实验报告

22371437 张智威

Test1-ORM.PY

首先对数据库中的各个关系与 Test1-ORM.PY 中的对象进行了一一对应，同时设置类的属性与关系的列的对应关系，按照题目要求设置各表的主键，唯一约束，外键等等。同时通过 relationship() 函数建立 .py 文件中对象之间的关系，如下图：

```
emp_dept_emps = db.relationship( *args: 'Dept_emp', backref='employee', lazy=True, cascade='all, delete')
man_dept_managers = db.relationship( *args: 'Dept_manager', backref='employee', lazy=True, cascade='all, delete')
```

同时为每一个类设置了一个 to_dict(self) 方法，便于获取一个对象对应的属性信息，即对应表中元组的属性信息，下面为 dept_emp 表的例子：

```
def to_dict(self):
    return {
        'emp_no': self.emp_no,
        'dept_no': self.dept_no,
        'from_date': self.from_date,
        'to_date': self.to_date,
    }
```

对于增加触发器的处理，初始时由于没看到后面的sql语句，因此尝试了在 .py 中对对象添加了触发器，而不是通过sql语句加入到数据库中，如下图：

```
    """
    不通过sql语句创建触发器

    @staticmethod
    def add_to_title(target, value, initiator):
        tmp = Dept_manager_title()
        tmp.emp_no = target.emp_no
        tmp.from_date = target.from_date
        tmp.to_date = target.to_date
        db.session.add(tmp)
        db.session.commit()

    @staticmethod
    def delete_title(target):
        db.session.query(Dept_manager_title).filter(Dept_manager_title.emp_no == target.emp_no).delete()

db.event.listen(Dept_manager, 'append', Dept_manager.add_to_title)
db.event.listen(Dept_manager, 'remove', Dept_manager.delete_title)
    """
```

在发现了有用于增加触发器的sql语句后，通过编写sql语句，使得在创建表单的时候，增加触发器到所需关系中，实现题目要求的同步添加数据和同步删除数据功能，分别在添加数据和删除数据后接着更新关联的表的数据，如下图：

```
sql_trigger_insert = text("""CREATE TRIGGER sql_trigger_insert
                                AFTER INSERT ON DEPT_MANAGER FOR EACH ROW
                             BEGIN
                                INSERT INTO DEPT_MANAGER_TITLE(EMP_NO,FROM_DATE,TO_DATE)
                                VALUES(NEW.EMP_NO,NEW.FROM_DATE,NEW.TO_DATE);
                             END;""")

sql_trigger_delete = text("""CREATE TRIGGER sql_trigger_delete
                                AFTER DELETE ON DEPT_MANAGER FOR EACH ROW
                            BEGIN
                                DELETE FROM DEPT_MANAGER_TITLE WHERE EMP_NO = OLD.EMP_NO AND FROM_DATE = OLD.FROM_DATE
                                AND TO_DATE = OLD.TO_DATE;
                            END;""")
```

随后通过代码中本身提供的读取 .csv 文件的函数，分别读取各个关系表的数据，并通过不同的列名读取不同列的数据，通过 `session.bulk_save_objects()` 函数以及建立的关系表和对象的联系，全部插入相对应的关系表中，完成数据的注入。

Test2-RESTful.PY

在 Test2-RESTful.PY 中要分别对insert，delete，update，select功能实现Restful接口，接下来一个一个分析具体实现方式：

1. insert

本身代码中给出，不再赘述。

2. update

在读入 request.json 文件内容（即传入的data）后，python自动为我们通过dict类型进行存储，于是我们将key（即列名）与 values（即数据）分别用两个列表进行存储，随后通过执行 sql 语句获取对应表的主键名，存入primary_key_columns中：

```
data = request.json
# 获取列和数据
keys = list(data.keys())
values = list(data.values())
cursor = db.cursor()
# 获取主键名称
cursor.execute(f"SHOW KEYS FROM {table_name} WHERE Key_name = 'PRIMARY'")
primary_key_columns = [row[4] for row in cursor.fetchall()]
```

