

استفاده از SQLAlchemy با ORM (Object-Relational Mapping)

SQLAlchemy یک ORM (Object-Relational Mapping) برای پایتون است که به شما اجازه می‌دهد بدون نوشتن SQL، با پایگاه داده کار کنید.

:SQLAlchemy مزایای استفاده از 

- کاهش پیچیدگی کد و خوانایی بیشتر
- انتقال‌پذیری بین پایگاه‌های داده (... و SQLite, MySQL, PostgreSQL)
- مدیریت خودکار تراکنش‌ها
- استفاده از شی‌گرایی در تعامل با پایگاه داده

۱. نصب و راهاندازی SQLAlchemy

برای استفاده از SQLAlchemy، ابتدا آن را نصب کنید:

```
pip install sqlalchemy
```

اگر از MySQL یا PostgreSQL استفاده می‌کنید، به درایورهای مربوطه نیز نیاز دارید:

```
pip install pymysql # برای MySQL  
pip install psycopg2 # برای PostgreSQL
```

۲. تعریف مدل‌های پایگاه داده با SQLAlchemy

↗ ایجاد پایگاه داده و جدول با ORM

از کلاس‌های پایتون برای نمایش جداول پایگاه داده استفاده می‌کند.

(۱) راهاندازی SQLAlchemy و ایجاد مدل

```
from sqlalchemy import create_engine, Column, Integer, String  
from sqlalchemy.ext.declarative import declarative_base  
from sqlalchemy.orm import sessionmaker
```

```
# برای مثال SQLite ایجاد اتصال به پایگاه داده  
DATABASE_URL = "sqlite:///test.db"  
engine = create_engine(DATABASE_URL, echo=True)
```

```
# تعریف بیس مدل‌ها  
Base = declarative_base()
```

```
# تعریف مدل کاربر (نماینده یک جدول در پایگاه داده)  
class User(Base):  
    __tablename__ = "users"  
  
    id = Column(Integer, primary_key=True, autoincrement=True)  
    name = Column(String(50), nullable=False)
```

```
email = Column(String(100), unique=True, nullable=False)
age = Column(Integer, nullable=True)
```

```
# ایجاد جداول در پایگاه داده
Base.metadata.create_all(engine)
```

```
# ایجاد سشن برای تعامل با پایگاه داده
SessionLocal = sessionmaker(bind=engine)
session = SessionLocal()
```

توضیحات:

- اتصال به پایگاه داده را مدیریت می‌کند.
- کلاس پایه‌ای برای مدل‌های پایگاه داده است.
- هر مدل، نماینده‌ی یک جدول در پایگاه داده است.
- جداول را در پایگاه داده ایجاد می‌کند.

۳. عملیات SQLAlchemy با CRUD

❖ ایجاد داده در پایگاه داده (Create)

```
new_user = User(name="Ali Ahmadi", email="ali@example.com", age=30)
session.add(new_user)
session.commit() # ذخیره تغییرات
```

توضیحات:

- یک کاربر جدید را به سشن اضافه می‌کند.
- تغییرات را در پایگاه داده ثبت می‌کند.

❖ خواندن داده‌ها از پایگاه داده (Read)

```
# دریافت همه کاربران
users = session.query(User).all()
for user in users:
    print(user.id, user.name, user.email, user.age)

# دریافت یک کاربر خاص
user = session.query(User).filter(User.email == "ali@example.com").first()
print(user.name)
```

نکات:

- همهی کاربران را دریافت می‌کند.
- اولین `(session.query(User).filter(User.email == "ali@example.com").first())` کاربری که ایمیلش مطابق باشد را دریافت می‌کند.

❖ به روزرسانی داده‌ها (Update)

```
user = session.query(User).filter(User.email == "ali@example.com").first()
if user:
    user.age = 35 # تغییر مقدار سن
    session.commit() # ذخیره تغییرات
```

نکته: پس از تغییر مقدار در شیء مدل، نیاز به `() commit` داریم تا تغییرات در پایگاه داده ذخیره شوند.

❖ حذف داده‌ها از پایگاه داده (Delete)

```
user = session.query(User).filter(User.email == "ali@example.com").first()
if user:
    session.delete(user)
    session.commit()
```

نکته: کاربر را حذف می‌کند، اما `() commit` برای اعمال نهایی تغییرات لازم است.

۴. مدیریت تراکنش‌ها در SQLAlchemy

SQLAlchemy به صورت خودکار تراکنش‌ها را مدیریت می‌کند، اما در صورت نیاز، می‌توانیم به صورت دستی تراکنش‌ها را کنترل کنیم:

```
try:
    new_user = User(name="Sara Karimi", email="sara@example.com", age=25)
    session.add(new_user)
    session.commit() # ذخیره تغییرات
    print("✓ کاربر اضافه شد")
except Exception as e:
    در صورت خطا، تغییرات لغو شود #
    print(f"✗ خطأ در تراکنش {e}")
```

نکات:

- تغییرات را ثبت می‌کند.
- در صورت بروز خطا تغییرات را لغو می‌کند.

۵. استفاده از MySQL و PostgreSQL در SQLAlchemy

اگر بخواهیم از SQLite یا MySQL به جای PostgreSQL استفاده کنیم، کافی است مقدار `DATABASE_URL` را تغییر دهیم.

MySQL اتصال به

```
DATABASE_URL = "mysql+pymysql://user:password@localhost/test_db"  
engine = create_engine(DATABASE_URL, echo=True)
```

استفاده می‌کند. //:mysql+pymysql از MySQL

PostgreSQL اتصال به

```
DATABASE_URL = "postgresql+psycopg2://user:password@localhost/test_db"  
engine = create_engine(DATABASE_URL, echo=True)
```

استفاده می‌کند. //:postgresql+psycopg2 از PostgreSQL

۶. مدیریت سشن‌ها به روش بهتر با with

برای جلوگیری از مشکلات مربوط به مدیریت سشن، می‌توان از Context Manager استفاده کرد:

```
from contextlib import contextmanager  
  
@contextmanager  
def get_session():  
    session = SessionLocal()  
    try:  
        yield session  
        session.commit()  
    except Exception as e:  
        session.rollback()  
        print(f"✗ خطأ: {e}")  
    finally:  
        session.close()
```

استفاده از تابع برای انجام عملیات پایگاه داده # است.

```
with get_session() as session:  
    user = session.query(User).filter(User.email == "sara@example.com").first()  
    print(user.name)
```

مزایا:

- مدیریت خودکار rollback و commit
- بستن خودکار سشن بعد از انجام عملیات

۷. جمع‌بندی

یک ORM قوی برای تعامل با پایگاه داده در پایتون است. SQLAlCHEMY
می‌توانیم مدل‌های شی‌گرا تعریف کنیم که به جداول پایگاه داده تبدیل می‌شوند.
می‌توان عملیات CRUD را بدون نوشتمن دستورات SQL مستقیم انجام داد.
مدیریت تراکنش‌ها و اتصال به پایگاه‌های داده PostgreSQL و MySQL بسیار ساده است.



(Many-to-Many و One-to-Many) بین جداول در SQLAlchemy

در SQLAlchemy، روابط بین جداول (مثل Many-to-Many و One-to-Many) را می‌توان به راحتی پیاده‌سازی کرد. در اینجا نحوه تعریف و کار با این نوع روابط آورده شده است.

۱. ارتباط One-to-Many

در ارتباط One-to-Many، یک رکورد در جدول والد می‌تواند چندین رکورد در جدول فرزند داشته باشد. مثلًاً یک کتاب می‌تواند چندین نویسنده داشته باشد، اما هر نویسنده تنها یک کتاب دارد.

نحوه پیاده‌سازی ارتباط One-to-Many

```
from sqlalchemy import create_engine, Column, Integer, String, ForeignKey
from sqlalchemy.orm import relationship, sessionmaker
from sqlalchemy.ext.declarative import declarative_base

# ایجاد پایگاه داده
DATABASE_URL = "sqlite:///test.db"
engine = create_engine(DATABASE_URL, echo=True)

Base = declarative_base()

# مدل والد (کتاب)
class Book(Base):
    __tablename__ = 'books'

    id = Column(Integer, primary_key=True, autoincrement=True)
    title = Column(String, nullable=False)

    # تعریف رابطه One-to-Many
    authors = relationship("Author", back_populates="book")

# مدل فرزند (نویسنده)
class Author(Base):
    __tablename__ = 'authors'

    id = Column(Integer, primary_key=True, autoincrement=True)
    name = Column(String, nullable=False)

    # خارجی کلید به مدل والد
    book_id = Column(Integer, ForeignKey('books.id'))
```

```

تعريف رابطه دو طرفه #
book = relationship("Book", back_populates="authors")

ایجاد جداول در پایگاه داده #
Base.metadata.create_all(engine)

ایجاد سشن #
SessionLocal = sessionmaker(bind=engine)
session = SessionLocal()

اضافه کردن داده‌ها
book = Book(title="Python Programming")
author1 = Author(name="Ali", book=book)
author2 = Author(name="Sara", book=book)

session.add(book)
session.commit()

خواندن داده‌ها
book = session.query(Book).filter_by(title="Python Programming").first()
for author in book.authors:
    print(f"{author.name} is an author of {book.title}")

بس تن سشن #
session.close()

```

توضیحات: 

- در این مثال، مدل `Author` دارای ارتباط `One-to-Many` با مدل `Book` است.
- با استفاده از `()` `relationship()` می‌توان رابطه بین جداول را تعریف کرد.
- برای ایجاد ارتباط دو طرفه استفاده `back_populates` می‌شود.
- کتاب می‌تواند چندین نویسنده داشته باشد، اما هر نویسنده فقط یک کتاب دارد.

۲. ارتباط Many-to-Many

در ارتباط **Many-to-Many**، چندین رکورد از یک جدول می‌تواند با چندین رکورد از جدول دیگر ارتباط داشته باشد. مثلاً یک دانشجو می‌تواند در چندین دوره شرکت کند، و هر دوره نیز می‌تواند شامل چندین دانشجو باشد.

نحوه پیاده‌سازی ارتباط Many-to-Many

برای پیاده‌سازی این ارتباط، معمولاً از یک جدول واسطه (Intermediate Table) استفاده می‌شود.

```

from sqlalchemy import create_engine, Column, Integer, String, Table, ForeignKey
from sqlalchemy.orm import relationship, sessionmaker
from sqlalchemy.ext.declarative import declarative_base

ایجاد پایگاه داده #
DATABASE_URL = "sqlite:///test.db"
engine = create_engine(DATABASE_URL, echo=True)

Base = declarative_base()

```

```

# جدول واسطه برای ارتباط Many-to-Many
student_course = Table('student_course', Base.metadata,
    Column('student_id', Integer, ForeignKey('students.id'), primary_key=True),
    Column('course_id', Integer, ForeignKey('courses.id'), primary_key=True)
)

# مدل دانشجو
class Student(Base):
    __tablename__ = 'students'

    id = Column(Integer, primary_key=True, autoincrement=True)
    name = Column(String, nullable=False)

    # تعریف رابطه Many-to-Many
    courses = relationship("Course", secondary=student_course, back_populates="students")

# مدل دوره
class Course(Base):
    __tablename__ = 'courses'

    id = Column(Integer, primary_key=True, autoincrement=True)
    title = Column(String, nullable=False)

    # تعریف رابطه Many-to-Many
    students = relationship("Student", secondary=student_course, back_populates="courses")

# ایجاد جداول در پایگاه داده
Base.metadata.create_all(engine)

# ایجاد سشن
SessionLocal = sessionmaker(bind=engine)
session = SessionLocal()

# اضافه کردن داده‌ها
student1 = Student(name="Ali")
student2 = Student(name="Sara")
course1 = Course(title="Python Basics")
course2 = Course(title="Advanced Python")

# بین دانشجویان و دوره‌ها ارتباط
student1.courses = [course1, course2]
student2.courses = [course1]

session.add(student1)
session.add(student2)
session.commit()

# خواندن داده‌ها
students_in_python_basics = session.query(Course).filter_by(title="Python Basics").first().students
for student in students_in_python_basics:
    print(f"{student.name} is enrolled in Python Basics")

# بستن سشن
session.close()

```

- برای ایجاد ارتباط Many-to-Many از یک جدول واسط به نام `student_course` استفاده کردیم که شامل کلیدهای خارجی به هر دو جدول `courses` و `students` است.
 - با استفاده از `Course` و `Student` ارتباط بین `secondary=student_course` برقرار می‌شود.
 - با استفاده از `Course` و `Student` رابطه دوطرفه بین `back_populates` تعریف شده است.
-

۳. جمع‌بندی

:One-to-Many

- ارتباطی که در آن یک رکورد از جدول والد می‌تواند چندین رکورد از جدول فرزند داشته باشد.
- برای ایجاد ارتباط استفاده `() relationship` و `ForeignKey` از

:Many-to-Many

- ارتباطی که در آن چندین رکورد از یک جدول می‌توانند با چندین رکورد از جدول دیگر ارتباط داشته باشند.
- یک جدول واسط برای نگهداری کلیدهای خارجی به هر دو جدول استفاده می‌شود.
- از `relationship` و `secondary` برای ایجاد ارتباط استفاده می‌شود.

