, either surrounding bound parameters or a column expression. For example, to build a Geometry type which will apply the PostGIS function ST\_GeomFromText to all outgoing values and the function ST\_AsText to all incoming data, we can create our own subclass of [UserDefinedType](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.UserDefinedType" \o "sqlalchemy.types.UserDefinedType) which provides these methods in conjunction with [func](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.func" \o "sqlalchemy.sql.expression.func):

TypeEngine，UserDefinedType或TypeDecorator子类可以包含TypeEngine.bind\_expression() 和/或TypeEngine.column\_expression() 的实现，当定义返回非None值时，应返回要注入到SQL语句中的ColumnElement表达式， 绑定参数或列表达式。 例如，要构建几何类型，将PostGIS函数ST\_GeomFromText应用于所有输出值，并将函数ST\_AsText应用于所有传入数据，我们可以创建自己的UserDefinedType子类，它们与func一起提供这些方法：

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

**from** **sqlalchemy.types** **import** UserDefinedType

**class** **Geometry**(UserDefinedType):

**def** get\_col\_spec(self):

**return** "GEOMETRY"

**def** bind\_expression(self, bindvalue):

**return** func.ST\_GeomFromText(bindvalue, type\_=self)

**def** column\_expression(self, col):

**return** func.ST\_AsText(col, type\_=self)

We can apply the Geometry type into [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) metadata and use it in a [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) construct:

我们可以将几何类型应用于表元数据，并在select() 结构中使用它：

geometry = Table('geometry', metadata,

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

Column('geom\_data', Geometry)

)

print(select([geometry]).where(

geometry.c.geom\_data == 'LINESTRING(189412 252431,189631 259122)'))

The resulting SQL embeds both functions as appropriate. ST\_AsText is applied to the columns clause so that the return value is run through the function before passing into a result set, and ST\_GeomFromText is run on the bound parameter so that the passed-in value is converted:

生成的SQL根据需要嵌入这两个函数。 ST\_AsText应用于columns子句，以便返回值在传入结果集之前运行，ST\_GeomFromText在bound参数上运行，以便转换传入值：

SELECT geometry.geom\_id, ST\_AsText(geometry.geom\_data) AS geom\_data\_1FROM geometryWHERE geometry.geom\_data = ST\_GeomFromText(:geom\_data\_2)

The [TypeEngine.column\_expression()](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.column_expression" \o "sqlalchemy.types.TypeEngine.column_expression) method interacts with the mechanics of the compiler such that the SQL expression does not interfere with the labeling of the wrapped expression. Such as, if we rendered a [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) against a [label()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.label" \o "sqlalchemy.sql.expression.label) of our expression, the string label is moved to the outside of the wrapped expression:

TypeEngine.column\_expression() 方法与编译器的机制进行交互，使得SQL表达式不会干扰包装表达式的标签。 例如，如果我们针对我们的表达式的label() 渲染了一个select() ，则字符串标签将被移动到包装表达式的外部：

print(select([geometry.c.geom\_data.label('my\_data')]))

Output:

SELECT ST\_AsText(geometry.geom\_data) AS my\_dataFROM geometry

For an example of subclassing a built in type directly, we subclass [postgresql.BYTEA](http://docs.sqlalchemy.org/en/rel_1_1/dialects/postgresql.html" \l "sqlalchemy.dialects.postgresql.BYTEA" \o "sqlalchemy.dialects.postgresql.BYTEA) to provide a PGPString, which will make use of the PostgreSQL pgcryptoextension to encrypt/decrypt values transparently:

对于直接对内置类进行子类化的示例，我们将postgresql.BYTEA子类化以提供PGPString，它将利用PostgreSQL pgcryptoextension透明地加密/解密值：

**from** **sqlalchemy** **import** create\_engine, String, select, func, \

MetaData, Table, Column, type\_coerce

**from** **sqlalchemy.dialects.postgresql** **import** BYTEA

**class** **PGPString**(BYTEA):

**def** \_\_init\_\_(self, passphrase, length=**None**):

super(PGPString, self).\_\_init\_\_(length)

self.passphrase = passphrase

**def** bind\_expression(self, bindvalue):

*# convert the bind's type from PGPString to*

*# String, so that it's passed to psycopg2 as is without*

*# a dbapi.Binary wrapper*

bindvalue = type\_coerce(bindvalue, String)

**return** func.pgp\_sym\_encrypt(bindvalue, self.passphrase)

**def** column\_expression(self, col):

**return** func.pgp\_sym\_decrypt(col, self.passphrase)

metadata = MetaData()message = Table('message', metadata,

Column('username', String(50)),

Column('message',

PGPString("this is my passphrase", length=1000)),

)

engine = create\_engine("postgresql://scott:tiger@localhost/test", echo=**True**)**with** engine.begin() **as** conn:

metadata.create\_all(conn)

conn.execute(message.insert(), username="some user",

message="this is my message")

print(conn.scalar(

select([message.c.message]).\

where(message.c.username == "some user")

))

The pgp\_sym\_encrypt and pgp\_sym\_decrypt functions are applied to the INSERT and SELECT statements:

INSERT INTO message (username, message)

VALUES (%(username)s, pgp\_sym\_encrypt(%(message)s, %(pgp\_sym\_encrypt\_1)s))

{'username': 'some user', 'message': 'this is my message',

'pgp\_sym\_encrypt\_1': 'this is my passphrase'}

SELECT pgp\_sym\_decrypt(message.message, %(pgp\_sym\_decrypt\_1)s) AS message\_1

FROM message

WHERE message.username = %(username\_1)s

{'pgp\_sym\_decrypt\_1': 'this is my passphrase', 'username\_1': 'some user'}

*New in version 0.8:*Added the [TypeEngine.bind\_expression()](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.bind_expression" \o "sqlalchemy.types.TypeEngine.bind_expression) and [TypeEngine.column\_expression()](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.column_expression" \o "sqlalchemy.types.TypeEngine.column_expression) methods.

See also:

[PostGIS Integration](http://docs.sqlalchemy.org/en/rel_1_1/orm/examples.html" \l "examples-postgis)

4.2.6 Redefining and Creating New Operators

SQLAlchemy Core defines a fixed set of expression operators available to all column expressions. Some of these operations have the effect of overloading Python's built in operators; examples of such operators include [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__) (table.c.somecolumn == 'foo'),[ColumnOperators.\_\_invert\_\_()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators.__invert__" \o "sqlalchemy.sql.operators.ColumnOperators.__invert__) (~table.c.flag), and [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__) (table.c.x + table.c.y). Other operators are exposed as explicit methods on column expressions, such as [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_) (table.c.value.in\_(['x', 'y'])) and [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)(table.c.value.like('%ed%')).

SQLAlchemy Core定义了一组可用于所有列表达式的表达式运算符。 其中一些操作会影响Python内建的操作符的重载; 这些运算符的例子包括ColumnOperators .\_\_ eq \_\_() （table.c.somecolumn =='foo'），ColumnOperators .\_\_ invert \_\_() （〜table.c.flag）和ColumnOperators .\_\_ add \_\_() （table.cx + table。CY）。 其他操作符作为列表达式的显式方法暴露，如ColumnOperators.in\_() （table.c.value.in \_（['x'，'y']））和ColumnOperators.like() （table.c.value。像（ '％ED％'））。

The Core expression constructs in all cases consult the type of the expression in order to determine the behavior of existing operators, as well as to locate additional operators that aren't part of the built in set. The [TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine) base class defines a root "comparison" implementation [TypeEngine.Comparator](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.Comparator" \o "sqlalchemy.types.TypeEngine.Comparator), and many specific types provide their own sub-implementations of this class. User-defined [TypeEngine.Comparator](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.Comparator" \o "sqlalchemy.types.TypeEngine.Comparator) implementations can be built directly into a simple subclass of a particular type in order to override or define new operations. Below, we create a [Integer](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Integer" \o "sqlalchemy.types.Integer) subclass which overrides the [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__) operator:

在所有情况下，Core表达式构造都会查看表达式的类型，以确定现有运算符的行为，以及查找不是内置集的一部分的其他运算符。 TypeEngine基类定义了一个根“比较”实现TypeEngine.Comparator，并且许多特定类型提供了它们自己的这个类的子实现。 用户定义的TypeEngine.Comparator实现可以直接构建到特定类型的简单子类中，以覆盖或定义新的操作。 下面，我们创建一个覆盖ColumnOperators的整数子类.\_\_ add \_\_() 运算符：

**from** **sqlalchemy** **import** Integer

**class** **MyInt**(Integer):

**class** **comparator\_factory**(Integer.Comparator):

**def** \_\_add\_\_(self, other):

**return** self.op("goofy")(other)

The above configuration creates a new class MyInt, which establishes 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 as referring to a new class, subclassing the [TypeEngine.Comparator](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.Comparator" \o "sqlalchemy.types.TypeEngine.Comparator) class associated with the [Integer](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Integer" \o "sqlalchemy.types.Integer) type.

Usage:

上述配置创建一个新类MyInt，它将TypeEngine.comparator\_factory属性建立为引用一个新类，对与Integer类型相关联的TypeEngine.Comparator类进行子类化。

用法：

**>>>** sometable = Table("sometable", metadata, Column("data", MyInt))

**>>>** print(sometable.c.data + 5)sometable.data goofy :data\_1

The implementation for [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__) is consulted by an owning SQL expression, by instantiating the [TypeEngine.Comparator](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.Comparator" \o "sqlalchemy.types.TypeEngine.Comparator) with itself as the expr attribute. The mechanics of the expression system are such that operations continue recursively until an expression object produces a new SQL expression construct. Above, we could just as well have said self.expr.op("goofy")(other) instead of self.op("goofy")(other).

ColumnOperators.\_\_ add \_\_() 的实现由拥有的SQL表达式引用，通过将TypeEngine.Comparator与其自身实例化为expr属性进行实例化。 表达式系统的机制使得操作继续递归，直到表达式对象生成新的SQL表达式构造。 以上，我们也可以说self.expr.op（“goofy”）（其他）而不是self.op（“goofy”）（其他）。

New methods added to a [TypeEngine.Comparator](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.Comparator" \o "sqlalchemy.types.TypeEngine.Comparator) are exposed on an owning SQL expression using a \_\_getattr\_\_ scheme, which exposes methods added to[TypeEngine.Comparator](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.Comparator" \o "sqlalchemy.types.TypeEngine.Comparator) onto the owning [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement). For example, to add a log() function to integers:

添加到TypeEngine.Comparator的新方法使用\_\_getattr\_\_方案在拥有的SQL表达式上公开，该方案将添加到TypeEngine.Comparator的方法暴露给拥有的ColumnElement。 例如，要为整数添加一个log() 函数：

**from** **sqlalchemy** **import** Integer, func

**class** **MyInt**(Integer):

**class** **comparator\_factory**(Integer.Comparator):

**def** log(self, other):

**return** func.log(self.expr, other)

Using the above type:

**>>>** print(sometable.c.data.log(5))log(:log\_1, :log\_2)

Unary operations are also possible. For example, to add an implementation of the PostgreSQL factorial operator, we combine the [UnaryExpression](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.UnaryExpression" \o "sqlalchemy.sql.expression.UnaryExpression) construct along with a [custom\_op](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.custom_op" \o "sqlalchemy.sql.operators.custom_op) to produce the factorial expression:

一致的操作也是可能的。 例如，要添加PostgreSQL因子运算符的实现，我们将UnaryExpression构造与custom\_op相结合，以生成阶乘表达式：

**from** **sqlalchemy** **import** Integer

**from** **sqlalchemy.sql.expression** **import** UnaryExpression

**from** **sqlalchemy.sql** **import** operators

**class** **MyInteger**(Integer):

**class** **comparator\_factory**(Integer.Comparator):

**def** factorial(self):

**return** UnaryExpression(self.expr,

modifier=operators.custom\_op("!"),

type\_=MyInteger)

Using the above type:

**>>> from** **sqlalchemy.sql** **import** column

**>>>** print(column('x', MyInteger).factorial())x !

See also:

[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)

*New in version 0.8:*The expression system was enhanced to support customization of operators on a per-type level.

TypeEngine.comparator\_factory

版本0.8中的新功能：增强了表达式系统，以支持对每个类型的操作员进行定制。

4.2.7 Creating New Types

The [UserDefinedType](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.UserDefinedType" \o "sqlalchemy.types.UserDefinedType) class is provided as a simple base class for defining entirely new database types. Use this to represent native database types not known by SQLAlchemy. If only Python translation behavior is needed, use [TypeDecorator](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator" \o "sqlalchemy.types.TypeDecorator) instead.

UserDefinedType类作为简单的基类提供，用于定义全新的数据库类型。 使用它来表示SQLAlchemy不知道的本机数据库类型。 如果只需要Python翻译行为，请改用TypeDecorator。

*class*sqlalchemy.types.**UserDefinedType**

Bases: [sqlalchemy.types.TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine)

Base for user defined types.

基于用户定义的类型。

This should be the base of new types. Note that for most cases, [TypeDecorator](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator" \o "sqlalchemy.types.TypeDecorator) is probably more appropriate:

这应该是新类型的基础。 请注意，在大多数情况下，TypeDecorator可能更合适：

**import** **sqlalchemy.types** **as** **types**

**class** **MyType**(types.UserDefinedType):

**def** \_\_init\_\_(self, precision = 8):

self.precision = precision

**def** get\_col\_spec(self, \*\*kw):

**return** "MYTYPE(*%s*)" % self.precision

**def** bind\_processor(self, dialect):

**def** process(value):

**return** value

**return** process

**def** result\_processor(self, dialect, coltype):

**def** process(value):

**return** value

**return** process

Once the type is made, it's immediately usable:

一旦制作完成，它立即可用：

table = Table('foo', meta,

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

Column('data', MyType(16))

)

The get\_col\_spec() method will in most cases receive a keyword argument type\_expression which refers to the owning expression of the type as being compiled, such as a [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) or [cast()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.cast" \o "sqlalchemy.sql.expression.cast) construct. This keyword is only sent if the method accepts keyword arguments (e.g. \*\*kw) in its argument signature; introspection is used to check for this in order to support legacy forms of this function.

get\_col\_spec() 方法在大多数情况下会接收到一个关键字参数type\_expression，它将类型的所有表达式引用为已编译，例如Column或cast() 构造。 仅当该方法在其参数签名中接受关键字参数（例如\*\* kw）时才发送此关键字; 为了支持此功能的遗留形式，使用内省来检查这一点。

*New in version 1.0.0:*the owning expression is passed to the get\_col\_spec() method via the keyword argument type\_expression, if it receives \*\*kw in its signature.

**coerce\_compared\_value**(*op*, *value*)

Suggest a type for a 'coerced' Python value in an expression.

在表达式中建议一个“强制”Python值的类型。

Default behavior for [UserDefinedType](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.UserDefinedType" \o "sqlalchemy.types.UserDefinedType) is the same as that of [TypeDecorator](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator" \o "sqlalchemy.types.TypeDecorator); by default it returns self, assuming the compared value should be coerced into the same type as this one. See [TypeDecorator.coerce\_compared\_value()](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator.coerce_compared_value" \o "sqlalchemy.types.TypeDecorator.coerce_compared_value) for more detail.

UserDefinedType的默认行为与TypeDecorator的默认行为相同; 默认情况下，它返回自身，假设比较的值应该被强制为与此类型相同的类型。 有关详细信息，请参阅TypeDecorator.coerce\_compared\_value() 。

*Changed in version 0.8:*[UserDefinedType.coerce\_compared\_value()](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.UserDefinedType.coerce_compared_value" \o "sqlalchemy.types.UserDefinedType.coerce_compared_value) now returns self by default, rather than falling onto the more fundamental behavior of [TypeEngine.coerce\_compared\_value()](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.coerce_compared_value" \o "sqlalchemy.types.TypeEngine.coerce_compared_value).

在版本0.8中更改：UserDefinedType.coerce\_compared\_value() 默认返回自身，而不是落入TypeEngine.coerce\_compared\_value() 的更基本的行为。

# **4.3 Base Type API**

*class*sqlalchemy.types.**TypeEngine**

Bases: sqlalchemy.sql.visitors.Visitable

The ultimate base class for all SQL datatypes.

所有SQL数据类型的最终基类。

Common subclasses of [TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine) include [String](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.String" \o "sqlalchemy.types.String), [Integer](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Integer" \o "sqlalchemy.types.Integer), and [Boolean](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Boolean" \o "sqlalchemy.types.Boolean).

For an overview of the SQLAlchemy typing system, see [Column and Data Types](http://docs.sqlalchemy.org/en/rel_1_1/core/types.html).

TypeEngine的公共子类包括String，Integer和Boolean。

有关SQLAlchemy类型系统的概述，请参阅列和数据类型。

**See also**

[Column and Data Types](http://docs.sqlalchemy.org/en/rel_1_1/core/types.html)

*class***Comparator**(*expr*)

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)

Base class for custom comparison operations defined at the type level. See [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).

在类型级别定义的自定义比较操作的基类。见TypeEngine.comparator\_factory。

**adapt**(*cls*, *\*\*kw*)

Produce an "adapted" form of this type, given an "impl" class to work with.

产生一种“适应”的这种类型的形式，给予一个“impl”类与之配合。

This method is used internally to associate generic types with "implementation" types that are specific to a particular dialect.

内部使用此方法将通用类型与特定于特定方言的“实现”类型相关联。

**bind\_expression**(*bindvalue*)

"Given a bind value (i.e. a [BindParameter](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.BindParameter" \o "sqlalchemy.sql.expression.BindParameter) instance), return a SQL expression in its place.

“给定绑定值（即BindParameter实例），返回一个SQL表达式。

This is typically a SQL function that wraps the existing bound parameter within the statement. It is used for special data types that require literals being wrapped in some special database function in order to coerce an application-level value into a database-specific format. It is the SQL analogue of the[TypeEngine.bind\_processor()](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.bind_processor" \o "sqlalchemy.types.TypeEngine.bind_processor) method.

这通常是一个SQL函数，它将现有的绑定参数包含在语句中。 它用于需要将文字包装在某些特殊数据库函数中的特殊数据类型，以将应用程序级值强制为特定于数据库的格式。 它是TypeEngine.bind\_processor() 方法的SQL模拟。

The method is evaluated at statement compile time, as opposed to statement construction time.

该方法在语句编译时进行评估，而不是语句构建时间。

Note that this method, when implemented, should always return the exact same structure, without any conditional logic, as it may be used in an executemany() call against an arbitrary number of bound parameter sets.

请注意，该方法在实现时应始终返回完全相同的结构，而不需要任何条件逻辑，因为它可以用于针对任意数量的绑定参数集的executemany() 调用。

See also:

[Applying SQL-level Bind/Result Processing](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "types-sql-value-processing)

**bind\_processor**(*dialect*)

Return a conversion function for processing bind values.

返回一个用于处理绑定值的转换函数。

Returns a callable which will receive a bind parameter value as the sole positional argument and will return a value to send to the DB-API.

返回一个可以接收绑定参数值作为唯一位置参数的可调用值，并返回一个值以发送给DB-API。

If processing is not necessary, the method should return None.

如果不需要处理，则该方法应返回None。

|  |  |
| --- | --- |
| **Parameters:** | ****dialect**** – Dialect instance in use. |

**coerce\_compared\_value**(*op*, *value*)

Suggest a type for a 'coerced' Python value in an expression.

在表达式中建议一个“强制”Python值的类型。

Given an operator and value, gives the type a chance to return a type which the value should be coerced into.

给定一个运算符和值，给类型一个机会返回一个值应被强制转换的类型。

The default behavior here is conservative; if the right-hand side is already coerced into a SQL type based on its Python type, it is usually left alone.

这里的默认行为是保守的; 如果右侧已经被强制为基于其Python类型的SQL类型，那么它通常是单独的。

End-user functionality extension here should generally be via [TypeDecorator](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator" \o "sqlalchemy.types.TypeDecorator), which provides more liberal behavior in that it defaults to coercing the other side of the expression into this type, thus applying special Python conversions above and beyond those needed by the DBAPI to both ides. It also provides the public method [TypeDecorator.coerce\_compared\_value()](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator.coerce_compared_value" \o "sqlalchemy.types.TypeDecorator.coerce_compared_value) which is intended for end-user customization of this behavior.

这里的最终用户功能扩展通常应该是通过TypeDecorator，它提供了更自由的行为，因为它默认将强制表达式的另一面加入到这种类型中，从而将特殊的Python转换应用到DBAPI所需的以外。 它还提供了公共方法TypeDecorator.coerce\_compared\_value() ，该方法用于最终用户自定义此行为。

**column\_expression**(*colexpr*)

Given a SELECT column expression, return a wrapping SQL expression.

给定一个SELECT列表达式，返回一个包装SQL表达式。

This is typically a SQL function that wraps a column expression as rendered in the columns clause of a SELECT statement. It is used for special data types that require columns to be wrapped in some special database function in order to coerce the value before being sent back to the application. It is the SQL analogue of the [TypeEngine.result\_processor()](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.result_processor" \o "sqlalchemy.types.TypeEngine.result_processor) method.

这通常是一个SQL函数，它将列表达式包含在SELECT语句的columns子句中。 它用于特殊数据类型，需要将列包装在某些特殊数据库函数中，以便在发送回应用程序之前强制该值。 它是TypeEngine.result\_processor() 方法的SQL模拟。

The method is evaluated at statement compile time, as opposed to statement construction time.

该方法在语句编译时进行评估，而不是语句构建时间。

See also:

[Applying SQL-level Bind/Result Processing](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "types-sql-value-processing)

**comparator\_factory**

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)

A [TypeEngine.Comparator](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.Comparator" \o "sqlalchemy.types.TypeEngine.Comparator) class which will apply to operations performed by owning [ColumnElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ColumnElement" \o "sqlalchemy.sql.expression.ColumnElement) objects.

The [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 is a hook consulted by the core expression system when column and SQL expression operations are performed. When a [TypeEngine.Comparator](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.Comparator" \o "sqlalchemy.types.TypeEngine.Comparator) class is associated with this attribute, it allows custom re-definition of all existing operators, as well as definition of new operators. Existing operators include those provided by Python operator overloading such as [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__) and[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__), those provided as standard attributes of [operators.ColumnOperators](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.operators.ColumnOperators" \o "sqlalchemy.sql.operators.ColumnOperators) such as[operators.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) and [operators.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_).

Rudimentary usage of this hook is allowed through simple subclassing of existing types, or alternatively by using [TypeDecorator](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator" \o "sqlalchemy.types.TypeDecorator). See the documentation section [Redefining and Creating New Operators](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "types-operators) for examples.

*New in version 0.8:*The expression system was enhanced to support customization of operators on a per-type level.

alias of [Comparator](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.Comparator" \o "sqlalchemy.types.TypeEngine.Comparator)

**compare\_against\_backend**(*dialect*, *conn\_type*)

Compare this type against the given backend type.

将此类型与给定的后端类型进行比较。

This function is currently not implemented for SQLAlchemy types, and for all built in types will return None. However, it can be implemented by a user-defined type where it can be consumed by schema comparison tools such as Alembic autogenerate.

这个函数目前没有为SQLAlchemy类型实现，对于所有内置的类型都将返回None。 但是，它可以通过用户定义的类型实现，可以通过模式比较工具（如Alembic autogenerate）使用它。

A future release of SQLAlchemy will potentially impement this method for builtin types as well.

未来的SQLAlchemy版本也可能会阻碍这种内置类型的方法。

The function should return True if this type is equivalent to the given type; the type is typically reflected from the database so should be database specific. The dialect in use is also passed. It can also return False to assert that the type is not equivalent.

如果此类型与给定类型相同，则该函数应返回True; 该类型通常反映在数据库中，因此应该是数据库特定的。 使用的方言也通过了。 它也可以返回False来声明该类型不相同。

|  |  |
| --- | --- |
| **Parameters:** | * ****dialect**** – a [Dialect](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Dialect" \o "sqlalchemy.engine.interfaces.Dialect) that is involved in the comparison. * ****conn\_type**** – the type object reflected from the backend. |

*New in version 1.0.3.*

**compare\_values**(*x*, *y*)

Compare two values for equality.

**compile**(*dialect=None*)

Produce a string-compiled form of this [TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine).

When called with no arguments, uses a "default" dialect to produce a string result.

生成此TypeEngine的字符串编译形式。

当没有参数调用时，使用“默认”方言产生一个字符串结果。

|  |  |
| --- | --- |
| **Parameters:** | ****dialect**** – a [Dialect](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Dialect" \o "sqlalchemy.engine.interfaces.Dialect) instance. |

**dialect\_impl**(*dialect*)

Return a dialect-specific implementation for this [TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine).

返回此TypeEngine的特定于方言的实现。

**evaluates\_none**()

Return a copy of this type which has the [should\_evaluate\_none](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.should_evaluate_none" \o "sqlalchemy.types.TypeEngine.should_evaluate_none) flag set to True.

E.g.:

返回具有设置为True的should\_evaluate\_none标志的这种类型的副本。

例如。：

Table(

'some\_table', metadata,

Column(

String(50).evaluates\_none(),

nullable=**True**,

server\_default='no value'))

The ORM uses this flag to indicate that a positive value of None is passed to the column in an INSERT statement, rather than omitting the column from the INSERT statement which has the effect of firing off column-level defaults. It also allows for types which have special behavior associated with the Python None value to indicate that the value doesn't necessarily translate into SQL NULL; a prime example of this is a JSON type which may wish to persist the JSON value 'null'.

ORM使用这个标志来指示在INSERT语句中将None的正值传递给列，而不是从INSERT语句中省略具有触发列级别默认值的列。 它还允许具有与Python None值关联的特殊行为的类型指示该值不一定会转换为SQL NULL; 一个最好的例子就是JSON类型，它可能希望保持JSON值为null。

In all cases, the actual NULL SQL value can be always be persisted in any column by 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 in an INSERT statement or associated with an ORM-mapped attribute.

在所有情况下，通过在INSERT语句中使用NULL SQL构造或与ORM映射属性关联，实际的NULL SQL值可以始终保留在任何列中。

**Note**

The "evaulates none" flag does ****not**** apply to a value of None passed to [Column.default](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column.params.default" \o "sqlalchemy.schema.Column) or [Column.server\_default](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column.params.server_default" \o "sqlalchemy.schema.Column); in these cases, None still means "no default".

*New in version 1.1.*

**See also**

[Forcing NULL on a column with a default](http://docs.sqlalchemy.org/en/rel_1_1/orm/persistence_techniques.html" \l "session-forcing-null) - in the ORM documentation

[postgresql.JSON.none\_as\_null](http://docs.sqlalchemy.org/en/rel_1_1/dialects/postgresql.html" \l "sqlalchemy.dialects.postgresql.JSON.params.none_as_null" \o "sqlalchemy.dialects.postgresql.JSON) - PostgreSQL JSON interaction with this flag.

[TypeEngine.should\_evaluate\_none](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.should_evaluate_none" \o "sqlalchemy.types.TypeEngine.should_evaluate_none) - class-level flag

**get\_dbapi\_type**(*dbapi*)

Return the corresponding type object from the underlying DB-API, if any.

This can be useful for calling setinputsizes(), for example.

**hashable***= True*

Flag, if False, means values from this type aren't hashable.

Used by the ORM when uniquing result lists.

**literal\_processor**(*dialect*)

Return a conversion function for processing literal values that are to be rendered directly without using binds.

This function is used when the compiler makes use of the "literal\_binds" flag, typically used in DDL generation as well as in certain scenarios where backends don't accept bound parameters.

*New in version 0.9.0.*

**python\_type**

Return the Python type object expected to be returned by instances of this type, if known.

Basically, for those types which enforce a return type, or are known across the board to do such for all common DBAPIs (like int for example), will return that type.

If a return type is not defined, raises NotImplementedError.

Note that any type also accommodates NULL in SQL which means you can also get back None from any type in practice.

**result\_processor**(*dialect*, *coltype*)

Return a conversion function for processing result row values.

Returns a callable which will receive a result row column value as the sole positional argument and will return a value to return to the user.

If processing is not necessary, the method should return None.

|  |  |
| --- | --- |
| **Parameters:** | * ****dialect**** – Dialect instance in use. * ****coltype**** – DBAPI coltype argument received in cursor.description. |

**should\_evaluate\_none***= False*

If True, the Python constant None is considered to be handled explicitly by this type.

The ORM uses this flag to indicate that a positive value of None is passed to the column in an INSERT statement, rather than omitting the column from the INSERT statement which has the effect of firing off column-level defaults. It also allows types which have special behavior for Python None, such as a JSON type, to indicate that they'd like to handle the None value explicitly.

To set this flag on an existing type, use 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.

**See also**

[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)

*New in version 1.1.*

**with\_variant**(*type\_*, *dialect\_name*)

Produce a new type object that will utilize the given type when applied to the dialect of the given name.

e.g.:

**from** **sqlalchemy.types** **import** String

**from** **sqlalchemy.dialects** **import** mysql

s = String()

s = s.with\_variant(mysql.VARCHAR(collation='foo'), 'mysql')

The construction of [TypeEngine.with\_variant()](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.with_variant" \o "sqlalchemy.types.TypeEngine.with_variant) is always from the "fallback" type to that which is dialect specific. The returned type is an instance of [Variant](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.Variant" \o "sqlalchemy.types.Variant), which itself provides a Variant.with\_variant() that can be called repeatedly.

|  |  |
| --- | --- |
| **Parameters:** | * ****type\_**** – a [TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine) that will be selected as a variant from the originating type, when a dialect of the given name is in use. * ****dialect\_name**** – base name of the dialect which uses this type. (i.e. 'postgresql', 'mysql', etc.) |

*New in version 0.7.2.*

*class*sqlalchemy.types.**Concatenable**

A mixin that marks a type as supporting 'concatenation', typically strings.

*class*sqlalchemy.types.**Indexable**

A mixin that marks a type as supporting indexing operations, such as array or JSON structures.

*New in version 1.1.0.*

*class*sqlalchemy.types.**NullType**

Bases: [sqlalchemy.types.TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine)

An unknown type.

[NullType](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.NullType" \o "sqlalchemy.types.NullType) is used as a default type for those cases where a type cannot be determined, including:

* During table reflection, when the type of a column is not recognized by the [Dialect](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Dialect" \o "sqlalchemy.engine.interfaces.Dialect)
* When constructing SQL expressions using plain Python objects of unknown types (e.g. somecolumn == my\_special\_object)
* When a new [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) is created, and the given type is passed as None or is not passed at all.

The [NullType](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.NullType" \o "sqlalchemy.types.NullType) can be used within SQL expression invocation without issue, it just has no behavior either at the expression construction level or at the bind-parameter/result processing level. [NullType](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.NullType" \o "sqlalchemy.types.NullType) will result in a [CompileError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.CompileError" \o "sqlalchemy.exc.CompileError) if the compiler is asked to render the type itself, such as if it is used in a [cast()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.cast" \o "sqlalchemy.sql.expression.cast)operation or within a schema creation operation such as that invoked by [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) or the [CreateTable](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.CreateTable" \o "sqlalchemy.schema.CreateTable) construct.

*class*sqlalchemy.types.**Variant**(*base*, *mapping*)

Bases: [sqlalchemy.types.TypeDecorator](http://docs.sqlalchemy.org/en/rel_1_1/core/custom_types.html" \l "sqlalchemy.types.TypeDecorator" \o "sqlalchemy.types.TypeDecorator)

A wrapping type that selects among a variety of implementations based on dialect in use.

The [Variant](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.Variant" \o "sqlalchemy.types.Variant) type is typically constructed using the [TypeEngine.with\_variant()](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.with_variant" \o "sqlalchemy.types.TypeEngine.with_variant) method.

*New in version 0.7.2.*

**See also**

[TypeEngine.with\_variant()](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine.with_variant" \o "sqlalchemy.types.TypeEngine.with_variant) for an example of use.

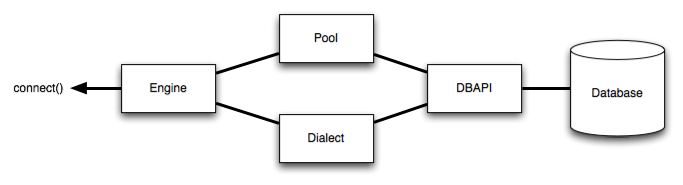
|  |  |
| --- | --- |
| **Members:** | with\_variant, \_\_init\_\_ |

# Chapter 5 Engine and Connection Use

## 5.1 Engine Configuration

The [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) is the starting point for any SQLAlchemy application. It's "home base" for the actual database and its [DBAPI](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-dbapi), delivered to the SQLAlchemy application through a connection pool and a [Dialect](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Dialect" \o "sqlalchemy.engine.interfaces.Dialect), which describes how to talk to a specific kind of database/DBAPI combination.

The general structure can be illustrated as follows:



Where above, an [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) references both a [Dialect](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Dialect" \o "sqlalchemy.engine.interfaces.Dialect) and a [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool), which together interpret the DBAPI's module functions as well as the behavior of the database.

Creating an engine is just a matter of issuing a single call, [create\_engine()](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "sqlalchemy.create_engine" \o "sqlalchemy.create_engine):

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

engine = create\_engine('postgresql://scott:tiger@localhost:5432/mydatabase')

The above engine creates a [Dialect](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Dialect" \o "sqlalchemy.engine.interfaces.Dialect) object tailored towards PostgreSQL, as well as a [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool) object which will establish a DBAPI connection at localhost:5432when a connection request is first received. Note that the [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) and its underlying [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool) do ****not**** establish the first actual DBAPI connection until the [Engine.connect()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.connect" \o "sqlalchemy.engine.Engine.connect) method is called, or an operation which is dependent on this method such as [Engine.execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.execute" \o "sqlalchemy.engine.Engine.execute) is invoked. In this way, [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) and [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool)can be said to have a *lazy initialization* behavior.

The [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine), once created, can either be used directly to interact with the database, or can be passed 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) object to work with the ORM. This section covers the details of configuring an [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine). The next section, [Working with Engines and Connections](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html), will detail the usage API of the [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) and similar, typically for non-ORM applications.

5.1.1 Supported Databases

SQLAlchemy includes many [Dialect](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Dialect" \o "sqlalchemy.engine.interfaces.Dialect) implementations for various backends. Dialects for the most common databases are included with SQLAlchemy; a handful of others require an additional install of a separate dialect.

See the section [Dialects](http://docs.sqlalchemy.org/en/rel_1_1/dialects/index.html) for information on the various backends available.

