

数据库开发

驱动

与MySQL通信就是典型的CS模式。Server就是服务器端，使用客户端先建立**连接**，数据库编程时，这个客户端变成了程序。

MySQL基于TCP协议之上开发，传输的数据必须遵循MySQL的协议。
封装好MySQL协议的包，习惯上称为驱动程序。

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
- mysqlclient
 - 在MySQLdb的基础上，增加了对Python 3的支持
- MySQL官方Connector
 - Mysql官网 <https://dev.mysql.com/downloads/connector/>
- pymysql
 - 语法兼容MySQLdb，使用纯Python写的MySQL客户端库，支持Python 3
 - CPython 2.7、3.4+
 - MySQL 5.5+、MariaDB 5.5+

pymysql使用

安装

```
1 $ pip install pymysql
2 $ pip install simplejson
```

simplejson库处理json文件方便。

创建数据库和表

```
1 CREATE DATABASE IF NOT EXISTS school;
2 SHOW DATABASES;
3 USE school;
4
5 CREATE TABLE `student` (
6   `id` int(11) NOT NULL AUTO_INCREMENT,
7   `name` varchar(30) NOT NULL,
8   `age` int(11) DEFAULT NULL,
9   PRIMARY KEY (`id`)
10 ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

连接Connect

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。

```
1 {
2     "host": "127.0.0.1",
3     "user": "wayne",
4     "password": "wayne",
5     "database": "test",
6     "port": 3306
7 }
```

```
1 import pymysql
2 import simplejson
3
4 conf = simplejson.load(open('conn.json'))
5 print(conf)
6
7 conn = None
8 try:
9     conn = pymysql.connect(**conf)
10    print(type(conn), conn)
11    conn.ping(False) # ping不同抛异常，True重连
12 finally:
13     if conn:
14         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。

Cursor类的实例，使用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。

```
1 import pymysql
2 import simplejson
3
4 with open('conn.json') as f:
5     conf = simplejson.load(f)
6
7 conn = None
8 try:
9     conn = pymysql.connect(**conf)
10    cursor = conn.cursor() # 获取一个游标
11
12    sql = "insert into student (name, age) values('tom', 20)"
13    rows = cursor.execute(sql)
14    print(rows)
15 finally:
16     if conn:
17         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

```
1 import pymysql
2 import simplejson
3
4 with open('conn.json') as f:
5     conf = simplejson.load(f)
6
7 conn = None
8 cursor = None
9 try:
10    conn = pymysql.connect(**conf)
11    cursor = conn.cursor() # 获取一个游标
12
13    for i in range(10):
14        sql = "insert into student (name, age) values('tom{}',
15        {}).format(i+1, 20+i)
16        rows = cursor.execute(sql)
17        conn.commit() # 事务提交
18 except:
19    conn.rollback() # 事务回滚
```

```

19 finally:
20     if cursor:
21         cursor.close()
22     if conn:
23         conn.close()

```

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()。

```

1 import pymysql
2 import simplejson
3
4 with open('conn.json') as f:
5     conf = simplejson.load(f)
6
7 conn = None
8 cursor = None
9 try:
10     conn = pymysql.connect(**conf)
11     cursor = conn.cursor() # 获取一个游标
12
13     sql = "select * from student"
14     rows = cursor.execute(sql)
15
16     print(cursor.fetchone())
17     print(cursor.fetchone())
18     print(cursor.rownumber, cursor.rowcount)
19     print('1 -----')
20     print(cursor.fetchmany(2))
21     print(cursor.rownumber, cursor.rowcount)
22     print('2 ~~~~~~')
23     print(cursor.fetchmany(2))
24     print(cursor.rownumber, cursor.rowcount)
25     print('3-----')
26     print(cursor.fetchall())
27     print(cursor.rownumber, cursor.rowcount)
28
29     for x in cursor.fetchall():
30         print(x, '~~~')
31     cursor.rownumber = 0 # 正负都支持
32     for x in cursor.fetchall():
33         print(x, '----')
34 finally:
35     if cursor:
36         cursor.close()
37     if conn:
38         conn.close()

```

名称	说明
fetchone()	获取结果集的下一行
fetchmany(size=None)	size指定返回的行数的行，None则返回空元组
fetchall()	返回剩余所有行，如果走到末尾，就返回空元组，否则返回一个元组，其元素是每一行的记录封装的一个元组
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。

