实现第一个 GPT 聊天机器人 - 章节介绍

OpenAI API 实现第一个聊天接口

- 解读 OpenAI Library 使用技巧与数据响应格式，实现项目内第一个 GPT 聊天接口，使用 Postman 调试并记录接口。
- 添加 PyTest 测试框架，从零基础开始掌握 PyTest 测试攻略，并约定项目测试规范，对第一个接口进行相应的单元测试。

规范化项目的响应 / 错误 / 数据校验

- 统一项目的响应数据格式、错误异常分类，为项目添加日志，将错误 / 日志信息存储到本地文件。

- 为项目添加 Flask-wft 扩展，实现对前端请求数据进行统一校验，确保数据安全。

使用 Postgres 作为数据库方案

- 为项目添加 Flask-SQLAlchemy 和 Flask-Migrate 扩展便于操作数据库。

- 在项目中创建 APP 模型，并将模型映射到 Postgres 数据库中，让项目联通数据库，测试基础增删改查。

30 行代码实现一个聊天机器人 API

校验 API 接口输入请求

将 .env 加载到环境变量实现配置分离

在 Python 项目中，如果想将 .env 文件的信息加载成环境变量，可以使用 python-dotenv 库。

通过 os.getenv 即可获取在环境变量中的配置信息，将代码与配置分离，方便后期维护。

为什么需要对接口请求进行校验

安全性：通过验证请求的数据，可以确保输入符合预期，避免存在的安全漏洞，如 SQL 注入等。

数据完整性：确保数据的完整，例如数据格式、类型、长度等，避免因为不完整或不正确的数据导致系统出错。

提高用户体验：对用户提交的不合法请求给出友好提示，引导用户按照规定格式或要求重新输入，从而提高用户体验。

Flask-wtf 的安装与使用

Flask-wtf 是基于 wtforms 封装的 Flask 插件，支持 CSRF 保护、快速提取数据、自定义验证规则及 wtforms 的所有规则。

wtforms 支持提取的数据类型：布尔值、日期、文本、时间、浮点数、邮箱、文件、音频、整数、自定义数据类型等。

wtforms 支持校验的规则：必填、长度、邮箱、正则、范围、 UUID 、数据可选、 URL 等。

课程总结

在项目中配置 python-dotenv 和加载配置的方式来实现配置 Flask 相关的配置。

为项目安装 flask-wtf 和 wtforms 实现对请求数据的校验，确保请求数据的安全。

统一响应接口设计与实现

统一响应接口设计与实现

01. 为什么要返回统一数据格式?

建立前后端的接口约定和规范，统一数据格式，方便处理数据，前后端可以更好地交互及通讯。如果不统一响应数据格式，一部分接口返回文本，一部分接口返回 json 数据，一部分接口直接抛出异常，会让前端判断非常困难，例如：

```
// 返回文本字符串
你好！有什么可以帮助你的吗?
```

```
// 返回json字典
{
    "content": "你好！有什么可以帮助你的吗?"
}
```

```
// 返回json字典
{
    "code": "success",
    "message": "",
    "data": {
        "content": "你好！有什么可以帮助你的吗?"
    }
}
```

```
// 返回错误信息
<!doctype html>
<html lang=en>
<title>404 Not Found</title>
<h1>Not Found</h1>
<p>The requested URL was not found on the server. If you entered the URL manually
please check your spelling and try again.</p>
```

前端需要对接口的 N 多种可能进行判断，确认为合格的数据后才能处理，效率非常低。

02. 接口返回值设计

HTTP 响应状态码，无论是成功还是失败，全部设置为 200，通过 业务状态码 来判断操作是否成功，减少前端的判断。发生错误时，需要在 `message` 和 `data` 中携带相应的错误与描述信息。

2.1 定义接口统一响应格式

```
{
    "code": "success",  // 响应状态
    "message": "获取AI应用数据成功",  // 业务消息提示
    "data": {}  // 业务获取的数据
}
```

为了简化开发，我们将将业务状态码定义成 `success` 、 `fail` 、 `not_found` 、 `unauthorized` 、 `forbidden` 、 `validate_error` 。