5.1.2 Database Urls

The [create\_engine()](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "sqlalchemy.create_engine" \o "sqlalchemy.create_engine) function produces an [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) object based on a URL. These URLs follow [RFC-1738](http://rfc.net/rfc1738.html), and usually can include username, password, hostname, database name as well as optional keyword arguments for additional configuration. In some cases a file path is accepted, and in others a "data source name" replaces the "host" and "database" portions. The typical form of a database URL is:

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

Dialect names include the identifying name of the SQLAlchemy dialect, a name such as sqlite, mysql, postgresql, oracle, or mssql. The drivername is the name of the DBAPI to be used to connect to the database using all lowercase letters. If not specified, a "default" DBAPI will be imported if available - this default is typically the most widely known driver available for that backend.

Examples for common connection styles follow below. For a full index of detailed information on all included dialects as well as links to third-party dialects, see [Dialects](http://docs.sqlalchemy.org/en/rel_1_1/dialects/index.html).

### PostgreSQL

The PostgreSQL dialect uses psycopg2 as the default DBAPI. pg8000 is also available as a pure-Python substitute:

*# default*engine = create\_engine('postgresql://scott:tiger@localhost/mydatabase')

*# psycopg2*engine = create\_engine('postgresql+psycopg2://scott:tiger@localhost/mydatabase')

*# pg8000*engine = create\_engine('postgresql+pg8000://scott:tiger@localhost/mydatabase')

More notes on connecting to PostgreSQL at [PostgreSQL](http://docs.sqlalchemy.org/en/rel_1_1/dialects/postgresql.html).

### MySQL

The MySQL dialect uses mysql-python as the default DBAPI. There are many MySQL DBAPIs available, including MySQL-connector-python and OurSQL:

*# default*engine = create\_engine('mysql://scott:tiger@localhost/foo')

*# mysql-python*engine = create\_engine('mysql+mysqldb://scott:tiger@localhost/foo')

*# MySQL-connector-python*engine = create\_engine('mysql+mysqlconnector://scott:tiger@localhost/foo')

*# OurSQL*engine = create\_engine('mysql+oursql://scott:tiger@localhost/foo')

More notes on connecting to MySQL at [MySQL](http://docs.sqlalchemy.org/en/rel_1_1/dialects/mysql.html).

### Oracle

The Oracle dialect uses cx\_oracle as the default DBAPI:

engine = create\_engine('oracle://scott:tiger@127.0.0.1:1521/sidname')

engine = create\_engine('oracle+cx\_oracle://scott:tiger@tnsname')

More notes on connecting to Oracle at [Oracle](http://docs.sqlalchemy.org/en/rel_1_1/dialects/oracle.html).

### Microsoft SQL Server

The SQL Server dialect uses pyodbc as the default DBAPI. pymssql is also available:

*# pyodbc*engine = create\_engine('mssql+pyodbc://scott:tiger@mydsn')

*# pymssql*engine = create\_engine('mssql+pymssql://scott:tiger@hostname:port/dbname')

More notes on connecting to SQL Server at [Microsoft SQL Server](http://docs.sqlalchemy.org/en/rel_1_1/dialects/mssql.html).

### SQLite

SQLite connects to file-based databases, using the Python built-in module sqlite3 by default.

As SQLite connects to local files, the URL format is slightly different. The "file" portion of the URL is the filename of the database. For a relative file path, this requires three slashes:

*# sqlite://<nohostname>/<path># where <path> is relative:*engine = create\_engine('sqlite:///foo.db')

And for an absolute file path, the three slashes are followed by the absolute path:

*#Unix/Mac - 4 initial slashes in total*engine = create\_engine('sqlite:////absolute/path/to/foo.db')*#Windows*engine = create\_engine('sqlite:///C:**\\**path**\\**to**\\**foo.db')*#Windows alternative using raw string*engine = create\_engine(r'sqlite:///C:\path\to\foo.db')

To use a SQLite :memory: database, specify an empty URL:

engine = create\_engine('sqlite://')

More notes on connecting to SQLite at [SQLite](http://docs.sqlalchemy.org/en/rel_1_1/dialects/sqlite.html).

### Others

See [Dialects](http://docs.sqlalchemy.org/en/rel_1_1/dialects/index.html), the top-level page for all additional dialect documentation.

5.1.3 Engine Creation API

sqlalchemy.**create\_engine**(*\*args*, *\*\*kwargs*)

Create a new [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) instance.

The standard calling form is to send the URL as the first positional argument, usually a string that indicates database dialect and connection arguments:

engine = create\_engine("postgresql://scott:tiger@localhost/test")

Additional keyword arguments may then follow it which establish various options on the resulting [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) and its underlying [Dialect](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Dialect" \o "sqlalchemy.engine.interfaces.Dialect) and [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool) constructs:

engine = create\_engine("mysql://scott:tiger@hostname/dbname",

encoding='latin1', echo=**True**)

The string form of the URL is dialect[+driver]://user:password@host/dbname[?key=value..], where dialect is a database name such as mysql, oracle, postgresql, etc., and driver the name of a DBAPI, such as psycopg2, pyodbc, cx\_oracle, etc. Alternatively, the URL can be an instance of [URL](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "sqlalchemy.engine.url.URL" \o "sqlalchemy.engine.url.URL).

\*\*kwargs takes a wide variety of options which are routed towards their appropriate components. Arguments may be specific to the [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine), the underlying [Dialect](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Dialect" \o "sqlalchemy.engine.interfaces.Dialect), as well as the [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool). Specific dialects also accept keyword arguments that are unique to that dialect. Here, we describe the parameters that are common to most [create\_engine()](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "sqlalchemy.create_engine" \o "sqlalchemy.create_engine) usage.

Once established, the newly resulting [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) will request a connection from the underlying [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool) once [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, or a method which depends on it such as [Engine.execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.execute" \o "sqlalchemy.engine.Engine.execute) is invoked. The [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool) in turn will establish the first actual DBAPI connection when this request is received. The [create\_engine()](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "sqlalchemy.create_engine" \o "sqlalchemy.create_engine) call itself does ****not**** establish any actual DBAPI connections directly.

**See also**

[Engine Configuration](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html)

[Dialects](http://docs.sqlalchemy.org/en/rel_1_1/dialects/index.html)

[Working with Engines and Connections](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html)

|  |  |
| --- | --- |
| **Parameters:** | * ****case\_sensitive=True –****if False, result column names will match in a case-insensitive fashion, that is, row['SomeColumn'].如果为False，结果列名称将以不区分大小写的方式进行匹配，即row ['SomeColumn']。   *Changed in version 0.8:*By default, result row names match case-sensitively. In version 0.7 and prior, all matches were case-insensitive.  版本0.8更改：默认情况下，结果行名称区分大小写。 在0.7及更高版本中，所有匹配都不区分大小写。   * ****connect\_args**** – a dictionary of options which will be passed directly to the DBAPI's connect() method as additional keyword arguments. See the example at [Custom DBAPI connect() arguments](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "custom-dbapi-args).一个选项的字典，将直接传递给DBAPI的connect() 方法作为附加的关键字参数。 请参阅Custom DBAPI connect() 参数的示例。 * ****convert\_unicode=False –****if set to True, sets the default behavior of convert\_unicode on the [String](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.String" \o "sqlalchemy.types.String) type to True, regardless of a setting of False on an individual [String](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.String" \o "sqlalchemy.types.String)type, thus causing all [String](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.String" \o "sqlalchemy.types.String) -based columns to accommodate Python unicode objects. This flag is useful as an engine-wide setting when using a DBAPI that does not natively support Python unicode objects and raises an error when one is received (such as pyodbc with FreeTDS).如果设置为True，则将String类型上的convert\_unicode的默认行为设置为True，而不管单个Stringtype上是否设置False，从而导致所有基于String的列适应Python unicode对象。 当使用不是本地支持Python unicode对象的DBAPI并且在收到错误时引起错误（例如具有FreeTDS的pyodbc），该标志在引擎范围内是有用的。   See [String](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.String" \o "sqlalchemy.types.String) for further details on what this flag indicates.   * ****creator**** – a callable which returns a DBAPI connection. This creation function will be passed to the underlying connection pool and will be used to create all new database connections. Usage of this function causes connection parameters specified in the URL argument to be bypassed. * ****echo=False**** – if True, the Engine will log all statements as well as a repr() of their parameter lists to the engines logger, which defaults to sys.stdout. The echoattribute of Engine can be modified at any time to turn logging on and off. If set to the string "debug", result rows will be printed to the standard output as well. This flag ultimately controls a Python logger; see [Configuring Logging](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "dbengine-logging) for information on how to configure logging directly. * ****echo\_pool=False**** – if True, the connection pool will log all checkouts/checkins to the logging stream, which defaults to sys.stdout. This flag ultimately controls a Python logger; see [Configuring Logging](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "dbengine-logging) for information on how to configure logging directly. * ****encoding –****Defaults to utf-8. This is the string encoding used by SQLAlchemy for string encode/decode operations which occur within SQLAlchemy, ****outside of the DBAPI.**** Most modern DBAPIs feature some degree of direct support for Python unicode objects, what you see in Python 2 as a string of the form u'somestring'. For those scenarios where the DBAPI is detected as not supporting a Python unicode object, this encoding is used to determine the source/destination encoding. It is ****not used**** for those cases where the DBAPI handles unicode directly.默认为utf-8。 这是由SQLAlchemy用于DBAPI之外的SQLAlchemy中发生的字符串编码/解码操作的字符串编码。 大多数现代DBAPI对Python unicode对象有一定程度的直接支持，您在Python 2中看到的是u'somestring形式的字符串。 对于DBAPI被检测为不支持Python unicode对象的情况，该编码用于确定源/目标编码。 它不用于DBAPI直接处理unicode的情况。   To properly configure a system to accommodate Python unicode objects, the DBAPI should be configured to handle unicode to the greatest degree as is appropriate - see the notes on unicode pertaining to the specific target database in use at [Dialects](http://docs.sqlalchemy.org/en/rel_1_1/dialects/index.html).  要正确配置系统以适应Python unicode对象，应将DBAPI配置为在最大程度上适当处理unicode - 请参阅与方言使用的特定目标数据库有关的unicode注释。  Areas where string encoding may need to be accommodated outside of the DBAPI include zero or more of:  字符串编码可能需要容纳在DBAPI外部的区域包括以下零个或多个：   * + the values passed to bound parameters, corresponding to the [Unicode](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Unicode" \o "sqlalchemy.types.Unicode) type or the [String](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.String" \o "sqlalchemy.types.String) type when convert\_unicode is True;   + the values returned in result set columns corresponding to the [Unicode](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.Unicode" \o "sqlalchemy.types.Unicode) type or the [String](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.String" \o "sqlalchemy.types.String) type when convert\_unicode is True;   + the string SQL statement passed to the DBAPI's cursor.execute() method;   + the string names of the keys in the bound parameter dictionary passed to the DBAPI's cursor.execute() as well as cursor.setinputsizes()methods;   + the string column names retrieved from the DBAPI's cursor.description attribute.   When using Python 3, the DBAPI is required to support *all* of the above values as Python unicode objects, which in Python 3 are just known as str. In Python 2, the DBAPI does not specify unicode behavior at all, so SQLAlchemy must make decisions for each of the above values on a per-DBAPI basis - implementations are completely inconsistent in their behavior.  当使用Python 3时，需要DBAPI来支持所有上述值作为Python unicode对象，在Python 3中，它被称为str。 在Python 2中，DBAPI根本没有指定unicode行为，所以SQLAlchemy必须在每个DBAPI的基础上为每个上述值做出决定 - 实现在其行为上完全不一致。   * ****execution\_options**** – Dictionary execution options which will be applied to all connections. See [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)将应用于所有连接的字典执行选项。见execution\_options() * ****implicit\_returning=True**** – When True, a RETURNING- compatible construct, if available, will be used to fetch newly generated primary key values when a single row INSERT statement is emitted with no existing returning() clause. This applies to those backends which support RETURNING or a compatible construct, including PostgreSQL, Firebird, Oracle, Microsoft SQL Server. Set this to False to disable the automatic usage of RETURNING.当为True时，如果使用RETURNING兼容的构造（如果可用）将用于在没有现有的return() 子句的情况下发出单行INSERT语句时获取新生成的主键值。 这适用于支持RETURNING或兼容结构的后端，包括PostgreSQL，Firebird，Oracle，Microsoft SQL Server。 将其设置为False以禁用RETURNING的自动使用。 * ****isolation\_level –****this string parameter is interpreted by various dialects in order to affect the transaction isolation level of the database connection. The parameter essentially accepts some subset of these string arguments: "SERIALIZABLE", "REPEATABLE\_READ", "READ\_COMMITTED", "READ\_UNCOMMITTED" and "AUTOCOMMIT". Behavior here varies per backend, and individual dialects should be consulted directly.该字符串参数由各种方言解释，以便影响数据库连接的事务隔离级别。 该参数基本上接受这些字符串参数的一些子集：“SERIALIZABLE”，“REPEATABLE\_READ”，“READ\_COMMITTED”，“READ\_UNCOMMITTED”和“AUTOCOMMIT”。 这里的行为因每个后端而异，应该直接咨询个别方言。   Note that the isolation level can also be set on a per-[Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) basis as well, using the [Connection.execution\_options.isolation\_level](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.execution_options.params.isolation_level" \o "sqlalchemy.engine.Connection.execution_options)feature.  **See also**  [Connection.default\_isolation\_level](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.default_isolation_level" \o "sqlalchemy.engine.Connection.default_isolation_level) - view default level  [Connection.execution\_options.isolation\_level](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.execution_options.params.isolation_level" \o "sqlalchemy.engine.Connection.execution_options) - set per [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) isolation level  [SQLite Transaction Isolation](http://docs.sqlalchemy.org/en/rel_1_1/dialects/sqlite.html" \l "sqlite-isolation-level)  [PostgreSQL Transaction Isolation](http://docs.sqlalchemy.org/en/rel_1_1/dialects/postgresql.html" \l "postgresql-isolation-level)  [MySQL Transaction Isolation](http://docs.sqlalchemy.org/en/rel_1_1/dialects/mysql.html" \l "mysql-isolation-level)  [Setting Transaction Isolation Levels](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_transaction.html" \l "session-transaction-isolation) - for the ORM   * ****label\_length=None**** – optional integer value which limits the size of dynamically generated column labels to that many characters. If less than 6, labels are generated as "\_(counter)". If None, the value of dialect.max\_identifier\_length is used instead. * ****listeners**** – A list of one or more [PoolListener](http://docs.sqlalchemy.org/en/rel_1_1/core/interfaces.html" \l "sqlalchemy.interfaces.PoolListener" \o "sqlalchemy.interfaces.PoolListener) objects which will receive connection pool events. * ****logging\_name**** – String identifier which will be used within the "name" field of logging records generated within the "sqlalchemy.engine" logger. Defaults to a hexstring of the object's id. * ****max\_overflow=10**** – the number of connections to allow in connection pool "overflow", that is connections that can be opened above and beyond the pool\_size setting, which defaults to five. this is only used with [QueuePool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.QueuePool" \o "sqlalchemy.pool.QueuePool). * ****module=None**** – reference to a Python module object (the module itself, not its string name). Specifies an alternate DBAPI module to be used by the engine's dialect. Each sub-dialect references a specific DBAPI which will be imported before first connect. This parameter causes the import to be bypassed, and the given module to be used instead. Can be used for testing of DBAPIs as well as to inject "mock" DBAPI implementations into the [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine). * ****paramstyle=None**** – The [paramstyle](http://legacy.python.org/dev/peps/pep-0249/" \l "paramstyle) to use when rendering bound parameters. This style defaults to the one recommended by the DBAPI itself, which is retrieved from the .paramstyle attribute of the DBAPI. However, most DBAPIs accept more than one paramstyle, and in particular it may be desirable to change a "named" paramstyle into a "positional" one, or vice versa. When this attribute is passed, it should be one of the values "qmark", "numeric", "named", "format" or "pyformat", and should correspond to a parameter style known to be supported by the DBAPI in use. * ****pool=None**** – an already-constructed instance of [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool), such as a [QueuePool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.QueuePool" \o "sqlalchemy.pool.QueuePool) instance. If non-None, this pool will be used directly as the underlying connection pool for the engine, bypassing whatever connection parameters are present in the URL argument. For information on constructing connection pools manually, see [Connection Pooling](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html). * ****poolclass=None**** – a [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool) subclass, which will be used to create a connection pool instance using the connection parameters given in the URL. Note this differs from pool in that you don't actually instantiate the pool in this case, you just indicate what type of pool to be used. * ****pool\_logging\_name**** – String identifier which will be used within the "name" field of logging records generated within the "sqlalchemy.pool" logger. Defaults to a hexstring of the object's id. * ****pool\_size=5**** – the number of connections to keep open inside the connection pool. This used with [QueuePool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.QueuePool" \o "sqlalchemy.pool.QueuePool) as well as [SingletonThreadPool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.SingletonThreadPool" \o "sqlalchemy.pool.SingletonThreadPool). With[QueuePool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.QueuePool" \o "sqlalchemy.pool.QueuePool), a pool\_size setting of 0 indicates no limit; to disable pooling, set poolclass to [NullPool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.NullPool" \o "sqlalchemy.pool.NullPool) instead. * ****pool\_recycle=-1 –****this setting causes the pool to recycle connections after the given number of seconds has passed. It defaults to -1, or no timeout. For example, setting to 3600 means connections will be recycled after one hour. Note that MySQL in particular will disconnect automatically if no activity is detected on a connection for eight hours (although this is configurable with the MySQLDB connection itself and the server configuration as well).   **See also**  [Setting Pool Recycle](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "pool-setting-recycle)   * ****pool\_reset\_on\_return='rollback'**** –   set the "reset on return" behavior of the pool, which is whether rollback(), commit(), or nothing is called upon connections being returned to the pool. See the docstring for reset\_on\_return at [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool).  *New in version 0.7.6.*   * ****pool\_timeout=30**** – number of seconds to wait before giving up on getting a connection from the pool. This is only used with [QueuePool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.QueuePool" \o "sqlalchemy.pool.QueuePool). * ****strategy='plain'**** –   selects alternate engine implementations. Currently available are:   * + the threadlocal strategy, which is described in [Using the Threadlocal Execution Strategy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "threadlocal-strategy);   + the mock strategy, which dispatches all statement execution to a function passed as the argument executor. See [example in the FAQ](http://docs.sqlalchemy.org/en/latest/faq/metadata_schema.html" \l "how-can-i-get-the-create-table-drop-table-output-as-a-string). * ****executor=None**** – a function taking arguments (sql, \*multiparams, \*\*params), to which the mock strategy will dispatch all statement execution. Used only by strategy='mock'. |

sqlalchemy.**engine\_from\_config**(*configuration*, *prefix='sqlalchemy.'*, *\*\*kwargs*)

Create a new Engine instance using a configuration dictionary.

使用配置字典创建一个新的Engine实例。

The dictionary is typically produced from a config file.

字典通常由配置文件生成。

The keys of interest to engine\_from\_config() should be prefixed, e.g. sqlalchemy.url, sqlalchemy.echo, etc. The 'prefix' argument indicates the prefix to be searched for. Each matching key (after the prefix is stripped) is treated as though it were the corresponding keyword argument to a [create\_engine()](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "sqlalchemy.create_engine" \o "sqlalchemy.create_engine) call.

engine\_from\_config() 的关键字应该是前缀，例如 sqlalchemy.url，sqlalchemy.echo等。'prefix'参数表示要搜索的前缀。 每个匹配的键（在前缀被剥离之后）被视为就是create\_engine() 调用的相应关键字参数。

The only required key is (assuming the default prefix) sqlalchemy.url, which provides the [database URL](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "database-urls).

唯一需要的键是（假设默认前缀）sqlalchemy.url，它提供了数据库URL。

A select set of keyword arguments will be "coerced" to their expected type based on string values. The set of arguments is extensible per-dialect using the engine\_config\_types accessor.

一组关键字参数将根据字符串值“强制”到其预期类型。 该参数集是使用engine\_config\_types访问器可扩展的每个方言。

|  |  |
| --- | --- |
| **Parameters:** | * ****configuration**** – A dictionary (typically produced from a config file, but this is not a requirement). Items whose keys start with the value of 'prefix' will have that prefix stripped, and will then be passed to create\_engine. * ****prefix**** – Prefix to match and then strip from keys in 'configuration'. * ****kwargs**** – Each keyword argument to engine\_from\_config() itself overrides the corresponding item taken from the 'configuration' dictionary. Keyword arguments should *not* be prefixed. |

sqlalchemy.engine.url.**make\_url**(*name\_or\_url*)

Given a string or unicode instance, produce a new URL instance.

给定一个字符串或unicode实例，生成一个新的URL实例。

The given string is parsed according to the RFC 1738 spec. If an existing URL object is passed, just returns the object.

给定的字符串根据RFC 1738规范进行解析。 如果传递了一个现有的URL对象，只需返回该对象。

*class*sqlalchemy.engine.url.**URL**(*drivername*, *username=None*, *password=None*, *host=None*, *port=None*, *database=None*, *query=None*)

Represent the components of a URL used to connect to a database.

表示用于连接到数据库的URL的组件。

This object is suitable to be passed directly to a [create\_engine()](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "sqlalchemy.create_engine" \o "sqlalchemy.create_engine) call. The fields of the URL are parsed from a string by the [make\_url()](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "sqlalchemy.engine.url.make_url" \o "sqlalchemy.engine.url.make_url) function. the string format of the URL is an RFC-1738-style string.

该对象适合直接传递到create\_engine() 调用。 URL的字段由make\_url() 函数从字符串中解析出来。 URL的字符串格式是RFC-1738样式的字符串。

All initialization parameters are available as public attributes.

所有初始化参数都可用作公共属性。

|  |  |
| --- | --- |
| **Parameters:** | * ****drivername**** – the name of the database backend. This name will correspond to a module in sqlalchemy/databases or a third party plug-in. * ****username**** – The user name. * ****password**** – database password. * ****host**** – The name of the host. * ****port**** – The port number. * ****database**** – The database name. * ****query**** – A dictionary of options to be passed to the dialect and/or the DBAPI upon connect. |

**get\_dialect**()

Return the SQLAlchemy database dialect class corresponding to this URL's driver name.

返回与此URL驱动程序名称相对应的SQLAlchemy数据库方言类。

**translate\_connect\_args**(*names=[]*, *\*\*kw*)

Translate url attributes into a dictionary of connection arguments.

将URL属性转换为连接参数的字典。

Returns attributes of this url (host, database, username, password, port) as a plain dictionary. The attribute names are used as the keys by default. Unset or false attributes are omitted from the final dictionary.

返回此url（主机，数据库，用户名，密码，端口）的属性作为普通字典。 默认情况下，属性名称用作键。 最终字典中省略了未设置或假属性。

|  |  |
| --- | --- |
| **Parameters:** | * ****\*\*kw**** – Optional, alternate key names for url attributes. * ****names**** – Deprecated. Same purpose as the keyword-based alternate names, but correlates the name to the original positionally. |

### 5.1.4 Pooling

The [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) will ask the connection pool for a connection when the connect() or execute() methods are called. The default connection pool, [QueuePool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.QueuePool" \o "sqlalchemy.pool.QueuePool), will open connections to the database on an as-needed basis. As concurrent statements are executed, [QueuePool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.QueuePool" \o "sqlalchemy.pool.QueuePool) will grow its pool of connections to a default size of five, and will allow a default "overflow" of ten. Since the [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) is essentially "home base" for the connection pool, it follows that you should keep a single [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) per database established within an application, rather than creating a new one for each connection.

当调用connect() 或execute() 方法时，引擎将要求连接池进行连接。 默认连接池QueuePool将根据需要打开与数据库的连接。 在执行并发语句时，QueuePool会将其连接池的大小增加到默认大小为5，并允许默认的“溢出”为10。 由于引擎本质上是连接池的“主站”，因此您应该在应用程序内建立一个数据库，而不是为每个连接创建一个新引擎。

**Note**

[QueuePool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.QueuePool" \o "sqlalchemy.pool.QueuePool) is not used by default for SQLite engines. See [SQLite](http://docs.sqlalchemy.org/en/rel_1_1/dialects/sqlite.html) for details on SQLite connection pool usage.

For more information on connection pooling, see [Connection Pooling](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html).

默认情况下，SQLite引擎不使用QueuePool。 有关SQLite连接池使用情况的详细信息，请参阅SQLite。

有关连接池的更多信息，请参阅连接池。

5.1.5 Custom DBAPI connect() arguments

Custom arguments used when issuing the connect() call to the underlying DBAPI may be issued in three distinct ways. String-based arguments can be passed directly from the URL string as query arguments:

对底层DBAPI发出connect() 调用时使用的自定义参数可能会以三种不同的方式发布。 基于字符串的参数可以直接从URL字符串作为查询参数传递：

db = create\_engine('postgresql://scott:tiger@localhost/test?argument1=foo&argument2=bar')

If SQLAlchemy's database connector is aware of a particular query argument, it may convert its type from string to its proper type.

如果SQLAlchemy的数据库连接器知道一个特定的查询参数，它可能会将其类型从字符串转换为正确的类型。

[create\_engine()](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "sqlalchemy.create_engine" \o "sqlalchemy.create_engine) also takes an argument connect\_args which is an additional dictionary that will be passed to connect(). This can be used when arguments of a type other than string are required, and SQLAlchemy's database connector has no type conversion logic present for that parameter:

create\_engine() 也接受一个参数connect\_args，这是一个额外的字典，将传递给connect() 。 当需要非string类型的参数时，可以使用此参数，SQLAlchemy的数据库连接器不存在该参数的类型转换逻辑：

db = create\_engine('postgresql://scott:tiger@localhost/test', connect\_args = {'argument1':17, 'argument2':'bar'})

The most customizable connection method of all is to pass a creator argument, which specifies a callable that returns a DBAPI connection:

最可定制的连接方法是传递一个创建者参数，该参数指定一个可返回DBAPI连接的可调用参数：

**def** connect():

**return** psycopg.connect(user='scott', host='localhost')

db = create\_engine('postgresql://', creator=connect)

### 5.1.6 Configuring Logging

Python's standard [logging](http://docs.python.org/library/logging.html) module is used to implement informational and debug log output with SQLAlchemy. This allows SQLAlchemy's logging to integrate in a standard way with other applications and libraries. The echo and echo\_pool flags that are present on [create\_engine()](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "sqlalchemy.create_engine" \o "sqlalchemy.create_engine), as well as the echo\_uow flag used on[Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session), all interact with regular loggers.

Python的标准日志记录模块用于使用SQLAlchemy实现信息和调试日志输出。 这允许SQLAlchemy的日志记录以标准方式与其他应用程序和库进行集成。 create\_engine() 上存在的echo和echo\_pool标志以及在session上使用的echo\_uow标志都与常规记录器进行交互。

This section assumes familiarity with the above linked logging module. All logging performed by SQLAlchemy exists underneath the sqlalchemy namespace, as used by logging.getLogger('sqlalchemy'). When logging has been configured (i.e. such as via logging.basicConfig()), the general namespace of SA loggers that can be turned on is as follows:

本节假设熟悉上述链接的日志记录模块。 SQLAlchemy执行的所有日志都存在于sqlalchemy命名空间的下面，由logging.getLogger（'sqlalchemy'）使用。 当配置了日志记录（例如通过logging.basicConfig() ）时，可以打开的SA记录器的通用命名空间如下所示：

* sqlalchemy.engine - controls SQL echoing. set to logging.INFO for SQL query output, logging.DEBUG for query + result set output.
* sqlalchemy.dialects - controls custom logging for SQL dialects. See the documentation of individual dialects for details.
* sqlalchemy.pool - controls connection pool logging. set to logging.INFO or lower to log connection pool checkouts/checkins.
* sqlalchemy.orm - controls logging of various ORM functions. set to logging.INFO for information on mapper configurations.

For example, to log SQL queries using Python logging instead of the echo=True flag:

例如，使用Python日志记录来记录SQL查询，而不是echo = True标志：

**import** **logging**

logging.basicConfig()logging.getLogger('sqlalchemy.engine').setLevel(logging.INFO)

By default, the log level is set to logging.WARN within the entire sqlalchemy namespace so that no log operations occur, even within an application that has logging enabled otherwise.

默认情况下，日志级别设置为整个sqlalchemy命名空间中的logging.WARN，以便在不启用日志记录的应用程序中发生日志操作。

The echo flags present as keyword arguments to [create\_engine()](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "sqlalchemy.create_engine" \o "sqlalchemy.create_engine) and others as well as the echo property on [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine), when set to True, will first attempt to ensure that logging is enabled. Unfortunately, the logging module provides no way of determining if output has already been configured (note we are referring to if a logging configuration has been set up, not just that the logging level is set). For this reason, any echo=True flags will result in a call to logging.basicConfig()using sys.stdout as the destination. It also sets up a default format using the level name, timestamp, and logger name. Note that this configuration has the affect of being configured ****in addition**** to any existing logger configurations. Therefore, ****when using Python logging, ensure all echo flags are set to False at all times****, to avoid getting duplicate log lines.

作为create\_engine() 和其他的关键字参数的echo标记以及Engine的echo属性设置为True时，将首先尝试确保启用日志记录。不幸的是，日志记录模块无法确定输出是否已经被配置（请注意，如果已经设置了日志记录配置，而不仅仅是记录级别设置）。因此，任何echo = True标志将导致使用sys.stdout作为目的地对logging.basicConfig() 进行调用。它还使用级别名称，时间戳记和记录器名称设置默认格式。请注意，除了任何现有的记录器配置之外，此配置还有被配置的影响。因此，当使用Python日志记录时，请确保始终将所有回显标志设置为False，以避免获得重复的日志行。

The logger name of instance such as an [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) or [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool) defaults to using a truncated hex identifier string. To set this to a specific name, use the "logging\_name" and "pool\_logging\_name" keyword arguments with [sqlalchemy.create\_engine()](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "sqlalchemy.create_engine" \o "sqlalchemy.create_engine).

例如Engine或Pool的实例的记录器名称默认使用截断的十六进制标识符字符串。要将其设置为特定名称，请使用sqlalchemy.create\_engine() 的“logging\_name”和“pool\_logging\_name”关键字参数。

**Note**

The SQLAlchemy [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) conserves Python function call overhead by only emitting log statements when the current logging level is detected as logging.INFO or logging.DEBUG. It only checks this level when a new connection is procured from the connection pool. Therefore when changing the logging configuration for an already-running application, any [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) that's currently active, or more commonly a [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) object that's active in a transaction, won't log any SQL according to the new configuration until a new [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) is procured (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), this is after the current transaction ends and a new one begins)

# **5.2 Working with Engines and Connections**

This section details direct usage of the [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine), [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection), and related objects. Its important to note that when using the SQLAlchemy ORM, these objects are not generally accessed; instead, 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 used as the interface to the database. However, for applications that are built around direct usage of textual SQL statements and/or SQL expression constructs without involvement by the ORM's higher level management services, the [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) and [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) are king (and queen?) - read on.

5.2.1 Basic Usage

Recall from [Engine Configuration](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html) that an [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) is created via the [create\_engine()](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "sqlalchemy.create_engine" \o "sqlalchemy.create_engine) call:

engine = create\_engine('mysql://scott:tiger@localhost/test')

The typical usage of [create\_engine()](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "sqlalchemy.create_engine" \o "sqlalchemy.create_engine) is once per particular database URL, held globally for the lifetime of a single application process. A single [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) manages many individual DBAPI connections on behalf of the process and is intended to be called upon in a concurrent fashion. The [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) is ****not**** synonymous to the DBAPI connect function, which represents just one connection resource - the [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) is most efficient when created just once at the module level of an application, not per-object or per-function call.

create\_engine() 的典型用法是每个特定数据库URL一次，在单个应用程序进程的整个生命周期内全局保留。单个引擎代表进程管理许多单独的DBAPI连接，并且旨在以并发方式调用。引擎不是DBAPI连接功能的同义词，它只表示一个连接资源 - 引擎在应用程序的模块级别（而不是每个对象或每个函数调用）上仅创建一次时效率最高。

For a multiple-process application that uses the os.fork system call, or for example the Python multiprocessing module, it's usually required that a separate [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) be used for each child process. This is because the [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) maintains a reference to a connection pool that ultimately references DBAPI connections - these tend to not be portable across process boundaries. An [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) that is configured not to use pooling (which is achieved via the usage of [NullPool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.NullPool" \o "sqlalchemy.pool.NullPool)) does not have this requirement.

对于使用os.fork系统调用的多进程应用程序，或者例如Python多处理模块，通常需要为每个子进程使用单独的引擎。这是因为引擎维护对最终引用DBAPI连接的连接池的引用 - 这些连接池往往不能跨过程边界移植。配置为不使用池（通过使用NullPool实现）的引擎没有此要求。

The engine can be used directly to issue SQL to the database. The most generic way is first procure a connection resource, which you get via the[Engine.connect()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.connect" \o "sqlalchemy.engine.Engine.connect) method:

引擎可以直接用于向数据库发出SQL。最通用的方法是首先获取连接资源，通过Engine.connect() 方法获取连接资源：

connection = engine.connect()

result = connection.execute("select username from users")

**for** row **in** result:

print("username:", row['username'])connection.close()

The connection is an instance of [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection), which is a ****proxy**** object for an actual DBAPI connection. The DBAPI connection is retrieved from the connection pool at the point at which [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) is created.

连接是Connection的一个实例，它是实际DBAPI连接的代理对象。 在创建连接点的连接池中检索DBAPI连接。

The returned result is an instance of [ResultProxy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy" \o "sqlalchemy.engine.ResultProxy), which references a DBAPI cursor and provides a largely compatible interface with that of the DBAPI cursor. The DBAPI cursor will be closed by the [ResultProxy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy" \o "sqlalchemy.engine.ResultProxy) when all of its result rows (if any) are exhausted. A [ResultProxy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy" \o "sqlalchemy.engine.ResultProxy) that returns no rows, such as that of an UPDATE statement (without any returned rows), releases cursor resources immediately upon construction.

返回的结果是ResultProxy的一个实例，它引用了一个DBAPI游标，并提供了一个与DBAPI游标相当的接口。 当所有结果行（如果有的话）耗尽时，DBAPI游标将被ResultProxy关闭。 返回没有行（如UPDATE语句（没有任何返回的行））的ResultProxy，在构建时会立即释放游标资源。

When the [close()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.close" \o "sqlalchemy.engine.Connection.close) method is called, the referenced DBAPI connection is [released](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-released) to the connection pool. From the perspective of the database itself, nothing is actually "closed", assuming pooling is in use. The pooling mechanism issues a rollback() call on the DBAPI connection so that any transactional state or locks are removed, and the connection is ready for its next usage.

The above procedure can be performed in a shorthand way by using the [execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.execute" \o "sqlalchemy.engine.Engine.execute) method of [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) itself:

上述过程可以通过使用引擎本身的execute() 方法以简写方式执行：

result = engine.execute("select username from users")**for** row **in** result:

print("username:", row['username'])

Where above, the [execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.execute" \o "sqlalchemy.engine.Engine.execute) method acquires a new [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) on its own, executes the statement with that object, and returns the [ResultProxy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy" \o "sqlalchemy.engine.ResultProxy). In this case, the [ResultProxy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy" \o "sqlalchemy.engine.ResultProxy) contains a special flag known as close\_with\_result, which indicates that when its underlying DBAPI cursor is closed, the [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection)object itself is also closed, which again returns the DBAPI connection to the connection pool, releasing transactional resources.

