Python Web开发项目教材（Flask版）

第5章 数据库操作（课堂笔记）

# 在开发Web程序时，绝大多数的网页中动态加载的数据通常会存储在数据库中，这样做的目的是将页面层面和数据层面的逻辑进行分离，也就是说涉及页面展示的逻辑交由模板文件处理，涉及数据层面的逻辑交由数据库处理。Flask中提供了一个扩展包Flask-SQLAlchemy，使用该扩展包可以轻松地对数据库进行操作。本章先为大家介绍数据库，再针对扩展包Flask-SQLAlchemy的相关内容进行介绍。

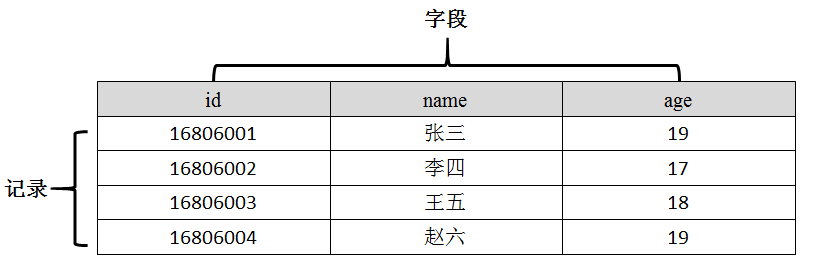
# 数据库概述

数据库是按照一定的数据结构组织、存储和管理数据的仓库，它可以被看作电子化的文件柜——存储文件的处所，用户可以对文件中的数据进行增加、删除、修改、查找等操作。值得一提的是，这里所说的数据不仅包括普通意义上的数字，还包括文字、图像、声音等。

根据存储数据时所用数据模型的不同，当今互联网中的数据库主要分成两大类，分别是关系型数据库和非关系型数据库。

关系型数据库是指采用关系模型（即二维表格模型）组织数据的数据库系统，它主要由数据库、数据表、记录和字段组成。

* 数据库：数据表的集合，可以包含一个或多个数据表。
* 数据表：记录的集合。
* 记录：由若干个字段组成，每条记录相当于表中的一行数据。
* 字段：每个字段相当于表中的一列数据。



关系型数据库历经了几十年的发展，技术已经非常成熟，这类数据库具有容易理解、操作简单、易于维护的特点，被广泛应用到各行业的数据管理工作中。目前主流的关系型数据库有Oracle、MySQL、IBM Db2、PostgreSQL、SQL Server、Microsoft Access等，其中使用较多的有Oracle和MySQL数据库。

# 安装Flask-SQLAlchemy（操作数据库的准备工作）

### 2.1 MySQL安装

本书使用的数据库是MySQL ,因此每个同学请在电脑上安装MySQL服务器。

### 2.2 Python操作MySQL驱动

Flask要操作数据库，必须要先安装Python操作MySQL的驱动，在Python中，有很多MySQL驱动包，我们采用pymysql，纯python实现的一个驱动，可以和python代码无缝衔接。

pip install pymysql

### 2.3 Flask-SQLAlchemy

在FLask中，我们很少会使用pymysql直接写原生SQL语句去操作数据库，更多的是通过SQLAlchemy提供的ORM技术，类似于操作普通Python对象一样，实现对数据库的增、删、改、查操作。而Flask-SQLAlchemy是对SQLAlchemy的饿一个封装，这使得Flask-SQLAlchemy依赖SQLAlchemy，所以只要安装了Flask-SQLAlchemy，SQLAlchemy会自动安装。安装命令如下

pip install flask-sqlalchemy

在Web程序中开发人员若使用原生SQL语句操作数据库，主要会存在以下两个问题：

（1）过多的SQL语句会降低代码的易读性，另外也容易出现诸如SQL注入（一种网络攻击方式，它利用开发人员编写SQL语句时的疏忽，使用SQL语句实现无账号登录甚至篡改数据库）等安全问题。

（2）开发人员开发时通常会使用SQLite数据库，而在部署时会切换到诸如MySQL等更为健壮的数据库，由于不同数据库需要用到不同的Python库，所以切换数据库就需要对代码中使用的Python库进行同步修改，增加了一定的工作量。

