第 3 章　PyMySQL 的安装和操作

第 2 章介绍了 MySQL 的基本操作，但没有引入 Python 操作 MySQL 的方式。本章将介绍 Python 如何通过 PyMySQL 连接关系型数据库，以 MySQL 作为关系型数据库进行操作示例讲解。

Python 通过 PyMySQL 操作 MySQL 类似学校通过工作人员安排 IT 大讲堂听讲人员的座次。

对于进入 IT 大讲堂听讲的人员座次安排，工作人员可以根据学校要求更换某排人员的位置，也可以替换某个座位的人员，这就如同对 MySQL 的更改操作，工作人员类似 PyMySQL，学校则相当于 Python。工作人员也可以将某排所有听众先撤空，类似 MySQL 中的删除操作，不影响其他排，也保持排列顺序。下面各节将进行更细致的讲解。

3.1　PyMySQL 的介绍与安装

为了使 Python 连接数据库，需要一个驱动，这个驱动是用于与数据库交互的库。在 Python 3.x 版本中，PyMySQL 是从 Python 连接到 MySQL 数据库服务器的接口，在 Python 2 中则使用 MySQLDB。PyMySQL 的目标是成为 MySQLDB 的替代品。

PyMySQL 是一个开源项目，支持如下 Python 版本：Python 2，Python 2.7，Python 3 及以上。PyMySQL 遵循 Python 数据库 API v2.0 规范，包含 pure-Python MySQL 客户端库。

在使用 PyMySQL 前，需要确保计算机上安装了 PyMySQL。如果没有安装 PyMySQL，在 Windows、Linux 或 Mac 系统下，都可以通过如下命令安装（使用 pip 或 pip3）：

```
pip install PyMySQL
```

怎么检查 PyMySQL 是否安装成功？检查比较简单，如在 Windows 系统中，可以按如下操作进行检查。打开命令提示符：

```
C:\Users\lyz>python
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 17:00:18) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

然后输入如下命令：

```
>>> import pymysql
>>>
```

若安装成功，则输入“import pymysql”命令后，光标会定位到下一行命令提示符，否则会提示如下错误信息：

```
>>> import pymysql
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
ModuleNotFoundError: No module named 'pymysql'
>>>
```

若在执行 import 语句时出现“ModuleNotFoundError: No module named 'pymysql'”这样的提示，则表示 pymysql 模块尚未安装，使用上面的安装语句进行安装即可。注意：使用 pip 安装模块时，可能需要管理员或 root 权限，安装时根据提示正确执行即可。

3.2 PyMySQL 连接 MySQL 数据库

PyMySQL 安装成功后，就可以连接 MySQL 数据库了。在连接之前有两个概念要先理解：连接对象和游标对象。

连接（Connect）对象：

```
class pymysql.connections.Connection(host=None, user=None, password='',
database=None, port=0, unix_socket=None, charset='', sql_mode=None, read_default_file=None,
conv=None, use_unicode=None, client_flag=0, cursorclass=<class 'pymysql.cursors.Cursor'>,
init_command=None, connect_timeout=10, ssl=None, read_default_group=None, compress=None,
named_pipe=None, autocommit=False, db=None, passwd=None, local_infile=False,
max_allowed_packet=16777216, defer_connect=False, auth_plugin_map=None, read_timeout=None,
write_timeout=None, bind_address=None, binary_prefix=False, program_name=None,
server_public_key=None)
```

用 MySQL 服务器表示套接字。

获取此类实例的正确方法是调用 connect()方法建立与 MySQL 数据库的连接。

连接对象中几个关键参数的解释如下（全部参数的解释参见附录 B）。

- host：数据库服务器所在的主机。
- user：以登录身份登录的用户名。
- password：要使用的密码。
- database：要使用的数据库，设置为 None，则指不使用特定的数据库。
- port：要使用的 MySQL 端口，默认即可（默认值为 3306）。

获取数据库连接的基本语法如下：

```
pymysql.connect(host, user, password, database, port)
```

其中的参数对应上面参数解释中的值。

一般使用连接对象时，还会使用连接对象的如下方法。

- close()：发送退出消息并关闭套接字。
- commit()：提交更改到稳定存储。
- cursor(cursor=None)：创建一个新游标以执行查询。cursor 参数指要创建的游标类型，即 Cursor、SSCursor、DictCursor、SSDictCursor 之一，None 指使用 Cursor。
- rollback()：回滚当前事务。

这里大概介绍，后面示例中使用时会有更详尽的描述。

游标（Cursor）对象：

