数据库与 Flask ORM

数据库简介

数据库(Database)是一种有组织的、相互关联的数据集合，并建模了现实世界中的某些方面。它是后端程序的**核心**，也是后端开发中最有趣的（最具有设计感的）部分之一。

如果你想写一个例如网易云音乐的 APP，你当然需要一个数据库来存储艺术家/专辑的信息. 你可以有很多方式来组织你的数据.

最简单的方式是用 CSV(comma-separated values) 文件存储[1]:

- 每个实体存储单独的文件
- 应用程序每次读取、更新记录时都要对文件进行解析(parse)

```
# Artist(name, year, country)
"Wu-Tang Clan",1992,"USA"
"Notorious BIG",1992,"USA"
"GZA",1990,"USA"
```

```
# Album(name, artist, year)
"Enter the Wu-Tang","Wu-Tang Clan",1993
"St.Ides Mix Tape","Wu-Tang Clan",1994
"Liquid Swords","GZA",1990
```

示例：Python 程序读取 GZA 对应的出道年份

```
for line in file.readlines():
  records = parse(line)
  if records[0] == "GZA":
    print(records[1])
```

但在实际的后端场景中，你需要解决的问题远比你想象中要多:

- **数据完整性**: 如果我要删除有专辑的艺术家怎么办(不然会出现 dangling pointer)? 新增年份是个非法的字符串怎么办? 如果一个专辑有多名艺术家怎么办?
- **数据持久化**: 如果程序在更新记录时崩溃了怎么办? 如果我们希望将数据存放在多个主机上要怎么同步?
- **实现**: 怎么不以 $O(n)$ 的复杂度找到特定记录? 如果另一个应用也要用到这个数据库怎么办(读写冲突)? 而且它还跑在别的主机上(怎么共享这个 csv 文件)?
- …

我们需要一个**数据库管理系统**(database management system, DBMS)，也即一个软件来帮助我们的应用程序存储以及分析数据库中的信息。

一个 DBMS 通常具备根据某种**数据模型**(data model)对数据库进行增删改查(Create, Read, Update, Delete, CRUD)以及其它操作的能力, 即, 管理的能力。

我们下面会看到, 在一个关系型数据库管理系统下, 上面的这些问题是如何解决的.

作为一名后端开发人员, 学习数据库, 以及 DBMS 是如何帮我们处理查询, 对我们实际工作有着巨大的帮助.

数据库的设计与使用是一个庞大且硬核的话题[2]. 本学期在预计的授课内容里, 主要介绍关系型数据库的使用.

---

[1] csv 文件可以拿 excel 解析打开，同时在数据分析领域因为它简单所以被广泛使用作为数据集格式。

[2] 如果你对这个话题感兴趣可以学习参考资料里提到的 CMU 15-445 课程.

数据模型

数据模型是描述数据库中数据的一组概念, 是数据在抽象层面的表示. 我们可以关注一些例子:

数据模型	代表	备注
关系型	MySQL, PostgreSQL, SQLite	大部分 DBMS 都是
键值对	Redis	简单应用或者缓存会用
图模型	Neo4j	NoSQL[3]
文档型/JSON/XML	MongoDB, Elasticsearch	NoSQL
向量/矩阵/张量	Pinecone, Faiss	针对大数据 AI 会用

Table 1: 常见的数据模型以及对应的 DBMS 代表

例如键值对的数据模型, 指的是数据以键值对的形式存储: 以某个键来查询数据的内容.

