# 标题

标题：数据库与Python交互应用

副标题：探索数据操作的无限可能

# 目录

1. 介绍数据库

2. 数据的操作

3. 介绍python

4. python如何使用数据库

# **介绍数据库**

### 一、数据库的定义

1、数据库是一个有组织的数据集合，通常以电子形式存储在计算机系统中。数据库的基本目的是为用户提供一种高效、可靠且易于访问和管理的数据存储解决方案

1. 数据库是按照数据结构来组织、存储和管理数据的系统，它是一个在计算机中长期存储、结构化、有序、可共享的大量数据集合

2-1、长期存储：

* 数据库的主要功能之一是长期存储数据。
* 数据不会因为程序的结束或计算机的关机而丢失。
* 数据库提供了持久化的存储机制，确保数据的安全性和可靠性

2-2 、结构化

* 数据库中的数据是按照特定的结构和模式进行组织的。
* 这种结构化的方式使得数据的存储和检索更加高效和有序。
* 常见的结构化数据类型包括表格、记录和字段。

2-3、 有序

* 数据库中的数据是有序排列的，便于快速查找和管理。
* 通过索引和排序机制，数据库能够高效地处理查询请求。

**有序的数据存储有助于提高数据的一致性和完整性。**

2-4、可共享的大数据集合

* 数据库是一个包含大量数据的集合，能够满足各种应用场景的需求。
* 这些数据是可以被多个用户或系统同时访问和共享的。
* 数据库提供了并发控制和访问权限管理，确保数据的安全性和一致性。

### 二、数据库的类型（关系型数据库 vs 非关系型数据库）

**1、数据模型**

* **关系型数据库**：基于严格的关系模型，使用表格和列来组织数据。
* **非关系型数据库**：采用灵活的数据模型，如文档、键值对、列族、图形等。

**2、查询语言**

* **关系型数据库**：使用标准化的SQL（Structured Query Language）进行数据操作。
* **非关系型数据库**：通常有自己的查询语言或API，如MongoDB的查询语言。

1. **事务支持**

* **关系型数据库**：支持ACID（原子性、一致性、隔离性、持久性）事务，确保数据完整性
* **非关系型数据库**：通常遵循BASE（基本可用、软状态、最终一致性）模型，事务支持较弱。

1. **扩展性**

* **关系型数据库**：主要通过垂直扩展（增加硬件资源）来提升性能，水平扩展较困难。
* **非关系型数据库**：设计之初就考虑了水平扩展，能够轻松应对大数据和高并发场景。

1. **使用场景**

* **关系型数据库**：适用于需要复杂查询、事务处理和数据一致性的应用，如金融系统、ERP等。
* **非关系型数据库**：适用于实时大数据分析、缓存、日志记录等对扩展性和性能要求高的场景。

1. **数据一致性**

* **关系型数据库**：强一致性模型，确保每次读取都能得到最新的数据。
* **非关系型数据库**：最终一致性模型，可能在短时间内读取到旧数据，但最终会达到一致状态。

1. **实力产品**

* **关系型数据库**：MySQL, Oracle, SQL Server, PostgreSQL
* **非关系型数据库**：MongoDB, Cassandra, Redis, Couchbase

### 三、数据库的重要性及其应用场景

### **数据库的重要性**

* 数据存储与管理：数据库提供了一种结构化的方式来存储大量数据，使得数据的存储和管理变得高效和可靠。
* 数据查询与分析：数据库支持复杂的数据查询和分析功能，使得用户可以快速从海量数据中获取所需信息。
* 数据安全与完整性：数据库通过多种机制来确保数据的安全性和完整性，包括用户身份验证、访问控制、数据加密和数据备份等。
* 支持业务决策：数据库系统在支持业务决策方面也发挥着重要作用，通过数据库，企业可以存储和管理大量业务数据，并通过查询和分析功能从数据中提取有价值的信息。
* 提高工作效率：数据库系统通过提供高效的数据存储、管理、查询和分析功能，大大提高了工作效率。

**数据库的应用场景**

* 电子商务：用于存储和管理商品、订单、客户等数据，支持在线购物和交易。
* 金融领域：用于存储和管理账户、交易、贷款等数据，支持金融交易和业务处理。
* 医疗领域：用于存储和管理患者、病历、药品等数据，支持医疗服务和管理。
* 教育领域：用于存储和管理学生、课程、成绩等数据，支持教育教学和管理。
* 政府和公共服务：用于存储和管理人口、税务、社保等数据，支持公共服务和管理。

