Швидкий старт ORM

Для нових користувачів, які хочуть швидко побачити, як виглядає базове використання ORM, ось скорочена форма зіставлень і прикладів, використаних у посібнику SQLAlchemy 1.4 / 2.0 . Код тут можна повністю запускати з чистого командного рядка.

Оскільки описи в цьому розділі навмисно **дуже короткі** , будь ласка, перейдіть до повного навчального посібника SQLAlchemy 1.4 / 2.0 , щоб отримати набагато більш глибокий опис кожної з концепцій, які тут проілюстровані.

Оголошення моделей

Тут ми визначаємо конструкції рівня модуля, які будуть формувати структури, які ми будемо запитувати з бази даних. Ця структура, відома як Декларативне відображення , визначає одночасно об'єктну модель Python, а також метадані бази даних , які описують реальні таблиці SQL, які існують або будуть існувати в конкретній базі даних:

```
>>> from sqlalchemy import Column
>>> from sqlalchemy import ForeignKey
>>> from sqlalchemy import Integer
>>> from sqlalchemy import String
>>> from sqlalchemy.orm import declarative_base
>>> from sqlalchemy.orm import relationship

>>> Base = declarative_base()

>>> class User(Base):
...     __tablename__ = "user_account"
...
...     id = Column(Integer, primary_key=True)
...     name = Column(String(30))
...     fullname = Column(String)
...
...     addresses = relationship(
...         "Address", back_populates="user", cascade="all, delete-or
...     )
...
...     def __repr__(self):
...         return f"User(id={self.id!r}, name={self.name!r}, fullnam

>>> class Address(Base):
...     __tablename__ = "address"
...
...     id = Column(Integer, primary_key=True)
...     email_address = Column(String, nullable=False)
...     user_id = Column(Integer, ForeignKey("user_account.id"), null
...
...     user = relationship("User", back_populates="addresses")
...
...     def __repr__(self):
...         return f"Address(id={self.id!r}, email_address={self.emai
```

Вище декларативне відображення використовує `Column` об'єкти для визначення основних одиниць зберігання даних, які будуть у базі даних. Конструкція `relationship()` визначає зв'язки між двома відображеними класами `User` та `Address` вище.

Схема містить такі необхідні елементи, як обмеження первинного ключа, встановлені `Column.primary_key` параметром, обмеження зовнішнього ключа, налаштованого за допомогою

Документація SQLAlchemy 1.4

ПОТОЧНИЙ ВИПУСК

Головна | Завантажте цю документацію

Пошукові терміни: search...

`ForeignKey` (яке також використовується `relationship()`), і типи даних для стовпців, включаючи `Integer` та `String`.

Більше про метадані таблиці та ознайомлення з оголошеним зіставленням ORM див. у посібнику Робота з метаданими бази даних .

Створення двигуна

Це `Engine` фабрика , яка може створювати для нас нові підключення до бази даних, яка також утримує підключення всередині пулу підключень для швидкого повторного використання. З метою навчання ми зазвичай використовуємо базу даних SQLite , яка займає лише пам'ять:

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine("sqlite://", echo=True, future=True)
```

Порада

Параметр `echo=True` вказує на те, що SQL, створений підключеннями, буде зареєстровано в стандартний вихід. `future=True` полягає в тому, щоб переконатися, що ми використовуємо останні API у стилі SQLAlchemy 2.0 .

Повний вступ до початку `Engine` починається з Встановлення зв'язку – механізм .

Видавати CREATE TABLE DDL

Використовуючи метадані нашої таблиці та наш механізм, ми можемо згенерувати нашу схему відразу в нашій цільовій базі даних SQLite, використовуючи метод під назвою `MetaData.create_all()`:

