ORM

ORM，对象关系映射，对象和关系之间的映射，使用面向对象的方式来操作数据库。

```
关系模型和Python对象之间的映射
table  => class     , 表映射为类
row    => object    , 行映射为实例
column => property  , 字段映射为属性
```

举例
有表student，字段为id int，name varchar，age int
映射到Python为

```
class Student:
    id = ?某类型字段
    name = ?某类型字段
    age = ?某类型字段

最终得到实例
class Student:
    def __init__(self):
        self.id = ?
        self.name = ?
        self.age = ?
```

SQLAlchemy

SQLAlchemy是一个ORM框架

安装

```
$ pip install sqlalchemy
```

文档

官方文档http://docs.sqlalchemy.org/en/latest/

查看版本

```
import sqlalchemy
print(sqlalchemy.__version__)
```

开发

SQLAlchemy内部使用了连接池

创建连接

数据库连接的事情，交给引擎

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

mysqldb的连接
mysql+mysqldb://<user>:<password>@<host>[:<port>]/<dbname>
engine = sqlalchemy.create_engine("mysql+mysqldb://wayne:wayne@127.0.0.1:3306/magedu")

pymysql的连接
mysql+pymysql://<username>:<password>@<host>/<dbname>[?<options>]
engine = sqlalchemy.create_engine("mysql+pymysql://wayne:wayne@127.0.0.1:3306/magedu")
engine = sqlalchemy.create_engine("mysql+pymysql://wayne:wayne@127.0.0.1:3306/magedu",
echo=True)

echo=True
引擎是否打印执行的语句，调试的时候打开很方便。

创建引擎并不会马上连接数据库，直到让数据库执行任务时才连接。
```

Declare a Mapping创建映射

创建基类

```
from sqlalchemy.ext.declarative import declarative_base
# 创建基类，便于实体类继承。SQLAlchemy大量使用了元编程
Base = declarative_base()
```

创建实体类

student表

```
CREATE TABLE student (
    id INTEGER NOT NULL AUTO_INCREMENT,
    name VARCHAR(64) NOT NULL,
    age INTEGER,
    PRIMARY KEY (id)
)
```

```
# 创建实体类
class Student(Base):
    # 指定表名
    __tablename__ = 'student'
    # 定义类属性对应字段
    id = Column(Integer, primary_key=True, autoincrement=True)
    name = Column(String(64), nullable=False)
```

```
    age = Column(Integer)
    # 第一参数是字段名，如果和属性名不一致，一定要指定
    # age = Column('age', Integer)

    def __repr__(self):
        return "{} id={} name={} age={}".format(
            self.__class__.__name__, self.id, self.name, self.age)

# 查看表结构
print(Student)
print(repr(Student.__table__))

# 显示结果
Table('student', MetaData(bind=None),
  Column('id', Integer(), table=<student>, primary_key=True, nullable=False),
  Column('name', String(length=64), table=<student>, nullable=False),
  Column('age', Integer(), table=<student>),
  schema=None)
```

`__tablename__` 指定表名
Column类指定对应的字段，必须指定

实例化

```
s = Student(name='tom')
print(s.name)
s.age = 20
print(s.age)
```

创建表

可以使用SQLAlchemy来创建、删除表

```
# 删除继承自Base的所有表
Base.metadata.drop_all(engine)
# 创建继承自Base的所有表
Base.metadata.create_all(engine)
```

生产环境很少这样创建表，都是系统上线的时候由脚本生成。
生成环境很少删除表，宁可废弃都不能删除。

创建会话session

在一个会话中操作数据库，会话建立在连接上，连接被引擎管理。
当第一次使用数据库时，从引擎维护的连接池中获取一个连接使用。