### 常见数据库管理系统简介（如 MySQL, PostgreSQL, MongoDB 等）

### **MySQL 数据库简介**

**定义**：MySQL是一个开源的关系型数据库管理系统，使用SQL语言进行数据操作和查询。

**特点**：

* 开源性：免费提供，拥有强大的社区支持和活跃的开发者社区。
* 可靠性和稳定性：经过多年发展，已成为稳定可靠的数据库系统。
* 多线程支持：充分利用CPU资源，提供高性能。
* 跨平台性：支持多种操作系统，如Linux、Windows、Mac OS等。

### **PostgreSQL 数据库简介**

**定义**：PostgreSQL是一个功能强大的开源数据库系统，以可靠性、稳定性和数据一致性著称。**特点**：

* 高度兼容SQL标准：支持子查询、授权读取和可序列化的事务隔离级别。
* 高级功能：包括多版本并发控制（MVCC）、按时间点恢复（PITR）、表空间、异步复制等。
* 可定制性：支持多种编程语言进行存储过程开发，如Java、Perl、Python等。

### **MongoDB 数据库简介**

**定义**：MongoDB是一个基于文档的NoSQL数据库，采用BSON（Binary JSON）格式存储数据。

**特点**：

* 灵活的文档模型：无需预先定义表结构，字段可动态添加。
* 高性能：支持索引、聚合框架等，优化查询效率。
* 高可用性：通过副本集和分片实现数据冗余和水平扩展。

# 数据的操作

### 一、数据的基本概念

#### **1. 数据的定义**

* **数据**：是对客观事件进行记录并可以鉴别的符号，是对客观事物的性质、状态以及相互关系等进行记载的物理符号或这些物理符号的组合。

#### **2. 记录**

* **记录**：在数据库中，记录是指表中的一行数据，它代表了某个实体或事件的具体信息。

#### **3. 字段**

* **字段**：构成记录的每个数据项，是数据库中用于存储特定类型数据的基本单位。每个字段都有其特定的名称、数据类型，以及可能有的约束条件（如唯一性、长度、格式等）。

#### **4. 表**

* **表**：由行和列组成的二维表，是数据库中用于存储和查询数据的基本结构。

#### **5. 数据类型**

* **数据类型**：在计算机科学和计算机编程中，数据类型是数据的属性，它告诉编译器或解释器，程序员打算如何使用数据。常见的数据类型包括实数、整数、布尔型、字符型、日期型等。

#### **6. 数据格式**

* **数据格式**：描述数据保存在文件或记录中的规则，可以是字符形式的文本格式，或二进制数据形式的压缩格式。

### 二、SQL语言简介及常用操作

### **SQL 语言简介**

**定义**：SQL（Structured Query Language）是一种用于管理关系型数据库的标准编程语言，主要用于数据的查询、插入、更新和删除操作

**历史**：SQL由IBM公司研制，并在1986年成为ANSI标准，随后成为国际标准

**特点**：

* **非过程化**：只需描述操作要求，不必描述操作步骤。
* **面向集合**：操作的对象可以是元组的集合。
* **统一语法**：提供多种使用方式，既可以作为自含式语言，也可以嵌入到其他语言中使用。

### **SQL 常用操作**

**SELECT**：用于从表中检索数据。

基本语法：SELECT column1, column2 FROM table\_name WHERE condition;

示例：SELECT \* FROM students WHERE age > 18;

**INSERT**：用于向表中插入新数据。

基本语法：INSERT INTO table\_name (column1, column2, ...) VALUES (value1, value2, ...);

示例：INSERT INTO students (id, name, age) VALUES (1, '张三', 20);

**UPDATE**：用于修改表中的现有数据。

基本语法：UPDATE table\_name SET column1 = value1, column2 = value2 WHERE condition;

示例：UPDATE students SET age = 21 WHERE id = 1;

**DELETE**：用于从表中删除数据。

基本语法：DELETE FROM table\_name WHERE condition;

示例：DELETE FROM students WHERE id = 1;

