浙江理工大学
ZHEJIANG SCI-TECH UNIVERSITY

数据库

《Web应用开发》 / 任课教师：罗志一

计算机科学与技术学院

数据库

- 数据库是大多数动态Web程序的基础设施，只要你想把数据存储下来，就离不开数据库。
- 我们这里提及的数据库指的是有存储数据的单个或多个文件组成的集合，它是一种容器，可以类比为文件柜。而人们通常使用数据库来表示操作数据库的软件，这类管理数据库的软件被称为数据库管理系统，常见的数据库管理系统有MySQL、PostgreSQL、SQLite、MongoDB等。
- 这节课我们来学习如何给Flask程序添加数据库支持，即学习如何在Python中使用这些数据库管理系统来对数据库进行管理和操作。

概览 Overview

- 数据库的分类
- 使用Flask-SQLAlchemy管理数据库
- 数据库操作

数据库的分类

数据库的分类

- 数据库一般分为两种，SQL（Structured Query Language，结构化查询语言）数据库和NoSQL（Not Only SQL，泛指非关系型）数据库。

SQL数据库

- SQL数据库指关系型数据库，常用的SQL数据库管理系统主要包括SQL Server、Oracle、MySQL、PostgreSQL、SQLite等。关系型数据库使用表来定义数据对象，不同的表之间使用关系连接。
- 下表是一个身份信息表的示例。

关系型数据库示例

id	name	Gender	occupation
1	Nick	Male	Journalist
2	Amy	Female	Writer

SQL数据库

- 在SQL数据库中，每一行代表一条记录，每条记录又由不同的列组成。在存储数据前，需要预先定义表模式，以定义表的结构并限定列的输入数据类型。

一些基本概念

概　念	含　　义
表（table）	存储数据的特定结构
模式（schema）	定义表的结构信息
列 / 字段 （column / field）	表中的列，存储一系列特定的数据，列组成表
行 / 记录 （row / record）	表中的行，代表一条记录
标量 （scalar）	指的是单一数据，与之相对的是集合（collection）

NoSQL数据库

- NoSQL最初指No SQL或No Relational，现在NoSQL社区一般会理解为Not Only SQL。NoSQL数据库泛指不使用传统关系型数据库中的表格形式的数据库。近年来，NoSQL数据库越来越流行，被大量应用在实时Web程序和大型程序中。
- 与传统的SQL数据库相比，它在速度和可扩展性方面有很大的优势，除此之外还拥有无模式、分布式、水平伸缩等特点。

NoSQL数据库

- 文档存储
  - 文档存储是NoSQL数据库中最流行的种类，它可以作为主数据库使用。文档存储使用的文档类似SQL数据库中的记录，文档使用类JSON格式来表示数据。常见的文档存储数据库管理系统有MongoDB、CouchDB等。

id	name	Gender	occupation
1	Nick	Male	Journalist
2	Amy	Female	Writer

```
{
    id: 1,
    name: "Nick",
    gender: "Male",
    occupation: "Journalist"
}
```

NoSQL数据库

- 键值对存储
  - 键值对存储再形态上类似Python中的字典，通过键来存取数据，在读取上非常快，通常用来存储临时内容，作为缓存使用。常见的键值对数据库管理系统有Redis、Riak等，其中Redis不仅可以管理键值对数据库，还可以作为缓存后端（cache backend）和消息代理（mesasge broker）。

SQL v.s. NoSQL

- NoSQL数据库不需要定义表和列等结构，也不限定存储的数据格式，在存储方式上比较灵活，在特定的场景下效率更高。SQL数据库稍显复杂，但不容易出错，能够适应大部分的应用场景。
- 两种数据库各有优势，也各有擅长的领域。两者并不是对立的，我们需要根据使用场景选择合适的数据库类型。大型项目通常会同时需要多种数据库，比如使用MySQL作为主数据库存储用户资料和文章，使用Redis（键值对型数据库）缓存数据，使用MongoDB（文档型数据库）存储实时消息。

ORM

- 在Web里使用原生SQL语句操作数据库主要存在下面两类问题：
  - 手动编写SQL语句比较乏味，而且视图函数中加入太多SQL语句会降低代码的易读性。另外，还会出现安全问题，比如SQL注入。
  - 常见的开发模式是在开发时使用简单的SQLite，而在部署时切换到MySQL等更健壮的数据库管理系统。但是对于不同的数据库管理系统，我们需要使用不同的Python接口库，这让数据库管理系统的切换变得不太容易。
- 使用ORM可以很大程度上解决上述问题。它会自动帮你处理查询参数的转义，从而避免SQL注入的发生。另外，它为不同的数据库管理系统提供统一的接口，让切换工作变得非常简单。ORM扮演翻译的角色，将Python语言转换为数据库管理系统能读懂的SQL指令，让我们能够使用Python来操控数据库。

