

19 访问数据库

程序运行的时候，数据都是在内存中的。当程序终止的时候，通常都需要将数据保存 to 磁盘上，无论是保存到本地磁盘，还是通过网络保存 to 服务器上，最终都会将数据写入磁盘文件。

而如何定义数据的存储格式就是一个大问题。如果我们自己来定义存储格式，比如保存一个班级所有学生的成绩单：

名字	成绩
Michael	99
Bob	85
Bart	59
Lisa	87

你可以用一个文本文件保存，一行保存一个学生，用 `,` 隔开：

```
Michael,99
Bob,85
Bart,59
Lisa,87
```

你还可以用JSON格式保存，也是文本文件：

```
[
  {"name":"Michael","score":99},
  {"name":"Bob","score":85},
  {"name":"Bart","score":59},
  {"name":"Lisa","score":87}
]
```

你还可以定义各种保存格式，但是问题来了：

存储和读取需要自己实现，JSON还是标准，自己定义的格式就各式各样了；

不能做快速查询，只有把数据全部读到内存中才能自己遍历，但有时候数据的大小远远超过了内存（比如蓝光电影，40GB的数据），根本无法全部读入内存。

为了便于程序保存和读取数据，而且，能直接通过条件快速查询到指定的数据，就出现了数据库（Database）这种专门用于集中存储和查询的软件。

数据库软件诞生的历史非常久远，早在1950年数据库就诞生了。经历了网状数据库，层次数据库，我们现在广泛使用的关系数据库是20世纪70年代基于关系模型的基础上诞生的。

关系模型有一套复杂的数学理论，但是从概念上是十分容易理解的。举个学校的例子：

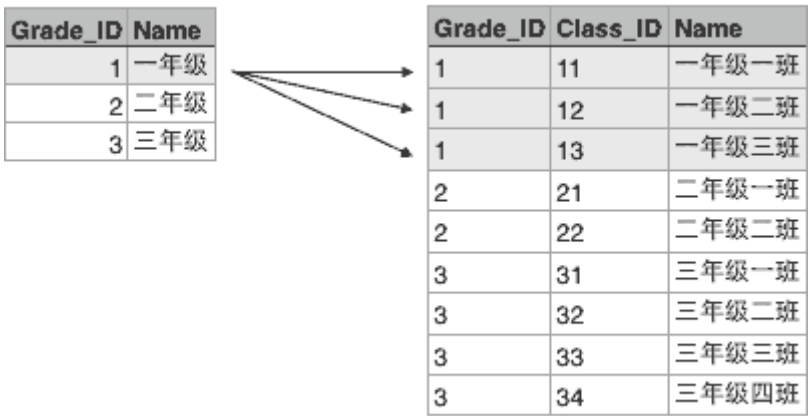
假设某个XX省YY市ZZ县第一实验小学有3个年级，要表示出这3个年级，可以在Excel中用一个表格画出来：

Grade_ID	Name
1	一年级
2	二年级
3	三年级

每个年级又有若干个班级，要把所有班级表示出来，可以在Excel中再画一个表格：

Grade_ID	Class_ID	Name
1	11	一年级一班
1	12	一年级二班
1	13	一年级三班
2	21	二年级一班
2	22	二年级二班
3	31	三年级一班
3	32	三年级二班
3	33	三年级三班
3	34	三年级四班

这两个表格有个映射关系，就是根据Grade_ID可以在班级表中查找到对应的所有班级：



也就是Grade表的每一行对应Class表的多行，在关系数据库中，这种基于表（Table）的一对多的关系就是关系数据库的基础。

根据某个年级的ID就可以查找所有班级的行，这种查询语句在关系数据库中称为SQL语句，可以写成：

```
SELECT * FROM classes WHERE grade_id = '1';
```

结果也是一个表：

grade_id	class_id	name
1	11	一年级一班
1	12	一年级二班
1	13	一年级三班

类似的，Class表的一行记录又可以关联到Student表的多行记录：

Grade_ID	Class_ID	Name
1	11	一年级一班
1	12	一年级二班
1	13	一年级三班
2	21	二年级一班
2	22	二年级二班
3	31	三年级一班
3	32	三年级二班
3	33	三年级三班
3	34	三年级四班

Class_ID	Num	Name	Score
11	10001	Michael	99
11	10002	Bob	85
11	10003	Bart	59
11	10004	Lisa	87
12	10005	Tracy	91
...	

