Python 数据存储

关系型数据库与DB-API

DB-API是Python中访问关系型数据库的标准API。使用它可以编写简单的程序来处理多种类型的关系型数据库，不需要为每种数据库编写独立的程序。其主要函数如下：

- `connect()` 用来连接数据库，包含参数用户名、密码、服务器地址等等。
- `cursor()` 创建一个cursor对象来管理查询。
- `execute()` 和 `executemany()` 对数据库执行一个或多个SQL命令。
- `fetchone()`、`fetchmany()` 和 `fetchall()` 得到 `execute` 之后的结果。

Python中表示多行数据的标准方式是一个由元组构成的序列。

```
students = [
    ('zhangsan', 10),
    ('lisi', 11),
    ('wangwu', 10),
    ('zhaoliu', 11),
]
```

通过这种形式提供数据，可以很容易的使用Python标准数据库API和关系型数据库进行交互。所有数据库上的操作都通过SQL查询语句来完成。每一行输入输出数据用一个元组来表示。

SQLite

SQLite是一种轻量级的、优秀的开源关系型数据库。其数据库存储在普通文件中，这些文件在不同机器和操作系统之间是可移植的，使得SQLite成为简易关系型数据库应用的可移植的解决方案。SQLite仅支持原生SQL以及多用户并发操作。浏览器、智能手机和其他应用会把SQLite作为嵌入数据库。

创建数据库

第一步是创建（连接）数据库。一个SQLite数据库就是存储在文件系统上的单一文件。Python提供了 `sqlite3` 模块来管理对SQLite数据库文件的访问，包括为其加锁来防止多个写入操作的并发冲突。数据库会在第一次访问时被创建，不过应用程序本身要负责数据库中数据表的定义，即模式（schema）。通常你要执行 `connect()` 函数，给它提供数据库名、主机、用户名、密码和其他一些必要的参数。例如：

```
import sqlite3

db = sqlite3.connect(':memory:')
```

字符串 `:memory:` 用于在内存中创建数据库，有助于方便快速地测试，但程序结束或计算机关闭时所有数据都会丢失。

下面的代码会在连接数据库文件之前首先检查其是否存在：

```
import os
import sqlite3

db_file = 'todo.db'

db_is_new = not os.path.exists(db_file)

if db_is_new:
    print('Need to create schema')
    os.system('sqlite3 %s ""' % db_file)
else:
    print('Database exists, assume schema does, too.')
```

为了处理数据，下一步你需要创建一个游标。

游标实际上是一种能从包括多条数据记录的结果集中每次提取一条记录的机制，其充当类似指针的作用。

游标的一个常见用途就是保存查询结果，以便以后使用。游标的结果集是由SELECT语句产生，如果处理过程需要重复使用一个记录集，那么创建一次游标而重复使用若干次，比重复查询数据库要快的多。

通俗理解就是将受影响的数据暂时放到了一个内存区域的虚表中，而这个虚表就是游标。

一旦你有了游标，那么你就可以执行SQL查询语句了。比如：

```
c = db.cursor()
c.execute('create table student (name VARCHAR(32), age integer)')
# <sqlite3.Cursor at 0x10dec0ab0>
db.commit()
```

检查特定的数据表是否存在于数据库中：

```
def is_table_exists(cursor, table_name):
    cursor.execute('select name from sqlite_master where type="table";')

    if not list(cursor) or (table_name,) not in list(cursor):
        cursor.close()
        return False

    cursor.close()
    return True
```

为了向数据库表中插入多条记录，使用类似下面这样的语句：

```
c.executemany('insert into student values (?,?)', students)
db.commit()
```

使用`commit()`方法来提交当前数据库事务，如果不调用此方法，那么从上次执行`commit()`方法后对数据库数据所做的更改操作都不会生效（如果数据库存在多个连接，对数据的改变操作不会被其他连接看到），如果数据没有被写入数据库中，请检查是否忘记了执行`commit()`方法。

为了执行某个查询，使用下面的语句：

```
for row in db.execute('select * from student'):
    print(row)
```

这里的`db.execute()`方法作为一种非标准用法，等价于先调用`cursor()`方法创建游标，然后调用游标的`execute()`方法并返回游标对象。

一般情况，你应该使用标准的用法：

```
c = db.cursor()
c.execute('select * from student')
print(c.description)
# 列名，类型，显示大小，内部大小，精度，范围，标志（指示是否接受null值）
# 因为sqlite3对插入到数据库的数据没有类型和大小约束，所以只填入列名值

print(c.fetchone())   # 返回查询结果集中的下一行
print(c.fetchmany(2)) # 返回查询结果集中下面size行，size小于等于0是，等价于fetchall
print(c.fetchall())   # 返回查询结果集中剩下的所有行
```