```
class pymysql.cursors.Cursor(connection)
```

这是用于与数据库交互的对象。

不要自己创建 Cursor 实例，调用 connections.Connection.cursor()即可。

游标对象的方法如下。

- close()：关闭光标，会释放所有剩余数据。
- execute(query, args=None)：执行查询。其中，query（字符型）参数为需要执行的查询。args（元组、列表或字典类型）为与查询一起使用的参数（可选）。返回受影响的行数（如果有），返回数据的类型为 INT。
- fetchone()：获取一行。
- fetchall()：获取所有行。
- fetchmany(size=None)：获取指定的 size 行。

这里大概介绍，后面示例中使用时会有更详尽的描述。

下面是使用 PyMySQL 连接 MySQL 数据库的示例（mysql_conn_exp.py）：

```
import pymysql

# 打开数据库连接，不加端口号写法
db = pymysql.connect("localhost", "root", "root", "data_school")
# 使用数据库连接对象的 cursor()方法创建一个游标对象 cursor
cursor = db.cursor()
# 使用游标对象的 execute()方法执行 SQL 查询
cursor.execute("SELECT VERSION()")
# 使用游标对象的 fetchone()方法获取单条数据.
data = cursor.fetchone()
print(f"Database version:{data}")
# 关闭数据库连接
db.close()
```

执行程序，得到执行结果如下：

```
Database version:('8.0.11',)
```

对 mysql_conn_exp.py 中代码的解释如下。

代码 import pymysql：导入 pymysql 库。

代码 db=pymysql.connect("localhost", "root", "root", "data_school")：打开数据库连接，参数值对应如下：

- host 赋值为 localhost，因为连接的是本地，非本地连接要填写对应的 IP 地址。
- user 赋值为 root，用户名为 root。
- password 赋值为 root，这是一个权限最高的用户，实际应用中要慎用这么高权限的用户名和密码。
- database 赋值为 data_school，是在第 2 章中已经创建好的一个数据库，这里直接拿来使用。

该行代码也可以写成如下形式：

```
# 打开数据库连接，添加端口号写法
db = pymysql.connect("localhost", "root", "root", "data_school", 3306)
```

或写成如下形式，效果也是一样的。

```
# 打开数据库连接，显示指明参数名写法
db = pymysql.connect(host="localhost", user="root", password="root", database="data_school", port=3306)
```

代码 cursor = db.cursor()：用数据库连接对象的 cursor()方法创建一个游标对象 cursor。

代码 cursor.execute("SELECT VERSION()")：用游标对象的 execute()方法执行查询。

代码 data = cursor.fetchone()：用游标对象的 fetchone()方法获取单条数据。

代码 db.close()：关闭数据库连接。

这里是对 MySQL 数据库连接的简单介绍，接下来展示对 MySQL 数据库的更多操作。

3.3 PyMySQL 对 MySQL 数据库的基本操作

本节将介绍通过 PyMySQL 对 MySQL 数据库的增加、查询、更改、删除等操作，在第 2 章创建的 python_class 表上进行。本章所有代码存放在源码目录的 chapter3 目录下。

1. 数据库插入操作

现在需要通过编写 Python 代码把“小强”信息插入到 python_class 表中，“小强”信息为：number 为 1005，name 为小强，class_name 为 Python 快乐学习班。

实现示例代码如下（insert_exp_01.py）：

```
import pymysql

def insert_record():
    """
    插入数据
    :return:
    """
    # 打开数据库连接，添加端口号写法
    db = pymysql.connect("localhost", "root", "root", "data_school", 3306)
    # 使用 cursor()方法获取操作游标
    cursor = db.cursor()

    # SQL 插入语句
    sql = "INSERT INTO python_class(number,name, class_name) \
           VALUES ({}, '{}', '{}')".format(1005, '小强', 'Python 快乐学习班')
    try:
        # 执行 SQL 语句
        cursor.execute(sql)
        # 执行 SQL 语句
        db.commit()
    except:
        # 发生错误时回滚
        db.rollback()

    # 关闭数据库连接
    db.close()

if __name__ == "__main__":
    insert_record()
```

执行 insert_exp_01.py 文件前先查看 python_class 表中的数据情况：

```
mysql> SELECT * FROM python_class;
+----+--------+------+----------------+
| id | number | name | class_name     |
```

```
+----+--------+------+-----------------+
|  1 |   1001 | 小萌  | Python 快乐学习班 |
|  2 |   1002 | 小智  | Python 快乐学习班 |
|  4 |   1003 | 小强  | Python 快乐学习班 |
+----+--------+------+-----------------+
3 rows in set (0.00 sec)
```

执行 insert_exp_01.py 文件后，python_class 表中的数据情况如下：

