数据库开发

驱动

MySQL基于TCP协议之上开发，但是网络连接后，传输的数据必须遵循MySQL的协议。
封装好MySQL协议的包，就是驱动程序。

MySQL的驱动

- MySQLdb
  最有名的库。对MySQL的C Client封装实现，支持Python 2，不更新了，不支持Python3
- MySQL官方Connector
- pymysql
  语法兼容MySQLdb，使用Python写的库，支持Python 3

pymysql使用

安装

```
$ pip install pymysql
```

创建数据库和表

```
CREATE DATABASE IF NOT EXISTS school;
SHOW DATABASES;
USE school;

CREATE TABLE `student` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(30) NOT NULL,
  `age` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

连接Connect

首先，必须建立一个传输数据通道——连接。
pymysql.connect()方法返回的是Connections模块下的Connection类实例。connect方法传参就是给Connection类的`__init__`提供参数

Connection初始化常用参数	说明
host	主机
user	用户名
password	密码
database	数据库
port	端口

Connection.ping()方法，测试数据库服务器是否活着。有一个参数reconnect表示断开与服务器连接是否重连。连接关闭抛出异常。

```
import pymysql

conn = None
try:
    conn = pymysql.connect('192.168.142.135', 'wayne', 'wayne', 'school')
    conn.ping(False) # ping不通则抛异常
finally:
    if conn:
        conn.close()
```

游标Cursor

操作数据库，必须使用游标，需要先获取一个游标对象。
Connection.cursor(cursor=None) 方法返回一个新的游标对象。
连接没有关闭前，游标对象可以反复使用。

cursor参数，可以指定一个Cursor类。如果为None，则使用默认Cursor类。

操作数据库

数据库操作需要使用Cursor类的实例，提供execute() 方法，执行SQL语句，成功返回影响的行数。

新增记录

使用insert into语句插入数据。

```
import pymysql

try:
    conn = pymysql.connect('192.168.142.135', 'wayne', 'wayne', 'school')
    conn.ping(False)

    cursor = conn.cursor()
    insert_sql = "insert into student (name,age) values('tom',20)"
    rows = cursor.execute(insert_sql)
    print(rows)
finally:
    if conn:
        conn.close()
```

发现数据库中没有数据提交成功，为什么?
原因在于，在Connection类的`__init__`方法的注释中有这么一句话
autocommit: Autocommit mode. None means use server default. (default: False)
那是否应该开启自动提交呢?
不用开启，一般我们需要手动管理事务。

事务管理

Connection类有三个方法:
begin 开始事务
commit 将变更提交
rollback 回滚事务

批量增加数据

```
import pymysql

conn = None
cursor = None
try:
    conn = pymysql.connect('192.168.142.135', 'wayne', 'wayne', 'school')
    print(conn)
    conn.ping(False) # ping不通则抛异常

    cursor = conn.cursor()
    for i in range(10):
        insert_sql = "insert into student (name, age) value('tom{}',{})".format(i+1, 20+i)
        rows = cursor.execute(insert_sql)

    conn.commit() # 提交
except:
    conn.rollback() # 回滚
finally:
    if cursor:
        cursor.close()
    if conn:
        conn.close()
```

使用executemany()方法，后面说

一般流程

- 建立连接
- 获取游标
- 执行SQL
- 提交事务
- 释放资源

查询

Cursor类的获取查询结果集的方法有fetchone()、fetchmany(size=None)、fetchall()。

```
import pymysql

conn = pymysql.connect('192.168.142.135', 'wayne', 'wayne', 'school')
cursor = conn.cursor()

sql = 'select * from student'
rows = cursor.execute(sql) # 返回影响的行数

print(cursor.fetchone())
print(cursor.fetchone())
print('1 -----------')
print(cursor.fetchmany(2))
print('2 ~~~~~~~~~~~')
print(cursor.fetchmany(2))
print('3-----------')
print(cursor.fetchall())


if cursor:
    cursor.close()
if conn:
    conn.close()
```

fetchone()方法，获取结果集的下一行
fetchmany(size=None)方法，size指定返回的行数的行，None则返回空元组。
fetchall()方法，获取所有行。
返回多行，如果走到末尾，就返回空元组，否则返回一个元组，其元素就是每一行的记录。
每一行的记录也封装在一个元组中。

cursor.rownumber 返回当前行号。可以修改，支持负数。
cursor.rowcount 返回的总行数

注意：fetch操作的是结果集，结果集是保存在客户端的，也就是说fetch的时候，查询已经结束了。

带列名查询

Cursor类有一个Mixin的子类DictCursor。
只需要 `cursor = conn.cursor(DictCursor)` 就可以了

```
# 返回结果
{'name': 'tom', 'age': 20, 'id': 4}
{'name': 'tom0', 'age': 20, 'id': 5}
```

返回一行，是一个字典。
返回多行，放在列表中，元素是字典，代表一行。

SQL注入攻击

找出用户id为6的用户信息的SQL语句如下
`SELECT * from student WHERE id = 6`
现在，要求可以找出某个id对应用户的信息，代码如下