 در ادامه: پیاده‌سازی فیلترها و جستجوهای پیچیده با SQLAlchemy

پیاده‌سازی فیلترها و جستجوهای پیچیده با SQLAlchemy

در SQLAlchemy، می‌توانیم فیلترهای پیچیده‌ای را برای انجام جستجوهای دقیق در پایگاه داده به کار ببریم. از جمله فیلترهایی مانند AND, OR, LIKE, IN, BETWEEN و همچنین توابع مقایسه‌ای برای انجام عملیات جستجوهای پیچیده.

در اینجا نحوه انجام فیلترها و جستجوهای پیچیده در SQLAlchemy توضیح داده شده است.

۱. استفاده از فیلترهای ساده

برای جستجوهای ساده می‌توان از متدهای `filter_by` و `filter` استفاده کرد.

جستجو با `() filter`

```
# جستجو بر اساس یک شرط ساده
student = session.query(Student).filter(Student.name == "Ali").first()
print(student.name)
```

جستجو با () filter_by

```
که از عملگرهای مقایسه‌ای مستقیم استفاده می‌کند `filter_by` جستجو با استفاده از #  
student = session.query(Student).filter_by(name="Ali").first()  
print(student.name)
```

۲. استفاده از عملگرهای منطقی

برای ترکیب چندین شرط از عملگرهای منطقی مانند AND, OR, NOT استفاده می‌کنیم.

استفاده از AND

برای انجام جستجوی چند شرطی که هم‌زمان باید برقرار باشند، از AND استفاده می‌کنیم:

```
# جستجو با شرط AND  
students = session.query(Student).filter(Student.age >= 18, Student.name == "Ali").all()  
for student in students:  
    print(student.name, student.age)
```

استفاده از OR

برای جستجوی رکوردهایی که حداقل یکی از شروط برقرار باشد، از OR استفاده می‌کنیم:

```
from sqlalchemy import or_  
  
# جستجو با شرط OR  
students = session.query(Student).filter(or_(Student.age < 18, Student.name == "Ali")).all()  
for student in students:  
    print(student.name, student.age)
```

۳. استفاده از عملگرهای مقایسه‌ای

در SQLAlchemy می‌توان از انواع عملگرهای مقایسه‌ای استفاده کرد.

استفاده از like برای جستجوی مشابه

اگر بخواهیم رشته‌هایی را که مشابه یک مقدار خاص هستند جستجو کنیم، می‌توان از like استفاده کرد:

```
# جستجو با `like`  
students = session.query(Student).filter(Student.name.like("Ali%")).  
all()  
for student in students:  
    print(student.name)
```

استفاده از `in` برای جستجو در مجموعه‌ای از مقادیر

برای جستجو در مجموعه‌ای از مقادیر می‌توان از `_in` استفاده کرد:

```
# جستجو با `in`  
students = session.query(Student).filter(Student.name.in_(['Ali', 'Sara'])).all()  
for student in students:  
    print(student.name)
```

استفاده از `between` برای جستجوی در محدوده‌ای از مقادیر

اگر بخواهیم رکوردهایی را که در یک بازه خاص قرار دارند پیدا کنیم، از `between` استفاده می‌کنیم:

```
# جستجو با `between`  
students = session.query(Student).filter(Student.age.between(18, 30)).all()  
for student in students:  
    print(student.name, student.age)
```

۴. استفاده از توابع SQL برای جستجوی پیچیده

این امکان را می‌دهد که از توابع مختلف SQL برای انجام جستجوهای پیچیده استفاده کنیم.

استفاده از `func` برای استفاده از توابع SQL

اگر بخواهیم توابع SQL مانند `COUNT()`, `SUM()`, `AVG()`, `MAX()`, `MIN` و ... را استفاده کنیم، از `func` استفاده می‌کنیم.

محاسبه تعداد رکوردها

```
from sqlalchemy import func  
  
محاسبه تعداد دانشجویان #  
student_count = session.query(func.count(Student.id)).scalar()  
print(f"تعداد دانشجویان: {student_count}")
```

محاسبه میانگین سنی

```
محاسبه میانگین سن دانشجویان #  
average_age = session.query(func.avg(Student.age)).scalar()  
print(f"میانگین سن: {average_age}")
```

محاسبه بیشترین سن

```
پیدا کردن بیشترین سن دانشجو #  
max_age = session.query(func.max(Student.age)).scalar()  
print(f"بیشترین سن: {max_age}")
```

۵. استفاده از `exists()` برای جستجو در صورتی که رکوردي وجود داشته باشد

اگر بخواهیم بررسی کنیم که آیا رکوردی با شرایط خاص وجود دارد یا نه، از `exists()` استفاده می‌کنیم:

```
from sqlalchemy import exists

# بررسی وجود یک دانشجو
exists_query = session.query(exists().where(Student.name == "Ali")).scalar()
if exists_query:
    print("دانشجو پیدا شد!")
else:
    print("دانشجو یافت نشد.")
```

۶. استفاده از `join()` برای انجام جستجوهای پیچیده بین چند جدول

در صورتی که بخواهیم بین جداول مختلف اطلاعات جستجو کنیم، می‌توانیم از `join()` استفاده کنیم.

استفاده از `join()` برای جستجو در جداول مرتبط

در اینجا فرض می‌کنیم که ارتباط بین `Course` و `Student` از نوع **Many-to-Many** است (مانند مثال‌های قبلی).

```
# جستجو با join
students_in_python = session.query(Student).join(student_course).join(Course).filter(Course.title == "Python Basics").all()
for student in students_in_python:
    print(student.name)
```

۷. استفاده از `group_by()` برای گروه‌بندی داده‌ها

اگر بخواهیم داده‌ها را بر اساس یک فیلد خاص گروه‌بندی کنیم (مانند محاسبه تعداد دانشجویان هر دوره)، می‌توانیم از `group_by()` استفاده کنیم.

گروه‌بندی بر اساس فیلد

```
# گروه‌بندی دانشجویان بر اساس سن
students_by_age = session.query(Student.age, func.count(Student.id)).group_by(Student.age).all()
for age, count in students_by_age:
    print(f"تعداد دانشجویان سن {age}: {count}")
```

۸. جمع‌بندی

- فیلترهای ساده: استفاده از `() filter_by` و `() filter` برای جستجو در پایگاه داده. ✓
- عملگرهای منطقی: استفاده از `AND`, `OR`, `NOT` برای ترکیب چندین شرط. ✓
- عملگرهای مقایسه‌ای: استفاده از `like`, `in_`, `between` برای جستجوهای خاص. ✓
- توابع SQL: استفاده از `COUNT()`, `SUM()`, `AVG` برای استفاده از توابع `func` و غیره. ✓
- وجود رکورد: استفاده از `exists()` برای بررسی وجود رکورد. ✓
- پیوست‌ها: استفاده از `join()` برای جستجو در جداول مرتبط. ✓
- گروه‌بندی: استفاده از `group_by()` برای گروه‌بندی داده‌ها. ✓

در ادامه: نحوه پیاده‌سازی اعتبارسنجی و مدیریت خطاهای در SQLAlchemy 

پیاده‌سازی اعتبارسنجی و مدیریت خطاهای در SQLAlchemy

در SQLAlchemy، می‌توان اعتبارسنجی‌ها و مدیریت خطاهای را برای جلوگیری از مشکلات و خطاهای رایج در زمان تعامل با پایگاه داده پیاده‌سازی کرد. اعتبارسنجی داده‌ها قبل از ذخیره‌سازی در پایگاه داده و همچنین مدیریت خطاهای احتمالی در طول عملیات مهم است.

در اینجا نحوه انجام اعتبارسنجی و مدیریت خطاهای در SQLAlchemy توضیح داده شده است.

۱. اعتبارسنجی داده‌ها قبل از ذخیره‌سازی در پایگاه داده

استفاده از ویژگی‌های SQLAlchemy برای اعتبارسنجی

در SQLAlchemy می‌توان از ویژگی‌های مدل‌ها برای اعمال اعتبارسنجی استفاده کرد. برای مثال، با استفاده از `nullable`, `default`, `unique` و دیگر ویژگی‌ها می‌توان محدودیت‌هایی را برای فیلد‌ها اعمال کرد.

مثال ۱: استفاده از ویژگی‌های مدل برای اعتبارسنجی

```
from sqlalchemy import create_engine, Column, Integer, String
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

# ایجاد پایگاه داده
DATABASE_URL = "sqlite:///test.db"
engine = create_engine(DATABASE_URL, echo=True)

Base = declarative_base()

# مدل با اعتبارسنجی برای فیلد‌ها
class Student(Base):
    __tablename__ = 'students'

    id = Column(Integer, primary_key=True, autoincrement=True)
    name = Column(String, nullable=False) # این فیلد نمی‌تواند خالی باشد
```

```
age = Column(Integer, nullable=False)
```

اعتبارسنجی نام دانشجو (حداقل ۳ کاراکتر)

```
def validate_name(self):
```

```
    if len(self.name) < 3:
```

```
        raise ValueError("نام دانشجو باید حداقل ۳ کاراکتر باشد")
```

ایجاد جداول در پایگاه داده

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

ایجاد سشن

```
SessionLocal = sessionmaker(bind=engine)
```

```
session = SessionLocal()
```

اضافه کردن داده‌ها

```
new_student = Student(name="Ali", age=22)
```

اعتبارسنجی قبل از ذخیره‌سازی

```
try:
```

```
    new_student.validate_name()
```

```
    session.add(new_student)
```

```
    session.commit()
```

```
except ValueError as e:
```

```
    session.rollback()
```

```
    print(f"خطا در اعتبارسنجی: {e}")
```

بستن سشن

```
session.close()
```

توضیحات:

- در اینجا از متدهای `validate_name` برای اعتبارسنجی نام دانشجو استفاده کردیم.
- اگر نام دانشجو کمتر از ۳ کاراکتر باشد، یک استثنای (Exception) ایجاد می‌شود و تغییرات باز می‌گردند.`(().rollback())`.

۲. مدیریت خطاها در پایگاه داده (Database Errors)

هنگام تعامل با پایگاه داده ممکن است خطاها مختلفی پیش بیاید، مانند خطاها در دسترسی به پایگاه داده، خطاها نقض محدودیت‌ها (مثل محدودیت‌های FOREIGN KEY یا UNIQUE)، و خطاها مرتبط با اتصال به پایگاه داده.

استفاده از `try-except` برای مدیریت خطاها در پایگاه داده

مثال ۲: مدیریت خطاها در پایگاه داده

```
from sqlalchemy.exc import IntegrityError, OperationalError
```

ایجاد سشن

```
SessionLocal = sessionmaker(bind=engine)
```

```
session = SessionLocal()
```

```
try:
```

```
    # اضافه کردن دانشجوی جدید
```

```
    new_student = Student(name="Ali", age=30)
```

```

session.add(new_student)
session.commit()

except IntegrityError as e:
    # خطای مربوط به نقض محدودیت‌ها مانند UNIQUE یا FOREIGN KEY
    session.rollback()
    print(f"خطای یکپارچگی پایگاه داده {e.orig}")

except OperationalError as e:
    # خطای مربوط به مشکلات اتصال به پایگاه داده
    session.rollback()
    print(f"خطای عملیاتی پایگاه داده {e.orig}")

except Exception as e:
    # سایر خطاهاي عمومي
    session.rollback()
    print(f"یک خطای غیرمنتظره رخ داده است {e}")

finally:
    # بستن سشن
    session.close()

```

توضیحات:

- در این مثال، از `try-except` برای مدیریت خطاها مربوط به پایگاه داده استفاده شده است.
- زمانی رخ می‌دهد که محدودیتی مانند `FOREIGN KEY` یا `UNIQUE` نقض شود.
- معمولًاً زمانی رخ می‌دهد که مشکلی در اتصال به پایگاه داده وجود داشته باشد (برای مثال پایگاه داده در دسترس نباشد).
- در صورت وقوع خطا، با استفاده از `() rollback` تغییرات به حالت اولیه برمی‌گردند.

۳. استفاده از اعتبارسنجی‌های سفارشی با SQLAlchemy Events

SQLAlchemy امکان استفاده از **رویدادها** (Events) را برای انجام اعتبارسنجی‌های پیشرفته فراهم می‌کند. به عنوان مثال، می‌توان از رویداد `before_update` و `before_insert` برای اعتبارسنجی و تغییر داده‌ها قبل از ذخیره‌سازی استفاده کرد.