```
mysql> SELECT * FROM python_class;
+----+--------+------+-----------------+
| id | number | name | class_name      |
+----+--------+------+-----------------+
|  1 |   1001 | 小萌  | Python 快乐学习班 |
|  2 |   1002 | 小智  | Python 快乐学习班 |
|  4 |   1003 | 小强  | Python 快乐学习班 |
|  5 |   1005 | 小张  | Python 快乐学习班 |
+----+--------+------+-----------------+
4 rows in set (0.00 sec)
```

由结果可知，小强的信息已成功插入 python_class 表中。

文件 insert_exp_01.py 的代码也可以写成 insert_exp_02.py 所示示例：

```
import pymysql

def insert_record():
    """
    插入数据
    :return:
    """
    # 打开数据库连接，添加端口号写法
    db = pymysql.connect("localhost", "root", "root", "data_school", 3306)
    # 使用 cursor()方法获取操作游标
    cursor = db.cursor()
    # SQL 插入语句
    sql = """INSERT INTO python_class(number,name, class_name)
             VALUES (1005, '小张', 'Python 快乐学习班')"""
    try:
        # 执行 SQL 语句
        cursor.execute(sql)
        # 提交到数据库执行
        db.commit()
    except:
        # 如果发生错误则回滚
        db.rollback()

    # 关闭数据库连接
    db.close()

if __name__ == "__main__":
    insert_record()
```

该代码的执行效果和 insert_exp_01.py 文件中代码执行效果一致。

在该示例代码中，不要忘记写 db.commit()这行代码，对于 MySQL 的更改操作，需

要显式做事务的提交操作，否则会导致数据库中数据未成功写入，即数据丢失。

2．数据库查询操作

如查看 python_class 表中 number 为 1002 的学生的所有信息，通过 Python 实现的示例代码如下（select_exp.py）：

```
import pymysql

def mysql_select():
    """
    数据查找
    :return:
    """
    # 打开数据库连接，添加端口号写法
    db = pymysql.connect("localhost", "root", "root", "data_school", 3306)
    # 使用 cursor()方法获取操作游标
    cursor = db.cursor()

    # 待查询的学号
    s_num = 1002
    # SQL 查询语句
    sql = "SELECT *  FROM python_class  WHERE number = {}".format(s_num)
    try:
        # 执行 SQL 语句
        cursor.execute(sql)
        # 获取所有记录列表
        results = cursor.fetchall()
        for row in results:
            num = row[1]
            name = row[2]
            cs_name = row[3]
            # 打印结果
            print(f"学号为{s_num}的详细信息为:" f"number={num},name={name},class_name={cs_name}")
    except:
        print("Error: unable to fetch data")

    # 关闭数据库连接
    db.close()

if __name__ == "__main__":
    mysql_select()
```

执行该示例代码，得到的结果如下：

```
学号为 1002 的详细信息为:number=1002,name=小智,class_name=Python 快乐学习班
```

3．数据库更新操作

若小张改名为小李，需要你在 python_class 表中将他的 name 值更改为小李，其他信息不变。实现示例代码如下（update_exp.py）：

```
import pymysql

def update_mysql():
    """
```

```
    数据更新
    :return:
    """
    # 打开数据库连接，添加端口号写法
    db = pymysql.connect("localhost", "root", "root", "data_school", 3306)
    # 使用 cursor()方法获取操作游标
    cursor = db.cursor()

    # SQL 更新语句
    sql = "UPDATE python_class SET name = '{}' WHERE name = '{}'".format('小李', '小张')
    try:
        # 执行 SQL 语句
        cursor.execute(sql)
        # 提交到数据库执行
        db.commit()
    except:
        # 发生错误时回滚
        db.rollback()

    # 关闭数据库连接
    db.close()

if __name__ == "__main__":
    update_mysql()
```

执行以上代码，可以从 MySQL 命令控制台查看执行结果。当然，前面已经学习了数据库查询操作，可以结合 Python 的函数编写方式，将 update_exp.py 中的代码改写为如下新的形式（update_exp_01.py）：

```
import pymysql

# 打开数据库连接，添加端口号写法
db = pymysql.connect("localhost", "root", "root", "data_school", 3306)
# 使用 cursor()方法获取操作游标
cursor = db.cursor()

def query_mysql(s_num):
    """
    根据条件查找数据
    :param s_num:
    :return:
    """
    # SQL 查询语句
    sql = "SELECT *  FROM python_class  WHERE number={}".format(s_num)
    try:
        # 执行 SQL 语句
        cursor.execute(sql)
        # 获取所有记录列表
        results = cursor.fetchall()
        for row in results:
            num = row[1]
            name = row[2]
            # 打印结果
```