```
# 创建session
Session = sessionmaker(bind=engine) # 返回类
session = Session() # 实例化
# 依然在第一次使用时连接数据库
```

session对象线程不安全。所以不同线程应该使用不用的session对象。
Session类和engine有一个就行了。

CRUD操作

增

add()：增加一个对象
add_all()：可迭代对象，元素是对象

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

USER = 'wayne'
PASSWORD = 'wayne'
HOST = '127.0.0.1'
PORT = 3306
DATABASE = 'test'

connstr = 'mysql+pymysql://{}:{}@{}:{}/{}'.format(
    USER, PASSWORD, HOST, PORT, DATABASE
)

engine = create_engine(connstr, echo=True)

# 创建基类
Base = declarative_base()

# 创建实体类
class Student(Base):
    # 指定表名
    __tablename__ = 'student'
    # 定义属性对应字段
    id = Column(Integer, primary_key=True, autoincrement=True)
    name = Column(String(64), nullable=False)
    age = Column(Integer)

    def __repr__(self):
        return "<{} {} {} {}>".format(
            self.__class__.__name__, self.id, self.name, self.age
        )

# 删除表
Base.metadata.drop_all(engine)
# 创建表
Base.metadata.create_all(engine)

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

s = Student(name='tom') # 构造时传入
```

```
s.age = 20 # 属性赋值
print(s)

session.add(s)
print(s)
session.commit()
print(s)
print('~~~~~~~~~~~')

try:
    s.name = 'jerry'
    session.add_all([s])
    print(s)
    session.commit() # 提交能成功吗?
    print(s)
except:
    session.rollback()
    print('roll back')
    raise
```

add_all()方法不会提交成功的，不是因为它不对，而是s，s成功提交后，s的主键就有了值，所以，只要s没有修改过，就认为没有改动。如下，s变化了，就可以提交修改了。

```
    s.name = 'jerry' # 修改
    session.add_all([s])
```

s主键没有值，就是新增；主键有值，就是找到主键对应的记录修改。

简单查询

使用query()方法，返回一个Query对象

```
students = session.query(Student) # 无条件
print(students) # 无内容，惰性的
for student in students:
    print(student)
print('~~~~~~~~~~~~~')

student = session.query(Student).get(2) # 通过主键查询
print(student)
```

query方法将实体类传入，返回类的对象可迭代对象，这时候并不查询。迭代它就执行SQL来查询数据库，封装数据到指定类的实例。
get方法使用主键查询，返回一条传入类的一个实例。

改

```
student = session.query(Student).get(2)
print(student)
student.name = 'sam'
student.age = 30
print(student)
session.add(student)
session.commit()
```

先查回来，修改后，再提交更改。

删除

先看下数据库，表中有

```
1   tom 20
2   sam 30
3   jerry   20
4   ben 20
5   ben 20
```

编写如下程序来删除数据，会发生什么?

```
try:
    student = Student(id=2, name="sam", age=30)

    session.delete(student)
    session.commit()
except Exception as e:
    session.rollback()
    print('~~~~~~~~')
    print(e)
```

会产生一个异常
`Instance '<Student at 0x3e654e0>' is not persisted` 未持久的异常!

状态

每一个实体，都有一个状态属性_sa_instance_state，其类型是sqlalchemy.orm.state.InstanceState，可以使用sqlalchemy.inspect(entity)函数查看状态。
常见的状态值有transient、pending、persistent、deleted、detached。

状态	说明
transient	实体类尚未加入到session中，同时并没有保存到数据库中
pending	transient的实体被add()到session中，状态切换到pending，但它还没有flush到数据库中
persistent	session中的实体对象对应着数据库中的真实记录。pending状态在提交成功后可以变成persistent状态，或者查询成功返回的实体也是persistent状态
deleted	实体被删除且已经flush但未commit完成。事务提交成功了，实体变成detached，事务失败，返回persistent状态
detached	删除成功的实体进入这个状态

新建一个实体，状态是transient临时的。
一旦add()后从transient变成pending状态。
成功commit()后从pending变成persistent状态。
成功查询返回的实体对象，也是persistent状态。

persistent状态的实体，修改依然是persistent状态。

persistent状态的实体，删除后，flush后但没有commit，就变成deteled状态，成功提交，变为detached状态，提交失败，还原到persistent状态。flush方法，主动把改变应用到数据库中去。

删除、修改操作，需要对应一个真实的记录，所以要求实体对象是persistent状态。

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

connstr = "{}://{}:{}@{}:{}/{}".format(
    'mysql+pymysql', 'wayne', 'wayne',
    '192.168.142.140', 3306, 'test'
)

engine = create_engine(connstr, echo=True)

Base = declarative_base()