مثال ۳: استفاده از SQLAlchemy Events

```

from sqlalchemy import event

# رویداد قبل از افزودن رکورد جدید
@event.listens_for(Student, 'before_insert')
def validate_student(mapper, connection, target):
    if target.age < 18:
        raise ValueError("سن دانشجو باید حداقل ۱۸ سال باشد")

# ایجاد سشن
SessionLocal = sessionmaker(bind=engine)
session = SessionLocal()

try:
    new_student = Student(name="Sara", age=17) # ۱۸
    session.add(new_student)
    session.commit()

```

```

session.add(new_student)
session.commit()

except ValueError as e:
    session.rollback()
    print(f"خطا در اعتبارسنجی: {e}")

finally:
    session.close()

```

توضیحات:

- در اینجا از `before_insert` برای اعتبارسنجی سن دانشجو قبل از افزودن به پایگاه داده استفاده کرده‌ایم.
- اگر سن دانشجو کمتر از ۱۸ باشد، یک استثنای `Exception` ایجاد شده و تغییرات باز می‌گردند.

۴. اعتبارسنجی داده‌ها با استفاده از کتابخانه‌های اضافی

برای اعتبارسنجی‌های پیچیده‌تر می‌توان از کتابخانه‌های اضافی مانند `Marshmallow` یا `Pydantic` استفاده کرد که امکان اعتبارسنجی داده‌ها را قبل از وارد کردن آنها به پایگاه داده فراهم می‌آورد.

۵. جمع‌بندی

اعتبارسنجی داده‌ها قبل از ذخیره‌سازی:

- استفاده از ویژگی‌های مدل `SQLAlchemy` مانند `nullable`, `unique`, `default` برای اعتبارسنجی ساده.
- پیاده‌سازی اعتبارسنجی سفارشی با استفاده از متدهای مدل.

مدیریت خطاها در پایگاه داده:

- استفاده از `try-except` برای مدیریت خطاها مختلف مانند خطاهای یکپارچگی و مشکلات اتصال به پایگاه داده.
- استفاده از رویدادهای `SQLAlchemy` برای اعتبارسنجی پیشرفته.

استفاده از رویدادهای `: SQLAlchemy`:

- پیاده‌سازی اعتبارسنجی با استفاده از `before_update` یا `before_insert` برای انجام عملیات قبل از وارد کردن داده‌ها به پایگاه داده.

در ادامه: نحوه پیاده‌سازی فیلترهای پیچیده و جستجوهای پیشرفته با `SQLAlchemy` 

پیاده‌سازی فیلترهای پیچیده و جستجوهای پیشرفته با SQLAlchemy

در SQLAlchemy، می‌توانیم از ترکیب ویژگی‌ها و توابع مختلف برای ایجاد فیلترهای پیچیده و جستجوهای پیشرفته استفاده کنیم. این کار به ویژه در هنگام کار با پایگاه داده‌های بزرگ یا نیاز به جستجوهای خاص بسیار مفید است.

در اینجا نحوه پیاده‌سازی فیلترهای پیچیده و جستجوهای پیشرفته با SQLAlchemy را توضیح می‌دهیم.

۱. استفاده از ترکیب فیلترها

برای انجام جستجوهای پیچیده با چندین شرط، می‌توان از عملگرهای منطقی `AND`, `OR`, `NOT` و ترکیب آن‌ها با متدهای `() filter_by` یا `() filter` استفاده کرد.

جستجو با ترکیب AND

برای جستجوهایی که چندین شرط باید همزمان برقرار باشند، از `AND` استفاده می‌کنیم.

```
from sqlalchemy import and_
# جستجو با AND
students = session.query(Student).filter(and_(Student.age >= 18, Student.name == "Ali")).all()
for student in students:
    print(student.name, student.age)
```

جستجو با ترکیب OR

برای جستجوهایی که حداقل یکی از شروط باید برقرار باشد، از `OR` استفاده می‌کنیم.

```
from sqlalchemy import or_
# جستجو با OR
students = session.query(Student).filter(or_(Student.age < 18, Student.name == "Ali")).all()
for student in students:
    print(student.name, student.age)
```

۲. استفاده از عملگرهای مقایسه‌ای

جستجو با like

برای جستجوهای مشابه می‌توان از `like` استفاده کرد. این برای یافتن رکوردهایی است که شامل یک رشته خاص در فیلد باشند.

```
# باشند "Ali" مثال: نام‌هایی که شامل like
students = session.query(Student).filter(Student.name.like("%Ali%")).all()
for student in students:
    print(student.name)
```

جستجو با `_in`

برای جستجو در مجموعه‌ای از مقادیر می‌توان از `_in` استفاده کرد.

```
#مثال: جستجو بر اساس چند نام خاص) _in_ جستجو با
students = session.query(Student).filter(Student.name.in_(['Ali', 'Sara'])).all()
for student in students:
    print(student.name)
```

جستجو با `between`

برای جستجوی داده‌هایی که در یک بازه خاص قرار دارند، از `between` استفاده می‌کنیم.

```
#مثال: سن دانشجو بین 18 و 30 باشد) between جستجو با
students = session.query(Student).filter(Student.age.between(18, 30)).all()
for student in students:
    print(student.name, student.age)
```

۳. استفاده از توابع SQL برای جستجوهای پیشرفته

امکان استفاده از توابع SQL مانند `COUNT()`, `AVG()`, `MAX()`, `MIN()` و دیگر توابع تجمعی را فراهم می‌کند.

استفاده از `func` برای توابع SQL

برای انجام جستجوهای پیچیده‌تر مانند محاسبات آماری، از `func` استفاده می‌کنیم.

محاسبه تعداد رکوردها

```
from sqlalchemy import func

#محاسبه تعداد دانشجویان
student_count = session.query(func.count(Student.id)).scalar()
print(f"تعداد دانشجویان: {student_count}")
```

محاسبه میانگین سن

```
#محاسبه میانگین سن
average_age = session.query(func.avg(Student.age)).scalar()
print(f"میانگین سن: {average_age}")
```

```
# پیدا کردن بیشترین سن
max_age = session.query(func.max(Student.age)).scalar()
print(f"بیشترین سن: {max_age}")
```

```
# پیدا کردن کمترین سن
min_age = session.query(func.min(Student.age)).scalar()
print(f"کمترین سن: {min_age}")
```

۴. گروه‌بندی داده‌ها با `group_by`

اگر بخواهیم داده‌ها را بر اساس یک ویژگی خاص گروه‌بندی کنیم (مثل شمارش دانشجویان بر اساس سن)، می‌توانیم از `group_by` استفاده کنیم.

گروه‌بندی بر اساس یک فیلد

```
# گروه‌بندی دانشجویان بر اساس سن
students_by_age = session.query(Student.age, func.count(Student.id)).group_by(Student.age).all()
for age, count in students_by_age:
    print(f"تعداد دانشجویان سن: {age}: {count}")
```

گروه‌بندی و فیلتر کردن نتایج

می‌توانیم پس از گروه‌بندی داده‌ها، نتایج را فیلتر کنیم تا فقط گروه‌های خاصی را نمایش دهیم.

```
# گروه‌بندی بر اساس سن و فیلتر برای سن‌های بزرگتر از 20
students_by_age = session.query(Student.age,
                                func.count(Student.id)).group_by(Student.age).having(func.count(Student.id) > 1).all()
for age, count in students_by_age:
    print(f"تعداد دانشجویان سن: {age}: {count}")
```

۵. استفاده از `join` برای جستجو در چندین جدول

به شما این امکان را می‌دهد که جداول مختلف را با یکدیگر پیوست (`join`) کرده و جستجوهایی انجام دهید که نیاز به داده‌های چندین جدول دارند.

پیوست ساده بین جداول

فرض کنید که می‌خواهید دانشجویانی را پیدا کنید که در دوره خاصی ثبت‌نام کردند.

```
# پیوست ساده برای جستجو در چند جدول
students_in_python = session.query(Student).join(Student.courses).filter(Course.title == "Python Basics").all()
for student in students_in_python:
    print(student.name)
```

پیوست پیچیده با استفاده از `outerjoin`

اگر بخواهیم جستجویی انجام دهیم که برخی رکوردها ممکن است در جداول دیگر موجود نباشند، می‌توانیم از استفاده کنیم.

```
# پیوست خارجی برای جستجو در جداول با داده‌های ناقص
students_without_courses = session.query(Student).outerjoin(Student.courses).filter(Course.title == "None").all()
for student in students_without_courses:
    print(student.name)
```

۶. استفاده از `exists()` برای بررسی وجود رکورد

اگر بخواهید بررسی کنید که آیا رکوردي با شرایط خاص وجود دارد، از `exists()` استفاده می‌کنید.

```
from sqlalchemy import exists

# بررسی وجود یک دانشجو با نام خاص
exists_query = session.query(exists().where(Student.name == "Ali")).scalar()
if exists_query:
    print("دانشجو پیدا شد")
else:
    print("دانشجو یافت نشد")
```

۷. پیاده‌سازی جستجو با فیلترهای داینامیک

گاهی اوقات ممکن است بخواهیم جستجوهایی با فیلترهای داینامیک داشته باشیم که کاربر بتواند بر اساس پارامترهای مختلف فیلتر کند.

استفاده از فیلترهای داینامیک

```
# فرض کنید که کاربر می‌خواهد بر اساس نام یا سن فیلتر کند
filters = []
if name:
    filters.append(Student.name.like(f"%{name}%"))
if age:
    filters.append(Student.age == age)

# اعمال فیلترهای داینامیک
students = session.query(Student).filter(*filters).all()
for student in students:
    print(student.name, student.age)
```

۸. جمع‌بندی

- ✓ ترکیب فیلترها: استفاده از `AND`, `OR` و ترکیب آن‌ها برای فیلترهای پیچیده.
 - ✓ عملگرهای مقایسه‌ای: استفاده از `like`, `in_`, `between` برای جستجوهای خاص.
 - ✓ تابع SQL: استفاده از `func` برای تابع تجمعی مانند `COUNT()`, `AVG()`, `MAX()` و غیره.
 - ✓ گروه‌بندی: استفاده از `group_by` برای گروه‌بندی داده‌ها و انجام محاسبات آماری.
 - ✓ پیوست‌ها: استفاده از `join` و `outerjoin` برای جستجو در جداول مرتبط.
 - ✓ بررسی وجود رکورد: استفاده از `exists` برای بررسی وجود رکوردها.
 - ✓ فیلترهای داینامیک: پیاده‌سازی فیلترهای داینامیک بر اساس ورودی‌های کاربر.
- در ادامه: نحوه پیاده‌سازی پیاده‌سازی مدیریت اتصال و سشن در SQLAlchemy 

پیاده‌سازی مدیریت اتصال و سشن در SQLAlchemy

در SQLAlchemy، مدیریت اتصال و سشن به صورت مؤثر نقش بسیار مهمی در بهینه‌سازی عملکرد و جلوگیری از مشکلات مرتبط با اتصالات باز یا منابع غیر بهینه دارد. در اینجا نحوه پیاده‌سازی مدیریت اتصال و سشن به طور دقیق آورده شده است.

۱. نصب SQLAlchemy و تنظیمات اولیه

قبل از هر چیز باید SQLAlchemy را نصب کرده و به‌طور کلی اتصال به پایگاه داده را راه‌اندازی کنیم.

```
pip install sqlalchemy
pip install psycopg2 # برای PostgreSQL
pip install mysql-connector-python # برای MySQL
```

۲. اتصال به پایگاه داده

برای اتصال به پایگاه داده، باید URI اتصال را مشخص کنیم. این URI شامل اطلاعاتی مانند نوع پایگاه داده، نام کاربری، رمز عبور، آدرس سرور و نام پایگاه داده است.

اتصال به پایگاه داده MySQL

```
from sqlalchemy import create_engine

# URI اتصال به MySQL
DATABASE_URL = "mysql+mysqlconnector://username:password@localhost/mydatabase"

# ایجاد موتور اتصال
engine = create_engine(DATABASE_URL, echo=True) # echo=True برای مشاهده درخواست‌های SQL
```

اتصال به پایگاه داده PostgreSQL

```
# URI اتصال به PostgreSQL
DATABASE_URL = "postgresql+psycopg2://username:password@localhost/mydatabase"

# ایجاد موتور اتصال
engine = create_engine(DATABASE_URL, echo=True)
```

۳. تنظیمات سشن (Session)

برای ارتباط و انجام عملیات در پایگاه داده، باید از `Session` استفاده کنیم. در `Session` برای انجام عملیات CRUD (Create, Read, Update, Delete) استفاده می‌شود.

ایجاد یک Session

برای ایجاد یک `Session` باید یک `sessionmaker` بسازیم که به طور خودکار با موتور (engine) اتصال برقرار می‌کند.

```
from sqlalchemy.orm import sessionmaker

# ایجاد یک sessionmaker
Session = sessionmaker(bind=engine)

# ساخت یک سشن
session = Session()
```

بستن سشن

پس از اتمام کار با پایگاه داده، باید سشن را ببندیم تا منابع آزاد شوند.

```
# بستن سشن
session.close()
```

۴. استفاده از سشن در عملیات پایگاه داده

پس از راه اندازی و ایجاد یک سشن، می‌توانیم عملیات مختلفی را در پایگاه داده انجام دهیم.