```
            print(f"学号为{s_num}的详细信息为:number={num},name={name}")
    except:
        print("Error: unable to fetch data")

def update_mysql():
    """
    数据更新
    :return:
    """
    # SQL 更新语句
    sql = "UPDATE python_class SET name = '{}' WHERE name='{}'".format('小李', '小张')
    try:
        # 执行 SQL 语句
        cursor.execute(sql)
        # 提交到数据库执行
        db.commit()
    except:
        # 发生错误时回滚
        db.rollback()

if __name__ == "__main__":
    number = 1005
    print("--------------更改之前---------------")
    query_mysql(number)
    update_mysql()
    print("--------------更改之后---------------")
    query_mysql(number)
# 关闭连接
    db.close()
```

执行 update_exp_01.py 文件中的代码，得到输出结果如下：

```
--------------更改之前---------------
学号为 1005 的详细信息为:number=1005,name=小张
--------------更改之后---------------
学号为 1005 的详细信息为:number=1005,name=小李
```

4．删除操作

通过 Python 代码也可以对 MySQL 数据库进行删除操作。现在需要将 python_class 表中 number 为 1005 的记录删除，示例代码如下（delete_exp.py）：

```
import pymysql

# 打开数据库连接，添加端口号写法
db = pymysql.connect("localhost", "root", "root", "data_school", 3306)
# 使用 cursor()方法获取操作游标
cursor = db.cursor()

def query_mysql(s_num):
    """
    根据条件查找数据
    :param s_num:
    :return:
    """
```

```
        # SQL 查询语句
        sql = "SELECT *  FROM python_class  WHERE number={}".format(s_num)
        try:
            # 执行 SQL 语句
            cursor.execute(sql)
            # 获取所有记录列表
            results = cursor.fetchall()
            if results is None or len(results) == 0:
                print(f'没有找到学号为{s_num}的信息。')

            for row in results:
                num = row[1]
                name = row[2]
                # 打印结果
                print(f"学号为{s_num}的详细信息为:number={num},name={name}")
        except:
            print("Error: unable to fetch data")

    def delete_mysql(num):
    """
        根据指定条件做删除
        :param num:
        :return:
        """
        # SQL 删除语句
        sql = "DELETE FROM python_class  WHERE number={}".format(num)
        try:
            # 执行 SQL 语句
            cursor.execute(sql)
            # 提交修改
            db.commit()
        except:
            # 发生错误时回滚
            db.rollback()

    if __name__ == "__main__":
        number = 1005
        print('------删除之前--------')
        query_mysql(number)
        delete_mysql(number)
        print('------删除之后--------')
        query_mysql(number)
        # 关闭连接
        db.close()
```

执行该示例代码，得到执行结果如下：

```
------删除之前--------
学号为 1005 的详细信息为:number=1005,name=小李
------删除之后--------
没有找到学号为 1005 的信息。
```

5．执行事务

前面的示例代码中多处出现 db.rollback()这样的代码，含义是发生错误时回滚。为什么有些操作需要回滚，有些却不需要呢？这就涉及事务问题，而事务机制可以确保数据一致性。

事务具有 4 个属性：原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Durability）。这 4 个属性通常称为 ACID 特性，在第 1 章已经讲解。在对数据库表做操作时，对数据库表的插入、更新、删除操作都会涉及对数据的变更，为确保数据的一致性，一般这三个操作要在发生错误时做数据回滚，查询操作不会更改数据，不需要做数据的回滚。

对于支持事务的数据库，在 Python 数据库编程中，当游标建立时，就自动开始了一个隐形的数据库事务。

commit()方法中，游标的所有更新操作在遇到 rollback()方法时都会回滚当前游标的所有操作。每个 commit()方法都开始了一个新的事务。

6．错误处理

同 Python 程序一样，通过 Python 代码操作 MySQL 数据库时会经常遇到不同数据库层的异常，DB API 中定义了一些数据库操作的异常，如表 3-1 所示。

表 3-1　数据库操作的异常

异　常	描　　述
Warning	当有严重警告时触发，如插入数据时被截断等，必须是 StandardError 的子类
Error	警告以外所有其他错误类，必须是 StandardError 的子类
InterfaceError	当有数据库接口模块本身的错误（而不是数据库的错误）发生时触发，必须是 Error 的子类
DatabaseError	与数据库有关的错误发生时触发，必须是 Error 的子类
DataError	当有数据处理时的错误发生时触发，如除零错误、数据超范围等，必须是 DatabaseError 的子类
OperationalError	指非用户控制的而是操作数据库时发生的错误，如连接意外断开、数据库名未找到、事务处理失败、内存分配错误等操作数据库时发生的错误，必须是 DatabaseError 的子类
IntegrityError	完整性相关的错误，如外键检查失败等，必须是 DatabaseError 的子类
InternalError	数据库的内部错误，如游标（cursor）失效了、事务同步失败等，必须是 DatabaseError 子类
ProgrammingError	程序错误，如数据表（table）没找到或已存在、SQL 语句语法错误、参数数量错误等，必须是 DatabaseError 的子类
NotSupportedError	不支持错误，指使用了数据库不支持的函数或 API 等。例如在连接对象上使用.rollback()函数，然而数据库并不支持事务或者事务已关闭。必须是 DatabaseError 的子类

3.4　PyMySQL 操作多表

到目前为止，所有操作都是在单表上操作的，实际应用中，在一个 SQL 语句中操作多张表的操作是比较常见的。本节介绍 Python 代码通过 PyMySQL 操作 MySQL 中多张表的操作。

在 data_school 库中准备另一个表：学生地址表 st_addr，包含学生学号、学生家庭住址等信息。st_addr 表的创建通过 Python 代码实现，实现如下（create_table_exp.py）：