通过这些点，我们可以看出SQL语言在现代数据库管理中的核心作用，它不仅支持基本的数据操作，还具备强大的查询和数据处理能力。

### 数据库索引和查询优化技巧

#### **1. 索引优化**

* **创建合适的索引**：根据查询条件创建索引，如CREATE INDEX idx\_user\_email ON users(email);
* **联合索引优化**：避免创建多个单列索引，而是创建包含多个字段的联合索引，如CREATE INDEX idx\_users\_search ON users(name, email, phone);
* **索引排序优化**：考虑排序方向的索引，如CREATE INDEX idx\_products\_category\_price ON products(category asc, price desc);
* **前缀索引**：对于长字符串字段，使用前缀索引，如CREATE INDEX idx\_description ON products(description(50));
* **覆盖索引**：创建包含所有查询所需字段的索引，如CREATE INDEX idx\_orders\_covering ON orders(order\_id, status, created\_at, total\_amount);

#### **2. 查询重写**

* **避免使用**SELECT \*：明确指定需要查询的列，减少数据传输量，如SELECT o.order\_id, o.status, u.name, u.email FROM orders o JOIN users u ON o.user\_id = u.id;
* **使用**EXISTS**替代**IN：提高查询效率，如SELECT \* FROM orders WHERE EXISTS (SELECT 1 FROM users WHERE u.id = o.user\_id AND u.status = 'active');
* **使用**UNION ALL**替代**UNION：避免去重操作，提高查询速度，如SELECT order\_id FROM completed\_orders UNION ALL SELECT order\_id FROM pending\_orders;
* **使用**JOIN**替代子查询**：减少查询层级，提高查询效率。

#### **3. 数据库配置优化**

* **调整缓存大小**：根据数据库的内存大小调整缓存设置，如MySQL的innodb\_buffer\_pool\_size。

#### **4. 表结构优化**

* **定期优化表结构**：随着业务需求的变化，定期审查表结构，添加或删除列、拆分或合并表。

#### **5. 分区和分表优化**

* **分区表**：将大表分成多个小表，提高查询效率。

#### **6. 缓存优化**

* **缓存查询结果**：对于频繁被查询但很少被修改的数据，可以考虑将查询结果缓存起来，如使用Redis。

#### **7. 监控和维护**

* **持续监控数据库性能**：使用性能监控工具，如MySQL的Performance Schema或PostgreSQL的pg\_stat\_statements，实时监测数据库的性能指标。