Python中引入了ORM技术。ORM的全称为Object Relational Mapping，表示对象关系映射，它是一种解决面向对象与关系数据库存在的互不匹配现象的技术，用于实现面向对象编程语言中模型对象到关系数据库数据的映射。

对于Python语言来说，ORM会将底层的SQL语句操作的数据实体转化成Python对象，这样一来，我们无需了解 SQL语句的编写规则，通过Python代码即可完成数据库操作。ORM 主要实现了以下三种映射关系。

数据表→Python类。

字段（列）→类属性。

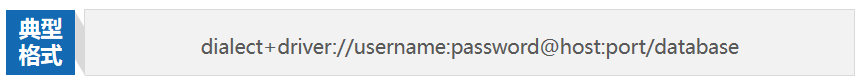
记录（行）→ 类实例。

# 使用Flask-SQLAlchemy操作MySQL

### 3.1连接MySQL数据库

在操作数据库之前，我们需要先建立Flask程序与数据库的连接，这样才能让Flask程序访问数据库，并进一步对数据库中的数据进行操作。Flask 为Flask-SQLAlchemy扩展包提供了一个配置项SQLALCHEMY\_DATABASE\_URI，该配置项用于指定数据库的连接，它的值是一个有着特殊格式的URI，URI涵盖了连接数据库所需要的全部信息，包括用户名、密码、主机名、数据库名称以及用于额外配置的可选关键字参数。

URI的典型格式如下所示。



* dialect+driver：表示数据库类型和驱动程序。数据库类型的取值可以为postgresql（PostgreSQL 数据库）、mysql（MySQL数据库）、oracle（Oracle数据库）、sqlite（SQLite数据库）等。如果未指定驱动程序，则说明选择默认的驱动程序，这时可以省略加号。
* username：表示数据库的用户名。
* password：表示数据库的密码。
* host：表示主机地址。
* port：表示端口号。
* database：表示连接的数据库名。

常见数据库的URI

|  |  |
| --- | --- |
| 数据库 | URL |
| PostgreSQL | postgresql://root:123@localhost/flask\_data |
| MySQL | mysql://root:123@localhost/flask\_data |
| Oracle | oracle://root:123@127.0.0.1:1521/flask\_data |
| SQLite（Windows平台） | sqlite:///C:\\absolute\\path\\to\\foo.db |
| SQLite（Unix/Mac平台） | sqlite:////absolute/path/to/foo.db |

通过一个示例演示如何在Flask程序中连接MySQL数据库。创建一个名称为Chapter05的项目，在该项目中新建app.py文件，并在app.py文件中编写连接MySQL数据库的代码。

from flask import Flask

from flask\_sqlalchemy import SQLAlchemy

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

# 通过URI连接数据库

app.config['SQLALCHEMY\_DATABASE\_URI']='mysql+pymysql://root:123456@localhost/flask\_data'

# 动态追踪数据库的修改，不建议开启

app.config['SQLALCHEMY\_TRACK\_MODIFICATIONS'] = False

# 在app.config中配置好数据库连接的信息

# 然后使用SQLAlchemy(app)创建一个db对象

# SQLAlchemy（app）会自动读取app.config中连接数据库的信息

db=SQLAlchemy(app)

测试连接是否成功，在程序中添加如下代码：

with app.app\_context():

with db.engine.connect() as conn:

rs = conn.execute('select 1')

print(rs.fetchone()) #控制台输出 (1,)

如果报错解决方法如下：

1. 错误 AttributeError: module 'time' has no attribute 'clock'

把 time.clock() 修改成 time.perf\_counter()

1. 错误AttributeError: ‘sqlalchemy.cimmutabledict.immutabledict‘ object has no attribute ‘setdefault‘

pip uninstall sqlalchemy

Pip install sqlalchemy == 1.3

### 3.2 定义模型

对象关系映射（object relationship mapping，简称ORM）是一种可以用Python面向对象的方式来操作关系数据库的技术，具有可以映射到数据库表能力的Python类我们称之为ORM模型。一个ORM模型与数据库中的一个表相对应，ORM模型中的每个类属性分别对应表中的每个字段，ORM模型的每个实例对象对应表中的每条记录。