عملیات ایجاد (Insert)

```
from sqlalchemy import Column, Integer, String
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class User(Base):
    __tablename__ = 'users'

    id = Column(Integer, primary_key=True)
    name = Column(String)
    age = Column(Integer)
```

```
# ایجاد یک شی جدید از مدل User
new_user = User(name="Ali", age=25)

افزودن شی به سشن و ذخیره در پایگاه داده
session.add(new_user)
session.commit() # ذخیره تغییرات در پایگاه داده
```

عملیات خواندن (Select)

```
# خواندن تمام رکوردها از جدول Users
users = session.query(User).all()

for user in users:
    print(user.name, user.age)
```

عملیات بهروزرسانی (Update)

```
# به روزرسانی یک رکورد
user_to_update = session.query(User).filter(User.name == "Ali").first()
user_to_update.age = 30
session.commit() # ذخیره تغییرات
```

عملیات حذف (Delete)

```
# حذف یک رکورد
user_to_delete = session.query(User).filter(User.name == "Ali").first()
session.delete(user_to_delete)
session.commit() # ذخیره تغییرات
```

۵. مدیریت تراکنش‌ها با commit() و rollback()

تراکنش‌ها در SQLAlchemy برای اطمینان از صحت داده‌ها و انجام چندین عملیات به صورت اتمیک (یا همه یا هیچ) به کار می‌روند.

استفاده از commit()

اگر بخواهید تغییرات را در پایگاه داده ذخیره کنید، از commit() استفاده می‌کنید.

```
# ذخیره تغییرات
session.commit()
```

استفاده از rollback()

اگر یک خطأ رخ دهد و بخواهید تغییرات را لغو کنید، از rollback() استفاده می‌کنید.

```
from sqlalchemy.exc import SQLAlchemyError
```

try:

عملیات در پایگاه داده

```
new_user = User(name="Sara", age=28)
```

```
session.add(new_user)
```

```
session.commit()
```

```
except SQLAlchemyError:
```

در صورت وقوع خطا تغییرات برگشت داده می‌شود

```
session.rollback()
```

```
print("خطا در انجام عملیات!")
```

۶. استفاده از `with` برای مدیریت خودکار سشن‌ها

یکی از روش‌های خوب برای مدیریت سشن‌ها استفاده از ساختار `with` است. این روش به طور خودکار سشن را باز و بسته می‌کند و به شما این امکان را می‌دهد که کد خود را ساده‌تر و ایمن‌تر بنویسید.

```
from sqlalchemy.orm import sessionmaker
```

```
# ایجاد یک sessionmaker
```

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

برای مدیریت سشن `with` استفاده از

```
with Session() as session:
```

```
new_user = User(name="Reza", age=22)
```

```
session.add(new_user)
```

```
session.commit()
```

در این روش، به‌طور خودکار پس از اتمام عملیات سشن بسته می‌شود، بدون نیاز به فراخوانی `session.close()`.

۷. مدیریت چندین اتصال (Connection Pooling)

SQLAlchemy به‌طور پیش‌فرض از اتصال به پایگاه داده به‌صورت pooling (گروه‌بندی اتصالات) استفاده می‌کند که باعث می‌شود چندین اتصال به‌طور همزمان به پایگاه داده باز شود، بدون اینکه نیاز به ایجاد مجدد اتصال در هر درخواست باشد.

Pooling پیشرفته

```
from sqlalchemy import create_engine
```

تنظیمات پیشرفته برای pooling

```
engine = create_engine(DATABASE_URL, pool_size=10, max_overflow=20, echo=True)
```

• `pool_size`: تعداد اتصالات در دسترس برای استفاده.

• `max_overflow`: تعداد اتصالات اضافی که می‌توانند ساخته شوند.

۸. مدیریت خطاهای داده و استثنایها

هنگام انجام عملیات پایگاه داده، ممکن است خطاهایی رخ دهند که باید مدیریت شوند. استفاده از ساختار `try-except` به شما کمک می‌کند تا خطاهای را به درستی مدیریت کنید.

```
from sqlalchemy.exc import IntegrityError, SQLAlchemyError
```

`try:`

 انجام عملیات پایگاه داده #

```
    session.add(new_user)
```

```
    session.commit()
```

`except IntegrityError:`

 اگر خطای یکپارچگی رخ دهد تغییرات برگشت داده می‌شود ()

```
    print("خطای یکپارچگی داده!")
```

`except SQLAlchemyError as e:`

 اگر خطای عمومی رخ دهد #

```
    print(f"خطای عمومی: {e}")
```

جمع‌بندی

- اتصال به پایگاه داده: استفاده از `create_engine` برای ایجاد اتصال به پایگاه داده.
- مدیریت سشن: استفاده از `sessionmaker` برای ایجاد و مدیریت سشن‌ها.
- عملیات CRUD: انجام عملیات `Create`, `Read`, `Update`, `Delete` با استفاده از سشن.
- مدیریت تراکنش‌ها: استفاده از `rollback()` و `commit()` برای مدیریت تغییرات.
- استفاده از `with`: مدیریت خودکار سشن‌ها با استفاده از ساختار `with`.
- استفاده از Pooling: استفاده از اتصال‌های گروهی (pooling) برای بهینه‌سازی اتصالات.

با استفاده از این روش‌ها، می‌توانید اتصال به پایگاه داده و عملیات مربوطه را به طور مؤثر و بهینه در SQLAlchemy مدیریت کنید! 

مهاجرت‌های پایگاه داده با Alembic

Alembic ابزاری برای مدیریت مهاجرت‌های پایگاه داده در SQLAlchemy است. مهاجرت‌ها به شما این امکان را می‌دهند که تغییرات در ساختار پایگاه داده (مثل اضافه کردن یا حذف جداول، ستون‌ها، ایندکس‌ها و ...) را به صورت سیستماتیک و قابل پیگیری انجام دهید.

در پروژه‌هایی که نیاز به تغییرات مکرر در ساختار پایگاه داده دارند، استفاده از یک سیستم مهاجرت، نظیر Alembic، بسیار مفید است تا همه تغییرات به صورت کنترل شده انجام شوند و از مشکلات سازگاری نسخه‌های مختلف پایگاه داده جلوگیری شود.

1. نصب Alembic

برای استفاده از Alembic، ابتدا باید آن را نصب کنید:

```
pip install alembic
```

2. راهاندازی Alembic

پس از نصب Alembic، باید آن را در پروژه خود راهاندازی کنید. در اینجا مراحلی برای راهاندازی اولیه آورده شده است.

1. در ریشه پروژه خود یک دایرکتوری جدید ایجاد کنید و در آن دستور زیر را وارد کنید:

```
alembic init alembic
```

این دستور یک پوشه به نام `alembic` ایجاد می‌کند که شامل فایل‌های پیکربندی و اسکریپت‌های مهاجرت است. ساختار پروژه پس از اجرای دستور به صورت زیر خواهد بود:

```
myproject/
├── alembic/
│   ├── versions/
│   └── alembic.ini
└── app/
    └── ...
```

در این ساختار:

- فایل پیکربندی Alembic `alembic.ini`
- پوشه `/versions` محل ذخیره اسکریپت‌های مهاجرتی است.

1. در فایل `alembic.ini` تنظیمات مربوط به اتصال به پایگاه داده را وارد کنید. به طور مثال، اگر از پایگاه Dade SQLite استفاده می‌کنید:

```
sqlalchemy.url = sqlite:///your_database.db
```

1. فایل `env.py` در پوشه `alembic` قرار دارد که می‌توانید آن را برای انجام تنظیمات بیشتر اصلاح کنید. این فایل مسئول بارگذاری تنظیمات مربوط به پایگاه داده و مدل‌های SQLAlchemy شما است.

3. ایجاد مهاجرت‌های جدید

برای ایجاد مهاجرت‌های جدید، می‌توانید از دستور `alembic revision --autogenerate` استفاده کنید که به طور خودکار تغییرات بین مدل‌های فعلی و پایگاه داده را شناسایی کرده و یک فایل مهاجرت جدید ایجاد می‌کند.

برای ایجاد یک مهاجرت جدید، دستور زیر را اجرا کنید:

```
alembic revision -autogenerate -m "Add new column to user table"
```

این دستور یک فایل جدید در پوشه `/versions` ایجاد می‌کند که شامل تغییرات شناسایی شده است. اگر تغییرات شناسایی نشدنی یا نیاز به اصلاح دارند، باید آنها را به صورت دستی وارد کنید.

4. محتوای فایل مهاجرت

فایل‌های مهاجرتی به صورت خودکار یا دستی ایجاد می‌شوند و شامل دو متد اصلی هستند:

- `() upgrade`: تغییرات جدیدی که باید در پایگاه داده اعمال شود.
- `() downgrade`: تغییراتی که برای بازگشت به وضعیت قبلی نیاز است.

یک مثال از محتوای فایل مهاجرت:

```
"""Add new column to user table"""

from alembic import op
import sqlalchemy as sa

# این کد برای اعمال تغییرات جدید
def upgrade():
    op.add_column('user', sa.Column('age', sa.Integer(), nullable=True))

# این کد برای برگشت به وضعیت قبلی
def downgrade():
    op.drop_column('user', 'age')
```

در این مثال:

- ستون جدید `user age` را به جدول `user` اضافه می‌کند.
- این ستون را حذف می‌کند.

5. اجرای مهاجرت‌ها

پس از ایجاد فایل مهاجرت، برای اعمال تغییرات به پایگاه داده از دستور `alembic upgrade` استفاده می‌کنید. این دستور مهاجرت‌ها را از نقطه فعلی پایگاه داده اعمال می‌کند.

برای اعمال تمام مهاجرت‌ها:

```
alembic upgrade head
```

اگر بخواهید به یک نسخه خاص از پایگاه داده بروید، می‌توانید از شناسه نسخه استفاده کنید:

```
alembic upgrade <revision_id>
```

6. برگشت به نسخه قبلی (Downgrade)

اگر بخواهید تغییرات را معکوس کنید و به نسخه قبلی پایگاه داده برگردید، می‌توانید از دستور `alembic downgrade` استفاده کنید:

```
alembic downgrade -1
```

این دستور آخرین مهاجرت را برگشت می‌دهد و به نسخه قبلی پایگاه داده می‌رود.

7. شناسایی تغییرات

گاهی اوقات نیاز دارید که تغییرات پایگاه داده به صورت دستی شناسایی و تنظیم شوند. مثلاً اگر در ساختار پایگاه داده تغییرات اساسی دادهاید که قادر به شناسایی آنها نیست، باید به صورت دستی در فایل مهاجرت تغییرات را اعمال کنید.

8. مهاجرت‌های پیشرفته

- **تغییر نوع داده‌ها:** گاهی اوقات نیاز است که نوع داده یک ستون تغییر کند. این کار نیاز به عملیات پیچیده‌ای دارد که باید به طور دستی در فایل‌های مهاجرت مشخص شود.
- **شروط پیچیده برای اعمال مهاجرت:** مانند انتقال داده‌ها به ستون‌های جدید یا تبدیل داده‌های موجود.

9. مدیریت چند پایگاه داده

اگر پروژه شما از چند پایگاه داده مختلف استفاده می‌کند، می‌توانید با استفاده از چندین موتور (engine) در Alembic از آنها برای مدیریت مهاجرت‌ها به طور همزمان استفاده کنید. این کار معمولاً نیازمند پیکربندی اضافی در فایل `env.py` است.

10. نکات مهم برای استفاده از Alembic

- **کنترل نسخه پایگاه داده:** استفاده از Alembic می‌تواند به شما کمک کند تا نسخه‌های مختلف پایگاه داده را مدیریت کنید و مهاجرت‌های مختلف را به صورت دقیق پیگیری نمایید.
- **پشتیبانی از محیط‌های مختلف:** امکان مدیریت مهاجرت‌های پایگاه داده در محیط‌های مختلف (مانند توسعه، تست، و تولید) را فراهم می‌کند.
- **ایجاد و اعمال تغییرات با احتیاط:** همیشه قبل از اعمال تغییرات، پایگاه داده خود را پشتیبان‌گیری کنید، بهویژه زمانی که تغییرات عملدهای انجام می‌دهید.

جمع‌بندی

Alembic ابزاری قدرتمند برای مدیریت مهاجرت‌های پایگاه داده در پروژه‌های استفاده‌کننده از SQLAlchemy است. این ابزار به شما این امکان را می‌دهد که تغییرات ساختاری در پایگاه داده را به صورت خودکار و پیوسته اعمال کنید و از مشکلات مربوط به هم‌زمانی نسخه‌ها و تغییرات پایگاه داده جلوگیری نمایید.