```
import pymysql
```

```
def create_table():
    """
    创建表
    :return:
    """
    # 打开数据库连接，添加端口号写法
    db = pymysql.connect("localhost", "root", "root", "data_school", 3306)
    # 使用 cursor()方法获取操作游标
    cursor = db.cursor()

    # 使用预处理语句创建表
    sql = """CREATE table st_addr (id INT UNSIGNED AUTO_INCREMENT,
                                   number INT(10) NOT NULL,
                                   addr VARCHAR(100) NOT NULL,
                                   PRIMARY KEY (id)
                                  )ENGINE=InnoDB DEFAULT CHARSET=utf8"""

    cursor.execute(sql)

    # 关闭数据库连接
    db.close()

if __name__ == "__main__":
    create_table()
```

执行代码后，通过指令面板查看 MySQL 数据库表，结果如下：

```
mysql> SHOW tables;
+-----------------------+
| Tables_in_data_school |
+-----------------------+
| python_class          |
| st_addr               |
+-----------------------+
2 rows in set (0.01 sec)
```

由结果可知，data_school 中已经新增了一个名为 st_addr 的表。

创建 st_addr 表后，用批量插入的方式向表中插入几条数据，代码如下（query_exp.py）：

```
import pymysql

def query_record():
    """
    记录查询
    :return:
    """
    # 打开数据库连接，不添加端口号写法
    db = pymysql.connect("localhost", "root", "root", "data_school")
    # 使用 cursor()方法获取操作游标
    cursor = db.cursor()

    # SQL 查询语句
    sql = "SELECT * FROM st_addr"
    try:
```

```
            # 执行 SQL 语句
            cursor.execute(sql)
            # 获取所有记录列表
            results = cursor.fetchall()
            for row in results:
                id_v = row[0]
                num = row[1]
                addr = row[2]
                # 打印结果
                print(f"详细信息为:id={id_v},number={num},addr={addr}")
        except:
            print("Error: unable to fetch data")

        # 关闭数据库连接
        db.close()

    if __name__ == "__main__":
        query_record()
```

执行代码，得到结果如下：

```
详细信息为:id=1,number=1001,addr=A 城区
详细信息为:id=2,number=1002,addr=B 城区
详细信息为:id=3,number=1003,addr=C 城区
```

由执行结果可知，批量插入成功，数据查询也成功。

现在需要查找 number 为 1002 的同学的 name 和 addr，怎样能更方便查找呢？

这里需要引进 MySQL 连接的使用。“连接”概念在第 2 章有提到，但没有展开，这里对这个概念做一个补充。

MySQL 中一般通过 JOIN 在两个或多个表中查询数据，即一般通过 JOIN 做连接查询。按照功能，JOIN 大致分为如下三类：

① INNER JOIN（内连接，或等值连接，可以直接写成 JOIN）：获取两个表中字段匹配关系的记录。比如，A、B 两张表做内连接，查询到的记录是既在 A 表又在 B 表的记录，相当于数学中两个集合的交集。

② LEFT JOIN（左连接）：获取左表所有记录，即使右表没有对应匹配的记录。比如 A、B 两张表做左连接，A 为左表，B 为右表，左连接得到的查询结果是 A 表所有满足条件的记录都显示，B 表满足条件的显示对应结果，不满足条件的显示空，保持和 A 表查找到的记录条数一致。即以 A 表为标准，B 表不够的以空填补。

③ RIGHT JOIN（右连接）：与 LEFT JOIN 相反，用于获取右表所有记录，即使左表没有对应匹配的记录。比如，A、B 两张表做右连接，A 为左表，B 为右表，右连接得到的查询结果是 B 表所有满足条件的记录都显示，A 表满足条件的显示对应结果，不满足条件的显示空，保持与 B 表查找到的记录条数一致。即以 B 表为标准，A 表不够的以空填补。

若想知道更多，读者可以自行查找相关资料。

查找 number 为 1002 的同学的 name 和 addr 这个需求需要通过内连接来实现，在 MySQL 指令面板中的写法及结果如下：

```
mysql> SELECT a.number,a.name,b.addr  FROM python_class a join st_addr b on a.number=b.number
```