### 数据安全与备份策略

### **一、数据安全性**

**1、访问控制**

* 实施强密码策略
* 多因素认证（MFA）
* 基于角色的访问权限管理

**2、加密技术**

* 数据传输加密（如SSL/TLS）
* 数据存储加密（如AES-256）
* 端到端加密保护

**3、防火墙与入侵检测系统**

* 配置网络防火墙规则
* 使用IDS/IPS监控异常活动

**4、反病毒和恶意软件防护**

* 定期更新防病毒软件
* 进行恶意软件扫描和清理

**5、安全审计与日志记录**

* 记录关键操作日志
* 定期审查和分析日志文件
* 实施合规性检查和审计

### **二、备份策略**

**1、3-2-1备份法则**

* 至少有三份数据副本
* 使用两种不同的存储媒介
* 其中一份副本存放在异地或云端

**2、定期备份计划**

* 设定明确的备份频率和时间点
* 包括全量备份和增量/差异备份

**3、自动化备份流程**

* 利用软件工具自动执行备份任务
* 减少人为错误和提高效率

**4、备份验证与测试**

* 定期检查备份数据的完整性和可用性
* 执行恢复演练以确认流程有效性

**5、灾难恢复计划**

* 制定详细的应急响应方案
* 明确职责分工和时间表
* 准备必要的硬件和软件资源

**6、云备份与服务选择**

* 评估不同云服务提供商的安全性能
* 利用可靠的云存储解决方案进行备份
* 确保遵守相关法律法规和行业标准

# 介绍Python

### Python的特点与优势

**1、简单易学**：Python的语法简洁清晰，强调代码的可读性，使用缩进来表示代码块，非常适合初学者入门

**2、丰富的库和框架**：Python自带庞大的标准库，涵盖文件操作、网络通信、正则表达式、数据库连接等众多功能模块，同时有大量的第三方库可供使用，如NumPy、Pandas、Matplotlib等。

**3、跨平台性**：Python可以在Windows、macOS、Linux等多种操作系统上运行，无需进行大量修改

**4、面向对象编程**：支持面向对象编程范式，使用类和对象的概念来组织和封装代码，提高代码的可重用性和可维护性。

**5、动态类型系统**：Python是动态类型语言，变量的类型在运行时确定，提高了编程的灵活性和便捷性

**6、强大的社区支持**：Python拥有庞大的社区支持，社区成员提供丰富的教程、文档、工具和项目，帮助开发者解决问题、分享经验和创新

### Python 在数据处理和分析中的应用

1、**NumPy**：提供高性能的数值计算功能，包括多维数组对象和用于处理这些数组的函数。

2、**Pandas**：用于数据处理和分析的库，提供DataFrame对象，简化数据的清洗、转换和整理。

3、**Matplotlib**：用于数据可视化，可以创建各种类型的图表和可视化。

4、**SciPy**：用于科学计算，提供数值积分、优化、信号处理等功能。

5、**Scikit-learn**：用于机器学习，提供数据预处理、特征选择和降维等工具。

6、**Statsmodels**：专注于统计建模，提供丰富的统计方法和模型。

7、**Dask**：支持分布式计算和流式处理，适合处理大规模数据集。

### Python编程基础

#### **1. 变量**

* **定义**：变量是存储数据的容器。
* **动态类型**：Python不需要预先声明变量的类型，变量的类型在运行时自动确定。
* **命名规范**：使用小写字母和下划线组合，避免使用保留字和特殊字符。

#### **2. 数据类型**

* **基本数据类型**：
  + **整数（int）**：如 1, 100
  + **浮点数（float）**：如 3.14, -0.001
  + **字符串（str）**：如 "Hello, World!", 'Python'
  + **布尔值（bool）**：True, False
* **复合数据类型**：
  + **列表（list）**：有序集合，如 [1, 2, 3]
  + **元组（tuple）**：有序不可变集合，如 (1, 2, 3)
  + **字典（dict）**：键值对集合，如 {'name': 'Alice', 'age': 25}
  + **集合（set）**：无序不重复元素集合，如 {1, 2, 3}

#### **3. 控制流**

* **条件语句**：
  + 使用 if, elif, else 进行条件判断。
  + 示例：if x > 0: print("Positive")
* **循环语句**：
  + **for 循环**：遍历序列，如 for i in range(5): print(i)
  + **while 循环**：重复执行代码块，直到条件不满足，如 while x < 10: x += 1
* **异常处理**：
  + 使用 try, except, finally 进行异常捕获和处理。
  + 示例：try: result = 10 / 0 except ZeroDivisionError: print("Error")

#### **4. 函数**

* **定义和调用**：使用 def 关键字定义函数，通过函数名调用。
* **参数传递**：支持位置参数、默认参数、可变参数和关键字参数。
* **返回值**：使用 return 语句返回值。

#### **5. 模块和包**

* **模块**：将功能相关的代码组织在一个文件中，通过 import 语句导入使用。
* **包**：包含多个模块的目录，通过 . 进行访问。

#### **6. 文件操作**

* **读写文件**：使用 open 函数打开文件，通过 read, write 等方法进行读写操作。
* **文件路径**：使用 os 模块处理文件路径和目录操作。

### 常用的Python库介绍

#### **1. NumPy**

* **科学计算**：提供高性能的多维数组对象和数学函数。
* **数组操作**：支持大规模多维数组和矩阵运算。
* **线性代数**：包含线性代数、随机数生成和傅里叶变换等功能。

#### **2. Pandas**

* **数据处理**：提供高效的数据结构和数据分析工具。
* **DataFrame**：类似于表格的数据结构，便于数据清洗和处理。
* **数据操作**：支持数据筛选、排序、分组和合并等操作。

#### **3. Matplotlib**

* **数据可视化**：提供丰富的绘图功能，创建各种类型的图表。
* **图表类型**：支持折线图、柱状图、散点图、饼图等。
* **自定义设置**：允许高度自定义图表的样式和布局。

#### **4. SciPy**

* **科学计算**：基于 NumPy，提供更多高级的科学计算功能。
* **优化与积分**：包含优化算法、数值积分和插值方法。
* **信号处理**：支持信号处理和图像处理等功能。

#### **5. Scikit-learn**

* **机器学习**：提供简单高效的机器学习工具。
* **分类与回归**：支持多种分类和回归算法。
* **模型评估**：包含数据预处理、模型选择和评估等功能。

