Бесплатная электронная книга

учусь sqlalchemy

Free unaffiliated eBook created from
Stack Overflow contributors.

#sqlalchem

y

Около

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: sqlalchemy

It is an unofficial and free sqlalchemy ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official sqlalchemy.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

глава 1: Начало работы с sqlalchemy

замечания

Философия SQLALCHEMY

С сайта SQLAlchemy :

> Базы данных SQL ведут себя не так, как коллекции объектов, тем больше размер и производительность начинают иметь значение; коллекции объектов ведут себя не так, как таблицы и строки, тем больше абстракция начинает иметь значение. SQLAlchemy стремится учесть оба этих принципа.
>
> SQLAlchemy рассматривает базу данных как механизм реляционной алгебры, а не только набор таблиц. Строки могут быть выбраны не только из таблиц, но и для соединений и других операторов выбора; любой из этих блоков может быть составлен в более крупную структуру. Язык выражения SQLAlchemy основывается на этой концепции из ее ядра.
>
> SQLAlchemy наиболее известен своим объектно-реляционным картографом (ORM), необязательным компонентом, который предоставляет шаблон отображения данных, где классы могут быть сопоставлены с базой данных открытым и несколькими способами - позволяя модели объектов и схемы базы данных разрабатываться в чисто развязанный путь с самого начала.
>
> Общий подход SQLAlchemy к этим проблемам полностью отличается от подхода большинства других инструментов SQL / ORM, внедренных в так называемый подход, ориентированный на совместимость; вместо того, чтобы скрывать детали репликации SQL и объектов за стеной автоматизации, все процессы полностью раскрываются в серии композиционных, прозрачных инструментов. Библиотека берет на себя задачу автоматизации избыточных задач, в то время как разработчик по-прежнему контролирует, как организована база данных и как построен SQL.
>
> Основная цель SQLAlchemy - изменить способ, который вы думаете о базах данных и SQL!

Версии

Версия	Статус выпуска	Журнал изменений	Дата выхода
1,1	Бета	1,1	2016-07-26

Версия	Статус выпуска	Журнал изменений	Дата выхода
1,0	Текущий выпуск	1,0	2015-04-16
0.9	техническое обслуживание	0.9	2013-12-30
0.8	Безопасность	0.8	2013-03-09

Examples

Установка или настройка

```
pip install sqlalchemy
```

Для большинства распространенных приложений, особенно веб-приложений, обычно рекомендуется, чтобы новички рассматривали использование дополнительной библиотеки, такой как `flask-sqlalchemy` .

```
pip install flask-sqlalchemy
```

Привет, мир! (SQLAlchemy Core)

В этом примере показано, как создать таблицу, вставить данные и выбрать из базы данных с помощью SQLAlchemy Core. Для получения информации re: ORM SQLAlchemy, см. Здесь .

Во-первых, нам нужно подключиться к нашей базе данных.

```
from sqlalchemy import create_engine

engine = create_engine('sqlite://')
```

> Двигатель является отправной точкой для любого приложения SQLAlchemy. Это «базовая база» для фактической базы данных и ее DBAPI, поставляемая в приложение SQLAlchemy через пул соединений и диалект, в котором описывается, как разговаривать с конкретным видом комбинации базы данных / DBAPI. Двигатель ссылается как на диалект, так и на пул соединений, которые вместе интерпретируют функции модуля DBAPI, а также поведение базы данных.

После создания нашего движка нам нужно определить и создать наши таблицы .

```
from sqlalchemy import Column, Integer, Text, MetaData, Table

metadata = MetaData()
messages = Table(
    'messages', metadata,
```

```
    Column('id', Integer, primary_key=True),
    Column('message', Text),
)

messages.create(bind=engine)
```

Чтобы подробнее объяснить объект MetaData, см. Ниже в документах:

> Коллекция объектов таблицы и связанных с ними дочерних объектов называется метаданными базы данных
>
> Мы определяем наши таблицы в каталоге MetaData, используя конструкцию Table, которая похожа на обычные инструкции SQL CREATE TABLE.