اگر پروژه شما شامل چند مدل SQLAlchemy باشد (مثلاً چند فایل جداگانه برای مدل‌ها)، باید اطمینان حاصل کنید که تمامی این مدل‌ها دسترسی داشته باشند. در این صورت، باید تغییراتی در فایل `env.py` اعمال کنید تا Alembic تمام مدل‌های موجود را تشخیص دهد.

1. ساختار پروژه با چند مدل

فرض کنید ساختار پروژه شما به شکل زیر است:

```
my_project/
|
|   models/
|   |   __init__.py
|   |   user.py
|   |   product.py
|   |   order.py
|
|   alembic/
|   |   versions/
|   |   env.py
|   |   script.py.mako
|   |   alembic.ini
|
|   main.py
```

در اینجا:

- `User` مدل: `models/user.py`
- `Product` مدل: `models/product.py`
- `Order` مدل: `models/order.py`

2. تنظیم فایل `env.py` برای چند مدل

برای اینکه Alembic بتواند تمام مدل‌ها را تشخیص دهد، باید در فایل `env.py` از `declarative_base` استفاده کنید. در ادامه نحوه انجام این کار توضیح داده شده است:

مرحله 1: وارد کردن تمام مدل‌ها

در قسمت بالای فایل `env.py`، تمام مدل‌های موجود را وارد کنید:

```
# Import all your models here
from models.user import Base as UserBase
from models.product import Base as ProductBase
from models.order import Base as OrderBase
```

مرحله 2: ترکیب Metadata

Alembic نیاز دارد که تمام جداول مرتبط در یک `Base` واحد قرار گیرند. بنابراین، می‌توانید از یک `Base` واحد استفاده کنید یا `Metadata` های مختلف را ترکیب کنید.

اگر از یک `Base` واحد استفاده می‌کنید:

اگر تمام مدل‌های شما از یک `Base` مشترک ارث بری می‌کنند، می‌توانید آن `Base` را مستقیماً در `target_metadata` قرار دهید:

```
from models.user import Base # All models inherit from this Base  
  
target_metadata = Base.metadata
```

اگر از چند `Base` مختلف استفاده می‌کنید:

اگر مدل‌های شما از بیش از یک `Base` مختلف استفاده می‌کنند، باید `Metadata`‌های آن‌ها را ترکیب کنید:

```
from sqlalchemy import MetaData  
  
# Combine metadata from all bases  
metadata = MetaData()  
for base in [UserBase, ProductBase, OrderBase]:  
    for table in base.metadata.tables.values():  
        table.tometadata(metadata)  
  
target_metadata = metadata
```

3. تنظیم `sys.path` برای دسترسی به مدل‌ها

برای اینکه Alembic بتواند به فایل‌های مدل شما دسترسی داشته باشد، باید مسیر پروژه شما به `sys.path` اضافه شود. این کار در ابتدا در فایل `env.py` انجام می‌شود:

```
import sys  
import os  
  
# Add the project directory to the Python path  
sys.path.append(os.path.abspath(os.path.dirname(__file__) + '/../'))
```

4. مثال کامل فایل `:env.py`

```
from logging.config import fileConfig  
from sqlalchemy import engine_from_config  
from sqlalchemy import pool  
from alembic import context  
  
# Import all your models here  
from models.user import Base as UserBase  
from models.product import Base as ProductBase  
from models.order import Base as OrderBase  
  
# Combine metadata from all bases  
from sqlalchemy import MetaData  
  
metadata = MetaData()  
for base in [UserBase, ProductBase, OrderBase]:  
    for table in base.metadata.tables.values():  
        table.tometadata(metadata)  
  
target_metadata = metadata
```

```
# other imports ...
import sys
import os

# Add the project directory to the Python path
sys.path.append(os.path.abspath(os.path.dirname(__file__) + '/../'))

# this is the Alembic Config object, which provides
# access to the values within the .ini file in use.
config = context.config

# Interpret the config file for Python logging.
fileConfig(config.config_file_name)

# Add your database URL to the Alembic configuration
config.set_main_option("sqlalchemy.url",
"postgresql://your_username:your_password@localhost/your_database")

def run_migrations_offline():
    url = config.get_main_option("sqlalchemy.url")
    context.configure(
        url=url,
        target_metadata=target_metadata,
        literal_binds=True,
        dialect_opts={"paramstyle": "named"},
    )

    with context.begin_transaction():
        context.run_migrations()

def run_migrations_online():
    connectable = engine_from_config(
        config.get_section(config.config_ini_section),
        prefix="sqlalchemy.",
        poolclass=pool.NullPool,
    )

    with connectable.connect() as connection:
        context.configure(
            connection=connection, target_metadata=target_metadata
        )

        with context.begin_transaction():
            context.run_migrations()

if context.is_offline_mode():
    run_migrations_offline()
else:
    run_migrations_online()
```

5. ایجاد مهاجرت برای چند مدل

حالا که تمام مدل‌های شما دسترسی دارد، می‌توانید مهاجرت‌ها را به طور معمول ایجاد کنید:

```
alembic revision -autogenerate -m "Add user, product, and order models"  
alembic upgrade head
```

6. نکات مهم

- هر مدل باید از یک `Base` مشترک یا `MetaData` مناسب استفاده کند: این اطمینان می‌دهد که بتواند تمام جداول را تشخیص دهد.
- اطلاعات دیتابیس را در `alembic.ini` تنظیم کنید: اطمینان حاصل کنید که درستی تنظیم شده باشد.
- آپدیت `env.py` با توجه به ساختار پروژه خود: اگر مدل‌ها در فایل‌های مختلف یا پوشش‌های مختلفی قرار دارند، مسیرهای وارد شده را بررسی کنید.

خلاصه

وقتی چند مدل دارید، باید:

1. تمام مدل‌ها را در فایل `env.py` وارد کنید.
2. `Metadata`‌های آن‌ها را ترکیب کنید.
3. مطمئن شوید که تمام مدل‌ها دسترسی داشته باشد.

با این روش، Alembic به راحتی می‌تواند تمام تغییرات ساختاری دیتابیس را مدیریت کند. 😊

مدیریت داده‌های JSON و داده‌های پیچیده در SQLAlchemy

این امکان را می‌دهد که داده‌های پیچیده و ساختارهای مختلف، از جمله داده‌های JSON را به‌طور مستقیم در پایگاه داده ذخیره و بازیابی کنید. این ویژگی به‌ویژه برای پایگاه‌داده‌ای مانند PostgreSQL که از نوع داده `JSON` به‌صورت بومی پشتیبانی می‌کنند، مفید است، اما حتی در پایگاه‌های داده‌ای مثل MySQL و SQLite نیز می‌توان از آن بهره برد.

در اینجا روش‌های مدیریت داده‌های JSON و داده‌های پیچیده در SQLAlchemy شرح داده شده است.

1. تعریف فیلد JSON در مدل

در SQLAlchemy می‌توان از نوع داده `JSON` برای ذخیره داده‌ها به‌صورت JSON در پایگاه‌داده استفاده کرد. این نوع داده به شما این امکان را می‌دهد که داده‌ها را به صورت آرایه‌ها، اشیاء و انواع دیگر پیچیده ذخیره کنید.

PostgreSQL 1.1

از نوع داده `JSON` و `JSONB` به‌طور بومی پشتیبانی می‌کند. در SQLAlchemy، می‌توانید از این نوع داده‌ها برای ذخیره و مدیریت داده‌های JSON استفاده کنید.

```
from sqlalchemy import create_engine, Column, Integer, String, JSON
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

Base = declarative_base()

# است JSON مدل که شامل فیلد
class User(Base):
    __tablename__ = 'users'

    id = Column(Integer, primary_key=True)
    name = Column(String, nullable=False)
    preferences = Column(JSON)

    # اتصال به پایگاه داده PostgreSQL
    engine = create_engine('postgresql://username:password@localhost:5432/mydatabase')
    Base.metadata.create_all(engine)

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

    # به پایگاه داده JSON اضافه کردن داده‌های
    new_user = User(name='John', preferences={"theme": "dark", "notifications": True})
    session.add(new_user)
    session.commit()
```

در این مثال:

- فیلد `preferences` از نوع `JSON` است که به شما این امکان را می‌دهد که داده‌های JSON را در پایگاه داده ذخیره کنید.

SQLite 1.2

در SQLite از نوع داده `TEXT` برای ذخیره JSON استفاده می‌شود. اگرچه SQLite به طور بومی از نوع داده `JSON` پشتیبانی نمی‌کند، می‌توانید داده‌های JSON را به صورت رشته ذخیره کنید و از `json` در Python برای تجزیه و تحلیل آنها استفاده کنید.

```
from sqlalchemy import create_engine, Column, Integer, String
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker
import json

Base = declarative_base()

# است (ذخیره به عنوان رشته) JSON مدل که شامل فیلد:
class User(Base):
    __tablename__ = 'users'

    id = Column(Integer, primary_key=True)
    name = Column(String, nullable=False)
    preferences = Column(String) # ذخیره به عنوان رشته JSON

# اتصال به پایگاه داده SQLite
engine = create_engine('sqlite:///users.db')
Base.metadata.create_all(engine)

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

# به پایگاه داده JSON اضافه کردن داده‌های
preferences_data = json.dumps({"theme": "light", "notifications": False})
new_user = User(name='Alice', preferences=preferences_data)
session.add(new_user)
session.commit()

# از پایگاه داده JSON خواندن داده‌های
user = session.query(User).first()
preferences = json.loads(user.preferences)
print(preferences)
```

در این مثال:

- فیلد `preferences` به عنوان رشته ذخیره می‌شود که داده‌های JSON را نگهداری می‌کند.
- هنگام خواندن داده‌ها از پایگاه داده، از `json.loads()` برای تبدیل رشته به دیکشنری استفاده می‌شود.

MySQL 1.3

در MySQL نسخه‌های جدید، می‌توان از نوع داده `JSON` استفاده کرد. در SQLAlchemy، این نوع داده مانند PostgreSQL تعریف می‌شود.

```
from sqlalchemy import create_engine, Column, Integer, String, JSON
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker
```

```
Base = declarative_base()
```

است JSON مدل که شامل فیلد #

```
class User(Base):
```

```
    __tablename__ = 'users'
```

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

```
    name = Column(String, nullable=False)
```

```
    preferences = Column(JSON)
```

اتصال به پایگاه داده MySQL

```
engine = create_engine('mysql+mysqlconnector://username:password@localhost/mydatabase')
```

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

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

```
session = Session()
```

به پایگاه داده JSON اضافه کردن داده‌های

```
new_user = User(name='David', preferences={"theme": "dark", "language": "en"})
```

```
session.add(new_user)
```

```
session.commit()
```

در این مثال:

- فیلد preferences به طور مستقیم به عنوان JSON در MySQL ذخیره می‌شود.

2. عملیات با داده‌های JSON

پس از اینکه داده‌های JSON را در پایگاه داده ذخیره کردید، می‌توانید عملیات مختلفی روی آنها انجام دهید.

2.1 خواندن داده‌های JSON

برای خواندن داده‌های JSON از پایگاه داده، فقط کافی است فیلد JSON را بازیابی کنید و سپس آن را تجزیه کنید (در صورت لزوم).

```
# از پایگاه داده JSON خواندن داده‌های
```

```
user = session.query(User).filter_by(name='Alice').first()
```

```
preferences = json.loads(user.preferences)
```

```
print(preferences) # {"theme": "light", "notifications": False}
```

2.2 جستجو و فیلتر کردن داده‌های JSON

شما می‌توانید از شرایط خاص برای جستجو در داده‌های JSON استفاده کنید، به ویژه در پایگاه داده‌هایی مثل که از قابلیت‌های پیشرفته JSON پشتیبانی می‌کنند.

مثال برای PostgreSQL :

```
# جستجو بر اساس مقدار در JSON
```

```
user = session.query(User).filter(User.preferences['theme'].astext == 'dark').first()
```

```
print(user.name) # "John"
```

در اینجا از قابلیت‌های PostgreSQL در JSON برای فیلتر کردن بر اساس مقادیر داخل JSON استفاده شده است.

2.3 به روزرسانی داده‌های JSON

شما می‌توانید داده‌های JSON را به روزرسانی کنید. برای این کار، ابتدا باید داده‌های موجود را با رگذاری کرده و سپس آنها را ویرایش کنید.

```
# به روزرسانی داده‌های JSON
user = session.query(User).filter_by(name='John').first()
preferences = json.loads(user.preferences)
preferences["theme"] = "light"
user.preferences = json.dumps(preferences)
session.commit()
```

3. داده‌های پیچیده (Complex Data Types)

از انواع مختلف داده‌های پیچیده مانند `ARRAY`, `JSON`, `PickleType` و غیره پشتیبانی می‌کند.