在上面，execute() 方法自己获取一个新的Connection，执行该对象的语句，并返回ResultProxy。 在这种情况下，ResultProxy包含一个称为close\_with\_result的特殊标志，表示当其底层DBAPI游标关闭时，Connectionobject本身也被关闭，它再次返回到连接池的DBAPI连接，释放事务资源。

If the [ResultProxy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy" \o "sqlalchemy.engine.ResultProxy) potentially has rows remaining, it can be instructed to close out its resources explicitly:

如果ResultProxy可能有剩余行，则可以指示它明确地关闭它的资源：

result.close()

If the [ResultProxy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy" \o "sqlalchemy.engine.ResultProxy) has pending rows remaining and is dereferenced by the application without being closed, Python garbage collection will ultimately close out the cursor as well as trigger a return of the pooled DBAPI connection resource to the pool (SQLAlchemy achieves this by the usage of weakref callbacks - *never* the \_\_del\_\_ method) - however it's never a good idea to rely upon Python garbage collection to manage resources.

如果ResultProxy有待处理的行剩余，并被应用程序取消引用而不被关闭，Python垃圾收集将最终关闭游标以及触发将池DBAPI连接资源返回到池（SQLAlchemy通过使用weakref来实现这一点 回调 - 从不使用\_\_del\_\_方法） - 但是依靠Python垃圾收集来管理资源绝非一个好主意。

Our example above illustrated the execution of a textual SQL string. The [execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.execute" \o "sqlalchemy.engine.Connection.execute) method can of course accommodate more than that, including the variety of SQL expression constructs described in [SQL Expression Language Tutorial](http://docs.sqlalchemy.org/en/rel_1_1/core/tutorial.html).

我们上面的例子说明了一个文本SQL字符串的执行。 当然，execute() 方法可以容纳更多的内容，包括SQL表达式语言教程中描述的各种SQL表达式构造。

5.2.2 Using Transactions

**Note**

This section describes how to use transactions when working directly with [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) and [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) objects. When using the SQLAlchemy ORM, the public API for transaction control is 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) object, which makes usage of the [Transaction](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction" \o "sqlalchemy.engine.Transaction) object internally. See [Managing Transactions](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_transaction.html" \l "unitofwork-transaction) for further information.

本节介绍如何在直接使用Engine和Connection对象时使用事务。 当使用SQLAlchemy ORM时，用于事务控制的公共API是通过Session对象，它使内部使用Transaction对象。 有关更多信息，请参阅管理事务

The [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) object provides a [begin()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.begin" \o "sqlalchemy.engine.Connection.begin) method which returns a [Transaction](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction" \o "sqlalchemy.engine.Transaction) object. This object is usually used within a try/except clause so that it is guaranteed to invoke [Transaction.rollback()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction.rollback" \o "sqlalchemy.engine.Transaction.rollback) or [Transaction.commit()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction.commit" \o "sqlalchemy.engine.Transaction.commit):

Connection对象提供了一个返回一个Transaction对象的begin() 方法。 此对象通常在try / except子句中使用，以保证可以调用Transaction.rollback() 或Transaction.commit() ：

connection = engine.connect()trans = connection.begin()**try**:

r1 = connection.execute(table1.select())

connection.execute(table1.insert(), col1=7, col2='this is some data')

trans.commit()**except**:

trans.rollback()

**raise**

The above block can be created more succinctly using context managers, either given an [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine):

*# runs a transaction***with** engine.begin() **as** connection:

r1 = connection.execute(table1.select())

connection.execute(table1.insert(), col1=7, col2='this is some data')

Or from the [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection), in which case the [Transaction](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction" \o "sqlalchemy.engine.Transaction) object is available as well:

**with** connection.begin() **as** trans:

r1 = connection.execute(table1.select())

connection.execute(table1.insert(), col1=7, col2='this is some data')

### Nesting of Transaction Blocks

The [Transaction](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction" \o "sqlalchemy.engine.Transaction) object also handles "nested" behavior by keeping track of the outermost begin/commit pair. In this example, two functions both issue a transaction on a [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection), but only the outermost [Transaction](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction" \o "sqlalchemy.engine.Transaction) object actually takes effect when it is committed.

Transaction对象还通过跟踪最外面的begin / commit对来处理“嵌套”行为。 在这个例子中，两个函数都在Connection上发出一个事务，但只有最外层的Transaction对象在提交时才会生效。

*# method\_a starts a transaction and calls method\_b***def** method\_a(connection):

trans = connection.begin() *# open a transaction*

**try**:

method\_b(connection)

trans.commit() *# transaction is committed here*

**except**:

trans.rollback() *# this rolls back the transaction unconditionally*

**raise**

*# method\_b also starts a transaction***def** method\_b(connection):

trans = connection.begin() *# open a transaction - this runs in the context of method\_a's transaction*

**try**:

connection.execute("insert into mytable values ('bat', 'lala')")

connection.execute(mytable.insert(), col1='bat', col2='lala')

trans.commit() *# transaction is not committed yet*

**except**:

trans.rollback() *# this rolls back the transaction unconditionally*

**raise**

*# open a Connection and call method\_a*conn = engine.connect()method\_a(conn)conn.close()

Above, method\_a is called first, which calls connection.begin(). Then it calls method\_b. When method\_b calls connection.begin(), it just increments a counter that is decremented when it calls commit(). If either method\_a or method\_b calls rollback(), the whole transaction is rolled back. The transaction is not committed until method\_a calls the commit() method. This "nesting" behavior allows the creation of functions which "guarantee" that a transaction will be used if one was not already available, but will automatically participate in an enclosing transaction if one exists.

以上，method\_a首先被调用，它调用connection.begin() 。 然后调用method\_b。 当method\_b调用connection.begin() 时，它只会增加一个在调用commit() 时递减的计数器。 如果method\_a或method\_b调用rollback() ，则整个事务将回滚。 在method\_a调用commit() 方法之前，事务不会被提交。 这种“嵌套”行为允许创建“保证”如果一个不可用的事务将被使用的功能，但是如果存在的话，它将自动参与一个封闭的事务。

5.2.3 Understanding Autocommit

The previous transaction example illustrates how to use [Transaction](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction" \o "sqlalchemy.engine.Transaction) so that several executions can take part in the same transaction. What happens when we issue an INSERT, UPDATE or DELETE call without using [Transaction](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction" \o "sqlalchemy.engine.Transaction)? While some DBAPI implementations provide various special "non-transactional" modes, the core behavior of DBAPI per PEP-0249 is that a *transaction is always in progress*, providing only rollback() and commit() methods but no begin(). SQLAlchemy assumes this is the case for any given DBAPI.

以前的事务示例说明了如何使用Transaction，以便几个执行可以参与相同的事务。当我们发出INSERT，UPDATE或DELETE调用而不使用Transaction时会发生什么？虽然一些DBAPI实现提供了各种特殊的“非事务”模式，但是每个PEP-0249的DBAPI的核心行为是事务始终在进行中，只提供rollback() 和commit() 方法，但没有begin() 。 SQLAlchemy假设任何给定的DBAPI是这种情况。

Given this requirement, SQLAlchemy implements its own "autocommit" feature which works completely consistently across all backends. This is achieved by detecting statements which represent data-changing operations, i.e. INSERT, UPDATE, DELETE, as well as data definition language (DDL) statements such as CREATE TABLE, ALTER TABLE, and then issuing a COMMIT automatically if no transaction is in progress. The detection is based on the presence of the autocommit=True execution option on the statement. If the statement is a text-only statement and the flag is not set, a regular expression is used to detect INSERT, UPDATE, DELETE, as well as a variety of other commands for a particular backend:

鉴于此要求，SQLAlchemy实现了自己的“自动提交”功能，可在所有后端完全一致。这通过检测表示数据更改操作（即INSERT，UPDATE，DELETE）以及数据定义语言（DDL）语句（如CREATE TABLE，ALTER TABLE）的语句来实现，然后如果没有事务正在进行，则自动发出COMMIT 。检测是基于语句上autocommit = True执行选项的存在。如果语句是纯文本语句，并且标志未设置，则使用正则表达式来检测INSERT，UPDATE，DELETE以及特定后端的各种其他命令：

conn = engine.connect()conn.execute("INSERT INTO users VALUES (1, 'john')") *# autocommits*

The "autocommit" feature is only in effect when no [Transaction](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction" \o "sqlalchemy.engine.Transaction) has otherwise been declared. This means the feature is not generally used with the ORM, 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) object by default always maintains an ongoing [Transaction](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction" \o "sqlalchemy.engine.Transaction).

“自动提交”功能仅在没有以其他方式声明的情况下生效。 这意味着该功能通常不与ORM一起使用，因为Session对象默认情况下始终保持正在进行的事务。

Full control of the "autocommit" behavior is available using the generative [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) method provided on [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection), [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine), [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable), using the "autocommit" flag which will turn on or off the autocommit for the selected scope. For example, 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 stored procedure that commits might use it so that a SELECT statement will issue a COMMIT:

使用“autocommit”标志可以使用Connection，Engine，Executable提供的生成Connection.execution\_options() 方法来完全控制“自动提交”行为，该标志将打开或关闭所选作用域的自动提交。 例如，表示提交的存储过程的text() 结构可能会使用它，以便SELECT语句将发出COMMIT：

engine.execute(text("SELECT my\_mutating\_procedure()").execution\_options(autocommit=**True**))

### 5.2.4 Connectionless Execution, Implicit Execution

Recall from the first section we mentioned executing with and without explicit usage of [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection). "Connectionless" execution refers to the usage of the execute() method on an object which is not a [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection). This was illustrated using the [execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.execute" \o "sqlalchemy.engine.Engine.execute) method of [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine):

从我们提到的第一节回忆一下，没有明确使用Connection。 “无连接”执行是指对不是Connection的对象使用execute() 方法。 这使用引擎的execute() 方法进行了说明：

result = engine.execute("select username from users")

**for** row **in** result:

print("username:", row['username'])

In addition to "connectionless" execution, it is also possible to use the [execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable.execute" \o "sqlalchemy.sql.expression.Executable.execute) method of any [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable) construct, which is a marker for SQL expression objects that support execution. The SQL expression object itself references 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) known as the ****bind****, which it uses in order to provide so-called "implicit" execution services.

除了“无连接”执行之外，还可以使用任何Executable结构的execute() 方法，该构造是支持执行的SQL表达式对象的标记。 SQL表达式对象本身引用称为绑定的引擎或连接，它使用它来提供所谓的“隐式”执行服务。

Given a table as below:

给出如下表格：

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

meta = MetaData()users\_table = Table('users', meta,

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

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

Explicit execution delivers the SQL text or constructed SQL expression to the [execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.execute" \o "sqlalchemy.engine.Connection.execute) method of [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection):

显式执行将SQL文本或构造的SQL表达式传递给Connection的execute() 方法：

engine = create\_engine('sqlite:///file.db')connection = engine.connect()result = connection.execute(users\_table.select())**for** row **in** result:

*# ....*connection.close()

Explicit, connectionless execution delivers the expression to the [execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.execute" \o "sqlalchemy.engine.Engine.execute) method of [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine):

显式的无连接执行将表达式传递给Engine的execute() 方法：

engine = create\_engine('sqlite:///file.db')result = engine.execute(users\_table.select())**for** row **in** result:

*# ....*result.close()

Implicit execution is also connectionless, and makes usage of the [execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable.execute" \o "sqlalchemy.sql.expression.Executable.execute) method on the expression itself. This method is provided as part of the [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable)class, which refers to a SQL statement that is sufficient for being invoked against the database. The method makes usage of the assumption that either 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) has been ****bound**** to the expression object. By "bound" we mean that the special attribute [MetaData.bind](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData.bind" \o "sqlalchemy.schema.MetaData.bind) has been used to associate a series of[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects and all SQL constructs derived from them with a specific engine:

engine = create\_engine('sqlite:///file.db')meta.bind = engineresult = users\_table.select().execute()**for** row **in** result:

*# ....*result.close()

Above, we associate an [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) with a [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) object using the special attribute [MetaData.bind](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData.bind" \o "sqlalchemy.schema.MetaData.bind). The [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) construct produced from the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object has a method [execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable.execute" \o "sqlalchemy.sql.expression.Executable.execute), which will search for an [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) that's "bound" to the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table).

Overall, the usage of "bound metadata" has three general effects:

* SQL statement objects gain an [Executable.execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable.execute" \o "sqlalchemy.sql.expression.Executable.execute) method which automatically locates a "bind" with which to execute themselves.
* 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) object supports using "bound metadata" in order to establish which [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) should be used to invoke SQL statements on behalf of a particular mapped class, though 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 features its own explicit system of establishing complex [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine)/ mapped class configurations.
* 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), [MetaData.drop\_all()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData.drop_all" \o "sqlalchemy.schema.MetaData.drop_all), [Table.create()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.create" \o "sqlalchemy.schema.Table.create), [Table.drop()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.drop" \o "sqlalchemy.schema.Table.drop), and "autoload" features all make usage of the bound[Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) automatically without the need to pass it explicitly.

**Note**

The concepts of "bound metadata" and "implicit execution" are not emphasized in modern SQLAlchemy. While they offer some convenience, they are no longer required by any API and are never necessary.

In applications where multiple [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) objects are present, each one logically associated with a certain set of tables (i.e. *vertical sharding*), the "bound metadata" technique can be used so that individual [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) can refer to the appropriate [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) automatically; in particular this is supported within the ORM 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) object as a means to associate [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects with an appropriate [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine), as an alternative to using the bind arguments accepted directly 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).

However, the "implicit execution" technique is not at all appropriate for use with the ORM, as it bypasses the transactional context 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).

Overall, in the *vast majority* of cases, "bound metadata" and "implicit execution" are ****not useful****. While "bound metadata" has a marginal level of usefulness with regards to ORM configuration, "implicit execution" is a very old usage pattern that in most cases is more confusing than it is helpful, and its usage is discouraged. Both patterns seem to encourage the overuse of expedient "short cuts" in application design which lead to problems later on.

Modern SQLAlchemy usage, especially the ORM, places a heavy stress on working within the context of a transaction at all times; the "implicit execution" concept makes the job of associating statement execution with a particular transaction much more difficult. The [Executable.execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable.execute" \o "sqlalchemy.sql.expression.Executable.execute) method on a particular SQL statement usually implies that the execution is not part of any particular transaction, which is usually not the desired effect.

In both "connectionless" examples, the [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) is created behind the scenes; the [ResultProxy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy" \o "sqlalchemy.engine.ResultProxy) returned by the execute() call references the [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) used to issue the SQL statement. When the [ResultProxy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy" \o "sqlalchemy.engine.ResultProxy) is closed, the underlying [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) is closed for us, resulting in the DBAPI connection being returned to the pool with transactional resources removed.

5.2.5 Translation of Schema Names

To support multi-tenancy applications that distribute common sets of tables into multiple schemas, the[Connection.execution\_options.schema\_translate\_map](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.execution_options.params.schema_translate_map" \o "sqlalchemy.engine.Connection.execution_options) execution option may be used to repurpose a set of [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects to render under different schema names without any changes.

Given a table:

user\_table = Table(

'user', metadata,

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

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

The "schema" of this [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) as defined by the [Table.schema](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.params.schema" \o "sqlalchemy.schema.Table) attribute is None. The [Connection.execution\_options.schema\_translate\_map](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.execution_options.params.schema_translate_map" \o "sqlalchemy.engine.Connection.execution_options) can specify that all [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) objects with a schema of None would instead render the schema as user\_schema\_one:

connection = engine.connect().execution\_options(

schema\_translate\_map={**None**: "user\_schema\_one"})

result = connection.execute(user\_table.select())

The above code will invoke SQL on the database of the form:

SELECT user\_schema\_one.user.id, user\_schema\_one.user.name FROMuser\_schema.user

That is, the schema name is substituted with our translated name. The map can specify any number of target->destination schemas:

connection = engine.connect().execution\_options(

schema\_translate\_map={

**None**: "user\_schema\_one", *# no schema name -> "user\_schema\_one"*

"special": "special\_schema", *# schema="special" becomes "special\_schema"*

"public": **None** *# Table objects with schema="public" will render with no schema*

})

The [Connection.execution\_options.schema\_translate\_map](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.execution_options.params.schema_translate_map" \o "sqlalchemy.engine.Connection.execution_options) parameter affects all DDL and SQL constructs generated from the SQL expression language, as derived from the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) or [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) objects. It does ****not**** impact literal string SQL used via the [expression.text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text) construct nor via plain strings passed to [Connection.execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.execute" \o "sqlalchemy.engine.Connection.execute).

The feature takes effect ****only**** in those cases where the name of the schema is derived directly from that of a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) or [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence); it does not impact methods where a string schema name is passed directly. By this pattern, it takes effect within the "can create" / "can drop" checks performed by methods such as [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) or [MetaData.drop\_all()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData.drop_all" \o "sqlalchemy.schema.MetaData.drop_all) are called, and it takes effect when using table reflection given a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object. However it does ****not****affect the operations present on the [Inspector](http://docs.sqlalchemy.org/en/rel_1_1/core/reflection.html" \l "sqlalchemy.engine.reflection.Inspector" \o "sqlalchemy.engine.reflection.Inspector) object, as the schema name is passed to these methods explicitly.

*New in version 1.1.*

5.2.6 Engine Disposal

The [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) refers to a connection pool, which means under normal circumstances, there are open database connections present while the [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) object is still resident in memory. When an [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) is garbage collected, its connection pool is no longer referred to by that [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine), and assuming none of its connections are still checked out, the pool and its connections will also be garbage collected, which has the effect of closing out the actual database connections as well. But otherwise, the [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) will hold onto open database connections assuming it uses the normally default pool implementation of [QueuePool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.QueuePool" \o "sqlalchemy.pool.QueuePool).

The [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) is intended to normally be a permanent fixture established up-front and maintained throughout the lifespan of an application. It is ****not**** intended to be created and disposed on a per-connection basis; it is instead a registry that maintains both a pool of connections as well as configurational information about the database and DBAPI in use, as well as some degree of internal caching of per-database resources.

However, there are many cases where it is desirable that all connection resources referred to by the [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) be completely closed out. It's generally not a good idea to rely on Python garbage collection for this to occur for these cases; instead, the [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) can be explicitly disposed using the [Engine.dispose()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.dispose" \o "sqlalchemy.engine.Engine.dispose) method. This disposes of the engine's underlying connection pool and replaces it with a new one that's empty. Provided that the [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) is discarded at this point and no longer used, all ****checked-in**** connections which it refers to will also be fully closed.

Valid use cases for calling [Engine.dispose()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.dispose" \o "sqlalchemy.engine.Engine.dispose) include:

* When a program wants to release any remaining checked-in connections held by the connection pool and expects to no longer be connected to that database at all for any future operations.
* When a program uses multiprocessing or fork(), and an [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) object is copied to the child process, [Engine.dispose()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.dispose" \o "sqlalchemy.engine.Engine.dispose) should be called so that the engine creates brand new database connections local to that fork. Database connections generally do ****not**** travel across process boundaries.
* Within test suites or multitenancy scenarios where many ad-hoc, short-lived [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) objects may be created and disposed.

Connections that are ****checked out**** are ****not**** discarded when the engine is disposed or garbage collected, as these connections are still strongly referenced elsewhere by the application. However, after [Engine.dispose()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.dispose" \o "sqlalchemy.engine.Engine.dispose) is called, those connections are no longer associated with that [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine); when they are closed, they will be returned to their now-orphaned connection pool which will ultimately be garbage collected, once all connections which refer to it are also no longer referenced anywhere. Since this process is not easy to control, it is strongly recommended that [Engine.dispose()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.dispose" \o "sqlalchemy.engine.Engine.dispose) is called only after all checked out connections are checked in or otherwise de-associated from their pool.

An alternative for applications that are negatively impacted by the [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) object's use of connection pooling is to disable pooling entirely. This typically incurs only a modest performance impact upon the use of new connections, and means that when a connection is checked in, it is entirely closed out and is not held in memory. See [Switching Pool Implementations](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "pool-switching) for guidelines on how to disable pooling.

5.2.7 Using the Threadlocal Execution Strategy

The "threadlocal" engine strategy is an optional feature which can be used by non-ORM applications to associate transactions with the current thread, such that all parts of the application can participate in that transaction implicitly without the need to explicitly reference a [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection).

**Note**

The "threadlocal" feature is generally discouraged. It's designed for a particular pattern of usage which is generally considered as a legacy pattern. It has ****no impact**** on the "thread safety" of SQLAlchemy components or one's application. It also should not be used when using an ORM [Session](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_api.html" \l "sqlalchemy.orm.session.Session" \o "sqlalchemy.orm.session.Session) object, 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) itself represents an ongoing transaction and itself handles the job of maintaining connection and transactional resources.

Enabling threadlocal is achieved as follows:

db = create\_engine('mysql://localhost/test', strategy='threadlocal')

The above [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) will now acquire a [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) using connection resources derived from a thread-local variable whenever [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.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) is called. This connection resource is maintained as long as it is referenced, which allows multiple points of an application to share a transaction while using connectionless execution:

**def** call\_operation1():

engine.execute("insert into users values (?, ?)", 1, "john")

**def** call\_operation2():

users.update(users.c.user\_id==5).execute(name='ed')

db.begin()**try**:

call\_operation1()

call\_operation2()

db.commit()**except**:

db.rollback()

Explicit execution can be mixed with connectionless execution by using the [Engine.connect()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.connect" \o "sqlalchemy.engine.Engine.connect) method to acquire a [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) that is not part of the threadlocal scope:

db.begin()conn = db.connect()**try**:

conn.execute(log\_table.insert(), message="Operation started")

call\_operation1()

call\_operation2()

db.commit()

conn.execute(log\_table.insert(), message="Operation succeeded")**except**:

db.rollback()

conn.execute(log\_table.insert(), message="Operation failed")**finally**:

conn.close()

To access the [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) that is bound to the threadlocal scope, call [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):

conn = db.contextual\_connect()call\_operation3(conn)conn.close()

Calling [close()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.close" \o "sqlalchemy.engine.Connection.close) on the "contextual" connection does not [release](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-release) its resources until all other usages of that resource are closed as well, including that any ongoing transactions are rolled back or committed.

5.2.8 Working with Raw DBAPI Connections

There are some cases where SQLAlchemy does not provide a genericized way at accessing some [DBAPI](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-dbapi) functions, such as calling stored procedures as well as dealing with multiple result sets. In these cases, it's just as expedient to deal with the raw DBAPI connection directly.

The most common way to access the raw DBAPI connection is to get it from an already present [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) object directly. It is present using the [Connection.connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.connection" \o "sqlalchemy.engine.Connection.connection) attribute:

connection = engine.connect()dbapi\_conn = connection.connection

The DBAPI connection here is actually a "proxied" in terms of the originating connection pool, however this is an implementation detail that in most cases can be ignored. As this DBAPI connection is still contained within the scope of an owning [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) object, it is best to make use of the [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) object for most features such as transaction control as well as calling the [Connection.close()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.close" \o "sqlalchemy.engine.Connection.close) method; if these operations are performed on the DBAPI connection directly, the owning [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) will not be aware of these changes in state.

To overcome the limitations imposed by the DBAPI connection that is maintained by an owning [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection), a DBAPI connection is also available without the need to procure a [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) first, using the [Engine.raw\_connection()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.raw_connection" \o "sqlalchemy.engine.Engine.raw_connection) method of [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine):

dbapi\_conn = engine.raw\_connection()

This DBAPI connection is again a "proxied" form as was the case before. The purpose of this proxying is now apparent, as when we call the .close() method of this connection, the DBAPI connection is typically not actually closed, but instead [released](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-released) back to the engine's connection pool:

dbapi\_conn.close()

While SQLAlchemy may in the future add built-in patterns for more DBAPI use cases, there are diminishing returns as these cases tend to be rarely needed and they also vary highly dependent on the type of DBAPI in use, so in any case the direct DBAPI calling pattern is always there for those cases where it is needed.

Some recipes for DBAPI connection use follow.

### Calling Stored Procedures

For stored procedures with special syntactical or parameter concerns, DBAPI-level [callproc](http://legacy.python.org/dev/peps/pep-0249/" \l "callproc) may be used:

connection = engine.raw\_connection()**try**:

cursor = connection.cursor()

cursor.callproc("my\_procedure", ['x', 'y', 'z'])

results = list(cursor.fetchall())

cursor.close()

connection.commit()**finally**:

connection.close()

### Multiple Result Sets

Multiple result set support is available from a raw DBAPI cursor using the [nextset](http://legacy.python.org/dev/peps/pep-0249/" \l "nextset) method:

connection = engine.raw\_connection()**try**:

cursor = connection.cursor()

cursor.execute("select \* from table1; select \* from table2")

results\_one = cursor.fetchall()

cursor.nextset()

results\_two = cursor.fetchall()

cursor.close()**finally**:

connection.close()

### 5.2.9 Registering New Dialects

The [create\_engine()](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "sqlalchemy.create_engine" \o "sqlalchemy.create_engine) function call locates the given dialect using setuptools entrypoints. These entry points can be established for third party dialects within the setup.py script. For example, to create a new dialect "foodialect://", the steps are as follows:

Create a package called foodialect.

The package should have a module containing the dialect class, which is typically a subclass of [sqlalchemy.engine.default.DefaultDialect](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.default.DefaultDialect" \o "sqlalchemy.engine.default.DefaultDialect). In this example let's say it's called FooDialect and its module is accessed via foodialect.dialect.

The entry point can be established in setup.py as follows:

entry\_points="""[sqlalchemy.dialects]foodialect = foodialect.dialect:FooDialect"""

If the dialect is providing support for a particular DBAPI on top of an existing SQLAlchemy-supported database, the name can be given including a database-qualification. For example, if FooDialect were in fact a MySQL dialect, the entry point could be established like this:

entry\_points="""[sqlalchemy.dialects]mysql.foodialect = foodialect.dialect:FooDialect"""

The above entrypoint would then be accessed as create\_engine("mysql+foodialect://").

### Registering Dialects In-Process

SQLAlchemy also allows a dialect to be registered within the current process, bypassing the need for separate installation. Use the register() function as follows:

**from** **sqlalchemy.dialects** **import** registryregistry.register("mysql.foodialect", "myapp.dialect", "MyMySQLDialect")

The above will respond to create\_engine("mysql+foodialect://") and load the MyMySQLDialect class from the myapp.dialect module.

*New in version 0.8.*

5.2.10 Connection / Engine API

*class*sqlalchemy.engine.**Connection**(*engine*, *connection=None*, *close\_with\_result=False*, *\_branch\_from=None*, *\_execution\_options=None*, *\_dispatch=None*, *\_has\_events=None*)

Bases: [sqlalchemy.engine.Connectable](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connectable" \o "sqlalchemy.engine.Connectable)

Provides high-level functionality for a wrapped DB-API connection.

Provides execution support for string-based SQL statements as well as [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement), [Compiled](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Compiled" \o "sqlalchemy.engine.interfaces.Compiled) and [DefaultGenerator](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.DefaultGenerator" \o "sqlalchemy.schema.DefaultGenerator) objects. Provides a [begin()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.begin" \o "sqlalchemy.engine.Connection.begin) method to return [Transaction](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction" \o "sqlalchemy.engine.Transaction) objects.

The Connection object is ****not**** thread-safe. While a Connection can be shared among threads using properly synchronized access, it is still possible that the underlying DBAPI connection may not support shared access between threads. Check the DBAPI documentation for details.

The Connection object represents a single dbapi connection checked out from the connection pool. In this state, the connection pool has no affect upon the connection, including its expiration or timeout state. For the connection pool to properly manage connections, connections should be returned to the connection pool (i.e. connection.close()) whenever the connection is not in use.

**\_\_init\_\_**(*engine*, *connection=None*, *close\_with\_result=False*, *\_branch\_from=None*, *\_execution\_options=None*, *\_dispatch=None*, *\_has\_events=None*)

Construct a new Connection.

The constructor here is not public and is only called only by an [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine). See [Engine.connect()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.connect" \o "sqlalchemy.engine.Engine.connect) and [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) methods.

**begin**()

Begin a transaction and return a transaction handle.

The returned object is an instance of [Transaction](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction" \o "sqlalchemy.engine.Transaction). This object represents the "scope" of the transaction, which completes when either the [Transaction.rollback()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction.rollback" \o "sqlalchemy.engine.Transaction.rollback) or [Transaction.commit()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction.commit" \o "sqlalchemy.engine.Transaction.commit) method is called.

Nested calls to [begin()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.begin" \o "sqlalchemy.engine.Connection.begin) on the same [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) will return new [Transaction](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction" \o "sqlalchemy.engine.Transaction) objects that represent an emulated transaction within the scope of the enclosing transaction, that is:

trans = conn.begin() *# outermost transaction*trans2 = conn.begin() *# "nested"*trans2.commit() *# does nothing*trans.commit() *# actually commits*

Calls to [Transaction.commit()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction.commit" \o "sqlalchemy.engine.Transaction.commit) only have an effect when invoked via the outermost [Transaction](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction" \o "sqlalchemy.engine.Transaction) object, though the[Transaction.rollback()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction.rollback" \o "sqlalchemy.engine.Transaction.rollback) method of any of the [Transaction](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction" \o "sqlalchemy.engine.Transaction) objects will roll back the transaction.

See also:

[Connection.begin\_nested()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.begin_nested" \o "sqlalchemy.engine.Connection.begin_nested) - use a SAVEPOINT

[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) - use a two phase /XID transaction

[Engine.begin()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.begin" \o "sqlalchemy.engine.Engine.begin) - context manager available from [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine).

**begin\_nested**()

Begin a nested transaction and return a transaction handle.

The returned object is an instance of [NestedTransaction](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.NestedTransaction" \o "sqlalchemy.engine.NestedTransaction).

Nested transactions require SAVEPOINT support in the underlying database. Any transaction in the hierarchy may commit and rollback, however the outermost transaction still controls the overall commit or rollback of the transaction of a whole.

See also [Connection.begin()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.begin" \o "sqlalchemy.engine.Connection.begin), [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).

**begin\_twophase**(*xid=None*)

Begin a two-phase or XA transaction and return a transaction handle.

The returned object is an instance of [TwoPhaseTransaction](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.TwoPhaseTransaction" \o "sqlalchemy.engine.TwoPhaseTransaction), which in addition to the methods provided by [Transaction](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction" \o "sqlalchemy.engine.Transaction), also provides a[prepare()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.TwoPhaseTransaction.prepare" \o "sqlalchemy.engine.TwoPhaseTransaction.prepare) method.

|  |  |
| --- | --- |
| **Parameters:** | ****xid**** – the two phase transaction id. If not supplied, a random id will be generated. |

See also [Connection.begin()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.begin" \o "sqlalchemy.engine.Connection.begin), [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).

**close**()

Close this [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection).

This results in a release of the underlying database resources, that is, the DBAPI connection referenced internally. The DBAPI connection is typically restored back to the connection-holding [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool) referenced by the [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) that produced this [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection). Any transactional state present on the DBAPI connection is also unconditionally released via the DBAPI connection's rollback() method, regardless of any [Transaction](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction" \o "sqlalchemy.engine.Transaction) object that may be outstanding with regards to this [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection).

After [close()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.close" \o "sqlalchemy.engine.Connection.close) is called, the [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) is permanently in a closed state, and will allow no further operations.

**closed**

Return True if this connection is closed.

**connect**()

Returns a branched version of this [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection).

The [Connection.close()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.close" \o "sqlalchemy.engine.Connection.close) method on the returned [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) can be called and this [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) will remain open.

This method provides usage symmetry with [Engine.connect()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.connect" \o "sqlalchemy.engine.Engine.connect), including for usage with context managers.

**connection**

The underlying DB-API connection managed by this Connection.

**See also**

[Working with Raw DBAPI Connections](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "dbapi-connections)

**contextual\_connect**(*\*\*kwargs*)

Returns a branched version of this [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection).

The [Connection.close()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.close" \o "sqlalchemy.engine.Connection.close) method on the returned [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) can be called and this [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) will remain open.

This method provides usage symmetry with [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), including for usage with context managers.

**default\_isolation\_level**

The default isolation level assigned to this [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection).

This is the isolation level setting that the [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) has when first procured via the [Engine.connect()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.connect" \o "sqlalchemy.engine.Engine.connect) method. This level stays in place until the[Connection.execution\_options.isolation\_level](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.execution_options.params.isolation_level" \o "sqlalchemy.engine.Connection.execution_options) is used to change the setting on a per-[Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) basis.

Unlike [Connection.get\_isolation\_level()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.get_isolation_level" \o "sqlalchemy.engine.Connection.get_isolation_level), this attribute is set ahead of time from the first connection procured by the dialect, so SQL query is not invoked when this accessor is called.

*New in version 0.9.9.*

**See also**

[Connection.get\_isolation\_level()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.get_isolation_level" \o "sqlalchemy.engine.Connection.get_isolation_level) - view current level

[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) - set per [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) isolation level

[Connection.execution\_options.isolation\_level](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.execution_options.params.isolation_level" \o "sqlalchemy.engine.Connection.execution_options) - set per [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) isolation level

**detach**()

Detach the underlying DB-API connection from its connection pool.

E.g.:

**with** engine.connect() **as** conn:

conn.detach()

conn.execute("SET search\_path TO schema1, schema2")

*# work with connection*

*# connection is fully closed (since we used "with:", can# also call .close())*

This [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) instance will remain usable. When closed (or exited from a context manager context as above), the DB-API connection will be literally closed and not returned to its originating pool.

This method can be used to insulate the rest of an application from a modified state on a connection (such as a transaction isolation level or similar).