在 DBMS 里，我们可能还会遇上一个新的名词 schema(**模式**)[4]，它描述的是在给定数据模型下，对一组特殊数据集合的描述。

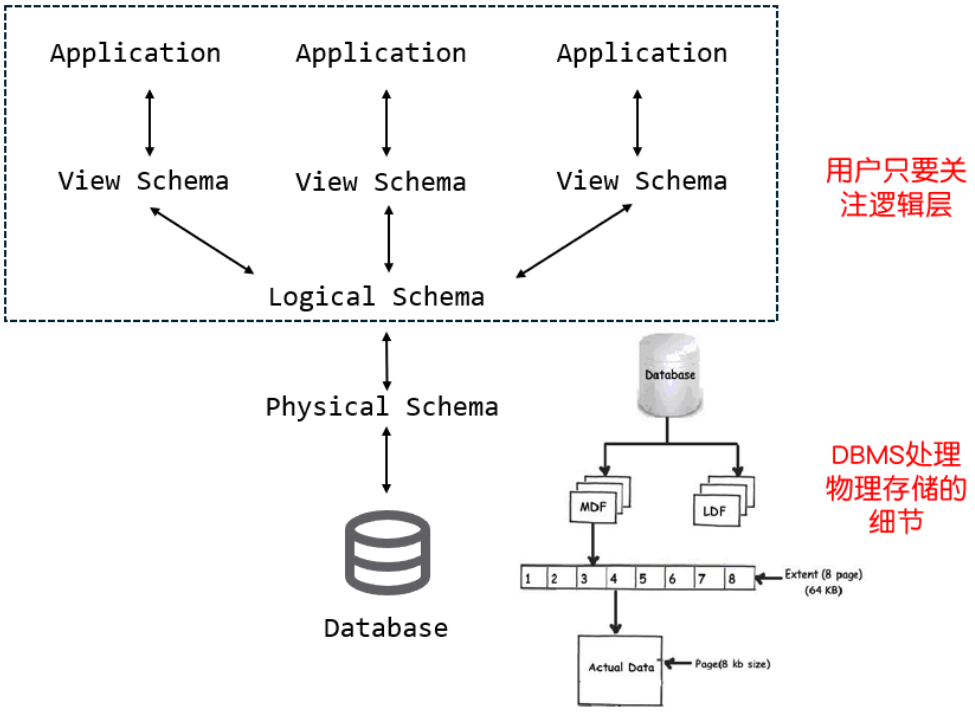
- 它定义了在一个数据模型下, 数据的结构.
- 脱离数据模型的上下文, 存放在硬盘上的数据只是一组组的字节.

关系型的数据模型

关系模型定义了一种基于**关系**(Relation)的数据抽象:

- 用关系这种简单的数据结构来存储数据库.
- 具体怎么实现的, 数据怎么存放在硬盘上, 留给 DBMS 去做.
- 用户只需要关注应用逻辑: 通过更高层次的语言, 让 DBMS 帮我们做出最佳的查询策略.

关系型 DBMS 内部可以抽象出下面的分层模型:



[3]No SQL, 现在越来越接近 Not only SQL :)

[4]这个名词的中文翻译并不统一, 有的地方还叫它架构, 纲要等等. 下面我们仅采用 schema 作为它的引用.

自底向上来说:

- 物理层面的 schema 定义了数据在物理存储上的组织方式, 定义了属于它的文件系统: 对存放的数据做出**页**, **文件**, **块**的划分.
- 逻辑层面上的 schema 定义了数据库存放数据的**逻辑结构**, 如: **表结构**, **关系**, **约束**.
- 视图层面的 schema 提供用户或应用程序的数据视图. 通过定义视图(类似一个虚拟表)来屏蔽逻辑 schema 中的复杂性, 为不同的应用提供不同的逻辑结构.

关系代数

关系型的数据模型有着深厚的数学理论依据. 大家很快会在离散数学这门课里了解到**关系代数**. 我们这里仅介绍一些基本概念:

一种**关系**(relation)是包含实体性质之间关系的无序集合(unordered set).

- n 元关系等于一个拥有 n 列的表格.

一个**元组**(tuple)是关系中的一组属性值, 也即它的**域**(domain).

- 值通常是原子的/标量.
- `NULL` 值可以是任何域的成员.
- 一个元组就是一条记录, 表中的一行(row)数据.

主键与外键

一个关系的**主键**(primary key)唯一标识一个元组.

一些 DBMS 会自动创建主键(如果你不指定的话), 同时保证生成唯一: 如 MySQL 会指定一个 `BIGINT` 类型的自增(`AUTO_INCREMENT`)主键.

id	name	year	country
101	Wu-Tang Clan	1992	USA
102	Notorious BIG	1992	USA
103	GZA	1990	USA

Table 2: Artist 表, 除了 `name`, `year`, `country` 字段外还有 `id` 作为主键

外键标志了一个关系里的属性映射到另一个关系的元组.

id	name	artist_id	year
11	Enter the Wu-Tang	101	1993
22	St.Ides Mix Tape	101	1994
33	Liquid Swords	103	1995

Table 3: Album 表, 用 `artist_id` 作为外键来引用艺术家

如果一个专辑有多名艺术家怎么办?

artist_id	album_id
101	11
101	22
103	22
103	33

Table 4: ArtistAlbum 中间表, 维护一个多对多关系

外键约束使得插入元组时, 必须引用已存在的主键; 通过添加 `CASCADE` 关键字, 在删除主表的元组时, 也会删除从表的相关元组(通过中间表来找到全部的元组).