由于本教程不涉及到关系数据库的详细内容，如果你想从零学习关系数据库和基本的SQL语句，如果你想从零学习关系数据库和基本的SQL语句，请参考[SQL教程](#)。

NoSQL

你也许还听说过NoSQL数据库，很多NoSQL宣传其速度和规模远远超过关系数据库，所以很多同学觉得有了NoSQL是否就不需要SQL了呢？千万不要被他们忽悠了，连SQL都不明白怎么可能搞明白NoSQL呢？

数据库类别

既然我们要使用关系数据库，就必须选择一个关系数据库。目前广泛使用的关系数据库也就这么几种：

付费的商用数据库：

- Oracle，典型的高富帅；
- SQL Server，微软自家产品，Windows定制专款；
- DB2，IBM的产品，听起来挺高端；
- Sybase，曾经跟微软是好基友，后来关系破裂，现在家境惨淡。

这些数据库都是不开源而且付费的，最大的好处是花了钱出了问题可以找厂家解决，不过在Web的世界里，常常需要部署成千上万的数据库服务器，当然不能把大把大把的银子扔给厂家，所以，无论是Google、Facebook，还是国内的BAT，无一例外都选择了免费的开源数据库：

- MySQL，大家都在用，一般错不了；

- PostgreSQL，学术气息有点重，其实挺不错，但知名度没有MySQL高；
- sqlite，嵌入式数据库，适合桌面和移动应用。

作为Python开发工程师，选择哪个免费数据库呢？当然是MySQL。因为MySQL普及率最高，出了错，可以很容易找到解决方法。而且，围绕MySQL有一大堆监控和运维的工具，安装和使用很方便。

为了能继续后面的学习，你需要从MySQL官方网站下载并安装[MySQL Community Server 5.6](#)，这个版本是免费的，其他高级版本是要收钱的（请放心，收钱的功能我们用不上）。

使用SQLite

SQLite是一种嵌入式数据库，它的数据库就是一个文件。由于SQLite本身是C写的，而且体积很小，所以，经常被集成到各种应用程序中，甚至在iOS和Android的App中都可以集成。

Python就内置了SQLite3，所以，在Python中使用SQLite，不需要安装任何东西，直接使用。

在使用SQLite前，我们先要搞清楚几个概念：

表是数据库中存放关系数据的集合，一个数据库里面通常都包含多个表，比如学生的表，班级的表，学校的表，等等。表和表之间通过外键关联。

要操作关系数据库，首先需要连接到数据库，一个数据库连接称为Connection；

连接到数据库后，需要打开游标，称之为Cursor，通过Cursor执行SQL语句，然后，获得执行结果。

Python定义了一套操作数据库的API接口，任何数据库要连接到Python，只需要提供符合Python标准的数据库驱动即可。

由于SQLite的驱动内置在Python标准库中，所以我们可以直接来操作SQLite数据库。

我们在Python交互式命令行实践一下：

```
# 导入SQLite驱动：
>>> import sqlite3
# 连接到SQLite数据库
# 数据库文件是test.db
# 如果文件不存在，会自动在当前目录创建：
>>> conn = sqlite3.connect('test.db')
# 创建一个Cursor：
>>> cursor = conn.cursor()
# 执行一条SQL语句，创建user表：
>>> cursor.execute('create table user (id varchar(20)
primary key, name varchar(20))')
<sqlite3.Cursor object at 0x10f8aa260>
# 继续执行一条SQL语句，插入一条记录：
```

```
>>> cursor.execute('insert into user (id, name) values
(\'1\', \'Michael\')')
<sqlite3.Cursor object at 0x10f8aa260>
# 通过rowcount获得插入的行数:
>>> cursor.rowcount
1
# 关闭Cursor:
>>> cursor.close()
# 提交事务:
>>> conn.commit()
# 关闭Connection:
>>> conn.close()
```