```
>>> Base.metadata.create_all(engine)
BEGIN (implicit)
PRAGMA main.table_...info("user_account")
...
PRAGMA main.table_...info("address")
...
CREATE TABLE user_account (
    id INTEGER NOT NULL,
    name VARCHAR(30),
    fullname VARCHAR,
    PRIMARY KEY (id)
)
...
CREATE TABLE address (
    id INTEGER NOT NULL,
    email_address VARCHAR NOT NULL,
    user_id INTEGER NOT NULL,
    PRIMARY KEY (id),
    FOREIGN KEY(user_id) REFERENCES user_account (id)
)
...
COMMIT
```

Багато що сталося з того шматка коду Python, який ми написали. Щоб отримати повний огляд того, що відбувається з метаданими таблиці, перейдіть до підручника щодо роботи з метаданими бази даних .

Створення об'єктів і збереження

Тепер ми готові вставити дані в базу даних. Ми досягаємо цього, створюючи екземпляри `User` та `Address` об'єкти, які вже мають `__init__()` метод, автоматично встановлений процесом декларативного відображення. Потім ми передаємо їх до бази даних за допомогою об'єкта під назвою Session , який використовує `Engine` для взаємодії з базою даних. Метод `Session.add_all()` використовується тут для додавання кількох об'єктів одночасно, і `Session.commit()` метод буде використано для очищення будь-яких очікуваних змін у базі даних, а потім фіксації поточної транзакції бази даних, яка завжди виконується, коли `Session` використовується:

```
>>> from sqlalchemy.orm import Session

>>> with Session(engine) as session:
...
...     spongebob = User(
...         name="spongebob",
...         fullname="Spongebob Squarepants",
...         addresses=[Address(email_address="spongebob@sqlalchemy.or
...     )
...     sandy = User(
...         name="sandy",
...         fullname="Sandy Cheeks",
...         addresses=[
...             Address(email_address="sandy@sqlalchemy.org"),
...             Address(email_address="sandy@squirrelpower.org"),
...         ],
...     )
...     patrick = User(name="patrick", fullname="Patrick Star")
...
...     session.add_all([spongebob, sandy, patrick])
...
...     session.commit()
```

```
BEGIN (implicit)
INSERT INTO user_account (name, fullname) VALUES (?, ?)
[...] ('spongebob', 'Spongebob Squarepants')
INSERT INTO user_account (name, fullname) VALUES (?, ?)
[...] ('sandy', 'Sandy Cheeks')
INSERT INTO user_account (name, fullname) VALUES (?, ?)
[...] ('patrick', 'Patrick Star')
INSERT INTO address (email_address, user_id) VALUES (?, ?)
[...] ('spongebob@sqlalchemy.org', 1)
INSERT INTO address (email_address, user_id) VALUES (?, ?)
[...] ('sandy@sqlalchemy.org', 2)
INSERT INTO address (email_address, user_id) VALUES (?, ?)
[...] ('sandy@squirrelpower.org', 2)
COMMIT
```

Порада

Рекомендовано `Session` використовувати у стилі диспетчера контексту, як зазначено вище, тобто за допомогою `with:` оператора Python. Об’єкт `Session` представляє активні ресурси бази даних, тому добре переконатися, що він закритий після завершення ряду операцій. У наступному розділі ми залишимо `Session` відкритим лише для ілюстрації.

Основи створення `Session` можна знайти в розділі Виконання за допомогою сеансу ORM, а більше – у розділі Основи використання сеансу .

Далі в розділі Вставлення рядків за допомогою ORM представлені деякі різновиди базових операцій збереження .

Простий SELECT

З деякими рядками в базі даних, ось найпростіша форма випуску оператора SELECT для завантаження деяких об’єктів. Щоб створити оператори SELECT, ми використовуємо `select()` функцію для створення нового `Select` об’єкта, який потім викликаємо за допомогою `Session`. Метод, який часто буває корисним під час запиту об’єктів ORM, це `Session.scalars()` метод, який повертає `ScalarResult` об’єкт, який буде проходити через об’єкти ORM, які ми вибрали:

```
>>> from sqlalchemy import select

>>> session = Session(engine)