**execute**(*object*, *\*multiparams*, *\*\*params*)

Executes a SQL statement construct and returns a [ResultProxy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy" \o "sqlalchemy.engine.ResultProxy).

|  |  |
| --- | --- |
| **Parameters:** | * ****object**** –The statement to be executed. May be one of:   + a plain string   + any [ClauseElement](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.ClauseElement" \o "sqlalchemy.sql.expression.ClauseElement) construct that is also a subclass of [Executable](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable" \o "sqlalchemy.sql.expression.Executable), such as a [select()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.select" \o "sqlalchemy.sql.expression.select) construct   + a [FunctionElement](http://docs.sqlalchemy.org/en/rel_1_1/core/functions.html" \l "sqlalchemy.sql.functions.FunctionElement" \o "sqlalchemy.sql.functions.FunctionElement), such as that generated by [func](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.func" \o "sqlalchemy.sql.expression.func), will be automatically wrapped in a SELECT statement, which is then executed.   + a [DDLElement](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.DDLElement" \o "sqlalchemy.schema.DDLElement) object   + a [DefaultGenerator](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.DefaultGenerator" \o "sqlalchemy.schema.DefaultGenerator) object   + a [Compiled](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Compiled" \o "sqlalchemy.engine.interfaces.Compiled) object * ****\*multiparams/\*\*params –****represent bound parameter values to be used in the execution. Typically, the format is either a collection of one or more dictionaries passed to \*multiparams:   conn.execute(  table.insert(),  {"id":1, "value":"v1"},  {"id":2, "value":"v2"})  …or individual key/values interpreted by \*\*params:  conn.execute(  table.insert(), id=1, value="v1")  In the case that a plain SQL string is passed, and the underlying DBAPI accepts positional bind parameters, a collection of tuples or individual values in \*multiparams may be passed:  conn.execute(  "INSERT INTO table (id, value) VALUES (?, ?)",  (1, "v1"), (2, "v2"))  conn.execute(  "INSERT INTO table (id, value) VALUES (?, ?)",  1, "v1")  Note above, the usage of a question mark "?" or other symbol is contingent upon the "paramstyle" accepted by the DBAPI in use, which may be any of "qmark", "named", "pyformat", "format", "numeric". See [pep-249](http://www.python.org/dev/peps/pep-0249/) for details on paramstyle.  To execute a textual SQL statement which uses bound parameters in a DBAPI-agnostic way, use the [text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text) construct. |

**execution\_options**(*\*\*opt*)

Set non-SQL options for the connection which take effect during execution.

为执行期间生效的连接设置非SQL选项。

The method returns a copy of this [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) which references the same underlying DBAPI connection, but also defines the given execution options which will take effect for a call to [execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.execute" \o "sqlalchemy.engine.Connection.execute). As the new [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) references the same underlying resource, it's usually a good idea to ensure that the copies will be discarded immediately, which is implicit if used as in:

该方法返回引用相同底层DBAPI连接的此Connection的副本，还可以定义将对execute（）的调用生效的给定执行选项。 由于新的连接引用了相同的底层资源，所以通常最好确保副本将立即被丢弃，如果使用的话，这是隐式的：

result = connection.execution\_options(stream\_results=**True**).\

execute(stmt)

Note that any key/value can 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), and it will be stored in the \_execution\_options dictionary of the [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection). It is suitable for usage by end-user schemes to communicate with event listeners, for example.

请注意，任何键/值都可以传递给Connection.execution\_options（），它将被存储在Connection的\_execution\_options字典中。 例如，它适用于最终用户方案与事件侦听器进行通信。

The keywords that are currently recognized by SQLAlchemy itself include all those listed under [Executable.execution\_options()](http://docs.sqlalchemy.org/en/rel_1_1/core/selectable.html" \l "sqlalchemy.sql.expression.Executable.execution_options" \o "sqlalchemy.sql.expression.Executable.execution_options), as well as others that are specific to [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection).

当前由SQLAlchemy识别的关键字包括Executable.execution\_options（）中列出的所有关键字，以及特定于Connection的其他关键字。

|  |  |
| --- | --- |
| **Parameters:** | * ****autocommit**** – Available on: Connection, statement. When True, a COMMIT will be invoked after execution when executed in 'autocommit' mode, i.e. when an explicit transaction is not begun on the connection. Note that DBAPI connections by default are always in a transaction - SQLAlchemy uses rules applied to different kinds of statements to determine if COMMIT will be invoked in order to provide its "autocommit" feature. Typically, all INSERT/UPDATE/DELETE statements as well as CREATE/DROP statements have autocommit behavior enabled; SELECT constructs do not. Use this option when invoking a SELECT or other specific SQL construct where COMMIT is desired (typically when calling stored procedures and such), and an explicit transaction is not in progress. * ****compiled\_cache –****Available on: Connection. A dictionary where [Compiled](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Compiled" \o "sqlalchemy.engine.interfaces.Compiled) objects will be cached when the [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) compiles a clause expression into a [Compiled](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Compiled" \o "sqlalchemy.engine.interfaces.Compiled) object. It is the user's responsibility to manage the size of this dictionary, which will have keys corresponding to the dialect, clause element, the column names within the VALUES or SET clause of an INSERT or UPDATE, as well as the "batch" mode for an INSERT or UPDATE statement. The format of this dictionary is not guaranteed to stay the same in future releases.   Note that the ORM makes use of its own "compiled" caches for some operations, including flush operations. The caching used by the ORM internally supersedes a cache dictionary specified here.  **isolation\_level –**Available on: [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection). Set the transaction isolation level for the lifespan of this [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) object (*not* the underlying DBAPI connection, for which the level is reset to its original setting upon termination of this [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) object).  Valid values include those string values accepted by 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 passed to [create\_engine()](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "sqlalchemy.create_engine" \o "sqlalchemy.create_engine). These levels are semi-database specific; see individual dialect documentation for valid levels.  Note that this option necessarily affects the underlying DBAPI connection for the lifespan of the originating [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection), and is not per-execution. This setting is not removed until the underlying DBAPI connection is returned to the connection pool, i.e. the [Connection.close()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.close" \o "sqlalchemy.engine.Connection.close) method is called.  **Warning**  The isolation\_level execution option should ****not**** be used when a transaction is already established, that is, the [Connection.begin()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.begin" \o "sqlalchemy.engine.Connection.begin) method or similar has been called. A database cannot change the isolation level on a transaction in progress, and different DBAPIs and/or SQLAlchemy dialects may implicitly roll back or commit the transaction, or not affect the connection at all.  *Changed in version 0.9.9:*A warning is emitted when the isolation\_level execution option is used after a transaction has been started with [Connection.begin()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.begin" \o "sqlalchemy.engine.Connection.begin) or similar.  **Note**  The isolation\_level execution option is implicitly reset if the [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) is invalidated, e.g. via 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, or if a disconnection error occurs. The new connection produced after the invalidation will not have the isolation level re-applied to it automatically.  **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) - set per [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) isolation level  [Connection.get\_isolation\_level()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.get_isolation_level" \o "sqlalchemy.engine.Connection.get_isolation_level) - view current level  [SQLite Transaction Isolation](http://docs.sqlalchemy.org/en/rel_1_1/dialects/sqlite.html" \l "sqlite-isolation-level)  [PostgreSQL Transaction Isolation](http://docs.sqlalchemy.org/en/rel_1_1/dialects/postgresql.html" \l "postgresql-isolation-level)  [MySQL Transaction Isolation](http://docs.sqlalchemy.org/en/rel_1_1/dialects/mysql.html" \l "mysql-isolation-level)  [SQL Server Transaction Isolation](http://docs.sqlalchemy.org/en/rel_1_1/dialects/mssql.html" \l "mssql-isolation-level)  [Setting Transaction Isolation Levels](http://docs.sqlalchemy.org/en/rel_1_1/orm/session_transaction.html" \l "session-transaction-isolation) - for the ORM   * ****no\_parameters –****When True, if the final parameter list or dictionary is totally empty, will invoke the statement on the cursor as cursor.execute(statement), not passing the parameter collection at all. Some DBAPIs such as psycopg2 and mysql-python consider percent signs as significant only when parameters are present; this option allows code to generate SQL containing percent signs (and possibly other characters) that is neutral regarding whether it's executed by the DBAPI or piped into a script that's later invoked by command line tools.   *New in version 0.7.6.*   * ****stream\_results**** – Available on: Connection, statement. Indicate to the dialect that results should be "streamed" and not pre-buffered, if possible. This is a limitation of many DBAPIs. The flag is currently understood only by the psycopg2, mysqldb and pymysql dialects. * ****schema\_translate\_map –****Available on: Connection, Engine. A dictionary mapping schema names to schema names, that will be applied to the [Table.schema](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.params.schema" \o "sqlalchemy.schema.Table) element of each[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) encountered when SQL or DDL expression elements are compiled into strings; the resulting schema name will be converted based on presence in the map of the original name.   *New in version 1.1.*  **See also**  [Translation of Schema Names](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "schema-translating) |

**get\_isolation\_level**()

Return the current isolation level assigned to this [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection).

返回分配给此Connection的当前隔离级别。

This will typically be the default isolation level as determined by the dialect, unless if the [Connection.execution\_options.isolation\_level](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.execution_options.params.isolation_level" \o "sqlalchemy.engine.Connection.execution_options)feature has been used to alter the isolation level on a per-[Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) basis.

这通常是由方言确定的默认隔离级别，除非已使用Connection.execution\_options.isolation\_levelfeature来更改每个连接的隔离级别。

This attribute will typically perform a live SQL operation in order to procure the current isolation level, so the value returned is the actual level on the underlying DBAPI connection regardless of how this state was set. Compare to the [Connection.default\_isolation\_level](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.default_isolation_level" \o "sqlalchemy.engine.Connection.default_isolation_level) accessor which returns the dialect-level setting without performing a SQL query.

此属性通常将执行实时SQL操作以获取当前隔离级别，因此返回的值是底层DBAPI连接上的实际级别，无论此状态如何设置。 与Connection.default\_isolation\_level访问器进行比较，该访问器返回方言级设置，而不执行SQL查询。

*New in version 0.9.9.*

**See also**

[Connection.default\_isolation\_level](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.default_isolation_level" \o "sqlalchemy.engine.Connection.default_isolation_level) - view default level

[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) - set per [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) isolation level

[Connection.execution\_options.isolation\_level](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.execution_options.params.isolation_level" \o "sqlalchemy.engine.Connection.execution_options) - set per [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) isolation level

**in\_transaction**()

Return True if a transaction is in progress.

如果事务正在进行中，则返回True。

**info**

Info dictionary associated with the underlying DBAPI connection referred to by this [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection), allowing user-defined data to be associated with the connection.

信息字典与此连接引用的底层DBAPI连接相关联，允许用户定义的数据与连接相关联。

The data here will follow along with the DBAPI connection including after it is returned to the connection pool and used again in subsequent instances of [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection).

这里的数据将跟随DBAPI连接，包括在返回到连接池并在连接的后续实例中再次使用。

**invalidate**(*exception=None*)

Invalidate the underlying DBAPI connection associated with this [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection).

The underlying DBAPI connection is literally closed (if possible), and is discarded. Its source connection pool will typically lazily create a new connection to replace it.

Upon the next use (where "use" typically means using the [Connection.execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.execute" \o "sqlalchemy.engine.Connection.execute) method or similar), this [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) will attempt to procure a new DBAPI connection using the services of the [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool) as a source of connectivity (e.g. a "reconnection").

If a transaction was in progress (e.g. the [Connection.begin()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.begin" \o "sqlalchemy.engine.Connection.begin) method has been called) when [Connection.invalidate()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.invalidate" \o "sqlalchemy.engine.Connection.invalidate) method is called, at the DBAPI level all state associated with this transaction is lost, as the DBAPI connection is closed. The [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) will not allow a reconnection to proceed until the [Transaction](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction" \o "sqlalchemy.engine.Transaction) object is ended, by calling the [Transaction.rollback()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction.rollback" \o "sqlalchemy.engine.Transaction.rollback) method; until that point, any attempt at continuing to use the [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) will raise an [InvalidRequestError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.InvalidRequestError" \o "sqlalchemy.exc.InvalidRequestError). This is to prevent applications from accidentally continuing an ongoing transactional operations despite the fact that the transaction has been lost due to an invalidation.

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, just like auto-invalidation, will at the connection pool level invoke the [PoolEvents.invalidate()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.PoolEvents.invalidate" \o "sqlalchemy.events.PoolEvents.invalidate)event.

**See also**

[More on Invalidation](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "pool-connection-invalidation)

**invalidated**

Return True if this connection was invalidated.

**run\_callable**(*callable\_*, *\*args*, *\*\*kwargs*)

Given a callable object or function, execute it, passing a [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) as the first argument.

The given \*args and \*\*kwargs are passed subsequent to the [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) argument.

This function, along with [Engine.run\_callable()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.run_callable" \o "sqlalchemy.engine.Engine.run_callable), allows a function to be run with a [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) or [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) object without the need to know which one is being dealt with.

**scalar**(*object*, *\*multiparams*, *\*\*params*)

Executes and returns the first column of the first row.

The underlying result/cursor is closed after execution.

**schema\_for\_object***= <sqlalchemy.sql.schema.\_SchemaTranslateMap object>*

Return the ".schema" attribute for an object.

Used for [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table), [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) and similar objects, and takes into account the [Connection.execution\_options.schema\_translate\_map](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.execution_options.params.schema_translate_map" \o "sqlalchemy.engine.Connection.execution_options)parameter.

*New in version 1.1.*

**See also**

[Translation of Schema Names](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "schema-translating)

**transaction**(*callable\_*, *\*args*, *\*\*kwargs*)

Execute the given function within a transaction boundary.

The function is passed this [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) as the first argument, followed by the given \*args and \*\*kwargs, e.g.:

**def** do\_something(conn, x, y):

conn.execute("some statement", {'x':x, 'y':y})

conn.transaction(do\_something, 5, 10)

The operations inside the function are all invoked within the context of a single [Transaction](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction" \o "sqlalchemy.engine.Transaction). Upon success, the transaction is committed. If an exception is raised, the transaction is rolled back before propagating the exception.

**Note**

The [transaction()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.transaction" \o "sqlalchemy.engine.Connection.transaction) method is superseded by the usage of the Python with: statement, which can be used with [Connection.begin()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.begin" \o "sqlalchemy.engine.Connection.begin):

**with** conn.begin():

conn.execute("some statement", {'x':5, 'y':10})

As well as with [Engine.begin()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.begin" \o "sqlalchemy.engine.Engine.begin):

**with** engine.begin() **as** conn:

conn.execute("some statement", {'x':5, 'y':10})

See also:

[Engine.begin()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.begin" \o "sqlalchemy.engine.Engine.begin) - engine-level transactional context

[Engine.transaction()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.transaction" \o "sqlalchemy.engine.Engine.transaction) - engine-level version of [Connection.transaction()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.transaction" \o "sqlalchemy.engine.Connection.transaction)

*class*sqlalchemy.engine.**Connectable**

Interface for an object which supports execution of SQL constructs.

The two implementations of [Connectable](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connectable" \o "sqlalchemy.engine.Connectable) are [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) and [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine).

Connectable must also implement the 'dialect' member which references a [Dialect](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Dialect" \o "sqlalchemy.engine.interfaces.Dialect) instance.

**connect**(*\*\*kwargs*)

Return a [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) object.

Depending on context, this may be self if this object is already an instance of [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection), or a newly procured [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) if this object is an instance of [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine).

**contextual\_connect**()

Return a [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) object which may be part of an ongoing context.

Depending on context, this may be self if this object is already an instance of [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection), or a newly procured [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) if this object is an instance of [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine).

**create**(*entity*, *\*\*kwargs*)

Emit CREATE statements for the given schema entity.

*Deprecated since version 0.7:*Use the create() method on the given schema object directly, i.e. [Table.create()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.create" \o "sqlalchemy.schema.Table.create), [Index.create()](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.Index.create" \o "sqlalchemy.schema.Index.create), [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)

**drop**(*entity*, *\*\*kwargs*)

Emit DROP statements for the given schema entity.

*Deprecated since version 0.7:*Use the drop() method on the given schema object directly, i.e. [Table.drop()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.drop" \o "sqlalchemy.schema.Table.drop), [Index.drop()](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.Index.drop" \o "sqlalchemy.schema.Index.drop), [MetaData.drop\_all()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData.drop_all" \o "sqlalchemy.schema.MetaData.drop_all)

**execute**(*object*, *\*multiparams*, *\*\*params*)

Executes the given construct and returns a [ResultProxy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy" \o "sqlalchemy.engine.ResultProxy).

**scalar**(*object*, *\*multiparams*, *\*\*params*)

Executes and returns the first column of the first row.

The underlying cursor is closed after execution.

*class*sqlalchemy.engine.**CreateEnginePlugin**(*url*, *kwargs*)

A set of hooks intended to augment the construction of an [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) object based on entrypoint names in a URL.

一组钩子旨在基于URL中的入口名称来增加Engine对象的构造。

The purpose of [CreateEnginePlugin](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.CreateEnginePlugin" \o "sqlalchemy.engine.CreateEnginePlugin) is to allow third-party systems to apply engine, pool and dialect level event listeners without the need for the target application to be modified; instead, the plugin names can be added to the database URL. Target applications for [CreateEnginePlugin](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.CreateEnginePlugin" \o "sqlalchemy.engine.CreateEnginePlugin) include:

CreateEnginePlugin的目的是允许第三方系统应用引擎，池和方言级事件侦听器，而不需要修改目标应用程序; 而是可以将插件名称添加到数据库URL。 CreateEnginePlugin的目标应用包括：

* connection and SQL performance tools, e.g. which use events to track number of checkouts and/or time spent with statements连接和SQL性能工具，例如 其使用事件来跟踪结帐次数和/或花费在语句上的时间
* connectivity plugins such as proxies连接插件，如代理

Plugins are registered using entry points in a similar way as that of dialects:

插件使用入口点以与方言相似的方式进行注册：

entry\_points={

'sqlalchemy.plugins': [

'myplugin = myapp.plugins:MyPlugin'

]

A plugin that uses the above names would be invoked from a database URL as in:

使用以上名称的插件将从数据库URL调用，如：

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

engine = create\_engine(

"mysql+pymysql://scott:tiger@localhost/test?plugin=myplugin")

The plugin argument supports multiple instances, so that a URL may specify multiple plugins; they are loaded in the order stated in the URL:

插件参数支持多个实例，以便URL可以指定多个插件; 它们按照URL中的顺序加载：

engine = create\_engine(

"mysql+pymysql://scott:tiger@localhost/"

"test?plugin=plugin\_one&plugin=plugin\_twp&plugin=plugin\_three")

A plugin can receive additional arguments from the URL string as well as from the keyword arguments passed to [create\_engine()](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "sqlalchemy.create_engine" \o "sqlalchemy.create_engine). The [URL](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "sqlalchemy.engine.url.URL" \o "sqlalchemy.engine.url.URL) object and the keyword dictionary are passed to the constructor so that these arguments can be extracted from the url's URL.query collection as well as from the dictionary:

插件可以从URL字符串以及传递给create\_engine() 的关键字参数接收其他参数。 URL对象和关键字字典传递给构造函数，以便可以从url的URL.query集合以及字典中提取这些参数：

**class** **MyPlugin**(CreateEnginePlugin):

**def** \_\_init\_\_(self, url, kwargs):

self.my\_argument\_one = url.query.pop('my\_argument\_one')

self.my\_argument\_two = url.query.pop('my\_argument\_two')

self.my\_argument\_three = kwargs.pop('my\_argument\_three', **None**)

Arguments like those illustrated above would be consumed from the following:

以上所示的参数将从以下内容中消耗：

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

engine = create\_engine(

"mysql+pymysql://scott:tiger@localhost/"

"test?plugin=myplugin&my\_argument\_one=foo&my\_argument\_two=bar",

my\_argument\_three='bat')

The URL and dictionary are used for subsequent setup of the engine as they are, so the plugin can modify their arguments in-place. Arguments that are only understood by the plugin should be popped or otherwise removed so that they aren't interpreted as erroneous arguments afterwards.

URL和字典用于引擎的后续设置，因此插件可以原位修改其参数。 插件只能理解的参数应弹出或以其他方式删除，以便之后不会将其解释为错误的参数。

When the engine creation process completes and produces the [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) object, it is again passed to the plugin via the[CreateEnginePlugin.engine\_created()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.CreateEnginePlugin.engine_created" \o "sqlalchemy.engine.CreateEnginePlugin.engine_created) hook. In this hook, additional changes can be made to the engine, most typically involving setup of events (e.g. those defined in [Core Events](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html)).

当引擎创建过程完成并生成Engine对象时，它将再次通过CreateEnginePlugin.engine\_created() 钩子传递给插件。 在这个钩子中，可以对引擎进行额外的改变，其中通常涉及事件的设置（例如在核心事件中定义的事件）。

*New in version 1.1.*

**\_\_init\_\_**(*url*, *kwargs*)

Contruct a new [CreateEnginePlugin](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.CreateEnginePlugin" \o "sqlalchemy.engine.CreateEnginePlugin).

The plugin object is instantiated individually for each call to [create\_engine()](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "sqlalchemy.create_engine" \o "sqlalchemy.create_engine). A single [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) will be passed to the [CreateEnginePlugin.engine\_created()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.CreateEnginePlugin.engine_created" \o "sqlalchemy.engine.CreateEnginePlugin.engine_created) method corresponding to this URL.

每个对create\_engine() 的调用都会单独实例化插件对象。 单个引擎将传递给与此URL相对应的CreateEnginePlugin.engine\_created() 方法。

|  |  |
| --- | --- |
| **Parameters:** | * ****url**** – the [URL](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "sqlalchemy.engine.url.URL" \o "sqlalchemy.engine.url.URL) object. The plugin should inspect what it needs here as well as remove its custom arguments from the URL.query collection. The URL can be modified in-place in any other way as well.URL对象。 插件应该检查它在这里需要什么，以及从URL.query集合中删除它的自定义参数。 URL可以以任何其他方式原位进行修改。 * ****kwargs**** – The keyword arguments passed to :func`.create\_engine`. The plugin can read and modify this dictionary in-place, to affect the ultimate arguments used to create the engine. It should remove its custom arguments from the dictionary as well.关键字参数传递给：func`.create\_engine`。 该插件可以原位读取和修改该字典，以影响用于创建引擎的最终参数。 它也应该从字典中删除其自定义参数。 |

**engine\_created**(*engine*)

Receive the [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) object when it is fully constructed.

完全构造后接收Engine对象。

The plugin may make additional changes to the engine, such as registering engine or connection pool events.

该插件可能会对引擎进行其他更改，例如注册引擎或连接池事件。

**handle\_dialect\_kwargs**(*dialect\_cls*, *dialect\_args*)

parse and modify dialect kwargs

**handle\_pool\_kwargs**(*pool\_cls*, *pool\_args*)

parse and modify pool kwargs

*class*sqlalchemy.engine.**Engine**(*pool*, *dialect*, *url*, *logging\_name=None*, *echo=None*, *proxy=None*, *execution\_options=None*)

Bases: [sqlalchemy.engine.Connectable](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connectable" \o "sqlalchemy.engine.Connectable), [sqlalchemy.log.Identified](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.log.Identified" \o "sqlalchemy.log.Identified)

Connects a [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool) and [Dialect](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Dialect" \o "sqlalchemy.engine.interfaces.Dialect) together to provide a source of database connectivity and behavior.

将池和方言连接在一起，以提供数据库连接和行为的来源。

An [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) object is instantiated publicly using the [create\_engine()](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "sqlalchemy.create_engine" \o "sqlalchemy.create_engine) function.

使用create\_engine() 函数公开实例引擎对象。

See also:

[Engine Configuration](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html)

[Working with Engines and Connections](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html)

**begin**(*close\_with\_result=False*)

Return a context manager delivering a [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) with a [Transaction](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction" \o "sqlalchemy.engine.Transaction) established.

返回一个上下文管理器，交付一个建立的事务的连接。

E.g.:

**with** engine.begin() **as** conn:

conn.execute("insert into table (x, y, z) values (1, 2, 3)")

conn.execute("my\_special\_procedure(5)")

Upon successful operation, the [Transaction](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction" \o "sqlalchemy.engine.Transaction) is committed. If an error is raised, the [Transaction](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction" \o "sqlalchemy.engine.Transaction) is rolled back.

The close\_with\_result flag is normally False, and indicates that the [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) will be closed when the operation is complete. When set to True, it indicates the [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) is in "single use" mode, where the [ResultProxy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy" \o "sqlalchemy.engine.ResultProxy) returned by the first call to [Connection.execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.execute" \o "sqlalchemy.engine.Connection.execute) will close the [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) when that [ResultProxy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy" \o "sqlalchemy.engine.ResultProxy) has exhausted all result rows.

*New in version 0.7.6.*

See also:

[Engine.connect()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.connect" \o "sqlalchemy.engine.Engine.connect) - procure a [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) from an [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine).

[Connection.begin()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.begin" \o "sqlalchemy.engine.Connection.begin) - start a [Transaction](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction" \o "sqlalchemy.engine.Transaction) for a particular [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection).

**connect**(*\*\*kwargs*)

Return a new [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) object.

The [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) object is a facade that uses a DBAPI connection internally in order to communicate with the database. This connection is procured from the connection-holding [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool) referenced by this [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine). When the [close()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.close" \o "sqlalchemy.engine.Connection.close) method of the [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) object is called, the underlying DBAPI connection is then returned to the connection pool, where it may be used again in a subsequent call to [connect()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.connect" \o "sqlalchemy.engine.Engine.connect).

**contextual\_connect**(*close\_with\_result=False*, *\*\*kwargs*)

Return a [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) object which may be part of some ongoing context.

By default, this method does the same thing as [Engine.connect()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.connect" \o "sqlalchemy.engine.Engine.connect). Subclasses of [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) may override this method to provide contextual behavior.

|  |  |
| --- | --- |
| **Parameters:** | ****close\_with\_result**** – When True, the first [ResultProxy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy" \o "sqlalchemy.engine.ResultProxy) created by the [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) will call the [Connection.close()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.close" \o "sqlalchemy.engine.Connection.close) method of that connection as soon as any pending result rows are exhausted. This is used to supply the "connectionless execution" behavior provided by the [Engine.execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.execute" \o "sqlalchemy.engine.Engine.execute)method. |

**dispose**()

Dispose of the connection pool used by this [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine).

处理此引擎使用的连接池。

This has the effect of fully closing all ****currently checked in**** database connections. Connections that are still checked out will ****not**** be closed, however they will no longer be associated with this [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine), so when they are closed individually, eventually the [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool) which they are associated with will be garbage collected and they will be closed out fully, if not already closed on checkin.

这具有完全关闭所有当前检入的数据库连接的效果。 仍然签出的连接不会关闭，但是它们将不再与此引擎关联，因此当它们单独关闭时，最终与它们相关联的池将被垃圾回收，并且将完全关闭，如果 签到时尚未关闭

A new connection pool is created immediately after the old one has been disposed. This new pool, like all SQLAlchemy connection pools, does not make any actual connections to the database until one is first requested, so as long as the [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) isn't used again, no new connections will be made.

在旧的连接池被丢弃之后立即创建一个新的连接池。 这个新的池像所有的SQLAlchemy连接池一样，直到首次被请求之前才能与数据库建立任何实际的连接，所以只要引擎不再被使用，就不会有新的连接。

**See also**

[Engine Disposal](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "engine-disposal)

**driver**

Driver name of the [Dialect](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Dialect" \o "sqlalchemy.engine.interfaces.Dialect) in use by this [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine).

该引擎正在使用的方言的驱动程序名称。

**execute**(*statement*, *\*multiparams*, *\*\*params*)

Executes the given construct and returns a [ResultProxy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy" \o "sqlalchemy.engine.ResultProxy).

执行给定的结构并返回一个ResultProxy。

The arguments are the same as those used by [Connection.execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.execute" \o "sqlalchemy.engine.Connection.execute).

参数与Connection.execute() 使用的参数相同。

Here, a [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) is acquired using the [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, and the statement executed with that connection. The returned [ResultProxy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy" \o "sqlalchemy.engine.ResultProxy) is flagged such that when the [ResultProxy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy" \o "sqlalchemy.engine.ResultProxy) is exhausted and its underlying cursor is closed, the [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) created here will also be closed, which allows its associated DBAPI connection resource to be returned to the connection pool.

这里，使用contextual\_connect() 方法获取连接，并使用该连接执行语句。 返回的ResultProxy被标记为使得当ResultProxy耗尽并且其底层游标关闭时，此处创建的连接也将被关闭，这允许将其关联的DBAPI连接资源返回到连接池。

**execution\_options**(*\*\*opt*)

Return a new [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) that will provide [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) objects with the given execution options.

返回一个新的引擎，它将为Connection对象提供给定的执行选项。

The returned [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) remains related to the original [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) in that it shares the same connection pool and other state:

返回的引擎仍然与原始引擎相关，因为它共享相同的连接池和其他状态：

* The [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool) used by the new [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) is the same instance. The [Engine.dispose()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.dispose" \o "sqlalchemy.engine.Engine.dispose) method will replace the connection pool instance for the parent engine as well as this one.新引擎使用的池是同一个实例。 Engine.dispose() 方法将替换父引擎的连接池实例以及此引擎。
* Event listeners are "cascaded" - meaning, the new [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) inherits the events of the parent, and new events can be associated with the new [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) individually.事件侦听器是“级联” - 意思是新的引擎继承父级事件，新的事件可以单独与新引擎相关联。
* The logging configuration and logging\_name is copied from the parent [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine)日志配置和logging\_name从父引擎复制。



The intent of 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 is to implement "sharding" schemes where multiple [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) objects refer to the same connection pool, but are differentiated by options that would be consumed by a custom event:

Engine.execution\_options() 方法的目的是实现“分片”方案，其中多个引擎对象引用相同的连接池，但由自定义事件消耗的选项区分开来：

primary\_engine = create\_engine("mysql://")

shard1 = primary\_engine.execution\_options(shard\_id="shard1")

shard2 = primary\_engine.execution\_options(shard\_id="shard2")

Above, the shard1 engine serves as a factory for [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) objects that will contain the execution option shard\_id=shard1, and shard2 will produce [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) objects that contain the execution option shard\_id=shard2.

以上，shard1引擎作为连接对象的工厂，它将包含执行选项shard\_id = shard1，shard2将产生包含执行选项shard\_id = shard2的Connection对象。

An event handler can consume the above execution option to perform a schema switch or other operation, given a connection. Below we emit a MySQL use statement to switch databases, at the same time keeping track of which database we've established using the [Connection.info](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.info" \o "sqlalchemy.engine.Connection.info) dictionary, which gives us a persistent storage space that follows the DBAPI connection:

给定连接，事件处理程序可以使用上述执行选项来执行模式切换或其他操作。 下面我们发出一个MySQL使用语句来切换数据库，同时跟踪我们使用Connection.info字典建立的数据库，这为我们提供了一个在DBAPI连接之后的持久存储空间：

**from** **sqlalchemy** **import** event

**from** **sqlalchemy.engine**

**import** Engine

shards = {"default": "base", shard\_1: "db1", "shard\_2": "db2"}

**@event**.listens\_for(Engine, "before\_cursor\_execute")**def** \_switch\_shard(conn, cursor, stmt,

params, context, executemany):

shard\_id = conn.\_execution\_options.get('shard\_id', "default")

current\_shard = conn.info.get("current\_shard", **None**)

**if** current\_shard != shard\_id:

cursor.execute("use *%s*" % shards[shard\_id])

conn.info["current\_shard"] = shard\_id

*New in version 0.8.*

**See also**

[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) - update execution options on a [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) object.

[Engine.update\_execution\_options()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.update_execution_options" \o "sqlalchemy.engine.Engine.update_execution_options) - update the execution options for a given [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) in place.

**has\_table**(*table\_name*, *schema=None*)

Return True if the given backend has a table of the given name.

**See also**

[Fine Grained Reflection with Inspector](http://docs.sqlalchemy.org/en/rel_1_1/core/reflection.html" \l "metadata-reflection-inspector) - detailed schema inspection using the [Inspector](http://docs.sqlalchemy.org/en/rel_1_1/core/reflection.html" \l "sqlalchemy.engine.reflection.Inspector" \o "sqlalchemy.engine.reflection.Inspector) interface.

[quoted\_name](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.elements.quoted_name" \o "sqlalchemy.sql.elements.quoted_name) - used to pass quoting information along with a schema identifier.

**name**

String name of the [Dialect](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Dialect" \o "sqlalchemy.engine.interfaces.Dialect) in use by this [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine).

**raw\_connection**(*\_connection=None*)

Return a "raw" DBAPI connection from the connection pool.

The returned object is a proxied version of the DBAPI connection object used by the underlying driver in use. The object will have all the same behavior as the real DBAPI connection, except that its close() method will result in the connection being returned to the pool, rather than being closed for real.

This method provides direct DBAPI connection access for special situations when the API provided by [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) is not needed. When a [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) object is already present, the DBAPI connection is available using the [Connection.connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.connection" \o "sqlalchemy.engine.Connection.connection) accessor.

**See also**

[Working with Raw DBAPI Connections](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "dbapi-connections)

**run\_callable**(*callable\_*, *\*args*, *\*\*kwargs*)

Given a callable object or function, execute it, passing a [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) as the first argument.

The given \*args and \*\*kwargs are passed subsequent to the [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) argument.

This function, along with [Connection.run\_callable()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.run_callable" \o "sqlalchemy.engine.Connection.run_callable), allows a function to be run with a [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) or [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) object without the need to know which one is being dealt with.

**schema\_for\_object***= <sqlalchemy.sql.schema.\_SchemaTranslateMap object>*

Return the ".schema" attribute for an object.

Used for [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table), [Sequence](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.Sequence" \o "sqlalchemy.schema.Sequence) and similar objects, and takes into account the [Connection.execution\_options.schema\_translate\_map](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.execution_options.params.schema_translate_map" \o "sqlalchemy.engine.Connection.execution_options)parameter.

*New in version 1.1.*

**See also**

[Translation of Schema Names](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "schema-translating)

**table\_names**(*schema=None*, *connection=None*)

Return a list of all table names available in the database.

|  |  |
| --- | --- |
| **Parameters:** | * ****schema**** – Optional, retrieve names from a non-default schema. * ****connection**** – Optional, use a specified connection. Default is the contextual\_connect for this Engine. |

**transaction**(*callable\_*, *\*args*, *\*\*kwargs*)

Execute the given function within a transaction boundary.

The function is passed a [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) newly procured from [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) as the first argument, followed by the given \*args and \*\*kwargs.

e.g.:

**def** do\_something(conn, x, y):

conn.execute("some statement", {'x':x, 'y':y})

engine.transaction(do\_something, 5, 10)

The operations inside the function are all invoked within the context of a single [Transaction](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction" \o "sqlalchemy.engine.Transaction). Upon success, the transaction is committed. If an exception is raised, the transaction is rolled back before propagating the exception.

**Note**

The [transaction()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.transaction" \o "sqlalchemy.engine.Engine.transaction) method is superseded by the usage of the Python with: statement, which can be used with [Engine.begin()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.begin" \o "sqlalchemy.engine.Engine.begin):

**with** engine.begin() **as** conn:

conn.execute("some statement", {'x':5, 'y':10})

See also:

[Engine.begin()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.begin" \o "sqlalchemy.engine.Engine.begin) - engine-level transactional context

[Connection.transaction()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.transaction" \o "sqlalchemy.engine.Connection.transaction) - connection-level version of [Engine.transaction()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.transaction" \o "sqlalchemy.engine.Engine.transaction)

**update\_execution\_options**(*\*\*opt*)

Update the default execution\_options dictionary of this [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine).