```
and a.number=1002;
       +--------+--------+---------+
       | number | name | addr  |
       +--------+------+-------+
       |   1002 | 小智  | B 城区 |
       +--------+------+-------+
       1 row in set (0.00 sec)
```

由结果可知，使用 JOIN 得到了想要的结果。这里需要注意后面的条件开始用的是 ON，不是 WHERE。

接下来看 Python 代码的实现方式，示例如下（mult_table_query_exp.py）：

```
import pymysql

def query_mysql(num):
    """
    记录查询
    :return:
    """
    # 打开数据库连接，不添加端口号写法
    db = pymysql.connect("localhost", "root", "root", "data_school")
    # 使用 cursor()方法获取操作游标
    cursor = db.cursor()

    # SQL 查询语句
    sql = "SELECT a.number,a.name,b.addr FROM python_class a " \
          "JOIN st_addr b ON a.number=b.number AND a.number={}".format(num)

    try:
        # 执行 SQL 语句
        cursor.execute(sql)
        # 获取所有记录列表
        results = cursor.fetchall()
        for row in results:
            num_v = row[0]
            name = row[1]
            addr = row[2]
            # 打印结果
            print(f"学号为{num}的详细信息为:number={num_v},name={name},addr={addr}")
    except:
        print("Error: unable to fetch data")

    # 关闭数据库连接
    db.close()

if __name__ == "__main__":
    query_mysql(1002)
```

执行以上代码，得到执行结果如下：

```
学号为 1002 的详细信息为:number=1002,name=小智,addr=B 城区
```

由结果可知，以上代码已实现多表的连接操作。

多表的连接操作还支持更改和删除等操作，此处不具体举例，大家可以自行尝试。

3.5 高级封装

从 3.3 节和 3.4 节的代码中可以看到，每个 PY 文件中都出现了不少需要重复编写的代码段，如下面两段代码，几乎每个 PY 文件中都要编写一遍：

```
# 打开数据库连接，添加端口号写法
db = pymysql.connect("localhost", "root", "root", "data_school", 3306)
# 使用 cursor()方法获取操作游标
cursor = db.cursor()

# 关闭数据库连接
db.close()
```

那么，这些代码是否可以封装到一个 PY 文件中，从而在需要时直接调用呢？答案是可以的，现在在 chapter3 目录下创建一个名为 common 的目录，在其下创建一个名为 mysql_conn.py 的文件。在 mysql_conn.py 中按如下方式添加代码。创建一个名为 MySQLConnection 的类，代码如下：

```
class MySQLConnection(object):
```

观察前面编写的打开数据库连接的语句，在 MySQLConnection 类中创建一个初始化方法，方法定义如下：

```
def __init__(self, host=None, user=None, password=None, database=None, port=3306):
    self.host = host
    self.user = user
    self.password = password
    self.database = database
    self.port = port
```

其中，5 个参数分别对应 host、user、password、database 和 port。port 默认值为 3306，即 MySQL 连接的默认端口号。这 5 个参数对应 MySQL 数据库连接的 5 个基本参数，除了 port 参数，缺少其中任何一个参数都不能连接 MySQL。

这里定义了参数初始化方法，但只是参数，还需打开数据库连接。

在 MySQLConnection 类中增加如下代码：

```
def get_db(self):
    """
    打开数据库连接
    :return:
    """
    db = pymysql.connect(self.host, self.user, self.password, self.database, self.port)
    return db
```

get_db()方法中实现了打开数据库连接的方式，并最终返回数据库连接对象。得到数据库连接对象后，还需要获取操作游标，但获取操作游标时需要数据库连接对象，所以需要在__init__()方法中初始化数据库连接对象。__init__()方法更改如下：

```
def __init__(self, host=None, user=None, password=None, database=None, port=3306):
    self.host = host
    self.user = user
    self.password = password
    self.database = database
```

```
        self.port = port
        self.db = self.get_db()
```

增加一行初始化获取数据库连接对象的代码。同时，在 MySQLConnection 类中增加如下方法：

```
    def conn(self):
        """
        使用 cursor()方法获取操作游标
        :return:
        """
        cursor = self.db.cursor()
        return cursor
```

由 conn()方法即可得到操作游标。得到操作游标后，需要在 MySQLConnection 类初始化时先初始化，在_init__()方法中增加一行代码。_init__()方法更改如下：

```
    def __init__(self, host=None, user=None, password=None, database=None, port=3306):
        self.host = host
        self.user = user
        self.password = password
        self.database = database
        self.port = port
        self.db = self.get_db()
        self.conn = self.conn
```

至此，MySQLConnection 类的初始化就完成了。

当然，在实际应用中，close()方法也是重复编写比较多的，把 close()方法也封装到类 MySQLConnection 中。在 MySQLConnection 类中添加 close()方法的代码如下：

