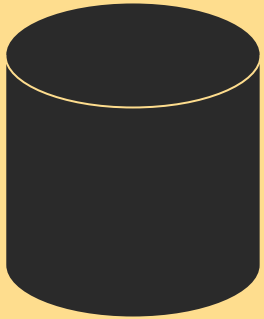


Novidades do SQLAlchemy 1.4+

Live de Python # 166

Roteiro

1. Criando um modelo
Um breve resumo

2. Conexão
Engine async, gerenciador de contexto e run_sync

3. Manipulando dados
AsyncSession e gerenciamento de contexto

4. Queries
A união entre o core e o ORM; Scalars

Maaaaas

Vamos falar exatamente sobre:

- 2.0 style
- Async ORM
- Novo estilo de queries
- Eventos

Já falamos sobre isso, não?

- Série de corrotinas (# 152, # 153, #154)
- Objetos assíncronos (#59)
- SQLAlchemy ORM (# 139)

picpay.me/dunossauro

apoia.se/livedepython

PIX

Ajude o projeto

Ademar Peixoto, Alex Lima, Alexandre Harano, Alexandre Santos, Alexandre Tsuno, Alexandre Villares, Alynne Ferreira, Alysson Oliveira, Amaziles Carvalho, Andre Rodrigues, André Rocha, Arnaldo Turque, Bruno Batista, Bruno Oliveira, Caio Nascimento, César Almeida, César Moreira, Davi Ramos, David Kwast, Diego Guimarães, Dilenon Delfino, Douglas Bastos, Elias Soares, Eugenio Mazzini, Everton Alves, Fabiano Gomes, Fabio Barros, Fabio Castro, Fabrício Coelho, Flavkaze Flavkaze, Franklin Silva, Fábio Serrão, Gabriel Simonetto, Gabriel Soares, Gabriela Santiago, Geandreson Costa, Guilherme Felitti, Guilherme Marson, Guilherme Ostrock, Gustavo Chacon, Henrique Machado, Hélio Neto, Israel Fabiano, Italo Silva, Johnny Tardin, Jonatas Leon, Jorge Plautz, José Prado, João Lugão, João Schiavon, Juan Gutierrez, Jônatas Silva, Júlia Kastrup, Kaneson Alves, Leonardo Cruz, Leonardo Galani, Leonardo Mello, Lidiane Monteiro, Lorena Ribeiro, Lucas Barros, Lucas Mello, Lucas Mendes, Lucas Teixeira, Lucas Valino, Luciano Ratamero, Marcelo Rodrigues, Maiquel Leonel, Maiquel Leonel, Marcela Campos, Maria Clara, Matheus Vian, Melissa Mendonça, Natan Cervinski, Nicolas Teodosio, Patric Lacouth, Patricia Minamizawa, Patrick Gomes, Paulo Tadei, Pedro Pereira, Peterson Santos, Rafael Lino, Reinaldo Silva, Revton Silva, Rodrigo Ferreira, Rodrigo Mende, Rodrigo Vaccari, Sandro Mio, Silvio Xm, Thiago Araujo, Thiago Borges, Thiago Bueno, Tyrone Damasceno, Victor Geraldo, Vinícius Bastos, Vinícius Ferreira, Vítor Gomes, Wesley Mendes, Willian Lopes, Willian Lopes, Willian Rosa, Wilson Duarte, Ronaldo Silva, Wendel Rios, Érico Andrei

Obrigado você

Aqui nada mudou,
mas sem modelo
não tem ORM

Modelo

Um modelo mais do que simples

Como minha ideia é exemplificar as novas features vamos trabalhar com o modelo mais simples possível.

```
1   from sqlalchemy.orm import declarative_base
2
3   Base = declarative_base()
4
5
6   class Pessoa(Base):
7     __tablename__ = 'pessoa'
8
9     id = Column(Integer, primary_key=True)
10    nome = Column(String)
11    email = Column(String)
12
13    def __repr__(self):
14        return f'Pessoa({self.nome})'
```

pip install sqlalchemy>=1.4

Instalando

pip install sqlalchemy>=1.4

pip install aiosqlite

Instalando

Con exão

Conectando assincronamente

Conexão

Agora a engine de conexão do banco pode ser feita de maneira assíncrona. O que precisamos fazer é importar o criador de engine da extensão do `asyncio`.

```python
from sqlalchemy.ext.asyncio import create_async_engine

url_sqlite3_async = 'sqlite+aiosqlite:///./db.db'

engine = create_async_engine(url_sqlite3_async)
```