默认情况下，使用`fetch*`方法从数据库中返回的作为行(rows)的值是由元组构成的。调用者负责了解查询中列的顺序，并从元组中抽取单个的值。查询的值个数增加时，或者处理数据的代码分布在一个库的不同位置时，通常更容易的做法是处理一个对象，并使用其列名来访问值。

Connection对象有个`row_factory`属性，允许调用代码控制所创建对象的类型来表示查询结果集中的各行。sqlite3还包括一个`Row`类，这个类将被用作一个行工厂。可以通过`Row`实例使用列索引或名来访问值。

```
# change the row factory to use Row
db.row_factory = sqlite3.Row

c = db.cursor()
c.execute('select * from student')
name, age = c.fetchone()
row = c.fetchone()
print(row[0], row[1], sep=': ')
print(row.keys())
print(row['name'], row['age'], sep=': ')

print(name, age)
```

查询中使用变量

通过建立SQL语句的方式可能因为特殊字符无法正确转义导致SQL解析错误，更糟糕的可能会导致安全漏洞（SQL注入攻击，使得入侵者可以在数据库中执行任意的SQL语句）。

要在查询中使用动态值，正确的方法是利用随SQL指令一起传入`execute()`的宿主变量。SQL语句执行时，语句中的占位符值会替换为宿主变量的值。

SQLite支持两种形式的带占位符的查询，分别是**位置参数**和**命名参数**。

位置参数

问号`?`指示一个位置参数，将作为元组的一个成员传至`execute()`。

```
min_age = 11
for row in db.execute('select * from student where age >= ?',
                      (min_age,)):
    print(row)
```

命名参数

对于包含大量参数的更为复杂的查询，或者如果查询中某些参数会重复多次，则可以使用命名参数。命名参数前面有一个冒号作为前缀。

```
for row in db.execute('select * from student where age >= :age',
                      {'age': min_age}):
    print(row['name'], row['age'], sep=': ')
```

位置或命名参数都不需要加引号或转义，查询解释器会对它们做特殊处理。

查询参数可以在`select`、`insert`、`update`语句中使用。查询中字面量能够出现的位置都可以放置查询参数。

最后，将已经打开的连接（connection）或者游标（cursor），不需要时应该关掉它们：

```
c.close()
db.close()
```

Peewee

Peewee是个简单轻量的ORM（对象关系映射），易于使用，默认支持SQLite、MySQL和PostgreSQL。由于其功能比较简单，所以效率比**SQLAlchemy**要略高一些。

Model定义

Model类，字段（fields）和Model实例和数据库概念的对应关系：

名称	对应于
Model类	数据库表
字段实例	数据表中的列
Model实例	数据表中的行

下面的代码展示了使用Peewee定义Model，连接数据库并创建数据表：

```
import peewee as pw

db = pw.SqliteDatabase('people.db')

class Person(pw.Model):
    name = pw.CharField()
    birthday = pw.DateField()
    is_relative = pw.BooleanField()

    class Meta:
        database = db  # This model uses the "people.db" database


class Pet(pw.Model):
    owner = pw.ForeignKeyField(Person, related_name='pets')
    name = pw.CharField()
    animal_type = pw.CharField()

    class Meta:
        database = db


db.connect()
db.create_tables([Person, Pet])
```

存储数据

```
from datetime import date

uncle_bob = Person(name='Bob', birthday=date(1960, 1, 15), is_relative=True)
uncle_bob.save()  # 返回修改的行数
```

或者通过Model的`create()`方法来创建Model实例：

```
grandma = Person.create(name='Grandma', birthday=date(1935, 3, 1), is_relative=True)
herb = Person.create(name='Herb', birthday=date(1950, 5, 5), is_relative=False)
```

为了更新某一行数据，修改Model实例并调用`save()`方法来持久化修改：

```
grandma.name = 'Grandma L.'
grandma.save()
```

现在我们在数据库中存储了3个人，让我们给他们一些pets。

```
bob_kitty = Pet.create(owner=uncle_bob, name='Kitty', animal_type='cat')
herb_fido = Pet.create(owner=herb, name='Fido', animal_type='dog')
herb_mittens = Pet.create(owner=herb, name='Mittens', animal_type='cat')
herb_mittens_jr = Pet.create(owner=herb, name='Mittens Jr', animal_type='cat')
```

后来，Mittens生病死了，我们需要在数据库中删除它：

```
herb_mittens.delete_instance()  # 返回从数据库中删除的行数
```