PickleType 3.1

اگر نیاز دارید داده‌های پیچیده‌تر (مثل کلاس‌های Python) را ذخیره کنید، می‌توانید از `PickleType` استفاده کنید. این نوع داده به شما اجازه می‌دهد تا اشیاء Python را به صورت باینری ذخیره کنید.

```
from sqlalchemy import create_engine, Column, Integer, String
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker
from sqlalchemy.types import PickleType

Base = declarative_base()

class User(Base):
    __tablename__ = 'users'

    id = Column(Integer, primary_key=True)
    name = Column(String)
    data = Column(PickleType)

# اتصال به پایگاه داده
engine = create_engine('sqlite:///example.db')
Base.metadata.create_all(engine)

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

# ذخیره داده‌های پیچیده
new_user = User(name="John", data={"key": "value", "nested": {"a": 1}})
session.add(new_user)
session.commit()

# بازیابی داده‌های پیچیده
user = session.query(User).first()
print(user.data) # {'key': 'value', 'nested': {'a': 1}}
```

SQLAlchemy قابلیت‌های بسیاری برای کار با داده‌های JSON و داده‌های پیچیده دارد. این ویژگی‌ها به شما این امکان را می‌دهند که داده‌های مختلف را به صورت موثر و کارا در پایگاه داده‌ها ذخیره و بازیابی کنید. این ابزارها به‌ویژه در پروژه‌هایی که نیاز به ذخیره داده‌های پویا یا پیچیده دارند، بسیار مفید هستند.

استفاده از دستورات RAW SQL در SQLAlchemy

در SQLAlchemy، شما می‌توانید علاوه بر استفاده از ORM (Object-Relational Mapping) برای تعامل با پایگاه داده، از دستورات RAW SQL نیز برای انجام عملیات پایگاه داده استفاده کنید. این روش به شما امکان می‌دهد تا از دستورات SQL خام (خام به معنای بدون پردازش ORM) برای انجام عملیات مستقیم بر روی پایگاه داده استفاده کنید.

استفاده از دستورات RAW SQL می‌تواند در مواقعی که نیاز دارید عملکرد خاصی را به صورت مستقیم با SQL پیاده‌سازی کنید یا زمانی که نیاز به تعامل با پایگاه داده‌های پیچیده‌تر دارید مفید باشد.

1. اجرای دستورات SELECT با RAW SQL

با استفاده از RAW SQL، می‌توانید دستورات SELECT را به صورت مستقیم با SQL خام اجرا کنید. برای این کار از استفاده می‌شود. متد `() execute`

مثال:

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

# اتصال به پایگاه داده
engine = create_engine('sqlite:///example.db')
Session = sessionmaker(bind=engine)
session = Session()

# اجرای دستورات RAW SQL
result = session.execute("SELECT * FROM users WHERE age > :age", {"age": 25})

# پردازش نتایج
for row in result:
    print(row)
```

در این مثال:

- از دستور RAW SQL به صورت SELECT برای گرفتن کاربران با سن بیشتر از ۲۵ استفاده می‌شود.
- از پارامتر `age` برای جلوگیری از حملات SQL Injection و استفاده از متغیرهای Python در دستور SQL استفاده شده است.

2. اجرای دستورات RAW SQL با INSERT

برای درج داده‌ها در پایگاه داده، می‌توانید از دستور `INSERT INTO` به صورت مستقیم با استفاده از RAW SQL استفاده کنید.

:مثال

```
# اجرای INSERT با RAW SQL
session.execute(
    "INSERT INTO users (name, age) VALUES (:name, :age)",
    {"name": "Alice", "age": 30}
)

# ذخیره تغییرات
session.commit()
```

در این مثال:

- از دستور `INSERT INTO` برای افزودن کاربر جدید استفاده می‌شود.
- متغیرهای `age` و `name` از پارامترهای Python در دستور SQL استفاده می‌کنند.

3. اجرای دستورات RAW SQL با UPDATE

برای بهروزرسانی داده‌ها در پایگاه داده، می‌توانید از دستور `UPDATE` با استفاده از SQL خام استفاده کنید.

:مثال

```
# اجرای UPDATE با RAW SQL
session.execute(
    "UPDATE users SET age = :age WHERE name = :name",
    {"age": 35, "name": "Alice"}
)

# ذخیره تغییرات
session.commit()
```

در این مثال:

- از دستور `UPDATE` برای بهروزرسانی سن کاربر با نام `Alice` استفاده شده است.

4. اجرای دستورات RAW SQL با DELETE

برای حذف داده‌ها از پایگاه داده، می‌توانید از دستور `DELETE` به صورت مستقیم با SQL خام استفاده کنید.

مثال:

```
# اجرای DELETE با RAW SQL
session.execute(
    "DELETE FROM users WHERE name = :name",
    {"name": "Alice"}
)

# ذخیره تغییرات
session.commit()
```

در این مثال:

- از دستور `DELETE` برای حذف کاربری با نام `Alice` استفاده شده است.

5. دریافت نتایج به صورت متفاوت

برای دریافت نتایج اجرای دستور SQL، می‌توانید از متدهای مختلفی مانند `fetchall()`, `fetchone()`، یا `scalar()` استفاده کنید.

(`fetchall`) 5.1

برای دریافت همه نتایج:

```
result = session.execute("SELECT * FROM users WHERE age > :age", {"age": 25})
rows = result.fetchall()
for row in rows:
    print(row)
```

(`fetchone`) 5.2

برای دریافت یک نتیجه واحد:

```
result = session.execute("SELECT * FROM users WHERE name = :name", {"name": "Alice"})
row = result.fetchone()
print(row)
```

(`scalar`) 5.3

برای دریافت یک مقدار واحد (مانند مجموع یا تعداد):

```
result = session.execute("SELECT COUNT(*) FROM users WHERE age > :age", {"age": 25})
count = result.scalar()
print(count)
```

6. اجرای تراکنش‌های RAW SQL

اگر در حال استفاده از دستورات SQL هستید و می‌خواهید تراکنش‌ها را مدیریت کنید، می‌توانید از متدهای استفاده کنید. `() rollback`, `() begin()`, `commit`

مثال:

```
# شروع تراکنش
with session.begin():
    session.execute(
        "UPDATE users SET age = :age WHERE name = :name",
        {"age": 40, "name": "Bob"})
    )
# در صورت لزوم می‌توانید خطاهایی را در اینجا مدیریت کنید و تراکنش را لغو کنید
```

در این مثال:

- از `() session.begin` برای شروع یک تراکنش استفاده شده است.
- تغییرات فقط زمانی ذخیره می‌شوند که `() commit` فراخوانی شود، و در صورت وجود خطا، تراکنش برگشت می‌کند.

SQL Injection و SQLAlchemy .7

هنگام استفاده از دستورات RAW SQL در SQLAlchemy، بسیار مهم است که از **SQL Injection** جلوگیری کنید. برای جلوگیری از این حملات، همیشه از پارامترهای پیوست شده (Bound Parameters) مانند `param:` در دستور `session.execute` استفاده کنید.

به عنوان مثال، به جای نوشتن مستقیم داده‌ها در دستور SQL:

```
# خطناک و مستعد SQL Injection
session.execute("SELECT * FROM users WHERE name = '{user_name}'")
```

از پارامترهای پیوست شده استفاده کنید:

```
# امن تر
session.execute("SELECT * FROM users WHERE name = :name", {"name": user_name})
```

جمع‌بندی

استفاده از دستورات RAW SQL در SQLAlchemy به شما این امکان را می‌دهد که عملیات پیچیده‌تر و خاص‌تری را با SQL انجام دهید، اما به یاد داشته باشید که استفاده از دستورات SQL به صورت خام ممکن است پیچیدگی‌هایی ایجاد کند و در صورت عدم دقیقت به مسائل امنیتی مثل **SQL Injection** باید مراقب باشید. SQLAlchemy این امکان را فراهم می‌کند که با استفاده از پارامترهای ایمن، دستورات SQL را به طور کارا و امن اجرا کنید.

استفاده از Joinها و ارتباطات پیچیده در SQLAlchemy

در SQLAlchemy، شما می‌توانید از Joinها برای ارتباط بین جداول مختلف استفاده کنید و داده‌ها را به‌طور پیچیده‌تر از طریق این ارتباطات به دست آورید. این کار به شما این امکان را می‌دهد که از داده‌های جداول مختلف با هم ترکیب کنید و نتایج جستجو را با استفاده از روابط مختلف بین جداول استخراج نمایید.

از دو روش عمدۀ برای انجام Join استفاده می‌کند:

1. ORM Join‌های: از روابط تعریف شده در مدل‌ها استفاده می‌شود.
2. SQL Join‌های: استفاده از دستورات `join()` به صورت مستقیم در SQLAlchemy خام.

(One-to-Many / Many-to-One) Join‌های ساده .1

اگر دو جدول با رابطه‌ی یک به چند (One-to-Many) یا چند به یک (Many-to-One) مرتبه باشند، می‌توان از دستور `join()` برای ارتباط میان آنها استفاده کرد.

مثال: یک به چند (One-to-Many)

فرض کنید دو جدول داریم: `Post` و `User`. یک کاربر می‌تواند چندین پست داشته باشد (یک به چند).

```
from sqlalchemy import create_engine, Column, Integer, String, ForeignKey
from sqlalchemy.orm import relationship, sessionmaker
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

# تعریف مدل User
class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    name = Column(String)

    posts = relationship('Post', back_populates='user')

# تعریف مدل Post
class Post(Base):
    __tablename__ = 'posts'
    id = Column(Integer, primary_key=True)
    title = Column(String)
    user_id = Column(Integer, ForeignKey('users.id'))

    user = relationship('User', back_populates='posts')

# اتصال به پایگاه داده و ساخت سشن
engine = create_engine('sqlite:///example.db')
Base.metadata.create_all(engine)
Session = sessionmaker(bind=engine)
session = Session()

# بین دو جدول Join عملیات
result = session.query(User.name, Post.title).join(Post).filter(User.name == 'Alice').all()

for row in result:
    print(row)
```

در این مثال:

- دو مدل `User` و `Post` با یکدیگر ارتباط دارند.
- از متدهای `join()` برای پیوستن داده‌ها از هر دو جدول استفاده می‌شود.
- از `filter()` برای جستجوی کاربران با نام خاص استفاده می‌شود.

(Many-to-Many) Join‌های پیچیده .2

اگر دو جدول با رابطه‌ی چند به چند (Many-to-Many) مرتبط باشند، باید از یک جدول واسطه (Association Table) استفاده کنید که ارتباط میان دو جدول را تعریف کند.

مثال: چند به چند (Many-to-Many)

فرض کنید دو جدول داریم: `Course` و `Student`. هر دانشآموز می‌تواند در چندین دوره ثبت‌نام کند و هر دوره می‌تواند دانشآموزان مختلفی داشته باشد.

```
from sqlalchemy import Table, create_engine, Column, Integer, String, ForeignKey
from sqlalchemy.orm import relationship, sessionmaker
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

# جدول واسطه برای ارتباط Many-to-Many
student_courses = Table('student_courses', Base.metadata,
    Column('student_id', Integer, ForeignKey('students.id')),
    Column('course_id', Integer, ForeignKey('courses.id'))
)

# تعریف مدل Student
class Student(Base):
    __tablename__ = 'students'
    id = Column(Integer, primary_key=True)
    name = Column(String)

    courses = relationship('Course', secondary=student_courses, back_populates='students')

# تعریف مدل Course
class Course(Base):
    __tablename__ = 'courses'
    id = Column(Integer, primary_key=True)
    name = Column(String)

    students = relationship('Student', secondary=student_courses, back_populates='courses')

# اتصال به پایگاه داده و ساخت سشن
engine = create_engine('sqlite:///example.db')
Base.metadata.create_all(engine)
Session = sessionmaker(bind=engine)
session = Session()

# پیچیده برای دریافت دانشآموزان در یک دوره خاص Join انجام عملیات
result = session.query(Student.name, Course.name).join(student_courses).join(Course).filter(Course.name == 'Mathematics').all()
```

```
for row in result:  
    print(row)
```

در این مثال:

- از یک جدول واسط Course و Student برای ایجاد ارتباط Many-to-Many بین student_courses برای پیوستن داده‌ها از هر دو جدول و جدول واسط استفاده شده است.
- از متدهای join() برای پیوستن داده‌ها از هر دو جدول و جدول واسط استفاده می‌شود.

3. Join‌های پیچیده‌تر با شرایط

شما می‌توانید شرایط خاص را برای Join‌ها به‌طور مستقیم در SQLAlchemy اعمال کنید، مانند اضافه کردن شرط‌های مختلف یا مرتب‌سازی نتایج.

مثال: استفاده از شرایط پیچیده در Join

```
# پیچیده با شرایط مختلف Join انجام  
result = session.query(User.name, Post.title).join(Post).filter(User.name == 'Alice',  
Post.title.like('%Python%')).all()  
  
for row in result:  
    print(row)
```

در این مثال:

- از filter() برای اعمال چندین شرط به‌طور همزمان استفاده شده است.
- پست‌هایی که عنوان آن‌ها شامل کلمه "Python" است، برای کاربر "Alice" جستجو می‌شود.

4. Right Join و Left Join

گاهی اوقات شما نیاز دارید که از Right Join یا Left Join استفاده کنید، که نتایج را به‌گونه‌ای برمی‌گرداند که تمام داده‌ها از یک جدول (مثلاً جدول چپ) در نتایج ظاهر شود حتی اگر در جدول دیگر داده‌ای وجود نداشته باشد.

Left Join: مثال