#### **6. TensorFlow**

* **深度学习**：由 Google 开发，用于构建和训练深度学习模型。
* **灵活架构**：支持多种平台和设备，可在 CPU、GPU 和 TPU 上运行。
* **生态系统**：拥有庞大的社区和丰富的扩展库。

#### **7. Keras**

* **深度学习框架**：基于 TensorFlow，提供高级 API 简化模型构建。
* **快速原型设计**：易于搭建和测试新的想法。
* **多后端支持**：可在 TensorFlow、Theano 和 CNTK 上运行。

# Python 如何使用数据库

### 一、Python 连接数据库的方法

#### **1. SQLite**

* **轻量级数据库**：无需单独安装，Python自带SQLite库。
* **文件存储**：数据库存储在单个文件中，便于管理和部署。
* **简单易用**：适合小型应用和学习数据库操作。

#### **2. MySQL**

* **开源关系型数据库**：广泛用于Web应用和企业级应用。
* **连接方式**：使用mysql-connector-python或PyMySQL库进行连接。
* **性能优越**：支持高并发和大数据量处理。

#### **3. PostgreSQL**

* **高级关系型数据库**：支持复杂查询和事务处理。
* **连接方式**：使用psycopg2库进行连接。
* **功能丰富**：支持自定义数据类型和扩展。

#### **4. MongoDB**

* **NoSQL数据库**：文档型数据库，适合存储非结构化数据。
* **连接方式**：使用pymongo库进行连接。
* **灵活高效**：支持水平扩展和高并发读写。

### 二、使用Pyton执行SQL查询和处理结果

#### **1. 安装 MySQL 连接器**

* 使用 mysql-connector-python 库连接 MySQL 数据库。
* 安装命令：pip install mysql-connector-python

#### **2. 建立数据库连接**

* 使用 connect() 方法建立与 MySQL 数据库的连接。
* 示例：

import mysql.connector

*def* connect\_to\_database(*host*, *user*, *password*, *database*, *pool\_size*):

    """

    创建一个到指定数据库的连接池。

    参数:

    - host: 数据库服务器的主机名或IP地址。

    - user: 用于连接数据库的用户名。

    - password: 用户的密码。

    - database: 要连接的数据库名称。

    - pool\_size: 连接池中的连接数量。

    返回:

    无。成功时创建一个连接池，失败时抛出异常。

    """

    try:

        # 使用提供的参数创建一个MySQL连接池

        self.pool = pooling.MySQLConnectionPool(

*pool\_name*=*f*"{host}:{database}",

*pool\_size*=pool\_size,

*host*=host,

*user*=user,

*password*=password,

*database*=database

        )

        # 记录连接池创建成功的信息

        logging.info("MySQL连接池创建成功")

    except Error as e:

        # 如果创建连接池过程中出现错误，记录错误信息并抛出异常

        logging.error(*f*"连接池创建失败: {e}")

        raise

#### **3. 创建游标**

* 使用 cursor() 方法创建游标对象。
* 示例：

cursor = conn.cursor()

#### **4. 执行 SQL 查询**

* 使用游标的 execute() 方法执行 SQL 查询。
* 示例：

*def* fetch\_one(*self*, *table\_name*, *fields*, *condition*):

    # 将字段列表转换为以逗号分隔的字符串

    fields\_str = ', '.join(fields)

    # 构造SQL查询语句

    query = *f*"SELECT {fields\_str} FROM {table\_name} WHERE {condition} LIMIT 1"

    # 执行查询语句

    results = self.execute\_query(query)

    # 如果查询结果不为空，则返回第一个结果，否则返回None

    return results[0] if results else None

#### **5. 处理查询结果**

* 使用 fetchone(), fetchmany(), 或 fetchall() 方法获取查询结果。
  + fetchone(): 获取单条记录。
  + fetchmany(size): 获取指定数量的记录。
  + fetchall(): 获取所有记录。

### ORM（对象关系映射）工具简介

#### **1. 定义与作用**

* **ORM定义**：对象关系映射是一种编程技术，用于将对象模型与关系数据库模型之间进行相互转换。
* **作用**：简化数据库操作，提高开发效率，减少SQL语句的编写。

#### **2. 主要特点**

* **透明性**：开发者无需关心底层数据库的实现细节，通过操作对象即可完成数据库交互。
* **抽象化**：将数据库表映射为类，表中的记录映射为类的实例。
* **跨平台**：支持多种数据库系统，如MySQL、PostgreSQL、SQLite等。