```
1 from pymysql.cursors import DictCursor
2 cursor = conn.cursor(DictCursor)
3
4 # 返回结果
5 {'name': 'tom', 'age': 20, 'id': 4}
6 {'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

```
1 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

```
1 userid = 5 # 用户id可以变
2 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？

```
1 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且安全的!!!

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)

query查询字符串使用c printf风格。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。

```
1 import pymysql
2 import simplejson
3
4 with open('conn.json') as f:
5     conf = simplejson.load(f)
6
7 conn = None
8 cursor = None
9 try:
10     conn = pymysql.connect(**conf)
11     cursor = conn.cursor() # 获取一个游标
12
13     sql = "select * from student where id=%s"
14     userid = '2 or 1=1'
15     rows = cursor.execute(sql, userid) # (userid,)
16     print(cursor.fetchall())
17     print('-' * 30)
18     sql = "select * from student where name like %(name)s and age > %(age)s"
19     # 仅测试用，通常不要用like
20     cursor.execute(sql, {'name': 'tom%', 'age': 25})
21     print(cursor.fetchall())
22 finally:
23     if cursor:
24         cursor.close()
25     if conn:
26         conn.close()
```

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指令也不会被当做指令执行。

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。

开发时，应该使用参数化查询。主要目的不是为了语句缓存，而是为了有效消除注入攻击。

注意：这里说的是查询字符串的缓存，不是查询结果的缓存。

上下文支持

查看连接类和游标类的源码

```
1 # 连接类，老版本中
2 class Connection(object):
```

```

3     def __enter__(self):
4         """Context manager that returns a Cursor"""
5         return self.cursor()
6
7     def __exit__(self, exc, value, traceback):
8         """On successful exit, commit. On exception, rollback"""
9         if exc:
10            self.rollback()
11        else:
12            self.commit()
13
14    # 连接类，新版做了修改
15    class Connection(object):
16        def __enter__(self):
17            return self
18
19        def __exit__(self, *exc_info):
20            del exc_info
21            self.close()
22
23    # 游标类
24    class Cursor(object):
25        def __enter__(self):
26            return self
27
28        def __exit__(self, *exc_info):
29            del exc_info
30            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。

```

1  import pymysql
2  import simplejson
3
4  with open('conn.json') as f:
5      conf = simplejson.load(f)
6
7  conn = None
8  try:
9      conn = pymysql.connect(**conf)
10     with conn.cursor() as cursor: # 获取一个游标
11         sql = "select * from student where id=%s"
12         userid = 2
13         rows = cursor.execute(sql, userid)
14         print(cursor.fetchall())
15         print('-' * 30)
16         cursor.close() # 手动关闭
17 finally:
18     # 注意连接未关闭
19     if conn:
20         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

```
1 import pymysql
2 import simplejson
3
4 with open('conn.json') as f:
5     conf = simplejson.load(f)
6
7 try:
8     conn = pymysql.connect(**conf)
9     with conn:
10         with conn.cursor() as cursor: # 获取一个游标
11             sql = "select * from student where id=%s"
12             userid = 2
13             rows = cursor.execute(sql, userid)
14             print(cursor.fetchall())
15             print('-' * 30)
16         with conn.cursor() as cursor:
17             sql = "select * from student where id=6"
18             cursor.execute(sql)
19             print(cursor.fetchall())
20 except Exception as e:
21     print(e)
```

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。

mysqlclient

```
1 $ pip install mysqlclient
```

```
1 import MySQLdb
2
3 conn = MySQLdb.connect(host='127.0.0.1', user='wayne', password='wayne',
4                         port=3306,
5                         database='test'
6                         #autocommit=False # 缺省
7                         )
8 print(type(conn), conn)
9 cursor = conn.cursor()
10
11 with cursor:
12     x = cursor.execute('select * from employees')
13     print(x)
14     print(cursor.fetchall())
15
16 conn.close()
```