对数据库中**任意**实例需要满足用户定义的条件:

- 可以对单个元组验证或对整个关系验证.
- DBMS 会阻止不满足约束的操作.
- 主键约束和外键约束是最常见的约束.

SQL 语句示例:

```sql
CREATE TABLE Artist(
  name VARCHAR(255) NOT NULL,
  year INT,
  country CHAR(60),
  CHECK(year > 1900)
)
```

数据操纵语言

数据操纵语言 (Data Manipulation Language, DML) 是 DBMS 暴露给应用存取数据的接口. 具体来说, 分成两种类型:

- 过程式 DML: 指定如何获取所需数据的详细步骤或过程.
- 声明式(非过程式) DML: 指定要查询的数据是什么.

运算符	作用
σ	选择
Π	投影
$\cup$	并
$\cap$	交
$-$	差
$\times$	笛卡尔积
$\bowtie$	自然连接

Table 5: 关系代数的七种基本运算符

- 每种运算符接受一种或多种关系, 并输出新的关系.
- 运算符之间可以**复合**(chain), 得到更复杂的关系.

SQL 语句示例:

```sql
SELECT * FROM R
WHERE a_id = 'a2' AND b_id>102;
```

a_id	b_id
a1	101
a2	102
a2	103
a3	104

a_id	b_id
a2	103

Table 6: 记左表为 R(原关系), 那么上面的查询对应表达式 $\sigma_{\text{a\_id='a2'} \wedge \text{b\_id>102}}(R)$

- $\sigma_{\text{predicate}}(R)$ 来按某一**谓词**(predicate)来查询关系中元组的子集.

- 谓词之间可以通过析取(or)和合取(and)来组合.

对于一次的数据查询, 关系代数需要指定每一步的顺序.

例如:

- $\sigma_{\text{b\_id}=102}(R \bowtie S)$ 和 $(R \bowtie \sigma_{\text{b\_id}=102}(S))$ 在结果上是一样的.
- 但如果 S 中有 10 亿条的数据, 先进行连接再筛选会非常缓慢. 显然后者会更快.

对一个更复杂的查询, 就不好手动进行优化了. 更好的方式是, 对你要查询的数据进行描述, 让 DBMS 计算具体的步骤.

- 正如我们不会手写汇编而让 complier 帮你优化编译.
- 查询 R 与 S 经过连接得到的元组, 筛选 `b_id` 等于 102 的对象.

构化查询语言 SQL(Structured Query Language) 语句基本就是对上面描述的一种复述. 它也是事实上声明式的 DML 的标准实现(有许多变体 dialects):

```
SELECT * FROM R JOIN S ON R.a_id = S.a_id
WHERE S.b_id = 102;
```

DBMS 会根据上面的查询, 通过内置的 query optimizer 计算出最优的执行计划, 然后执行.