#### **3. 常见ORM工具**

**Django ORM**：

* + Django框架内置的ORM工具。
  + 通过Python类定义模型，自动生成数据库表。
  + 支持数据库迁移和版本控制。

**SQLAlchemy**：

* + 功能强大的开源ORM工具。
  + 支持多种数据库，兼容SQLite、MySQL、PostgreSQL等。
  + 提供高级功能，如事务管理、查询优化和缓存。

**Peewee**：

* + 轻量级ORM工具，简单易用。
  + 支持多种数据库，适合小型项目和快速开发。
  + 允许直接编写原生SQL查询。

**SQLObject**：

* + 另一个流行的开源ORM工具。
  + 提供简洁的API，易于上手。
  + 支持多种数据库系统。

#### **4. ORM的优势**

* **提高开发效率**：减少手动编写SQL语句的工作量。
* **增强代码可读性**：使用面向对象的方式操作数据库。
* **简化数据库迁移**：ORM工具通常提供数据库迁移支持。
* **提高代码可维护性**：将数据库逻辑与业务逻辑分离。

#### **5. 使用场景**

* **Web应用开发**：适用于Django、Flask等Web框架。
* **桌面应用开发**：如PyQt、Tkinter等桌面应用。
* **数据处理与分析**：如Pandas结合ORM进行数据操作

from flask import Flask, render\_template, request, redirect, url\_for, flash, jsonify

from db import get\_connection, shutdown  # 使用 \_\_init\_\_.py 中定义的函数

app = Flask(\_\_name\_\_)

app.secret\_key = 'your\_secret\_key'  # 设置密钥用于flash消息

# 创建全局数据库连接实例

db\_connection = get\_connection()

@app.route('/')

*def* index():

    students = db\_connection.fetch\_all('students', ['\*'], 'is\_deleted = 0')  # 查询未删除的学生

    # students = db\_connection.fetch\_all('students', ['\*'], '1=1')  # 查询所有学生

    return render\_template('index.html', *students*=students)

@app.route('/add\_student', *methods*=['GET', 'POST'])

*def* add\_student():

    # 判断请求方法是否为POST

    if request.method == 'POST':

        # 获取表单中的姓名

        name = request.form['name']

        # 获取表单中的年龄，并转换为整数

        age = *int*(request.form['age'])

        # 获取表单中的年级

        grade = request.form['grade']

        # 获取表单中的地址

        address = request.form['address']

        # 定义要插入的列名

        columns = ['name', 'age', 'grade', 'address']

        # 定义要插入的值

        values = [name, age, grade, address]

        # 调用数据库连接对象的insert\_record方法，插入学生信息

        db\_connection.insert\_record('students', columns, values)

        # 闪现提示信息

        flash('学生信息已添加')

        # 返回JSON响应

        return jsonify({'status': 'success'})

    else:

        # 如果请求方法不是POST，则返回添加学生信息的HTML页面

        return render\_template('add\_student.html')

@app.route('/edit/<int:id>', *methods*=['GET', 'POST'])

*def* edit\_student(*id*):

    # 检查请求方法是否为POST

    if request.method == 'POST':

        # 获取表单数据并转换为字典

        set\_values = {

            'name': request.form['name'],  # 学生姓名

            'age': *int*(request.form['age']),  # 学生年龄

            'grade': request.form['grade'],  # 学生年级

            'address': request.form['address']  # 学生地址

        }

        # 更新数据库记录

        db\_connection.update\_record('students', set\_values, *f*"id = {id}")

        # 显示更新成功的提示信息

        flash('学生信息已更新')

        # 返回JSON格式的响应，表示操作成功

        return jsonify({'status': 'success'})

    else:

        # 获取指定ID的学生信息

        student = db\_connection.fetch\_one('students', ['id', 'name', 'age', 'grade', 'address'], *f*"id = {id}")

        # 渲染编辑学生信息的HTML模板

        return render\_template('edit\_student.html', *student*=student)

@app.route('/delete/<int:id>', *methods*=['DELETE'])

*def* delete\_student(*id*):

    # 使用update\_record方法将'students'表中id对应记录的'is\_deleted'字段设置为1，实现逻辑删除

    db\_connection.update\_record('students', {'is\_deleted': 1}, *f*"id = {id}")  # 逻辑删除