Bob叔叔收养了Herb的Fido：

```
herb_fido.owner = uncle_bob
herb_fido.save()
bob_fido = herb_fido
```

检索数据

数据库的强项就是其允许我们通过查询检索数据。关系数据库特别适合检索查询。

获取单条记录

使用`SelectQuery.get()`从数据库中获取单条记录：

```
grandma_query = Person.select().where(Person.name == 'Grandma L.')
print(grandma_query.sql())
grandma = grandma_query.get()
type(grandma)
```

我们还可以使用等价的简写方式`Model.get()`：

```
grandma = Person.get(Person.name == 'Grandma L.')
```

获取多条记录

列举数据中的存储的人：

```
for person in Person.select():
    print(person.name, person.is_relative)
```

列出所有的猫和它们的主人：

```
query = Pet.select().where(Pet.animal_type == 'cat')
for pet in query:
    print(pet.name, pet.owner.name)
```

上面的查询中有一个大的问题：当我们访问`pet.owner.name`时并没有在原来的查询中选择查询它，Peewee将执行一次额外的查询来获取宠物的主人。

我们可以通过`join`同时查询宠物和人：

```
query = (Pet
         .select(Pet, Person)
         .join(Person)
         .where(Pet.animal_type == 'cat'))

for pet in query:
    print(pet.name, pet.owner.name)
```

获取Bob的所有宠物：

```
for pet in Pet.select().join(Person).where(Person.name == 'Bob'):
    print(pet.name)

for pet in Pet.select().where(Pet.owner == uncle_bob):
    print(pet.name)
```

通过`order_by()`语句确保上面查询到的宠物按照字母表序进行排序：

```
for pet in Pet.select().where(Pet.owner == uncle_bob).order_by(Pet.name):
    print(pet.name)
```

列出所有人，根据年龄，从小到大：

```
for person in Person.select().order_by(Person.birthday.desc()):
    print(person.name, person.birthday)
```

列出所有人以及他们宠物的一些信息：

```
for person in Person.select():
    print(person.name, person.pets.count(), 'pets')
    for pet in person.pets:
        print('    ', pet.name, pet.animal_type)
```

上面的代码又会导致额外查询问题，我们通过`JOIN`查询和聚合查询记录来避免这个问题：

```
subquery = Pet.select(pw.fn.COUNT(Pet.id)).where(Pet.owner == Person.id)
query = (Person
         .select(Person, Pet, subquery.alias('pet_count'))
         .join(Pet, pw.JOIN.LEFT_OUTER)
         .order_by(Person.name))
print(query.sql())

for person in query.aggregate_rows():
    print(person.name, person.pet_count, 'pets')
    for pet in person.pets:
        print('    ', pet.name, pet.animal_type)
```

尽管我们创建了一个子查询，但是只有一个查询真正地被执行了。

最后，让我们做一个复杂点的。获取所有人，他们的生日符合下面的要求之一：

- 晚于1940（grandma）
- 早于1960（bob）

```
d1940 = date(1940, 1, 1)
d1960 = date(1960, 1, 1)

query = (Person
         .select()
         .where((Person.birthday < d1940) | (Person.birthday > d1960)))

for person in query:
    print(person.name, person.birthday)
```

现在做相反的查询，某人的生日在1940和1960之间：

```
query = (Person
         .select()
         .where((Person.birthday > d1940) & (Person.birthday < d1960)))
for person in query:
    print(person.name, person.birthday)
```

最后一个查询。这次将使用一个SQL函数来查找名字是以大写或小写的`G`开头的人：

```
expression = (pw.fn.Lower(pw.fn.Substr(Person.name, 1, 1)) == 'g')
for person in Person.select().where(expression):
    print(person.name)
```

想了解更多的sqlite3核心函数，点击这里查看。数据库操作完了之后，关闭连接：

```
db.close()
```

上面的代码示例展示的仅仅是一些基础，你可以执行自己想要的更加复杂的查询。

所有其他的SQL语句也是支持的，比如：

- `group_by()`
- `having()`
- `limit()` and `offset()`

Redis

Redis是一个开源（BSD许可）的基于内存的键值对存储系统，常用作**数据库**（非关系数据库）、**缓存**和**消息中间件**。它支持多种类型的数据结构，如**字符串**、**散列（hashes）**、**列表**、**集合（sets）**、**有序集合（sorted sets）**、**位图（bitmaps）**、**地理位置**等，所以其常常被称为数据结构服务器。Redis支持**主从复制（replication）**，事务（transactions），不同级别的**磁盘持久化**等很多特性。