The given keys/values in \*\*opt are added to the default execution options that will be used for all connections. The initial contents of this dictionary can be sent via the execution\_options parameter to [create\_engine()](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "sqlalchemy.create_engine" \o "sqlalchemy.create_engine).

**See also**

[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)

[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)

*class*sqlalchemy.engine.**ExceptionContext**

Encapsulate information about an error condition in progress.

This object exists solely to be passed to the [ConnectionEvents.handle\_error()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents.handle_error" \o "sqlalchemy.events.ConnectionEvents.handle_error) event, supporting an interface that can be extended without backwards-incompatibility.

*New in version 0.9.7.*

**chained\_exception***= None*

The exception that was returned by the previous handler in the exception chain, if any.

If present, this exception will be the one ultimately raised by SQLAlchemy unless a subsequent handler replaces it.

May be None.

**connection***= None*

The [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) in use during the exception.

This member is present, except in the case of a failure when first connecting.

**See also**

[ExceptionContext.engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ExceptionContext.engine" \o "sqlalchemy.engine.ExceptionContext.engine)

**cursor***= None*

The DBAPI cursor object.

May be None.

**engine***= None*

The [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) in use during the exception.

This member should always be present, even in the case of a failure when first connecting.

*New in version 1.0.0.*

**execution\_context***= None*

The [ExecutionContext](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.ExecutionContext" \o "sqlalchemy.engine.interfaces.ExecutionContext) corresponding to the execution operation in progress.

This is present for statement execution operations, but not for operations such as transaction begin/end. It also is not present when the exception was raised before the [ExecutionContext](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.ExecutionContext" \o "sqlalchemy.engine.interfaces.ExecutionContext) could be constructed.

Note that the [ExceptionContext.statement](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ExceptionContext.statement" \o "sqlalchemy.engine.ExceptionContext.statement) and [ExceptionContext.parameters](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ExceptionContext.parameters" \o "sqlalchemy.engine.ExceptionContext.parameters) members may represent a different value than that of the [ExecutionContext](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.ExecutionContext" \o "sqlalchemy.engine.interfaces.ExecutionContext), potentially in the case where a [ConnectionEvents.before\_cursor\_execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents.before_cursor_execute" \o "sqlalchemy.events.ConnectionEvents.before_cursor_execute) event or similar modified the statement/parameters to be sent.

May be None.

**invalidate\_pool\_on\_disconnect***= True*

Represent whether all connections in the pool should be invalidated when a "disconnect" condition is in effect.

表示当"断开连接"条件有效时，池中的所有连接是否应该被无效。

Setting this flag to False within the scope of the [ConnectionEvents.handle\_error()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents.handle_error" \o "sqlalchemy.events.ConnectionEvents.handle_error) event will have the effect such that the full collection of connections in the pool will not be invalidated during a disconnect; only the current connection that is the subject of the error will actually be invalidated.

在ConnectionEvents.handle\_error() 事件的范围内将此标志设置为False将具有这样的效果，使得池中的连接的完整集合在断开连接期间不会失效; 只有作为错误主题的当前连接才会实际无效。

The purpose of this flag is for custom disconnect-handling schemes where the invalidation of other connections in the pool is to be performed based on other conditions, or even on a per-connection basis.

该标志的目的是针对自定义断开连接处理方案，其中基于其他条件或甚至基于每个连接基于其他条件执行池中其他连接的无效。

*New in version 1.0.3.*

**is\_disconnect***= None*

Represent whether the exception as occurred represents a "disconnect" condition.

表示发生的异常是否表示"断开"条件。

This flag will always be True or False within the scope of the [ConnectionEvents.handle\_error()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents.handle_error" \o "sqlalchemy.events.ConnectionEvents.handle_error) handler.

在ConnectionEvents.handle\_error() 处理程序的范围内，此标志将始终为True或False。

SQLAlchemy will defer to this flag in order to determine whether or not the connection should be invalidated subsequently. That is, by assigning to this flag, a "disconnect" event which then results in a connection and pool invalidation can be invoked or prevented by changing this flag.

SQLAlchemy将推迟到此标志，以确定连接是否应该在以后失效。 也就是说，通过分配给该标志，可以通过改变该标志来调用或防止"断开连接"事件，从而导致连接和池失效。

**original\_exception***= None*

The exception object which was caught.

被捕获的异常对象。

This member is always present.

这个会员总是在场。

**parameters***= None*

Parameter collection that was emitted directly to the DBAPI.

直接发送到DBAPI的参数集合。

May be None.

可能是无。

**sqlalchemy\_exception***= None*

The [sqlalchemy.exc.StatementError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.StatementError" \o "sqlalchemy.exc.StatementError) which wraps the original, and will be raised if exception handling is not circumvented by the event.

[sqlalchemy.exc.StatementError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.StatementError" \o "sqlalchemy.exc.StatementError)包装原始的，如果异常处理没有被事件规避，则会引发。

May be None, as not all exception types are wrapped by SQLAlchemy. For DBAPI-level exceptions that subclass the dbapi's Error class, this field will always be present.

可能是None，因为并非所有的异常类型都被SQLAlchemy包装。 对于dbapi的Error类子类的DBAPI级异常，此字段将始终存在。

**statement***= None*

String SQL statement that was emitted directly to the DBAPI.

May be None.

*class*sqlalchemy.engine.**NestedTransaction**(*connection*, *parent*)

Bases: [sqlalchemy.engine.Transaction](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction" \o "sqlalchemy.engine.Transaction)

Represent a 'nested', or SAVEPOINT transaction.

A new [NestedTransaction](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.NestedTransaction" \o "sqlalchemy.engine.NestedTransaction) object may be procured using the [Connection.begin\_nested()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.begin_nested" \o "sqlalchemy.engine.Connection.begin_nested) method.

The interface is the same as that of [Transaction](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction" \o "sqlalchemy.engine.Transaction).

*class*sqlalchemy.engine.**ResultProxy**(*context*)

Wraps a DB-API cursor object to provide easier access to row columns.

Individual columns may be accessed by their integer position, case-insensitive column name, or by schema.Column object. e.g.:

row = fetchone()

col1 = row[0] *# access via integer position*

col2 = row['col2'] *# access via name*

col3 = row[mytable.c.mycol] *# access via Column object.*

ResultProxy also handles post-processing of result column data using TypeEngine objects, which are referenced from the originating SQL statement that produced this result set.

ResultProxy还使用TypeEngine对象处理结果列数据的后处理，该引用从产生此结果集的始发SQL语句引用。

**\_cursor\_description**()

May be overridden by subclasses.

**\_process\_row**

alias of [RowProxy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.RowProxy" \o "sqlalchemy.engine.RowProxy)

**\_soft\_close**()

Soft close this [ResultProxy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy" \o "sqlalchemy.engine.ResultProxy).

软关闭此ResultProxy。

This releases all DBAPI cursor resources, but leaves the ResultProxy "open" from a semantic perspective, meaning the fetchXXX() methods will continue to return empty results.

这将释放所有的DBAPI游标资源，但从语义的角度离开ResultProxy"open"，这意味着fetchXXX() 方法将继续返回空的结果。

This method is called automatically when:

以下方法自动调用：

* all result rows are exhausted using the fetchXXX() methods.
* cursor.description is None.

This method is ****not public****, but is documented in order to clarify the "autoclose" process used.

*New in version 1.0.0.*

**See also**

[ResultProxy.close()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.close" \o "sqlalchemy.engine.ResultProxy.close)

**close**()

Close this ResultProxy.

关闭此ResultProxy。

This closes out the underlying DBAPI cursor corresonding to the statement execution, if one is still present. Note that the DBAPI cursor is automatically released when the [ResultProxy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy" \o "sqlalchemy.engine.ResultProxy) exhausts all available rows. [ResultProxy.close()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.close" \o "sqlalchemy.engine.ResultProxy.close) is generally an optional method except in the case when discarding a [ResultProxy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy" \o "sqlalchemy.engine.ResultProxy) that still has additional rows pending for fetch.

这将关闭与语句执行相关的底层DBAPI游标（如果还存在）。请注意，当ResultProxy排除所有可用行时，DBAPI游标将自动释放。 ResultProxy.close() 通常是一个可选方法，除非在抛出一个仍然有额外的行等待进行抓取的ResultProxy的情况。

In the case of a result that is the product of [connectionless execution](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "dbengine-implicit), the underlying [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) object is also closed, which [releases](http://docs.sqlalchemy.org/en/rel_1_1/glossary.html" \l "term-releases) DBAPI connection resources.

在结果为无连接执行的结果的情况下，底层的Connection对象也被关闭，这释放了DBAPI连接资源。

After this method is called, it is no longer valid to call upon the fetch methods, which will raise a [ResourceClosedError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.ResourceClosedError" \o "sqlalchemy.exc.ResourceClosedError) on subsequent use.

调用此方法后，调用fetch方法不再有效，这将在后续使用时引发[ResourceClosedError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.ResourceClosedError" \o "sqlalchemy.exc.ResourceClosedError)。

*Changed in version 1.0.0:*- the [ResultProxy.close()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.close" \o "sqlalchemy.engine.ResultProxy.close) method has been separated out from the process that releases the underlying DBAPI cursor resource. The "auto close" feature of the [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) now performs a so-called "soft close", which releases the underlying DBAPI cursor, but allows the [ResultProxy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy" \o "sqlalchemy.engine.ResultProxy) to still behave as an open-but-exhausted result set; the actual [ResultProxy.close()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.close" \o "sqlalchemy.engine.ResultProxy.close) method is never called. It is still safe to discard a [ResultProxy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy" \o "sqlalchemy.engine.ResultProxy) that has been fully exhausted without calling this method.

在版本1.0.0中更改： - ResultProxy.close() 方法已从释放底层DBAPI游标资源的进程中分离出来。连接的"自动关闭"功能现在执行所谓的"软关闭"，它释放底层的DBAPI游标，但允许ResultProxy仍然表现为一个开放而不尽尽的结果集;实际的ResultProxy.close() 方法从不被调用。丢弃没有调用此方法的ResultProxy完全耗尽，仍然是安全的。

**See also**

[Working with Engines and Connections](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html)

[ResultProxy.\_soft\_close()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy._soft_close" \o "sqlalchemy.engine.ResultProxy._soft_close)

**fetchall**()

Fetch all rows, just like DB-API cursor.fetchall().

After all rows have been exhausted, the underlying DBAPI cursor resource is released, and the object may be safely discarded.

Subsequent calls to [ResultProxy.fetchall()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.fetchall" \o "sqlalchemy.engine.ResultProxy.fetchall) will return an empty list. After the [ResultProxy.close()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.close" \o "sqlalchemy.engine.ResultProxy.close) method is called, the method will raise [ResourceClosedError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.ResourceClosedError" \o "sqlalchemy.exc.ResourceClosedError).

*Changed in version 1.0.0:*- Added "soft close" behavior which allows the result to be used in an "exhausted" state prior to calling the [ResultProxy.close()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.close" \o "sqlalchemy.engine.ResultProxy.close) method.

**fetchmany**(*size=None*)

Fetch many rows, just like DB-API cursor.fetchmany(size=cursor.arraysize).

获取很多行，就像DB-API cursor.fetchmany（size = cursor.arraysize）一样。

After all rows have been exhausted, the underlying DBAPI cursor resource is released, and the object may be safely discarded.

在所有行耗尽之后，底层DBAPI游标资源被释放，并且对象可能被安全地丢弃。

Calls to [ResultProxy.fetchmany()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.fetchmany" \o "sqlalchemy.engine.ResultProxy.fetchmany) after all rows have been exhausted will return an empty list. After the [ResultProxy.close()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.close" \o "sqlalchemy.engine.ResultProxy.close) method is called, the method will raise [ResourceClosedError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.ResourceClosedError" \o "sqlalchemy.exc.ResourceClosedError).

在所有行耗尽后调用ResultProxy.fetchmany() 将返回一个空列表。 调用ResultProxy.close() 方法后，该方法将引发ResourceClosedError。

*Changed in version 1.0.0:*- Added "soft close" behavior which allows the result to be used in an "exhausted" state prior to calling the [ResultProxy.close()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.close" \o "sqlalchemy.engine.ResultProxy.close) method.

在版本1.0.0中更改： - 添加了"软关闭"行为，允许在调用ResultProxy.close() 方法之前将结果用于"耗尽"状态。

**fetchone**()

Fetch one row, just like DB-API cursor.fetchone().

提取一行，就像DB-API cursor.fetchone() 一样。

After all rows have been exhausted, the underlying DBAPI cursor resource is released, and the object may be safely discarded.

在所有行耗尽之后，底层DBAPI游标资源被释放，并且对象可能被安全地丢弃。

Calls to [ResultProxy.fetchone()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.fetchone" \o "sqlalchemy.engine.ResultProxy.fetchone) after all rows have been exhausted will return None. After the [ResultProxy.close()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.close" \o "sqlalchemy.engine.ResultProxy.close) method is called, the method will raise [ResourceClosedError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.ResourceClosedError" \o "sqlalchemy.exc.ResourceClosedError).

在所有行耗尽后调用ResultProxy.fetchone() 将返回None。 调用ResultProxy.close() 方法后，该方法将引发ResourceClosedError。

*Changed in version 1.0.0:*- Added "soft close" behavior which allows the result to be used in an "exhausted" state prior to calling the [ResultProxy.close()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.close" \o "sqlalchemy.engine.ResultProxy.close) method.

在版本1.0.0中更改： - 添加了"软关闭"行为，允许在调用ResultProxy.close() 方法之前将结果用于"耗尽"状态。

**first**()

Fetch the first row and then close the result set unconditionally.

Returns None if no row is present.

After calling this method, the object is fully closed, e.g. the [ResultProxy.close()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.close" \o "sqlalchemy.engine.ResultProxy.close) method will have been called.

**inserted\_primary\_key**

Return the primary key for the row just inserted.

返回刚插入的行的主键。

The return value is a list of scalar values corresponding to the list of primary key columns in the target table.

返回值是与目标表中主键列列表对应的标量值列表。

This only applies to single row [insert()](http://docs.sqlalchemy.org/en/rel_1_1/dialects/postgresql.html" \l "sqlalchemy.dialects.postgresql.dml.insert" \o "sqlalchemy.dialects.postgresql.dml.insert) constructs which did not explicitly specify [Insert.returning()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.Insert.returning" \o "sqlalchemy.sql.expression.Insert.returning).

这仅适用于未明确指定Insert.returning() 的单行insert() 结构。

Note that primary key columns which specify a server\_default clause, or otherwise do not qualify as "autoincrement" columns (see the notes at [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column)), and were generated using the database-side default, will appear in this list as None unless the backend supports "returning" and the insert statement executed with the "implicit returning" enabled.

请注意，指定server\_default子句的主键列或以其他方式不符合"autoincrement"列（参见列中的注释）并使用数据库端默认值生成的主键列将在此列表中显示为"无"，除非后端支持 "返回"和使用"隐式返回"执行的插入语句。

Raises [InvalidRequestError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.InvalidRequestError" \o "sqlalchemy.exc.InvalidRequestError) if the executed statement is not a compiled expression construct or is not an insert() construct.

如果执行的语句不是编译的表达式构造，或者不是insert() 构造，则引发InvalidRequestError。

**is\_insert**

True if this [ResultProxy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy" \o "sqlalchemy.engine.ResultProxy) is the result of a executing an expression language compiled [expression.insert()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.insert" \o "sqlalchemy.sql.expression.insert) construct.

When True, this implies that the [inserted\_primary\_key](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.inserted_primary_key" \o "sqlalchemy.engine.ResultProxy.inserted_primary_key) attribute is accessible, assuming the statement did not include a user defined "returning" construct.

**keys**()

Return the current set of string keys for rows.

**last\_inserted\_params**()

Return the collection of inserted parameters from this execution.

Raises [InvalidRequestError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.InvalidRequestError" \o "sqlalchemy.exc.InvalidRequestError) if the executed statement is not a compiled expression construct or is not an insert() construct.

**last\_updated\_params**()

Return the collection of updated parameters from this execution.

Raises [InvalidRequestError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.InvalidRequestError" \o "sqlalchemy.exc.InvalidRequestError) if the executed statement is not a compiled expression construct or is not an update() construct.

**lastrow\_has\_defaults**()

Return lastrow\_has\_defaults() from the underlying [ExecutionContext](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.ExecutionContext" \o "sqlalchemy.engine.interfaces.ExecutionContext).

See [ExecutionContext](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.ExecutionContext" \o "sqlalchemy.engine.interfaces.ExecutionContext) for details.

**lastrowid**

return the 'lastrowid' accessor on the DBAPI cursor.

返回DBAPI游标上的"lastrowid"访问器。

This is a DBAPI specific method and is only functional for those backends which support it, for statements where it is appropriate. It's behavior is not consistent across backends.

这是一个DBAPI特定的方法，只适用于那些支持它的后端，对于适合的语句。 后端的行为是不一致的。

Usage of this method is normally unnecessary when using insert() expression constructs; the [inserted\_primary\_key](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.inserted_primary_key" \o "sqlalchemy.engine.ResultProxy.inserted_primary_key) attribute provides a tuple of primary key values for a newly inserted row, regardless of database backend.

当使用insert() 表达式构造时，通常不需要使用此方法; inserted\_primary\_key属性提供新插入行的主键值的元组，而不管数据库后端。

**postfetch\_cols**()

Return postfetch\_cols() from the underlying [ExecutionContext](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.ExecutionContext" \o "sqlalchemy.engine.interfaces.ExecutionContext).

See [ExecutionContext](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.ExecutionContext" \o "sqlalchemy.engine.interfaces.ExecutionContext) for details.

Raises [InvalidRequestError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.InvalidRequestError" \o "sqlalchemy.exc.InvalidRequestError) if the executed statement is not a compiled expression construct or is not an insert() or update() construct.

**prefetch\_cols**()

Return prefetch\_cols() from the underlying [ExecutionContext](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.ExecutionContext" \o "sqlalchemy.engine.interfaces.ExecutionContext).

See [ExecutionContext](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.ExecutionContext" \o "sqlalchemy.engine.interfaces.ExecutionContext) for details.

Raises [InvalidRequestError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.InvalidRequestError" \o "sqlalchemy.exc.InvalidRequestError) if the executed statement is not a compiled expression construct or is not an insert() or update() construct.

**returned\_defaults**

Return the values of default columns that were fetched using the [ValuesBase.return\_defaults()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase.return_defaults" \o "sqlalchemy.sql.expression.ValuesBase.return_defaults) feature.

The value is an instance of [RowProxy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.RowProxy" \o "sqlalchemy.engine.RowProxy), or None if [ValuesBase.return\_defaults()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase.return_defaults" \o "sqlalchemy.sql.expression.ValuesBase.return_defaults) was not used or if the backend does not support RETURNING.

*New in version 0.9.0.*

**See also**

[ValuesBase.return\_defaults()](http://docs.sqlalchemy.org/en/rel_1_1/core/dml.html" \l "sqlalchemy.sql.expression.ValuesBase.return_defaults" \o "sqlalchemy.sql.expression.ValuesBase.return_defaults)

**returns\_rows**

True if this [ResultProxy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy" \o "sqlalchemy.engine.ResultProxy) returns rows.

I.e. if it is legal to call the methods [fetchone()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.fetchone" \o "sqlalchemy.engine.ResultProxy.fetchone), [fetchmany()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.fetchmany" \o "sqlalchemy.engine.ResultProxy.fetchmany) [fetchall()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.fetchall" \o "sqlalchemy.engine.ResultProxy.fetchall).

**rowcount**

Return the 'rowcount' for this result.

The 'rowcount' reports the number of rows *matched* by the WHERE criterion of an UPDATE or DELETE statement.

**Note**

Notes regarding [ResultProxy.rowcount](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.rowcount" \o "sqlalchemy.engine.ResultProxy.rowcount):

* This attribute returns the number of rows *matched*, which is not necessarily the same as the number of rows that were actually *modified* - an UPDATE statement, for example, may have no net change on a given row if the SET values given are the same as those present in the row already. Such a row would be matched but not modified. On backends that feature both styles, such as MySQL, rowcount is configured by default to return the match count in all cases.This属性返回匹配的行数，这不一定与实际修改的行数相同 - 例如，如果给定的值相同，UPDATE语句可能没有给定行的净更改 就像已经在行中的那样。 这样的一行将被匹配但不被修改。 在具有两种样式的后台（如MySQL）中，默认情况下，rowcount配置为在所有情况下返回匹配计数。
* [ResultProxy.rowcount](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.rowcount" \o "sqlalchemy.engine.ResultProxy.rowcount) is *only* useful in conjunction with an UPDATE or DELETE statement. Contrary to what the Python DBAPI says, it does *not* return the number of rows available from the results of a SELECT statement as DBAPIs cannot support this functionality when rows are unbuffered.ResultProxy.rowcount仅适用于UPDATE或DELETE语句。 与Python DBAPI所说的相反，它不返回SELECT语句结果中可用的行数，因为当行不缓冲时，DBAPI无法支持此功能。
* [ResultProxy.rowcount](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.rowcount" \o "sqlalchemy.engine.ResultProxy.rowcount) may not be fully implemented by all dialects. In particular, most DBAPIs do not support an aggregate rowcount result from an executemany call. ResultProxy.rowcount可能没有被所有方言完全实现。 特别地，大多数DBAPI不支持执行者调用的聚合rowcount结果。
* The [ResultProxy.supports\_sane\_rowcount()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.supports_sane_rowcount" \o "sqlalchemy.engine.ResultProxy.supports_sane_rowcount) and [ResultProxy.supports\_sane\_multi\_rowcount()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.supports_sane_multi_rowcount" \o "sqlalchemy.engine.ResultProxy.supports_sane_multi_rowcount) methods will report from the dialect if each usage is known to be supported.如果知道每个使用情况被支持，ResultProxy.supports\_sane\_rowcount() 和ResultProxy.supports\_sane\_multi\_rowcount() 方法将从方言报告。
* Statements that use RETURNING may not return a correct rowcount.使用RETURNING的语句可能不会返回正确的行数。

**scalar**()

Fetch the first column of the first row, and close the result set.

获取第一行的第一列，并关闭结果集。

Returns None if no row is present.

如果没有行，则返回None。

After calling this method, the object is fully closed, e.g. the [ResultProxy.close()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.close" \o "sqlalchemy.engine.ResultProxy.close) method will have been called.

调用此方法后，对象完全关闭，例如 ResultProxy.close() 方法将被调用。

**supports\_sane\_multi\_rowcount**()

Return supports\_sane\_multi\_rowcount from the dialect.

See [ResultProxy.rowcount](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.rowcount" \o "sqlalchemy.engine.ResultProxy.rowcount) for background.

**supports\_sane\_rowcount**()

Return supports\_sane\_rowcount from the dialect.

See [ResultProxy.rowcount](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy.rowcount" \o "sqlalchemy.engine.ResultProxy.rowcount) for background.

*class*sqlalchemy.engine.**RowProxy**(*parent*, *row*, *processors*, *keymap*)

Bases: sqlalchemy.engine.BaseRowProxy

Proxy values from a single cursor row.

来自单个游标行的代理值。

Mostly follows "ordered dictionary" behavior, mapping result values to the string-based column name, the integer position of the result in the row, as well as Column instances which can be mapped to the original Columns that produced this result set (for results that correspond to constructed SQL expressions).

主要遵循"有序字典"行为，将结果值映射到基于字符串的列名称，行中结果的整数位置以及可以映射到生成此结果集的原始列的列实例（对于结果 对应于构造的SQL表达式）。

**has\_key**(*key*)

Return True if this RowProxy contains the given key.

**items**()

Return a list of tuples, each tuple containing a key/value pair.

**keys**()

Return the list of keys as strings represented by this RowProxy.

*class*sqlalchemy.engine.**Transaction**(*connection*, *parent*)

Represent a database transaction in progress.

The [Transaction](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction" \o "sqlalchemy.engine.Transaction) object is procured by calling the [begin()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.begin" \o "sqlalchemy.engine.Connection.begin) method of [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection):

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

engine = create\_engine("postgresql://scott:tiger@localhost/test")

connection = engine.connect()

trans = connection.begin()

connection.execute("insert into x (a, b) values (1, 2)")trans.commit()

The object provides [rollback()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction.rollback" \o "sqlalchemy.engine.Transaction.rollback) and [commit()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction.commit" \o "sqlalchemy.engine.Transaction.commit) methods in order to control transaction boundaries. It also implements a context manager interface so that the Python with statement can be used with the [Connection.begin()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.begin" \o "sqlalchemy.engine.Connection.begin) method:

该对象提供[rollback()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction.rollback" \o "sqlalchemy.engine.Transaction.rollback)和[commit()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction.commit" \o "sqlalchemy.engine.Transaction.commit)方法，以便控制事务边界。 它还实现一个上下文管理器接口，以便Python with语句可以与[Connection.begin()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.begin" \o "sqlalchemy.engine.Connection.begin)方法一起使用：

**with** connection.begin():

connection.execute("insert into x (a, b) values (1, 2)")

The Transaction object is ****not**** threadsafe.

See also: [Connection.begin()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.begin" \o "sqlalchemy.engine.Connection.begin), [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), [Connection.begin\_nested()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.begin_nested" \o "sqlalchemy.engine.Connection.begin_nested).

**close**()

Close this [Transaction](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction" \o "sqlalchemy.engine.Transaction).

关闭此交易。

If this transaction is the base transaction in a begin/commit nesting, the transaction will rollback(). Otherwise, the method returns.

如果此事务是开始/提交嵌套中的基本事务，则事务将rollback() 。 否则，该方法返回。

This is used to cancel a Transaction without affecting the scope of an enclosing transaction.

这用于取消事务而不影响封闭事务的范围。

**commit**()

Commit this [Transaction](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction" \o "sqlalchemy.engine.Transaction).

**rollback**()

Roll back this [Transaction](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction" \o "sqlalchemy.engine.Transaction).

*class*sqlalchemy.engine.**TwoPhaseTransaction**(*connection*, *xid*)

Bases: [sqlalchemy.engine.Transaction](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction" \o "sqlalchemy.engine.Transaction)

Represent a two-phase transaction.

A new [TwoPhaseTransaction](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.TwoPhaseTransaction" \o "sqlalchemy.engine.TwoPhaseTransaction) object may be procured using the [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) method.

The interface is the same as that of [Transaction](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction" \o "sqlalchemy.engine.Transaction) with the addition of the [prepare()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.TwoPhaseTransaction.prepare" \o "sqlalchemy.engine.TwoPhaseTransaction.prepare) method.

**prepare**()

Prepare this [TwoPhaseTransaction](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.TwoPhaseTransaction" \o "sqlalchemy.engine.TwoPhaseTransaction).

After a PREPARE, the transaction can be committed.

# **5.3 Connection Pooling**

A connection pool is a standard technique used to maintain long running connections in memory for efficient re-use, as well as to provide management for the total number of connections an application might use simultaneously.

连接池是用于在内存中维持长时间运行连接以进行有效重用的标准技术，以及为应用程序可能同时使用的连接总数提供管理。

Particularly for server-side web applications, a connection pool is the standard way to maintain a "pool" of active database connections in memory which are reused across requests.

特别是对于服务器端Web应用程序，连接池是维护内存中活动数据库连接"池"的标准方法，这些连接在跨请求重复使用。

SQLAlchemy includes several connection pool implementations which integrate with the [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine). They can also be used directly for applications that want to add pooling to an otherwise plain DBAPI approach.

SQLAlchemy包括与引擎集成的多个连接池实现。 它们也可以直接用于想要添加池到另外简单的DBAPI方法的应用程序。

5.3.1 Connection Pool Configuration

The [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) returned by the [create\_engine()](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "sqlalchemy.create_engine" \o "sqlalchemy.create_engine) function in most cases has a [QueuePool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.QueuePool" \o "sqlalchemy.pool.QueuePool) integrated, pre-configured with reasonable pooling defaults. If you're reading this section only to learn how to enable pooling - congratulations! You're already done.

在大多数情况下，由[create\_engine()](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "sqlalchemy.create_engine" \o "sqlalchemy.create_engine)函数返回的[Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine)具有集成的[QueuePool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.QueuePool" \o "sqlalchemy.pool.QueuePool)，预配置有合理的池默认值。 如果您正在阅读本节仅了解如何启用池 - 恭喜您！ 你已经完成了

The most common [QueuePool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.QueuePool" \o "sqlalchemy.pool.QueuePool) tuning parameters can be passed directly to [create\_engine()](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "sqlalchemy.create_engine" \o "sqlalchemy.create_engine) as keyword arguments: pool\_size, max\_overflow, pool\_recycle and pool\_timeout. For example:

最常见的QueuePool调优参数可以直接传递给create\_engine() 作为关键字参数：pool\_size，max\_overflow，pool\_recycle和pool\_timeout。 例如：

engine = create\_engine('postgresql://me@localhost/mydb',

pool\_size=20, max\_overflow=0)

In the case of SQLite, the [SingletonThreadPool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.SingletonThreadPool" \o "sqlalchemy.pool.SingletonThreadPool) or [NullPool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.NullPool" \o "sqlalchemy.pool.NullPool) are selected by the dialect to provide greater compatibility with SQLite's threading and locking model, as well as to provide a reasonable default behavior to SQLite "memory" databases, which maintain their entire dataset within the scope of a single connection.

在SQLite的情况下，SingletonThreadPool或NullPool由方言选择，以提供与SQLite的线程和锁定模型更大的兼容性，以及为SQLite"内存"数据库提供合理的默认行为，从而将其整个数据集保留在范围内的单一连接。

All SQLAlchemy pool implementations have in common that none of them "pre create" connections - all implementations wait until first use before creating a connection. At that point, if no additional concurrent checkout requests for more connections are made, no additional connections are created. This is why it's perfectly fine for [create\_engine()](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "sqlalchemy.create_engine" \o "sqlalchemy.create_engine) to default to using a [QueuePool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.QueuePool" \o "sqlalchemy.pool.QueuePool) of size five without regard to whether or not the application really needs five connections queued up - the pool would only grow to that size if the application actually used five connections concurrently, in which case the usage of a small pool is an entirely appropriate default behavior.

所有的SQLAlchemy池实现都有一个共同点，它们都不是"预先创建"连接 - 所有的实现都等到创建连接之前首先使用。在这一点上，如果没有进行更多连接的额外并发检出请求，则不会创建其他连接。这就是为什么create\_engine() 默认使用大小为5的QueuePool是完全正确的，而不考虑应用程序是否真的需要排队五个连接 - 如果应用程序实际上同时使用五个连接，那么池只会增长到该大小在这种情况下，小池的使用是完全合适的默认行为。

5.3.2 Switching Pool Implementations

The usual way to use a different kind of pool with [create\_engine()](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "sqlalchemy.create_engine" \o "sqlalchemy.create_engine) is to use the poolclass argument. This argument accepts a class imported from the sqlalchemy.pool module, and handles the details of building the pool for you. Common options include specifying [QueuePool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.QueuePool" \o "sqlalchemy.pool.QueuePool) with SQLite:

使用与create\_engine() 不同类型的池的通常方法是使用poolclass参数。 此参数接受从sqlalchemy.pool模块导入的类，并处理为您构建池的详细信息。 常用选项包括使用SQLite指定QueuePool：

**from** **sqlalchemy.pool** **import** QueuePool

engine = create\_engine('sqlite:///file.db', poolclass=QueuePool)

Disabling pooling using [NullPool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.NullPool" \o "sqlalchemy.pool.NullPool):

**from** **sqlalchemy.pool** **import** NullPool

engine = create\_engine(

'postgresql+psycopg2://scott:tiger@localhost/test',

poolclass=NullPool)

### 5.3.3 Using a Custom Connection Function

All [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool) classes accept an argument creator which is a callable that creates a new connection. [create\_engine()](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "sqlalchemy.create_engine" \o "sqlalchemy.create_engine) accepts this function to pass onto the pool via an argument of the same name:

所有Pool类都接受一个可创建新连接的参数创建者。 create\_engine() 接受此函数，通过相同名称的参数传递到池中：

**import** **sqlalchemy.pool** **as** **pool**

**import** **psycopg2**

**def** getconn():

c = psycopg2.connect(username='ed', host='127.0.0.1', dbname='test')

*# do things with 'c' to set up*

**return** c

engine = create\_engine('postgresql+psycopg2://', creator=getconn)

For most "initialize on connection" routines, it's more convenient to use the [PoolEvents](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.PoolEvents" \o "sqlalchemy.events.PoolEvents) event hooks, so that the usual URL argument to [create\_engine()](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "sqlalchemy.create_engine" \o "sqlalchemy.create_engine) is still usable. creator is there as a last resort for when a DBAPI has some form of connect that is not at all supported by SQLAlchemy.

对于大多数"初始化连接"例程，使用PoolEvents事件钩子更方便，因此create\_engine() 的通常URL参数仍然可用。 创建者是最后的手段，当DBAPI具有某种形式的连接，SQLAlchemy完全不支持时。

5.3.4 Constructing a Pool

To use a [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool) by itself, the creator function is the only argument that's required and is passed first, followed by any additional options:

**import** **sqlalchemy.pool** **as** **pool**

**import** **psycopg2**

**def** getconn():

c = psycopg2.connect(username='ed', host='127.0.0.1', dbname='test')

**return** c

mypool = pool.QueuePool(getconn, max\_overflow=10, pool\_size=5)

DBAPI connections can then be procured from the pool using the [Pool.connect()](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool.connect" \o "sqlalchemy.pool.Pool.connect) function. The return value of this method is a DBAPI connection that's contained within a transparent proxy:

*# get a connection*conn = mypool.connect()

*# use it*cursor = conn.cursor()cursor.execute("select foo")

The purpose of the transparent proxy is to intercept the close() call, such that instead of the DBAPI connection being closed, it is returned to the pool:

*# "close" the connection. Returns# it to the pool.*conn.close()

The proxy also returns its contained DBAPI connection to the pool when it is garbage collected, though it's not deterministic in Python that this occurs immediately (though it is typical with cPython).