`message` 为相应的接口描述，如果业务状态码为非成功状态，需要添加上相应的描述信息。

2.2 接口分页数据响应格式

如果返回的数据为列表型分页数据，则在 data 中存在两个字段 `paginator` 和 `list`， 分别代表分页的数据信息，响应的数据列表信息，数据格式如下所示：

```
{
    "code": "success",
    "message": "",
    "data": {
        "paginator": {
            "page_size": 10,   // 当前页数每页条数
            "current_page": 1,   // 当前页数
            "total_page": 10,   // 总页数
            "total_record": 100   // 总记录条数
        },
        "list": [],   // 分页的列表数据
    }
}
```

异常错误状态统一设计与实现

PyTest 配置与 API 测试用例编写

PyTest配置与API测试用例

01. Pytest 简介与安装使用

Pytest 是一个功能强大且易于使用的 Python 测试框架，用于编写和运行单元测试用例、集成测试和功能测试，对比 unittest 框架，Pytest 的优势非常明显：

1. 简单易用：Pytest 的语法简单明了，学习曲线低，它采用了自然的测试用例编写方式，使得编写测试代码变得更加简单和直观。
2. 灵活性：Pytest 支持多种测试样式，包括函数式、类式、模块级别的测试，甚至支持测试生成器，而且插件生态非常丰富，可以通过插件来扩展 Pytest 的功能，满足各种测试场景。
3. 集成性：Pytest 可以很好地与其他测试工具和框架集成，比如与 `Jenkins`、`Travis CI`、`Coverage.py` 等。它还支持与其他测试框架（如 `unittest` 和 `doctest`）协同工作。
4. 丰富的输出信息：Pytest 在测试运行过程中提供丰富的输出信息，包括测试结果、失败原因、详细的堆栈跟踪等，便于开发人员定位和修复问题。

官网文档：https://docs.pytest.org/en/8.0.x/

安装命令：

```
pip install pytest
```

在 Pytest 中，测试用例的编写需要遵循以下规则：

1. 测试文件以 test_ 开头或 _test 结尾；
2. 测试类以 Test 开头，并且不能带有 init 方法；
3. 测试函数以 test_ 开头；
4. 断言使用基本的 assert 即可；

pytest 会递归查找当前目录及子目录下的所有以 test_ 开始或者 _test 结尾的 Python 脚本，执行其中符合规则的函数和方法，不需要显示调用，一些常见的运行命令：

```
# 直接运行文件夹内符合规则的所有用例
pytest folder_name

# 执行某个 Python 文件中的用例
pytest test_file.py

# 执行某个 Python 文件内的某个函数
pytest test_file.py::test_func

# 执行某个 Python 文件内某个测试类的某个方法
pytest test_file.py::TestClass::test_method

# 运行测试时显示标准输出(stdout)，允许测试中的 print() 语句直接输出到终端
pytest -s

# 运行测试时显示详细的信息，包括每个测试用例的名称及结果(通过/失败/跳过等)，-v 代表 verbose
pytest -v
```

02. fixture 设置测试环境

在 Pytest 中，fixture 是一种用于为测试函数提供预设数据或设置测试环境的机制。简单来说，可以使用 fixture 来为测试提供预设数据和设置测试环境的功能。

fixture 通常写在 conftest.py 文件中，并且对应的目录及子目录都可以使用 conftest.py 提供的 fixture，如果存在嵌套 conftest.py 文件，则使用的是最接近的 fixture。定义好 fixture 后，在测试函数中，可以使用同名的参数直接使用，Pytest 会自动注入。

示例：

```
# test/conftest.py

import pytest

from app.server.app import app


@pytest.fixture
def client():
    app.config["TESTING"] = True
    with app.test_client() as client:
        yield client
```

```
# test/internal/handler/test_app_handler.py
import pytest

from pkg.response import HttpCode


class TestAppHandler:
    def test_completion(self, client):
        r = client.post("/app/completion", json={"query": "你好,你是?"})
        assert r.status_code == 200
        assert r.json.get("code") == HttpCode.SUCCESS
```

03. Pytest 参数化测试

在 Pytest 中可以使用 `@pytest.mark.parametrize` 装饰测试函数，实现使用不同的参数值多次测试运行相同的代码，以覆盖多种测试场景，可以传递 2 个参数，分别是 `参数名` 和 `参数值`：