>>> stmt = select(User).where(User.name.in_(["spongebob", "sandy"]))

>>> for user in session.scalars(stmt):
...     print(user)
```

```
BEGIN (implicit)
SELECT user_account.id, user_account.name, user_account.fullname
FROM user_account
WHERE user_account.name IN (?, ?)
[...] ('spongebob', 'sandy')
```

```
User(id=1, name='spongebob', fullname='Spongebob Squarepants')
User(id=2, name='sandy', fullname='Sandy Cheeks')
```

Наведений вище запит також використовував `Select.where()` метод для додавання критеріїв WHERE, а також використовував `ColumnOperators.in_()` метод, який є частиною всіх конструкцій, подібних до стовпців SQLAlchemy, для використання оператора SQL IN.

Детальніше про те, як вибирати об’єкти та окремі стовпці, див. у розділі Вибір сутностей і стовпців ORM .

SELECT за допомогою JOIN

Дуже поширеним є запит між кількома таблицями одночасно, і в SQL ключове слово JOIN є основним способом, яким це відбувається. Конструкція `Select` створює об’єднання за допомогою `Select.join()` методу:

```
>>> stmt = (
...  select(Address)
...  .join(Address.user)
...  .where(User.name == "sandy")
...  .where(Address.email_address == "sandy@sqlalchemy.org")
... )
>>> sandy_address = session.scalars(stmt).one()
```

```
SELECT address.id, address.email_address, address.user_id
FROM address JOIN user_account ON user_account.id = address.user_id
WHERE user_account.name = ? AND address.email_address = ?
[...] ('sandy', 'sandy@sqlalchemy.org')
```

```
>>> sandy_address
Address(id=2, email_address='sandy@sqlalchemy.org')
```

Наведений вище запит ілюструє кілька критеріїв WHERE, які автоматично об’єднуються разом за допомогою оператора AND, а також те, як використовувати об’єкти, подібні до стовпців SQLAlchemy, для створення порівнянь «рівності», які використовують перевизначений метод Python `ColumnOperators.__eq__()` для створення об’єкта критеріїв SQL.

Докладніше про наведені вище концепції можна знайти в реченні WHERE та явних реченнях FROM і JOIN .

Внести зміни

Об’єкт `Session` у поєднанні з нашими ORM-відображеними класами `User` та `Address`, автоматично відстежує зміни в об’єктах у міру їх внесення, що призводить до операторів SQL, які будуть видані під час наступного `Session` очищення. Нижче ми змінюємо одну адресу електронної пошти, пов’язану з «sandy», а також додаємо нову адресу електронної пошти до «patrick» після виведення SELECT для отримання рядка для «patrick»:

```
>>> stmt = select(User).where(User.name == "patrick")
>>> patrick = session.scalars(stmt).one()
```

```
SELECT user_account.id, user_account.name, user_account.fullname
FROM user_account
WHERE user_account.name = ?
[...] ('patrick',)
```

```
>>> patrick.addresses.append(
...     Address(email_address="patrickstar@sqlalchemy.org")
... )
```

```
SELECT address.id AS address_id, address.email_address AS address_ema
FROM address
```

```
WHERE ? = address.user_id
[...] (3,)
```

```
>>> sandy_address.email_address = "sandy_cheeks@sqlalchemy.org"