# 创建实体类
class Student(Base):
    # 指定表名
    __tablename__ = 'student'
    # 定义属性对应字段
    id = Column(Integer, primary_key=True, autoincrement=True)
    name = Column(String(64), nullable=False)
    age = Column(Integer)
    # 第一参数是字段名，如果和属性名不一致，则一定要指定
    # age = Column('age', Integer)

    def __repr__(self):
        return "{} id={} name={} age={}".format(
```

```
            self.__class__.__name__, self.id, self.name, self.age)

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

from sqlalchemy.orm.state import InstanceState
def getstate(entity, i):
    insp = inspect(entity)
    state = 'sessionid={}, attached={}, transient={}, pending={},\npersistent={}, deleted={},
detached={}'.format(
        insp.session_id, insp._attached, insp.transient,
        insp.pending, insp.persistent, insp.deleted, insp.detached
    )
    print(i, state)
    print('key =', insp.key)
    print('-' * 30)


student = session.query(Student).get(2)
getstate(student, 1) # persistent

try:
    student = Student(id=2, name='sam', age=30)
    getstate(student, 2) # transit

    student = Student(name='sammy', age=30)
    getstate(student, 3) # transient
    session.add(student) # add后变成pending
    getstate(student, 4) # pending
    # session.delete(student) # 异常，删除的前提必须是persistent，也就是说先查后删
    # getstate(student, 5)
    session.commit() # 提交后，变成persistent
    getstate(student, 6) # persistent
except Exception as e:
    session.rollback()
    print(e, '~~~~~~~~~~~~~~~~')

# 运行结果
1 sessionid=1, attached=True, transient=False, pending=False,
persistent=True, deleted=False, detached=False
key = (<class '__main__.Student'>, (2,), None)
persistent就是key不为None，附加的，且不是删除的，有sessionid
------------------------------
2 sessionid=None, attached=False, transient=True, pending=False,
persistent=False, deleted=False, detached=False
key = None
transient的key为None，且无附加
------------------------------
3 sessionid=None, attached=False, transient=True, pending=False,
persistent=False, deleted=False, detached=False
key = None
同上
------------------------------
```

```
4 sessionid=1, attached=True, transient=False, pending=True,
persistent=False, deleted=False, detached=False
key = None
add后变成pending，已附加，但是没有key，有了sessionid
------------------------------
sqlalchemy.engine.base.Engine COMMIT
6 sessionid=1, attached=True, transient=False, pending=False,
persistent=True, deleted=False, detached=False
key = (<class '__main__.Student'>, (6,), None)
提交成功后，变成persistent
------------------------------
```

```python
student = session.query(Student).get(5)
getstate(student, 10) # persistent

try:
    session.delete(student) # 删除的前提是persistent
    getstate(student, 11) # persistent
    session.flush()
    getstate(student, 12) # deleted
    session.commit()
    getstate(student, 13) # detached
except Exception as e:
    session.rollback()
    print('~~~~~~~~')
    print(e)

# 运行结果
10 sessionid=1, attached=True, transient=False, pending=False,
persistent=True, deleted=False, detached=False
key = (<class '__main__.Student'>, (5,), None)
------------------------------
11 sessionid=1, attached=True, transient=False, pending=False,
persistent=True, deleted=False, detached=False
key = (<class '__main__.Student'>, (5,), None)
------------------------------

sqlalchemy.engine.base.Engine DELETE FROM student WHERE student.id = %(id)s
sqlalchemy.engine.base.Engine {'id': 5}

12 sessionid=1, attached=True, transient=False, pending=False,
persistent=False, deleted=True, detached=False
key = (<class '__main__.Student'>, (5,), None)
delete后flush，状态变成deleted，不过是附加的
------------------------------

sqlalchemy.engine.base.Engine COMMIT
一旦提交后
13 sessionid=None, attached=False, transient=False, pending=False,
persistent=False, deleted=False, detached=True
key = (<class '__main__.Student'>, (5,), None)
状态转为detached
```