Engine async

Com a engine async, precisamos usar um gerenciador de contexto async

```
1  async def create_database():
2    '''Deleta todas as tabelas do banco e cria todas novamente.'''
3    async with engine.begin() as conn:
4        await conn.run_sync(Base.metadata.drop_all)
5        await conn.run_sync(Base.metadata.create_all)
```

Jobs síncronos

Caso a operação necessite que o loop "trave" e não execute nada enquanto isso, podemos usar o `run_sync`

```
1  async def create_database():
2    '''Deleta todas as tabelas do banco e cria todas novamente.'''
3    async with engine.begin() as conn:
4        await conn.run_sync(Base.metadata.drop_all)
5        await conn.run_sync(Base.metadata.create_all)
```

Rodando

Agora as ações da conexão, por serem assíncronas, precisam ser executadas dentro de uma função assíncrona.

```
1   from asyncio import run
2
3   run(create_database())
```

Aqui temos novidades significativas

Sessão

Criando uma sessão

Agora, tanto nos casos assíncronos, quanto nos casos síncronos, a session deve ser sempre invocada em um gerenciador de contexto

```python
from sqlalchemy.orm import sessionmaker
from sqlalchemy.ext.asyncio import AsyncSession

session = sessionmaker(
  engine,
  expire_on_commit=False,
  future=True,  # Estilo 2.0 (Caso seja necessário)
  class_=AsyncSession  # caso contrário, síncrono
)
```

Criando uma sessão

Agora, tanto nos casos assíncronos, q
session deve ser sempre invocada em

- Session.begin desativado
- Cascade sempre false

```
1  from sqlalchemy.orm impor
2  from sqlalchemy.ext.asyn
3
4  session = sessionmak
5    engine,
6    expire_on_commi=False,
7    future=True,  # Estilo 2.0 (Caso seja necessário)
8    class_=AsyncSession  # caso contrário, síncrono
9  )
```

Criando uma sessão

Agora, tanto nos casos assíncronos, quanto nos casos síncronos, a session deve ser sempre invocada em

```
1  from sqlalchemy.orm impor
2  from sqlalchemy.ext.async
3
4  session = sessionmaker(
5    engine,
6    expire_on_commit=False,
7    future=True,  # Estilo 2.0 (Caso seja necessário)
8    class_=AsyncSession  # caso contrário, síncrono
9  )
```

Caso não seja invocado com a classe async, sempre será sync

Inserindo um registro

No estilo 2.0, todas as interações com a sessão devem estar em um gerenciador de contexto (mesmo que sejam síncronas)

```
1  async def criar_pessoa(nome, email):
2    async with session() as s:
3      s.add(Pessoa(nome=nome, email=email))
4
5      await s.commit()
```

Queries

Iguais, mais diferentes

O novo select

Agora as queries, no estilo 2.0, serão executadas pela session, mas serão "compiladas" pelo select.

```python
from sqlalchemy.future import select

async def buscar_pessoa(nome):
  async with session() as s:
      query = await s.execute(
          select(Pessoa).where(Pessoa.nome == nome)
      )
      # resturn query.scalars().all()
      return query.all()
```

Scalars

O resultado das queries, usando select, é retornado em Rows. Modificado o estilo anterior de retornar o objeto do ORM. Para que isso aconteça é necessário usar o método `scalar` para um único objeto, `scalars` para múltiplos objetos

```
5         query = await s.execute(
6             select(Pessoa).where(Pessoa.nome == nome)
7         )
8
9         return query.scalars().all()
```

Updates

Os updates podem ser feitos de duas formas, pelo `select` igual fazíamos anteriormente, ou pelo update, que não é uma feature nova. Porém, segue o estilo 2.0

```python
from sqlalchemy import update

async def atualizar_pessoa(old, new):
  async with session() as s:
      await s.execute(
          update(Pessoa).where(
            Pessoa.nome == old
          ).values(nome=new)
      )
      result = await s.commit()
      return result
```

Delete

Os deletes podem ser feitos de duas formas, pelo `select` igual fazíamos anteriormente, ou pelo delete, que não é uma feature nova. Porém, segue o estilo 2.0

```python
from sqlalchemy import delete


async def deletar_pessoa(nome):
  async with session() as s:
      query = await s.execute(
          delete(Pessoa).where(
            Pessoa.nome == nome
          )
      )
      await s.commit()
```

Como ficam os joins no novo formato?