我们再试试查询记录:

```
>>> conn = sqlite3.connect('test.db')
>>> cursor = conn.cursor()
# 执行查询语句:
>>> cursor.execute('select * from user where id=?',
('1',))
<sqlite3.Cursor object at 0x10f8aa340>
# 获得查询结果集:
>>> values = cursor.fetchall()
>>> values
[('1', 'Michael')]
>>> cursor.close()
>>> conn.close()
```

使用Python的DB-API时, 只要搞清楚 **Connection** 和 **Cursor** 对象, 打开后一定记得关闭, 就可以放心地使用。

使用 **Cursor** 对象执行 **insert**, **update**, **delete** 语句时, 执行结果由 **rowcount** 返回影响的行数, 就可以拿到执行结果。

使用 **Cursor** 对象执行 **select** 语句时, 通过 **fetchall()** 可以拿到结果集。结果集是一个list, 每个元素都是一个tuple, 对应一行记录。

如果SQL语句带有参数, 那么需要把参数按照位置传递给 **execute()** 方法, 有几个 **?** 占位符就必须对应几个参数, 例如:

```
cursor.execute('select * from user where name=? and
pwd=?', ('abc', 'password'))
```

SQLite支持常见的标准SQL语句以及几种常见的数据类型。具体文档请参阅SQLite官方网站。

小结

- 在Python中操作数据库时，要先导入数据库对应的驱动，然后，通过 `Connection` 对象和 `Cursor` 对象操作数据。
- 要确保打开的 `Connection` 对象和 `Cursor` 对象都正确地被关闭，否则，资源就会泄露。
- 如何才能确保出错的情况下也关闭掉 `Connection` 对象和 `Cursor` 对象呢？请回忆 `try:...except:...finally:...` 的用法。

练习

请编写函数，在Sqlite中根据分数段查找指定的名字：

```
# -*- coding: utf-8 -*-

import os, sqlite3

db_file = os.path.join(os.path.dirname(__file__),
                        'test.db')
if os.path.isfile(db_file):
    os.remove(db_file)

# 初始数据:
conn = sqlite3.connect(db_file)
cursor = conn.cursor()
cursor.execute('create table user(id varchar(20) primary
key, name varchar(20), score int)')
cursor.execute(r"insert into user values ('A-001', 'Adam',
95)")
cursor.execute(r"insert into user values ('A-002', 'Bart',
62)")
cursor.execute(r"insert into user values ('A-003', 'Lisa',
78)")
cursor.close()
conn.commit()
conn.close()

def get_score_in(low, high):
    ' 返回指定分数区间的名字，按分数从低到高排序 '
    ----
    pass
    ----

# 测试:
assert get_score_in(80, 95) == ['Adam'], get_score_in(80,
95)
assert get_score_in(60, 80) == ['Bart', 'Lisa'],
get_score_in(60, 80)
assert get_score_in(60, 100) == ['Bart', 'Lisa', 'Adam'],
get_score_in(60, 100)

print('Pass')
```

使用MySQL

MySQL是Web世界中使用最广泛的数据库服务器。SQLite的特点是轻量级、可嵌入，但不能承受高并发访问，适合桌面和移动应用。而MySQL是为服务器端设计的数据库，能承受高并发访问，同时占用的内存也远远大于SQLite。