1. 参数名：`query` 为单个参数，`email, password` 为 2 个参数；
2. 参数值：`[None, "你好，你是?"]` 为单个参数对应的参数列表，`[("admin@imooc.com", "123456"), ("hi@imooc.com", "123456")]` 为多个参数对应的 元组参数列表；

使用示例如下：

```
# test/internal/handler/test_app_handler.py
import pytest

from pkg.response import HttpCode


class TestAppHandler:
```

```
    @pytest.mark.parametrize("query", [None, "你好，你是?"])
    def test_completion(self, query, client):
        r = client.post("/app/completion", json={"query": query})
        assert r.statuc_code == 200
        if query is None:
            assert r.json.get("code") == HttpCode.VALIDATE_ERROR
        else:
            assert r.json.get("code") == HttpCode.SUCCESS
```

上面这段代码会重复测试两次，第 1 次传递的 `query` 值为 `None`，并执行代码；第 2 次测试时传递的 `query` 值为 `你好，你是?`，通过对参数的不同判断测试结果，以检测代码是否正常运行。

04. pytest.ini 配置运行时

在 Pytest 中，基础的配置可以放在 `pytest.ini` 配置文件中，格式如下：

```
[pytest]
config_key=config_value
```

`pytest.ini` 中的支持的配置项非常多，录入：

```
[pytest]
# 设置测试缓存文件存储在 tmp/.pytest_cache 中
cache_dir = tmp/.pytest_cache

# 设置默认的命令行选项
addopts = -v -s

# 指定 Pytest 在哪些文件中查找测试函数
python_files = test_*.py *_test.py

# 指定 Pytest 在哪些类中查找测试方法
python_classes = Test*

# 指定 Pytest 在哪些函数中查找测试用例
python_functions = test_*
```

Flask-SQLAlchemy 扩展的配置与使用

Flask-SQLAlchemy扩展的配置与使用

在 Flask 中，所有扩展插件基本上都是以 `Flask-Xxx` 进行命名， `Flask-SQLAlchemy` 就是一个为 Flask 应用增加 `SQLAlchemy` 支持的扩展，致力于简化在 Flask 中 `SQLAlchemy` 的使用，而 `SQLAlchemy` 是目前 Python 中最强大的 ORM 框架，功能全面，使用简单。

文档资料：

1. Flask-SQLAlchemy 文档： https://flask-sqlalchemy.palletsprojects.com/en/3.1.x/
2. SQLAlchemy 文档： https://www.sqlalchemy.org/

01. ORM 是什么

`ORM 其实是 对象映射关系(Object-Relational Mapping) ，即将数据库中的表与面向对象编程中的类关联起来，它把数据库中的表映射为类，表中的行映射为类的示例，表中的列映射为类的属性。这样一来，就可以通过对类的操作来进行数据库的增删改查，而不必直接操作数据库，让程序员更加专注于业务逻辑，减少了与数据库交互的复杂性。`

例如有一张基础表为 `account`，字段信息如下：

```
CREATE TABLE "account" (
  "id" uuid NOT NULL,
    "name" varchar(255) NOT NULL,
    "email" varchar(255) NOT NULL,
    "avatar" varchar(255) NOT NULL,
    "password" varchar(255) NOT NULL,
    "updated_at" timestamp(6) NOT NULL,
    "created_at" timestamp(6) NOT NULL,
)
```

将其映射到 Python 的 Class 为：

```
class Account(db.Model):
    __tablename__ = "account"
    __table_args__ = (
        PrimaryKeyConstraint("id", name="pk_account_id"),
        Index("idx_account_email", "email"),
    )

    id = Column(UUID, default=uuid.uuid4, nullable=False)
    name = Column(String(255), default="", nullable=False)
    email = Column(String(255), default="", nullable=False)
    avatar = Column(String(255), default="", nullable=False)
    password = Column(String(255), default="", nullable=False)
    updated_at = Column(DateTime, default=datetime.now, onupdate=datetime.now,
nullable=False)
    created_at = Column(DateTime, default=datetime.now, nullable=False)
```