>>> session.commit()
```

```
UPDATE address SET email_address=? WHERE address.id = ?
[...] ('sandy_cheeks@sqlalchemy.org', 2)
INSERT INTO address (email_address, user_id) VALUES (?, ?)
[...] ('patrickstar@sqlalchemy.org', 3)
COMMIT
```

Зауважте, коли ми звернулися до `patrick.addresses`, було видано SELECT. Це називається відкладеним завантаженням . Основні відомості про різні способи доступу до пов'язаних елементів за допомогою більшої чи меншої кількості SQL представлені на Loader Strategies .

Детальний опис маніпулювання даними ORM починається з Маніпулювання даними за допомогою ORM .

Деякі видалення

Усьому має прийти кінець, як у випадку з деякими рядками нашої бази даних. Ось коротка демонстрація двох різних форм видалення, обидві з яких важливі в залежності від конкретного випадку використання.

Спочатку ми видалимо один із `Address` об'єктів у «пісочного» користувача. Під час `Session` наступного очищення це призведе до видалення рядка. Цю поведінку ми налаштували в нашому відображенні під назвою каскад видалення . Ми можемо отримати дескриптор `sandy` об'єкта за допомогою первинного ключа `Session.get()`, а потім працювати з об'єктом:

```
>>> sandy = session.get(User, 2)
```

```
BEGIN (implicit)
SELECT user_account.id AS user_account_id, user_account.name AS user_
FROM user_account
WHERE user_account.id = ?
[...] (2,)
```

```
>>> sandy.addresses.remove(sandy_address)
```

```
SELECT address.id AS address_id, address.email_address AS address_em
FROM address
WHERE ? = address.user_id
[...] (2,)
```

Останній SELECT вище був операцією відкладеного завантаження `sandy.addresses` , щоб можна було завантажити колекцію, щоб ми могли видалити `sandy_address` член. Існують інші способи виконання цієї серії операцій, які не видаватимуть стільки SQL.

Ми можемо вибрати видалення SQL DELETE для того, що налаштовано на зміну, без фіксації транзакції, використовуючи `Session.flush()` метод:

```
>>> session.flush()
```

```
DELETE FROM address WHERE address.id = ?
[...] (2,)
```

Далі ми повністю видалимо користувача «patrick». Для видалення верхнього рівня самого об'єкта ми використовуємо `Session.delete()` метод; цей метод фактично не виконує видалення, але встановлює об'єкт, який буде видалено під час наступного очищення. Операція також каскадується до пов'язаних об'єктів на основі параметрів каскаду, які ми налаштували, у цьому випадку, для пов'язаних `Address` об'єктів:

```
>>> session.delete(patrick)
SELECT user_account.id AS user_account_id, user_account.name AS user_
FROM user_account
WHERE user_account.id = ?
[...] (3,)
SELECT address.id AS address_id, address.email_address AS address_ema
FROM address
WHERE ? = address.user_id
[...] (3,)
```

Метод `Session.delete()` у цьому конкретному випадку випромінював два оператори SELECT, навіть якщо він не випустив DELETE, що може здатися дивним. Це пов'язано з тим, що коли метод перейшов на перевірку об'єкта, виявилося, що термін дії `patrick` об'єкта закінчився , що сталося, коли ми востаннє викликали upon `Session.commit()`, і виданий SQL мав повторно завантажити рядки з нової транзакції. Цей термін дії є необов'язковим, і за нормального використання ми часто вимикаємо його в ситуаціях, коли він не застосовується належним чином.

Щоб проілюструвати рядки, які видаляються, ось фіксація:

```
>>> session.commit()
DELETE FROM address WHERE address.id = ?
[...] (4,)
DELETE FROM user_account WHERE user_account.id = ?
[...] (3,)
COMMIT
```

Підручник обговорює видалення ORM на сторінці Видалення об'єктів ORM . Фон закінчення терміну дії об'єкта знаходиться на Термін дії закінчується / Оновлюється ; каскади детально обговорюються на сайті Cascades .

Глибоко вивчіть наведені вище концепції

Для нового користувача наведені вище розділи, ймовірно, були бурхливим туром. На кожному кроці вище є багато важливих понять, які не були охоплені. Завдяки швидкому огляду того, як все виглядає, рекомендується ознайомитися з підручником SQLAlchemy 1.4 / 2.0 , щоб отримати міцні робочі знання про те, що насправді відбувається вище. Удачі!

Попередній: SQLAlchemy ORM Далі: Навчальний посібник із реляційних об'єктів (1.x API)