    # 使用delete\_record方法从'students'表中删除id对应的记录，实现物理删除

    # db\_connection.delete\_record('students', f"id = {id}") # 物理删除

    return jsonify({'status': 'success'})

if \_\_name\_\_ == '\_\_main\_\_':

    try:

        app.run(*debug*=True)

    finally:

        shutdown()  # 确保关闭所有数据库连接

db.connection.py

import logging

import mysql.connector

from mysql.connector import Error, pooling

# 设置日志记录

logging.basicConfig(*level*=logging.INFO, *format*='%(asctime)s - %(levelname)s - %(message)s')

*class* DatabaseConnection:

*def* \_\_init\_\_(*self*, *host*, *user*, *password*, *database*, *pool\_size*=5):

    """

    初始化数据库连接池

    参数:

    - host: 数据库主机地址

    - user: 数据库用户名

    - password: 数据库密码

    - database: 要连接的数据库名称

    - pool\_size: 连接池的大小，默认为5

    该方法在实例化数据库连接池对象时被调用，负责创建一个数据库连接池并进行初始化配置

    """

    # 初始化连接池属性为None，后续将通过connect方法进行设置

    self.pool = None

    # 调用connect方法创建数据库连接池

    self.connect(host, user, password, database, pool\_size)

*def* connect(*self*, *host*, *user*, *password*, *database*, *pool\_size*):

    """

    建立数据库连接

    参数:

    host (str): 数据库服务器地址

    user (str): 数据库用户名

    password (str): 数据库密码

    database (str): 数据库名

    pool\_size (int): 连接池大小

    返回:

    无

    """

    try:

        # 尝试初始化数据库连接

*def* \_\_init\_\_(*self*, *host*, *user*, *password*, *database*, *pool\_size*=5):

            # 初始化数据库连接参数

            self.host = host

            self.user = user

            self.password = password

            self.database = database

            self.pool\_size = pool\_size

            # 初始化数据库连接池

            self.pool = self.create\_pool()

    except *Exception* as e:

        # 处理连接过程中可能发生的异常

        print(*f*"连接数据库失败: {e}")

    # 初始化连接池属性为None，后续将通过connect方法进行设置

    self.pool = None

    # 调用connect方法创建数据库连接池

# 连接到数据库并创建一个MySQL连接池

*def* connect\_to\_database(*host*, *user*, *password*, *database*, *pool\_size*):

    """

    创建一个到指定数据库的连接池。

    参数:

    - host: 数据库服务器的主机名或IP地址。

    - user: 用于连接数据库的用户名。

    - password: 用户的密码。

    - database: 要连接的数据库名称。

    - pool\_size: 连接池中的连接数量。

    返回:

    无。成功时创建一个连接池，失败时抛出异常。

    """

    try:

        # 使用提供的参数创建一个MySQL连接池

        self.pool = pooling.MySQLConnectionPool(

*pool\_name*=*f*"{host}:{database}",

*pool\_size*=pool\_size,

*host*=host,

*user*=user,

*password*=password,

*database*=database

        )

        # 记录连接池创建成功的信息

        logging.info("MySQL连接池创建成功")

    except Error as e:

        # 如果创建连接池过程中出现错误，记录错误信息并抛出异常

        logging.error(*f*"连接池创建失败: {e}")

        raise

*def* get\_connection(*self*):

        try:

            return self.pool.get\_connection()

        except Error as e:

            logging.error(*f*"获取连接失败: {e}")

            raise

    @*staticmethod*

*def* is\_connection\_valid(*connection*):

        """

        检查数据库连接是否仍然有效。

        """

        try:

            return connection.is\_connected()

        except Error:

            return False

    @*staticmethod*

*def* reconnect\_if\_needed(*connection*):

        """

        如果连接无效，则尝试重新连接。

        """

        if not DatabaseConnection.is\_connection\_valid(connection):

            logging.info("连接无效，尝试重新连接...")

            connection.reconnect(*attempts*=3, *delay*=2)  # 尝试重连3次，每次延迟2秒

*def* execute\_query(*self*, *query*, *params*=None, *commit*=False):

    """

    执行SQL查询

    :param query: 待执行的SQL查询语句

    :param params: 查询参数，用于替换SQL语句中的占位符

    :param commit: 是否需要提交事务，默认为False

    :return: 如果不需要提交事务，则返回查询结果；否则返回None

    """