这样，在 Python中，只需要对映射类进行特定方法的调用，即可实现数据数据库的增删改查，写成伪代码示例如下：

```
# 类映射伪代码
account = Account(id="xxx-xxx-xxx-xxx")

account.add()  # 增:映射到sql中的 create table xxx (xxx, xxx, xxx);
account.delete()  # 删:映射到sql中的 delete from xxx where xxx;
account.update()  # 改:映射到sql中的 update table set xxx=xxx, xxx=xxx where xxx;
account.get()  # 查:映射到sql中的 select * from xxx where xxx;
```

通过 ORM 可以在代码中少写甚至不写 SQL，去实现数据库的 `增删改查等` 复杂业务，同时确保 SQL 安全，不被注入。从上面可以看出 `ORM` 的优缺点其实非常明显：

- 优点：
  - 有语法提示，省去自己拼写 SQL，保证 SQL 语法的正确性；
  - ORM 提供方言功能（dialect，可以转换为多种数据库语法），减少学习成本与迁移数据库的成本；
  - 面相对象，可读性强，开发效率高；
  - 防止 sql 注入攻击；
  - 搭配数据库迁移，更新数据库方便；
- 缺点：
  - 需要语法转换，效率比原生 sql 低；
  - 复杂的查询往往语法比较复杂（可以使用原生 sql 代替）；

02. 安装 Flask-SQLAlchemy

使用 pip 可以直接安装 Flask-SQLAlchemy，命令如下：

```
pip install flask-sqlalchemy
```

期间如果提示 `ModuleNotFoundError: No module named 'psycopg2'` 是因为使用 `Postgres` 作为数据库，但是缺少驱动引擎导致的，安装上对应的驱动引擎即可，命令如下：

```
pip install psycopg2
```

也可以一次性安装两个命令：

```
pip install flask-sqlalchemy psycopg2
```

和其他的 Flask 扩展一样，几乎绝大部分扩展都需要进行初始化与配置，Flask-SQLAlchemy 也一样，并且 Flask-SQLAlchemy 的相关配置也封装到了 Flask 的配置项中，可以通过 `app.config` **配置或配置加载方案**（config.from_object）进行设置。

SQLALCHEMY 的主要配置如下：

配置项	说明	示例
SQLALCHEMY_DATABASE_URI	设置数据库的连接地址	postgresql://root:123456@127.0.0.1:5432/test31
SQLALCHEMY_BINDS	访问多个数据库时，用于设置数据库的连接地址	{"other_db": "sqlite:///other.db"}

配置项	说明	示例
SQLALCHEMY_ECHO	是否打印底层执行的 SQL 语句	True
SQLALCHEMY_RECORD_QUERIES	是否记录执行的查询语句, 用于慢查询分析, 调试模式下自动启动	False
SQLALCHEMY_TRACK_MODIFICATIONS	是否追踪数据库变化(触发钩子函数), 会消耗额外的内存	False
SQLALCHEMY_ENGINE_OPTIONS	设置针对 `sqlalchemy` 本体的配置项	SQLALCHEMY_POOL_SIZE: 连接池最大数量 SQLALCHEMY_POOL_RECYCLE: 单个连接最长生命周期

要从 Flask 中将配置信息加载到 Flask-SQLAlchemy，可以使用 `db.init(flask_app)` 即可。

应用 ORM 模型的创建与增删改查

应用 ORM 模型的创建与增删改查

01. 创建模型

要想在项目中使用 ORM，必须创建对应的类，并继承 `db.Model`，即可实现与对应数据库进行关联。在类中还可以通过 `__tablename__` 属性来设置 表名，通过 `__table_args__` 来为表添加相应的 主键、index索引、unique索引 等约束。

其中关于 ORM 模型的字段类型、索引约束、主键等均可以从 `sqlalchemy` 包中导入，从这个包导入的类支持所有的数据库，在切换数据库时不需要进行调整，除此之外还可以导入特定数据库特有的结构，例如 Postgres 中特有的 JSONB 数据类型，特有的类型在切换数据库时，需要修改代码进行调整。

一些常见的 `sqlalchemy` 的数据类型：

```
from sqlalchemy import (
    Column,   # 用户定义字段
      UUID,   # UUID类型，例如：123e4567-e89b-12d3-a456-426655440000
      String,   # 字符串，对应数据库的 varchar
      Text,   # 长文本类型，对应数据库的 text
      DateTime,   # 事件类型，对应数据库的 timestamp
      Integer,   # 整形，对应数据库的 int
      Decimal,   # 小数型，对应数据库的 decimal
      PrimaryKeyConstraint,   # 创建主键约束
      Index,   # 创建索引
      UniqueConstraint,   # 创建唯一值约束
)
from sqlalchemy.dialects.postgresql import JSONB   # postgres 特有的 jsonb 数据，使用二进制存储 json，性能更强
```