```
# اجرای Left Join  
result = session.query(User.name, Post.title).outerjoin(Post).all()  
  
for row in result:  
    print(row)
```

در این مثال:

- از outerjoin() برای اجرای Left Join استفاده می‌شود.
- در نتیجه، حتی اگر یک کاربر پست نداشته باشد، همچنان نام کاربر نمایش داده می‌شود.

۵. Join‌های پیچیده با Subquery

در بعضی موارد شما نیاز دارید که از Subquery (زیرپرس و جو) برای انجام عملیات پیچیده‌تر در Join استفاده کنید. به شما این امکان را می‌دهد که از زیرپرس‌وجوها برای انجام عملیات پیشرفته بھر ببرید.

مثال: Subquery با Join

```
# پیچیده Join برای استفاده از Subquery
subquery = session.query(Post.user_id, Post.title).filter(Post.title.like('%Python%')).subquery()

result = session.query(User.name, subquery.c.title).join(subquery, subquery.c.user_id == User.id).all()

for row in result:
    print(row)
```

در این مثال:

- ابتدا یک زیرپرس‌جو (Subquery) برای گرفتن پست‌هایی که عنوان آن‌ها شامل "Python" است ایجاد می‌شود.
- سپس این زیرپرس‌جو با جدول `User` پیوند داده می‌شود.

جمع‌بندی

استفاده از Join‌ها در SQLAlchemy به شما این امکان را می‌دهد که از ارتباطات مختلف بین جداول برای انجام جستجوهای پیچیده‌تر استفاده کنید. همچنین با استفاده از Left Join، Right Join، Subquery و شرایط مختلف، می‌توانید نتایج دقیق و پیچیده‌تری را از پایگاه داده استخراج کنید.

استفاده از متدهای ویژه برای بهینه‌سازی پرس‌وجوها در SQLAlchemy

یک ORM به شما این امکان را می‌دهد که با داده‌های پایگاه داده تعامل داشته باشید و به طور کارا و بهینه پرس‌وجوها را اجرا کنید. بهینه‌سازی پرس‌وجوها می‌تواند تأثیر زیادی بر عملکرد برنامه‌های شما بگذارد، بهویژه وقتی با داده‌های بزرگ یا پیچیده کار می‌کنید.

در SQLAlchemy، ابزارها و متدهای ویژه‌ای برای بهینه‌سازی پرس‌وجوها وجود دارد. این ابزارها شامل lazy loading، eager loading، selectin loading، indexed queries، query filtering

Eager Loading و Lazy Loading .1

در SQLAlchemy، می‌توانید نحوه بارگذاری داده‌های مرتبط بین جداول را تنظیم کنید. دو روش اصلی برای بارگذاری داده‌ها وجود دارد:

- داده‌ها فقط زمانی بارگذاری می‌شوند که به آن‌ها نیاز داشته باشید. **Lazy Loading**
- داده‌ها به طور همزمان با داده‌های اصلی بارگذاری می‌شوند. **Eager Loading**

Lazy Loading

در Lazy Loading، ارتباطات بین جداول به طور خودکار بارگذاری نمی‌شوند و فقط زمانی که به آن‌ها نیاز پیدا می‌کنید، بارگذاری می‌شوند.

```
# Lazy Loading (پیش‌فرض)
class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    name = Column(String)

    posts = relationship('Post', back_populates='user', lazy='select')
```

پس از فراخوانی داده‌های کاربر، پست‌ها فقط زمانی بارگذاری می‌شوند که به آن‌ها دسترسی پیدا کنید #
user = session.query(User).first()
print(user.posts) # پست‌ها به‌طور خودکار بارگذاری خواهند شد.

Eager Loading

در Eager Loading، داده‌های مرتبط با داده‌های اصلی به‌طور همزمان بارگذاری می‌شوند تا از درخواست‌های اضافی جلوگیری شود.

```
# Eager Loading با استفاده از joinedload
from sqlalchemy.orm import joinedload

# بارگذاری پست‌ها همراه با بارگذاری کاربر
user = session.query(User).options(joinedload(User.posts)).first()
print(user.posts) # پست‌ها همزمان با بارگذاری کاربر بارگذاری می‌شوند.
```

در اینجا از `joinedload()` برای Eager Loading استفاده شده است. به این معنی که داده‌های پست‌ها به‌طور همزمان با داده‌های کاربر بارگذاری می‌شوند.

2. selectinload برای بهینه‌سازی تعداد درخواست‌ها

در برخی مواقع، اگر تعداد زیادی ارتباط یک به چند یا چند بین جداول داشته باشد، استفاده از `selectinload()` از چندین درخواست بهینه استفاده می‌کند.

```
from sqlalchemy.orm import selectinload

# بارگذاری پست‌ها به‌طور بهینه
user = session.query(User).options(selectinload(User.posts)).first()
print(user.posts) # پست‌ها به‌طور بهینه بارگذاری می‌شوند.
```

به‌طور کلی برای بارگذاری مجموعه‌های بزرگ‌تر از داده‌ها مفید است، زیرا SQLAlchemy برای هر مجموعه از داده‌ها یک درخواست جداگانه ارسال می‌کند که می‌تواند عملکرد را بهبود بخشد.

3. Limit و Offset برای کاهش تعداد رکوردها

برای محدود کردن تعداد رکوردهایی که از پایگاه داده می‌خوانید، می‌توانید از `limit()` و `offset()` استفاده کنید. این کار برای جلوگیری از بارگذاری داده‌های غیرضروری مفید است.

```
# برای محدود کردن نتایج offset و limit استفاده از
result = session.query(User).limit(10).offset(20).all()
```

در این مثال:

- فقط ۱۰ رکورد اول را برمی‌گرداند.
- باعث می‌شود که نتایج از رکورد ۲۱ شروع شود.

4. ایجاد ایندکس‌ها برای جستجوهای سریع‌تر

برای بهینه‌سازی جستجوها در پایگاه داده، می‌توانید از ایندکس‌ها استفاده کنید. ایندکس‌ها به جستجو در پایگاه داده سرعت می‌بخشند، به ویژه زمانی که در جداول بزرگ جستجو می‌کنید.

تعریف ایندکس در SQLAlchemy

```
from sqlalchemy import Index

class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    name = Column(String, index=True) # ایجاد ایندکس روی ستون name

# برای ایجاد ایندکس پیچیده‌تر Index یا استفاده از کلاس
Index('my_index', User.name, User.id)
```

در این مثال، ایندکسی روی ستون `name` ایجاد می‌شود تا جستجوها در این ستون سریع‌تر انجام شوند.

5. استفاده از `exists()` برای بررسی وجود داده‌ها

در برخی مواقع نیاز دارید که فقط بررسی کنید که آیا داده‌ای وجود دارد یا خیر. استفاده از `exists()` می‌تواند این عملیات را بهینه کند.

```
from sqlalchemy import exists

# بررسی اینکه آیا کاربری با نام خاص وجود دارد یا خیر
exists_query = session.query(exists().where(User.name == 'Alice')).scalar()

if exists_query:
    print("کاربر وجود دارد")
else:
    print("کاربر وجود ندارد")
```

در اینجا، از `exists()` برای بررسی وجود رکورد استفاده شده است که از نظر عملکرد بهینه‌تر از اجرای یک جستجوی کامل است.

6. استفاده از `(()with_for_update()` برای قفل کردن رکوردها

اگر نیاز به قفل کردن رکوردها دارید (مثلاً برای جلوگیری از دستکاری همزمان داده‌ها)، می‌توانید از `(()with_for_update()` استفاده کنید تا رکوردها قفل شوند.

قفل کردن رکوردها برای جلوگیری از تغییرات هم‌زمان #

```
user = session.query(User).filter(User.name == 'Alice').with_for_update().first()
```

در اینجا، رکورد مربوط به کاربر `Alice` قفل می‌شود و دیگر تراکنش‌ها نمی‌توانند آن را تغییر دهند تا تراکنش جاری تکمیل شود.

7. استفاده از `filter()` به جای `all()` برای کاهش حجم داده‌ها

گاهی اوقات شما فقط نیاز به یک یا چند رکورد خاص دارید. استفاده از `filter()` می‌تواند عملکرد را بهبود بخشد زیرا نیازی به خواندن تمام رکوردها نیست.

به جای filter() استفاده از all()

```
user = session.query(User).filter(User.name == 'Alice').first()
```

در این مثال، به جای خواندن همه رکوردها با `filter()` استفاده شده تا فقط رکوردهای خاصی که با شرایط مطابقت دارند بازگردانده شوند.

8. استفاده از `distinct()` برای حذف رکوردهای تکراری

اگر نیاز دارید که رکوردهای تکراری را از نتایج حذف کنید، می‌توانید از متدهای `distinct()` استفاده کنید.

برای حذف رکوردهای تکراری distinct() استفاده از

```
result = session.query(User.name).distinct().all()
```

در این مثال، از `distinct()` برای بازگرداندن نام‌های منحصر به فرد استفاده می‌شود.

جمع‌بندی

برای بهینه‌سازی پرس‌وپرس‌ها در SQLAlchemy می‌توانید از تکنیک‌های مختلفی مانند `Lazy Loading`، `Eager Loading`، `selectinload`، `exists()` و `distinct()` استفاده کنید. این روش‌ها می‌توانند عملکرد برنامه‌های شما را بهبود بخشیده و از بار اضافی روی پایگاه داده جلوگیری کنند.

مدیریت خطاهای و استثناهای (Advanced Error Handling) در SQLAlchemy

مدیریت صحیح خطاهای و استثناهای در هر برنامه‌ای، به ویژه برنامه‌هایی که با پایگاه داده تعامل دارند، از اهمیت ویژه‌ای برخوردار است. در SQLAlchemy نیز، امکان مدیریت دقیق و بهینه خطاهای وجود دارد. این امر به شما کمک می‌کند تا برنامه‌تان پایدارتر، قابل اعتمادتر و مقاوم‌تر در برابر مشکلات احتمالی باشد.

استفاده از SQLAlchemy از Python's built-in exception handling system استفاده می‌کند و علاوه بر آن، خطاها خاص پایگاه داده و ORM را نیز ایجاد می‌کند. در این بخش، به بررسی انواع مختلف خطاها و نحوه مدیریت آن‌ها در SQLAlchemy می‌پردازیم.

1. استفاده از Try-Except

در ابتدایی‌ترین حالت، می‌توانید از بلاک `try-except` برای مدیریت خطاها و استثناهای پایگاه داده استفاده کنید.

```
from sqlalchemy.exc import SQLAlchemyError

try:
    # انجام عملیات پایگاه داده
    user = session.query(User).filter_by(name='Alice').first()
    print(user.name)
except SQLAlchemyError as e:
    print(f"خطا در هنگام اجرای کوئری {e}")
```

در این مثال:

- يك استثنای عمومي است که تمام خطاهاي مرتبط با SQLAlchemy را پوشش مي‌دهد.
- در صورتی که خطای در هنگام اجرای کوئری رخ دهد، پیام خطا چاپ می‌شود.

2. استفاده از خطاهاي خاص پایگاه داده

تعدادی از خطاهاي خاص را نيز به طور پيش‌فرض ارائه مي‌دهد. اين خطاها می‌توانند به طور دقیق‌تری به شناسایی و رفع مشکلات کمک کنند.

2.1 IntegrityError

این خطا زمانی رخ می‌دهد که نقض‌های انسجام داده (مانند اضافه کردن رکوردی که نقض‌کننده قوانین `UNIQUE` یا `NOT NULL` است) اتفاق بیفتد.

```
from sqlalchemy.exc import IntegrityError

try:
    # تلاش برای اضافه کردن رکوردی که نقض‌کننده محدودیت‌های پایگاه داده باشد
    new_user = User(name=None) # نمی‌تواند 'name' ستون None باشد
    session.add(new_user)
    session.commit()
except IntegrityError as e:
    print(f"خطای انسجام داده {e}")
    session.rollback() # برای برگشت تغییرات
```

2.2 OperationalError

زمانی که یک خطای عملیاتی در پایگاه داده رخ می‌دهد (مثلًا اتصال به پایگاه داده برقرار نشود یا دستورات SQL به درستی اجرا نشوند)، `OperationalError` فعال می‌شود.

```

from sqlalchemy.exc import OperationalError

try:
    # تلاش برای اتصال به پایگاه داده غیرمجاز یا قطع ارتباط
    session.query(User).all()
except OperationalError as e:
    print(f"خطای عملیاتی: {e}")
    # برگشت تراکنش برای جلوگیری از مشکلات بعدی
    session.rollback()

```

ProgrammingError .2.3

این خطای زمانی رخ می‌دهد که یک مشکل در نحوه نوشتن کوئری SQL وجود داشته باشد. به عنوان مثال، زمانی که ستون یا جدول نادرستی استفاده شود.