```
userid = 5 # 用户id可以变
sql = 'SELECT * from student WHERE id = {}'.format(userid)
```

userid可以变，例如从客户端request请求中获取，直接拼接到查询字符串中。
可是，如果userid = '5 or 1=1'呢?
`sql = 'SELECT * from student WHERE id = {}'.format('5 or 1=1')`
运行的结果竟然是返回了全部数据。

SQL注入攻击
猜测后台数据库的查询语句使用拼接字符串等方式，从而经过设计为服务端传参，令其拼接出特殊字符串的SQL语句，返回攻击者想要的结果。

永远不要相信客户端传来的数据是规范及安全的！！！

如何解决注入攻击?
参数化查询，可以有效防止注入攻击，并提高查询的效率。

Cursor.execute(query, args=None)
args，必须是元组、列表或字典。如果查询字符串使用%(name)s，就必须使用字典。

```
import pymysql
from pymysql.cursors import DictCursor

conn = pymysql.connect('192.168.142.135', 'wayne', 'wayne', 'school')
cursor = conn.cursor(DictCursor)

userid = '5 or 1=1'
sql = 'SELECT * from student WHERE id = %s'
cursor.execute(sql, (userid,)) # 参数化查询
print(cursor.fetchall())
print('~~~~~~~~~~~~~~~~~')
sql = 'SELECT * from student WHERE name like %(name)s and age > %(age)s'
cursor.execute(sql, {'name':'tom%', 'age':25}) # 参数化查询
print(cursor.fetchall())

if cursor:
    cursor.close()
if conn:
```

```
    conn.close()

# 运行结果
\Python35\lib\site-packages\pymysql\cursors.py:323: Warning: (1292, "Truncated incorrect DOUBLE
value: '5 or 1=1'")
  self._do_get_result()
[{'age': 20, 'id': 5, 'name': 'tom0'}]
```

参数化查询为什么提高效率?
原因就是——SQL语句缓存。
数据库服务器一般会对SQL语句编译和缓存，编译只对SQL语句部分，所以参数中就算有SQL指令也不会被执行。
编译过程，需要词法分析、语法分析、生成AST、优化、生成执行计划等过程，比较耗费资源。
服务端会先查找是否对同一条查询语句进行了缓存，如果缓存未失效，则不需要再次编译，从而降低了编译的成本，降低了内存消耗。
可以认为SQL语句字符串就是一个key，如果使用拼接方案，每次发过去的SQL语句都不一样，都需要编译并缓存。
大量查询的时候，首选使用参数化查询，以节省资源。
开发时，应该使用参数化查询。
注意：这里说的是查询字符串的缓存，不是查询结果的缓存。

批量执行executemany()

```
import pymysql

conn = None
cursor = None
try:
    conn = pymysql.connect('192.168.142.135', 'wayne', 'wayne', 'school')
    cursor = conn.cursor()

    sql = "insert into student (name, age) values(%s, %s)"
    cursor.executemany(sql, (
        ('jerry{}'.format(i), 30 + i) for i in range(5)
    ))

    conn.commit()
except Exception as e:
    print(e)
    conn.rollback()
finally:
    if cursor:
        cursor.close()
    if conn:
        conn.close()
```

上下文支持

查看连接类和游标类的源码

```
# 连接类
class Connection(object):
    def __enter__(self):
```

```
        """Context manager that returns a Cursor"""
        return self.cursor()

    def __exit__(self, exc, value, traceback):
        """On successful exit, commit. On exception, rollback"""
        if exc:
            self.rollback()
        else:
            self.commit()

# 游标类
class Cursor(object):
    def __enter__(self):
        return self

    def __exit__(self, *exc_info):
        del exc_info
        self.close()
```

连接类进入上下文的时候会返回一个游标对象，退出时如果没有异常会提交更改。
游标类也使用上下文，在退出时关闭游标对象。

```
import pymysql

conn = pymysql.connect('192.168.142.135', 'wayne', 'wayne', 'school')
try:
    with conn.cursor() as cursor:
        for i in range(3):
            insert_sql = "insert into student (name,age) values('tom{0}',20+{0})".format(i)
            rows = cursor.execute(insert_sql)
    conn.commit()
    # 如果此时使用这个关闭的cursor，会抛异常
    # sql = "select * from student"
    # cursor.execute(sql) # cursor 已经关闭了
    # print(cursor.fetchall())
except Exception as e:
    print(e)
    conn.rollback()
finally:
    conn.close()
```

换一种写法，使用连接的上下文

```
import pymysql

conn = pymysql.connect('192.168.142.135', 'wayne', 'wayne', 'school')

with conn as cursor:
    for i in range(3):
        insert_sql = "insert into student (name,age) values('tom{0}',20+{0})".format(i)
```

```
        rows = cursor.execute(insert_sql)

sql = "select * from student"
cursor.execute(sql)
print(cursor.fetchall())

# 关闭
cursor.close()
conn.close()
```

conn的with进入是返回一个新的cursor对象，退出时，只是提交或者回滚了事务。并没有关闭cursor和conn。不关闭cursor就可以接着用，省的反复创建它。

如果想关闭cursor对象，这样写

```
import pymysql

conn = pymysql.connect('192.168.142.135', 'wayne', 'wayne', 'school')

with conn as cursor:
    with cursor:
        sql = "select * from student"
        cursor.execute(sql)
        print(cursor.fetchall())

# 关闭
conn.close()
```

通过上面的实验，我们应该知道，连接应该不需要反反复复创建销毁，应该是多个cursor共享一个conn。