完整示例：

```
import uuid
from datetime import datetime

from sqlalchemy import (
      Column,
      UUID,
      String,
      DateTime,
      PrimaryKeyConstraint,
      Index,
)
from sqlalchemy.dialects.postgresql import JSONB

from internal.extension.database_extension import db


class App(db.Model):
      """AI应用模型"""
      __tablename__ = "app"
      __table_args__ = (
            PrimaryKeyConstraint("id", name="pk_app_id"),
            Index("idx_app_account_id", "account_id"),
```

```
    )

    id = Column(UUID, default=uuid.uuid4, nullable=False)
    account_id = Column(UUID, nullable=False)
    name = Column(String(255), default="", nullable=False)
    icon = Column(String(255), default="", nullable=False)
    config = Column(JSONB, default={}, nullable=False)
    updated_at = Column(DateTime, default=datetime.now, onupdate=datetime.now,
nullable=False)
    created_at = Column(DateTime, default=datetime.now, nullable=False)
```

创建完模型后，可以通过调用 db.create_all() 在对应的数据库中创建相应的模型表，例如：

```
from flask import Flask
from internal.model import App

class Http(Flask):
    def __init__():
        ...
        with self.app_context():
            _ = App()
            db.create_all()
        ...
```

对于这类规律性+重复性的代码，可以考虑使用 ChatGPT 来生成，创建 ORM 模型提示词：

```
## 角色
你是一个拥有10年经验的资深Python工程师，精通Flask，Flask-SQLAlchemy，Postgres，以及其他
Python开发工具，能够为用户提出的需求或者提供的代码段生成指定的完整代码。

## 技能说明
- 如果需要实现Flask-SQLAlchemy的ORM类，集成`db.Model`时，从`from
internal.extension.database_extension import db`这里导入db；
- 创建ORM模型时，表名`__tablename__`及类名全部都是单数；
- 所有的字段都要添加`nullable=False`代表字段不允许为空；
- UUID类型的字段添加默认值`default=uuid.uuid4`，String类型的字段长度均设置为
`String(255)`，默认值设置为`default=""`；
- 所有模型都有`updated_at`和`created_at`字段，类型均是`DateTime`，其中`updated_at`包含
`default`和`onupdate`，而`created_at`仅包含`default`，值全部都是`datetime.now`；
- 请给ORM模型添加上`__table_args__`属性，涵盖`PrimaryKeyConstraint`为主键，所有模型都以
`id`为主键，主键的类型为`UUID`，如果用户声明其他约束，例如`UniqueConstraint`，`Index`等时，
请按照需求进行添加；
- 属性的类型全部从`sqlalchemy`包中导入，例如：`from sqlalchemy import (Column, UUID,
String, DateTime, PrimaryKeyConstraint, UniqueConstraint)`；
- `uuid.uuid4`从`import uuid`中导入，`datetime.now`从`from datetime import
datetime`导入；
- 对于`description`等字段，通过字面意思，可以看出是描述，一般内容比较长，可以使用`Text`类型；
- 用户如果表名了某个字段类型为json，则统一设置成`JSONB`，并从`from
sqlalchemy.dialects.postgresql import JSONB`导入，这是Postgres特有的；
- 其他的规范请根据你的知识库进行操作，项目使用的数据库是Postgres；

## 操作示例
```json
import uuid
```
```

```
from datetime import datetime

from sqlalchemy import (
    Column,
    UUID,
    String,
    DateTime,
    PrimaryKeyConstraint,
    UniqueConstraint,
    Index,
)

from internal.extension.database_extension import db


class AccountOAuth(db.Model):
    """第三方授权认证账号模型"""
    __tablename__ = "account_oauth"
    __table_args__ = (
        PrimaryKeyConstraint("id", name="pk_account_oauth_id"),
        UniqueConstraint("account_id", "provider",
name="uk_account_oauth_account_id_provider"),
        UniqueConstraint("provider", "openid",
name="uk_account_oauth_provider_openid"),
        Index("idx_account_oauth_account_id", "account_id")
    )

    id = Column(UUID, default=uuid.uuid4, nullable=False)
    account_id = Column(UUID, nullable=False)
    provider = Column(String(255), default="", nullable=False)
    openid = Column(String(255), default="", nullable=False)
    encrypted_token = Column(String(255), default="", nullable=False)
    updated_at = Column(DateTime, default=datetime.now, onupdate=datetime.now,
nullable=False)
    created_at = Column(DateTime, default=datetime.now, nullable=False)
```

```
## 注意事项
- 只处理与生成Python 测试用例相关的提问，对于其他非相关行业问题，请婉拒回答。
- 只使用用户使用的语言进行回答，不使用其他语言。
- 确保回答的针对性和专业性。

用户的需求是：
```