```
------------------------------
```

复杂查询

实体类

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

Base = declarative_base()

connstr = "{}://{}:{}@{}:{}/{}".format(
    'mysql+pymysql', 'wayne', 'wayne',
    '192.168.142.140', 3306, 'test'
)

engine = create_engine(connstr, echo=True)
Session = sessionmaker(bind=engine)
session = Session()

class MyEnum(enum.Enum):
    M = 'M'
    F = 'F'

class Employee(Base):
    # 指定表名
    __tablename__ = 'employees'

    # 定义属性对应字段
    emp_no = Column(Integer, primary_key=True)
    birth_date = Column(Date, nullable=False)
    first_name = Column(String(14), nullable=False)
    last_name = Column(String(16), nullable=False)
    gender = Column(Enum(MyEnum), nullable=False)
    hire_date = Column(Date, nullable=False)

    def __repr__(self):
        return "{} no={} name={} {} gender={}".format(
            self.__class__.__name__, self.emp_no, self.first_name, self.last_name,
            self.gender.value
        )

# 打印函数
def show(emps):
    for x in emps:
        print(x)
    print('~~~~~~~~~~~~~~~',end='\n\n')


# 简单条件查询
emps = session.query(Employee).filter(Employee.emp_no > 10015)
```

```
show(emps)

# 与或非
from sqlalchemy import or_, and_, not_
# AND 条件
emps = session.query(Employee).filter(Employee.emp_no > 10015).filter(Employee.gender ==
MyEnum.F)
show(emps)

emps = session.query(Employee).filter(and_(Employee.emp_no > 10015, Employee.gender ==
MyEnum.M))
show(emps)

# & 一定要注意&符号两边表达式都要加括号
emps = session.query(Employee).filter((Employee.emp_no > 10015) & (Employee.gender == MyEnum.M))
show(emps)

# OR 条件
emps = session.query(Employee).filter((Employee.emp_no > 10018) | (Employee.emp_no < 10003))
show(emps)

emps = session.query(Employee).filter(or_(Employee.emp_no > 10018, Employee.emp_no < 10003))
show(emps)

# Not
emps = session.query(Employee).filter(not_(Employee.emp_no < 10018))
show(emps)
# 一定注意要加括号
emps = session.query(Employee).filter(~(Employee.emp_no < 10018))
show(emps)
# 总之，与或非的运算符&、|、~，一定要在表达式上加上括号

# in
emplist = [10010, 10015, 10018]
emps = session.query(Employee).filter(Employee.emp_no.in_(emplist))
show(emps)

# not in
emps = session.query(Employee).filter(~Employee.emp_no.in_(emplist))
emps = session.query(Employee).filter(Employee.emp_no.notin_(emplist))
show(emps)

# like
emps = session.query(Employee).filter(Employee.last_name.like('P%'))
show(emps)
# not like
emps = session.query(Employee).filter(Employee.last_name.notlike('P%'))
# ilike可以忽略大小写匹配
```

排序

```
# 排序
# 升序
emps = session.query(Employee).filter(Employee.emp_no > 10010).order_by(Employee.emp_no)
emps = session.query(Employee).filter(Employee.emp_no > 10010).order_by(Employee.emp_no.asc())
show(emps)

# 降序
emps = session.query(Employee).filter(Employee.emp_no > 10010).order_by(Employee.emp_no.desc())
show(emps)

# 多列排序
emps = session.query(Employee).filter(Employee.emp_no >
10010).order_by(Employee.last_name).order_by(Employee.emp_no.desc())
show(emps)
```

分页

```
# 分页
emps = session.query(Employee).limit(4)
show(emps)

emps = session.query(Employee).limit(4).offset(18)
show(emps)
```

消费者方法

消费者方法调用后，Query对象（可迭代）就转换成了一个容器。

```
# 总行数
emps = session.query(Employee)
print(len(list(emps))) # 返回大量的结果集，然后转换list
print(emps.count()) # 聚合函数count(*)的查询

# 取所有数据
print(emps.all()) # 返回列表，查不到返回空列表

# 取首行
print(emps.first()) # 返回首行，查不到返回None，等价limit

# 有且只能有一行
#print(emps.one()) #如果查询结果是多行抛异常
print(emps.limit(1).one())

# 删除 delete by query
session.query(Employee).filter(Employee.emp_no > 10018).delete()
#session.commit() # 提交则删除
```

first方法本质上就是limit语句

聚合、分组

```
# 聚合函数
# count
from sqlalchemy import func
query = session.query(func.count(Employee.emp_no))
print(query.one()) # 只能有一行结果
print(query.scalar()) # 取one()返回元组的第一个元素

# max/min/avg
print(session.query(func.max(Employee.emp_no)).scalar())
print(session.query(func.min(Employee.emp_no)).scalar())
print(session.query(func.avg(Employee.emp_no)).scalar())

# 分组
print(session.query(Employee.gender,
func.count(Employee.emp_no)).group_by(Employee.gender).all())
```