Flask中的模型以Python类的形式进行定义，所有的模型类都需要继承Flask-SQLAlchemy提供的基类db.Model，通常保存在Flask程序的model.py文件中。Flask-SQLAlchemy会按照ORM的映射关系将模型类转换成数据表，数据表的名称具体分成以下两种情况：

（1）若模型类中包含类属性\_\_tablename\_\_，则会将类属性\_\_tablename\_\_的值作为数据表的名称。

（2）若模型类中没有包含类属性\_\_tablename\_\_，则会将模型类的名称按照一定的规则转换成数据表的名称。

转换的规则主要有两种情况：若模型类的名称是一个单词，则会将所有字母转换为小写的单词作为数据表的名称，例如，模型类User对应的数据表为user；若模型类的名称是多个单词，则会将所有字母转换为小写，以下画线连接的多个单词作为数据表的名称，例如，模型类MyUser对应的数据表为my\_user。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 列名 | 含义 | 数据类型 | 长度 | 能否取空值 | 备注 |
| id | 用户编号 | int |  | no | 主键，自动增长 |
| username | 用户名 | varchar | 20 | yes | 唯一键 |
| password | 密码 | varchar | 30 | no |  |
| email | 邮箱 | varchar | 100 | yes |  |

定义一个表示用户的模型类User，具体代码如下所示。

class User(db.Model):

\_\_tablename\_\_='tb\_user'

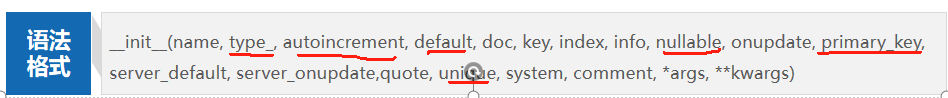
id = db.Column(db.Integer, primary\_key=True)

username = db.Column(db.String(80), unique=True, nullable=False)

email = db.Column(db.String(120), unique=True, nullable=False)

在上述代码中，定义了一个继承自db.Model的模型类User，User类中包含id、username和email共3个属性，每个属性的值是db.Column类的对象。db.Column类封装了字段的相关属性或方法。