Relacionamentos

Incrementando o modelo

```python
class Pessoa(Base):
    __tablename__ = 'pessoa'

    id = Column(Integer, primary_key=True)
    nome = Column(String)
    email = Column(String)

    posts = relationship('Post', backref='pessoa')

    def __repr__(self):
        return f'Pessoa({self.nome})'
```

```python
class Post(Base):
    __tablename__ = 'post'

    id = Column(Integer, primary_key=True)
    titulo = Column(String)
    conteudo = Column(String)

    autor_id = Column(Integer, ForeignKey('pessoa.id'))
    autor = relationship('Pessoa', backref='post')

    def __repr__(self):
        return f'Post({self.titulo}, {self.autor_id})'
```

Select + Join

O select nesse momento é feito com as duas classes, o que faz duplicação. O join junta os registros referentes as duas tabelas.

```python
async def buscar_post_por_nome_do_autor(nome):
    async with session() as s:
        query = await s.execute(
            select(Post, Pessoa).join(Post.autor).where(Pessoa.nome == nome)
        )
        result = query.scalars().all()
        return result
```

Eventos

:(

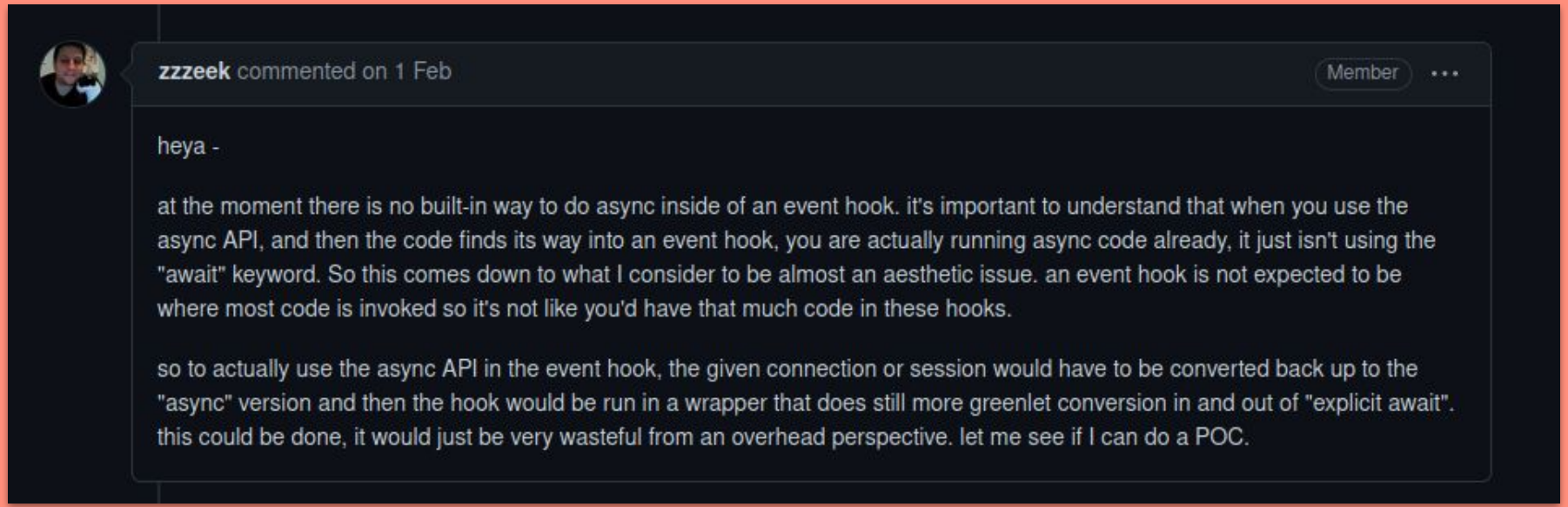
Eventos

Infelizmente os eventos não são suportados, pelo menos até o momento, na session async.

```python
from sqlalchemy import event


@event.listens_for(session.class_.sync_session, 'after_transaction_end')
@event.listens_for(session, 'after_transaction_end')
def _do_orm_execute(session, transaction):
  breakpoint()
```

Mais informações?