关联查询

```
CREATE TABLE `departments` (
  `dept_no` char(4) NOT NULL,
  `dept_name` varchar(40) NOT NULL,
  PRIMARY KEY (`dept_no`),
  UNIQUE KEY `dept_name` (`dept_name`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `employees` (
  `emp_no` int(11) NOT NULL,
  `birth_date` date NOT NULL,
  `first_name` varchar(14) NOT NULL,
  `last_name` varchar(16) NOT NULL,
  `gender` enum('M','F') NOT NULL,
  `hire_date` date NOT NULL,
  PRIMARY KEY (`emp_no`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `dept_emp` (
  `emp_no` int(11) NOT NULL,
  `dept_no` char(4) NOT NULL,
  `from_date` date NOT NULL,
  `to_date` date NOT NULL,
  PRIMARY KEY (`emp_no`,`dept_no`),
  KEY `dept_no` (`dept_no`),
  CONSTRAINT `dept_emp_ibfk_1` FOREIGN KEY (`emp_no`) REFERENCES `employees` (
  `emp_no`) ON DELETE CASCADE,
  CONSTRAINT `dept_emp_ibfk_2` FOREIGN KEY (`dept_no`) REFERENCES `departments
  ` (`dept_no`) ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

从语句看出员工、部门之间的关系是多对多关系。
先把这些表的Model类和字段属性建立起来。

```
# 创建实体类
```

```
class Employee(Base):
    # 指定表名
    __tablename__ = 'employees'
    # 定义属性对应字段
    emp_no = Column(Integer, primary_key=True)
    birth_date = Column(Date, nullable=False)
    first_name = Column(String(14), nullable=False)
    last_name = Column(String(16), nullable=False)
    gender = Column(Enum(MyEnum), nullable=False)
    hire_date = Column(Date, nullable=False)
    # 第一参数是字段名，如果和属性名不一致，一定要指定
    # age = Column('age', Integer)

    def __repr__(self):
        return "{} no={} name={} {} gender={}".format(
            self.__class__.__name__, self.emp_no, self.first_name, self.last_name,
            self.gender.value
        )

class Department(Base):
    __tablename__ = 'departments'

    dept_no = Column(String(4), primary_key=True)
    dept_name = Column(String(40), nullable=False, unique=True)

    def __repr__(self):
        return "{} no={} name={}".format(
            type(self).__name__, self.dept_no, self.dept_name)

class Dept_emp(Base):
    __tablename__ = "dept_emp"

    emp_no = Column(Integer, ForeignKey('employees.emp_no',
ondelete='CASCADE'),primary_key=True)
    dept_no = Column(String(4), ForeignKey('departments.dept_no', ondelete='CASCADE'),
primary_key=True)
    from_date = Column(Date, nullable=False)
    to_date = Column(Date, nullable=False)

    def __repr__(self):
        return "{} empno={} deptno={}".format(
            type(self).__name__, self.emp_no, self.dept_no)
```

`ForeignKey('employees.emp_no', ondelete='CASCADE')` 定义外键约束

需求
查询10010员工的所在的部门编号及员工信息
1、使用隐式内连接

```
# 查询10010员工的所在的部门编号及员工信息
results = session.query(Employee, Dept_emp).filter(Employee.emp_no ==
Dept_emp.emp_no).filter(Employee.emp_no == 10010).all()
show(results)

# 查询结果2行
(Employee no=10010 name=Duangkaew Piveteau gender=F, Dept_emp empno=10010 deptno=d004)
(Employee no=10010 name=Duangkaew Piveteau gender=F, Dept_emp empno=10010 deptno=d006)
```

这种方式产生隐式连接的语句

```
SELECT *
FROM employees, dept_emp
WHERE employees.emp_no = dept_emp.emp_no AND employees.emp_no = 10010
```

2、使用join

