

✉ (mailto:mail@dozer.cc)

🐦 (http://twitter.com/dozer47528)

🔗 (http://github.com/dozer47528)

Dozer Zone (/)

Python 中的 MySQL 数据库连接池

JULY 27TH, 2016

从 Java 到 Python

本文为我和同事的共同研究成果

当跨语言的时候，有些东西在一门语言中很常见，但到了另一门语言中可能会很少见。

例如 C# 中，经常会关注装箱装箱，但到了 Java 中却发现，根本没人关注这个。

后来才知道，原来是因为 Java 中没有真泛型，就算放到泛型集合中，一样会装箱。既然不可避免，那也就没人去关注这块的性能影响了。

而 C# 中要是写出这样的代码，那你明天不用来上班了。

同样的场景发生在了学习 Python 的过程中。

什么？数据库连接竟然没有连接池！？

完全不可理解啊，Java 中不用连接池对性能影响挺大的。

Python 程序员是因为 Python 本来就慢，然后就自暴自弃了吗？

突然想到一个笑话

问：为什么 Python 程序员很少谈论内存泄漏？

答：因为 Python 重启很快。

☹ 说多了都是泪，我之前排查 Java 内存泄漏的问题，超高并发的程序跑了1-2个月后就崩溃。我排查了好久，Java GC 参数也研究了很多，最后还是通过控制变量法找到了原因。

如果在 Python 中，多简单的事啊，写一个定时重启脚本，解决...

从 Java 到 Python

问题来源

解决问题第一步：网上找答案

解决问题第二步：分析异常日志

解决问题第三步：一层层看源码分析
神奇的数字

Root Cause

解决问题

问题来源

那本文的问题怎么来的呢？正是我给公司的代码加上连接池后产生的。一加连接池，就会有一定几率出现；一去掉连接池，就不会有。

```
db = get_connection()
try:
    cursor = db.cursor()
    if not cursor.execute("SELECT id FROM user WHERE name = %s", ['david']) > 0:
        return None
    return cursor.fetchone()[0]
finally:
    db.close()
```

一段很简单的代码，基本上整个项目中所有的数据库查询都是这么写的，本来没有任何问题。

但当我给底层加上连接池后，问题来了。

这边后报出这样一个异常：'NoneType' object has no attribute '__getitem__'

意思就是说 `cursor.fetchone()` 取出来的结果是 `None`。

但是，代码在调用之前命名已经检查过 `affected rows` 了，根据文档 `cursor.execute()` 返回的就是 `affected rows`。

文档也是这么写的：Returns long integer rows affected, if any。

解决问题第一步：网上找答案

什么测试驱动开发，敏捷开发，我觉得都不对，一句话形容我们那应该是：基于 Google 的 Bug 驱动开发。□

可惜网上无任何结果，去 stackoverflow (<http://stackoverflow.com/questions/38241440/MySQL-Pythondb-execute-result-doesnt-match-fetch-result>) 上问也没人知道。

感觉又来到了一片无人区.....

目前唯一能确认的就是和连接池相关了。

大致分析下应该是和连接复用有关，代码没写好？底层连接池并发处理的代码有 Bug？

先抓个详细的异常看看吧。

解决问题第二步：分析异常日志

我们项目用了 Sentry，一个异常跟踪系统。可以把报错时的调用堆栈和临时变量都记录下来。

第一个有用的信息是，我们竟然发现 `cursor.execute()` 的返回结果在 Sentry 上记录的是 18446744073709552000。

这是一个非常诡异的数字，因为它接近 $2^{64}-1$ (18446744073709551615)，而且还比它大了一点。

网上也找不到太多相关资料，和这个数字相关的都是 Javascript 相关的问题。

因为 Javascript 中是无法表示 $2^{64}-1$ 的，相关讨论：传送门

(<http://stackoverflow.com/questions/21568738/parseint18446744073709551616-returning-18446744073709552000-in-javascript/21568758>)

简单的一句话解释就是：这个数字超过了 Javascript Integer 的最大范围，所以底层用 Float 来表示了，所以导致丢失了精度。

但我们的程序没用 Javascript。到了这边，我们的第一反应一定是，要么 MySQL 出了 Bug。要么 MySQL-Python 出了 Bug。

解决问题第三步：一层层看源码分析

先看 MySQL-Python 源码，`cursor.execute()` 内部调用了 `affected_rows()` 方法得到了这个数字，而 `affected_rows()` 这个方法内部使用 C 实现了。

MySQL-Python 的 C 部分源码很简单，没什么逻辑：

```
return PyLong_FromUnsignedLongLong(mysql_affected_rows(&(self->connection))));
```

看样子也没什么特别的，这里就两个地方可能有问题，`PyLong_FromUnsignedLongLong()` 和 `mysql_affected_rows()`。

先自己尝试写了一段代码，调用 `PyLong_FromUnsignedLongLong()` 函数，发现无论如何都不会出现 18446744073709552000 这个数字。

然后看 MySQL 源码，`mysql_affected_rows()` 返回类型是 `my_ulonglong`，源码中其实是这么定义的：

```
typedef unsigned long long my_ulonglong;
```

也就是说，在 C 代码中，这个数字最大就是 $2^{64}-1$ (18446744073709551615)，不可能返回 18446744073709552000 的。

然后在 `mysql_affected_rows()` 的官方文档 (<http://dev.mysql.com/doc/refman/5.7/en/mysql-affected-rows.html>) 中又发现了一些有用的信息：

An integer greater than zero indicates the number of rows affected or retrieved. Zero indicates that no records were updated for an UPDATE statement, no rows matched the WHERE clause in the query or that no query has yet been executed. -1 indicates that the query returned an error or that, for a SELECT query, `mysql_affected_rows()` was called prior to calling `mysql_store_result()`.