02. 新增数据

在 Flask-SQLAlchemy 中有两种主要的方式来执行数据库查询：

1. 使用 `db.session.query(Model).xxx`，这种方法使用 SQLAlchemy 原生查询语法来操作数据库，可以使用 `.query` 来访问一个表的查询，然后使用 `.xxx` 来指定你要执行的操作，比如 `.filter()`、`.order_by()` 等。这种方法更加灵活，可以用于复杂的查询需求。
2. 也可以使用 ORM 模型的查询，每个继承 `db.Model` 的模型都有一个 `query` 属性，可以通过这个属性来执行查询，例如 `App.query.filter()` 等。

示例：

```
# 使用 db.session.query.xxx 的方式
from yourapp import db
from yourapp.models import User

# 查询所有用户，并按照ID升序排列
users = db.session.query(User).order_by(User.id).all()

# 查询用户名为'john'的用户
user = db.session.query(User).filter_by(username='john').first()
```

```
# 使用 ORM 模型的方式
from yourapp.models import User

# 查询所有用户，并按照ID升序排列
users = User.query.order_by(User.id).all()

# 查询用户名为'john'的用户
user = User.query.filter_by(username='john').first()
```

在 LLMOps 项目中，我们约定如果是涉及到单表的增删改查，均使用 `ORM模型` 的方式进行操作，如果涉及到多表的，使用 `db.session.query.xxx` 的方式进行操作。

如果想要往表里添加数据，只需依次调用 `add(model)` 和 `.commit()` 即可，示例：

```
def demo():
    # 实例化 Students 模型对象
    user = Students(name='yy', fullname='yoyo')
    # 添加到会话，并用commit提交数据
    db.session.add(user)
    db.session.commit()
    return {
        "code": 0,
        "msg": "create success!"
    }
```

03. 查询数据

在 Flask-SQLAlchemy 中执行数据查询的方式非常多，示例：

```
Model.query.all()  # 获取表内的所有数据
Model.query.first()  # 获取第一个查询结果
Model.query.get(id)  # 获取 id 进行查询
Model.query.filter().all()  # 筛选查询，相当于 sql 中的 where 语句
Model.query.filter_by().all()  # 筛选查询，相当于 sql 中的 where 语句
Model.query.filter().one_or_none()  # 筛选查询，相当于 sql 中的 where+limit，并且只获取
一条数据
```

04. 修改和删除数据

在 Flask-SQLAlchemy 中，无论是更新还是删除，只要使用了 `db.session` 进行操作都需要调用 `commit()` 才会将数据同步到实际数据库中，如果使用 ORM 模型执行的操作则不需要。

```
# 1.通过调用 updated 方法进行更新（支持批量）
Students.query.filter_by(name='yy').update({"fullname": "xx"})

# 2.通过修改模型实例属性进行更新
student = Student.query.first()
student.name = "imooc"
db.session.commit()
```

```
# 1.通过调用 delete 方法删除数据（支持批量）
Student.query.filter_by(name="imooc").delete()

# 2.通过 db.session.delete 进行删除
student = Student.query.first()
db.session.delete(student)
db.session.commit()
```

重写 SQLAlchemy 核心类实现自动提交

重写SQLAlchemy核心类实现自动提交

01. commit与rollback

对于新增、修改、删除的操作，每次都需要写 `db.session.commit()` 进行提交，并且在业务逻辑中，如果触发了异常，还需要调用 `db.session.rollback()` 进行回滚，例如：

```
try:
    student = Student.query.first()
   student.name = "imooc"
   db.session.commit()
except Exception as e:
    db.session.rollback()
```