```
# 查询10010员工的所在的部门编号及员工信息
# 第一种写法
results = session.query(Employee).join(Dept_emp).filter(Employee.emp_no == 10010).all()

# 第二种写法
results = session.query(Employee).join(Dept_emp, Employee.emp_no ==
Dept_emp.emp_no).filter(Employee.emp_no == 10010).all()

print(results)
```

这两种写法，返回都只有一行数据，为什么?
它们生成的SQL语句是一样的，执行该SQL语句返回确实是2行记录，可以Python中的返回值列表中只有一个元素?
原因在于 `query(Employee)` 这个只能返回一个实体对象中去，为了解决这个问题，需要修改实体类Employee，增加属性用来存放部门信息

sqlalchemy.orm.relationship(实体类名字符串)

```
class Employee(Base):
    # 指定表名
    __tablename__ = 'employees'
    # 定义属性对应字段
    emp_no = Column(Integer, primary_key=True)
    birth_date = Column(Date, nullable=False)
    first_name = Column(String(14), nullable=False)
    last_name = Column(String(16), nullable=False)
    gender = Column(Enum(MyEnum), nullable=False)
    hire_date = Column(Date, nullable=False)

    departments = relationship('Dept_emp') #

    def __repr__(self): # 注意增加self.dept_emps
```

```
        return "{} no={} name={} {} gender={} depts={}".format(
            self.__class__.__name__, self.emp_no, self.first_name, self.last_name,
            self.gender.value, self.departments
        )
```

查询信息

```
# 查询10010员工的所在的部门编号及员工信息
# 第一种
results = session.query(Employee).join(Dept_emp).filter(Employee.emp_no ==
Dept_emp.emp_no).filter(Employee.emp_no == 10010)

# 第二种
results = session.query(Employee).join(Dept_emp, Employee.emp_no ==
Dept_emp.emp_no).filter(Employee.emp_no == 10010)

# 第三种
results = session.query(Employee).join(Dept_emp, (Employee.emp_no == Dept_emp.emp_no) &
(Employee.emp_no == 10010))

show(results.all()) # 打印结果
```

第一种方法join(Dept_emp)中没有等值条件，会自动生成一个等值条件，如果后面有filter，哪怕是filter(Employee.emp_no == Dept_emp.emp_no)，这个条件会在where中出现。第一种这种自动增加join的等值条件的方式不好，不要这么写

第二种方法在join中增加等值条件，阻止了自动的等值条件的生成。这种方式推荐

第三种方法就是第二种，这种方式也可以。

再看一个现象

```
class Employee(Base):
    # 指定表名
    __tablename__ = 'employees'
    # 定义属性对应字段
    emp_no = Column(Integer, primary_key=True)
    birth_date = Column(Date, nullable=False)
    first_name = Column(String(14), nullable=False)
    last_name = Column(String(16), nullable=False)
    gender = Column(Enum(MyEnum), nullable=False)
    hire_date = Column(Date, nullable=False)

    departments = relationship('Dept_emp')

    def __repr__(self):
        return "{} no={} name={} {} gender={} depts={}".format(
            self.__class__.__name__, self.emp_no, self.first_name, self.last_name,
            self.gender.value, self.departments
        )
```

```
results = session.query(Employee).join(Dept_emp, (Employee.emp_no == Dept_emp.emp_no) &
(Employee.emp_no == 10010))

for x in results:
    print(x.emp_no)
    #print(x.departments) # 观察有无此条语句打印的结果及生成SQL语句的变化
    #print(x) # 查询结果有什么变化
```

可以看出只要不访问departments属性，就不会查dept_emp这张表。

总结

在开发中，一般都会采用ORM框架，这样就可以使用对象操作表了。
定义表映射的类，使用Column的描述器定义类属性，使用ForeignKey来定义外键约束。
如果在一个对象中，想查看其它表对应的对象的内容，就要使用relationship来定义关系。

是否使用外键约束?
1、力挺派
能使数据保证完整性一致性
2、弃用派
开发难度增加，大数据的时候影响插入、修改、删除的效率。
在业务层保证数据的一致性。

作业

使用SqlAlchemy
10009号员工的工号、姓名、所有的头衔title
10010号员工的工号、姓名、所在部门名称