Теперь, когда мы определили и создали наши таблицы, мы можем начать вставлять данные! Вставка включает два шага. Составление конструкции вставки и выполнение окончательного запроса.

```
insert_message = messages.insert().values(message='Hello, World!')
engine.execute(insert_message)
```

Теперь, когда у нас есть данные, мы можем использовать функцию select для запроса наших данных. Объекты столбцов доступны как именованные атрибуты атрибута c в объекте Table, что упрощает выбор столбцов напрямую. Выполнение этого оператора select возвращает объект `ResultProxy` который имеет доступ к нескольким методам, fetchone () , fetchall () и fetchmany () , все из которых возвращают количество строк базы данных, запрошенных в нашем операторе select.

```
from sqlalchemy import select
stmt = select([messages.c.message])
message, = engine.execute(stmt).fetchone()
print(message)
```

```
Hello, World!
```

И это все! См. Учебное пособие SQLAlchemy SQL Expressions для получения дополнительных примеров и информации.

Привет, мир! (ORM SQLAlchemy)

В этом примере показано, как создать таблицу, вставить данные и выбрать из базы данных с помощью **ORM SQLAlchemy** . Для получения информации re: SQLAlchemy Core, см. Здесь .

Во-первых, нам нужно подключиться к нашей базе данных, которая идентична тому, как мы будем подключаться с помощью SQLAlchemy Core (Core).

```
from sqlalchemy import create_engine

engine = create_engine('sqlite://')
```

После подключения и создания нашего движка нам необходимо определить и создать наши таблицы. Именно здесь язык SQLAlchemy ORM начинает сильно отличаться от Core. В ORM процесс создания и определения таблицы начинается с определения таблиц и классов, которые мы будем использовать для сопоставления с этими таблицами. Этот процесс выполняется за один шаг в ORM, который SQLAlchemy вызывает декларативную систему.

```
from sqlalchemy.ext.declarative import declarative_base
Base = declarative_base()
```

Теперь, когда объявлен базовый mapper, мы можем подклассировать его для построения наших декларативных сопоставлений или `models` .

```
from sqlalchemy import Column, Integer, String

class Message(Base):
    __tablename__ = 'messages'

    id = Column(Integer, primary_key=True)
    message = Column(String)
```

Используя декларативный базовый класс, мы создаем объект `Table` и `Mapper` . Из документов:

> Объект Table является членом более крупной коллекции, известной как MetaData. При использовании Declarative этот объект доступен с использованием атрибута .metadata нашего декларативного базового класса.

Имея это в виду, чтобы создать все таблицы, которые еще не существуют, мы можем вызвать команду ниже, которая использует реестр метаданных SQLAlchemy Core.

```
Base.metadata.create_all(engine)
```

Теперь, когда наши таблицы отображаются и создаются, мы можем вставлять данные! Вставка выполняется с помощью создания экземпляров картпера .

```
message = Message(message="Hello World!")
message.message # 'Hello World!
```

На этом этапе все, что у нас есть, является экземпляром сообщения на уровне уровня абстракции ORM, но пока ничего не было сохранено в базе данных. Для этого сначала нужно создать сеанс .

```
from sqlalchemy.orm import sessionmaker

Session = sessionmaker(bind=engine)
session = Session()
```

Этот объект сеанса является нашим обработчиком базы данных. Согласно документам SQLAlchemy:

> он извлекает соединение из пула соединений, поддерживаемых движком, и удерживает его, пока мы не зафиксируем все изменения и / или не закрываем объект сеанса.

Теперь, когда у нас есть наш сеанс, мы можем добавить наше новое сообщение на сеанс и зафиксировать наши изменения в базе данных.

```
session.add(message)
session.commit()
```

Теперь, когда у нас есть данные, мы можем использовать язык запросов ORM, чтобы вытащить наши данные.

```
query = session.query(Message)
instance = query.first()
print (instance.message) # Hello World!
```

Но это только начало! Есть гораздо больше функций, которые можно использовать для составления запросов, таких как `filter` , `order_by` и многое другое. См. Учебное пособие по SQLAlchemy ORM для получения дополнительных примеров и информации.