db.Column类构造方法的声明如下所示。

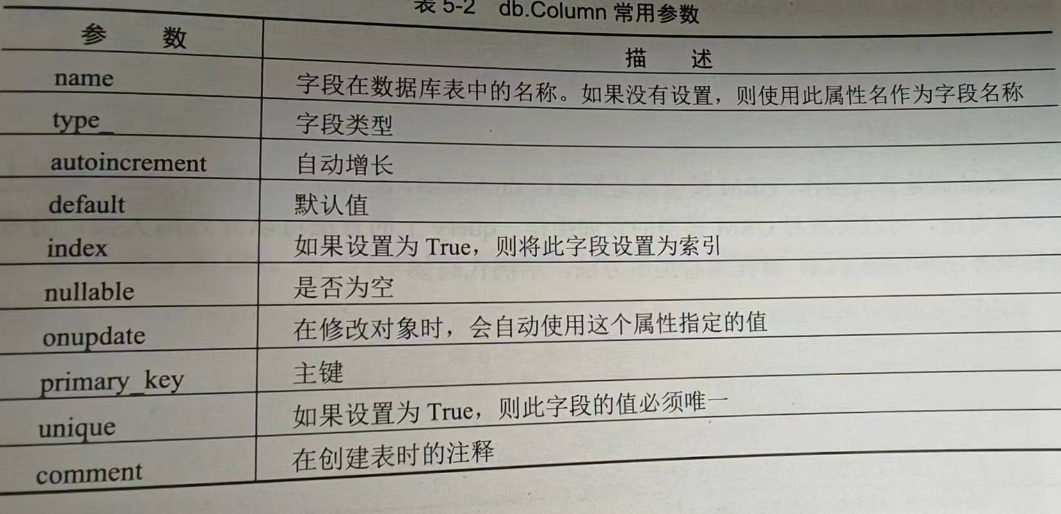


* name：表示数据表中此列的名称。若省略，默认使用类属性的名称。
* type\_：表示字段的类型。若该参数的值为None或省略，则将使用默认类型NullType，表示未知类型。
* default：表示为字段设置默认值。
* index：表示是否为字段创建索引。
* nullable：确定字段的值是否为空，若设置为False，则会在为此列生成DDL时加上NOT NULL语句；若设置为True，通常不会生成任何内容。
* primary\_key：表示是否将字段设置为主键，若设为True，则会将此列标记为主键列。多个列可以设置此标志以指定复合主键。
* unique：表示该字段是否具有唯一约束，若设为True，则该字段将不允许出现重复值。
* \*args：其他位置参数，该参数的值可以为Constraint（表级SQL约束）、ForeignKey（外键）、ColumnDefault、Sequence和Computed Identity类实例。

db.Column类构造方法的type\_参数用于指定字段的类型，比如User类中的Integer和String都属于字段类型。常用的字段类型如下表所示。

|  |  |
| --- | --- |
| 字段类型 | 说明 |
| Integer | 整数 |
| String(size) | 字符串，可通过size设置最大长度 |
| Text | 较长的Unicode文本 |
| DateTime | 日期和时间，存储Python的 datetime对象 |
| Float | 浮点数 |
| Boolean | 布尔值 |
| PickleType | 存储Pickle列化的Python对象 |
| LargeBinary | 存储任意二进制数据 |

字段在数据库中的约束，都是通过db.Column上的参数实现的。db.Column常用参数如下表



### 3.3 创建数据表

定义了模型类以后，如果希望根据模型类生成一个对应的数据表，这时我们可以通过db对象调用create\_all()方法实现。以定义的模型类User为例，演示如何通过create\_all()方法创建数据表。在命令行窗口中进入虚拟环境，输入flask shell命令启用Flask Shell工具，在该工具中输入创建数据表的命令，具体命令如下所示。

(flask\_env) E:\PythonProject\测试>flask shell

Python 3.8.2 (tags/v3.8.2:7b3ab59, Feb 25 2020, 23:03:10) [MSC v.1916 64 bit (AMD64)] on win32

App: app [production]

Instance: E:\PythonProject\Chapter05\instance

>>> from app import db

>>> db.create\_all()

另外一种方式，在app.py中添加如下代码：

with app.app\_context():

db.create\_all()

若希望删除数据表，则可以通过db对象调用drop\_all()方法实现。

# 数据操作

### 4.1 增加数据

使用ORM创建一条数据非常简单，先使用ORM模型创建一个对象，然后添加到会话中，再进行commit操作即可，示例代码如下：



在Flask-SQLAlchemy中，db.session提供了增加数据的add()和add\_all()方法，其中add()方法用于向数据库中增加一条记录，add\_all()方法用于向数据库中增加多条记录。

需要说明的是，创建User类的对象时并没有传入id字段的值，这是因为主键由SQLAlchemy管理。创建User类对象后会将这些对象作为临时对象，直至程序提交至数据库会话后才会将这些对象转换为记录写入到数据库中，并且自动获得id字段的值。

### 4.2 查询数据

Flask-SQLAlchemy的Model类中提供了一个query属性，模型对象通过访问query属性可以获取一个查询对象（Query类实例），该对象中提供了一些过滤方法和查询方法对查询结果进行过滤和筛选，以便精准地查找。



在上述格式中，过滤方法和查询方法是可选的，由于使用过滤方法查询后会返回一个新的查询对象，所以过滤方法和查询方法可以叠加使用，另外也可以单独使用查询方法。

查询对象的查询方法如下表所示。

|  |  |
| --- | --- |
| 方法 | 说明 |
| first() | 返回查找到的第一条记录，若没有找到，则返回None |
| first\_or\_404() | 返回查找到的第一条记录，若没有找到，则返回404错误响应 |
| get() | 返回指定主键值对应的记录，若没有找到，则返回None |
| get\_or\_404() | 返回指定主键值对应的记录，若没有找到，则返回404错误响应 |
| count() | 返回查找到的记录数量 |
| all() | 以列表形式返回查找到的所有记录 |
| paginate() | 返回一个Paginate类的对象，用于对记录进行分页 |