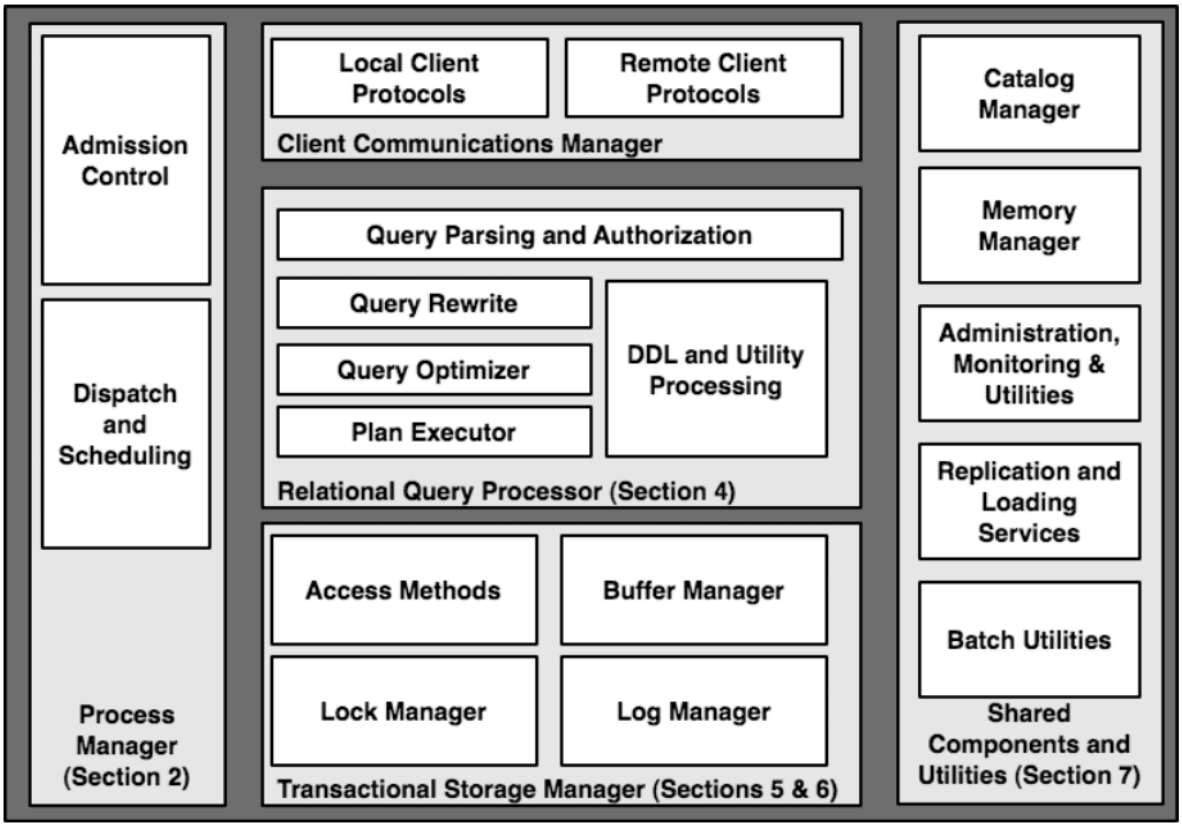


Fig. 1.1 Main components of a DBMS.

应用层使用

DBMS 提供对外的使用接口. 例如 MySQL 提供一套原始的 C API, 允许通过该接口来与 MySQL 进行底层上的交互.

```
#include <mysql.h>
#include <stdlib.h>
```

```c
int main(void) {
  if (mysql_library_init(0, NULL, NULL)) {
    fprintf(stderr, "could not initialize MySQL client library\n");
    exit(1);
  }

  // do something :)

  mysql_library_end();

  return EXIT_SUCCESS;
}
```

根据 DBMS 提供的 API, 我们可以写出与 DBMS 交互的**客户端**程序: 如, MySQL 的默认命令行界面, MySQL Workbench 等. 它们内部交互部分通常是通过 MySQL C API 编写的:

```
mysql> SELECT * FROM artist;
+----+----------+------+
| id | name     | year |
+----+----------+------+
| 101| Wu-Tang Clan| 1992 |
| 102| Notorious BIG| 1992 |
| 103| GZA      | 1990 |
+----+----------+------+
3 rows in set (0.01 sec)
```

对于其它编程语言, 也有类似的 DBMS 客户端实现, 也称为数据库驱动:

- Python: `mysqlclient`(C API 封装), `pymysql`(根据 PEP 249 实现的纯 python 库)
- Java: `JDBC`(Java Database Connectivity)

根据这些 DBMS client, 你可以连接至 DBMS Server, 然后执行 SQL 语句:

```python
import pymysql.cursors

connection = pymysql.connect(host='localhost',
                             user='user',
                             password='passwd',
                             database='db',
                             cursorclass=pymysql.cursors.DictCursor)

with connection:
    with connection.cursor() as cursor:
        sql = "INSERT INTO `users` (`email`, `password`) VALUES (%s, %s)"
        cursor.execute(sql, ('webmaster@python.org', 'very-secret'))

    connection.commit()

    with connection.cursor() as cursor:
        sql = "SELECT `id`, `password` FROM `users` WHERE `email`=%s"
        cursor.execute(sql, ('webmaster@python.org',))
        result = cursor.fetchone()
        # {'password': 'very-secret', 'id': 1}
        print(result)
```

解决了应用程序与 DBMS 交互的问题. 但在实际编码中, 我们还会有这样的问题:

- 将数据存放至 DBMS, 需要我们将存放数据转换成对应的 SQL 语句, 也即, 需要**序列化**.
- 对于从 DBMS 读取的数据, 我也需要经过一次**反序列化**使其变成编程语言能理解的对象.

这种频繁的转换十分繁琐. 有没有更方便的方法进行解封装? 答案是有的.

对象关系映射(Object Relational Mapping, ORM): 它把数据库中的表格映射成程序中的对象. 这样我们就可以用类似编程语言的方式来操作数据库, 而不用直接编写 SQL 查询语句.