Redis属于人们常说的**NoSQL数据库**或者**非关系数据库**：Redis不使用表，它的数据库也没有预定义或者强制的方式来要求用户关联Redis存储的不同数据。

高性能键值缓存服务器**memcached**也经常被拿来与Redis进行比较：这两者都可用于存储键值映射，彼此的性能也相差无几，但是Redis能够自动以两种不同的方式将数据写入磁盘，并且Redis除了能存储普通的字符串键之外，还可以存储

其他4种数据结构，而memcached只能存储普通的字符串键。这些及其他不同使得Redis可以用于解决更为广泛的问题，既可以作为主数据库使用，又可以作为其他存储系统的辅助数据库使用。

一般来说，许多用户只会在Redis的性能或者功能是必要的情况下，才会将数据存储到Redis里面：如果程序对性能的要求不高，又或者因为费用原因而没办法将大量数据存储到内存里面，那么用户可能会选择使用关系数据库，或者其他非关系数据库。在实际中，读者应该根据自己的需求来决定是否使用Redis，并考虑是将Redis用作主存储还是辅助存储，以及如何通过复制、持久化和事务等手段保证数据的完整性。

一些数据库和缓存服务器的特性与功能：

名称	类型	数据存储选项	查询类型	附加功能
Redis	使用内存存储的非关系数据库	字符串、列表、集合、散列表、有序集合	每种数据类型都有自己的专属命令， 另外还有批量操作和不完全的事务支持	发布与订阅，主从复制，持久化，脚本（存储过程）
memcached	使用内存存储的键值缓存	键值之间的映射	创建命令、读取命令、更新命令、删除命令以及其他几个命令	为提升性能而设的多线程服务器
MySQL	关系数据库	每个数据库可以包含多个表，每个表可以包含多个行；可以处理多个表的视图；支持空间和第三方扩展	SELECT 、 INSERT 、 UPDATE 、 DELETE 、函数、存储过程	支持ACID性质（需要使用InnoDB），主从复制和主主复制
PostgreSQL	关系数据库	每个数据库可以包含多个表，每个表可以包含多个行；可以处理多个表的视图；支持空间和第三方扩展；支持可定制类型	SELECT 、 INSERT 、 UPDATE 、 DELETE 、内置函数、自定义的存储过程	支持ACID性质，主从复制，由第三方支持的多主复制

安装Redis

使用**linuxbrew**安装：

```
brew install redis
```

运行redis服务器：

```
redis-server
```

运行redis命令行客户端：

```
redis-cli
```

在虚拟环境中安装redis-py（redis的Python客户端）：

```
pip install redis
```

Redis数据结构简介

结构类型	结构存储的值	结构的读写能力
STRING	可以是字符串、整数或者浮点数	对整个字符串或者字符串的其中一部分执行操作；对整数和浮点数执行自增（increment）或者自减（decrement）操作
LIST	一个链表，链表上的每个节点都包含了一个字符串	从链表的两端推入或者弹出元素；根据偏移量对链表进行修剪（trim）；读取单个或者多个元素；根据值查找或者移除元素
SET	包含字符串的无序收集器（unordered collection），并且被包含的每个字符串都是独一无二、各不相同的	添加、获取、移除单个元素；检查一个元素是否存在于集合中；计算交集、并集、差集；从集合里面随机获取元素
HASH	包含键值对的无序散列表	添加、获取、移除单个键值对；获取所有键值对
ZSET（有序集合）	字符串成员（member）与浮点数分值（score）之间的有序映射，元素的排列顺序由分值的大小决定	添加、获取、删除单个元素；根据分值范围（range）或者成员来获取元素

Redis基本数据类型操作

这里只列举Redis列表和散列的基本操作，更多的操作命令请参看官方文档。

```
import redis

conn = redis.Redis()
conn.rpush('a', '1')  # a 就是要操作的键
conn.lrange('a', 0, -1)  # 表示从索引为0的元素到最后一个元素
# ['1']
conn.rpush('a', '2')
# 2L
conn.lrange('a', 0, -1)
# ['1', '2']
```

Redis还支持对于列表的其他类型的操作：

```
conn.lpush('a', '3')  # 将值推入到列表的左端
# 3L
conn.lindex('a', 0)  # 返回列表中第0个元素的值
# '3'
conn.rpush('a', *[4, 5, 6])  # 一次性推入3个元素
# 6L
conn.lrange('a', 0, -1)
# [b'3', b'1', b'2', b'4', b'5', b'6']
conn.ltrim('a', 1, 4)  # 对列表进行修剪，只保留索引从1到4的值
conn.lrange('a', 0, -1)
# [b'3', b'1', b'2', b'4']
conn.lpop('a')  # 移除并返回列表最左端的元素
conn.rpop('a')  # 移除并返回列表最右端的元素
```