```
    def close(self):
        """
        关闭数据库连接
        :return:
        """
        self.db.close()
```

这些代码添加完成后，MySQLConnection 类的代码结构如下：

```
    import pymysql

    class MySQLConnection(object):
        """
        MySQL 连接
        """
        def __init__(self, host=None, user=None, password=None, database=None, port=3306):
            self.host = host
            self.user = user
            self.password = password
            self.database = database
            self.port = port
            self.db = self.get_db()
            self.conn = self.conn
```

```
    def get_db(self):
        """
        打开数据库连接
        :return:
        """
        db = pymysql.connect(self.host, self.user, self.password, self.database, self.port)
        return db

    def conn(self):
        """
        使用 cursor()方法获取操作游标
        :return:
        """
        cursor = self.db.cursor()
        return cursor

    def close(self):
        """
        关闭数据库连接
        :return:
        """
        self.db.close()
```

这种形式封装好的代码可以实现数据库连接的复用，还可以进一步封装，可以把数据查询和数据更新的方法也加以封装，并在封装的查询和更新方法中关闭数据库连接，从而不需在外部调用时关心数据库连接是否关闭的问题。

数据查询分为全量查询和查询一条数据。全量查询代码封装如下：

```
def query_all(self, query_sql):

    """
    根据查询语句查询所有数据
    :param query_sql:
    :return:
    """
    try:
        cursor = self.conn()
        # 执行 SQL 语句
        cursor.execute(query_sql)
        # 获取所有记录列表
        results = cursor.fetchall()
        return results
    except Exception as ex:
        print("query all Error: {}".format(ex))
    finally:
        self.close()
```

查询一条数据代码封装如下：

```
def query_one(self, query_sql):
    """
```

```
        根据查询语句查询一条语句
        :param query_sql:
        :return:
        """
        try:
            cursor = self.conn()
            # 执行 SQL 语句
            cursor.execute(query_sql)
            # 获取一条记录
            results = cursor.fetchone()
            return results
        except Exception as ex:
            print("query all Error: {}".format(ex))
        finally:
            self.close()
```

数据更新代码封装如下：

```
    def update(self, update_sql):
        """
        根据更新语句更新数据
        :param update_sql:
        :return:
        """
        try:
            cursor = self.conn()
            # 执行 SQL 语句
            cursor.execute(update_sql)
            # 提交到数据库执行
            self.db.commit()
        except Exception as ex:
            print("update Error: {}".format(ex))
            # 发生错误时回滚
            self.db.rollback()
        finally:
            self.close()
```

MySQLConnection 类的完整代码如下（mysql_conn.py）：

```
    import pymysql

    class MySQLConnection(object):
        """
        MySQL 连接
        """
        def __init__(self, host=None, user=None, password=None, database=None, port=3306):
            self.host = host
            self.user = user
            self.password = password
            self.database = database
            self.port = port
            self.db = self.get_db()
            self.conn = self.conn
```

```
    def get_db(self):
        """
        打开数据库连接
        :return:
        """
        db = pymysql.connect(self.host, self.user, self.password, self.database, self.port)
        return db

    def conn(self):
        """
        使用 cursor()方法获取操作游标
        :return:
        """
        cursor = self.db.cursor()
        return cursor

    def close(self):
        """
        关闭数据库连接
        :return:
        """
        self.db.close()

    def query_all(self, query_sql):
        """
        根据查询语句查询所有数据
        :param query_sql:
        :return:
        """
        try:
            cursor = self.conn()
            # 执行 SQL 语句
            cursor.execute(query_sql)
            # 获取所有记录列表
            results = cursor.fetchall()
            return results
        except Exception as ex:
            print("query all Error: {}".format(ex))
        finally:
            self.close()

    def query_one(self, query_sql):
        """
        根据查询语句查询一条语句
        :param query_sql:
        :return:
        """
        try:
            cursor = self.conn()
            # 执行 SQL 语句
            cursor.execute(query_sql)
```

```
            # 获取一条记录
            results = cursor.fetchone()
            return results
        except Exception as ex:
            print("query all Error: {}".format(ex))
        finally:
            self.close()

    def update(self, update_sql):
        """
        根据更新语句更新数据
        :param update_sql:
        :return:
        """
        try:
            cursor = self.conn()
            # 执行 SQL 语句
            cursor.execute(update_sql)
            # 提交到数据库执行
            self.db.commit()
        except Exception as ex:
            print("update Error: {}".format(ex))
            # 发生错误时回滚
            self.db.rollback()
        finally:
            self.close()
```

到此为止，数据库连接、数据查询和数据更新的代码封装完成，接下来看如何使用。数据查询文件 select_exp.py 内容更改如下（select_exp_01.py）：

