第 1 章

Engine

1.1　用語

定義 1. ***Dialect***

SQL の DBMS 固有のバージョンを指します。例えば **MySQL** と **PostgreSQL** では文法が違ったりしますが、その仕様を定義したものが Database Dialect になります。

定義 2. ***DBAPI***

Python から DBMS への接続するためのインターフェース仕様を定義したものです。最新版は PEP 249 – Python Database API Specification v2.0 です。ライブラリはこの標準仕様に沿って作成されています。

定義 3. ***Driver***

定義 2 を実装したものの総称です。PostgreSQL のドライバの代表的なものは **pg8000**、MySQL であれば **aiomysql** などがあります。

定義 4. ***Pool***

コネクションプール。長時間稼働する接続をメモリ上に保持して効率的に再利用したり、アプリケーションが同時に使用する接続の総数を管理したりするために使用される標準的な技術です。

1.2　Engine の役割

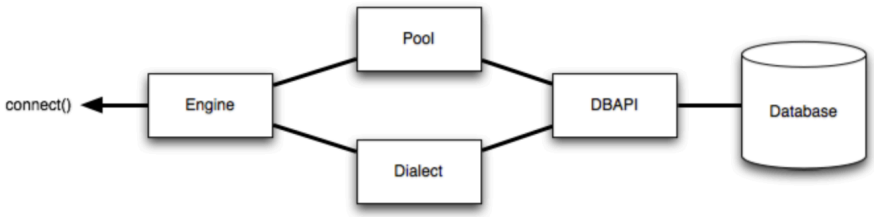

図 1.1　engine の役割

engine は Pool と Dialect オブジェクトを内部に持ち、DBAPI を操作します。engine の使い方には 2 パターンあり

ます。

1. engine から Session を生成して、ORM で DB にアクセスする。
2. engine から直接 SQL を発行する。

1.3 Engine の生成

engine は **create_engine()** 関数に RFC-1738 に準拠したデータベースへの URL を与えることで生成できます。その仕様は以下の通りです。

定義 5. *RFC-1738*

dialect+driver://username:password@host:port/database

engine は、生成されたタイミングで DB への接続を行うわけではないことに注意してください。のちに紹介する **Engine.execute()** や **Engine.connect()** が呼ばれたタイミングで初めて DB への接続が開始されます。

1.4 コネクションプール

1.5 Asynchronous I/O (asyncio)

注意

SQLAlchemy 1.4.3 においては非同期接続はベータ版の機能です。

1.5.1 環境設定

ソースコード 1.1 asyncio のインストール

```
1 pip install sqlalchemy[asyncio]
```

第 2 章

Session

2.1 用語

定義 6. ***Context Manager***

with 文に渡されるオブジェクトの総称を context manager と言います。context manager の中では、__enter__と__exit__というそれぞれ前処理と後処理を定義した特殊メソッドが実装されています。session においては、SQL を発行した後にそれが必ず commit され、さらにデータベースへの接続を必ず閉じるためにこの context manager が利用されます。

2.2 いつ使う？

- ORM を使って DB を操作するときに必ず使います。

2.3 Session の役割

2.3.1 インターフェースの提供

Engine の役割は Pool と Dialect を操作して DBAPI から DBMS を操作することでした。対して Session の役割は、**Engine を操作するためのインターフェースを提供すること**です。図 2.1 を見てください。一般的に、engine はプロセス内で 1 つだけ生成されます。engine に対して、session はクエリが発行されるなどの DB に対する処理を行うたびに生成され、その度に破棄されます。以下に session が生成されてから破棄されるまでのライフサイクルを説明していきます。

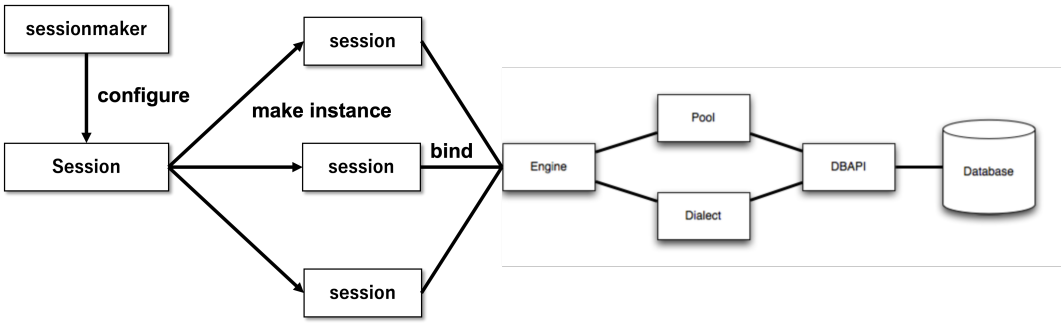

図 2.1 session の役割

2.3.2 コンテクストマネージャ

Session はコンテクストマネージャとしての役割を持っています。コンテクストマネージャは、特定の処理の最後に必ず行う __exit__を自動的に行ってくれるのでした。Session クラスをインスタンス化した session が生成され、処理が終了したのちに、**session.close()** が必ず呼び出され、DB への処理の間に保持していた ORM オブジェクトや connection オブジェクトを全て破棄します。