Прочитайте Начало работы с sqlalchemy онлайн:
https://riptutorial.com/ru/sqlalchemy/topic/1697/начало-работы-с-sqlalchemy

глава 2: ORM

замечания

ORM SQLAlchemy построен поверх ядра SQLAlchemy . Например, хотя классы модели используют объекты `Column` , они являются частью ядра, и там будет найдена соответствующая документация.

Основными частями ORM являются классы сеанса , запроса и сопоставления (обычно с использованием декларативного расширения в современной SQLAlchemy.)

Examples

Преобразование результата запроса в dict

Сначала выполните настройку для примера:

```
import datetime as dt
from sqlalchemy import Column, Date, Integer, Text, create_engine, inspect
from sqlalchemy.orm import sessionmaker
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()
Session = sessionmaker()

class User(Base):
    __tablename__ = 'users'

    id = Column(Integer, primary_key=True)
    name = Column(Text, nullable=False)
    birthday = Column(Date)

engine = create_engine('sqlite://')
Base.metadata.create_all(bind=engine)
Session.configure(bind=engine)

session = Session()
session.add(User(name='Alice', birthday=dt.date(1990, 1, 1)))
session.commit()
```

Если вы запрашиваете столбцы отдельно, строка является `KeyedTuple` которая имеет метод `_asdict` . Имя метода начинается с одного подчеркивания, чтобы соответствовать API-интерфейсу `namedtuple` (он не является приватным!).

```
query = session.query(User.name, User.birthday)
for row in query:
    print(row._asdict())
```

При использовании ORM для извлечения объектов это по умолчанию недоступно. Следует

использовать систему инспекции SQLAlchemy.

```
def object_as_dict(obj):
    return {c.key: getattr(obj, c.key)
            for c in inspect(obj).mapper.column_attrs}

query = session.query(User)
for user in query:
    print(object_as_dict(user))
```

Здесь мы создали функцию для преобразования, но одним из вариантов было бы добавить метод к базовому классу.

Вместо использования `declarative_base` как указано выше, вы можете создать его из своего собственного класса:

```
from sqlalchemy.ext.declarative import as_declarative

@as_declarative()
class Base:
    def _asdict(self):
        return {c.key: getattr(self, c.key)
                for c in inspect(self).mapper.column_attrs}
```

фильтрация

Учитывая следующую модель

```
class User(Base):
    __tablename__ = 'users'

    id = Column(Integer, primary_key=True)
    name = Column(Text, nullable=False)
    birthday = Column(Date)
```

Вы можете фильтровать столбцы в запросе:

```
import datetime as dt
session.query(User).filter(User.name == 'Bob')
session.query(User).filter(User.birthday < dt.date(2000, 1, 1))
```

В первом случае есть ярлык:

```
session.query(User).filter_by(name='Bob')
```

Фильтры могут быть скомпонованы с использованием отношения AND путем связывания метода `filter` :

```
(session.query(User).filter(User.name.like('B%'))
                    .filter(User.birthday < dt.date(2000, 1, 1)))
```

Или более гибко, используя перегруженные побитовые операторы `&` и `|` :

```
session.query(User).filter((User.name == 'Bob') | (User.name == 'George'))
```

Не забывайте, что внутренние круглые скобки имеют дело с приоритетом оператора.

Сортировать по

Учитывая базовую модель:

```
class SpreadsheetCells(Base):
    __tablename__ = 'spreadsheet_cells'

    id = Column(Integer, primary_key=True)
    y_index = Column(Integer)
    x_index = Column(Integer)
```

Вы можете получить упорядоченный список, `order_by` метод `order_by` .

```
query = session.query(SpreadsheetCells).order_by(SpreadsheetCells.y_index)
```

Это может быть закодировано после `filter` ,

```
query = session.query(...).filter(...).order_by(...)
```

или для дальнейшего составления существующего запроса.

```
query = session.query(...).filter(...)
ordered_query = query.order_by(...)
```

Вы также можете определить направление сортировки одним из двух способов:

1. Доступ к свойствам поля `asc` и `dsc` :

```
query.order_by(SpreadsheetCells.y_index.desc()) # desc
query.order_by(SpreadsheetCells.y_index.asc()) # asc
```

2. Использование функций asc и desc:

```
from sqlalchemy import asc, desc

query.order_by(desc(SpreadsheetCells.y_index)) # desc
query.order_by(asc(SpreadsheetCells.y_index)) # asc
```

Доступ к результатам запроса

Когда у вас есть запрос, вы можете сделать больше с ним, чем просто повторять результаты в цикле for.