代理还将其包含的DBAPI连接返回到池，当它被垃圾回收时，尽管它在Python中是不确定的，但是这种情况立即发生（尽管它是典型的cPython）。

The close() step also performs the important step of calling the rollback() method of the DBAPI connection. This is so that any existing transaction on the connection is removed, not only ensuring that no existing state remains on next usage, but also so that table and row locks are released as well as that any isolated data snapshots are removed. This behavior can be disabled using the reset\_on\_return option of [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool).

close() 步骤也执行调用DBAPI连接的rollback() 方法的重要步骤。 这样就可以删除连接上的任何现有事务，不仅可以确保下一次使用中不存在现有状态，还可以释放表和行锁以及删除任何隔离的数据快照。 可以使用Pool的reset\_on\_return选项禁用此行为。

A particular pre-created [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool) can be shared with one or more engines by passing it to the pool argument of [create\_engine()](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "sqlalchemy.create_engine" \o "sqlalchemy.create_engine):

e = create\_engine('postgresql://', pool=mypool)

### 3.4.5 Pool Events

Connection pools support an event interface that allows hooks to execute upon first connect, upon each new connection, and upon checkout and checkin of connections. See [PoolEvents](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.PoolEvents" \o "sqlalchemy.events.PoolEvents) for details.

连接池支持一个事件接口，允许钩子在第一次连接时，每次新的连接以及检出和检查连接时执行。 有关详细信息，请参阅PoolEvents。

3.4.6 Dealing with Disconnects

The connection pool has the ability to refresh individual connections as well as its entire set of connections, setting the previously pooled connections as "invalid". A common use case is allow the connection pool to gracefully recover when the database server has been restarted, and all previously established connections are no longer functional. There are two approaches to this.

连接池可以刷新单个连接以及其整套连接，将先前合并的连接设置为"无效"。 一个常见的用例是允许连接池在数据库服务器重新启动时正常恢复，并且所有以前建立的连接都不再起作用。 这有两种方法。

### Disconnect Handling - Optimistic

The most common approach is to let SQLAlchemy handle disconnects as they occur, at which point the pool is refreshed. This assumes the [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool) is used in conjunction with a [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine). The [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) has logic which can detect disconnection events and refresh the pool automatically.

最常见的方法是让SQLAlchemy处理断开连接，从而刷新池。 这假定池与引擎一起使用。 引擎具有可以自动检测断开事件和刷新池的逻辑。

When the [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) attempts to use a DBAPI connection, and an exception is raised that corresponds to a "disconnect" event, the connection is invalidated. The [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) then calls the [Pool.recreate()](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool.recreate" \o "sqlalchemy.pool.Pool.recreate) method, effectively invalidating all connections not currently checked out so that they are replaced with new ones upon next checkout:

当连接尝试使用DBAPI连接，并且引发与"disconnect"事件相对应的异常时，连接无效。 连接然后调用Pool.recreate() 方法，有效地使当前未检出的所有连接失效，以便在下次检出时将其替换为新的：

**from** **sqlalchemy** **import** create\_engine, exc

e = create\_engine(...)

c = e.connect()

**try**:

*# suppose the database has been restarted.*

c.execute("SELECT \* FROM table")

c.close()

**except** exc.DBAPIError, e:

*# an exception is raised, Connection is invalidated.*

**if** e.connection\_invalidated:

print("Connection was invalidated!")

*# after the invalidate event, a new connection# starts with a new Pool*c = e.connect()c.execute("SELECT \* FROM table")

The above example illustrates that no special intervention is needed, the pool continues normally after a disconnection event is detected. However, an exception is raised. In a typical web application using an ORM Session, the above condition would correspond to a single request failing with a 500 error, then the web application continuing normally beyond that. Hence the approach is "optimistic" in that frequent database restarts are not anticipated.

上述示例说明，不需要特殊的干预，检测到断开连接事件后，池会正常运行。 但是，提出了一个例外。 在使用ORM会话的典型Web应用程序中，上述条件将对应于一个失败的500个错误的请求，然后Web应用程序继续正常执行。 因此，方法是"乐观的"，因为频繁的数据库重新启动是不可预料的。

#### Setting Pool Recycle

An additional setting that can augment the "optimistic" approach is to set the pool recycle parameter. This parameter prevents the pool from using a particular connection that has passed a certain age, and is appropriate for database backends such as MySQL that automatically close connections that have been stale after a particular period of time:

可以增加"乐观"方法的附加设置是设置池回收参数。 此参数可防止池使用特定年龄的特定连接，适用于数据库后端（如MySQL），可自动关闭在特定时间段之后已过时的连接：

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

e = create\_engine("mysql://scott:tiger@localhost/test", pool\_recycle=3600)

Above, any DBAPI connection that has been open for more than one hour will be invalidated and replaced, upon next checkout. Note that the invalidation ****only**** occurs during checkout - not on any connections that are held in a checked out state. pool\_recycle is a function of the [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool) itself, independent of whether or not an [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) is in use.

以上，已经打开超过一个小时的任何DBAPI连接将在下次结帐时无效并被替换。 请注意，无效仅在检出期间发生 - 不在保持在检出状态的任何连接上。 pool\_recycle是Pool本身的一个功能，与Engine是否正在使用无关。

### Disconnect Handling - Pessimistic

At the expense of some extra SQL emitted for each connection checked out from the pool, a "ping" operation established by a checkout event handler can detect an invalid connection before it is used. In modern SQLAlchemy, the best way to do this is to make use of the [ConnectionEvents.engine\_connect()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents.engine_connect" \o "sqlalchemy.events.ConnectionEvents.engine_connect) event, assuming the use of a [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) and not just a raw [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool) object:

以从池中检出的每个连接发出的一些额外的SQL为代价，由checkout事件处理程序建立的"ping"操作可以在使用之前检测到无效的连接。 在现代SQLAlchemy中，最好的方式是使用ConnectionEvents.engine\_connect() 事件，假设使用Engine而不仅仅是一个Raw Pool对象：

**from** **sqlalchemy** **import** exc

**from** **sqlalchemy** **import** event

**from** **sqlalchemy** **import** select

some\_engine = create\_engine(...)

**@event**.listens\_for(some\_engine, "engine\_connect")**def** ping\_connection(connection, branch):

**if** branch:

*# "branch" refers to a sub-connection of a connection,*

*# we don't want to bother pinging on these.*

**return**

*# turn off "close with result". This flag is only used with*

*# "connectionless" execution, otherwise will be False in any case*

save\_should\_close\_with\_result = connection.should\_close\_with\_result

connection.should\_close\_with\_result = **False**

**try**:

*# run a SELECT 1. use a core select() so that*

*# the SELECT of a scalar value without a table is*

*# appropriately formatted for the backend*

connection.scalar(select([1]))

**except** exc.DBAPIError **as** err:

*# catch SQLAlchemy's DBAPIError, which is a wrapper*

*# for the DBAPI's exception. It includes a .connection\_invalidated*

*# attribute which specifies if this connection is a "disconnect"*

*# condition, which is based on inspection of the original exception*

*# by the dialect in use.*

**if** err.connection\_invalidated:

*# run the same SELECT again - the connection will re-validate*

*# itself and establish a new connection. The disconnect detection*

*# here also causes the whole connection pool to be invalidated*

*# so that all stale connections are discarded.*

connection.scalar(select([1]))

**else**:

**raise**

**finally**:

*# restore "close with result"*

connection.should\_close\_with\_result = save\_should\_close\_with\_result

The above recipe has the advantage that we are making use of SQLAlchemy's facilities for detecting those DBAPI exceptions that are known to indicate a "disconnect" situation, as well as the [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) object's ability to correctly invalidate the current connection pool when this condition occurs and allowing the current [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection)to re-validate onto a new DBAPI connection.

上述配方的优点是，我们正在利用SQLAlchemy的功能来检测已知表示"断开连接"情况的DBAPI异常，以及Engine对象在发生此情况时正确使当前连接池无效的能力，并允许 当前的Connection去重新验证到一个新的DBAPI连接。

For the much less common case of where a [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool) is being used without an [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine), an older approach may be used as below:

对于在没有引擎的情况下使用池的情况不太常见的情况，可以使用较旧的方法，如下所示：

**from** **sqlalchemy** **import** exc

**from** **sqlalchemy** **import** event

**from** **sqlalchemy.pool** **import** Pool

**@event**.listens\_for(Pool, "checkout")

**def** ping\_connection(dbapi\_connection, connection\_record, connection\_proxy):

cursor = dbapi\_connection.cursor()

**try**:

cursor.execute("SELECT 1")

**except**:

*# raise DisconnectionError - pool will try*

*# connecting again up to three times before raising.*

**raise** exc.DisconnectionError()

cursor.close()

Above, the [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool) object specifically catches [DisconnectionError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.DisconnectionError" \o "sqlalchemy.exc.DisconnectionError) and attempts to create a new DBAPI connection, up to three times, before giving up and then raising [InvalidRequestError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.InvalidRequestError" \o "sqlalchemy.exc.InvalidRequestError), failing the connection. The disadvantage of the above approach is that we don't have any easy way of determining if the exception raised is in fact a "disconnect" situation, since there is no [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) or [Dialect](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Dialect" \o "sqlalchemy.engine.interfaces.Dialect) in play, and also the above error would occur individually for all stale connections still in the pool.

以上，Pool对象特别捕获DisconnectionError，并尝试创建一个新的DBAPI连接，最多三次，然后放弃然后提高InvalidRequestError，连接失败。 上述方法的缺点是，我们没有任何简单的方法来确定引发的异常是否实际上是"断开"的情况，因为没有引擎或方言，并且上述错误将单独发生 所有陈旧的连接仍然在游泳池。

### More on Invalidation

The [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool) provides "connection invalidation" services which allow both explicit invalidation of a connection as well as automatic invalidation in response to conditions that are determined to render a connection unusable.

池提供"连接无效"服务，其允许连接的显式无效以及响应于确定使连接不可用的条件而自动失效。

"Invalidation" means that a particular DBAPI connection is removed from the pool and discarded. The .close() method is called on this connection if it is not clear that the connection itself might not be closed, however if this method fails, the exception is logged but the operation still proceeds.

"无效"表示特定的DBAPI连接从池中删除并被丢弃。 如果不清楚连接本身可能未关闭，则在此连接上调用.close() 方法，但是如果此方法失败，则记录异常，但操作仍然继续。

When using a [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine), 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 is the usual entrypoint to explicit invalidation. Other conditions by which a DBAPI connection might be invalidated include:

使用引擎时，Connection.invalidate() 方法是明确无效的通常入口点。 DBAPI连接可能无效的其他条件包括：

* a DBAPI exception such as [OperationalError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.OperationalError" \o "sqlalchemy.exc.OperationalError), raised when a method like connection.execute() is called, is detected as indicating a so-called "disconnect" condition. As the Python DBAPI provides no standard system for determining the nature of an exception, all SQLAlchemy dialects include a system called is\_disconnect() which will examine the contents of an exception object, including the string message and any potential error codes included with it, in order to determine if this exception indicates that the connection is no longer usable. If this is the case, the [\_ConnectionFairy.invalidate()](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool._ConnectionFairy.invalidate" \o "sqlalchemy.pool._ConnectionFairy.invalidate) method is called and the DBAPI connection is then discarded.在调用类似connect.execute() 的方法时引发的诸如OperationalError之类的DBAPI异常被检测为指示所谓的"断开"条件。 由于Python DBAPI没有提供用于确定异常性质的标准系统，所有SQLAlchemy方言都包含一个名为is\_disconnect() 的系统，该系统将检查异常对象的内容，包括字符串消息及其中包含的任何潜在的错误代码 以确定此异常是否指示连接不再可用。 如果是这种情况，则调用\_ConnectionFairy.invalidate() 方法，然后丢弃DBAPI连接。
* When the connection is returned to the pool, and calling the connection.rollback() or connection.commit() methods, as dictated by the pool's "reset on return" behavior, throws an exception. A final attempt at calling .close() on the connection will be made, and it is then discarded.当连接返回到池中，并调用connection.rollback() 或connection.commit() 方法时，如池的"重置返回"行为所规定的那样，会引发异常。 将在连接上调用.close() 的最后一次尝试，然后将其丢弃。
* When a listener implementing [PoolEvents.checkout()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.PoolEvents.checkout" \o "sqlalchemy.events.PoolEvents.checkout) raises the [DisconnectionError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.DisconnectionError" \o "sqlalchemy.exc.DisconnectionError) exception, indicating that the connection won't be usable and a new connection attempt needs to be made.当监听器实现PoolEvents.checkout() 引发DisconnectionError异常时，指示连接将不可用，并且需要进行新的连接尝试。

All invalidations which occur will invoke the [PoolEvents.invalidate()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.PoolEvents.invalidate" \o "sqlalchemy.events.PoolEvents.invalidate) event.

5.3.7 Using Connection Pools with Multiprocessing

It's critical that when using a connection pool, and by extension when using an [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) created via [create\_engine()](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "sqlalchemy.create_engine" \o "sqlalchemy.create_engine), that the pooled connections ****are not shared to a forked process****. TCP connections are represented as file descriptors, which usually work across process boundaries, meaning this will cause concurrent access to the file descriptor on behalf of two or more entirely independent Python interpreter states.

当使用通过create\_engine() 创建的引擎时，使用连接池以及扩展时，重要的是池池连接不会共享到分叉进程。 TCP连接表示为文件描述符，通常可以跨过程边界工作，这意味着代表两个或更多完全独立的Python解释器状态的并发访问文件描述符。

There are two approaches to dealing with this.

有两种处理方法。

The first is, either create a new [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) within the child process, or upon an existing [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine), call [Engine.dispose()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.dispose" \o "sqlalchemy.engine.Engine.dispose) before the child process uses any connections. This will remove all existing connections from the pool so that it makes all new ones. Below is a simple version using multiprocessing.Process, but this idea should be adapted to the style of forking in use:

第一个是在子进程中创建一个新引擎，或者在一个现有的引擎下，在子进程使用任何连接之前调用Engine.dispose() 。 这将从池中删除所有现有连接，从而使其成为所有新连接。 下面是一个使用multiprocessing.Process的简单版本，但是这个想法应该适应于正在使用的分支样式：

eng = create\_engine("...")

**def** run\_in\_process():

eng.dispose()

**with** eng.connect() **as** conn:

conn.execute("...")

p = Process(target=run\_in\_process)

The next approach is to instrument the [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool) itself with events so that connections are automatically invalidated in the subprocess. This is a little more magical but probably more foolproof:

下一个方法是使用事件来监测池本身，以便子流程中的连接自动失效。 这有点更神奇，但可能更加愚蠢：

**from** **sqlalchemy** **import** event

**from** **sqlalchemy** **import** exc

**import** **os**

eng = create\_engine("...")

**@event**.listens\_for(engine, "connect")**def** connect(dbapi\_connection, connection\_record):

connection\_record.info['pid'] = os.getpid()

**@event**.listens\_for(engine, "checkout")**def** checkout(dbapi\_connection, connection\_record, connection\_proxy):

pid = os.getpid()

**if** connection\_record.info['pid'] != pid:

connection\_record.connection = connection\_proxy.connection = **None**

**raise** exc.DisconnectionError(

"Connection record belongs to pid *%s*, "

"attempting to check out in pid *%s*" %

(connection\_record.info['pid'], pid)

)

Above, we use an approach similar to that described in [Disconnect Handling - Pessimistic](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "pool-disconnects-pessimistic) to treat a DBAPI connection that originated in a different parent process as an "invalid" connection, coercing the pool to recycle the connection record to make a new connection.

以上，我们使用类似于Disconnect Handling - Pessimistic中描述的方法来将源自不同父进程的DBAPI连接视为"无效"连接，强制池回收连接记录以创建新连接。

5.3.8 API Documentation - Available Pool Implementations

*class*sqlalchemy.pool.**Pool**(*creator*, *recycle=-1*, *echo=None*, *use\_threadlocal=False*, *logging\_name=None*, *reset\_on\_return=True*, *listeners=None*, *events=None*, *dialect=None*, *\_dispatch=None*)

Bases: [sqlalchemy.log.Identified](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.log.Identified" \o "sqlalchemy.log.Identified)

Abstract base class for connection pools.

**\_\_init\_\_**(*creator*, *recycle=-1*, *echo=None*, *use\_threadlocal=False*, *logging\_name=None*, *reset\_on\_return=True*, *listeners=None*, *events=None*, *dialect=None*, *\_dispatch=None*)

Construct a Pool.

|  |  |
| --- | --- |
| **Parameters:** | * ****creator**** – a callable function that returns a DB-API connection object. The function will be called with parameters. * ****recycle**** – If set to non -1, number of seconds between connection recycling, which means upon checkout, if this timeout is surpassed the connection will be closed and replaced with a newly opened connection. Defaults to -1. * ****logging\_name**** – String identifier which will be used within the "name" field of logging records generated within the "sqlalchemy.pool" logger. Defaults to a hexstring of the object's id. * ****echo**** – If True, connections being pulled and retrieved from the pool will be logged to the standard output, as well as pool sizing information. Echoing can also be achieved by enabling logging for the "sqlalchemy.pool" namespace. Defaults to False. * ****use\_threadlocal –****If set to True, repeated calls to [connect()](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool.connect" \o "sqlalchemy.pool.Pool.connect) within the same application thread will be guaranteed to return the same connection object, if one has already been retrieved from the pool and has not been returned yet. Offers a slight performance advantage at the cost of individual transactions by default. The [Pool.unique\_connection()](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool.unique_connection" \o "sqlalchemy.pool.Pool.unique_connection) method is provided to return a consistently unique connection to bypass this behavior when the flag is set.   **Warning**  The [Pool.use\_threadlocal](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool.params.use_threadlocal" \o "sqlalchemy.pool.Pool) flag ****does not affect the behavior**** of [Engine.connect()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.connect" \o "sqlalchemy.engine.Engine.connect). [Engine.connect()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.connect" \o "sqlalchemy.engine.Engine.connect) makes use of the[Pool.unique\_connection()](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool.unique_connection" \o "sqlalchemy.pool.Pool.unique_connection) method which ****does not use thread local context****. To produce a [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) which refers to the [Pool.connect()](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool.connect" \o "sqlalchemy.pool.Pool.connect) method, use [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).  Note that other SQLAlchemy connectivity systems such as [Engine.execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.execute" \o "sqlalchemy.engine.Engine.execute) as well as 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) make use of [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)internally, so these functions are compatible with the [Pool.use\_threadlocal](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool.params.use_threadlocal" \o "sqlalchemy.pool.Pool) setting.  **See also**  [Using the Threadlocal Execution Strategy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "threadlocal-strategy) - contains detail on the "threadlocal" engine strategy, which provides a more comprehensive approach to "threadlocal" connectivity for the specific use case of using [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) and [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) objects directly.   * ****reset\_on\_return –****Determine steps to take on connections as they are returned to the pool. reset\_on\_return can have any of these values:   + "rollback" - call rollback() on the connection, to release locks and transaction resources. This is the default value. The vast majority of use cases should leave this value set.   + True - same as 'rollback', this is here for backwards compatibility.   + "commit" - call commit() on the connection, to release locks and transaction resources. A commit here may be desirable for databases that cache query plans if a commit is emitted, such as Microsoft SQL Server. However, this value is more dangerous than 'rollback' because any data changes present on the transaction are committed unconditionally.   + None - don't do anything on the connection. This setting should only be made on a database that has no transaction support at all, namely MySQL MyISAM. By not doing anything, performance can be improved. This setting should ****never be selected**** for a database that supports transactions, as it will lead to deadlocks and stale state.   + "none" - same as None   *New in version 0.9.10.*   * + False - same as None, this is here for backwards compatibility.   *Changed in version 0.7.6:*[Pool.reset\_on\_return](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool.params.reset_on_return" \o "sqlalchemy.pool.Pool) accepts "rollback" and "commit" arguments.   * ****events**** – a list of 2-tuples, each of the form (callable, target) which will be passed to [event.listen()](http://docs.sqlalchemy.org/en/rel_1_1/core/event.html" \l "sqlalchemy.event.listen" \o "sqlalchemy.event.listen) upon construction. Provided here so that event listeners can be assigned via [create\_engine()](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "sqlalchemy.create_engine" \o "sqlalchemy.create_engine) before dialect-level listeners are applied. * ****listeners**** – Deprecated. A list of [PoolListener](http://docs.sqlalchemy.org/en/rel_1_1/core/interfaces.html" \l "sqlalchemy.interfaces.PoolListener" \o "sqlalchemy.interfaces.PoolListener)-like objects or dictionaries of callables that receive events when DB-API connections are created, checked out and checked in to the pool. This has been superseded by [listen()](http://docs.sqlalchemy.org/en/rel_1_1/core/event.html" \l "sqlalchemy.event.listen" \o "sqlalchemy.event.listen). * ****dialect –****a [Dialect](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Dialect" \o "sqlalchemy.engine.interfaces.Dialect) that will handle the job of calling rollback(), close(), or commit() on DBAPI connections. If omitted, a built-in "stub" dialect is used. Applications that make use of [create\_engine()](http://docs.sqlalchemy.org/en/rel_1_1/core/engines.html" \l "sqlalchemy.create_engine" \o "sqlalchemy.create_engine) should not use this parameter as it is handled by the engine creation strategy.   *New in version 1.1:*- dialect is now a public parameter to the [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool). |

**connect**()

Return a DBAPI connection from the pool.

The connection is instrumented such that when its close() method is called, the connection will be returned to the pool.

从池返回一个DBAPI连接。

连接被调用，以便在调用其close() 方法时，连接将返回到池中。

**dispose**()

Dispose of this pool.

This method leaves the possibility of checked-out connections remaining open, as it only affects connections that are idle in the pool.

See also the [Pool.recreate()](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool.recreate" \o "sqlalchemy.pool.Pool.recreate) method.

**recreate**()

Return a new [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool), of the same class as this one and configured with identical creation arguments.

This method is used in conjunction with [dispose()](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool.dispose" \o "sqlalchemy.pool.Pool.dispose) to close out an entire [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool) and create a new one in its place.

**unique\_connection**()

Produce a DBAPI connection that is not referenced by any thread-local context.

This method is equivalent to [Pool.connect()](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool.connect" \o "sqlalchemy.pool.Pool.connect) when the [Pool.use\_threadlocal](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool.params.use_threadlocal" \o "sqlalchemy.pool.Pool) flag is not set to True. When [Pool.use\_threadlocal](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool.params.use_threadlocal" \o "sqlalchemy.pool.Pool) is True, the [Pool.unique\_connection()](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool.unique_connection" \o "sqlalchemy.pool.Pool.unique_connection) method provides a means of bypassing the threadlocal context.

*class*sqlalchemy.pool.**QueuePool**(*creator*, *pool\_size=5*, *max\_overflow=10*, *timeout=30*, *\*\*kw*)

Bases: [sqlalchemy.pool.Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool)

A [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool) that imposes a limit on the number of open connections.

[QueuePool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.QueuePool" \o "sqlalchemy.pool.QueuePool) is the default pooling implementation used for all [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) objects, unless the SQLite dialect is in use.

**\_\_init\_\_**(*creator*, *pool\_size=5*, *max\_overflow=10*, *timeout=30*, *\*\*kw*)

Construct a QueuePool.

|  |  |
| --- | --- |
| **Parameters:** | * ****creator**** – a callable function that returns a DB-API connection object, same as that of [Pool.creator](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool.params.creator" \o "sqlalchemy.pool.Pool). * ****pool\_size**** – The size of the pool to be maintained, defaults to 5. This is the largest number of connections that will be kept persistently in the pool. Note that the pool begins with no connections; once this number of connections is requested, that number of connections will remain. pool\_size can be set to 0 to indicate no size limit; to disable pooling, use a [NullPool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.NullPool" \o "sqlalchemy.pool.NullPool) instead. * ****max\_overflow**** – The maximum overflow size of the pool. When the number of checked-out connections reaches the size set in pool\_size, additional connections will be returned up to this limit. When those additional connections are returned to the pool, they are disconnected and discarded. It follows then that the total number of simultaneous connections the pool will allow is pool\_size + max\_overflow, and the total number of "sleeping" connections the pool will allow is pool\_size. max\_overflow can be set to -1 to indicate no overflow limit; no limit will be placed on the total number of concurrent connections. Defaults to 10. * ****timeout**** – The number of seconds to wait before giving up on returning a connection. Defaults to 30. * ****\*\*kw**** – Other keyword arguments including [Pool.recycle](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool.params.recycle" \o "sqlalchemy.pool.Pool), [Pool.echo](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool.params.echo" \o "sqlalchemy.pool.Pool), [Pool.reset\_on\_return](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool.params.reset_on_return" \o "sqlalchemy.pool.Pool) and others are passed to the [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool)constructor. |

**connect**()

*inherited from the* [connect()](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool.connect" \o "sqlalchemy.pool.Pool.connect) *method of* [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool)

Return a DBAPI connection from the pool.

The connection is instrumented such that when its close() method is called, the connection will be returned to the pool.

**unique\_connection**()

*inherited from the* [unique\_connection()](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool.unique_connection" \o "sqlalchemy.pool.Pool.unique_connection) *method of* [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool)

Produce a DBAPI connection that is not referenced by any thread-local context.

This method is equivalent to [Pool.connect()](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool.connect" \o "sqlalchemy.pool.Pool.connect) when the [Pool.use\_threadlocal](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool.params.use_threadlocal" \o "sqlalchemy.pool.Pool) flag is not set to True. When [Pool.use\_threadlocal](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool.params.use_threadlocal" \o "sqlalchemy.pool.Pool) is True, the [Pool.unique\_connection()](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool.unique_connection" \o "sqlalchemy.pool.Pool.unique_connection) method provides a means of bypassing the threadlocal context.

*class*sqlalchemy.pool.**SingletonThreadPool**(*creator*, *pool\_size=5*, *\*\*kw*)

Bases: [sqlalchemy.pool.Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool)

A Pool that maintains one connection per thread.

Maintains one connection per each thread, never moving a connection to a thread other than the one which it was created in.

**Warning**

the [SingletonThreadPool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.SingletonThreadPool" \o "sqlalchemy.pool.SingletonThreadPool) will call .close() on arbitrary connections that exist beyond the size setting of pool\_size, e.g. if more unique ****thread identities**** than what pool\_size states are used. This cleanup is non-deterministic and not sensitive to whether or not the connections linked to those thread identities are currently in use.

[SingletonThreadPool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.SingletonThreadPool" \o "sqlalchemy.pool.SingletonThreadPool) may be improved in a future release, however in its current status it is generally used only for test scenarios using a SQLite :memory: database and is not recommended for production use.

Options are the same as those of [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool), as well as:

|  |  |
| --- | --- |
| **Parameters:** | ****pool\_size**** – The number of threads in which to maintain connections at once. Defaults to five. |

[SingletonThreadPool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.SingletonThreadPool" \o "sqlalchemy.pool.SingletonThreadPool) is used by the SQLite dialect automatically when a memory-based database is used. See [SQLite](http://docs.sqlalchemy.org/en/rel_1_1/dialects/sqlite.html).

**\_\_init\_\_**(*creator*, *pool\_size=5*, *\*\*kw*)

*class*sqlalchemy.pool.**AssertionPool**(*\*args*, *\*\*kw*)

Bases: [sqlalchemy.pool.Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool)

A [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool) that allows at most one checked out connection at any given time.

This will raise an exception if more than one connection is checked out at a time. Useful for debugging code that is using more connections than desired.

*Changed in version 0.7:*[AssertionPool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.AssertionPool" \o "sqlalchemy.pool.AssertionPool) also logs a traceback of where the original connection was checked out, and reports this in the assertion error raised.

*class*sqlalchemy.pool.**NullPool**(*creator*, *recycle=-1*, *echo=None*, *use\_threadlocal=False*, *logging\_name=None*, *reset\_on\_return=True*, *listeners=None*, *events=None*, *dialect=None*, *\_dispatch=None*)

Bases: [sqlalchemy.pool.Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool)

A Pool which does not pool connections.

不连接池的池。

Instead it literally opens and closes the underlying DB-API connection per each connection open/close.

相反，它会逐个打开和关闭每个连接打开/关闭的底层DB-API连接。

Reconnect-related functions such as recycle and connection invalidation are not supported by this Pool implementation, since no connections are held persistently.

此池实现不支持重新连接相关的功能，如回收和连接无效，因为不持续连接。

*Changed in version 0.7:*[NullPool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.NullPool" \o "sqlalchemy.pool.NullPool) is used by the SQlite dialect automatically when a file-based database is used. See [SQLite](http://docs.sqlalchemy.org/en/rel_1_1/dialects/sqlite.html).

*class*sqlalchemy.pool.**StaticPool**(*creator*, *recycle=-1*, *echo=None*, *use\_threadlocal=False*, *logging\_name=None*, *reset\_on\_return=True*, *listeners=None*, *events=None*, *dialect=None*, *\_dispatch=None*)

Bases: [sqlalchemy.pool.Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool)

A Pool of exactly one connection, used for all requests.

一个完全一个连接的池，用于所有请求。

Reconnect-related functions such as recycle and connection invalidation (which is also used to support auto-reconnect) are not currently supported by this Pool implementation but may be implemented in a future release.

此重新连接相关功能（如回收和连接无效（也用于支持自动重新连接））目前不受此池实现的支持，但可能在将来的版本中实现。

*class*sqlalchemy.pool.**\_ConnectionFairy**(*dbapi\_connection*, *connection\_record*, *echo*)

Proxies a DBAPI connection and provides return-on-dereference support.

代理一个DBAPI连接，并提供返还引用支持。

This is an internal object used by the [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool) implementation to provide context management to a DBAPI connection delivered by that [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool).

这是Pool实现使用的一个内部对象，用于向该池提供的DBAPI连接提供上下文管理。

The name "fairy" is inspired by the fact that the [\_ConnectionFairy](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool._ConnectionFairy" \o "sqlalchemy.pool._ConnectionFairy) object's lifespan is transitory, as it lasts only for the length of a specific DBAPI connection being checked out from the pool, and additionally that as a transparent proxy, it is mostly invisible.

“仙女”这个名字的灵感来自于\_ConnectionFairy对象的生命周期是短暂的，因为它仅持续从池中检出的特定DBAPI连接的长度，另外作为透明代理，它大都是不可见的。

**See also**

[\_ConnectionRecord](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool._ConnectionRecord" \o "sqlalchemy.pool._ConnectionRecord)

**\_connection\_record***= None*

A reference to the [\_ConnectionRecord](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool._ConnectionRecord" \o "sqlalchemy.pool._ConnectionRecord) object associated with the DBAPI connection.

This is currently an internal accessor which is subject to change.

**connection***= None*

A reference to the actual DBAPI connection being tracked.

**cursor**(*\*args*, *\*\*kwargs*)

Return a new DBAPI cursor for the underlying connection.

This method is a proxy for the connection.cursor() DBAPI method.

**detach**()

Separate this connection from its Pool.

将此连接与池相分离。

This means that the connection will no longer be returned to the pool when closed, and will instead be literally closed. The containing ConnectionRecord is separated from the DB-API connection, and will create a new connection when next used.

这意味着关闭时，连接将不再返回到池中，而是将其逐字地关闭。 包含的ConnectionRecord与DB-API连接分离，并在下一次使用时创建一个新的连接。

Note that any overall connection limiting constraints imposed by a Pool implementation may be violated after a detach, as the detached connection is removed from the pool's knowledge and control.

请注意，由于脱离的连接从池的知识和控制中移除，在执行分离后，由Pool实现强加的任何总体连接限制约束可能会被违反。

**info**

Info dictionary associated with the underlying DBAPI connection referred to by this ConnectionFairy, allowing user-defined data to be associated with the connection.

与此ConnectionFairy引用的底层DBAPI连接相关联的信息字典，允许用户定义的数据与连接相关联。

The data here will follow along with the DBAPI connection including after it is returned to the connection pool and used again in subsequent instances of [\_ConnectionFairy](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool._ConnectionFairy" \o "sqlalchemy.pool._ConnectionFairy). It is shared with the [\_ConnectionRecord.info](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool._ConnectionRecord.info" \o "sqlalchemy.pool._ConnectionRecord.info) and [Connection.info](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.info" \o "sqlalchemy.engine.Connection.info) accessors.

这里的数据将跟随DBAPI连接，包括在它返回到连接池之后，再次在\_ConnectionFairy的后续实例中使用。 它与\_ConnectionRecord.info和Connection.info访问器共享。

The dictionary associated with a particular DBAPI connection is discarded when the connection itself is discarded.

当连接本身被丢弃时，与特定DBAPI连接关联的字典将被丢弃。

**invalidate**(*e=None*, *soft=False*)

Mark this connection as invalidated.

将此连接标记为无效。

This method can be called directly, and is also called as a result of 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. When invoked, the DBAPI connection is immediately closed and discarded from further use by the pool. The invalidation mechanism proceeds via the [\_ConnectionRecord.invalidate()](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool._ConnectionRecord.invalidate" \o "sqlalchemy.pool._ConnectionRecord.invalidate)internal method.

该方法可以直接调用，也可以作为Connection.invalidate() 方法的结果调用。 当调用时，DBAPI连接立即关闭，并被池的进一步使用丢弃。 无效机制通过\_ConnectionRecord.invalidate() 内部方法进行。

|  |  |
| --- | --- |
| **Parameters:** | * ****e**** – an exception object indicating a reason for the invalidation. * ****soft –****if True, the connection isn't closed; instead, this connection will be recycled on next checkout.   *New in version 1.0.3.* |

**See also**

[More on Invalidation](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "pool-connection-invalidation)

**is\_valid**

Return True if this [\_ConnectionFairy](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool._ConnectionFairy" \o "sqlalchemy.pool._ConnectionFairy) still refers to an active DBAPI connection.

**record\_info**

Info dictionary associated with the \_ConnectionRecord container referred to by this :class:.ConnectionFairy`.

Unlike the [\_ConnectionFairy.info](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool._ConnectionFairy.info" \o "sqlalchemy.pool._ConnectionFairy.info) dictionary, the lifespan of this dictionary is persistent across connections that are disconnected and/or invalidated within the lifespan of a [\_ConnectionRecord](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool._ConnectionRecord" \o "sqlalchemy.pool._ConnectionRecord).

*New in version 1.1.*

*class*sqlalchemy.pool.**\_ConnectionRecord**(*pool*, *connect=True*)

Internal object which maintains an individual DBAPI connection referenced by a [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool).

The [\_ConnectionRecord](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool._ConnectionRecord" \o "sqlalchemy.pool._ConnectionRecord) object always exists for any particular DBAPI connection whether or not that DBAPI connection has been "checked out". This is in contrast to the [\_ConnectionFairy](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool._ConnectionFairy" \o "sqlalchemy.pool._ConnectionFairy) which is only a public facade to the DBAPI connection while it is checked out.

A [\_ConnectionRecord](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool._ConnectionRecord" \o "sqlalchemy.pool._ConnectionRecord) may exist for a span longer than that of a single DBAPI connection. For example, if the [\_ConnectionRecord.invalidate()](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool._ConnectionRecord.invalidate" \o "sqlalchemy.pool._ConnectionRecord.invalidate)method is called, the DBAPI connection associated with this [\_ConnectionRecord](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool._ConnectionRecord" \o "sqlalchemy.pool._ConnectionRecord) will be discarded, but the [\_ConnectionRecord](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool._ConnectionRecord" \o "sqlalchemy.pool._ConnectionRecord) may be used again, in which case a new DBAPI connection is produced when the [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool) next uses this record.