ORM

- ORM把底层的SQL数据实体转化成高层的Python对象，这样一来，你甚至不需要了解SQL，只需要通过Python代码即可完成数据库操作，ORM主要实现了三层映射关系：
  - 表 -> Python类
  - 字段（列） -> 类属性
  - 记录（行） -> 类实例

ORM

- 比如，我们要创建一个contacts表来存储留言，其中包含用户名称和电话号码两个字段。在SQL中，下面的代码用来创建这个表：

```
CREATE TABLE contacts (
    name varchar(100) NOT NULL,
    phone_number varchar(32)
) ;
```

- 如果使用ORM，我们可以使用类似下面的Python类来定义这个表：

```
from foo_orm import Model, Column, String

class Contact(Model):
    __tablename__ = "contacts"
    name = Column(String(100), nullable=False)
    phone_number = Column(String(32))
```

ORM

- 要向表中插入一条记录，需要使用下面的SQL语句：

```
INSERT INTO contacts (name, phone_number)
VALUES('Grey Li', '12345678') ;
```

- 如果使用ORM，只需要创建一个Contact类的实例，传入对应的参数表示各个列的数据即可。

```
contact = Contact(name='Grey Li', phone_number='12345678')
```

使用Flask-SQLAlchemy管理数据库

Flask-SQLAlchemy扩展

- 扩展Flask-SQLAlchemy继承了SQLAlchemy，它简化了连接数据库服务器，管理数据库操作会话等各类工作，让Flask中的数据处理体验变得更加轻松。

```
$ pip install flask-sqlalchemy
```

Flask-SQLAlchemy扩展

- 扩展初始化：实例化Flask-SQLAlchemy提供的SQLAlchemy类，传入程序实例以完成扩展的初始化：

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
db = SQLAlchemy(app)
```

db对象代表数据库，可以使用Flask-SQLAlchemy提供的所有功能

连接数据库服务器

- 数据库管理系统通常会提供数据库服务器运行在操作系统中。要连接数据库服务器，首先要为我们的程序指定数据库URI。数据库URI是一串包含各种属性的字符串，其中包含了各种用于连接数据库的信息。
  - URI代表统一资源标识符，是用来标示资源的一组字符串。URL是它的自己。在大多数情况下，这两者可以交替使用。

数据库管理系统	URI
模式	dialect+driver://username-password@host:port/databasename(数据库接口/driver部分省略会使用默认选项)
Microsoft SQL Server	mssql://username:password@host:port/databasename
PostgreSQL	postgresql://username:password@host/databasename
MySQL	mysql://username:password@host/databasename
SQLite (UNIX)	sqlite:////absolute/path/to/foo.db
SQLite (内存型)	sqlite://// 或 sqlite:////memory:

连接数据库服务器

- 数据库的URI通过配置变量SQLALCHEMY_DATABASE_URI设置，默认为SQLite内存型数据库(sqlite:///:memory:)。SQLite是基于文件的数据库管理系统，不需要设置数据库服务器，只需要指定数据库文件的绝对路径。我们使用app.root_path来定位数据库文件的路径。

```
import os
...
app.config['SQLALCHEMY_DATABASE_URI'] = os.getenv('DATABASE_URL', 'sqlite:///' +
os.path.join(app.root_path, 'data.db'))
```

在生产环境下更换到其他类型的数据库管理系统时，数据库URI会包含敏感信息，所以这里优先从环境变量DATABASE_URL获取。

SQLite数据库文件名不限定后缀，常用的命名方式有foo.sqlite, foo.db,或是注明SQLite版本的foo.sqlite3。

定义数据库模型

- 用来映射到数据库表的Python类通常被称为数据库模型（model），一个数据库模型类对应数据库中的一个表。定义模型即使用Python类定义表模式，并声明映射关系。所有的模型类都需要继承Flask-SQLAlchemy提供的db.Model基类。
- 我们以一个笔记程序为例，将笔记保存到数据库中，你可以通过程序查询、添加、更新和删除笔记。

```
class Note(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    body = db.Colum(db.Text)
```

模型类中，表的字段（列）由db.Column类的实例表示，字段的类型通过Column类构造方法的第一个参数传入。在这个模型中，我们创建了一个类型为db.Integer的id字段和类型为db.Text的body列，分别存储整型和文本。

定义数据库模型

- 默认情况下，Flask-SQLAlchemy会根据模型类的名称生成一个表名称，生成规则如下：
  - `Message → message # 单个单词转换为小写`
  - `FooBar → foo_bar # 多个单词转换为小写并使用下划线分隔`
- Note类对应的表名称即note。如果你想自己定义表名称，可以通过定义__tablename__属性来实现。字段名默认为类属性名，你也可以通过字段类构造方法的第一个参数指定，或使用关键字name。

定义数据库模型

- 除了name参数，实例化字段时常用的字段参数如下：

参 数 名	说　明
primary_key	如果设为True，该字段为主键
unique	如果设为True，该字段不允许出现重复值
index	如果设为True，为该字段创建索引
nullable	确定字段值可否为空，值为True或False，默认值为True
default	为字段设置默认值

创建数据库和表

- 如果把数据库（文件）看做一个仓库，为了方便取用，我们需要把货物按照类型分别放置在不同货架上，这些货架就是数据库中的表。创建模型类后，我们需要手动创建数据库和对应的表，也就是我们常说的建库和建表。
  - 这通过对我们的db对象调用`create_all()`方法实现

创建数据库和表

- 我们也可以实现一个自定义flask命令完成这个工作。