ORM 模式可以**提供更高层次的抽象**, 使得数据库操作更易于管理和维护, 并且可以帮助你避免一些常见的安全问题, 比如 SQL 注入攻击.

我们下面就会介绍 Python Web 后端中最常用的 SQLAlchemy ORM 框架.

前沿数据库概览

在结束数据库的话题之前, 我们希望介绍一些前沿数据库的相关概念.

先前我们提到一种文档型数据模型: 它提供一种更灵活的组织数据的形式, 很多后端项目也会采用这种数据模型.

- 所谓文档就是包含命名字段/值对的层次结构.
- 一个字段的值可以是标量, 值的数组, 还可以是另一个文档.
- 现代最常用 JSON 格式的文档. 旧系统可能会用 XML 或者自定义一个 object.

在实现上, 对于一个多对多关系, 我们不再需要构建一个中间表, 而是直接在文档中记录另一个文档的信息:

```
class Artist {
  int id;
  String name;
  int year;
  Album albums[];
}
class Album {
  int id;
  String name;
  int year;
}
```

```
{
"name": "GZA",
"year": 1990,
"albums": [
    {
    "name": "Liquid Swords",
    "year": 1995
    },
    {
    "name": "Beneath the Surface",
    "year": 1999
    },
  ]
}
```

它能够解决关系型模型的**阻抗失配**问题.

但在实际应用中, 关系型数据库与这种文档型数据库的界限越来越模糊.

- 你可以在 MySQL(>5.7) 定义一个 JSON 字段, 在 8.0 版本又新增了 JSON 数据转换成关系型数据进行查询.
- 而文档型数据库, 如 MongoDB 也逐渐支持对 JSON 的类 SQL(MongoDB Query Language)语句.

另一种前沿数据模型是**向量数据**. 上下文语料会被转换成用于最近邻搜索（精确或近似）的一维数组, 利用相似度搜索算法在数据库内搜索最匹配的答案向量. 在实际中:

- 用于通过 LLM 训练 Transformer 模型（比如 ChatGPT）的语义搜索
- 与现代机器学习工具和 API（例如 LangChain、OpenAI）的集成

向量数据库是对 LLM token 限制的一种解决方案.

总结

数据库与数据库管理系统是两个不同的概念, 但狭义上我们并不加以区分.

了解数据库管理系统的实现对于后端开发是**很有帮助**的: 了解 DBMS 的优化查询与索引, 帮助我们写出更高效的 SQL 语句; 理解事务、锁机制和隔离级别, 可以更好地管理**数据一致性**; 学习 DBMS 如何存储数据, 开发者能更合理地设计数据库**表结构**和**索引方案**.

一个数据库管理系统的可靠性是饱经企业测试、部署验证的. 而它的复杂性和重要性, 在软件层面称与操作系统齐平也不为过. 专业的事情交给专业的软件去做: 当然你要实现也是很浪漫的一件事.

我们在 DBMS 的设计上再次看到了**抽象**的重要性: 而这也是整个计算机科学的核心. 得益于此, 我们不必关心底层的复杂性, 而只需关注上层的逻辑建模, 编写高层次的应用级代码.

最后, 数据库也是**与时俱进的**一个领域. 新兴基于非关系型的数据库在实践中得到越来越多的应用, 如大模型的发展带动了向量化数据库的发展. 在该领域深耕, 说不定将来你也能成为一名优秀的 infra engineer.

Flask ORM

安装依赖

```
pip install flask-sqlalchemy
```

`flask-sqlalchemy` 在使用上很像 `SQLAlchemy`.

连接数据库

连接 SQLite 数据库(一个本地的 DBMS):

```
from flask_sqlalchemy import SQLAlchemy

app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///mydb.db' # unix dir
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False

db = SQLAlchemy(app)
```

之后 `db` 就是我们连接数据库的对象.

Flask-SQLAlchemy 文档建议把 `SQLALCHEMY_TRACK_MODIFICATIONS` 键设为 `False`, 以便在不需要跟踪对象变化时降低内存消耗.

如果你需要连接 MySQL 数据库, 你需要安装 `mysqlclient`(但这个库在 Windows 上的构建并不容易, 你需要预安装许多构建开发包, 详情见 安装文档), 然后:

```
app.config['SQLALCHEMY_DATABASE_URI'] = 'mysql://username:password@hostname/database'
```

推荐数据库密码通过环境变量进行管理, 然后 f-string 导入:

```
SQLALCHEMY_DATABASE_URI = f'mysql://
    {USERNAME}:{PASSWORD}@{HOSTNAME}:{PORT}/{DB_NAME}'
```

模型定义

```
class User(db.Model):
    __tablename__ = 'users'
    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)

    def __repr__(self):
        return f'{self.name}'
```

- `Column` 内的参数可以对字段进行约束: 指定字段类型, 以及其它约束.
- 类变量 `__tablename__` 定义在数据库中使用的表名(不指定 Flask 自己会创建).