ソースコード 2.1 コンテクストマネージャ

```
1  with Session(engine) as session:
2    session.add(some_object)
3    session.add(some_other_object)
4    session.commit()
```

2.4 sessionmaker

2.4.1 どこで呼び出すべきか？

チュートリアルや他の解説記事では 1 つの.py ファイルの中で session の生成を行っていますが、実際にはモジュールごとに分割して管理することが多いはずです。その際に、sessionmaker はグローバルスコープで 1 回のみ呼ぶようにして、他のモジュールは import で Session を呼ぶようにしてください。

第 3 章

ORM

3.1 リレーション

3.1.1 多対多

関連テーブルを作ります。例として、Group テーブルと Location テーブルを相互に参照する関係を記述します。

ソースコード 3.1 多対多

```
GroupLocation = Table('group_location', Base.metadata,
  Column('group_id', ForeignKey('groups.id'), primary_key=True),
  Column('location_id', ForeignKey('locations.id'), primary_key=True)
)
```

各テーブル間のリレーションは **relation** パラメータを与えることで実現します。また、relationship.secondary 引数にこの関連テーブルを指定します。

ソースコード 3.2 テーブル定義

```
class Group(Base):
  __tablename__ = "groups"
  id = Column(Integer, primary_key=True)
  name = Column(String(255))
  label = Column(String(255))
  users = relationship("UserTable", back_populates="group")
  locations = relationship(
      "Location",
      secondary=GroupLocation,
      backref="groups")

class Location(Base):
  __tablename__ = "locations"
  id = Column(Integer, primary_key=True)
  name = Column(String(255))
  label = Column(String(255))
  latitude = Column(Float)
  longitude = Column(Float)
```

第 4 章

FastAPI

本章では、Web フレームワークである FastAPI を利用し、SQLAlchemy の実用例を見ていきます。

4.1 用語

定義 7. パス

最初の / から始まる URL の最後の部分です。

定義 8. オペレーション

HTTP メソッドの一つを指します。

定義 9. パスオペレーション関数

特定のパスに対して要求があったときに実行される関数です。

定義 10. モデル

ここでは、データベースのテーブルを SQLAlchemy で実装したオブジェクトを指します。

定義 11. スキーマ

ここでは、Pydantic によって定義されたオブジェクトのことを指します。

4.2 Session の注入

FastAPI の基本概念である Dendency Injection を利用して、各パスオペレーション関数にてデータベースを操作するための Session を使えるようにします。2.3.1 にて述べたように、Session はデータベース操作のたびに生成され、処理が終わったら破棄されます。この部分を実現するジェネレータ関数を、以下のように定義します。

ソースコード 4.1 セッションジェネレータ

```
1 # sessionmaker to make Session with api version 2.0 (future)
2 Session = sessionmaker(engine, future=True)
3
4 # dependency for each api call
5 def get_db_session():
6     db = Session()
```

```
    try:
        yield db
    finally:
        db.close()
```

ソースコード 4.2 パスオペレーション関数

```
@router.get('/get', response_model=List[schema.Group])
async def get_groups(request: Request, session: Session = Depends(get_db_session)):
    stmt = select(Group)
    groups = session.execute(stmt).scalars().all()
    return groups
```

Depends を利用して get_db_session 関数をパスオペレーションにて呼び出すと、呼ばれたときに Session がインスタンス化されて利用できるようになります。そしてレスポンスが返されて session 変数がスコープを抜けると、finally に定義された close が呼ばれ、session が必ず破棄されます。

4.3 スキーマの定義

データベース上の各モデルに対応するスキーマを定義する必要があります。スキーマを利用することで、SQLAlchemy のモデルを自動的に Map Object に変換してレスポンスを返せるようになります。例えば、以下のように Location というモデルがデータベースに定義されているとします。

ソースコード 4.3 Location Model

```
class Location(Base):
  __tablename__ = "locations"
  id = Column(Integer, primary_key=True)
  name = Column(String(255))
  label = Column(String(255))
  latitude = Column(Float)
  longitude = Column(Float)
```

これに対応するスキーマは以下のようにかけます。

ソースコード 4.4 Location schema

```
class LocationBase(BaseModel):
  name: str
  label: str
  latitude: Optional[float]
  longitude: Optional[float]

class LocationCreate(LocationBase):
  pass

class Location(LocationBase):
  id: str
  class Config:
      orm_mode = True
```