zzzeek commented on 1 Feb — Member

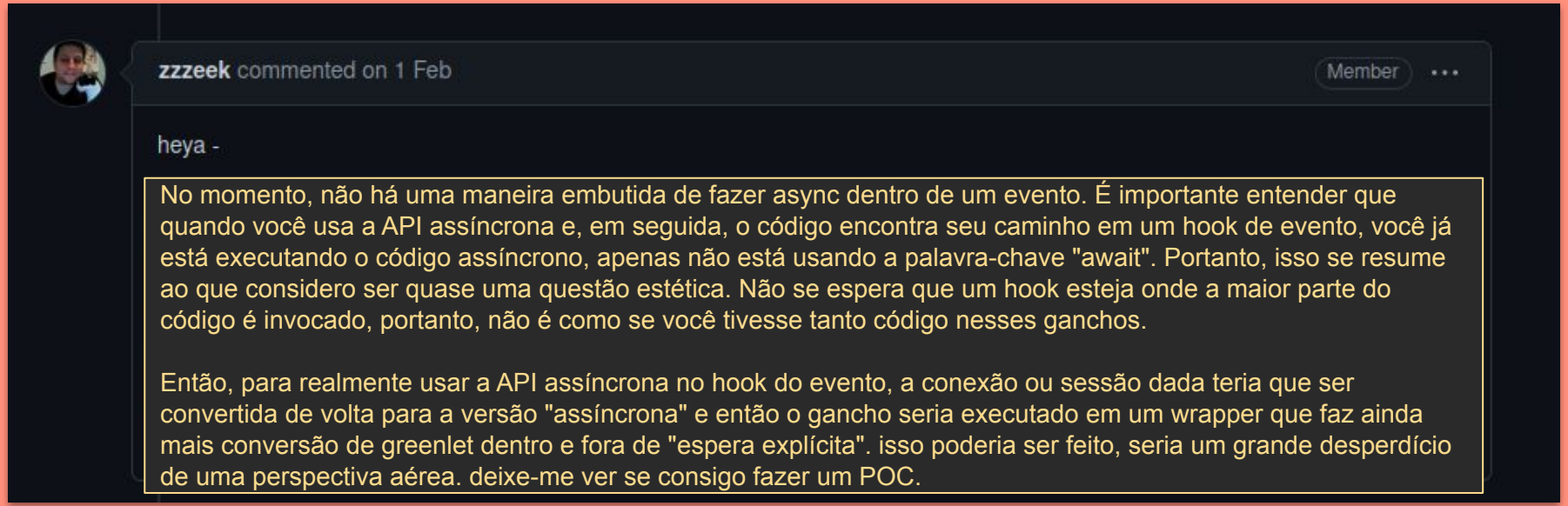
heya -

at the moment there is no built-in way to do async inside of an event hook. it's important to understand that when you use the async API, and then the code finds its way into an event hook, you are actually running async code already, it just isn't using the "await" keyword. So this comes down to what I consider to be almost an aesthetic issue. an event hook is not expected to be where most code is invoked so it's not like you'd have that much code in these hooks.

so to actually use the async API in the event hook, the given connection or session would have to be converted back up to the "async" version and then the hook would be run in a wrapper that does still more greenlet conversion in and out of "explicit await". this could be done, it would just be very wasteful from an overhead perspective. let me see if I can do a POC.

https://github.com/sqlalchemy/sqlalchemy/issues/5905

Mais informações?



zzzeek commented on 1 Feb — Member

heya -

No momento, não há uma maneira embutida de fazer async dentro de um evento. É importante entender que quando você usa a API assíncrona e, em seguida, o código encontra seu caminho em um hook de evento, você já está executando o código assíncrono, apenas não está usando a palavra-chave "await". Portanto, isso se resume ao que considero ser quase uma questão estética. Não se espera que um hook esteja onde a maior parte do código é invocado, portanto, não é como se você tivesse tanto código nesses ganchos.

Então, para realmente usar a API assíncrona no hook do evento, a conexão ou sessão dada teria que ser convertida de volta para a versão "assíncrona" e então o gancho seria executado em um wrapper que faz ainda mais conversão de greenlet dentro e fora de "espera explícita". isso poderia ser feito, seria um grande desperdício de uma perspectiva aérea. deixe-me ver se consigo fazer um POC.

https://github.com/sqlalchemy/sqlalchemy/issues/5905