```
from chapter3.common.mysql_conn import MySQLConnection

def mysql_select():
    """
    数据查找
    :return:
    """
    # 待查询的学号
    s_num = 1002
    # SQL 查询语句
    sql = "SELECT * FROM python_class WHERE number = {}".format(s_num)
    try:
        # 获得数据库连接对象及游标
        conn = MySQLConnection("localhost", "root", "root", "data_school")
        # 执行 SQL 语句，并获取所有记录列表
        results = conn.query_all(sql)
        for row in results:
            num = row[1]
            name = row[2]
            cs_name = row[3]
            # 打印结果
            print(f"学号为{s_num}的详细信息为:"
```

```
                f"number={num},name={name},class_name={cs_name}")
    except:
        print("Error: unable to fetch data")

if __name__ == "__main__":
    mysql_select()
```

在 select_exp_01.py 文件中不再需要导入 pymysql 库，直接从 mysql_conn 中导入 MySQLConnection 类即可。初始化 MySQLConnection 类即获得数据库连接对象及游标。

通过代码可知，已经不需要编写获取数据库连接、游标的代码了，也不需要编写数据库关闭的代码。

数据插入文件 insert_exp_02.py 内容更改如下（insert_exp_03.py）：

```
from chapter3.common.mysql_conn import MySQLConnection

def get_conn():
    # 获得数据库连接对象及游标
    conn = MySQLConnection("localhost", "root", "root", "data_school")
    return conn

def insert_record():
    """
    插入数据
    :return:
    """
    # SQL 插入语句
    sql = """INSERT INTO python_class(number,name, class_name)
             VALUES (1006, '小王', 'Python 快乐学习班')"""
    try:
        # 执行 SQL 语句
        get_conn().update(sql)
        print('记录插入成功。')
    except Exception as ex:
        print("update Error: {}".format(ex))

if __name__ == "__main__":
    insert_record()
```

数据更改文件 updata_exp_01.py 内容更改如下（updata_exp_02.py）：

```
from chapter3.common.mysql_conn import MySQLConnection

def get_conn():
    # 获得数据库连接对象及游标
    conn = MySQLConnection("localhost", "root", "root", "data_school")
    return conn

def query_mysql(s_num):
    """
    根据条件查找数据
    :param s_num:
    :return:
```

```
    """
    # SQL 查询语句
    sql = "SELECT * FROM python_class WHERE number={}".format(s_num)
    try:
        # 执行 SQL 语句，并获取所有记录列表
        results = get_conn().query_all(sql)
        for row in results:
            num = row[1]
            name = row[2]
            # 打印结果
            print(f"学号为{s_num}的详细信息为:number={num},name={name}")
    except Exception as ex:
        print("query Error: {}".format(ex))

def update_mysql():
    """
    数据更新
    :return:
    """
    # SQL 更新语句
    sql = "UPDATE python_class SET name = '{}'  WHERE name='{}'".format('小李', '小张')
    try:
        # 执行 SQL 语句
        get_conn().update(sql)
        # 提交到数据库执行
    except Exception as ex:
        print("update Error: {}".format(ex))

if __name__ == "__main__":
    number = 1005
    print("--------------更改之前---------------")
    query_mysql(number)
    update_mysql()
    print("--------------更改之后---------------")
    query_mysql(number)
```

限于篇幅，本书暂时介绍这几种封装方式。

这里介绍的高级封装也只是一种比较简单的封装方式，大家有兴趣可以探索更简洁和抽象的封装方式。如对数据库访问的 IP 地址、用户名、密码、数据库等内容的读取采用文件配置方式配置，封装一个方法去读取配置文件，从而避免数据库连接的敏感信息泄露。

3.6 小结

本章主要讲解了 Python 通过 PyMySQL 操作 MySQL 数据库的各种基本操作，都是一些实战性的操作。

在本章学习过程中，学有余力的同学可以对本章的内容再做进一步的封装，如加上数据库连接池的封装、对大表操作的封装、对数据库读取超时的封装等。

数据库的操作在数据处理中是非常关键的一环，如果对数据库的操作不熟悉，在数

据处理中就会走很多弯路。需要与数据打交道的工作者掌握好数据库处理技术是非常有必要的。

3.7 实战演练

1．完成 PyMySQL 环境安装。

2．通过 PyMySQL 连接第 2 章中创建的数据库，并打印其中一张表的记录。

3．使用 PyMySQL 对 python_class 表中记录进行插入、删除、修改操作。

4．结合前面所学，查找网络资源，编写 Python 代码，使用 PyMySQL 对表数据进行排序、分组等操作。

5．参照 3.4 节的示例，编写代码实现多表操作，并打印出多表操作的结果。

6．根据 3.5 节的高级封装，将实战 5 的代码进行封装。若根据示例代码封装，再思考是否有其他操作可以进行封装。