Настроить:

```
from datetime import date

class User(Base):
    __tablename__ = 'users'

    id = Column(Integer, primary_key=True)
    name = Column(Text, nullable=False)
    birthday = Column(Date)

# Find users who are older than a cutoff.
query = session.query(User).filter(User.birthday < date(1995, 3, 3))
```

Чтобы вернуть результаты в виде списка, используйте `all()` :

```
reslist = query.all() # all results loaded in memory
nrows = len(reslist)
```

Вы можете получить счет, используя `count()` :

```
nrows = query.count()
```

Чтобы получить только первый результат, используйте `first()` . Это наиболее полезно в сочетании с `order_by()` .

```
oldest_user = query.order_by(User.birthday).first()
```

Для запросов, которые должны возвращать только одну строку, используйте `one()` :

```
bob = session.query(User).filter(User.name == 'Bob').one()
```

Это вызывает исключение, если запрос возвращает несколько строк или возвращает его. Если строка может еще не существовать, используйте `one_or_none()` :

```
bob = session.query(User).filter(User.name == 'Bob').one_or_none()
if bob is None:
    create_bob()
```

Это все равно вызовет исключение, если несколько строк имеют имя «Боб».

Прочитайте ORM онлайн: https://riptutorial.com/ru/sqlalchemy/topic/2020/orm

глава 3: Колба-SQLAlchemy

замечания

Flask-SQLAlchemy добавляет некоторые дополнительные функции, такие как автоматическое разрушение сеанса, предполагая некоторые вещи для вас, которые очень часто не то, что вам нужно.

Examples

Минимальное приложение

Для общего случая использования одного приложения Flask вам нужно создать приложение Flask, загрузить выбранную конфигурацию и затем создать объект SQLAlchemy, передав ему приложение.

После создания этот объект затем содержит все функции и помощники как от sqlalchemy, так и от sqlalchemy.orm. Кроме того, он предоставляет класс под названием «Модель», который является декларативной базой, которая может использоваться для объявления моделей:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:////tmp/test.db'
db = SQLAlchemy(app)


class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True)
    email = db.Column(db.String(120), unique=True)

    def __init__(self, username, email):
        self.username = username
        self.email = email

    def __repr__(self):
        return '<User %r>' % self.username
```

Прочитайте Колба-SQLAlchemy онлайн: https://riptutorial.com/ru/sqlalchemy/topic/4601/колба-sqlalchemy

глава 4: Сессия

замечания

Сеанс отслеживает объекты ORM и их изменения, управляет транзакциями и используется для выполнения запросов .

Examples

Создание сеанса

Сессию обычно получают с помощью `sessionmaker` , который создает класс `Session` уникальный для вашего приложения. Чаще всего класс `Session` привязан к движку, что позволяет экземплярам использовать движок неявно.

```
from sqlalchemy.orm import sessionmaker

# Initial configuration arguments
Session = sessionmaker(bind=engine)
```

`engine` и `Session` должны создаваться только один раз.

Сеанс - это экземпляр класса, который мы создали:

```
# This session is bound to provided engine
session = Session()
```

`Session.configure()` можно использовать для настройки класса позже, например, запуска приложения, а не времени импорта.

```
Session = sessionmaker()

# later
Session.configure(bind=engine)
```

Аргументы, переданные `Session` напрямую переопределяют аргументы, переданные `sessionmaker` .