The [\_ConnectionRecord](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool._ConnectionRecord" \o "sqlalchemy.pool._ConnectionRecord) is delivered along with connection pool events, including [PoolEvents.connect()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.PoolEvents.connect" \o "sqlalchemy.events.PoolEvents.connect) and [PoolEvents.checkout()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.PoolEvents.checkout" \o "sqlalchemy.events.PoolEvents.checkout), however [\_ConnectionRecord](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool._ConnectionRecord" \o "sqlalchemy.pool._ConnectionRecord) still remains an internal object whose API and internals may change.

**See also**

[\_ConnectionFairy](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool._ConnectionFairy" \o "sqlalchemy.pool._ConnectionFairy)

**connection***= None*

A reference to the actual DBAPI connection being tracked.

May be None if this [\_ConnectionRecord](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool._ConnectionRecord" \o "sqlalchemy.pool._ConnectionRecord) has been marked as invalidated; a new DBAPI connection may replace it if the owning pool calls upon this [\_ConnectionRecord](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool._ConnectionRecord" \o "sqlalchemy.pool._ConnectionRecord) to reconnect.

**info**

The .info dictionary associated with the DBAPI connection.

This dictionary is shared among the [\_ConnectionFairy.info](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool._ConnectionFairy.info" \o "sqlalchemy.pool._ConnectionFairy.info) and [Connection.info](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.info" \o "sqlalchemy.engine.Connection.info) accessors.

**Note**

The lifespan of this dictionary is linked to the DBAPI connection itself, meaning that it is ****discarded**** each time the DBAPI connection is closed and/or invalidated. The[\_ConnectionRecord.record\_info](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool._ConnectionRecord.record_info" \o "sqlalchemy.pool._ConnectionRecord.record_info) dictionary remains persistent throughout the lifespan of the [\_ConnectionRecord](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool._ConnectionRecord" \o "sqlalchemy.pool._ConnectionRecord) container.

**invalidate**(*e=None*, *soft=False*)

Invalidate the DBAPI connection held by this [\_ConnectionRecord](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool._ConnectionRecord" \o "sqlalchemy.pool._ConnectionRecord).

This method is called for all connection invalidations, including when the [\_ConnectionFairy.invalidate()](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool._ConnectionFairy.invalidate" \o "sqlalchemy.pool._ConnectionFairy.invalidate) or [Connection.invalidate()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.invalidate" \o "sqlalchemy.engine.Connection.invalidate)methods are called, as well as when any so-called "automatic invalidation" condition occurs.

|  |  |
| --- | --- |
| **Parameters:** | * ****e**** – an exception object indicating a reason for the invalidation. * ****soft –****if True, the connection isn't closed; instead, this connection will be recycled on next checkout.   *New in version 1.0.3.* |

**See also**

[More on Invalidation](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "pool-connection-invalidation)

**record\_info**

An "info' dictionary associated with the connection record itself.

Unlike the [\_ConnectionRecord.info](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool._ConnectionRecord.info" \o "sqlalchemy.pool._ConnectionRecord.info) dictionary, which is linked to the lifespan of the DBAPI connection, this dictionary is linked to the lifespan of the [\_ConnectionRecord](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool._ConnectionRecord" \o "sqlalchemy.pool._ConnectionRecord) container itself and will remain persisent throughout the life of the [\_ConnectionRecord](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool._ConnectionRecord" \o "sqlalchemy.pool._ConnectionRecord).

*New in version 1.1.*

5.3.9 Pooling Plain DB-API Connections

Any **[PEP 249](https://www.python.org/dev/peps/pep-0249)** DB-API module can be "proxied" through the connection pool transparently. Usage of the DB-API is exactly as before, except the connect() method will consult the pool. Below we illustrate this with psycopg2:

**import** **sqlalchemy.pool** **as** **pool**

**import** **psycopg2** **as** **psycopg**

psycopg = pool.manage(psycopg)

*# then connect normally*connection = psycopg.connect(database='test', username='scott',

password='tiger')

This produces a \_DBProxy object which supports the same connect() function as the original DB-API module. Upon connection, a connection proxy object is returned, which delegates its calls to a real DB-API connection object. This connection object is stored persistently within a connection pool (an instance of [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool)) that corresponds to the exact connection arguments sent to the connect() function.

The connection proxy supports all of the methods on the original connection object, most of which are proxied via \_\_getattr\_\_(). The close() method will return the connection to the pool, and the cursor() method will return a proxied cursor object. Both the connection proxy and the cursor proxy will also return the underlying connection to the pool after they have both been garbage collected, which is detected via weakref callbacks (\_\_del\_\_ is not used).

Additionally, when connections are returned to the pool, a rollback() is issued on the connection unconditionally. This is to release any locks still held by the connection that may have resulted from normal activity.

By default, the connect() method will return the same connection that is already checked out in the current thread. This allows a particular connection to be used in a given thread without needing to pass it around between functions. To disable this behavior, specify use\_threadlocal=False to the manage() function.

sqlalchemy.pool.**manage**(*module*, *\*\*params*)

Return a proxy for a DB-API module that automatically pools connections.

Given a DB-API 2.0 module and pool management parameters, returns a proxy for the module that will automatically pool connections, creating new connection pools for each distinct set of connection arguments sent to the decorated module's connect() function.

|  |  |
| --- | --- |
| **Parameters:** | * ****module**** – a DB-API 2.0 database module * ****poolclass**** – the class used by the pool module to provide pooling. Defaults to [QueuePool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.QueuePool" \o "sqlalchemy.pool.QueuePool). * ****\*\*params**** – will be passed through to *poolclass* |

sqlalchemy.pool.**clear\_managers**()

Remove all current DB-API 2.0 managers.

All pools and connections are disposed.

## 5.4 Core Events

This section describes the event interfaces provided in SQLAlchemy Core. For an introduction to the event listening API, see [Events](http://docs.sqlalchemy.org/en/rel_1_1/core/event.html). ORM events are described in [ORM Events](http://docs.sqlalchemy.org/en/rel_1_1/orm/events.html).

*class*sqlalchemy.event.base.**Events**

Define event listening functions for a particular target type.

5.4.1 Connection Pool Events

*class*sqlalchemy.events.**PoolEvents**

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)

Available events for [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool).

The methods here define the name of an event as well as the names of members that are passed to listener functions.

e.g.:

**from** **sqlalchemy** **import** event

**def** my\_on\_checkout(dbapi\_conn, connection\_rec, connection\_proxy):

"handle an on checkout event"

event.listen(Pool, 'checkout', my\_on\_checkout)

In addition to accepting the [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool) class and [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool) instances, [PoolEvents](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.PoolEvents" \o "sqlalchemy.events.PoolEvents) also accepts [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) objects and the [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) class as targets, which will be resolved to the .pool attribute of the given engine or the [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool) class:

engine = create\_engine("postgresql://scott:tiger@localhost/test")

*# will associate with engine.pool*

event.listen(engine, 'checkout', my\_on\_checkout)

**checkin**(*dbapi\_connection*, *connection\_record*)

Called when a connection returns to the pool.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeEngineOrPool, 'checkin')

**def** receive\_checkin(dbapi\_connection, connection\_record):

"listen for the 'checkin' event"

*# ... (event handling logic) ...*

Note that the connection may be closed, and may be None if the connection has been invalidated. checkin will not be called for detached connections. (They do not return to the pool.)

|  |  |
| --- | --- |
| **Parameters:** | * ****dbapi\_connection**** – a DBAPI connection. * ****connection\_record**** – the [\_ConnectionRecord](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool._ConnectionRecord" \o "sqlalchemy.pool._ConnectionRecord) managing the DBAPI connection. |

**checkout**(*dbapi\_connection*, *connection\_record*, *connection\_proxy*)

Called when a connection is retrieved from the Pool.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeEngineOrPool, 'checkout')

**def** receive\_checkout(dbapi\_connection, connection\_record, connection\_proxy):

"listen for the 'checkout' event"

*# ... (event handling logic) ...*

|  |  |
| --- | --- |
| **Parameters:** | * ****dbapi\_connection**** – a DBAPI connection. * ****connection\_record**** – the [\_ConnectionRecord](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool._ConnectionRecord" \o "sqlalchemy.pool._ConnectionRecord) managing the DBAPI connection. * ****connection\_proxy**** – the [\_ConnectionFairy](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool._ConnectionFairy" \o "sqlalchemy.pool._ConnectionFairy) object which will proxy the public interface of the DBAPI connection for the lifespan of the checkout. |

If you raise a [DisconnectionError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.DisconnectionError" \o "sqlalchemy.exc.DisconnectionError), the current connection will be disposed and a fresh connection retrieved. Processing of all checkout listeners will abort and restart using the new connection.

**See also**

[ConnectionEvents.engine\_connect()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents.engine_connect" \o "sqlalchemy.events.ConnectionEvents.engine_connect) - a similar event which occurs upon creation of a new [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection).

**close**(*dbapi\_connection*, *connection\_record*)

Called when a DBAPI connection is closed.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeEngineOrPool, 'close')

**def** receive\_close(dbapi\_connection, connection\_record):

"listen for the 'close' event"

*# ... (event handling logic) ...*

The event is emitted before the close occurs.

The close of a connection can fail; typically this is because the connection is already closed. If the close operation fails, the connection is discarded.

The [close()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.close" \o "sqlalchemy.engine.Connection.close) event corresponds to a connection that's still associated with the pool. To intercept close events for detached connections use [close\_detached()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.PoolEvents.close_detached" \o "sqlalchemy.events.PoolEvents.close_detached).

*New in version 1.1.*

**close\_detached**(*dbapi\_connection*)

Called when a detached DBAPI connection is closed.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeEngineOrPool, 'close\_detached')

**def** receive\_close\_detached(dbapi\_connection):

"listen for the 'close\_detached' event"

*# ... (event handling logic) ...*

The event is emitted before the close occurs.

The close of a connection can fail; typically this is because the connection is already closed. If the close operation fails, the connection is discarded.

*New in version 1.1.*

**connect**(*dbapi\_connection*, *connection\_record*)

Called at the moment a particular DBAPI connection is first created for a given [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool).

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeEngineOrPool, 'connect')

**def** receive\_connect(dbapi\_connection, connection\_record):

"listen for the 'connect' event"

*# ... (event handling logic) ...*

This event allows one to capture the point directly after which the DBAPI module-level .connect() method has been used in order to produce a new DBAPI connection.

|  |  |
| --- | --- |
| **Parameters:** | * ****dbapi\_connection**** – a DBAPI connection. * ****connection\_record**** – the [\_ConnectionRecord](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool._ConnectionRecord" \o "sqlalchemy.pool._ConnectionRecord) managing the DBAPI connection. |

**detach**(*dbapi\_connection*, *connection\_record*)

Called when a DBAPI connection is "detached" from a pool.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeEngineOrPool, 'detach')

**def** receive\_detach(dbapi\_connection, connection\_record):

"listen for the 'detach' event"

*# ... (event handling logic) ...*

This event is emitted after the detach occurs. The connection is no longer associated with the given connection record.

*New in version 1.1.*

**first\_connect**(*dbapi\_connection*, *connection\_record*)

Called exactly once for the first time a DBAPI connection is checked out from a particular [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool).

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeEngineOrPool, 'first\_connect')

**def** receive\_first\_connect(dbapi\_connection, connection\_record):

"listen for the 'first\_connect' event"

*# ... (event handling logic) ...*

The rationale for [PoolEvents.first\_connect()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.PoolEvents.first_connect" \o "sqlalchemy.events.PoolEvents.first_connect) is to determine information about a particular series of database connections based on the settings used for all connections. Since a particular [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool) refers to a single "creator" function (which in terms of a [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) refers to the URL and connection options used), it is typically valid to make observations about a single connection that can be safely assumed to be valid about all subsequent connections, such as the database version, the server and client encoding settings, collation settings, and many others.

|  |  |
| --- | --- |
| **Parameters:** | * ****dbapi\_connection**** – a DBAPI connection. * ****connection\_record**** – the [\_ConnectionRecord](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool._ConnectionRecord" \o "sqlalchemy.pool._ConnectionRecord) managing the DBAPI connection. |

**invalidate**(*dbapi\_connection*, *connection\_record*, *exception*)

Called when a DBAPI connection is to be "invalidated".

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style***@event**.listens\_for(SomeEngineOrPool, 'invalidate')**def** receive\_invalidate(dbapi\_connection, connection\_record, exception):

"listen for the 'invalidate' event"

*# ... (event handling logic) ...*

This event is called any time the [\_ConnectionRecord.invalidate()](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool._ConnectionRecord.invalidate" \o "sqlalchemy.pool._ConnectionRecord.invalidate) method is invoked, either from API usage or via "auto-invalidation", without the soft flag.

The event occurs before a final attempt to call .close() on the connection occurs.

|  |  |
| --- | --- |
| **Parameters:** | * ****dbapi\_connection**** – a DBAPI connection. * ****connection\_record**** – the [\_ConnectionRecord](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool._ConnectionRecord" \o "sqlalchemy.pool._ConnectionRecord) managing the DBAPI connection. * ****exception**** – the exception object corresponding to the reason for this invalidation, if any. May be None. |

*New in version 0.9.2:*Added support for connection invalidation listening.

**See also**

[More on Invalidation](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "pool-connection-invalidation)

**reset**(*dbapi\_connection*, *connection\_record*)

Called before the "reset" action occurs for a pooled connection.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeEngineOrPool, 'reset')

**def** receive\_reset(dbapi\_connection, connection\_record):

"listen for the 'reset' event"

*# ... (event handling logic) ...*

This event represents when the rollback() method is called on the DBAPI connection before it is returned to the pool. The behavior of "reset" can be controlled, including disabled, using the reset\_on\_return pool argument.

The [PoolEvents.reset()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.PoolEvents.reset" \o "sqlalchemy.events.PoolEvents.reset) event is usually followed by the [PoolEvents.checkin()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.PoolEvents.checkin" \o "sqlalchemy.events.PoolEvents.checkin) event is called, except in those cases where the connection is discarded immediately after reset.

|  |  |
| --- | --- |
| **Parameters:** | * ****dbapi\_connection**** – a DBAPI connection. * ****connection\_record**** – the [\_ConnectionRecord](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool._ConnectionRecord" \o "sqlalchemy.pool._ConnectionRecord) managing the DBAPI connection. |

*New in version 0.8.*

**See also**

[ConnectionEvents.rollback()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents.rollback" \o "sqlalchemy.events.ConnectionEvents.rollback)

[ConnectionEvents.commit()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents.commit" \o "sqlalchemy.events.ConnectionEvents.commit)

**soft\_invalidate**(*dbapi\_connection*, *connection\_record*, *exception*)

Called when a DBAPI connection is to be "soft invalidated".

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style***@event**.listens\_for(SomeEngineOrPool, 'soft\_invalidate')**def** receive\_soft\_invalidate(dbapi\_connection, connection\_record, exception):

"listen for the 'soft\_invalidate' event"

*# ... (event handling logic) ...*

This event is called any time the [\_ConnectionRecord.invalidate()](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool._ConnectionRecord.invalidate" \o "sqlalchemy.pool._ConnectionRecord.invalidate) method is invoked with the soft flag.

Soft invalidation refers to when the connection record that tracks this connection will force a reconnect after the current connection is checked in. It does not actively close the dbapi\_connection at the point at which it is called.

*New in version 1.0.3.*

5.4.2 SQL Execution and Connection Events

*class*sqlalchemy.events.**ConnectionEvents**

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)

Available events for [Connectable](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connectable" \o "sqlalchemy.engine.Connectable), which includes [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) and [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine).

The methods here define the name of an event as well as the names of members that are passed to listener functions.

An event listener can be associated with any [Connectable](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connectable" \o "sqlalchemy.engine.Connectable) class or instance, such as an [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine), e.g.:

**from** **sqlalchemy** **import** event, create\_engine

**def** before\_cursor\_execute(conn, cursor, statement, parameters, context,

executemany):

log.info("Received statement: *%s*", statement)

engine = create\_engine('postgresql://scott:tiger@localhost/test')event.listen(engine, "before\_cursor\_execute", before\_cursor\_execute)

or with a specific [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection):

**with** engine.begin() **as** conn:

**@event**.listens\_for(conn, 'before\_cursor\_execute')

**def** before\_cursor\_execute(conn, cursor, statement, parameters,

context, executemany):

log.info("Received statement: *%s*", statement)

When the methods are called with a statement parameter, such as in [after\_cursor\_execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents.after_cursor_execute" \o "sqlalchemy.events.ConnectionEvents.after_cursor_execute), [before\_cursor\_execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents.before_cursor_execute" \o "sqlalchemy.events.ConnectionEvents.before_cursor_execute) and [dbapi\_error()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents.dbapi_error" \o "sqlalchemy.events.ConnectionEvents.dbapi_error), the statement is the exact SQL string that was prepared for transmission to the DBAPI cursor in the connection's [Dialect](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Dialect" \o "sqlalchemy.engine.interfaces.Dialect).

The [before\_execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents.before_execute" \o "sqlalchemy.events.ConnectionEvents.before_execute) and [before\_cursor\_execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents.before_cursor_execute" \o "sqlalchemy.events.ConnectionEvents.before_cursor_execute) events can also be established with the retval=True flag, which allows modification of the statement and parameters to be sent to the database. The [before\_cursor\_execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents.before_cursor_execute" \o "sqlalchemy.events.ConnectionEvents.before_cursor_execute) event is particularly useful here to add ad-hoc string transformations, such as comments, to all executions:

**from** **sqlalchemy.engine** **import** Engine

**from** **sqlalchemy** **import** event

**@event**.listens\_for(Engine, "before\_cursor\_execute", retval=**True**)

**def** comment\_sql\_calls(conn, cursor, statement, parameters,

context, executemany):

statement = statement + " -- some comment"

**return** statement, parameters

**Note**

[ConnectionEvents](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents" \o "sqlalchemy.events.ConnectionEvents) can be established on any combination of [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine), [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection), as well as instances of each of those classes. Events across all four scopes will fire off for a given instance of [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection). However, for performance reasons, the [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) object determines at instantiation time whether or not its parent [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) has event listeners established. Event listeners added to the [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) class or to an instance of [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) *after* the instantiation of a dependent [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) instance will usually*not* be available on that [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) instance. The newly added listeners will instead take effect for [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) instances created subsequent to those event listeners being established on the parent [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) class or instance.

|  |  |
| --- | --- |
| **Parameters:** | ****retval=False**** – Applies to the [before\_execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents.before_execute" \o "sqlalchemy.events.ConnectionEvents.before_execute) and [before\_cursor\_execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents.before_cursor_execute" \o "sqlalchemy.events.ConnectionEvents.before_cursor_execute) events only. When True, the user-defined event function must have a return value, which is a tuple of parameters that replace the given statement and parameters. See those methods for a description of specific return arguments. |

*Changed in version 0.8:*[ConnectionEvents](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents" \o "sqlalchemy.events.ConnectionEvents) can now be associated with any [Connectable](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connectable" \o "sqlalchemy.engine.Connectable) including [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection), in addition to the existing support for [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine).

**after\_cursor\_execute**(*conn*, *cursor*, *statement*, *parameters*, *context*, *executemany*)

Intercept low-level cursor execute() events after execution.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeEngine, 'after\_cursor\_execute')

**def** receive\_after\_cursor\_execute(conn, cursor, statement, parameters, context, executemany):

"listen for the 'after\_cursor\_execute' event"

*# ... (event handling logic) ...*

*# named argument style (new in 0.9)*

**@event**.listens\_for(SomeEngine, 'after\_cursor\_execute', named=**True**)

**def** receive\_after\_cursor\_execute(\*\*kw):

"listen for the 'after\_cursor\_execute' event"

conn = kw['conn']

cursor = kw['cursor']

*# ... (event handling logic) ...*

|  |  |
| --- | --- |
| **Parameters:** | * ****conn**** – [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) object * ****cursor**** – DBAPI cursor object. Will have results pending if the statement was a SELECT, but these should not be consumed as they will be needed by the [ResultProxy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy" \o "sqlalchemy.engine.ResultProxy). * ****statement**** – string SQL statement, as passed to the DBAPI * ****parameters**** – Dictionary, tuple, or list of parameters being passed to the execute() or executemany() method of the DBAPI cursor. In some cases may be None. * ****context**** – [ExecutionContext](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.ExecutionContext" \o "sqlalchemy.engine.interfaces.ExecutionContext) object in use. May be None. * ****executemany**** – boolean, if True, this is an executemany() call, if False, this is an execute() call. |

**after\_execute**(*conn*, *clauseelement*, *multiparams*, *params*, *result*)

Intercept high level execute() events after execute.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeEngine, 'after\_execute')

**def** receive\_after\_execute(conn, clauseelement, multiparams, params, result):

"listen for the 'after\_execute' event"

*# ... (event handling logic) ...*

*# named argument style (new in 0.9)*

**@event**.listens\_for(SomeEngine, 'after\_execute', named=**True**)

**def** receive\_after\_execute(\*\*kw):

"listen for the 'after\_execute' event"

conn = kw['conn']

clauseelement = kw['clauseelement']

*# ... (event handling logic) ...*

|  |  |
| --- | --- |
| **Parameters:** | * ****conn**** – [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) object * ****clauseelement**** – SQL expression construct, [Compiled](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Compiled" \o "sqlalchemy.engine.interfaces.Compiled) instance, or string statement passed to [Connection.execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.execute" \o "sqlalchemy.engine.Connection.execute). * ****multiparams**** – Multiple parameter sets, a list of dictionaries. * ****params**** – Single parameter set, a single dictionary. * ****result**** – [ResultProxy](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ResultProxy" \o "sqlalchemy.engine.ResultProxy) generated by the execution. |

**before\_cursor\_execute**(*conn*, *cursor*, *statement*, *parameters*, *context*, *executemany*)

Intercept low-level cursor execute() events before execution, receiving the string SQL statement and DBAPI-specific parameter list to be invoked against a cursor.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeEngine, 'before\_cursor\_execute')

**def** receive\_before\_cursor\_execute(conn, cursor, statement, parameters, context, executemany):

"listen for the 'before\_cursor\_execute' event"

*# ... (event handling logic) ...*

*# named argument style (new in 0.9)*

**@event**.listens\_for(SomeEngine, 'before\_cursor\_execute', named=**True**)

**def** receive\_before\_cursor\_execute(\*\*kw):

"listen for the 'before\_cursor\_execute' event"

conn = kw['conn']

cursor = kw['cursor']

*# ... (event handling logic) ...*

This event is a good choice for logging as well as late modifications to the SQL string. It's less ideal for parameter modifications except for those which are specific to a target backend.

This event can be optionally established with the retval=True flag. The statement and parameters arguments should be returned as a two-tuple in this case:

**@event**.listens\_for(Engine, "before\_cursor\_execute", retval=**True**)

**def** before\_cursor\_execute(conn, cursor, statement,

parameters, context, executemany):

*# do something with statement, parameters*

**return** statement, parameters

See the example at [ConnectionEvents](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents" \o "sqlalchemy.events.ConnectionEvents).

|  |  |
| --- | --- |
| **Parameters:** | * ****conn**** – [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) object * ****cursor**** – DBAPI cursor object * ****statement**** – string SQL statement, as to be passed to the DBAPI * ****parameters**** – Dictionary, tuple, or list of parameters being passed to the execute() or executemany() method of the DBAPI cursor. In some cases may be None. * ****context**** – [ExecutionContext](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.ExecutionContext" \o "sqlalchemy.engine.interfaces.ExecutionContext) object in use. May be None. * ****executemany**** – boolean, if True, this is an executemany() call, if False, this is an execute() call. |

See also:

[before\_execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents.before_execute" \o "sqlalchemy.events.ConnectionEvents.before_execute)

[after\_cursor\_execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents.after_cursor_execute" \o "sqlalchemy.events.ConnectionEvents.after_cursor_execute)

**before\_execute**(*conn*, *clauseelement*, *multiparams*, *params*)

Intercept high level execute() events, receiving uncompiled SQL constructs and other objects prior to rendering into SQL.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style***@event**.listens\_for(SomeEngine, 'before\_execute')

**def** receive\_before\_execute(conn, clauseelement, multiparams, params):

"listen for the 'before\_execute' event"

*# ... (event handling logic) ...*

*# named argument style (new in 0.9)*

**@event**.listens\_for(SomeEngine, 'before\_execute', named=**True**)

**def** receive\_before\_execute(\*\*kw):

"listen for the 'before\_execute' event"

conn = kw['conn']

clauseelement = kw['clauseelement']

*# ... (event handling logic) ...*

This event is good for debugging SQL compilation issues as well as early manipulation of the parameters being sent to the database, as the parameter lists will be in a consistent format here.

This event can be optionally established with the retval=True flag. The clauseelement, multiparams, and params arguments should be returned as a three-tuple in this case:

**@event**.listens\_for(Engine, "before\_execute", retval=**True**)

**def** before\_execute(conn, conn, clauseelement, multiparams, params):

*# do something with clauseelement, multiparams, params*

**return** clauseelement, multiparams, params

|  |  |
| --- | --- |
| **Parameters:** | * ****conn**** – [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) object * ****clauseelement**** – SQL expression construct, [Compiled](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Compiled" \o "sqlalchemy.engine.interfaces.Compiled) instance, or string statement passed to [Connection.execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.execute" \o "sqlalchemy.engine.Connection.execute). * ****multiparams**** – Multiple parameter sets, a list of dictionaries. * ****params**** – Single parameter set, a single dictionary. |

See also:

[before\_cursor\_execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents.before_cursor_execute" \o "sqlalchemy.events.ConnectionEvents.before_cursor_execute)

**begin**(*conn*)

Intercept begin() events.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeEngine, 'begin')

**def** receive\_begin(conn):

"listen for the 'begin' event"

*# ... (event handling logic) ...*

|  |  |
| --- | --- |
| **Parameters:** | ****conn**** – [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) object |

**begin\_twophase**(*conn*, *xid*)

Intercept begin\_twophase() events.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeEngine, 'begin\_twophase')

**def** receive\_begin\_twophase(conn, xid):

"listen for the 'begin\_twophase' event"

*# ... (event handling logic) ...*

|  |  |
| --- | --- |
| **Parameters:** | * ****conn**** – [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) object * ****xid**** – two-phase XID identifier |

**commit**(*conn*)

Intercept commit() events, as initiated by a [Transaction](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction" \o "sqlalchemy.engine.Transaction).

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeEngine, 'commit')

**def** receive\_commit(conn):

"listen for the 'commit' event"

*# ... (event handling logic) ...*

Note that the [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool) may also "auto-commit" a DBAPI connection upon checkin, if the reset\_on\_return flag is set to the value 'commit'. To intercept this commit, use the [PoolEvents.reset()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.PoolEvents.reset" \o "sqlalchemy.events.PoolEvents.reset) hook.

|  |  |
| --- | --- |
| **Parameters:** | ****conn**** – [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) object |

**commit\_twophase**(*conn*, *xid*, *is\_prepared*)

Intercept commit\_twophase() events.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeEngine, 'commit\_twophase')

**def** receive\_commit\_twophase(conn, xid, is\_prepared):

"listen for the 'commit\_twophase' event"

*# ... (event handling logic) ...*

|  |  |
| --- | --- |
| **Parameters:** | * ****conn**** – [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) object * ****xid**** – two-phase XID identifier * ****is\_prepared**** – boolean, indicates if [TwoPhaseTransaction.prepare()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.TwoPhaseTransaction.prepare" \o "sqlalchemy.engine.TwoPhaseTransaction.prepare) was called. |

**dbapi\_error**(*conn*, *cursor*, *statement*, *parameters*, *context*, *exception*)

Intercept a raw DBAPI error.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeEngine, 'dbapi\_error')

**def** receive\_dbapi\_error(conn, cursor, statement, parameters, context, exception):

"listen for the 'dbapi\_error' event"

*# ... (event handling logic) ...*

*# named argument style (new in 0.9)*

**@event**.listens\_for(SomeEngine, 'dbapi\_error', named=**True**)

**def** receive\_dbapi\_error(\*\*kw):

"listen for the 'dbapi\_error' event"

conn = kw['conn']

cursor = kw['cursor']

*# ... (event handling logic) ...*

This event is called with the DBAPI exception instance received from the DBAPI itself, *before* SQLAlchemy wraps the exception with it's own exception wrappers, and before any other operations are performed on the DBAPI cursor; the existing transaction remains in effect as well as any state on the cursor.

The use case here is to inject low-level exception handling into an [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine), typically for logging and debugging purposes.

**Warning**

Code should ****not**** modify any state or throw any exceptions here as this will interfere with SQLAlchemy's cleanup and error handling routines. For exception modification, please refer to the new [ConnectionEvents.handle\_error()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents.handle_error" \o "sqlalchemy.events.ConnectionEvents.handle_error) event.

Subsequent to this hook, SQLAlchemy may attempt any number of operations on the connection/cursor, including closing the cursor, rolling back of the transaction in the case of connectionless execution, and disposing of the entire connection pool if a "disconnect" was detected. The exception is then wrapped in a SQLAlchemy DBAPI exception wrapper and re-thrown.

|  |  |
| --- | --- |
| **Parameters:** | * ****conn**** – [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) object * ****cursor**** – DBAPI cursor object * ****statement**** – string SQL statement, as passed to the DBAPI * ****parameters**** – Dictionary, tuple, or list of parameters being passed to the execute() or executemany() method of the DBAPI cursor. In some cases may be None. * ****context**** – [ExecutionContext](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.ExecutionContext" \o "sqlalchemy.engine.interfaces.ExecutionContext) object in use. May be None. * ****exception**** – The ****unwrapped**** exception emitted directly from the DBAPI. The class here is specific to the DBAPI module in use. |

*Deprecated since version 0.9.7:*- replaced by [ConnectionEvents.handle\_error()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents.handle_error" \o "sqlalchemy.events.ConnectionEvents.handle_error)

**engine\_connect**(*conn*, *branch*)

Intercept the creation of a new [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection).

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeEngine, 'engine\_connect')

**def** receive\_engine\_connect(conn, branch):

"listen for the 'engine\_connect' event"

*# ... (event handling logic) ...*

This event is called typically as the direct result of calling the [Engine.connect()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.connect" \o "sqlalchemy.engine.Engine.connect) method.

这个事件通常被称为调用Engine.connect() 方法的直接结果。

It differs from the [PoolEvents.connect()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.PoolEvents.connect" \o "sqlalchemy.events.PoolEvents.connect) method, which refers to the actual connection to a database at the DBAPI level; a DBAPI connection may be pooled and reused for many operations. In contrast, this event refers only to the production of a higher level [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) wrapper around such a DBAPI connection.

它与PoolEvents.connect() 方法不同，它引用了DBAPI级别到数据库的实际连接; DBAPI连接可能被汇集并重用于许多操作。相比之下，此事件仅指生产围绕这样的DBAPI连接的更高级别的连接包装。

It also differs from the [PoolEvents.checkout()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.PoolEvents.checkout" \o "sqlalchemy.events.PoolEvents.checkout) event in that it is specific to the [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) object, not the DBAPI connection that [PoolEvents.checkout()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.PoolEvents.checkout" \o "sqlalchemy.events.PoolEvents.checkout) deals with, although this DBAPI connection is available here via the [Connection.connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection.connection" \o "sqlalchemy.engine.Connection.connection) attribute. But note there can in fact be multiple [PoolEvents.checkout()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.PoolEvents.checkout" \o "sqlalchemy.events.PoolEvents.checkout) events within the lifespan of a single [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) object, if that [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) is invalidated and re-established. There can also be multiple [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) objects generated for the same already-checked-out DBAPI connection, in the case that a "branch" of a [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) is produced.

它也不同于PoolEvents.checkout() 事件，因为它特定于Connection对象，而不是PoolEvents.checkout() 处理的DBAPI连接，尽管此DBAPI连接可通过Connection.connection属性在此处使用。但是请注意，如果Connection无效并重新建立，实际上可能会在单个Connection对象的生存期内存在多个PoolEvents.checkout() 事件。在生成连接的"分支"的情况下，也可以为同一已经签出的DBAPI连接生成多个连接对象。

|  |  |
| --- | --- |
| **Parameters:** | * ****conn**** – [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) object. * ****branch**** – if True, this is a "branch" of an existing [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection). A branch is generated within the course of a statement execution to invoke supplemental statements, most typically to pre-execute a SELECT of a default value for the purposes of an INSERT statement. |

*New in version 0.9.0.*

**See also**

[Disconnect Handling - Pessimistic](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "pool-disconnects-pessimistic) - illustrates how to use [ConnectionEvents.engine\_connect()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents.engine_connect" \o "sqlalchemy.events.ConnectionEvents.engine_connect) to transparently ensure pooled connections are connected to the database.

[PoolEvents.checkout()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.PoolEvents.checkout" \o "sqlalchemy.events.PoolEvents.checkout) the lower-level pool checkout event for an individual DBAPI connection

[ConnectionEvents.set\_connection\_execution\_options()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents.set_connection_execution_options" \o "sqlalchemy.events.ConnectionEvents.set_connection_execution_options) - a copy of a [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) is also made when 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)method is called.

**engine\_disposed**(*engine*)

Intercept when the [Engine.dispose()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.dispose" \o "sqlalchemy.engine.Engine.dispose) method is called.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeEngine, 'engine\_disposed')

**def** receive\_engine\_disposed(engine):

"listen for the 'engine\_disposed' event"

*# ... (event handling logic) ...*

The [Engine.dispose()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.dispose" \o "sqlalchemy.engine.Engine.dispose) method instructs the engine to "dispose" of it's connection pool (e.g. [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool)), and replaces it with a new one. Disposing of the old pool has the effect that existing checked-in connections are closed. The new pool does not establish any new connections until it is first used.

Engine.dispose() 方法指示引擎"配置"其连接池（例如池），并将其替换为新的连接池。 处理旧池具有已关闭的现有签入连接的效果。 新池在首次使用之前不会建立任何新的连接。

This event can be used to indicate that resources related to the [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) should also be cleaned up, keeping in mind that the [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) can still be used for new requests in which case it re-acquires connection resources.

此事件可用于指示与引擎相关的资源也应被清除，同时请注意，引擎仍然可以用于新的请求，在这种情况下，它会重新获取连接资源。

*New in version 1.0.5.*

**handle\_error**(*exception\_context*)

Intercept all exceptions processed by the [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection).