对于重复性的操作，一般的方式是提供一个工具方法或工具类，所以可以重写 SQLAlchemy 核心类，创建 `auto_commit`，并使用 `contextmanager` 装饰该函数。

```
from contextlib import contextmanager

from flask_sqlalchemy import SQLAlchemy as _SQLAlchemy


class SQLAlchemy(_SQLAlchemy):
    """重写SQLAlchemy核心，实现数据自动提交"""

    @contextmanager
    def auto_commit(self):
        try:
            yield
            self.session.commit()
        except Exception as e:
            self.session.rollback()
            raise e
```

`@contextmanager` 装饰器的原理其实非常简单，它将生成器函数中 `yield` 之前的部分视为上下文的进入操作，而 `yield` 之后的部分视为离开操作。因此，我们可以在 `yield` 之前对数据进行操作（新增、修改、删除），在 `yield` 之后执行 `self.session.commit()` 提交更改。

有了该方法后，对于属于数据库操作，我们可以这样简化一部分代码：

```
with db.auto_commit():
    student = Student.query.first()
   student.name = "imooc"
```

Flask-Migrate 扩展介绍与使用

Flask-Migrate扩展介绍与使用

在前面的课时中，我们使用 `db.create_all()` 将 ORM 模型映射到数据库的表中，但是如果新增了 ORM 模型，或者更新了 ORM 模型，使用 `db.create_all()` 都不会重新创建表或是更新表。

需要先使用 `db.drop_all()` 删除数据库中的所有表之后再调用 `db.create_all()` 才能重新创建，但是这样的话，原来表中的数据就都被删除了，这肯定是不行的，于是就有了数据库迁移。

01. Flask-Migrate 初始化

在开发过程中，随着需求的变化，有可能需要添加或修改表的一些字段，但是原表中的数据不能删除，此时就需要创建新表，并将旧表中的数据迁移至新表中，`Flask-Migrate` 这个扩展就可以在不破坏数据的情况下更新数据库表的结构，并完成数据从旧表到新表的迁移。

安装命令：

```
pip install flask-migrate
```

安装后进行初始化即可：

```
from flask_migrate import Migrate

migrate = Migrate()
migrate.init_app(app, db, directory="internal/migration")
```

02. Flask-Migrate 常见命令

2.1 初始化迁移环境

在开始迁移数据之前，需要先使用 `init` 命令初始化创建一个迁移环境：

```
# 当 flask 的应用入口在项目根目录，且文件名为 app.py，并且实例变量名为 app 时
flask db init

# LLMOps项目运行命令
flask --app app.server.app db init
```

运行好命令后，就会在 `directory` 指定的位置创建一个迁移环境，默认位置为 `./migrations`。

2.2 生成迁移脚本

使用如下命令自动生成迁移脚本：

```
flask --app app.server.app db migrate -m "create_table"
```

其中 `-m "create_table"` 是可选的，意思是为这个迁移写上一个简单的注释。

在生成的迁移脚本中有两个函数：

- `upgrade()`：把迁移中的改动应用到数据库中；
- `downgrade()`：将改动撤销；

注意下，在生产环境中一般不使用 `Flask-Migrate` 迁移，一般人工生成相应的 SQL 执行迁移，如果要使用迁移，一定要仔细检查生成的迁移文件是否符合预期，确认后才可以使用。

2.3 更新及回滚数据库

生成迁移脚本后，可以使用 `upgrade` 命令来更新数据库，如下：

```
flask --app app.server.app db upgrade
```

每一次更新 ORM 模型，都需要执行 `migrate` 命令生成迁移文件，然后才可以使用 `upgrade` 命令将更新同步到数据库中。

如果想要回滚数据库，可以使用 `downgrade` 命令，如下：

```
flask --app app.server.app db downgrade
```

每执行一次命令会向上回滚一个版本，如果想一次性回滚到最原始的版本，即删除所有数据库，可以使用如下命令：

```
flask --app app.server.app db downgrade base
```

如果想回滚到特定的版本，可以在 `downgrade` 后带上特定的版本号，如下：

```
flask --app app.server.app db downgrade 版本号
```