```
session_bound_to_engine2 = Session(bind=engine2)
```

Добавление экземпляров

Новые или отдельные объекты могут быть добавлены в сеанс, используя `add()` :

```
session.add(obj)
```

Последовательность объектов может быть добавлена с помощью `add_all()` :

```
session.add_all([obj1, obj2, obj3])
```

INSERT будет отправлен в базу данных во время следующего сброса, что произойдет автоматически. Изменения сохраняются при завершении сеанса.

Удаление экземпляров

Для удаления сохраненных объектов используйте `delete()` :

```
session.delete(obj)
```

Фактическое удаление из базы данных произойдет на следующем флеше .

Прочитайте Сессия онлайн: https://riptutorial.com/ru/sqlalchemy/topic/2258/сессия

глава 5: соединительный

Examples

двигатель

Механизм используется для подключения к различным базам данных с использованием URL-адреса подключения:

```
from sqlalchemy import create_engine

engine = create_engine('postgresql://user:pass@localhost/test')
```

Обратите внимание, однако, что двигатель фактически не устанавливает соединение до его первого использования.

Механизм автоматически создает пул соединений, но лениво открывает новые соединения (т. Е. SQLAlchemy не откроет 5 соединений, если вы только попросите их).

Использование соединения

Вы можете открыть соединение (т. Е. Запросить один из пула) с помощью диспетчера контекстов:

```
with engine.connect() as conn:
    result = conn.execute('SELECT price FROM products')
    for row in result:
        print('Price:', row['price'])
```

Или без, но он должен быть закрыт вручную:

```
conn = engine.connect()
result = conn.execute('SELECT price FROM products')
for row in result:
    print('Price:', row['price'])
conn.close()
```

Неявное выполнение

Если вы хотите выполнить только один оператор, вы можете напрямую использовать движок, и он откроет и закроет соединение для вас:

```
result = engine.execute('SELECT price FROM products')
for row in result:
    print('Price:', row['price'])
```

операции

Вы можете использовать `engine.begin` чтобы открыть соединение и начать транзакцию, которая будет `engine.begin` если возникнет исключение или будет выполнено иначе. Это неявный способ использования транзакции, поскольку у вас нет возможности отката вручную.

```
with engine.begin() as conn:
    conn.execute(products.insert(), price=15)
```

Более конкретно, вы можете начать транзакцию с использованием соединения:

```
with conn.begin() as trans:
    conn.execute(products.insert(), price=15)
```

Обратите внимание, что мы по-прежнему вызываем `execute` в соединении. Как и ранее, эта транзакция будет зафиксирована или отменена, если возникнет исключение, но у нас также есть доступ к транзакции, что позволяет нам `trans.rollback()` вручную, используя `trans.rollback()` .

Это можно сделать более явно:

```
trans = conn.begin()
try:
    conn.execute(products.insert(), price=15)
    trans.commit()
except:
    trans.rollback()
    raise
```

Прочитайте соединительный онлайн: https://riptutorial.com/ru/sqlalchemy/topic/2025/соединительный

глава 6: Ядро SQLAlchemy

Examples

Преобразование результата в dict

В ядре SQLAlchemy результатом является `RowProxy` . В случаях, когда требуется явный словарь, вы можете вызвать `dict(row)` .

Сначала выполните настройку для примера:

```
import datetime as dt
from sqlalchemy import (
    Column, Date, Integer, MetaData, Table, Text, create_engine, select)

metadata = MetaData()
users = Table(
    'users', metadata,
    Column('id', Integer, primary_key=True),
    Column('name', Text, nullable=False),
    Column('birthday', Date),
)

engine = create_engine('sqlite://')
metadata.create_all(bind=engine)

engine.execute(users.insert(), name='Alice', birthday=dt.date(1990, 1, 1))
```

Затем, чтобы создать словарь из строки результата:

```
with engine.connect() as conn:
    result = conn.execute(users.select())
    for row in result:
        print(dict(row))

    result = conn.execute(select([users.c.name, users.c.birthday]))
    for row in result:
        print(dict(row))
```

Прочитайте Ядро SQLAlchemy онлайн: https://riptutorial.com/ru/sqlalchemy/topic/2022/ядро-sqlalchemy

кредиты

S. No	Главы	Contributors
1	Начало работы с sqlalchemy	adarsh, Community, Mattew Whitt, RazerM, Stephen Fuhry
2	ORM	Ilja Everilä, Mattew Whitt, RazerM, Tom Hunt
3	Колба-SQLAlchemy	Ilya Rusin
4	Сессия	Ilja Everilä, RazerM
5	соединительный	RazerM
6	Ядро SQLAlchemy	RazerM