ここで、LocationCreate と Location を分けているのは、**リクエストボディで利用するのか、レスポンスボディで利用するのか**によって使い分けるためです。まず、リクエストボディとして Location の Map Object が送られてくるようなケースを想定します。パスオペレーション関数は次のようになるでしょう。

ソースコード 4.5 Create New Location

```
@router.post('/post')
async def post_location(
    location: schema.LocationCreate,
    session: Session = Depends(get_db_session)
):
    ...
    return location
```

リクエストボディに schema.LocationCreate を定義しています。もしこれを schema.Location に変更するとどうなるでしょうか。Location は、必須フィールドである id を持っています。これはモデルの primary key として設定している id に対応させるためです。通常、id は自動インクリメントで振られるため、ユーザからのリクエストボディに含まれることはありません。しかし、これがリクエストボディのスキーマとして与えられてしまうと、id がないためにバリデーションエラーになり、リクエストを受け付けることができません。そのため、リクエストボディには専用の LocationCreate スキーマを定義して利用しています。では GET の方はどうでしょうか。

ソースコード 4.6 Get Location List

```
@router.get('/get', response_model=List[schema.Location])
async def get_location(request: Request, session: Session = Depends(get_db_session)):
    stmt = select(Location)
    locations = session.execute(stmt).scalars().all()
    return locations
```

こちらの URL にアクセスすると、Location テーブルのデータの配列を全て取得することができます。ポイントは、**response_model** に Location スキーマを与えている点です。4 行目の locations は、クエリを発行して得られた location モデルの配列です。response_model を与えることにより、このモデルを自動的に Map object に変換して返してくれます。ポイントは、モデルとスキーマのフィールド定義が完全に一致していることです。フィールドの型が異なったり不足するフィールドがあると、バリデーションエラーにより値を返すことができません。ちなみに、response_model を定義せずに Map Object を返そうとすると、以下のような感じになります。

ソースコード 4.7 スキーマを使わない GET

```
@router.get('/get')
async def get_location(request: Request, session: Session = Depends(get_db_session)):
    stmt = select(Location)
    locations = session.execute(stmt).scalars().all()
    data = []
    for l in locations:
        data.append({
            "id": l.id,
            "label": l.label,
            "name": l.name,
            "latitude": l.latitude,
            "longitude": l.longitude
```

```
13          })
14      return data
```

モデルから必要なフィールドを取り出して、Map Object に変換したものを配列に詰めて返します。

4.4 各オペレーションの基本構文

この節では、各オペレーション（定義 8）における基本的な構文を見ていきます。

4.4.1 GET

全てのオブジェクトを取り出す

ソースコード 4.8 Get Location List

```
1  @router.get('/get', response_model=List[schema.Location])
2  async def get_location(session: Session = Depends(get_db_session)):
3      stmt = select(Location)
4      locations = session.execute(stmt).scalars().all()
5      return locations
```

条件に合致するオブジェクトを取り出す

ソースコード 4.9 Get Location on condition

```
1  @router.get('/get', response_model=schema.Location)
2  async def get_location(location_id: str, session: Session = Depends(get_db_session)):
3      stmt = select(Location).filter_by(id=location_id)
4      location = session.execute(stmt).scalar_one()
5      return location
```

リレーションがある場合

Group と Location が多対多のリレーションを貼っているとします。すると、モデル定義は以下のようになります。

ソースコード 4.10 モデル定義

```
1  GroupLocation = Table('group_location', Base.metadata,
2    Column('group_id', ForeignKey('groups.id'), primary_key=True),
3    Column('location_id', ForeignKey('locations.id'), primary_key=True)
4  )
5
6  class Group(Base):
7    __tablename__ = "groups"
8    id = Column(Integer, primary_key=True)
9    name = Column(String(255))
10   label = Column(String(255))
11   users = relationship("UserTable", back_populates="group")
12   locations = relationship(
13       "Location",
```

```
14        secondary=GroupLocation,
15        backref="groups")
16
17 class Location(Base):
18   __tablename__ = "locations"
19   id = Column(Integer, primary_key=True)
20   name = Column(String(255))
21   label = Column(String(255))
22   latitude = Column(Float)
23   longitude = Column(Float)
```

Group を GET で取り出すと、以下のようになります。

4.5 プロジェクトの構成

参考までに、FastAPIxSQLAlchemy のプロジェクト構成を載せておきます。

ソースコード 4.11 プロジェクト構成

```
1 models/
2   - __init__.py # ここにmodel の定義をここにします。
3 schemas/
4   - __init__.py # スキーマの定語をここにします。
5 routers/
6   - __init__.py # APIRouter の定義をここにします。
7   - some_router.py
8 config.py # 各種設定値を格納します。
9 main.py # エントリポイント
```