此外，MySQL内部有多种数据库引擎，最常用的引擎是支持数据库事务的InnoDB。

安装MySQL

可以直接从MySQL官方网站下载最新的[Community Server 5.6.x](#)版本。MySQL是跨平台的，选择对应的平台下载安装文件，安装即可。

安装时，MySQL会提示输入 `root` 用户的口令，请务必记清楚。如果怕记不住，就把口令设置为 `password`。

在Windows上，安装时请选择 `UTF-8` 编码，以便正确地处理中文。

在Mac或Linux上，需要编辑MySQL的配置文件，把数据库默认的编码全部改为UTF-8。MySQL的配置文件默认存放在 `/etc/my.cnf` 或者 `/etc/mysql/my.cnf`：

```
[client]
default-character-set = utf8

[mysqld]
default-storage-engine = INNODB
character-set-server = utf8
collation-server = utf8_general_ci
```

重启MySQL后，可以通过MySQL的客户端命令行检查编码：

```
$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor...
...

mysql> show variables like '%char%';
+-----+-----+
| variable_name | value |
+-----+-----+
| character_set_client | utf8 |
| character_set_connection | utf8 |
| character_set_database | utf8 |
| character_set_filesystem | binary |
```

```

| character_set_results | utf8
|
| character_set_server | utf8
|
| character_set_system | utf8
|
| character_sets_dir | /usr/local/mysql-5.1.65-
osx10.6-x86_64/share/charsets/ |
+-----+-----+
-----+
8 rows in set (0.00 sec)

```

看到 **utf8** 字样就表示编码设置正确。

注：如果MySQL的版本 $\geq 5.5.3$ ，可以把编码设置为 **utf8mb4**，**utf8mb4** 和 **utf8** 完全兼容，但它支持最新的Unicode标准，可以显示emoji字符。

安装MySQL驱动

由于MySQL服务器以独立的进程运行，并通过网络对外服务，所以，需要支持Python的MySQL驱动来连接到MySQL服务器。MySQL官方提供了mysql-connector-python驱动，但是安装的时候需要给pip命令加上参数 **--allow-external**：

```
$ pip install mysql-connector-python --allow-external
mysql-connector-python
```

如果上面的命令安装失败，可以试试另一个驱动：

```
$ pip install mysql-connector
```

我们演示如何连接到MySQL服务器的test数据库：

```

# 导入MySQL驱动：
>>> import mysql.connector
# 注意把password设为你的root口令：
>>> conn = mysql.connector.connect(user='root',
password='password', database='test')
>>> cursor = conn.cursor()
# 创建user表：
>>> cursor.execute('create table user (id varchar(20)
primary key, name varchar(20))')
# 插入一行记录，注意MySQL的占位符是%s：
>>> cursor.execute('insert into user (id, name) values
(%s, %s)', ['1', 'Michael'])
>>> cursor.rowcount
1
# 提交事务：
>>> conn.commit()

```



```
>>> cursor.close()
# 运行查询:
>>> cursor = conn.cursor()
>>> cursor.execute('select * from user where id = %s',
('1',))
>>> values = cursor.fetchall()
>>> values
[('1', 'Michael')]
# 关闭Cursor和Connection:
>>> cursor.close()
True
>>> conn.close()
```

由于Python的DB-API定义都是通用的，所以，操作MySQL的数据库代码和SQLite类似。

小结

- 执行INSERT等操作后要调用 `commit()` 提交事务；
- MySQL的SQL占位符是 `%s`。

使用SQLAlchemy

数据库表是一个二维表，包含多行多列。把一个表的内容用Python的数据结构表示出来的话，可以用一个list表示多行，list的每一个元素是tuple，表示一行记录，比如，包含 `id` 和 `name` 的 `user` 表：

```
[
    ('1', 'Michael'),
    ('2', 'Bob'),
    ('3', 'Adam')
]
```

Python的DB-API返回的数据结构就是像上面这样表示的。

但是用tuple表示一行很难看出表的结构。如果把一个tuple用class实例来表示，就可以更容易地看出表的结构来：

```
class User(object):
    def __init__(self, id, name):
        self.id = id
        self.name = name

[
    User('1', 'Michael'),
    User('2', 'Bob'),
    User('3', 'Adam')
]
```