在Redis中使用散列（字典），仅包含字符串，只能有一层结构，不能进行嵌套，可以使用散列来模拟文档数据库中的文档或者关系数据库里的行。

下面是使用散列来存储文章信息的例子：

```
+----------------------------+
 |     article:92617          |
 +--------+-------------------+
 | title  | Hello World       |
 +--------+-------------------+
 | link   | http://t.cn/kAueS |
 +--------+-------------------+
 | poster | user:86274        |
 +--------+-------------------+
 | time   | 1331382699.33     |
 +--------+-------------------+
 | votes  | 528               |
 +--------+-------------------+
```

针对散列的常见的操作：

```
conn.hset('d', 'a', 1)  # 给名字为d的键添加一个名字为a，值为1的字段
conn.hmset('d', {'b': 2, 'c': 3})
conn.hget('d', 'b')  # 获取字段b的值
conn.hmget('d', ['a', 'b'])
# [b'1', b'2']

conn.hgetall('d')
# {b'a': b'1', b'b': b'2', b'c': b'3'}

conn.hkeys('d')  # 获取所有字段的键
conn.hvals('d')  # 获取所有字段的值
conn.hlen('d')  # 返回字段的总数
conn.hsetnx('d', 'e', 4)  # 对字段中不存在的键赋值
```

关于Redis中其他数据结构的介绍，包括基本的操作，参考教材p182-189。

下面是使用Redis实现的取TOP N应用的例子：

取TOP N操作

列出分数和排名。使用传统的思路：

1. 创建一个游戏表，表结构如下：

```
CREATE TABLE `game_score`(
`id` int(10) unsigned NOT NULL AUTO_INCREMENT,
`game_id` smallint(6) unsigned NOT NULL,
`user_id` int(11) DEFAULT 0,
`create_time` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
`score` float NOT NULL,
PRIMARY KEY(`id`),
KEY `idx_score`(`game_id`, `score`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

2. 每当用户完成一次小游戏，就把对应的游戏id、用户和得分插入数据库：

```
insert into game_score(game_id, user_id, score) values (GAME_ID, USER_ID, SCORE)
```

3. 通过如下两个SQL语句计算得分的排名情况：

```
select count(1) from game_score where game_id = GAME_ID  # 获取完成游戏总次数
select count(1) from game_score where game_id = %s and score < SCORE
```

4. 由于插入和查询操作太频繁，对`COUNT`语句添加缓存，插入语句的执行放到异步队列，插入完成后会更新对应的缓存。

上面的做法实现起来不仅复杂，而且插入之后看到的排名由于缓存可能会有稍微的延迟。

如果使用Redis实现，将会比较简单：

```
import string
import random

import redis

r = redis.StrictRedis(host='localhost', port=6379, db=0)
GAME_BOARD_KEY = 'game.board'

# 插入1000条随机用户名和分数组成的记录。
for _ in range(1000):
    score = round((random.random() * 100), 2)
    user_id = ''.join(random.sample(string.ascii_letters, 6))
    r.zadd(GAME_BOARD_KEY, score, user_id)

# 随机获得一个用户和他的得分。
user_id, score = r.zrevrange(GAME_BOARD_KEY, 0, -1,
                             withscores=True)[random.randint(0, 200)]
print(user_id, score)

# 获取全部记录条目数
board_count = r.zcount(GAME_BOARD_KEY, 0, 100)
# 这个用户分数超过了多少用户
current_count = r.zcount(GAME_BOARD_KEY, 0, score)
print(current_count, board_count)

print('TOP 10')
print('-' * 20)

# 获取排行榜前10为的用户名和得分
for user_id, score in r.zrevrangebyscore(
        GAME_BOARD_KEY, 100, 0, start=0,
        num=10, withscores=True):
    print(user_id, score)
```

上面使用了Redis中的有序集合（sorted set或zset），其里面的值都是唯一的，每一个值都关联对应浮点值分数（score）。可以通过值或者分数取得每一项。

Redis是一个可以用来解决问题的工具，它拥有其他数据库所不具备的数据结构，并且因为它是内存数据库（这使得Redis的速度非常快），具有远程（这使得Redis可以连接多个客户端和服务器）、持久化（这使得服务器可以在重启之后仍然保持重启之前的数据）和可扩展（通过主从复制和分片）等多个特性，使得用户可以以熟悉的方式来为各种不同的问题构建解决方案。