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeEngine, 'handle\_error')

**def** receive\_handle\_error(exception\_context):

"listen for the 'handle\_error' event"

*# ... (event handling logic) ...*

This includes all exceptions emitted by the DBAPI as well as within SQLAlchemy's statement invocation process, including encoding errors and other statement validation errors. Other areas in which the event is invoked include transaction begin and end, result row fetching, cursor creation.

这包括DBAPI发出的所有异常以及SQLAlchemy的语句调用过程中的所有异常，包括编码错误和其他语句验证错误。 调用事件的其他区域包括事务开始和结束，结果行提取，游标创建。

Note that [handle\_error()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents.handle_error" \o "sqlalchemy.events.ConnectionEvents.handle_error) may support new kinds of exceptions and new calling scenarios at *any time*. Code which uses this event must expect new calling patterns to be present in minor releases.

请注意，[handle\_error()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents.handle_error" \o "sqlalchemy.events.ConnectionEvents.handle_error)可以随时支持新的异常和新的调用方案。 使用此事件的代码必须预期新的调用模式将存在于次要版本中。

To support the wide variety of members that correspond to an exception, as well as to allow extensibility of the event without backwards incompatibility, the sole argument received is an instance of [ExceptionContext](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ExceptionContext" \o "sqlalchemy.engine.ExceptionContext). This object contains data members representing detail about the exception.

为了支持与异常相对应的各种成员，并且允许事件的可扩展性而不会出现向后不兼容，所接收的唯一参数是[ExceptionContext](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ExceptionContext" \o "sqlalchemy.engine.ExceptionContext).的一个实例。 此对象包含表示异常详细信息的数据成员。

Use cases supported by this hook include:

此钩子支持的用例包括：

* read-only, low-level exception handling for logging and debugging purposes只读，低级异常处理用于日志记录和调试目的
* exception re-writing异常重写
* Establishing or disabling whether a connection or the owning connection pool is invalidated or expired in response to a specific exception.建立或禁用连接或拥有的连接池是否响应特定异常而失效或过期。

The hook is called while the cursor from the failed operation (if any) is still open and accessible. Special cleanup operations can be called on this cursor; SQLAlchemy will attempt to close this cursor subsequent to this hook being invoked. If the connection is in "autocommit" mode, the transaction also remains open within the scope of this hook; the rollback of the per-statement transaction also occurs after the hook is called.

当来自失败操作（如果有的话）的游标仍然打开并可访问时，调用该钩子。 可以在此游标上调用特殊的清理操作; 在调用此钩子之后，SQLAlchemy将尝试关闭此游标。 如果连接处于"autocommi"模式，交易在该挂钩的范围内也保持打开; 每个语句事务的回滚也发生在调用钩子之后。

The user-defined event handler has two options for replacing the SQLAlchemy-constructed exception into one that is user defined. It can either raise this new exception directly, in which case all further event listeners are bypassed and the exception will be raised, after appropriate cleanup as taken place:

用户定义的事件处理程序有两个用于将SQLAlchemy构造的异常替换为用户定义的异常的选项。 它可以直接引发这个新的异常，在这种情况下，所有进一步的事件侦听器都被绕过，并且在适当的清理发生之后会引发异常：

**@event**.listens\_for(Engine, "handle\_error")

**def** handle\_exception(context):

**if** isinstance(context.original\_exception,

psycopg2.OperationalError) **and** \

"failed" **in** str(context.original\_exception):

**raise** MySpecialException("failed operation")

**Warning**

Because the [ConnectionEvents.handle\_error()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents.handle_error" \o "sqlalchemy.events.ConnectionEvents.handle_error) event specifically provides for exceptions to be re-thrown as the ultimate exception raised by the failed statement, ****stack traces will be misleading**** if the user-defined event handler itself fails and throws an unexpected exception; the stack trace may not illustrate the actual code line that failed! It is advised to code carefully here and use logging and/or inline debugging if unexpected exceptions are occurring.

因为[ConnectionEvents.handle\_error()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents.handle_error" \o "sqlalchemy.events.ConnectionEvents.handle_error)事件特别提供了异常作为失败语句引发的最终异常重新抛出，如果用户定义的事件处理程序本身失败并引发意外的异常，堆栈跟踪将会产生误导; 堆栈跟踪可能无法说明失败的实际代码行！ 建议在这里仔细进行编码，如果发生意外异常，请使用日志记录和/或内联调试。

Alternatively, a "chained" style of event handling can be used, by configuring the handler with the retval=True modifier and returning the new exception instance from the function. In this case, event handling will continue onto the next handler. The "chained" exception is available using[ExceptionContext.chained\_exception](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ExceptionContext.chained_exception" \o "sqlalchemy.engine.ExceptionContext.chained_exception):

或者，可以通过使用retval=True修饰符配置处理程序并从函数返回新的异常实例来使用"链式"的事件处理方式。 在这种情况下，事件处理将继续到下一个处理程序。 "chained"异常可用[ExceptionContext.chained\_exception](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ExceptionContext.chained_exception" \o "sqlalchemy.engine.ExceptionContext.chained_exception)：

**@event**.listens\_for(Engine, "handle\_error", retval=**True**)

**def** handle\_exception(context):

**if** context.chained\_exception **is** **not** **None** **and** \

"special" **in** context.chained\_exception.message:

**return** MySpecialException("failed",

cause=context.chained\_exception)

Handlers that return None may remain within this chain; the last non-None return value is the one that continues to be passed to the next handler.

返回None的处理程序可能保留在此链中; 最后一个非无返回值是继续传递给下一个处理程序的值。

When a custom exception is raised or returned, SQLAlchemy raises this new exception as-is, it is not wrapped by any SQLAlchemy object. If the exception is not a subclass of [sqlalchemy.exc.StatementError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.StatementError" \o "sqlalchemy.exc.StatementError), certain features may not be available; currently this includes the ORM's feature of adding a detail hint about "autoflush" to exceptions raised within the autoflush process.

当引发或返回自定义异常时，SQLAlchemy按原样引发新的异常，它不会被任何SQLAlchemy对象包装。 如果异常不是[sqlalchemy.exc.StatementError](http://docs.sqlalchemy.org/en/rel_1_1/core/exceptions.html" \l "sqlalchemy.exc.StatementError" \o "sqlalchemy.exc.StatementError)的子类，某些功能可能不可用; 目前，这包括向autoflush 过程中引发的异常添加有关"autoflush"的详细信息提示的功能。

|  |  |
| --- | --- |
| **Parameters:** | ****context**** – an [ExceptionContext](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ExceptionContext" \o "sqlalchemy.engine.ExceptionContext) object. See this class for details on all available members. |

*New in version 0.9.7:*Added the [ConnectionEvents.handle\_error()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents.handle_error" \o "sqlalchemy.events.ConnectionEvents.handle_error) hook.

*Changed in version 1.1:*The [handle\_error()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents.handle_error" \o "sqlalchemy.events.ConnectionEvents.handle_error) event will now receive all exceptions that inherit from BaseException, including SystemExit and KeyboardInterrupt. The setting for [ExceptionContext.is\_disconnect](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ExceptionContext.is_disconnect" \o "sqlalchemy.engine.ExceptionContext.is_disconnect) is True in this case and the default for [ExceptionContext.invalidate\_pool\_on\_disconnect](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.ExceptionContext.invalidate_pool_on_disconnect" \o "sqlalchemy.engine.ExceptionContext.invalidate_pool_on_disconnect) is False.

1.1版更改：handle\_error() 事件现在将接收所有从BaseException继承的异常，包括SystemExit和KeyboardInterrupt。在这种情况下，ExceptionContext.is\_disconnect的设置为True，ExceptionContext.invalidate\_pool\_on\_disconnect的默认值为False。

*Changed in version 1.0.0:*The [handle\_error()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents.handle_error" \o "sqlalchemy.events.ConnectionEvents.handle_error) event is now invoked when an [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) fails during the initial call to [Engine.connect()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine.connect" \o "sqlalchemy.engine.Engine.connect), as well as when a [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) object encounters an error during a reconnect operation.

在版本1.0.0中更改：当引擎在初始调用Engine.connect() 期间发生故障时，以及一个Connection对象在重新连接操作期间遇到错误时，将调用[handle\_error()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents.handle_error" \o "sqlalchemy.events.ConnectionEvents.handle_error)事件。

*Changed in version 1.0.0:*The [handle\_error()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents.handle_error" \o "sqlalchemy.events.ConnectionEvents.handle_error) event is not fired off when a dialect makes use of the skip\_user\_error\_events execution option. This is used by dialects which intend to catch SQLAlchemy-specific exceptions within specific operations, such as when the MySQL dialect detects a table not present within the has\_table() dialect method. Prior to 1.0.0, code which implements [handle\_error()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents.handle_error" \o "sqlalchemy.events.ConnectionEvents.handle_error) needs to ensure that exceptions thrown in these scenarios are re-raised without modification.

在版本1.0.0中更改：当方言使用skip\_user\_error\_events执行选项时，[handle\_error()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents.handle_error" \o "sqlalchemy.events.ConnectionEvents.handle_error)事件不会被关闭。这是由特定操作中捕获SQLAlchemy特定异常的方言使用的，例如当MySQL方言检测到has\_table()方言中不存在的表时。在1.0.0之前，实现[handle\_error()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents.handle_error" \o "sqlalchemy.events.ConnectionEvents.handle_error)的代码需要确保在这些方案中抛出的异常被重新提升而无需修改。

**prepare\_twophase**(*conn*, *xid*)

Intercept prepare\_twophase() events.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeEngine, 'prepare\_twophase')

**def** receive\_prepare\_twophase(conn, xid):

"listen for the 'prepare\_twophase' event"

*# ... (event handling logic) ...*

|  |  |
| --- | --- |
| **Parameters:** | * ****conn**** – [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) object * ****xid**** – two-phase XID identifier |

**release\_savepoint**(*conn*, *name*, *context*)

Intercept release\_savepoint() events.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeEngine, 'release\_savepoint')

**def** receive\_release\_savepoint(conn, name, context):

"listen for the 'release\_savepoint' event"

*# ... (event handling logic) ...*

|  |  |
| --- | --- |
| **Parameters:** | * ****conn**** – [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) object * ****name**** – specified name used for the savepoint. * ****context**** – [ExecutionContext](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.ExecutionContext" \o "sqlalchemy.engine.interfaces.ExecutionContext) in use. May be None. |

**rollback**(*conn*)

Intercept rollback() events, as initiated by a [Transaction](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Transaction" \o "sqlalchemy.engine.Transaction).

Intercept rollback() 事件，由事务发起。

Example argument forms:

示例参数形式：

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeEngine, 'rollback')

**def** receive\_rollback(conn):

"listen for the 'rollback' event"

*# ... (event handling logic) ...*

Note that the [Pool](http://docs.sqlalchemy.org/en/rel_1_1/core/pooling.html" \l "sqlalchemy.pool.Pool" \o "sqlalchemy.pool.Pool) also "auto-rolls back" a DBAPI connection upon checkin, if the reset\_on\_return flag is set to its default value of 'rollback'. To intercept this rollback, use the [PoolEvents.reset()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.PoolEvents.reset" \o "sqlalchemy.events.PoolEvents.reset) hook.

请注意，如果reset\_on\_return标志设置为其默认值“rollback”，则池也会在签入时“自动回滚”一个DBAPI连接。 要拦截此回滚，请使用PoolEvents.reset() 钩子。

|  |  |
| --- | --- |
| **Parameters:** | ****conn**** – [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) object |

**See also**

[PoolEvents.reset()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.PoolEvents.reset" \o "sqlalchemy.events.PoolEvents.reset)

**rollback\_savepoint**(*conn*, *name*, *context*)

Intercept rollback\_savepoint() events.

拦截rollback\_savepoint() 事件。

Example argument forms:

示例参数形式：

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeEngine, 'rollback\_savepoint')

**def** receive\_rollback\_savepoint(conn, name, context):

"listen for the 'rollback\_savepoint' event"

*# ... (event handling logic) ...*

|  |  |
| --- | --- |
| **Parameters:** | * ****conn**** – [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) object * ****name**** – specified name used for the savepoint. * ****context**** – [ExecutionContext](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.ExecutionContext" \o "sqlalchemy.engine.interfaces.ExecutionContext) in use. May be None. |

**rollback\_twophase**(*conn*, *xid*, *is\_prepared*)

Intercept rollback\_twophase() events.

拦截rollback\_twophase() 事件。

Example argument forms:

示例参数形式：

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeEngine, 'rollback\_twophase')

**def** receive\_rollback\_twophase(conn, xid, is\_prepared):

"listen for the 'rollback\_twophase' event"

*# ... (event handling logic) ...*

|  |  |
| --- | --- |
| **Parameters:** | * ****conn**** – [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) object * ****xid**** – two-phase XID identifier * ****is\_prepared**** – boolean, indicates if [TwoPhaseTransaction.prepare()](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.TwoPhaseTransaction.prepare" \o "sqlalchemy.engine.TwoPhaseTransaction.prepare) was called. |

**savepoint**(*conn*, *name*)

Intercept savepoint() events.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeEngine, 'savepoint')

**def** receive\_savepoint(conn, name):

"listen for the 'savepoint' event"

*# ... (event handling logic) ...*

|  |  |
| --- | --- |
| **Parameters:** | * ****conn**** – [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) object * ****name**** – specified name used for the savepoint. |

**set\_connection\_execution\_options**(*conn*, *opts*)

Intercept when 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) method is called.

在调用Connection.execution\_options() 方法时侦听。

Example argument forms:

示例参数表

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeEngine, 'set\_connection\_execution\_options')

**def** receive\_set\_connection\_execution\_options(conn, opts):

"listen for the 'set\_connection\_execution\_options' event"

*# ... (event handling logic) ...*

This method is called after the new [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) has been produced, with the newly updated execution options collection, but before the [Dialect](http://docs.sqlalchemy.org/en/rel_1_1/core/internals.html" \l "sqlalchemy.engine.interfaces.Dialect" \o "sqlalchemy.engine.interfaces.Dialect) has acted upon any of those new options.

在新的连接生成之后，新方法将被调用，并且在新的更新的执行选项集合之前，但在方言对这些新选项执行操作之前。

Note that this method is not called when a new [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) is produced which is inheriting execution options from its parent [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine); to intercept this condition, use the [ConnectionEvents.engine\_connect()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents.engine_connect" \o "sqlalchemy.events.ConnectionEvents.engine_connect) event.

注意，当生成新的连接时，该方法不会被从其父引擎继承执行选项; 要拦截此条件，请使用ConnectionEvents.engine\_connect() 事件。

|  |  |
| --- | --- |
| **Parameters:** | * ****conn**** – The newly copied [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) object * ****opts**** – dictionary of options that were passed to 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) method. |

*New in version 0.9.0.*

**See also**

[ConnectionEvents.set\_engine\_execution\_options()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents.set_engine_execution_options" \o "sqlalchemy.events.ConnectionEvents.set_engine_execution_options) - event which is called when [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) is called.

**set\_engine\_execution\_options**(*engine*, *opts*)

Intercept when 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 is called.

当引用Engine.execution\_options() 方法时侦听。

Example argument forms:

示例参数形式：

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeEngine, 'set\_engine\_execution\_options')

**def** receive\_set\_engine\_execution\_options(engine, opts):

"listen for the 'set\_engine\_execution\_options' event"

*# ... (event handling logic) ...*

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 produces a shallow copy of the [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) which stores the new options. That new [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) is passed here. A particular application of this method is to add a [ConnectionEvents.engine\_connect()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents.engine_connect" \o "sqlalchemy.events.ConnectionEvents.engine_connect) event handler to the given [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) which will perform some per- [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) task specific to these execution options.

Engine.execution\_options() 方法生成一个存储新选项的引擎的浅拷贝。 那个新引擎在这里传递。 此方法的一个特殊应用是向给定的引擎添加一个ConnectionEvents.engine\_connect() 事件处理程序，该引擎将执行特定于这些执行选项的某些连接任务。

|  |  |
| --- | --- |
| **Parameters:** | * ****conn**** – The newly copied [Engine](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Engine" \o "sqlalchemy.engine.Engine) object * ****opts**** – dictionary of options that were passed to 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) method. |

*New in version 0.9.0.*

**See also**

[ConnectionEvents.set\_connection\_execution\_options()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents.set_connection_execution_options" \o "sqlalchemy.events.ConnectionEvents.set_connection_execution_options) - event which is called when [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) is called.

*class*sqlalchemy.events.**DialectEvents**

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)

event interface for execution-replacement functions.

事件接口用于执行替换功能。

These events allow direct instrumentation and replacement of key dialect functions which interact with the DBAPI.

这些事件允许直接检测和更换与DBAPI交互的关键方言功能。

**Note**注意

[DialectEvents](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.DialectEvents" \o "sqlalchemy.events.DialectEvents) hooks should be considered ****semi-public**** and experimental. These hooks are not for general use and are only for those situations where intricate re-statement of DBAPI mechanics must be injected onto an existing dialect. For general-use statement-interception events, please use the [ConnectionEvents](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents" \o "sqlalchemy.events.ConnectionEvents) interface.

DialectEvents钩子应该被认为是半公开和实验性的。 这些钩子不是一般用途，只适用于必须将DBAPI力学的错误重新声明注入现有方言的情况。 对于一般使用的语句拦截事件，请使用ConnectionEvents接口。

**See also**

[ConnectionEvents.before\_cursor\_execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents.before_cursor_execute" \o "sqlalchemy.events.ConnectionEvents.before_cursor_execute)

[ConnectionEvents.before\_execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents.before_execute" \o "sqlalchemy.events.ConnectionEvents.before_execute)

[ConnectionEvents.after\_cursor\_execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents.after_cursor_execute" \o "sqlalchemy.events.ConnectionEvents.after_cursor_execute)

[ConnectionEvents.after\_execute()](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.ConnectionEvents.after_execute" \o "sqlalchemy.events.ConnectionEvents.after_execute)

*New in version 0.9.4.*

**do\_connect**(*dialect*, *conn\_rec*, *cargs*, *cparams*)

Receive connection arguments before a connection is made.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeEngine, 'do\_connect')

**def** receive\_do\_connect(dialect, conn\_rec, cargs, cparams):

"listen for the 'do\_connect' event"

*# ... (event handling logic) ...*

*# named argument style (new in 0.9)*

**@event**.listens\_for(SomeEngine, 'do\_connect', named=**True**)

**def** receive\_do\_connect(\*\*kw):

"listen for the 'do\_connect' event"

dialect = kw['dialect']

conn\_rec = kw['conn\_rec']

*# ... (event handling logic) ...*

Return a DBAPI connection to halt further events from invoking; the returned connection will be used.

返回DBAPI连接以停止调用进一步的事件; 将使用返回的连接。

Alternatively, the event can manipulate the cargs and/or cparams collections; cargs will always be a Python list that can be mutated in-place and cparams a Python dictionary. Return None to allow control to pass to the next event handler and ultimately to allow the dialect to connect normally, given the updated arguments.

或者，事件可以操纵cargs和/或cparams集合; cargs将一直是一个Python列表，可以在现场进行突变，并且可以使用cparams Python字典。 返回None，允许控制传递给下一个事件处理程序，并最终允许方言正常连接，给定更新的参数。

*New in version 1.0.3.*

**do\_execute**(*cursor*, *statement*, *parameters*, *context*)

Receive a cursor to have execute() called.

接收一个游标以使execute() 被调用。

Example argument forms:

示例参数形式：

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeEngine, 'do\_execute')

**def** receive\_do\_execute(cursor, statement, parameters, context):

"listen for the 'do\_execute' event"

*# ... (event handling logic) ...*

*# named argument style (new in 0.9)*

**@event**.listens\_for(SomeEngine, 'do\_execute', named=**True**)**def** receive\_do\_execute(\*\*kw):

"listen for the 'do\_execute' event"

cursor = kw['cursor']

statement = kw['statement']

*# ... (event handling logic) ...*

Return the value True to halt further events from invoking, and to indicate that the cursor execution has already taken place within the event handler.

返回值True以停止调用进一步的事件，并指示游标执行已经在事件处理程序中进行。

**do\_execute\_no\_params**(*cursor*, *statement*, *context*)

Receive a cursor to have execute() with no parameters called.

接收一个游标以使execute() 没有调用参数。

Example argument forms:

示例参数形式：

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeEngine, 'do\_execute\_no\_params')

**def** receive\_do\_execute\_no\_params(cursor, statement, context):

"listen for the 'do\_execute\_no\_params' event"

*# ... (event handling logic) ...*

Return the value True to halt further events from invoking, and to indicate that the cursor execution has already taken place within the event handler.

返回值True以停止调用进一步的事件，并指示游标执行已经在事件处理程序中进行。

**do\_executemany**(*cursor*, *statement*, *parameters*, *context*)

Receive a cursor to have executemany() called.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeEngine, 'do\_executemany')

**def** receive\_do\_executemany(cursor, statement, parameters, context):

"listen for the 'do\_executemany' event"

*# ... (event handling logic) ...*

*# named argument style (new in 0.9)*

**@event**.listens\_for(SomeEngine, 'do\_executemany', named=**True**)

**def** receive\_do\_executemany(\*\*kw):

"listen for the 'do\_executemany' event"

cursor = kw['cursor']

statement = kw['statement']

*# ... (event handling logic) ...*

Return the value True to halt further events from invoking, and to indicate that the cursor execution has already taken place within the event handler.

返回值True以停止调用进一步的事件，并指示游标执行已经在事件处理程序中进行。

5.4.3 Schema Events

*class*sqlalchemy.events.**DDLEvents**

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 event listeners for schema objects, that is, [SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem) and other [SchemaEventTarget](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.SchemaEventTarget" \o "sqlalchemy.events.SchemaEventTarget) subclasses, including [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData), [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table), [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column).

为模式对象定义事件监听器，即[SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem)和其他[SchemaEventTarget](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.SchemaEventTarget" \o "sqlalchemy.events.SchemaEventTarget)子类，包括[MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData)，[Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table)，[Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column)。

[MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) and [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) support events specifically regarding when CREATE and DROP DDL is emitted to the database.

MetaData和Table支持事件，特别是当CREATE和DROP DDL发送到数据库时。

Attachment events are also provided to customize behavior whenever a child schema element is associated with a parent, such as, when a [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) is associated with its [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table), when a [ForeignKeyConstraint](http://docs.sqlalchemy.org/en/rel_1_1/core/constraints.html" \l "sqlalchemy.schema.ForeignKeyConstraint" \o "sqlalchemy.schema.ForeignKeyConstraint) is associated with a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table), etc.

还提供了附件事件，用于在子模式元素与父级关联时自定义行为，例如当Column与其关联时，当ForeignKeyConstraint与表相关联时，等等。

Example using the after\_create event:

使用after\_create事件的示例：

**from** **sqlalchemy** **import** event

**from** **sqlalchemy** **import** Table, Column, Metadata, Integer

m = MetaData()

some\_table = Table('some\_table', m, Column('data', Integer))

**def** after\_create(target, connection, \*\*kw):

connection.execute("ALTER TABLE *%s* SET name=foo\_*%s*" %

(target.name, target.name))

event.listen(some\_table, "after\_create", after\_create)

DDL events integrate closely with the [DDL](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.DDL" \o "sqlalchemy.schema.DDL) class and the [DDLElement](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.DDLElement" \o "sqlalchemy.schema.DDLElement) hierarchy of DDL clause constructs, which are themselves appropriate as listener callables:

DDL事件与DDL类和DDL子句构造的DDLElement层次结构紧密集成，它们本身适合作为侦听器调用：

**from** **sqlalchemy** **import** DDL

event.listen(

some\_table,

"after\_create",

DDL("ALTER TABLE *%(table)s* SET name=foo\_*%(table)s*"))

The methods here define the name of an event as well as the names of members that are passed to listener functions.

这里的方法定义事件的名称以及传递给监听器函数的成员的名称。

See also:

[Events](http://docs.sqlalchemy.org/en/rel_1_1/core/event.html)

[DDLElement](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.DDLElement" \o "sqlalchemy.schema.DDLElement)

[DDL](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "sqlalchemy.schema.DDL" \o "sqlalchemy.schema.DDL)

[Controlling DDL Sequences](http://docs.sqlalchemy.org/en/rel_1_1/core/ddl.html" \l "schema-ddl-sequences)

**after\_create**(*target*, *connection*, *\*\*kw*)

Called after CREATE statements are emitted.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeSchemaClassOrObject, 'after\_create')

**def** receive\_after\_create(target, connection, \*\*kw):

"listen for the 'after\_create' event"

*# ... (event handling logic) ...*

|  |  |
| --- | --- |
| **Parameters:** | * ****target**** – the [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) or [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object which is the target of the event. * ****connection**** – the [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) where the CREATE statement or statements have been emitted. * ****\*\*kw**** – additional keyword arguments relevant to the event. The contents of this dictionary may vary across releases, and include the list of tables being generated for a metadata-level event, the checkfirst flag, and other elements used by internal events. |

**after\_drop**(*target*, *connection*, *\*\*kw*)

Called after DROP statements are emitted.

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeSchemaClassOrObject, 'after\_drop')

**def** receive\_after\_drop(target, connection, \*\*kw):

"listen for the 'after\_drop' event"

*# ... (event handling logic) ...*

|  |  |
| --- | --- |
| **Parameters:** | * ****target**** – the [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) or [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object which is the target of the event. * ****connection**** – the [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) where the DROP statement or statements have been emitted. * ****\*\*kw**** – additional keyword arguments relevant to the event. The contents of this dictionary may vary across releases, and include the list of tables being generated for a metadata-level event, the checkfirst flag, and other elements used by internal events. |

**after\_parent\_attach**(*target*, *parent*)

Called after a [SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem) is associated with a parent [SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem).

Example argument forms:

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeSchemaClassOrObject, 'after\_parent\_attach')

**def** receive\_after\_parent\_attach(target, parent):

"listen for the 'after\_parent\_attach' event"

*# ... (event handling logic) ...*

|  |  |
| --- | --- |
| **Parameters:** | * ****target**** – the target object * ****parent**** – the parent to which the target is being attached. |

[event.listen()](http://docs.sqlalchemy.org/en/rel_1_1/core/event.html" \l "sqlalchemy.event.listen" \o "sqlalchemy.event.listen) also accepts a modifier for this event:

event.listen（）也接受此事件的修饰符：

|  |  |
| --- | --- |
| **Parameters:** | ****propagate=False**** – When True, the listener function will be established for any copies made of the target object, i.e. those copies that are generated when[Table.tometadata()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.tometadata" \o "sqlalchemy.schema.Table.tometadata) is used.当为True时，将为由目标对象创建的任何副本（即使用whenTable.tometadata（））生成的副本建立侦听器功能。 |

**before\_create**(*target*, *connection*, *\*\*kw*)

Called before CREATE statements are emitted.

在发出CREATE语句之前调用。

Example argument forms:

示例参数形式：

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeSchemaClassOrObject, 'before\_create')

**def** receive\_before\_create(target, connection, \*\*kw):

"listen for the 'before\_create' event"

*# ... (event handling logic) ...*

|  |  |
| --- | --- |
| **Parameters:** | * ****target**** – the [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) or [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object which is the target of the event.作为事件目标的MetaData或Table对象。 * ****connection**** – the [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) where the CREATE statement or statements will be emitted.将发出CREATE语句或语句的Connection。 * ****\*\*kw**** – additional keyword arguments relevant to the event. The contents of this dictionary may vary across releases, and include the list of tables being generated for a metadata-level event, the checkfirst flag, and other elements used by internal events.与事件相关的其他关键字参数。 该字典的内容可能会随版本而有所不同，并且包括为元数据级别事件，checkfirst标志和内部事件使用的其他元素生成的表的列表。 |

**before\_drop**(*target*, *connection*, *\*\*kw*)

Called before DROP statements are emitted.

在发出DROP语句之前调用。

Example argument forms:

示例参数形式：

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeSchemaClassOrObject, 'before\_drop')

**def** receive\_before\_drop(target, connection, \*\*kw):

"listen for the 'before\_drop' event"

*# ... (event handling logic) ...*

|  |  |
| --- | --- |
| **Parameters:** | * ****target**** – the [MetaData](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) or [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) object which is the target of the event.作为事件目标的MetaData或Table对象。 * ****connection**** – the [Connection](http://docs.sqlalchemy.org/en/rel_1_1/core/connections.html" \l "sqlalchemy.engine.Connection" \o "sqlalchemy.engine.Connection) where the DROP statement or statements will be emitted.连接 - 将发出DROP语句或语句的连接。 * ****\*\*kw**** – additional keyword arguments relevant to the event. The contents of this dictionary may vary across releases, and include the list of tables being generated for a metadata-level event, the checkfirst flag, and other elements used by internal events. * 与事件相关的其他关键字参数。 该字典的内容可能会随版本而有所不同，并且包括为元数据级别事件，checkfirst标志和内部事件使用的其他元素生成的表的列表。 |

**before\_parent\_attach**(*target*, *parent*)

Called before a [SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem) is associated with a parent [SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem).

在SchemaItem与父SchemaItem相关联之前调用。

Example argument forms:

示例参数表

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeSchemaClassOrObject, 'before\_parent\_attach')

**def** receive\_before\_parent\_attach(target, parent):

"listen for the 'before\_parent\_attach' event"

*# ... (event handling logic) ...*

|  |  |
| --- | --- |
| **Parameters:** | * ****target**** – the target object目标对象 * ****parent**** – the parent to which the target is being attached.正在附加目标的父级。 |

[event.listen()](http://docs.sqlalchemy.org/en/rel_1_1/core/event.html" \l "sqlalchemy.event.listen" \o "sqlalchemy.event.listen) also accepts a modifier for this event:

[event.listen()](http://docs.sqlalchemy.org/en/rel_1_1/core/event.html" \l "sqlalchemy.event.listen" \o "sqlalchemy.event.listen)也接受此事件的修饰符：

|  |  |
| --- | --- |
| **Parameters:** | ****propagate=False**** – When True, the listener function will be established for any copies made of the target object, i.e. those copies that are generated when[Table.tometadata()](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table.tometadata" \o "sqlalchemy.schema.Table.tometadata) is used.当为True时，将为由目标对象创建的任何副本（即使用whenTable.tometadata（））生成的副本建立侦听器功能。 |

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

Called for each unit of 'column info' retrieved when a [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) is being reflected.

当表被反映时，被称为每个单元的“列信息”。

Example argument forms:

示例参数形式：

**from** **sqlalchemy** **import** event

*# standard decorator style*

**@event**.listens\_for(SomeSchemaClassOrObject, 'column\_reflect')

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

"listen for the 'column\_reflect' event"

*# ... (event handling logic) ...*

The dictionary of column information as returned by the dialect is passed, and can be modified. The dictionary is that returned in each element of the list returned by [reflection.Inspector.get\_columns()](http://docs.sqlalchemy.org/en/rel_1_1/core/reflection.html" \l "sqlalchemy.engine.reflection.Inspector.get_columns" \o "sqlalchemy.engine.reflection.Inspector.get_columns):

通过方言返回的列信息字典，可以修改。 该字典是在reflection.Inspector.get\_columns() 返回的列表的每个元素中返回的：

name - the column's name

列的名称

type - the type of this column, which should be an instance of [TypeEngine](http://docs.sqlalchemy.org/en/rel_1_1/core/type_api.html" \l "sqlalchemy.types.TypeEngine" \o "sqlalchemy.types.TypeEngine)

该列的类型应该是TypeEngine的一个实例

nullable - boolean flag if the column is NULL or NOT NULL

如果列为NULL或NOT NULL，则为空 - boolean标志

default - the column's server default value. This is normally specified as a plain string SQL expression, however the event can pass a [FetchedValue](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.FetchedValue" \o "sqlalchemy.schema.FetchedValue), [DefaultClause](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.DefaultClause" \o "sqlalchemy.schema.DefaultClause), or [sql.expression.text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text) object as well.

默认 - 列的服务器默认值。 这通常指定为纯字符串SQL表达式，但事件也可以传递FetchedValue，DefaultClause或sql.expression.text() 对象。

*Changed in version 1.1.6:*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 allows a non string [FetchedValue](http://docs.sqlalchemy.org/en/rel_1_1/core/defaults.html" \l "sqlalchemy.schema.FetchedValue" \o "sqlalchemy.schema.FetchedValue), [sql.expression.text()](http://docs.sqlalchemy.org/en/rel_1_1/core/sqlelement.html" \l "sqlalchemy.sql.expression.text" \o "sqlalchemy.sql.expression.text), or derived object to be specified as the value of default in the column dictionary.

在版本1.1.6中更改：DDLEvents.column\_reflect() 事件允许在列字典中指定非字符串FetchedValue，sql.expression.text() 或派生对象作为默认值。

attrs - dict containing optional column attributes

attrs-dict包含可选列属性

The event is called before any action is taken against this dictionary, and the contents can be modified. The [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column) specific arguments info, key, and quote can also be added to the dictionary and will be passed to the constructor of [Column](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Column" \o "sqlalchemy.schema.Column).

在针对此字典采取任何行动之前，会调用该事件，并且可以修改内容。 列的具体参数info，key和quote也可以添加到字典中，并将其传递给Column的构造函数。

Note that this event is only meaningful if either associated with the [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) class across the board, e.g.:

请注意，此事件仅在与Table类相关联的情况下才有意义，例如：

**from** **sqlalchemy.schema** **import** Table

**from** **sqlalchemy** **import** event

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

"receive a column\_reflect event"

*# ...*

event.listen(

Table,

'column\_reflect',

listen\_for\_reflect)

…or with a specific [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table) instance using the listeners argument:

...或使用listeners参数使用特定的Table实例：

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

"receive a column\_reflect event"

*# ...*

t = Table(

'sometable',

autoload=**True**,

listeners=[

('column\_reflect', listen\_for\_reflect)

])

This because the reflection process initiated by autoload=True completes within the scope of the constructor for [Table](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.Table" \o "sqlalchemy.schema.Table).

这是因为由autoload = True启动的反射过程在表的构造函数的范围内完成。

*class*sqlalchemy.events.**SchemaEventTarget**

Base class for elements that are the targets of [DDLEvents](http://docs.sqlalchemy.org/en/rel_1_1/core/events.html" \l "sqlalchemy.events.DDLEvents" \o "sqlalchemy.events.DDLEvents) events.

This includes [SchemaItem](http://docs.sqlalchemy.org/en/rel_1_1/core/metadata.html" \l "sqlalchemy.schema.SchemaItem" \o "sqlalchemy.schema.SchemaItem) as well as [SchemaType](http://docs.sqlalchemy.org/en/rel_1_1/core/type_basics.html" \l "sqlalchemy.types.SchemaType" \o "sqlalchemy.types.SchemaType).

作为DDLEvents事件目标的元素的基类。

这包括SchemaItem以及SchemaType。