```

from sqlalchemy.exc import ProgrammingError

try:
    # کوئری اشتباه که شامل یک ستون نادرست بود
    session.query(User.non_existent_column).all()
except ProgrammingError as e:
    print(f"خطای برنامه‌نویسی: {e}")
    # برگشت تراکنش
    session.rollback()

```

3. مدیریت خطاها در تراکنش‌ها

در SQLAlchemy، شما می‌توانید از دستور `() rollback` برای برگشت یک تراکنش استفاده کنید تا تغییرات ناتمام یا نادرست به پایگاه داده اعمال نشوند.

```

from sqlalchemy.exc import SQLAlchemyError

try:
    # شروع تراکنش
    session.begin()
    # انجام چند عملیات پایگاه داده
    new_user = User(name='Bob')
    session.add(new_user)
    # تایید تغییرات
    session.commit()
except SQLAlchemyError as e:
    print(f"خطا در هنگام تراکنش: {e}")
    # برگشت تغییرات در صورت خطا
    session.rollback()

```

در این مثال:

- اگر خطایی در هر کجای تراکنش رخ دهد، `() rollback` فراخوانی می‌شود تا تغییرات اعمال شده به پایگاه داده برگشت داده شوند.
- در صورت موفقیت‌آمیز بودن تراکنش، `() commit` برای تایید تغییرات به پایگاه داده استفاده می‌شود.

4. استفاده از `() flush` و `() commit` در مدیریت خطاهای SQLAlCHEMY

در برای ارسال تغییرات به پایگاه داده بدون تایید نهایی (`commit`) استفاده می‌شود. اگر عملیات‌هایی در `() flush` به خطأ برخورد کنند، می‌توانید با استفاده از `() commit` تغییرات نهایی را تایید کنید یا آنها را بازگردانید.

```
try:  
    # تلاش برای انجام عملیات‌های پایگاه داده  
    user = User(name='Alice')  
    session.add(user)  
    session.flush() # تغییرات ارسال می‌شود، اما تایید نمی‌شود  
    # در اینجا می‌توانید سایر عملیات‌ها را انجام دهید  
    session.commit() # تایید نهایی  
  
except SQLAlchemyError as e:  
    print(f"خطا در ارسال تغییرات: {e}")  
    session.rollback() # برگشت تغییرات در صورت خطأ
```

5. خطاهای ناشی از تداخل تراکنش‌ها (Concurrency Errors)

اگر چندین تراکنش به‌طور همزمان در حال اجرا باشند و تغییرات در یک رکورد به‌صورت همزمان توسط چندین کاربر با فرآیند انجام شود، این می‌تواند منجر به خطاهای تداخل تراکنش شود. SQLAlchemy از مکانیزم‌هایی مانند **Optimistic Concurrency Control (OCC)** برای مدیریت این نوع مشکلات استفاده می‌کند.

برای مثال، شما می‌توانید از `() with_for_update` برای قفل کردن رکوردها در حین تغییرات همزمان استفاده کنید.

```
# قفل کردن رکوردها برای جلوگیری از تداخل تراکنش‌ها  
user = session.query(User).filter(User.id == 1).with_for_update().first()  
user.name = 'New Name'  
session.commit()
```

این عمل از تغییرات همزمان و تداخل تراکنش‌ها جلوگیری می‌کند و اطمینان حاصل می‌کند که رکوردهای قفل شده نمی‌توانند توسط سایر تراکنش‌ها تغییر یابند.

6. استفاده از Logging برای پیگیری خطاهای SQLAlCHEMY

برای بهبود شفافیت و پیگیری دقیق‌تر خطاهای SQLAlCHEMY، می‌توانید از `logging` برای ثبت و پیگیری استثناهای استفاده کنید.

```
import logging  
  
# تنظیمات logging  
logging.basicConfig(level=logging.ERROR)  
  
try:  
    # تلاش برای انجام عملیات پایگاه داده  
    user = session.query(User).filter_by(name='Alice').first()  
    print(user.name)  
  
except SQLAlchemyError as e:  
    logging.error(f"خطا در هنگام اجرای کوئری: {e}")  
    session.rollback()
```

در این مثال، خطاهای SQLAlCHEMY با استفاده از `() logging.error` ثبت می‌شوند.

جمع‌بندی

مدیریت خطاهای و استثنایها در SQLAlchemy بخش مهمی از هر اپلیکیشن است که با پایگاه داده تعامل دارد. از آنجا که خطاهای مختلفی در پایگاه داده‌ها و عملیات‌های ORM می‌توانند رخ دهند، استفاده از استثنای‌های خاص SQLAlchemy، مدیریت تراکنش‌ها، قفل کردن رکوردها و استفاده از `commit()` و `rollback()` به شما کمک می‌کند تا از بروز مشکلات جلوگیری کنید. همچنین، با استفاده از `logging` می‌توانید خطاهای را پیگیری و بررسی کنید تا عملکرد برنامه‌تان بهتر شود.

تنظیمات پیشرفتی Connection Pooling

فرآیند ذخیره‌سازی اتصالات پایگاه داده است تا به‌طور مجدد از آن‌ها در درخواست‌های بعدی استفاده شود، بدون اینکه هر بار اتصال جدیدی به پایگاه داده برقرار گردد. این کار باعث بهبود عملکرد و کاهش تأخیر در برنامه‌هایی می‌شود که نیاز به دسترسی مکرر به پایگاه داده دارند.

به‌طور پیش‌فرض از `connection pooling` پشتیبانی می‌کند و از پیکربندی‌های پیشرفتی برای تنظیم اتصالات و مدیریت منابع استفاده می‌کند. این بخش به توضیح تنظیمات پیشرفتی `Pooling` و نحوه پیکربندی آن‌ها در SQLAlchemy می‌پردازد.

1. اصول اولیه Connection Pooling

به‌طور پیش‌فرض از `QueuePool` برای مدیریت اتصالات استفاده می‌کند. این Pool به‌طور خودکار تعداد معینی از اتصالات را ایجاد کرده و آن‌ها را در حافظه ذخیره می‌کند. سپس، هنگام نیاز به اتصال، یکی از اتصالات موجود از Pool بازیابی می‌شود. پس از اتمام کار، اتصال به Pool بازمی‌گردد.

```
from sqlalchemy import create_engine
# پیش‌فرض connection pooling با استفاده از engine ایجاد یک
engine = create_engine('postgresql://user:password@localhost/mydatabase')

استفاده می‌کند SQLAlchemy Pooling از اینجا #
```

2. پیکربندی Pooling پیشرفتی

SQLAlchemy به شما اجازه می‌دهد که با استفاده از پارامترهای مختلف، نحوه مدیریت اتصالات را به دلخواه تنظیم کنید.

2.1. تعداد اتصالات (pool_size)

این تنظیم تعداد اتصالات موجود در Pool را مشخص می‌کند. هرچه این مقدار بیشتر باشد، بیشتر می‌توانید از اتصالات همزمان استفاده کنید.

```
engine = create_engine(  
    'postgresql://user:password@localhost/mydatabase',  
    pool_size=10 # تعداد اتصالات Pool  
)
```

در اینجا `pool_size=10` به این معنی است که حداقل ۱۰ اتصال به پایگاه داده به طور هم‌زمان باز خواهد بود.

2.2. حداقل اندازه اتصالات در (max_overflow)

این پارامتر حداقل تعداد اتصالات اضافی را که می‌توانند به Pool اضافه شوند، تعیین می‌کند. در صورتی که اتصالات موجود به پایان برسند، این تعداد اتصالات اضافی ایجاد می‌شود تا بار اضافی مدیریت شود.

```
engine = create_engine(  
    'postgresql://user:password@localhost/mydatabase',  
    pool_size=5,  
    max_overflow=10 # تعداد اتصالات اضافی که می‌توانند ایجاد شوند  
)
```

در اینجا `max_overflow=10` به این معنی است که می‌توان تا ۱۰ اتصال اضافی به Pool اضافه کرد.

2.3. زمان مجاز نگهداری اتصالات (pool_timeout)

این پارامتر مشخص می‌کند که یک درخواست اتصال برای چند ثانیه می‌تواند منتظر بماند تا یک اتصال موجود از Pool بازیابی شود. اگر هیچ اتصالی در دسترس نباشد، درخواست تا زمان معین شده منتظر خواهد ماند.

```
engine = create_engine(  
    'postgresql://user:password@localhost/mydatabase',  
    pool_size=5,  
    max_overflow=10,  
    pool_timeout=30 # زمان انتظار برای اتصالات (بر حسب ثانیه)  
)
```

در اینجا `pool_timeout=30` به این معنی است که اگر اتصالی موجود نباشد، درخواست می‌تواند تا ۳۰ ثانیه منتظر بماند.

2.4. زمان زوال اتصالات (pool_recycle)

اتصالات به طور پیش‌فرض برای همیشه در Pool باقی نمی‌مانند. در صورتی که اتصال بیش از حد در دسترس باشد و نیاز به بازنگشتن داشته باشد، از `pool_recycle` استفاده می‌شود. این پارامتر تعیین می‌کند که پس از گذشت چند ثانیه، اتصال‌ها باید بازنگشتن شوند.

```
engine = create_engine(  
    'postgresql://user:password@localhost/mydatabase',  
    pool_recycle=3600 # زمان بازنگشتن اتصالات (بر حسب ثانیه)  
)
```

در اینجا `pool_recycle=3600` به این معنی است که پس از ۱ ساعت (۳۶۰۰ ثانیه) اتصالات موجود در Pool بازنگشتن می‌شوند.

2.5. مقدار اولیه اتصالات (pool_pre_ping)

اگر شما از دیتابیس‌های استفاده می‌کنید که پس از مدت زمانی بدون استفاده قطع می‌شوند، می‌توانید از پارامتر استفاده کنید تا SQLAlchemy قبل از استفاده از یک اتصال موجود، بررسی کند که آیا اتصال `pool_pre_ping` هنوز فعال است یا خیر.

```
engine = create_engine(  
    'postgresql://user:password@localhost/mydatabase',  
    pool_pre_ping=True # بررسی سلامت اتصالات قبل از استفاده
```

در اینجا `pool_pre_ping=True` به این معنی است که قبل از استفاده از هر اتصال، صحت آن بررسی می‌شود.

3. استفاده از "Singleton" Pooling

یکی از ویژگی‌های Pooling در SQLAlchemy است که شما می‌توانید از یک اتصال `global` یا `singleton` برای تمام برنامه استفاده کنید. این به این معنی است که برای هر درخواست به پایگاه داده، تنها یک اتصال واحد ایجاد می‌شود و در بقیه زمان‌ها از همان اتصال استفاده می‌شود.

```
# استفاده از همان اتصال برای تمام اپلیکیشن  
from sqlalchemy.orm import sessionmaker  
  
Session = sessionmaker(bind=engine)  
session = Session()
```

در اینجا، هر بار که از `sessionmaker` استفاده می‌شود، از همان اتصال موجود در `engine` استفاده خواهد شد.

4. تست عملکرد Pooling

برای تست عملکرد `connection pooling`، می‌توانید از ابزارهای مختلفی مانند `SQLAlchemy's built-in` استفاده کنید. `engine.dialect`

```
# دریافت اطلاعات مربوط به Pool  
print(engine.pool.status())
```

این کد اطلاعاتی مانند تعداد اتصالات فعال، تعداد اتصالات در حال استفاده، و تعداد اتصالات در حال انتظار را نمایش می‌دهد.

5. تعیین سیاست‌های Connection Pooling برای هر

شما می‌توانید برای هر اتصال به صورت مجزا تنظیمات مختلفی برای Pooling ایجاد کنید. این کار می‌تواند زمانی مفید باشد که بخواهید برای هر نوع درخواست یک تنظیمات خاص اعمال کنید.

```
# با پیکربندی خاص Engine ایجاد
engine = create_engine(
    'postgresql://user:password@localhost/mydatabase',
    pool_size=10,
    max_overflow=5,
    pool_timeout=10
)
```

جمع‌بندی

در SQLAlchemy Connection Pooling به شما این امکان را می‌دهد که تعداد اتصالات پایگاه داده را به طور مؤثری مدیریت کنید و بار سیستم را کاهش دهید. با تنظیم پارامترهای مختلف مانند `pool_size`, `max_overflow`, می‌توانید عملکرد سیستم خود را بهینه‌سازی کرده و از اتصالات پایگاه داده به صورت مؤثر استفاده کنید.

در پروژه‌های بزرگ، تنظیمات Pooling می‌توانند تأثیر زیادی بر کارایی و پایداری سیستم داشته باشد، بنابراین توصیه می‌شود که این تنظیمات را بر اساس نیازهای خاص پروژه‌تان به دقت پیکربندی کنید.