关系

一对多关系:

```
class Role(db.Model):
    # ...
    users = db.relationship('User', backref='role')

class User(db.Model):
    # ...
    role_id = db.Column(db.Integer, db.ForeignKey('roles.id'))
```

多对多关系(需要建立中间表):

```
class Course(db.Model):
    __tablename__ = 'courses'

    id = db.Column(db.Integer, primary_key=True, info='Primary Key')
    name = db.Column(db.String(255))

    students = db.relationship('Student', secondary='enrollment', backref='courses')


class Student(db.Model):
    __tablename__ = 'students'

    id = db.Column(db.Integer, primary_key=True, info='Primary Key')
    student_name = db.Column(db.String(255))


t_enrollment = db.Table(
    'enrollment',
    db.Column('student_id', db.ForeignKey('students.id'), primary_key=True,
nullable=False),
    db.Column('course_id', db.ForeignKey('courses.id'), primary_key=True,
nullable=False, index=True)
)
```

- 为它传入一个 `secondary`(二级表) 参数, 也就是我们上面说的关联表的名称.
- `backref` 构建反向引用. 这样 `Student` 类的实例就有 `courses` 的一个一对多关系.(不需要在 `Student` 重复)

数据库操作

你可以在 flask shell 里实时进行数据库操作(实验), 或者加载好 `app` 的上下文.

```
(.venv) $ flask shell
Python 3.12.1 (main, Dec 10 2023, 15:07:36) [GCC 11.4.0] on linux
App: app
Instance: /home/besthope/backend-labs/instance
>>>
```

- 创建表 `db.create_all()`
- 删除表 `db.delete_all()`

插入行要分几个步骤, 类似于进行一次 git commit:

```
>>> test = User(id=1, username='123')
>>> db.session.add(test)
>>> test_2 = test(id=2, username='456')
>>> db.session.add_all([test_2])
>>> db.session.commit()
```

查询记录:

```
>>> User.query.all()
[<User 1>]
```

利用过滤器:

```
>>> User.query.filter_by(username='1').all()
```

查看原始 SQL:

```
>>> str(User.query.filter_by(username='1'))
'SELECT user.id AS user_id, user.username AS user_username, user.email AS user_email
\nFROM user \nWHERE user.username = %s'
```

编写一段学生选课简单的接口代码:

```
@api.route('/course')
def course_info():
    student_id = request.args.get('id')

    if student_id is None:
        return {'message' : 'Missing student id.'}, 400

    student = Student.query.get(int(student_id))

    course_info = [course.name for course in student.courses]

    return jsonify(course_info)
```

数据库更新与迁移

思考一个问题: 当我们修改了数据模型后, 而数据库已经存放了一些记录, 怎么能让旧结构的数据迁移到新结构的数据上呢?

最简单的做法: 直接删除旧表, 然后创建新表. 测试数据库支持我们大刀阔斧.

更好的做法: **使用数据库迁移工具**.

- 它可以自动生成 SQL 语句, 帮助我们将旧数据迁移到新表.
- 它还可以帮助我们管理数据库的版本, 并提供回滚功能.
- 多人协作开发时, 它可以确保中心数据库的一致性.

安装插件

```
pip install flask-migrate
```

创建迁移脚本

```
flask db init
```

创建迁移脚本

```
flask db migrate -m "message"
```

- `-m` 参数指定迁移脚本的描述信息. 留空会自动生成迁移消息.

升级数据库

```
flask db upgrade
```

回滚数据库

```
flask db downgrade
```

你也可以指定版本号回滚到指定版本的迁移脚本.

参考资料

CMU 15-445 数据库系统课程: 我们对关系型数据库的介绍来源于此. 对数据库感兴趣的同学应该对这门课不陌生. 网上有大量的帖子来论证这门公开课的质量之高. 主讲人 Andy 称 Database, second important thing in my life 可见一斑!

Readings in Database Systems, 5th Edition 想要深入研究 DBMS 的同学可以阅读该书.

Blog: 对向量数据库的介绍. 写的很好, 感兴趣的同学可以更加深度地了解下.