随后对进行数据更新的 where 语句进行拼接，查找到与主键名相同的 key 并把 对应的 value 拼接到 where 字符串中，最后用于更新时的 where 条件。同时对更新的值按 sql 的 SET 语句格式拼接为 `“key1=value1，key2=value2...”` 的格式。然后添加到 sql 语句中进行执行，最后返回一个json格式的信息：

```
i = 0
where = ""
for key in keys:
    for primary_key in primary_key_columns:
        if key == primary_key:
            where += key + " = '" + str(values[i]) + "'"
    keys[i] = key + " = '" + str(values[i]) + "'"
    i = 1 + i
word = ' ,'.join(keys)
sql = f'update {table_name} set {word} where {where}'
cursor.execute(sql)
db.commit()
return jsonify({'message': 'data updated successfully'})
```

3. delete

Delete 方法与 update 大致相同，除了对 sql 语句进行改变，以及多加了一个从 url 中读入要删除的元组的主键值外，其余部分的数据处理与 update 近乎相同：

```
@app.route( rule: '/api/v1/<table_name>/<path:args>', methods=['DELETE'])
def delete_data(table_name, args):
    cursor = db.cursor()
    args = args.split('/')
    cursor.execute(f"SHOW KEYS FROM {table_name} WHERE Key_name = 'PRIMARY'")
    primary_key_columns = [row[4] for row in cursor.fetchall()]

    for i in range(len(primary_key_columns)):
        primary_key_columns[i] += " = '" + args[i] + "'"

    where = ' and '.join(primary_key_columns)
    sql = f"delete from {table_name} where {where}"
    cursor.execute(sql)
    db.commit()
    return jsonify({'message': 'data deleted successfully'})
    # TODO: finish this fuction to update data
```

4. select

select 方法的实现较为复杂，需要处理两种情况：

(1).带 id 的查询

通过从 url 中获取主键属性（直接从方法的参数中获得），直接编写 where 子句，遇到多列为主键的表时进行处理，将筛选语句用 `and` 进行连接，随后获取到数据后，转化为字符串（使得返回的数据更加直接，避免出现形如 `datetime.date()` 的语句，转化为直接的时间输出），再转化为 json 文件后返回。

```python
@app.route( rule: '/api/v1/<table_name>/<path:args>', methods=['GET'])
def select_data(table_name, args):
    cursor = db.cursor()
    # 获取主键数据
    args = args.split('/')
    cursor.execute(f"SHOW KEYS FROM {table_name} WHERE Key_name = 'PRIMARY'")
    primary_key_columns = [row[4] for row in cursor.fetchall()]

    for i in range(len(primary_key_columns)):
        primary_key_columns[i] += " = '" + args[i] + "'"

    # 如果有多个主键用 and 连接
    where = ' and '.join(primary_key_columns)
    sql = f"select * from {table_name} where {where}"
    cursor.execute(sql)
    datas = [row for row in cursor.fetchall()]
    # 将数据转化为字符串进行输出
    strs = []
    for row in datas:
        string = []
        for data in row:
            string.append(str(data))
        strs.append(', '.join(string))

    db.commit()
    return jsonify({'data': f'{data}' for data in strs})
```

---

(2).带筛选条件 ?query 的查询

分情况讨论，带筛选条件或不带筛选条件的情况。通过 `request.args` 获取到要进行筛选的条件，并分别存储到 `filter_column, filter_value` 中，随后像前面一样形成 sql 语句随后查询，返回一个 json 文件。

```python
@app.route( rule: '/api/v1/<table_name>', methods=['GET'])
def select_data_condition(table_name):
    global filter_column, filter_value
    cursor = db.cursor()

    # 获取筛选条件
    string = request.args.to_dict()
    for i in string.keys():
        filter_column = i
    for i in string.values():
        filter_value = i

    # 分是否有筛选条件的情况进行讨论
    if filter_column and filter_value:
        sql = f"select * from {table_name} where {filter_column} = {filter_value}"
    else:
        sql = f"select * from {table_name}"

    cursor.execute(sql)
    datas = cursor.fetchall()
    columns = [col[0] for col in cursor.description]
    data = []
    for row in datas:
        row_data = {}
        for i in range(len(columns)):
            row_data[columns[i]] = str(row[i])
        data.append(row_data)
    db.commit()
    return jsonify({'data': f'{data}'})
```