使用查询方法，示例代码如下：



查询对象的过滤方法如下表所示。

|  |  |
| --- | --- |
| 方法 | 说明 |
| filter() | 根据指定的规则过滤记录，返回新产生的查询对象 |
| filter\_by() | 以关键字形式根据指定的规则过滤记录，返回新产生的查询对象 |
| order\_by() | 根据指定条件对原始查询对象进行排序，返回新产生的查询对象 |
| limit() | 根据指定的值限制原始查询对象返回的结果数量，返回新产生的查询对象 |
| offset() | 根据指定的值偏移原始查询对象，返回新产生的查询对象 |
| group\_by() | 根据指定的条件对记录进行分组，返回新产生的查询对象 |
| with\_entities() | 根据指定实体替换查询列表，返回新产生的查询对象 |

表中所有方法都会对原始查询对象进行过滤操作，并将过滤后的结果生成一个新的查询对象。

# 查询数据

@app.route('/query/')

def query():

# 查询姓名叫小明的信息

users=User.query.filter(User.username=='小明').all()

for user in users:

print(str(user.id)+','+user.username+","+user.email)

return '查询数据成功'

其中，filter()方法是最基础的过滤方法，该方法可以传入包含!=和==操作符的表达式，也可以传入使用了查询操作符的表达式，常用的查询操作符包括LIKE、IN、NOT IN、IS NULL、IS NOT NULL、AND和OR。

1. like:模糊查询，使用方式与SQL语句中的like类似，可以在搜索字符左右两边添加%来匹配任意字符。,示例代码如下：

users=User.query.filter(User.username.like('%小%')).all()

1. in:判断值是否在指定数据集中，如果是就提取，否则就不提取。为了不与Python中的in关键字混淆，在in后面加下划线，实际的方法名为in\_，示例代码如下：

users=User.query.filter(User.username.in\_(['小明','小红'])).all()

1. not in:作用与in相反。其使用方式是在in\_方法所在代码表达式前添加（~）,示例代码如下：

users=User.query.filter(~User.username.in\_(['小明','小红'])).all()

1. is null:判断值是否为空，如果为空就提取，否则不提取。通过方法is\_方法实现，示例代码如下：

users=User.query.filter(User.username.is\_(None))

1. Is not null:作用与is null相反，实际的方法名为isnot,示例代码如下：

users=User.query.filter(User.username.isnot(None))

1. And:用于同时满足多条件的查询，实际的方法名为and\_,示例代码如下：

from sqlalchemy import and\_

users=User.query.filter(and\_(User.username.in\_(['小明','小红']),User.id<2))

1. Or：用于满足一个或多个条件的查询，实际的方法名为or\_,示例代码如下：

### 4.3 更新数据

更新数据的方式比较简单，它只需要为模型类的字段重新赋值便可以将字段原先的值进行修改，修改完成后需要调用commit()方法提交至数据库。

@app.route('/update/')

def update():

# 返回主键值2对应的记录

result = User.query.get(2)

print(result.username)

# 将username的值修改为"小兰"

result.username = "小兰"

db.session.commit()

return '更新数据完成'

针对修改多条数据的情况，则是通过调用filter方法获取BaseQuery对象，然后再调用update方法，实现批量修改的。

@app.route('/update/')

def update():

User.query.filter(User.username.in\_(['小明','小红'])).update({'email':'999@qq.com'},synchronize\_session=False)

db.session.commit()

return '更新数据完成'

### 4.4 删除数据

删除数据可以使用数据库会话提供的delete()方法，删除完成后同样需要调用commit()方法提交至数据库。

例如，将数据库flask\_data中的id值为3的记录直接删除，具体代码如下所示。

@app.route("/delete/")

def delete():

# 返回主键值3对应的记录

result = User.query.get(3)

print(result)

db.session.delete(result)

db.session.commit()

return "删除数据完成"

删除多条数据的操作方式类似更新多条数据，通过BaseQuery的delete方法即可实现

@app.route('/delete/')

def delete():

User.query.filter(User.username.in\_(['小明','小红'])).delete(synchronize\_session=False)

db.session.commit()

return '删除数据完成'