Because `mysql_affected_rows()` returns an unsigned value, you can check for -1 by comparing the return value to `(my_ulonglong)-1` (or to `(my_ulonglong)~0`, which is equivalent).

好了，遇到第一个坑了，为什么 MySQL 官方文档说这里可能有 -1，而 MySQL-Python 的文档中却没说？而且返回类型是无符号的，-1 就变成 18446744073709551615 了。

那么如果我用 `if cursor.execute() > 0` 这种方式来判断命中行数时，明明出错了，我却会得到 True 的结果了。

很明显 MySQL-Python 写的是有问题的，同事联系了 MySQL-Python 的作者，作者承认了这里的问题，把代码修复了，下一个版本会修复。

神奇的数字

但是，看源码发现的东西还是没解决我们的问题，为什么我们的到的数字是 18446744073709552000，而不是 18446744073709551615？

整个调用链我们都检查过了，不可能出现这个数字。

然后一个周末，我在快睡醒的时候突然想到了一个问题，这个数字是不是在 Python 报错的时候，还是 18446744073709551615，而到了 Sentry 中，就变成了 18446744073709552000？

因为 Sentry Web 界面用的是 ajax，而 Javascript 中转换这个数字的时候就会出错。

最后一验证，果然是 Sentry 的问题，Javascript 真的处处是坑。

好了，到了这一步，等 MySQL-Python 作者修复完后，我们的代码也就不会报错了。问题解决？

但是，MySQL 官方却没有说为什么这里会出现 -1，而且为什么去掉了连接池就不会报错？

就算我们的代码不报错了，但如果这里的返回数字不符合我们预期或者说不可控的话，会导致更多隐形的数据上的问题。

Root Cause

目前为止，依然没找到 Root Cause。

别动，看好了，我要用压测大法了！既然这个问题是在高并发使用连接池时出现的，那就压测看看能不能重现吧。

用了同样的代码，10个进程，没有 sleep。没想到不需要一分钟，这个问题就会立刻重现。

而且每次重现时，都会有一些 MySQL 底层的警告，说出现了错误的调用顺序。

这时，我试了一下加了一行代码：

```
db = get_connection()
cursor = None
try:
    cursor = db.cursor()
    if not cursor.execute("SELECT id FROM user WHERE name = %s", ['david']) > 0:
        return None
    return cursor.fetchone()[0]
finally:
    if cursor: # new code
        cursor.close() # new code
    db.close()
```

加完后就再也没看到任何错误了。

嗯，这里我们的代码写的是不到位，我后来仔细看了官方教程，是有主动关闭 cursor 的代码的。（偷偷告诉你们，这里都是 CTO 以前写的 ☹）

粗略看了下 cursor.close() 的代码，里面其实就是在把未读完的数据读完：while self.nextset(): pass。

那这里出问题的原因也就好理解了，高并发情况下复用连接池，如果上一次请求由于某些原因没有读完所有数据，后面直接复用这个连接的时候，就会出现问题了。

然后，我又奇怪了，连接池框架在关闭连接的时候不应该做清理工作吗？

Java JDBC 源码也看过不少了，Connection 关闭的时候会清理 Statement，Statement 关闭的时候会清理 ResultSet。因为单个连接只会在单线程中操作，是线程安全的，所以实现这样的自动清理是非常简单的。

以前写 Java 中间件的时候，就总是把用户当 □，要尽量考虑各种情况避免内存泄漏。我们默认都是认为用户是从来不会去调用 close 方法的。所以常常会想方设法帮用户去自动处理。

解决问题

最后要来解决问题了，代码量很大，所有调用都改一遍其实也不难，因为这里都是有规律的，正则啊脚本啊什么的齐上阵，总是能解决的。

但是，其实也可以像 JDBC 那样搞自动关闭。

```
class AutoCloseCursorConnection(object):
    cursor = None
    conn = None

    def __init__(self, conn):
        self.conn = conn

    def __getattr__(self, key):
        return getattr(self.conn, key)


    def cursor(self, *args, **kwargs):
        self.cursor = self.conn.cursor(*args, **kwargs)
        return self.cursor

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

每次创建的连接包一下，就解决问题了。

本作品由 Dozer (<https://www.dozer.cc>) 创作，采用 知识共享署名-非商业性使用 4.0 国际许可协议 (<http://creativecommons.org/licenses/by-nc/4.0/>) 进行许可。

 编程技术 (86) (</categories.html#编程技术-ref>)

 MySQL (6) , (</tags.html#MySQL-ref>) 连接池 (1) , (</tags.html#连接池-ref>)
Python (2) (</tags.html#Python-ref>)