```
import click
...
@app.cli.command()
def initdb():
    db.create_all()
    click.echo('Initialized database.')
```

```
$ flask initdb
```

数据库操作

- 数据库操作主要是CRUD，即Create（创建）、Read（读取/查询）、Update（更新）和Delete（删除）。
- SQLAlchemy使用数据库会话来管理数据库操作，这里的数据库会话也成为事务（transaction）。Flask-SQLAlchemy自动帮我们创建会话，可以通过db.session属性获取。
- 数据库中的会话代表一个临时存储区，你对数据库做出的改动都会放在这里。你可以调用add()方法将新创建的对象添加到数据库会话中，或是对会话中的对象进行更新。只有当你对数据库会话对象调用commit()方法时，改动才被提交到数据库，这确保了数据提交的一致性。另外，数据库会话也支持回滚操作。当你对会话调用rollback()方法时，添加到会话中且未提交的改动都将被撤销。

数据库操作：Create

- 添加一条新记录到数据库主要分为三步：
  - 创建Python对象（实例化模型类）作为一条记录。
  - 添加新创建的记录到数据库会话。
  - 提交数据库会话。

```
from app import db, Note  # 导入db对象和Note类

# 创建两个Note实例表示两条记录，使用关键字参数传入字段数据
note1 = Note(body='remember Sammy Jankis')
note2 = Note(body='SHAVE')
# 把Note实例添加到会话对象
db.session.add(note1)
db.session.add(note2)
# 提交会话
db.session.commit()
```

数据库操作：Read

- 我们已经知道了如何向数据库里添加记录，那么如何从数据库里取回数据呢？使用模型类体用的query属性附加调用各种过滤方法及查询方法可以完成这个任务。
  - <模型类>.query.<过滤方法>.<查询方法>
- 从某个模型类出发，通过在query属性对应的Query对象上附加的过滤方法和查询函数对模型类对应的表中的记录进行各种筛选和调整，最终返回包含对应数据库记录数据的模型类实例，对返回的示例调用属性即可获取对应的字段数据。

数据库操作：Read

- 查询方法

```
# all()返回所有记录
Note.query.all()
# first()返回第一条记录
note1 = Note.query.all()
print(note1.body)
# get()返回指定主键值（id字段）的记录
note2 = Note.query.get(1)
# count()返回记录的数量
Note.query.count()
```

模型类中，表的字段（列）由db.Column类的实例表示，字段的类型通过Column类构造方法的第一个参数传入。在这个模型中，我们创建了一个类型为db.Integer的id字段和类型为db.Text的body列，分别存储整型和文本。

数据库操作：Read

- 过滤方法：SQLAlchemy还提供了许多过滤方法，使用这些过滤方法可以获取更精确的查询，比如获取指定字段值的记录。对模型类的query属性存储的Query对象调用过滤方法将返回一个更精确的Query对象（即查询对象）。因为每个过滤方法都会返回新的查询对象，所以过滤器可以叠加使用。在查询对象上调用前面介绍的查询方法，即可获得一个包含过滤后的记录的列表。

```
Note.query.filter(Note.body=='SHAVE').first()
```

数据库操作：Read

- 常用过滤方法

过滤方法	说　明
filter()	使用指定的规则过滤记录，返回新产生的查询对象
filter_by()	使用指定规则过滤记录（以关键字表达式的形式），返回新产生的查询对象
order_by()	根据指定条件对记录进行排序，返回新产生的查询对象
limit(limit)	使用指定的值限制查询返回的记录数量，返回新产生的查询对象
group_by()	根据指定条件对记录进行分组，返回新产生的查询对象

完整查询方法和过滤方法见：http://docs.sqlalchemy.org/en/latest/orm/query.html

数据库操作：Update

- 更新一条记录非常简单，直接赋值给模型类的字段属性就可以改变字段值，然后调用commit()方法提交会话即可。

```
note = Note.query.get(1)
note.body = 'SHAVE LEFT THIGH'
db.session.commit()
```

数据库操作：Delete

- 删除记录和添加记录很相似，不过要把add()方法换成delete()方法，最后都需要调用commit()方法提交修改。

```
note = Note.query.get(1)
db.session.delete(note)
db.session.commit()
```