这就是传说中的ORM技术：Object-Relational Mapping，把关系数据库的表结构映射到对象上。是不是很简单？

但是由谁来做这个转换呢？所以ORM框架应运而生。

在Python中，最有名的ORM框架是SQLAlchemy。我们来看看SQLAlchemy的用法。

首先通过pip安装SQLAlchemy：

```
$ pip install sqlalchemy
```

然后，利用上次我们在MySQL的test数据库中创建的user表，用SQLAlchemy来试试：

第一步，导入SQLAlchemy，并初始化DBSession：

```
# 导入：
from sqlalchemy import Column, String, create_engine
from sqlalchemy.orm import sessionmaker
from sqlalchemy.ext.declarative import declarative_base

# 创建对象的基类：
Base = declarative_base()

# 定义User对象：
class User(Base):
    # 表的名字：
    __tablename__ = 'user'

    # 表的结构：
    id = Column(String(20), primary_key=True)
    name = Column(String(20))

# 初始化数据库连接：
engine =
create_engine('mysql+mysqlconnector://root:password@localhost:3306/test')
# 创建DBSession类型：
DBSession = sessionmaker(bind=engine)
```

以上代码完成SQLAlchemy的初始化和具体每个表的class定义。如果有多个表，就继续定义其他class，例如School：

```
class School(Base):
    __tablename__ = 'school'
    id = ...
    name = ...
```

`create_engine()` 用来初始化数据库连接。SQLAlchemy用一个字符串表示连接信息：

```
'数据库类型+数据库驱动名称://用户名:口令@机器地址:端口号/数据库名'
```

你只需要根据需要替换掉用户名、口令等信息即可。

下面，我们看看如何向数据库表中添加一行记录。

由于有了ORM，我们向数据库表中添加一行记录，可以视为添加一个 `User` 对象：

```
# 创建session对象：
session = DBSession()
# 创建新用户对象：
new_user = User(id='5', name='Bob')
# 添加到session：
session.add(new_user)
# 提交即保存到数据库：
session.commit()
# 关闭session：
session.close()
```

可见，关键是获取 `session`，然后把对象添加到 `session`，最后提交并关闭。
`DBSession` 对象可视为当前数据库连接。

如何从数据库表中查询数据呢？有了ORM，查询出来的可以不再是 `tuple`，而是 `User` 对象。SQLAlchemy提供的查询接口如下：

```
# 创建Session：
session = DBSession()
# 创建Query查询，filter是where条件，最后调用one()返回唯一行，如果
调用all()则返回所有行：
user = session.query(User).filter(User.id=='5').one()
# 打印类型和对象的name属性：
print('type:', type(user))
print('name:', user.name)
# 关闭Session：
session.close()
```

运行结果如下：

```
type: <class '__main__.User'>
name: Bob
```

可见，ORM就是把数据库表的行与相应的对象建立关联，互相转换。

由于关系数据库的多个表还可以用外键实现一对多、多对多等关联，相应地，ORM框架也可以提供两个对象之间的一对多、多对多等功能。

例如，如果一个User拥有多个Book，就可以定义一对多关系如下：

```
class User(Base):
    __tablename__ = 'user'

    id = Column(String(20), primary_key=True)
    name = Column(String(20))
    # 一对多:
    books = relationship('Book')

class Book(Base):
    __tablename__ = 'book'

    id = Column(String(20), primary_key=True)
    name = Column(String(20))
    # “多”的一方的book表是通过外键关联到user表的:
    user_id = Column(String(20), ForeignKey('user.id'))
```

当我们查询一个User对象时，该对象的books属性将返回一个包含若干个Book对象的list。

小结

- ORM框架的作用就是把数据库表的一行记录与一个对象互相做自动转换。
- 正确使用ORM的前提是了解关系数据库的原理。