    # 获取数据库连接

    connection = self.get\_connection()

    cursor = None

    try:

        # 创建游标，设置dictionary=True以获取字典形式的查询结果

        cursor = connection.cursor(*dictionary*=True)

        # 执行SQL查询

        cursor.execute(query, params)

        # 根据commit参数决定是否提交事务

        if commit:

            connection.commit()

        else:

            # 如果有查询结果，则获取所有行；否则返回None

            result = cursor.fetchall() if cursor.with\_rows else None

            return result

    except Error as e:

        # 记录错误日志

        logging.error(*f*"执行查询失败: {e}")

        # 如果是事务操作，执行回滚

        if commit:

            connection.rollback()

        # 重新抛出异常

        raise

    finally:

        # 确保游标被关闭

        if cursor:

            cursor.close()

        # 关闭数据库连接

        connection.close()

*def* insert\_record(*self*, *table\_name*, *columns*, *values*):

    """

    向指定的数据库表中插入一条记录。

    该方法根据提供的表名、列名列表和值列表来构造并执行一个INSERT SQL语句。

    参数:

    - table\_name: 字符串，表示要插入记录的数据库表名。

    - columns: 列表，包含了需要插入数据的列名。

    - values: 列表，包含了与columns参数中列名对应的数据值。

    该方法没有返回值，但会在数据库中插入一条新记录。

    """

    # 生成SQL语句中的占位符部分，每个列对应一个占位符(%s)。

    placeholders = ', '.join(['%s'] \* len(columns))

    # 将列名列表转换为字符串，用于构造SQL语句。

    columns\_str = ', '.join(columns)

    # 构造完整的INSERT SQL语句。

    query = *f*"INSERT INTO {table\_name} ({columns\_str}) VALUES ({placeholders})"

    # 将values列表转换为元组，作为SQL执行的参数。

    params = *tuple*(values)

    # 执行SQL语句，对数据库进行插入操作，并提交事务。

    self.execute\_query(query, params, *commit*=True)

*def* update\_record(*self*, *table\_name*, *set\_values*, *where\_condition*):

    # 构造 SET 子句，将 set\_values 中的每个键值对转换为 "key = %s" 的形式，并用逗号连接

    set\_clause = ', '.join([*f*"{key} = %s" for key in set\_values.keys()])

    # 构造完整的 SQL 更新语句

    query = *f*"UPDATE {table\_name} SET {set\_clause} WHERE {where\_condition}"

    # 将 set\_values 中的值提取出来，形成参数列表

    params = *list*(set\_values.values())

    # 执行 SQL 更新语句，并提交事务

    self.execute\_query(query, params, *commit*=True)

*def* delete\_record(*self*, *table\_name*, *where\_condition*):

    # 构建删除语句

    query = *f*"DELETE FROM {table\_name} WHERE {where\_condition}"

    # 执行删除语句并提交更改

    self.execute\_query(query, *commit*=True)

*def* fetch\_all(*self*, *table\_name*, *fields*, *condition*):

    # 将字段列表转换为以逗号分隔的字符串

    fields\_str = ', '.join(fields)

    # 构建SQL查询语句

    query = *f*"SELECT {fields\_str} FROM {table\_name} WHERE {condition}"

    # 执行查询并返回结果

    return self.execute\_query(query)

*def* fetch\_one(*self*, *table\_name*, *fields*, *condition*):

    # 将字段列表转换为以逗号分隔的字符串

    fields\_str = ', '.join(fields)

    # 构造SQL查询语句

    query = *f*"SELECT {fields\_str} FROM {table\_name} WHERE {condition} LIMIT 1"

    # 执行查询语句

    results = self.execute\_query(query)

    # 如果查询结果不为空，则返回第一个结果，否则返回None

    return results[0] if results else None

*def* create\_db\_connection():

    # 配置数据库连接信息

    config = {

        'host': 'localhost',  # 数据库主机名

        'user': 'root',        # 数据库用户名

        'password': 'wzj00245',# 数据库密码

        'database': 'simsdemo'  # 数据库名称

    }

    # 返回数据库连接对象

    return DatabaseConnection(\*\*config)

*def* close\_connections():

    # 在这个版本中，每个连接都是从连接池获取并在使用后立即关闭，因此这里不需要显式地关闭所有连接。

    pass